The Pitfalls of Pseudo-Random Numbers in Machine Learning

Annabel Sophia Jakob

4th Year Project Report
Artificial Intelligence
School of Informatics
University of Edinburgh

2022

Abstract

Many machine learning methods rely on random numbers to initialise coefficients, generate data subsets, or shuffle the data. We investigated the effects and potential pitfalls of pseudo-random numbers on linear and logistic regression, random forests, simple neural networks and long short-term memory models. We set up experiments in which we compared the performance of models using pseudo-random numbers to those using true random numbers. The results indicated that the period of the random number generator has a far greater impact on logistic regression, random forest and long short-term memory models than their deterministic nature. Further, we found an increased standard deviation in the simple neural network models. We found no significant difference in performance between linear regression models using true random numbers and those using pseudo-random numbers. We conclude that the period of pseudo-random number generators is of greater concern than their lack of randomness in many machine learning applications.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Annabel Sophia Jakob*)

Acknowledgements

I would like to express my gratitude to my dissertation supervisor Dr. Arno Onken. It has been a pleasure to work under his guidance. He has been a great supervisor throughout the year, answering any of my many questions and providing advice when stuck.

I would also like to thank my partner Alex for telling me to take breaks when I forgot and making me laugh every day.

To my dear friends Wassim and Riccardo: thank you for being with me throughout my undergraduate degree. They are both great friends and supported me throughout the years.

Table of Contents

1 Introduction **1**

1.1 Context and Motivation 1

1.2 Objectives 2

1.3 Thesis Structure 3

2 Background and Literature Review **4**

2.1 Relevant Machine Learning Terminology 4

2.1.1 Linear Regression 4

2.1.2 Logistic Regression 4

2.1.3 Random Forest 5

2.1.4 Neural Networks 5

2.1.5 Long Short-Term Memory 6

2.1.6 Error Functions 6

2.1.7 Gradient Descent 6

2.2 Pseudo-Random Number Generators 6

2.3 Drawbacks of LCNGs 7

2.4 Related Research 9

3 Experiments **11**

3.1 Experimental Setup 11

3.2 Random Number Generators 11

3.3 Data Sets 12

3.4 Linear Regression Experiments 12

3.4.1 Setup 12

3.4.2 Test Suite 1 13

3.4.3 Test Suite 2 13

3.4.4 Test Suite 3 14

3.5 Logistic Regression 14

3.5.1 Setup 15

3.5.2 Test Suite 1 15

3.6 Random Forest Experiments 15

3.6.1 Setup 15

3.6.2 Test Suite 1 16

3.6.3 Test Suite 2 16

3.6.4 Test Suite 3 17

3.7 Neural Network Experiments . . . 17
3.7.1 Setup . . . 17
3.7.2 Test Suite 1 . . . 17
3.8 Time Series Experiments . . . 18
3.8.1 Setup . . . 18
3.8.2 Test Suite 1 . . . 18
3.9 Challenges . . . 19

4 Results **20**
4.1 Linear Regression Results . . . 20
4.1.1 Test suite 1 . . . 20
4.1.2 Test Suite 2 . . . 21
4.1.3 Test Suite 3 . . . 25
4.2 Logistic Regression Results . . . 26
4.2.1 Test Suite 1 . . . 26
4.3 Random Forest Results . . . 27
4.3.1 Test Suite 1 . . . 27
4.3.2 Test Suite 2 . . . 28
4.3.3 Test Suite 3 . . . 29
4.4 Neural Network Results . . . 30
4.4.1 Test Suite 1 . . . 30
4.5 Time Series Results . . . 31
4.5.1 Test Suite 1 . . . 31

5 Discussion **33**
5.1 Research Questions . . . 33
5.2 Future Work . . . 39

6 Conclusions **40**

Bibliography **41**

A Background - Additional Information **44**
A.1 Machine Learning Methods - Decision Trees . . . 44
A.2 Statistical Tests . . . 44

B Experiments - Additional Information **45**
B.1 Linear Regression . . . 45
B.2 Random Forest . . . 46
B.3 Neural Network . . . 48
B.4 Time Series . . . 49

C Code Implementation **52**
C.1 Linear Regression . . . 52
C.2 Logistic Regression . . . 53
C.3 Random Forest . . . 54

Chapter 1

Introduction

This research is concerned with the impact of pseudo-random number generators, specifically linear congruential number generators and their effect on various machine learning models.

1.1 Context and Motivation

Random numbers have many applications within machine learning (ML).
In *linear* and *logistic regression*, as well as *neural networks*, for instance, the goal is to find a function or combination of functions that best fits the data [10]. When such a machine learning model "learns", it attempts to find coefficients that make its underlying functions best fit the data. It is common practice to initialise the coefficients to small random numbers before training a model to reduce the possibility of model stagnation [8]. The most popular optimisation algorithm, gradient descent, iteratively finds coefficients that minimise the model error. Hence, it may produce different coefficients depending on the initial values. Further, it is common practice to shuffle the data [10] in a (pseudo-) random manner to prevent positional correlation as we assume that the data points are independently and identically distributed.

Random forests are an ensemble learning method used for classification and regression [10]. A random forest consists of several *decision trees* which, given an input, predict its class or value [10]. The final prediction is made as a majority vote across all decision trees. In Random forests, random numbers are used to randomly select features depending on the value on which to split the data [10]. Furthermore, random numbers are used to randomly select subsets of the training data, which are then used to train each decision tree in the forest.

Another use of random numbers is within biomedical artificial intelligence applications. A common task here is *time series classification*. Study participants are presented with a random sequence of stimuli [28, 33]. The stimulus response to each stimulus is measured and recorded [28]. This stimulus response depends on previously seen stimuli [28]. Thus, the task here is to classify the currently presented stimulus based on the current and previous stimulus responses [28]. To this end, a type of recurrent neural

network, called *Long Short-Term Memory* (LSTM) model, is suitable.

There exists an abundance of random number generators, such as the *Mersenne Twister* [25]. We generally distinguish *true* and *pseudo*-random number generators. Unlike true random number generators, pseudo-random number generators appear random but are completely deterministic (that is, not random).

Within this research, we focus on a type of pseudo-random number generator called *Linear Congruential* Number Generators (LCNG) first proposed by Lehmer [22]. These generators have the well-known trait of being cryptographically insecure[1] [4]. Moreover, they have been shown to generate points that fall in a finite number of parallel hyperplanes [24] and have a possibly small period[2] [25]. We will investigate LCNGs in the context of machine learning while focusing on these non-random traits.

While pseudo-random number generators and their disadvantages have been well-researched [24, 4], few attempts have been made to investigate them specifically in the context of machine learning. Initial attempts include the analysis of pseudo- and quantum random numbers in the context of neural networks, decision trees, and random forests [2], and investigations into the effects of the randomness of pseudo-random numbers in embedded machine learning systems [23].

Despite being an essential part of machine learning, the effect of pseudo-random numbers within machine learning has received little attention. It is thus vital to investigate these effects and point out potential pitfalls.

1.2 Objectives

We aim to investigate the effects of the choice of random number generators (RNG) on various machine learning models, with a focus on LCNGs. Due to limited previous research in the field, this thesis serves as a first step in discerning the specific effects of pseudo-random numbers.

The main goals of this research are exploratory and can be thus summarised as follows:

- Investigate the importance of randomness and period for initialising coefficients in linear and logistic regression problems.
- Determine disadvantages when using LCNGs to pick samples for Random Forests.
- Determine the effects of predictable sequences when training neural networks.
- Investigate the ability of LSTM models to learn from predictable sequences.

[1] Here, 'cryptographically insecure' means that, given an output sequence of the generator, another segment of the output can be predicted within feasible time and space complexity bounds [4].

[2] The period indicates the length of the sequence until the generator repeats itself.

1.3 Thesis Structure

The chapters of this dissertation are structured as follows:

Chapter 2 introduces relevant machine learning models and techniques, including methods used for optimization and error measurement. The chapter then explores pseudo-random number generators and their various traits in some depth. This is followed by a review of literature in this field of study.

Chapter 3 explains the general setup of the experiments conducted. Within the experiments, we use random numbers to initialise coefficients, sort the training data and pick random subsets of the training data. We further revisit any challenges faced in the project and suggest mitigation strategies.

Chapter 4 contains the results obtained from the experiments. We find that, in most models, the period of LCNGs has a far greater effect than their lack of true randomness.

Chapter 5 discusses and evaluates the results of the experiments. In this chapter, we draw conclusions about the effects of the different properties of LCNGs. Finally, we briefly discuss future research endeavours building upon the presented findings.

Chapter 6 briefly summarizes the main conclusions and findings of this research.

Chapter 2

Background and Literature Review

Section 2.1 first discusses the machine learning models and techniques pertaining to this work. Next, Section 2.2 presents pseudo-random number generators (PRNGs) and lists popular choices of generators. In addition, we introduce the linear congruential generator used in this project. We then elaborate on the drawbacks of linear congruential number generators and define the qualities of *good* pseudo-random number generators. Finally, Section 2.4 describes similar research conducted on the effects of randomness in machine learning tasks.

2.1 Relevant Machine Learning Terminology

2.1.1 Linear Regression

One of the simplest models for regression is the *linear regression* model [3]. Generally, a *regression* model is a model that produces predictions in the continuous space of the natural numbers rather than of a fixed set of classes. Given a D-dimensional input vector $\boldsymbol{x} = (x_1, ..., x_D)^T$ with $x_i \in \mathbb{R}$ where $i \in [0, D]$, we predict the label as

$$y(\boldsymbol{x}, \boldsymbol{w}) = w_0 + w_1 x_1 + ... + w_D x_D, y(\boldsymbol{x}, \boldsymbol{w}) \in \mathbb{R} \tag{2.1}$$

$\boldsymbol{w} = (w_0, w_1, ..., w_D)$ with $w_i \in \mathbb{R}$ where $i \in [0, D]$ is called the *weight vector*. The goal of the linear regression problem is to find $\boldsymbol{w}$ that best approximates the data [3]. We use a two-dimensional version of the linear regression problem in this project. That is, the weight vector is defined as $\boldsymbol{w} = (w_0, w_1)$. Throughout this report we will refer to the weights of the two-dimensional linear regression problem as *coefficients* and denote w_0 as b and w_1 as m.

2.1.2 Logistic Regression

Further, we will consider logistic regression models. Here, the term *regression* is somewhat misleading as logistic regression is often used for two-class classification [3] (note, however, that it is not limited to this domain). Denote these classes C_1 and

C_2. The logistic regression model computes the posterior probabilities of a particular sample belonging to class C_1 or C_2. In other words, we compute the probability that a sample belongs to class C_1 or C_2 given its feature values. The posterior probability of a D-dimensional sample vector $\boldsymbol{x} = (x_1, ..., x_D)^\intercal$ with $x_i \in \mathbb{R}$ where $i \in [0, D]$ belonging to class C_1 or class C_2, denoted $p(C_1|\boldsymbol{x})$ and $p(C_2|\boldsymbol{x})$, is defined by Equations 2.2 and 2.3[3].

$$p(C_1|\mathbf{x}) = \frac{1}{1 + \exp - (\mathbf{w}^\intercal \mathbf{x})} \tag{2.2}$$

$$p(C_2|\mathbf{x}) = 1 - p(C_1|\mathbf{x}) \tag{2.3}$$

Where the D-dimensional weight vector $\boldsymbol{w}$ is defined as $\boldsymbol{w} = (w_0, w_1, ..., w_D)^\intercal$ with $w_i \in \mathbb{R}$ where $i \in [0, D]$. Similarly to linear regression, we want to find a weight vector $\boldsymbol{w}$ that best approximates the data.

2.1.3 Random Forest

To understand random forest models, we must first explain decision tree classifiers. Decision trees can be used for classification or regression [10]; however, for our purposes, we will focus solely on their capabilities as classifiers. A Decision tree classifier builds a tree representation of the data [10]. Each node in the tree represents a feature of the data. At each node, the data is split on the value of that feature [10]. Appendix A.1 contains the visualisation of a simple decision tree example.

A Random forest consists of any number of decision trees [10]. Each decision tree is trained on a random, ideally different, subset of the training data, where a particular sample may occur more than once per tree [10]. Predictions are made based on a majority vote [10].

2.1.4 Neural Networks

The mathematical intricacies of neural networks are beyond the scope of this report. Hence we will restrict ourselves to an intuitive explanation. Neural networks were originally modelled after the human brain and consist of several ”node layers” [17]. Within a neural network, a node is a mathematical unit that transforms and passes on input provided to it [17]. Machine learning libraries, such as Tensorflow [1] allow users to create networks as several different types of layers. Here, we will use the following types of layers, as defined by Géron [10]:

- **Dense**: Every node of this layer is connected to every node in the preceding layer.
- **Convolutional**: Each node in the layer is focused on a small number of nodes in the previous layer.
- **Pooling**: These layers shrink the input given to them. The most popular variation of this layer is the `MaxPooling` layer.
- **Flatten**: This layer flattens the input.

[1] https://www.tensorflow.org

In the context of neural networks, *activation functions* determine how the weighted sum of inputs to a node or nodes in a layer is transformed into an output [7]. We are interested in using neural networks to solve classification problems for this research.

2.1.5 Long Short-Term Memory

Long Short-Term Memory (LSTM) models are a type of Recurrent Neural Network (RNN) [10]. Unlike the neural networks discussed thus far, a recurrent neural network can process sequences of data [10]. The networks described in Section 2.1.4 function of inputs of fixed size and are unable to consider temporal relations. RNNs allow us to analyse *time series* data such as stock prices [10]. LSTM models, specifically, can learn which information to store in their ”long-term memory”, which information to discard and which information to use [10]

2.1.6 Error Functions

Error functions are used to measure the performance of machine learning models. We will use two kinds of error functions. Given two n-dimensional vectors with $n \in \mathbb{Z}^+$ of true and predicted labels y and $\hat{y}$, we define according to Bishop [3]:

- **Mean Square Error (MSE)**: $MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$
- **Log Likelihood**: $L(\beta) = \frac{n}{2}\ln\beta - \frac{n}{2}\ln 2\pi - \beta(\frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2)$

β indicates the precision of a zero mean Gaussian random variable.

2.1.7 Gradient Descent

Gradient descent is an iterative optimisation approach that gradually changes the model parameters to minimise a cost function [10]. The number of epochs indicates the number of iterations the algorithm completes [10]. The learning rate indicates how drastically the parameters change in each iteration [10]. Gradient Descent uses error functions as described in Section 2.1.6 to measure the fit of the current weights and to determine how to alter them [10].

2.2 Pseudo-Random Number Generators

True random number generators exist and are offered, for instance, by Random.org [2], which generates random numbers based on atmospheric noise [12]. Computers are inherently deterministic. Thus, to generate true random numbers, we must introduce true randomness through physical phenomena, for instance [11]. This makes true random numbers both inefficient and hard to attain [11].

For this reason, considerable effort has been put into creating number generators that at least appear to output a random sequence of numbers. In reality, however, most of these generators are completely deterministic [11]. For this reason, these types of

[2] https://www.random.org

generators are known as *pseudo*-random number generators and there exist a variety of different implementations. The *period* of a pseudo-random number generator indicates the length of the sequence that it produces before the generated numbers start to repeat themselves.

Python 3 [3], for instance, uses the Mersenne Twister [25, 30], which is known to be robust and reliable due to its large period, efficiency, and the fact that it passes several statistical tests for randomness [25]. The Python library NumPy [4] uses the PCG-64 [27] random number generator (RNG), which is a permutation congruential number generator that outperforms the Mersenne Twister with regards to prediction difficulty and space usage [27].

The type of pseudo-random number generator on which we focus throughout this project is a *linear congruential* number generator. That is, generators of the form

$$r_{i+1} = k * r_i + c \bmod m$$

where k, c and m are coefficients and r_0 is given as a *seed*. This type of PRNG is one of the simplest generators, and their advantages and drawbacks have been researched thoroughly [4, 24, 25, 11], which renders it suitable for this project. These generators were regarded as reliable and served as standard pseudo-random number generators for a long time [29] but have been criticised heavily in recent times [24, 4].
Note that, within this report, we use a type of LCNG in which $c = 0$, often referred to as *Lehmer generator* [22]. We thus omit the coefficient c from further discussion and refer to these generators as LCNGs. We choose this limitation as reports investigating the drawbacks of LCNGs, which this thesis is building upon, make similar assumptions [4, 24]. Also, note that, for simplicity, an LCNG with coefficients k, modulus m and initial seed r_0 will be referred to as LCNGm, k, r_0.

2.3 Drawbacks of LCNGs

While LCNGs are simple, they exhibit several striking deficits. For instance, depending on the coefficients chosen, the period of the generator may be very small [29].

When considered geometrically, the sequence produced by a LCNG does not appear random in the least. Firstly, denote the ordered sequence of n numbers produced by an LCNG as n-tuples. Consider the sequence $[1, 2, 3]$. The 2-tuples of this sequence are $[(1, 2), (2, 3)]$. For any choice of modulus m and multiplier k, view the n-tuples $(r_1, r_2, ..., r_n), (r_2, r_3, ... r_{n+1}), ...$ produced by an LCNG as points in the unit n-cube. Then all points will fall into a finite number of parallel hyperplanes [24]. This is visualized in Figure 2.1a and 2.1b, which plots the 3- and 2-tuples produced by the LCNG with $m = 267$, $k = 7$, and an initial seed $r_0 = 3$. Note that this result can be extended to all congruential number generators of the form $r_{i+1} = k * r_i \pmod m$ [24]. Furthermore, given that m and the coefficient a are unknown, linear congruential number generators are not cryptographically secure [4]. That is, given an output sequence of the

[3] https://www.python.org/doc/
[4] https://numpy.org/

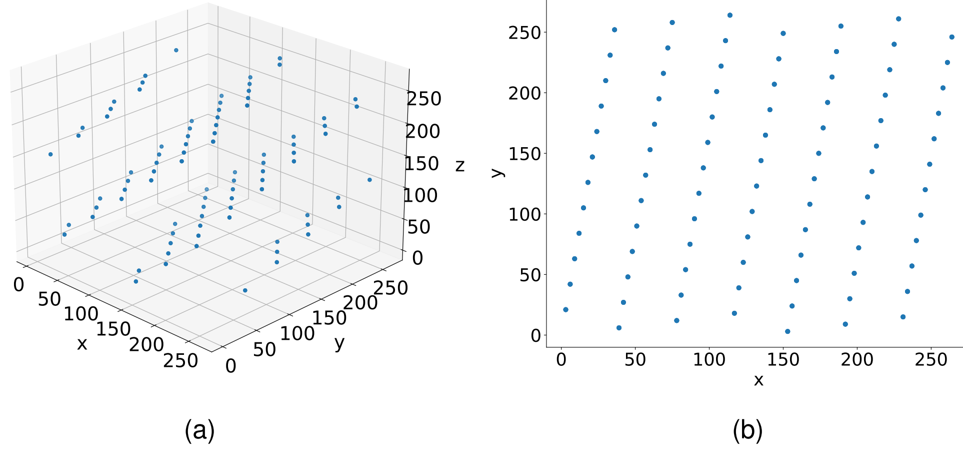


Figure 2.1: The 3-tuples (a) and 2-tuples (b) produced by a LCNG267,3,9.

generator, another segment of the output can be predicted within feasible time and space complexity bounds [4]. The proof of this property is based on the *extrapolation property* inherent to all linear congruential number generators. The extrapolation property states that if two generators with the same modulus m produce the same first r_0+n values, then they will continue to produce the same sequence [4].

While these flaws are inherent to all LCNGs, there still exist ones better than others. This depends on the chosen modulus and coefficients. While it is generally accepted that the modulus of an LCNG should be a (large) prime number [29], we require a heuristic in order to evaluate the suitability of the multiplier k. In [29], Park and Miller suggest three tests, which any multiplier k must pass in order to be considered. The tests are the following:

- T_1: $f(r_i) = k * r_i \bmod m$ must be a full period generating function.
- T_2: The full period sequence $\ldots, r_1, r_2, \ldots, r_{m-1}, r_1, \ldots$ must be sufficiently random.
- T_3: Can $f(.)$ be implemented sufficiently with 64-bit arithmetic?

The authors suggested that the algorithm should be implemented efficiently in 32-bit arithmetic [29]. However, this advice is outdated, as most computers use 64-bit arithmetic nowadays, and we altered the test definition of T_3 accordingly.

Park and Miller point out some notably bad pseudo-random number generators in their work. Firstly, any random number generator with too small of a period should be considered insufficient [29]. An example of such a generator is LCNG267,7,3 which has a period of just 87.

To facilitate efficient implementations, generators of the form $r_{i+1} = k * r_i \bmod 2^b$, where b is the integer word size of the computer, have been used in the past [29]. One ”truly horrible” [18] example is the IBM RANDU generator, defined by

$$r_{i+1} = 65539 * r_i \bmod 2^{31} \tag{2.4}$$

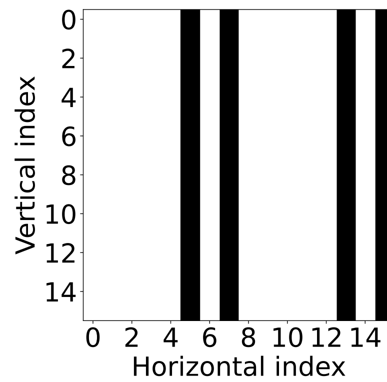


Figure 2.2: The lower eight bits of 100,000 numbers generated using the RANDU random number generator. The 16x16 graph visualizes all possible eight bit numbers. A black pixel indicates that the bit number occurred in the sequence of generated numbers.

This generator has neither a full period nor is the produced sequence sufficiently random [18].

This can be demonstrated by creating a 16 by 16 pixel image of all possible 8-bit numbers. Using RANDU, we generate a large sequence of random numbers (here, we generated 100,000 numbers). We discard all but the lowest 8 bits of each number and plot them in the corresponding cell in a 16 by 16 pixel image. Black pixels indicate that the 8-bit combination has been generated, and white indicates the opposite. Figure 2.2 shows the result. Out of 256 possible combinations, RANDU only generated 64 combinations. In other words, the lowest 8 bits of the generated numbers cycle between the same 64 combinations. Due to its infamy and demonstrated lack of randomness, RANDU is a valuable generator for this project.

2.4 Related Research

While pseudo-random number generators have been researched thoroughly [4, 18, 24, 25, 29], few attempts have been made at investigating their effects on machine learning.

In 2018, for instance, Lockhart et al. [23] have analysed the randomness and effects of the degree of randomness of pseudo-random numbers in embedded machine learning systems with regard to accuracy, resource utilisation and the speed of the learning process. As additional resource constraints are imposed on embedded systems, stochastic bit-stream computing is used. In stochastic computing, first introduced by John von Neumann [31], data is represented as probabilities. To this end, a random number generator is used to generate a stochastic bit-stream where the probability of any bit being 1 is p [23, 1]. By measuring the frequency of 1s and using logic gates, the results of arithmetic operations can be estimated.

To determine the randomness of an RNG, Lockhart et al. [23] use L'Ecuyer's C library TestU01 [21]. After the degree of randomness has been established, the random numbers are used on a four-layer deep neural network classifying digits using the MNIST handwritten digit database [20]. The random number generators tested by

Lockart et al. include the linear feedback shift register (LFSR), Galois and Fibonacci, SBoNG [26], Halton Sequences [13], Cellular Automata [16], and a PCG-variant. Note that many of these generators, namely LFSR, Galois and Fibonacci, and SBoNG require specific hardware implementations [16], and thus are not of interest for this project. While Halton Sequences and PCG are possible generators to be used for machine learning, they are outwith the scope of this work, as we are interested in analysing the effects of *bad randomness* on machine learning algorithms, as opposed to their performance. Note that, instead of using random numbers to initialise the weights and the bias of the deep neural network, the PRNGs are used to create random bit-streams used in stochastic computing. Nevertheless, Lockhart et al. found that, while the RNGs were not random according to L'Ecuyer's test library [21], this did not have a significant effect on the machine learning model [23].

Note that Lockhart et al. test the RNGs in a highly specialised area. Thus, these results may not translate to general machine learning applications. Further, Lockhart et al. only consider PRNGs that have been established to be ”good” [23]. However, the authors fail to test ”bad” pseudo-random number generators. Thus, these results may not be representative of a wide range of PRNGs.

Similarly, Bird et al. [2] analysed the effects of pseudo- and quantum random numbers on the performance of dense and convolutional neural networks, (quantum) random trees and random forests. Unlike in classical computing, in quantum computing true randomness is possible. This allows Bird et al. to compare the performance of PRNGs to the ”ideal” scenario, that is, to the performance of a machine learning model using true random numbers. For the generation of pseudo-random numbers, an AMD FX8320 processor and the Java Virtual Machine were used [2]. There is no mention of the degree of randomness, the exact method of deriving random numbers, or the PRNG period. In a task that aims to investigate the effect of PRNGs, this is detrimental. This information is vital when comparing the performance of models using true and pseudo-random numbers, as different pseudo-random number generators may impact models differently, as we find within our research.

The authors reported mixed results. For instance, a convolutional neural network trained on the CIFAR-10 dataset of coloured images [19] performed much better when initialised using quantum random number generators (QRNG)[2]. While a deep neural network trained on the MNIST data set [20] and initialised using QRNG had almost exclusively a higher initial accuracy than a model initialised using a PRNG, the accuracy after training did not differ significantly. No significant difference was found in classification tasks using random trees and quantum random trees, as well as random forests and quantum random forests [2]. Further, Bird et al. [2] do not investigate the causes of any performance differences. As we find here, it is not necessarily a lack of true randomness that impacts models but aspects such as the periodicity of a PRNG. As such, few conclusions can be drawn from this work.

While initial attempts to investigate the effects of pseudo-random numbers on machine learning have been made, there are few definitive results.

Chapter 3

Experiments

This chapter details the experiments conducted to discern the effects of linear congruential number generators on machine learning. We first introduce the general experiment setup. We then detail the experimental details of the linear regression, logistic regression, random forest, simple neural network, and LSTM experiments.

3.1 Experimental Setup

The project broadly consists of three test suites. Test suite 1 consists of experiments to gauge different machine learning models' general behavior when applying pseudo-random number generators. Test suite 2 aims to exploit specific behaviours of pseudo-random number generators, such as their small period. Finally, test suite 3 serves to find the causes of any behaviour witnessed in test suite 2, if necessary.

Most machine learning libraries, such as PyTorch [1] or TensorFlow [2] allow only a limited manipulation of their provided methods. For this project, it was necessary to have control of aspects such as the exact value to which parameters are initialized. As most ML libraries (e.g. PyTorch [3] or TensorFlow [4]) only allow limited control, we provided our own implementation where necessary. All models are implemented in Python 3. This language was chosen as many ML libraries provide methods in Python. Further, it is the language that the author is most familiar with.

3.2 Random Number Generators

This project makes use of three different random number generators. Firstly, we use LCNGs with the coefficients set according to the requirements of the experiment. Secondly, RANDU, although an LCNG as well, serves as notoriously bad example of a pseudo-random number generator (Section 2.3). Lastly, we compare the performance

[1] https://pytorch.org/
[2] https://www.tensorflow.org/
[3] https://pytorch.org/
[4] https://www.tensorflow.org/

of these generators to those of a true random number generator. For these purposes, we use the true random numbers provided by Random.org [5]. Each random number generator produces numbers within a different range. For instance, the largest number an LCNG with modulus m can generate is $m-1$. In the case of RANDU ($m = 2^{31}$) this is $2^{31}-1$. The true random number generator used for the purposes of this project can generate numbers as large as 1e9 [12]. Thus, we often use only the lower n bits of the numbers produced or compute the modulus. This allows us to produce numbers within the same range allowing for easier comparison.

3.3 Data Sets

We provide a brief overview of the datasets used for the following experiments. Note that we do not list data sets created specifically for the experiments, as these are described in the relevant experiment sections.

Some linear regression models are trained on the Boston Housing data set [14], which contains information about housing in the Boston Mass area, gathered by the U.S Census Service [14]. The data set consists of 506 entries in total. We use the median value of owner-occupied homes and the average number of rooms per dwelling to limit the problem to two dimensions. This allows for easier analysis and visualisation. This data set exhibits linear traits and is thus suitable for linear regression.

The logistic regression and random forest experiments use the Titanic data set [9]. This data set consists of 891 training and 418 test samples containing Titanic passenger information such as the age, sex, passenger class and ID, name and if the passenger survived [9]. As the test data does not contain the ”Survived”-column, which we use as label, the training data was used for training and testing.

Finally, we use the MNIST data set [20] for neural network experiments. This data set contains 60,000 training and 10,000 test samples. Each sample is the size-normalized and centered image of a handwritten digit [20].

3.4 Linear Regression Experiments

This section details the experiments set up to investigate the effects of using LCNGs to initialise the coefficients of linear regression models.

3.4.1 Setup

Linear regression is a popular introductory topic to machine learning. In [10], for instance, it is the first model introduced. Linear regression is a well-understood technique and thus lends itself as a toy problem. While there exists an analytical solution [3], we use a gradient descent approach to find the optimal coefficients. This allows us to investigate the effects of different coefficients on the learning and performance of a simple model. There exist plenty of linear regression implementations (for instance,

[5] https://www.random.org/

provided by Scikit Learn [6]); however, most of these implementations use the analytical solution. Similarly, there exists an abundance of gradient descent implementations (for instance, provided by Scikit Learn [7]). However, these implementations do not allow initialising the coefficients to specific values. Therefore, we provide a Python implementation based on code written by Brownlee [5]. The specific class used for our experiments can be found in Appendix C.1.

3.4.2 Test Suite 1

The first test suite uses the Boston Housing data set [14] (Section 3.3) to predict the median value of owner-occupied homes given the number of rooms. 80% of the data is used for training, while 20% is retained for testing purposes. We create 50 models per RNG, where each model trains for 100 epochs at a learning rate of 0.01. As the analytical solution does not change given the same dataset, this was only computed once.

We compare the performance of models initialised using LCNG267,3,31 and RANDU to models initialised using true random numbers and the analytical solution. The choice of coefficients for the LCNG is arbitrary and reflects a random choice one might make to initialise a model. We use only the lower eight bits produced by RANDU and the true random number generator, such that they produce numbers within a similar range as LCNG267,3,31. The models are compared in terms of their MSE. This experiment gauges the general effects of pseudo-random numbers in linear regression models.

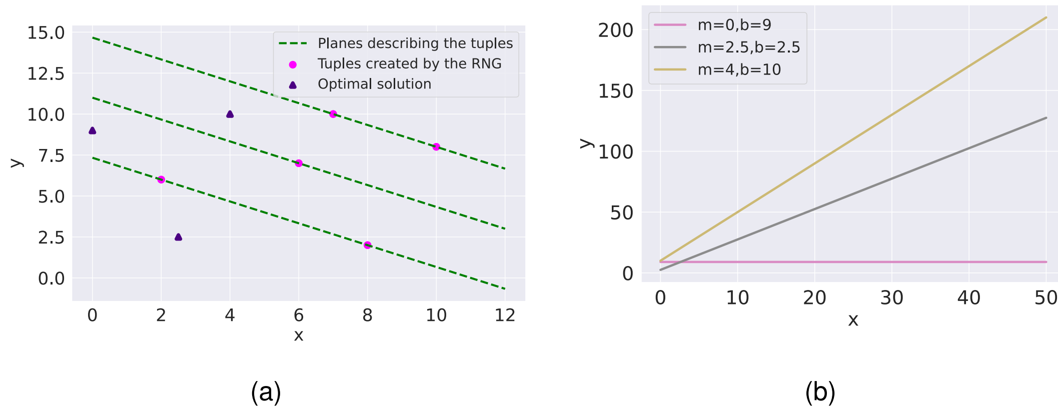
3.4.3 Test Suite 2

This test suite aims to exploit the property that the n-tuples created by an LCNG fall into a finite number of hyperplanes [24]. In test suite 2, we set up a linear regression problem such that the tuple containing the coefficients of the optimal solution lies between these hyperplanes. This means that the LCNG will not be able to generate the optimal solution as initial coefficients.

We use LCNG11,3,2 and choose three points that fall between the planes created by the 2-tuples of this LCNG. That is, we conduct three sets of experiments for this test suite. Each point serves as an optimal solution to a linear regression problem. This is visualised in Figure 3.1a. The tuples chosen are $(0,9),(2.5,2.5),(4,10)$, where the first element indicates coefficient m, and the second element indicates coefficient b (Section 2.1). Figure 3.1b depicts the lines created by these coefficients.

We generate 500 uniformly spaced points for each set of coefficients in the interval $[0,50]$, denoted X. In addition, we generate a set Y, where a particular element depends on an element X_i in X and is denoted Y_i. We have $Y_i = mX_i + b$. X serves as samples, while Y contains the labels. The model uses the data in an 80-20 split of training and test data. For each problem, we evaluate the performance of 50 models initialised using true random numbers, LCNG11,3,2 and RANDU. Note that only the four least

[6] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html
[7] https://scikit-learn.org/stable/modules/sgd.html



(a) (b)

Figure 3.1: (a) The figure represents the planes created by the numbers generated by LCNG11,3,2 and the optimal coefficients of the linear regression problems.
(b) The lines created by the coefficients of the optimal solutions of the linear regression problems.

significant bits of the numbers generated by RANDU and the true random number generators are used. Thus, the numbers produced by each RNG fall approximately within the same range. Each model runs for 100 epochs with a learning rate of 0.01. The experiment in which $(0,9)$ are the optimal coefficients is denoted experiment 1; the following experiments are denoted experiment 2 and experiment 3.

3.4.4 Test Suite 3

Test suite 3 investigates all possible coefficients generated by LCNG11,3,2 and their performance on each regression problem as described in test suite 2 (Section 3.4.3). After this, we compare the generated initial coefficients to all possible combinations of two coefficients in the range $[0,10]$. This experiment allows us to uncover patterns in the coefficients generated by LCNG11,3,2 and put them into context, given all possible combinations of coefficients. Recall that we set up three linear regression problems in which the optimal coefficients are $(0,9),(2.5,2.5),(4,10)$, where the first element in the tuple denotes m and the second element denotes b.

We create a dataset consisting of 500 samples X and labels Y within the range $[0,50]$ as described in test suite 2 (Section 3.4.3). Again, the data is divided in an 80-20 split of training and test data. We then compare the performance of the models initialised using the LCNG coefficients to all possible solutions using the MSE.

3.5 Logistic Regression

We conducted similar experiments to those conducted on the linear regression models for logistic regression wherein we initialise the weights to a value chosen by a random number generator and use gradient descent to approach the optimal solution.

3.5.1 Setup

We implement a Python class containing methods to train a logistic regression model using gradient descent and to predict the class of a sample in a two-class classification problem. The implementation used is based on code provided by Brownlee [8] [6]

As cost function, we use the log loss as detailed in Section 2.1.6.

3.5.2 Test Suite 1

We compare the performance of models initialised using true random numbers, RANDU, and LCNG267,3,7 to each other and the baseline Scikit Learn implementation [9].

We conduct a total of 300 experiments, 100 experiments per RNG. For half of the experiments, we only used the eight least significant bits of numbers produced. Each model, including the baseline model, is trained for 100 epochs at a learning rate of 0.01. Note that the baseline implementation does not allow to specify a learning rate. This experiment is conducted on the Titanic data set [9] with the goal to predict if a passenger survived, given their age, passenger ID, passenger class, number of siblings or spouses aboard, number of parents/children aboard, fare paid, and ticket number. As we use seven features, we decided to use this number as number of coefficients to predict. The models are compared based on their training and test accuracy.

This experiment allows us to gauge the general effects of pseudo-random numbers of logistic regression problems.

3.6 Random Forest Experiments

Within these experiments, we aim to discern the effects of different random number generators when selecting the samples used to train each decision tree in the random forest.

3.6.1 Setup

We use random numbers to select samples used to train each decision tree for these experiments. As most machine learning libraries do not allow for this (see, for instance, Scikit Learn [10]) we provide a Python 3 implementation. To this end, we create a `RandomForest`-class that creates any number of decision trees, as specified by the user. When fitting the model, the user must provide an array of indexes, indicating the indexes of the samples used for a particular tree. The `RandomForest`-class uses the `DecisionTreeClassifier`-class provided by Scikit Learn [11]. It is to note that the `RandomForest` presented here is a binary classifier. It outputs either zero (indicating that a sample does not belong to a class A) or one (indicating that a sample belongs

[8] https://machinelearningmastery.com/logistic-regression-for-machine-learning/
[9] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
[10] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
[11] https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

to class A). Further, the `RandomForest`-class provides a `predict`-method to classify given samples. To make a final decision, we sum the output of all trees. If the sum is larger than $k/2$, where k is the positive integer indicating the number of decision trees in the forest, then the forest returns true. It returns false otherwise. The random forest makes predictions based on a *majority* vote. Hence, if more than 50% of the trees return `True` (i.e. a one), the total sum will be larger than $k/2$. The source code is contained in Appendix C.3.

Throughout all test suites, we use the Titanic Data set [9] and try to predict if a given passenger survived. The data set can be set up as a binary prediction problem. Therefore, it lends itself to training a random forest classifier. We only use data regarding the passenger's ID, passenger class and fare paid. Thus, information about the passenger's sex, name, ticket and cabin number was discarded. We chose this limitation to limit the model complexity and allow for easier analysis. The models use 791 samples for training and retain 100 samples for testing.

3.6.2 Test Suite 1

The first test suite aims to gauge the general effects of different random number generators on Random Forest models. In addition to using a true random number generator and RANDU, we use LCNG637,7,3. We choose these parameters as the training data set contains 791 samples. Hence the largest number produced by the LCNG should be 790 (due to zero-indexing). We choose $m < 791$ to reflect a general random choice one may make for this purpose. We produce integers within the range $[0, 791)$ for the experiments using true random numbers. RANDU may produce numbers larger than 790. Therefore, each generated integer is computed modulus 791. We conduct 20 experiments per random number generator. Each tree trains on 500 samples, where a particular sample may occur more than once. The models are compared based on their average training and test accuracy, as well as the accuracy of the Scikit Learn implementation [12].

3.6.3 Test Suite 2

In this test suite, we are interested in the small periodicity of some LCNGs and how this affects the performance of random forest models compared to true random numbers and RANDU.

Some LCNGs have a short period. That is, given a modulus m, they generate only a fraction of the numbers in the range $[1, m)$ (see Section 2.3). random forests rely on the training data to be split randomly, such that each tree trains on a different data subset [10]. Depending on the coefficients of the LCNG, the random forest model may not make use of all the samples in the training set, disregarding potentially important information.

In an extremer case, every tree in the random forest may train on the same training samples and hence the random forest may be an ensemble of the same tree. For example,

[12] https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

consider an LCNG with period p. If each decision tree uses $s * p$ samples, where s is a positive integer, then each tree in the forest will use the same subset of samples.

We conduct an experiment using fourteen different LCNGs, all of which have a period that divides the number of samples per tree. As data set, we again use the Titanic data set [9]. We maintain the general test setup from test suite 1 3.6.2. Each random forest contains 100 trees, each using 500 samples (where one sample may repeat in a single tree). We then compare the training and testing accuracy of these Random Forests to those of the first fourteen models using true random number generators in the first test suite.

3.6.4 Test Suite 3

The goal of the final test suite is to narrow down the reasons for the performance difference between the LCNG and true random number generator witnessed in test suite 2.

To this end, we impose restrictions, similar to those of an LCNG, on the true random number generator. We conduct three sets of experiments. We imposed an upper limit on the true random number generator for the first set of experiments. In the second set of experiments, we imposed a ”period”. For instance, given period n, we generate an array of n true random numbers and duplicate this sequence to generate sufficient indexes for all decision trees in the random forest. Finally, we impose both the upper limit and the period on the true random number generator simultaneously. Per set of experiments, we created ten random forest models imitating the conditions of the first ten LCNGs used in the second test suite (Section 3.6.3). We do not conduct tests for all LCNGs from the second test suite due to the resource constraints of obtaining true random numbers (see Section 3.9).

3.7 Neural Network Experiments

When creating neural networks, it is common to shuffle the training dataset prior to training the model [10]. Within this test suite, we investigate if shuffling the data predictably affects the learning and performance of a neural network.

3.7.1 Setup

We run 20 experiments using the sorted and shuffled MNIST data [20], respectively. First, we sort the data by digit. That is, the model is shown all images of 1’s, then all images of 2’s, and so on. To implement the network, we use the Keras [13] library.

3.7.2 Test Suite 1

The network used here consists of a convolutional input layer, followed by a max-pooling layer. The output of the pooling layer is flattened and forwarded to a dense

[13] https://keras.io/

layer, followed by a dense output layer. The layers use RELU-activation functions, except for the final output layer, which uses softmax. A visualisation of the neural network can be found in Appendix B.3. We compare the networks in terms of their average test accuracy. Further, we conduct two-tailed tests to determine whether the results are statistically significant.

3.8 Time Series Experiments

Pseudorandom number generators are inherently deterministic [24] and hence predictable. In this experiment, we investigate the effects of predictable sequences on time series models.

3.8.1 Setup

In this experiment, we set up a toy problem in which we try to classify a stimulus based on the current and previously seen stimulus responses. This particular experiment is inspired by a similar experiment presented by Kai et al. [32]. To this end, we assume the existence of five stimuli, denoted 1, 2, 3, 4 and 5. We chose this number arbitrarily to introduce a certain complexity into the problem while still making it easy to visualise. The value of the stimulus response is given by Equation 3.1, where x_i and y_i denote the stimulus (1,2,3,4 or 5) and stimulus response at time i, respectively.

$$y_i = \begin{cases} x_i + y_{i-1} \text{ if } x_i > x_{i-1} \\ x_i - y_{i-1} \text{ otherwise} \end{cases} \tag{3.1}$$

This definition of stimulus responses means that the response at time i depends on that at time $i-1$

The model uses the Keras [14] library. It consists of an LSTM-layer using a RELU activation function followed by a Dense layer using a softmax activation function.

3.8.2 Test Suite 1

We generate ten sequences of 5000 stimuli and their responses using a true random number generator, RANDU and LCNG131,5,1, respectively. The parameters of the LCNG are random and simulate an arbitrary choice as one may be inclined to make to select stimuli. Lastly, we will generate a sequence using LCNG4,3,1. We choose this LCNG as a baseline in which no manipulation of the numbers is required to fit them to a certain range. Each integer generated by LCNG131,5,1 and RANDU will be computed modulus 5 to receive integers in the range $[0,4]$. In addition, we will conduct the same experiment while only using the lower two bits of the generated integers. The true random number generator supplies values within the required range, so no augmentation is required.

[14]https://keras.io/

In summary, we use six different RNGs to generate ten sequences of 5000 stimuli each. Each sequence will be used for a single experiment. Thus, in total, we conduct 60 experiments.

We investigate the model's ability to predict the next stimulus given the nine previous stimuli. Denote this task ”Stimulus Prediction”. In addition, we train LSTMs to classify stimuli given the previous nine stimulus responses and current stimulus response. Denote this task ”Stimulus Classification”. Finally, we evaluate the model performance using the mean square error of the actual and predicted labels. We will compare a model's ability to predict the next stimulus to its ability to classify a stimulus given several stimulus responses.

3.9 Challenges

We will briefly discuss any challenges faced during this project.

Firstly, we experienced limitations in procuring true random numbers from the Random.org API. For the purposes of this research, we used the Developer API which allows for 1,000 requests or 250,000 bits per day per API key [15]. A user can create up to ten API keys. The random forest experiments, however, required more than 50,000 random numbers per decision tree. This means that generating a sufficient amount of numbers required several days and multiple API keys. To mitigate these limitations, the generated numbers were saved to `.txt` files to be used in other experiments.

An additional challenge was the unavailability of suitable code for models such as linear and logistic regression and random forests. We provided our own implementations for these purposes, however, the creation and thorough testing of the code infrastructure was non-trivial.

[15] https://api.random.org/pricing

Chapter 4

Results

This chapter describes the results obtained from the various experiments, as detailed in Chapter 3.

4.1 Linear Regression Results

4.1.1 Test suite 1

In the first test suite, we initialise the coefficients of 50 models using LCNG267,3,31, and the lower eight bits of a true random number generator and RANDU. We use the Boston Housing data set [14] and train each model to predict the median value of owner-occupied houses based on the number of rooms.

Figure 4.1 depicts the average training and test error per RNG in terms of median value of houses. The training and test error of all models using RNGs are significantly larger than the analytical solution. This may be because each model trains for 100 epochs, not until the error converges or an optimal solution is attained. Further, Figure 4.1 shows that, overall, RANDU performs the worst on both the training and test set, followed by LCNG267,3,31 and the true random number generator. Note that LCNG267,3,31 performs slightly better on the test set than the models using true random numbers.

To determine if there are significant performance differences between the models using true random numbers and those using pseudo-random numbers, we conduct two-tailed tests (see Appendix A.2) on the training and test set errors using a significance threshold of 0.05. We compare the errors of the models using true random number generators and those using LCNG267,3,31, as these are the best-performing models besides the analytical solution. Figure 4.1 summarizes the results. The p-value is below 0.05 for neither the training set nor the test set. Thus, we conclude that the performance difference between the two model types is not statistically significant.

Further, we analyse the distribution of errors for each initialisation method (true random numbers, RANDU, and LCNG267,3,31). The distributions for the training and test set are displayed in Figure 4.2.

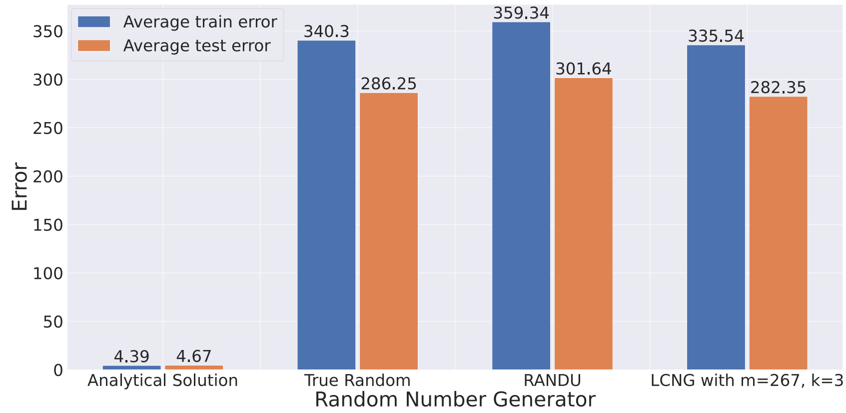


Figure 4.1: The average training and test errors for the linear regression problem over 50 experiments per random number generator (LCNG267,3,31, and the lowest eight bits of a true random number generator and RANDU) and the analytical solution.

Data Subset	Statistic	p-Value
Train set	-0.2072	0.8363
Test set	-0.3138	0.7543

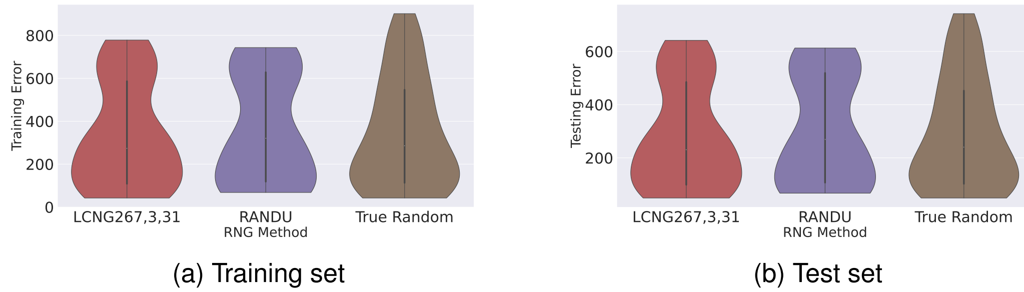
Table 4.1: The statistic and p-value of two-tailed tests conducted on the training and test errors of the models using true random numbers and LCNG267,3,31

The distributions of the training and test errors are similar for all initialisation methods. The models using true random number generators display a wider range of errors, reaching values as high as approximately 850 on the training set and over 600 on the test set. The models initialised using RANDU and LCNG267,3,31 show a lower maximum error on the training and test set. The LCNG and RANDU-models reach a maximum error of approximately 800 on the training set and 700 on the test set. While performing worst overall on the training and test set, the RANDU models have the smallest maximum error. Note, however, that the RANDU models also produce the largest minimum error for both the training and test set, as can be seen in Figure 4.2. The distribution of training and testing errors of the LCNG267,3,31 model and that of the RANDU model are similar. The error distribution of each RNG contains two values around which most errors are centred. This is approximately 200 and 650 for the training set and approximately 150 and 550 for the test set, respectively. The true random number models only have one such area. In these models, the error centre is approximately 150 for both the training and test set.

There is no statistically significant performance difference between the RNGs. However, the errors of the models using true random numbers are distributed more widely.

4.1.2 Test Suite 2

In this test suite, we explore the special case of the optimal coefficients of the linear regression model falling between the planes created by the tuples of an LCNG. Specifically, we investigate LCNG11,3,2 and choose three two-tuples for which to conduct the



(a) Training set (b) Test set

Figure 4.2: The distribution of training and test errors of linear regression models, across 50 models. The initial coefficients are chosen using true random numbers, LCNG267,3,31, RANDU, and true random numbers, respectively. This figure depicts the error distribution for the models using each type of RNG.

experiment. The tuples chosen are $(0,9), (2.5,2.5), (4,10)$ where the first element in the tuple indicates coefficient m and the second element indicates coefficient b.

We first consider the results of experiment 1. Figure 4.3 depicts the average training and testing errors.

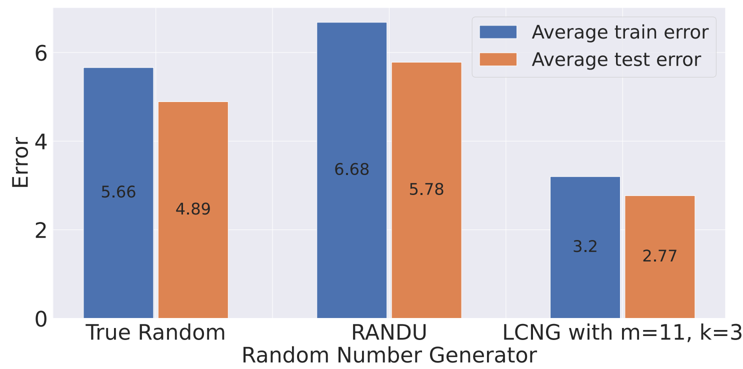
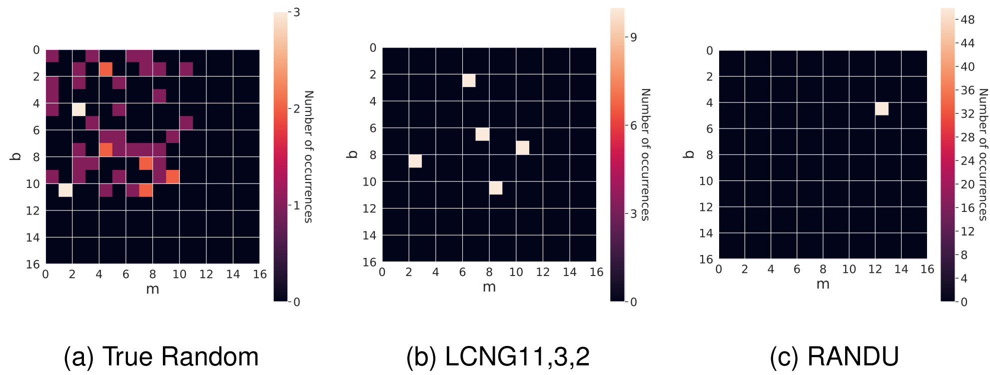


Figure 4.3: Experiment 1: Average training and testing error of the models initialised using different RNGS over 50 experiments with optimal coefficients $w_1 = 0, w_0 = 9$.

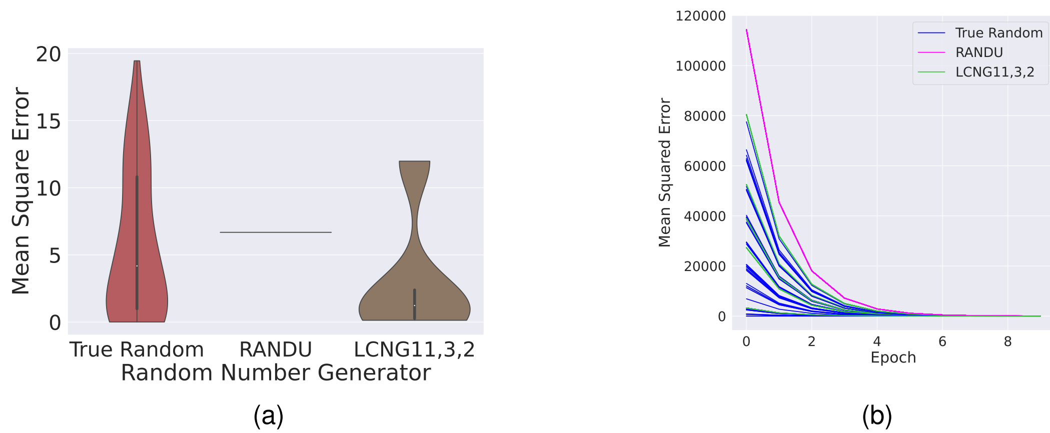
Figure 4.3 shows that LCNG11,3,2 achieves the lowest training and testing error at 3.20 and 2.77, respectively. This is followed by the true random number models with a training error of 6.30 and a test error of 5.45. A potential cause is that the largest number LCNG11,3,2 can produce is ten, while the true random number generator and RANDU can produce numbers up to sixteen. This is disproven when considering the initial coefficients generated by each RNG, as seen in Figure 4.4. While the true random number generator produces a wider variety of numbers, they lie within the same range as the coefficients produced by LCNG11,3,2. Further, it is noteworthy that RANDU only generates a single number. The difference between the optimal coefficients, $(0,9)$, and the initial coefficients produced by RANDU $(12,4)$ is large, producing a large final error.

We further consider the distribution of errors per RNG method. Similar to test suite 1 (Section 3.4.2), the range of errors of the models initialised using true random number



(a) True Random (b) LCNG11,3,2 (c) RANDU

Figure 4.4: The two-tuples of coefficients generated by different random number generators.



(a) (b)

Figure 4.5: Experiment 1: (a) depicts the distribution of errors per RNG. (b) shows the learning curves in the first ten epochs of all models initialised with true random numbers, RANDU, and LCNG11,3,2.

generators ranges further than that of RANDU and LCNG11,3,2 (Figure 4.5a). The wider range of initial coefficients may explain this spread in errors, as can be seen in Figure 4.4. Figure 4.5b depicts the error development within the first ten training epochs for each model and RNG method. Figure 4.5b shows that RANDU commences with the overall highest error, while the models initialised using LCNG11,3,2 and true random numbers perform similarly. We see, however, that the errors for all models and methods converge starting from epoch four.

We then conduct the same experiment with optimal coefficients $m = 2.5$ and $b = 2.5$. Figure 4.6 shows that in experiment 2, the models using true random numbers achieved the lowest average training and testing error, closely followed by the models using LCNG11,3,2. Note the increase in the average error from the training set to the test set for RANDU. A potential cause may be that the test samples are allocated inconveniently for RANDU.

Figures 4.7a and 4.7b show a similar behavior as in experiment 1.

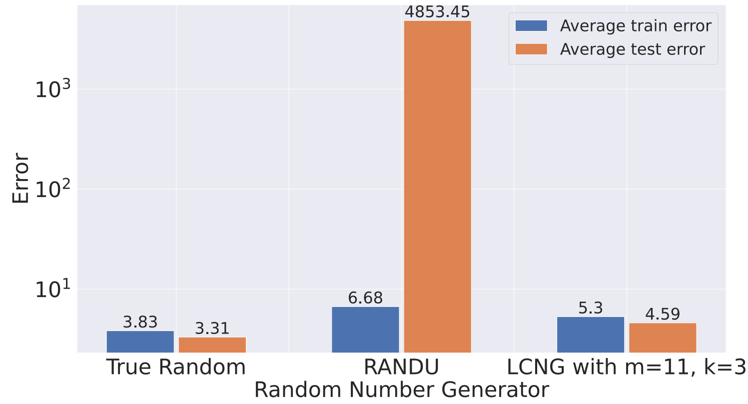


Figure 4.6: Experiment 2: Average training and testing error of the models initialised using different RNGS over 50 experiments with optimal coefficients $m = 2.5, b = 2.5$.

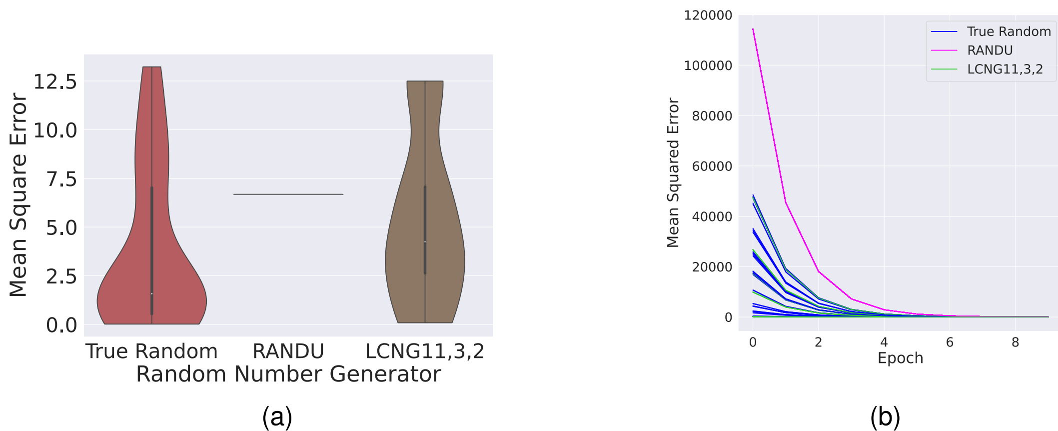


Figure 4.7: Experiment 2: (a) depicts the distribution of errors per RNG. (b) shows the learning curves in the first ten epochs of all models initialised with true random numbers, RANDU, and LCNG11,3,2.

Finally, we consider experiment 3, in which the optimal coefficients are $m = 4$ and $b = 10$. We find that the models using true random number generators perform worse than the models using RANDU or LCNG11,3,2 (Figure 4.8). This is somewhat surprising, given the performance of the RANDU models in experiment 1 and experiment 2. Figure 4.9a shows that the errors of the true random number models are further distributed than in experiment 1 (Figure 4.5a) and experiment 2 (Figure 4.7a) and the distribution does not show large areas of concentration as in previous experiments. However, the distribution of errors of the LCNG11,3,2-model shows that most errors are centred between the values 0 and 2.5, explaining the LCNG11,3,2-model's good performance.

Figure 4.9a shows similar initial errors for the model using true random numbers and the model using LCNG11,3,2. Again, RANDU starts with the largest error. The errors, however, converge as seen in experiment 1 (Figure 4.5b) and experiment 2 (Figure 4.7b).

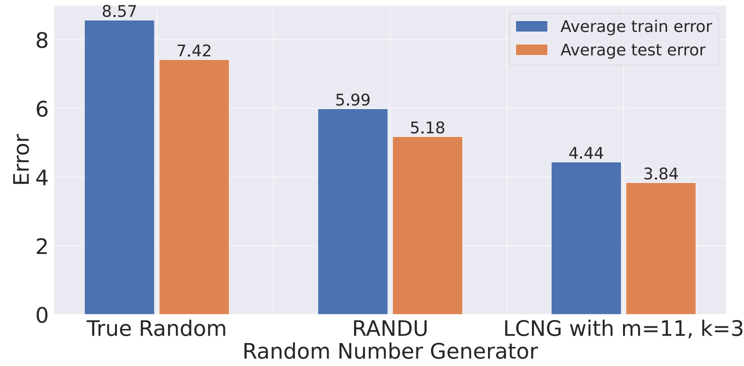


Figure 4.8: Experiment 3: Average training and testing error of the models initialised using different RNGS over 50 experiments with optimal coefficients $m = 4, b = 10$.

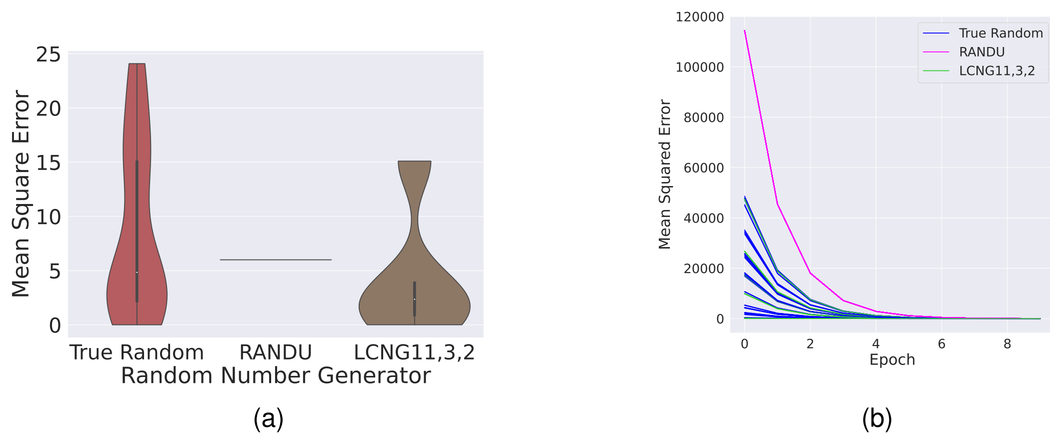
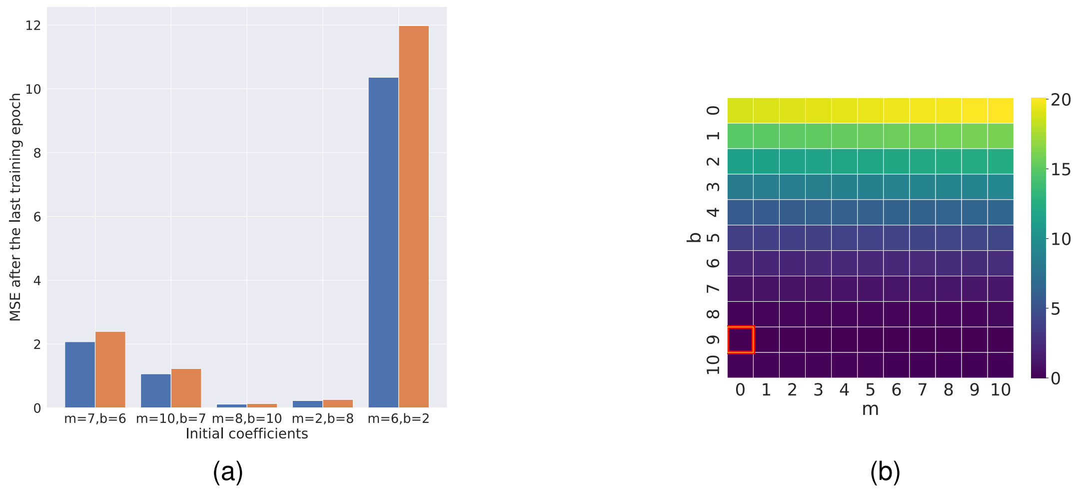


Figure 4.9: Experiment 3: (a) depicts the distribution of errors per RNG. (b) shows the learning curves in the first ten epochs of all models initialised with true random numbers, RANDU, and LCNG11,3,2.

None of the random number generators performs decidedly better than the others. It is to note that RANDU tends to have the largest error values in the first and second experiments (for both the training and the test set). At the same time, the true random number generator performs worse in the third experiment on average. On average, the LCNG performs best. In the violin plots, we can see that the performance of the true random number generator is skewed slightly by extreme error values and that the majority of errors centre around the same values as for the LCNG. We also see that RANDU seems to achieve a single error value exclusively for all tests. Additionally, the distribution of errors has a similar shape across all experiments per RNG.

4.1.3 Test Suite 3

Test suite 2 fails to uncover significant differences between the performance of the models initialised using different random number generators. Thus, we now turn to investigate the performance of all possible two-tuples generated by LCNG11,3,2.



(a) (b)

Figure 4.10: Experiment 1: (a) Error comparison between the models initialised using the different indices generated by the LCNG. (b)The final error of linear regression models per initialisation coefficients in experiment 1. The red square indicates the optimal coefficients. The orange square indicates the final coefficients found by the LCNG-model that result in the least error at training time.

Recall that, for experiment 1, the optimal coefficients are $m = 0, b = 9$. Figure 4.10a shows that, across all models initialised using LCNG11,3,2, the model initialised with the coefficients $m = 8, b = 10$ has the lowest training and testing error. This is followed by the model initialised with the coefficients $m = 2, b = 8$. The closer the initial b is to the optimal value of b, the lower the error. The initial value of m does not gravely impact the final error. This can also be seen in Figure 4.10b. Figure 4.10b shows a more drastic change in error as b changes, while m only causes slight changes.

A similar trend can be detected in experiment 2 and experiment 3 (see Appendix B.1).

4.2 Logistic Regression Results

4.2.1 Test Suite 1

Similarly to the experiments conducted concerning linear regression (Section 3.4), we conduct experiments initialising the coefficients of logistic regression models using LCNG267,3,7, RANDU and true random numbers. We first conduct the experiments without altering the range of numbers produced. Subsequently, we repeat the experiments using the eight least significant bits of the random number generators.

Figure 4.11a depicts the average training and testing errors of the Scikit Learn implementation, true random numbers, RANDU, and LCNG267,3,7. The Scikit Learn implementation performs best on both the training and test set. The models using true random numbers achieve the second-best average accuracy on the training and test set. The models using LCNGs and RANDU perform similarly. We conduct two-tailed statistical tests at a threshold of significance of 0.05 to discern if the gap in performance between the models using true random numbers and those using RANDU is statistically

significant. The results are depicted in Table 4.2. Note that we did not conduct tests comparing the models using true random numbers and those using LCNG267,3,7, as the performance of LCNG267,3,7 and RANDU are similar. We find that the training and test accuracy of the models initialised using true random numbers is statistically significantly better.

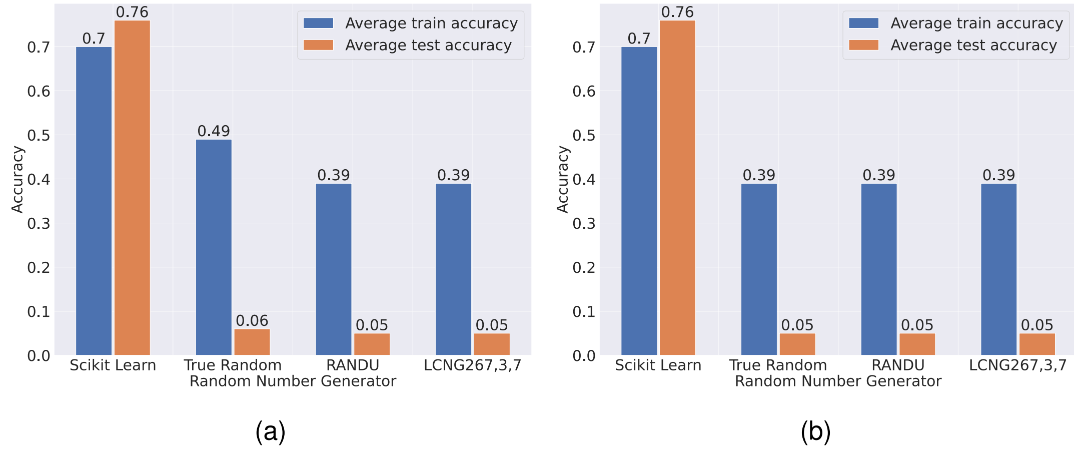


Figure 4.11: (a) Average training and testing accuracy when initialising logistic regression models using different random number generators, across 50 experiments per generator. (b) Average training and testing accuracy of logistic regression models using coefficients initialised using LCNG267,3,2, and the eight least significant bits of RANDU and a true random number generator, across 50 experiments per generator.

Experiment	Statistic	*p*-Value
Training Accuracy	5.3770	5.1419e-07
Testing Accuracy	4.8990	38.0657e-07

Table 4.2: The results of statistical tests conducted on the training and testing accuracy of logistic regression models using true random numbers to initialise coefficients and those using RANDU.

We repeat the experiment using the eight least significant bits of the numbers produced. The average errors, depicted in Figure 4.11b, now show no difference in performance between any of the RNGs, except for the Scikit Learn implementation.

4.3 Random Forest Results

4.3.1 Test Suite 1

In the first test suite, we use a true random number generator, LCNG637,7,3, and RANDU to randomly select samples for each decision tree in a random forest model. We use the Titanic data set [9] and train models to determine if an individual died based on factors such as the passenger ID, passenger class, and fare paid.

Figure 4.12 shows the resulting average accuracy per implementation per RNG, including the Scikit Learn implementation. The training and testing accuracy of the

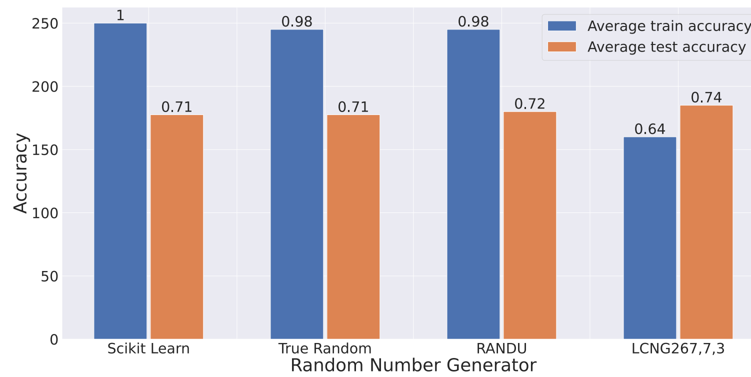


Figure 4.12: The training and testing accuracy of a random forest model when using different random number generators to select training samples for each tree.

Scikit Learn implementation, the true random number generator and RANDU do not differ significantly. While LCNG637,7,3 performs significantly worse on the training set, it achieves the highest accuracy on the testing set, surpassing the Scikit Learn implementation.

4.3.2 Test Suite 2

This test suite exploits the small period of LCNGs. Specifically, we choose LCNGs such that their period divides the number of samples per tree, such that each tree trains on the same subset of the training data. Finally, we compare the performance of these models to those picking samples using RANDU and true random numbers.

Table 4.3 depicts the training and test accuracy of fourteen Random Forest models with samples selected using true random numbers. We see that the training accuracy of all models is high: all models have a training accuracy of at least 0.98. Similarly, the test accuracy is at least 0.69. The average training accuracy is 0.98, while the average testing accuracy is 0.71 for the true random number models.

Model	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Training Acc.	0.98	0.99	0.99	0.98	0.99	0.98	0.98	0.98	0.98	0.99	0.99	0.98	0.98	0.98
Test Acc.	0.69	0.73	0.69	0.67	0.73	0.65	0.70	0.73	0.73	0.69	0.74	0.78	0.70	0.72

Table 4.3: The training and testing accuracy of a random forest model when using true random numbers to select samples.

Figure 4.4 shows the training and test accuracy per random forest model when using an LCNG to select samples, as well as the average training and testing accuracy. We see that the accuracy tends to be higher the larger the period. For instance, the LCNGs with a period of 250 generally have a training accuracy of over 0.70. On the other hand, the LCNGs with a period of 50 generally have an accuracy of at most 0.60. There is, however, no obvious trend in the testing accuracy.

To determine if the training and test performance of the model using true random numbers is statistically significantly better than that using LCNGs, we conduct two-tailed tests. We test the null hypothesis that the performance of the LCNG model does

m	k	Seed	Period	Training Accuracy	Testing Accuracy
758	681	569	125	0.67	0.65
721	622	616	50	0.60	0.66
503	12	11	250	0.73	0.58
508	15	13	125	0.65	0.59
381	374	368	125	0.70	0.69
254	228	220	125	0.67	0.68
127	56	54	125	0.63	0.52
103	16	9	50	0.60	0.60
206	119	86	50	0.57	0.54
412	367	258	50	0.57	0.54
412	367	258	50	0.53	0.44
503	7	6	250	0.72	0.57
503	14	13	250	0.73	0.59
503	50	31	250	0.71	0.70
503	64	19	250	0.71	0.70
Average				0.65	0.61

Table 4.4: The accuracy of LCNGs on the Titanic random forest problem. Each LCNG is initialized with different coefficients m, k and $seed$ such that their period divides the number of training samples.

not differ from that of the true random model, at the 5% level. Table 4.5 summarizes the results. We reject the null hypothesis for the training set: the p-value is only 0.0044, and thus the difference in performance on the training set is statistically significant. The p-value of the test on the test set is larger than the significance threshold, and hence we do not reject the null hypothesis for the test set.

	Statistic	P-value
Training accuracy	-3.2569	0.0044
Testing accuracy	-1.1294	0.2736

Table 4.5: Results of two-tailed tests on the training and testing performance of the true random and LCNG Random Forest models.

4.3.3 Test Suite 3

Test suite 2 showed differences in the accuracy of random forest models, depending on the random number generator used to pick samples for each tree. Test suite 3 further investigates these findings by imposing restrictions on the true random number generator.

Figure 4.6 shows the training and testing accuracy for each model. When only imposing an upper limit on the true random number generator, the training accuracy of the model is generally very high, reaching a maximum accuracy of 97% when imposing an upper limit of 758 or 721. On the other hand, the minimum accuracy is reached when imposing an upper limit of 127, with 59%. Generally, when imposing a higher upper limit, the training accuracy, too, increases. However, there does not seem to be such a correlation regarding the test accuracy. For instance, the models with an upper limit of 758 and 721 have a training accuracy of 68%. However, the model with an upper limit of 503 has a

testing accuracy of 75%. The average training and testing accuracy of these models is lower than those without any restrictions imposed (Table 4.3).

When restricting the number of distinct indices produced by the true random number generator, the training accuracy decreases significantly compared to the training accuracy of the model on which we imposed an upper limit. Within this set of experiments, the model for which we imposed a period of 250 achieves the highest accuracy with 0.71%. Conversely, we achieve the lowest training accuracy at 61% with a model which imposes a period of 50. For the test set, we achieve a maximum accuracy of 80% with a period of 125. The lowest accuracy, 35%, is achieved with a model with a period of 50. The performance decreases more significantly when imposing a period (from an average training and testing accuracy of 98% and 71% to 65% and 69%, respectively) than when imposing an upper limit (average training and testing accuracy of 73% and 65%, respectively). Further, while the training and test accuracy of the models on which we imposed an upper limit fluctuates dramatically, we see no such extreme fluctuations within the results of these models.

Finally, we restrict both the period and upper limit simultaneously. Here, we achieve a maximum training accuracy of 71% with a model with an upper limit of 503 and a period of 250. This model also achieves the maximum testing accuracy of 70%. The lowest training and testing accuracy, 45% and 34%, respectively, are achieved with an upper limit of 127 and a period of 125.

Overall, the models using a true random number generator with only an upper limit imposed perform best on both the training and the test set. These models achieve the highest training and testing accuracy for most settings (upper limits and periods). The models on which we impose a period perform similarly to the LCNG. The models operating under an upper limit and period perform the worst overall. A visualisation of the true random number indexes is in Appendix B.2.

4.4 Neural Network Results

4.4.1 Test Suite 1

We conduct experiments shuffling the data used to train neural networks in different manners. We first sort the data by digit and compare the performance of the resulting models with those trained on randomly shuffled data.

Table 4.7 shows the average test accuracy and standard deviation of the models. The mean testing accuracy is similar for both types of models. Conducting two-tailed statistical tests at a threshold of significance of 0.05 indicates that the results are not statistically significant (Table 4.8). That is, we must assume that both types of models perform equally well in terms of their test accuracy. The standard deviation of the models using sorted data is slightly higher than that of the model using shuffled data. Again, we use two-tailed statistical tests at a significance threshold of 0.05 to discern if the difference is statistically significant. At a p-value of 0.0076 (Table 4.8), the tests indicate that the standard deviation of the models using sorted data is statistically significantly higher.

		TR Upper Limit		TR Period		TR Upper Limit + Period	
Upper Limit	Period	Train Acc.	Test Acc.	Train Acc.	Test Acc.	Train Acc.	Test Acc.
758	125	**0.97**	*0.68*	**0.63**	0.62	0.64	*0.31*
721	50	**0.97**	*0.68*	0.61	*0.36*	0.66	0.65
503	250	**0.88**	*0.75*	**0.71**	0.65	**0.71**	0.70
508	125	**0.89**	*0.68*	0.65	*0.68*	**0.62**	*0.58*
381	125	**0.83**	*0.72*	**0.65**	*0.62*	0.67	0.65
254	125	**0.79**	0.62	**0.64**	*0.61*	**0.64**	0.64
127	125	0.59	*0.31*	**0.68**	*0.80*	**0.45**	0.34
103	50	**0.66**	0.63	0.63	*0.35*	0.65	*0.65*
206	50	**0.67**	*0.68*	0.62	0.63	**0.54**	*0.52*
412	50	**0.86**	*0.73*	0.66	0.72	**0.54**	*0.50*
Average		0.73	0.65	0.65	0.60	0.61	0.55

Table 4.6: Training and testing accuracy of random forest models when imposing different restrictions on the true random number generator to select samples for each Decision Tree in the forest, compared to an LCNG with similar restrictions. Bold marks the best training accuracy, bold and underline marks the worst training accuracy. Similarly, italic marks the best testing accuracy while italic and underline marks the worst testing accuracy.

Experiment	Mean Testing Accuracy	Standard Deviation
Unsorted	98.6818	0.1034
Sorted	98.6843	0.1313

Table 4.7: The mean testing accuracy and standard deviation across 20 neural network models when presenting the data in a sorted and unsorted manner.

4.5 Time Series Results

4.5.1 Test Suite 1

Within these experiments, we aim to investigate if predictable sequences of training data negatively affect the performance of LSTMs. We aim to provide insight into whether deterministic sequences cause the LSTM to learn from the sequence rather than the underlying patterns in the data.

To this end, we investigate the model's ability to predict the next stimulus given the nine previous stimuli. In addition, we train LSTMs to classify stimuli given the previous nine stimulus responses and current stimulus response. The performance is evaluated using the MSE (Section 2.1.6). To generate sequences of stimuli, we use the two least significant bits of RANDU and LCNG131,5,1. We further generate sequences using RANDU and LCNG131,5,1 modulus 5, LCNG4,3,1 and true random numbers in the range $[0,1]$. Table 4.9 shows the average MSE of the models using each type of RNG for the task of stimulus classification and stimulus prediction. The models using LCNG4,3,1-generated sequences achieve the lowest average MSE in the task of stimulus prediction at $2.58e^{-12}$. This is followed by the models trained on sequences generated using the two least significant bits of numbers produced by RANDU. Here, the models using true random sequences achieve the highest MSE with $1.7966e^{0}$.

Metric	Statistic	p-Value
Accuracy	-0.1485	0.8828
Standard Deviation	-2.8233	0.0076

Table 4.8: The results of two-tailed statistical tests comparison the accuracy and standard deviation across all models using sorted and unsorted data.

Experiment	MSE Stimulus Prediction	MSE Stimulus Classification
LCNG Mod	$0.0013e^{0}$	$0.0003e^{0}$
LCNG Bits	$0.0029e^{0}$	$0.0002e^{0}$
LCNG4,3,1	$2.5829e^{-12}$	$1.8924e^{-05}$
RANDU Mod	$1.7681e^{0}$	$0.3477e^{0}$
RANDU Bits	$1.5636e^{-07}$	$4.9011e^{-06}$
TR	$1.7966e^{0}$	$0.3870e^{0}$

Table 4.9: The mean square error of the LSTM classifying a stimulus based on the a sequence of previously seen stimulus responses (MSE Accuracy Stimulus Classification) and predicting the next stimulus based on previously presented stimuli (MSE Stimulus Prediction). **Bold** indicates the largest MSE and *italic* indicates the least MSE.

In the task of stimulus classification, however, the RANDU-Bits models perform the best at an MSE of $4.90e^{-06}$, followed by LCNG4,3,1. Again, the models trained on true random sequences achieve the highest error with an average MSE of $0.3870e^{0}$.

The models using sequences produced by LCNG131,5,1 achieve consistently low errors. Here, the MSE on the stimulus prediction task is higher than that of the stimulus classification task. The models trained on sequences generated by LCNG4,3,1 show the opposite behaviour: Here, the MSE on the stimulus prediction task is significantly lower than that on the stimulus classification task.

Chapter 5

Discussion

I this chapter, we consider the results obtained from the experiments conducted in Chapter 3. Based on these results, we evaluate how specific traits of LCNGs affect the respective machine learning models. We will then briefly discuss possible future work building up on these findings.

5.1 Research Questions

1. How are linear regression models affected by the randomness and period of LCNGs?
We investigated the difference in performance when initialising the coefficients of a linear regression model using true random numbers and numbers derived from RANDU and an LCNG. To approach the optimal coefficients, we use gradient descent. We find no significant difference in performance. However, different RNGs significantly impact the error distribution across the models. To this end, we conduct three test suites.

In test suite 1 of the linear regression experiments (Section 3.4.2), we compare the performance of linear regression models initialised using different random number generators on the Boston Housing Dataset [14]. We find no statistically significant difference in the performance of the models.

Test suite 2 (Section 3.4.3) examines the effects of different random number generators when the optimal coefficients lie between the planes created by LCNG11,3,2. Again, there is no statistically significant difference between the performance of the models. However, we find that the errors produced by the models initialised using true random numbers span a wider distribution than those using the LCNG or RANDU.

Finally, in test suite 3, we investigate the final training error of models initialised with all possible coefficients generated by LCNG11,3,2. The investigation shows that the closer the initial coefficient b is to the optimal b, the lower the final training error. On the other hand, the initial value of coefficient m does not significantly affect the final training error.
The results of test suite 2 suggest that the fact that the initial coefficients cannot be optimal does not impact the performance or learning of linear regression models.

From the findings in test suite 3, we deduce that the initial value of the coefficients, specifically coefficient b, matters more than the randomness of the initial coefficients. However, in practice, we do not know the optimal value of coefficient b. Therefore, to achieve a good performance in the particular two dimensional Boston Housing problem, one could plot a line through the data and derive approximate coefficients for the optimal solution. Then, the coefficients can be used to approach the optimal solution using gradient descent.

Since the errors produced by the models initialised using true random numbers generally span a wider range than those initialised using an LCNG, we conclude that the randomness of the initial coefficients may not be vital. We acknowledge that in two out of three experiments in test suite 3, the average error of the models initialised using LCNG11,3,2 is lower. However, since the average error of the models initialised using true random numbers is lower than those initialised using LCNG11,3,2 in the second experiment, we cannot conclude that it is generally advisable to initialise the coefficients of a linear regression problem using an LCNG.

Nevertheless, the better solution is to use the analytical solution for a linear regression problem. Thus, this experiment is not realistic in practice. Further, note that we restricted ourselves to two linear number generators: LCNG267,3,31 and LCNG11,3,2. These results may thus not be representative of all LCNGs. To obtain representative results, we must conduct these experiments with a broader range of LCNGs. However, given the limitations of gathering true random numbers (Section 3.9), it was not possible to train enough linear regression models for comparison. Moreover, we used a single data set in test suite 1.That is, we only consider the variation introduced by different parameters, while not capturing the variance over the results introduced by different data sets.

In summary, we could not find any indicators that the lack of randomness or period of LCNGs negatively affect the performance of linear regression problems but note the small number of LCNGs used within these experiments.

2. How are logistic regression models affected by the randomness and period of LCNGs?
We further investigate the difference in the performance of logistic regression models when initialising the coefficients using true random numbers, RANDU, and LCNG267,3,7. The experiments show that the models using a true random number generator perform significantly better when the generated numbers are not augmented to be within the same range.

We compare a true random number generator, RANDU and LCNG267,3,7 in a logistic regression problem using the Titanic data set [9]. We first use the generated numbers without augmentation to initialise the coefficients of a logistic regression problem. We subsequently repeat the experiment using the eight least significant bits of the numbers generated by a true random number generator and RANDU. The experiments show a statistically significant difference in performance for the first set of experiments. RANDU and LCNG267,3,7 achieve the same average training and test error. However, the models using a true random number generator have significantly higher training and test accuracy. The average training accuracy is much larger than the average test

accuracy of all models. This implies that severe overfitting occurs in all models. Thus, while having a significantly higher training and test accuracy, the models using true random numbers do not generalize well to the test set. Note that using only the eight least significant bits means that RANDU has a period of only 64 (Section 2.3). While this allows us to produce numbers within similar ranges, it may not reflect RANDUs general behaviour.

The first experiment shows stark differences in performance between the true random number generator and LCNG267,3,7 and RANDU. To determine the cause of these differences, we must consider the differences between the RNGs.

Firstly, LCNG267,3,7 and RANDU are deterministic and hence not random. Further, LCNG267,3,7 generated numbers in the range $[1,266]$. The range of RANDU is $[12^{31}-1]$. The true random number generator used here produces integers in the range $[1,1e9]$. Note that the true random number generator is generally able of producing numbers within the range $[-1e9,1e9]$. However, for this research, we only use positive numbers.
Further, the true random number generator can produce all 7-tuples within the range $[1,1e9]$ while neither RANDU nor LCNG267,3,7 are full-period-producing random number generators. When generating numbers within the same range, the only difference between the RNGs is their predictability. We see no difference in performance, so we speculate that the logistic regression models are not affected by the randomness of the numbers generated. As a result, we conclude that the range of numbers produced causes a difference in performance in the first experiment. Note that the upper limit of the true random number generator lies within that of LCNG267,3,7 and RANDU. Thus we cannot assume that larger coefficients enable better performance. Instead, the wider range of numbers produced by the true random number generator likely results in the difference in performance. Further, there is no difference in performance between LCNG267,3,7 and RANDU. This suggests that the choice of parameters for LCNGs in this task is arbitrary.

Similarly to the linear regression experiments, we must note that we conducted the experiments using a single LCNG besides RANDU. Again, this is due to the limitations in obtaining true random numbers for comparison (Section 3.9). Hence, using different LCNGs may result in different performances. Nevertheless, the experiments allow us to compare the performance of logistic regression models using a small and a large LCNG (RANDU being the large LCNG) compared to a true random number generator. Moreover, the average accuracy is the same across all random number generators when using numbers within the same range. This suggests that some of the models may have encountered local optima, which prevented them from reaching the global optimum. Increasing the learning rate, however, resulted in the same outcome.

The experiments showed a difference in performance between the models using true random number generators and LCNGs. While the results are not entirely clear, they suggest that a wider range in coefficient combinations allows for higher accuracy when using true random numbers.

3. What are the disadvantages of using LCNGs to select subsets for the training of random forest models?

For Random Forests, we investigated the general effects of linear congruential number generators on the models. Furthermore, we conducted experiments using LCNGs with very short periods that divide the number of samples per tree such that each tree trains on the same set of samples. We find that the small periodicity of LCNGs can pose a significant risk to Random Forest models.

In the first test suite, we find that, while achieving the highest average test accuracy, the training accuracy of the model selecting samples using LCNG637,7,3 has a significantly lower average training accuracy than the Scikit Learn implementation, RANDU, and the true random number generator.

Test suite 2 compares the performance of models using true random number generators to ten models in which each tree trains using the same samples due to the choice of LCNG. Two-tailed tests reveal that the training performance of the models using true random numbers is statistically significantly better. However, the results reveal that we can make no such assumption about the test performance.

Test suite 3 investigates the performance of models using true random number generators when imposing an upper limit and period. We find a decrease in training and test performance as we impose more restrictions.

We will first address the results obtained in the first test suite. In this test suite, the models using LCNG637,7,3 achieve a significantly lower training performance than the other models. We may have obtained this result due to the parameter m of the LCNG. We set $m = 637$. Hence, the largest number this LCNG can generate is 636. The training set, however, contains 791 samples. This means that the LCNG is restricted to the first 636 samples. However, the true random number generator and RANDU can generate all possible numbers in $[0, 790]$. In summary, the models using LCNG637,7,3 are restricted to a smaller number of samples which may have led to this difference in performance. It is unclear why these models achieve the highest average test accuracy. This may be possible due to outliers (a few models with a particularly high test accuracy) and does not necessarily reflect the general population of LCNG-models.

The statistical tests conducted in test suite 2 (Section 3.6.3) indicate that the test performance of the models using true random numbers is not statistically significantly better than those using LCNGs. However, the average test accuracy differs by approximately 10%. It is important to note that we only conducted 14 experiments per RNG due to limitations in generating true random numbers (Section 3.9). A larger number of experiments would be more representative and may result in a different outcome.

In test suite 3 (see Section 3.6.4), we notice a decrease in training and test performance the more restrictions we apply. The decrease is more significant as we apply a period instead of an upper limit. While both the period and upper limit influence the Random Forest model, this implies that the period has a more significant effect. Imposing an upper limit on the true RNG restricts the range of indexes generated but not the number of distinct indexes. Unlike the LCNGs used in test suite 3, the true random number generator can generate every index within the given range. This explains why the models using true random numbers with only an upper limit perform better than those using LCNGs on average. A period, however, restricts a true random number generator

much more than an upper limit. Suppose the period is smaller than the upper limit. In that case, the true random number generator may only generate a small fraction of indexes within the given range, similar to an LCNG. This explains the drastic increase in accuracy compared to the models using true random numbers without restrictions. The performance of the models restricted by a period is similar to that of an LCNG. As we impose both an upper limit and a period, the performance of the models using true random numbers drops below those using an LCNG. This result may be due to chance, as we note that these models have a higher maximum training and testing accuracy than the models using LCNGs.

We must note that these experiments were conducted on a single data set. Thus, the experiments should be conducted on additional data sets to ensure their representativeness.

We set up experiments maintaining the true random number generator's randomness while imposing restrictions similar to an LCNG. Nevertheless, the more restrictions we impose on the true random number generator, the more the training and test accuracy decrease. From this, we deduce that the potentially small period and upper limit of an LCNG negatively affect the performance of a Random Forest model, as opposed to the lack of randomness. As the randomness of the indexes generated does not affect the performance, we speculate that this result applies to all random number generators with a small period.

These results can be compared to those obtained by Bird et al. [2]: Bird et al. use random numbers to select features for the tree, while we use them to select samples. Bird et al. do not find significant performance differences between the models using pseudo-random and true random numbers [2]. Thus, the choice of samples may be more prone to the pitfalls of LCNGs. It is to note, however, that Bird et al. do not make the RNG used explicit and they do not specifically aim to exploit traits of pseudo-random number generators, which may skew the results.

4. How do predictable sequences affect neural networks?
We conduct experiments sorting the data used to train a simple neural network. While the models perform similarly in terms of accuracy, the models trained on sorted data show a significantly higher standard deviation.

The standard deviation indicates the degree of distribution of the model accuracy around the mean. A low standard deviation indicates that most models achieve accuracies close to the mean [10]. Conversely, a large standard deviation indicates that the model accuracy spread further. Thus, the test accuracy of the models trained on sorted data deviates more than that of the models using shuffled data. This can be considered a negative aspect, as the models perform less reliably. As all model settings, such as the number of layers, layer parameters, and learning rate, are fixed for all models, we can attribute these differences to sorting the data.

Again, we must note that only a single data set was used and that different data sets may produce different results. Bird et al. [2], for instance, find ambiguous results. Neural networks using true random numbers to initialise weights achieve better accuracy on some data sets but not others compared to models using pseudo-random numbers. While

the task in [2] is different, this remains a valid concern. We have conducted several experiments on the MNIST data set, so we are confident that the results represent this dataset. However, further experiments are required to ensure that this translates to the general case.

Further, we note that we did not use a random number generator to sort the training data. Thus, one may suggest that this experiment is unlikely to happen in practice. However, we aim to point out worst-case scenarios and inform about what can happen if one fails to be vigilant.

5. Can timeseries models learn from predictable sequences?
We conduct 60 experiments to investigate the effect of predictable sequences on LSTMs. Overall, we find that the models trained on sequences produced by LCNG4,3,1 perform the best, followed by models trained on sequences based on LCNG131,5,1. However, models using true random sequences and sequences produced by RANDU modulus 5 perform the worst overall.

We first note that LCNG4,3,1 has a period of only two and is thus highly predictable. The models perform excellently on both tasks (stimulus classification and stimulus prediction). Note that if the stimuli sequence is deterministic, so is the sequence of stimulus responses. Thus, the model may have learned this underlying sequence. The fact that the experiments using sequences produced by LCNG131,5,1 also result in a low MSE supports this.

Further supporting this speculation is the comparatively high average MSE on both tasks (stimulus prediction and classification) of the models trained using true random sequences. Moreover, note the drastic decrease in MSE between the tasks. The models show a high average MSE in the task of stimulus prediction. However, compared to this, their average MSE on the stimulus classification task is very low. This indicates that, while unable to predict a true random sequence, the models may pick up on underlying patterns in the sequences of stimulus responses, leading to a much lower average MSE. One may suggest that the high MSE of the models using numbers produced by RANDU modulus five on both tasks contradicts this. While it is true that RANDU modulus five has a fixed period, we must consider the *length* of the period. The period of RANDU modulus five is above 100. We gather this from plots depicting the sequences of stimuli and stimulus responses in Appendix B.4. Thus, a long period (that is, a period longer than the sequence length) may prevent the LSTM models from picking up on deterministic sequences. The results in Section 3.8 emphasize the role of the period of an RNG: The smaller the period, the lower the average MSE on the tasks of stimulus prediction and classification.

While a small period and deterministic sequence result in a lower MSE, this may not translate to practical situations, and thus the results are skewed.

We conclude that LSTMs pick up on the deterministic patterns in sequences produced by LCNGs. A long period, however, may prevent this. The major pitfalls here are the deterministic nature of LCNGs and a potentially small period. We speculate that this result translates to all pseudo-random number generators with the same properties.

For these experiments, we used five different LCNGs. While this allows us to speculate

about the performance of other LCNGs, further experiments need to be conducted to ensure that this translates to all LCNGs. Further note that we did not fine-tune the models used in this experiment. A longer sequence, larger samples, or different model parameter settings may have improved the performance of the models using true random numbers. Similarly, reducing the sample size may have negatively affected the models using LCNG- and RANDU-sequences. While we speculate about the causes for the results gathered here, our results do not allow us to make definitive statements. To pinpoint the exact causes of the behaviour seen here, we would have to look at the learning process of the LSTMs and the weights of the hidden units in each learning epoch.

5.2 Future Work

Within this work, we focused on the effects of linear congruential number generators, as their specific non-random traits have been well-studied in the past [4, 24, 25]. We utilised them as a first step to gauging the effects of bad randomness. Nowadays, however, LCNGs are rarely used. Most popular programming languages, such as Python [1] and libraries, for instance NumPy [2], use advanced pseudorandom number generators. Python, for example, uses the Mersenne Twister [25], which is considered to be state-of-the-art. As many scientists and programmers are likely to rely such pseudorandom number generators for machine learning purposes, it is important to thoroughly investigate these state-of-the-art generators in the context of machine learning to uncover potential pitfalls.

Further, we focused on the fundamental machine learning techniques within this research. There exists an abundance of machine learning models, each with its own traits and areas of utilisation. Conducting experiments similar to the ones detailed in Chapter 3 is vital to gain further insights into the effects of pseudorandom numbers.

The experiments detailed in Chapter 3 could be expanded in the future. For instance, we only consider the effect of pseudorandom numbers in the selection process of samples for a particular Decision Tree in a Random Forest. We do not, however, consider that random numbers can also be used to pick a feature on which to split.

In relation to neural networks and LSTMs, we focused on the order in which data was presented. Future endeavours should explore the effect of pseudorandom numbers in the initialisation of the weights of these models.

Finally, the experiments conducted here should be repeated with a wider variety of LCNGs and data sets to ensure representative results.

Pseudorandom numbers are a vital component of machine learning on which many methods rely. Therefore, it is vital to continue this research.

[1] https://www.python.org/
[2] https://numpy.org/

Chapter 6

Conclusions

In this report we first explained pseudo-random number generators and how they compare to true random number generators. We further reviewed previous research in the domain of linear congruential number generators and concluded that the research in the effects of pseudo-random numbers on machine learning methods is still in its infancy. However, extensive research has been done with regards to the non-random traits of linear congruential number generators.

We subsequently conducted a number of experiments aiming to exploit these traits. To this end, we investigated linear and logistic regression models, Random Forests, Neural Networks and Long-Short Term Memory models.

We found that the randomness of the numbers chosen to initialise a linear regression model had little effect on the performance of the model. The range in which we picked numbers had a far greater impact. We found that, the closer the initial coefficients chosen are to the optimal coefficients of the model, the better the model performs.

With regards to the random forest models, we found that the period and range of pseudo-random number generators have a grave impact. The period and range of an LCNG may be small. This affects how many samples are available to a decision tree in the random forest. This impacts the variety of data which the decision tree is trained on and hence reduces the accuracy. Further, the coefficients of an LCNG can be chosen such that every decision tree in the random forest is trained on the same training sample, obliterating the ensemble learning aspect of random forests.

While we find an increased standard deviation in neural network models trained on sorted data, further research is required to determine the exact causes of this.

Finally, we find that, again, the periodicity of pseudo-random number generators has a far greater impact. While the models appear to learn deterministic traits from sequences with small periods, this problem does not appear when the RNG has a sufficiently large period.

In summary, many machine learning models are impacted by the potentially short period of LCNGs, as opposed to their randomness. Thus, we speculate that this applies to all PRNGs with a small period.

Bibliography

[1] Armin Alaghi and John P. Hayes. Survey of stochastic computing. *Transactions on Embedded Computing Systems*, 12(2 SUPPL.):1–19, 5 2013.

[2] Jordan J. Bird, Anikó Ekárt, and Diego R. Faria. On the effects of pseudorandom and quantum-random number generators in soft computing. *Soft Computing*, 24(12):9243–9256, 2020.

[3] Christopher M. Bishop. *Pattern Recoginiton and Machine Learning*. Springer, 2006.

[4] Joan Boyar. Inferring Sequences Produced by Pseudo-Random Number Generators. *Journal of the ACM (JACM)*, 36(1):129–141, 1989.

[5] Jason Brownlee. How to Implement Linear Regression From Scratch in Python, 10 2016.

[6] Jason Brownlee. How To Implement Logistic Regression From Scratch in Python, 10 2016.

[7] Jason Brownlee. How to Choose an Activation Function for Deep Learning, 1 2021.

[8] Jason Brownlee. Weight Initialization for Deep Learning Neural Networks, 2 2021.

[9] Bruno S. Frey, David A. Savage, and Benno Torgler. Who perished on the titanic? The importance of social norms. *Rationality and Society*, 23(1), 2011.

[10] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly, 2017.

[11] Haahr and Mads. RANDOM.ORG - Introduction to Randomness and Random Numbers.

[12] Mads Haahr. random.org: Introduction to Randomness and Random Numbers. *Statistics*, pages 1–4, 1999.

[13] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7(12), 1964.

[14] Harrison D. and Rubinfeld D. Boston Dataset, 1978.

[15] Adam Hayes, Katharine Beer, and Jefreda R. Brown. T-Test Definition, 2022.

[16] David H.K. Hoe, Jonathan M. Comer, Juan C. Cerda, Chris D. Martinez, and Mukul V. Shirvaikar. Cellular automata-based parallel random number generators using FPGAs. *International Journal of Reconfigurable Computing*, 2012, 2012.

[17] IBM Cloud Education. What are Neural Networks? — IBM, 8 2020.

[18] Knuth D. E. *The Art Of Computer Programming - Third Edition*, volume 2 / Seminumerical A.... Addison-Wesley, 1997.

[19] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. ... *Science Department, University of Toronto, Tech. ...*, 2009.

[20] LeCun Yann, Cortes Corinna, and Burges Chris. MNIST handwritten digit database.

[21] Pierre L'ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):1–40, 8 2007.

[22] D.H. (University of California) Lehmer. Mathematical Methods in Large-scale Computing Units. In *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery*, 1949.

[23] Jonathan Lockhart, Khaled Al Rawashdeh, and Carla Purdy. Verification of Random Number Generators for Embedded Machine Learning. *Proceedings of the IEEE National Aerospace Electronics Conference, NAECON*, 2018-July:411–416, 2018.

[24] G. Marsaglia. Random Numbers Fall Mainly in the Planes. *Proceedings of the National Academy of Sciences*, 61(1):25–28, 1968.

[25] Makoto Matsumoto and Takuji Nishimura. This paper is to appear in \ ACM Transactions on Modeling and Computer Simulations : Special Issue on Uniform Random Number Generation " (Early in Mersenne Twister : A 623-dimensionally equidistributed uniform pseudorandom number generator. *Discrete Mathematics*, 1998.

[26] Florian Neugebauer, Ilia Polian, and John P. Hayes. Building a Better Random Number Generator for Stochastic Computing. In *Proceedings - 20th Euromicro Conference on Digital System Design, DSD 2017*, 2017.

[27] Melissa E O'neill. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. *ACM Transactions on Mathematical Software*, 2014.

[28] Leila Reddy, Andrea Alamia, Saeed Reza Kheradpisheh, Bin Yan, Kai Qiao, Jian Chen, Linyuan Wang, Chi Zhang, Lei Zeng, and Li Tong. Category Decoding of Visual Stimuli From Human Brain Activity Using a Bidirectional Recurrent Neural Network to Simulate Bidirectional Information Flows in Human Visual Category Decoding of Visual Stimuli From Human Brain Activity Using a Bidirectional Recurrent Neural Network to Simulate Bidirectional Information Flows in Human Visual Cortices. *Frontiers in Neuroscience*, 2019.

[29] Edgar H Sibley, Stephen K Park, and Keith W Miller. Computing Practices Random Number Geuerators: Good Ones Are Hard To Fin. *Communications of the ACM*, 31(10), 1988.

[30] The Python Developers. random — Generate pseudo-random numbers — Python 3.10.4 documentation.

[31] J Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, 34(34), 1956.

[32] Bin Yan, Kai Qiao, Jian Chen, Linyuan Wang, Chi Zhang, Lei Zeng, and Li Tong. Category Decoding of Visual Stimuli From Human Brain Activity Using a Bidirectional Recurrent Neural Network to Simulate Bidirectional Information Flows in Human Visual Category Decoding of Visual Stimuli From Human Brain Activity Using a Bidirectional Recurrent Neural Network to Simulate Bidirectional Information Flows in Human Visual Cortices. *frontiers in Neuroscience*, 7 2019.

[33] Guangyi Zhang, Vandad Davoodnia, Alireza Sepas-Moghaddam, Yaoxue Zhang, and Ali Etemad. Classification of Hand Movements from EEG Using a Deep Attention-Based LSTM Network. *IEEE Sensors Journal*, 20(6), 2020.

Appendix A

Background - Additional Information

This section contains additional, non-essential information for the machine learning methods and other techniques described in Chapter 2.

A.1 Machine Learning Methods - Decision Trees

In continuation of Section 2.1, we provide a simple decision tree example. Consider a dataset containing the petal widths and lengths of flowers of the type Rosa Alba and C. Indicum. Figure A.1 depicts a sample Decision Tree classifying the samples based on their petals widths and lengths.

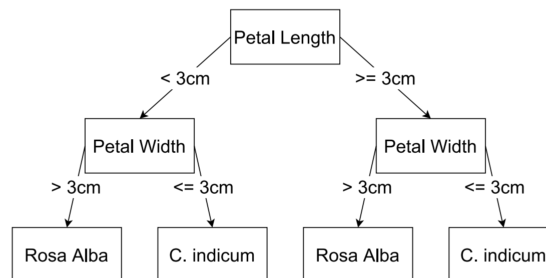


Figure A.1: A sample decision tree with the goal of classifying flowers of type Rosa Alba and C. Indicum based on their petal width and length. Note that this example is for demonstration purposes only and does not reflect real petal widths and lengths.

A.2 Statistical Tests

To determine if the performance difference between two models is not simply by chance, we will use statistical tests. Specifically, we will use paired t-tests, which measure if there is a significant difference between the means of two populations (here: model performances) [15]. In this context, we generally assume a null-hypothesis suggesting that the two means are equal (i.e. there is no difference in performance) [15]. The *p*-value indicates the probability that the results witnessed occurred merely by chance [15].

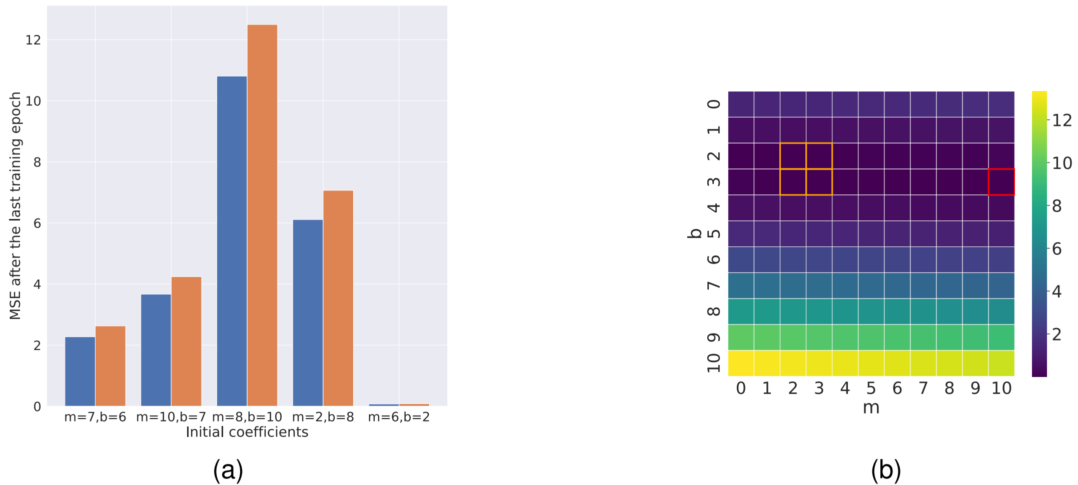
Appendix B

Experiments - Additional Information

This section contains additional, non-essential, information relating to the experiments conducted.

B.1 Linear Regression

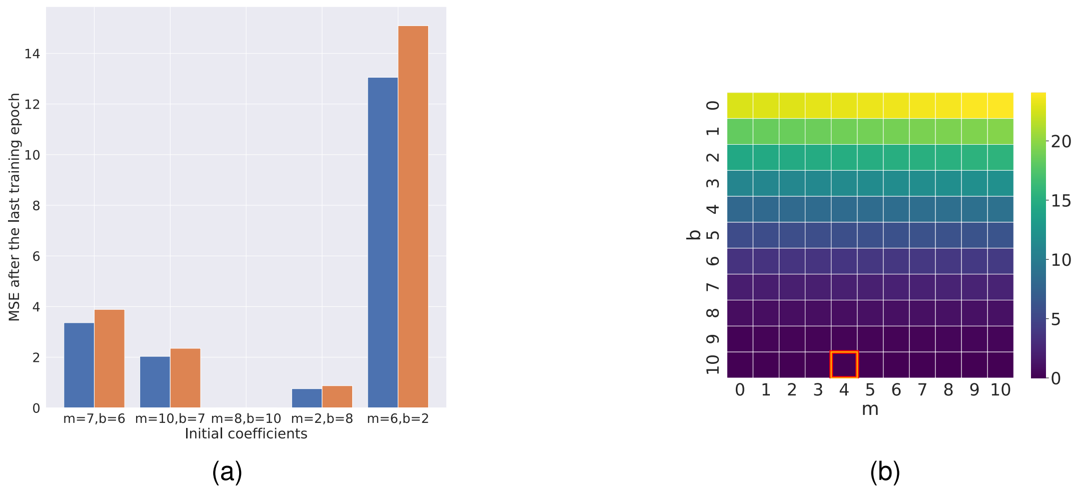
Figures B.1 and B.2 depict the error comparison and and final error depending on initial coefficients for the second and third experiment mentioned in Section 4.1.3.



(a)

(b)

Figure B.1: Figure (a) depicts the error comparison between the models initialised using the different indices generated by the LCNG. Figure (b) depicts the final error of linear regression models per initialisation coefficients in experiment 2. The red square indicates the optimal coefficients. The orange square indicates the final coefficients found by the LCNG-model that result in the least error at training time.



(a)

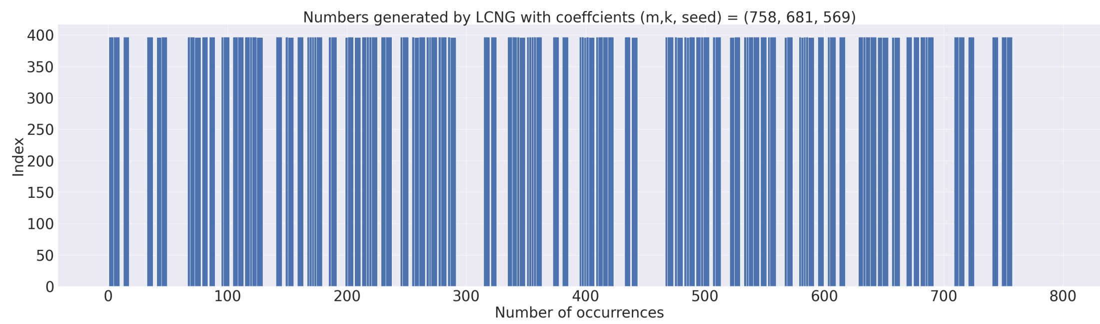
(b)

Figure B.2: Figure (a) depicts the error comparison between the models initialised using the different indices generated by the LCNG. Figure (b) depicts the final error of linear regression models per initialisation coefficients in experiment 3. The red square indicates the optimal coefficients. The orange square indicates the final coefficients found by the LCNG-model that result in the least error at training time.

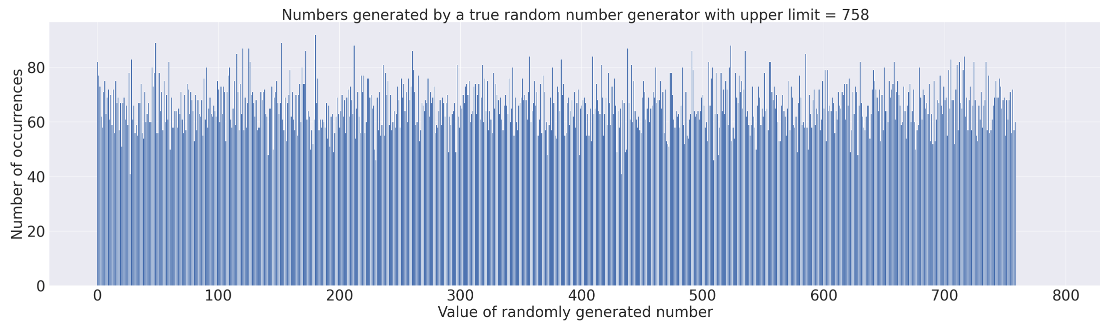
B.2 Random Forest

Figure B.3 visualises the indexes produced by an LCNG and true random number generator. Note that the graph depicting the true random number generators resembles that of the LCNG more as we impose more restrictions (that is, an upper limit and period).

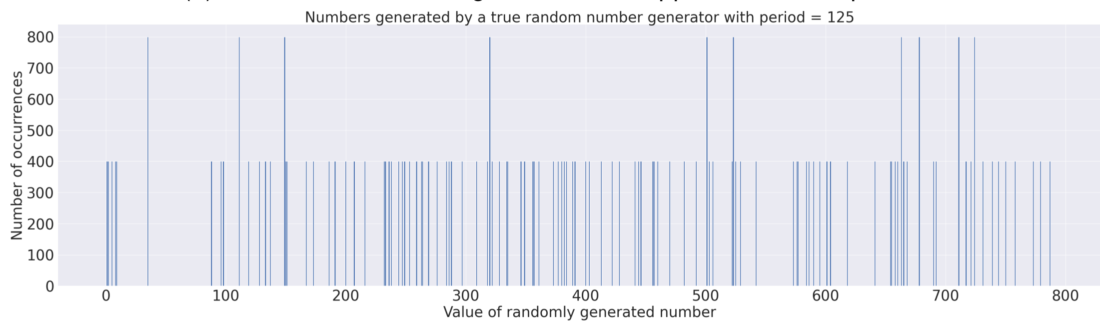
Figure B.3 shows the indices generated by an LCNG and a true random number generator with restrictions as described in Section 3.6.4. The graph in Figure B.3a depicts the indices generated by LCNG758,681,569. The graph is rather sparse, with several numbers never being generated. Further, the numbers that *are* generated occur the same number of times. Figure B.3b depicts the numbers generated by a true random number generator with an upper limit imposed. Every number is generated within this range, and the number of occurrences differs for each specific index. When imposing a period, see Figure B.3c, the graph becomes sparse. This makes sense, as a period restricts the number of indices that can be produced. Note that, unlike in the LCNG, there is some variation in the number of occurrences. Finally, Figure B.3 depicts the indices generated by a true random number generator when introducing both a period and an upper limit. The graph looks similar to that in Figure B.3c, with the difference that no indices above the upper limit are generated.



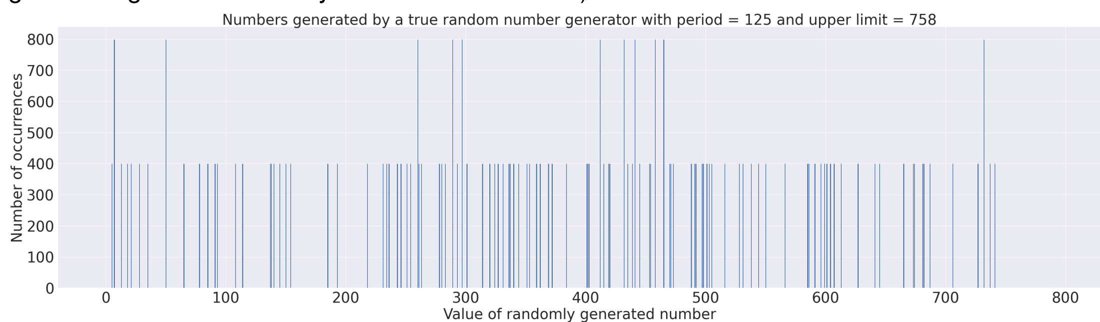
(a) Visualisation of the indexes generated by the LCNG with m=758, k = 681 and seed 569.



(b) True random number generator with upper limit 758 imposed.



(c) True random number generator with a range of 125 distinct numbers imposed (that is, the generator generates exactly 125 distinct numbers).



(d) True random number generator with a range of 125 and upper limit of 758 imposed.

Figure B.3: Visualisation of the numbers produced by random number generators with different restrictions imposed.

B.3 Neural Network

Figure B.4 shows the structure of the neural network used to classify MNIST digits.

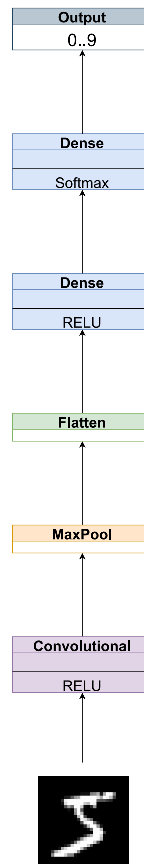


Figure B.4: The structure of the neural network used to classify MNIST digits. The colors indicate the different types of layers.

B.4 Time Series

Figure B.5 depicts the structure of the LSTM model used in the timeseries experiments.

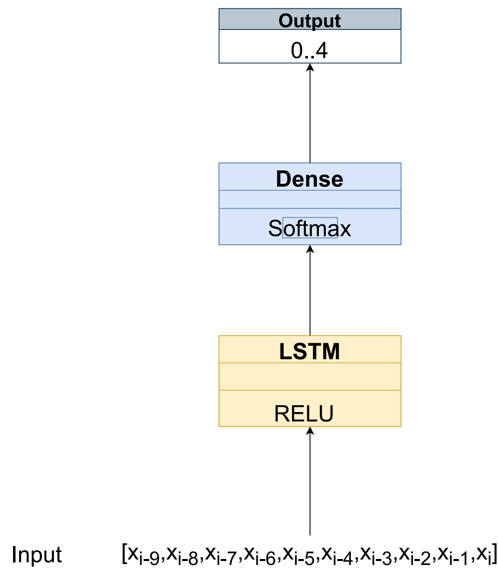


Figure B.5: The figure depicts the structure of the LSTM-model used within the timeseries experiments (Section 3.8). The model consists of a single LSTM layer using a RELU activation function followed by a Dense layer using a softmax activation function. The input to the model are the current stimulus response (that is, the response at time i), x_i, as well as the previous nine responses, $x_{i-9},..,x_{i-1}$. The model outputs the index of the stimulus shown (zero-indexed). The colors indicate the different types of layers.

Figure B.6 depicts the first 100 stimuli produced by the lowest two bits of numbers produced by LCNG131,5,1 and RANDU, the numbers generated by LCNG131,5,1 and RANDU modulus 5 and a true random number generator. The figure shows obvious repetitions for the sequences produced by both variations of LCNG131,5,1. The former repeats starting between index 60 and index 70. The latter repeats within the same range of indexes. The sequence produced by the lowest two bits of numbers generated by RANDU repeats at a similar point. When considering the sequence produced by RANDU mod 5, however, we see no such repetition. This result is rather surprising as both variations of LCNG131,5,1 showed similar periods.

Unsurprisingly, the graph depicting the true random number generator is aperiodic.

Figure B.7 depicts the stimulus responses corresponding to Figure B.6. We see that the stimulus responses behave in sync with the the sequence of stimuli and also show similar periods.

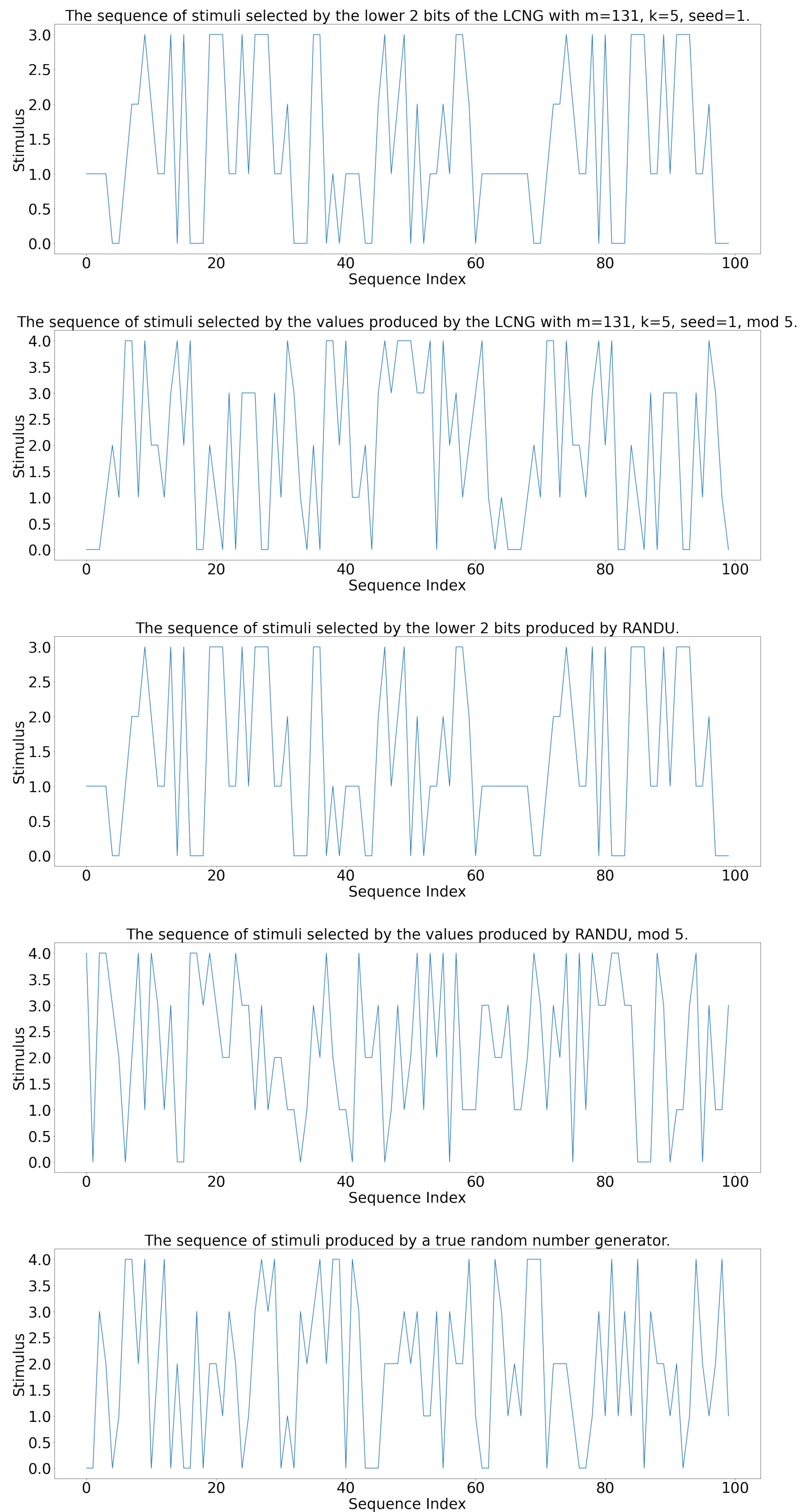


Figure B.6: This figure shows a visual representation of the first 100 stimuli generated by different RNG.

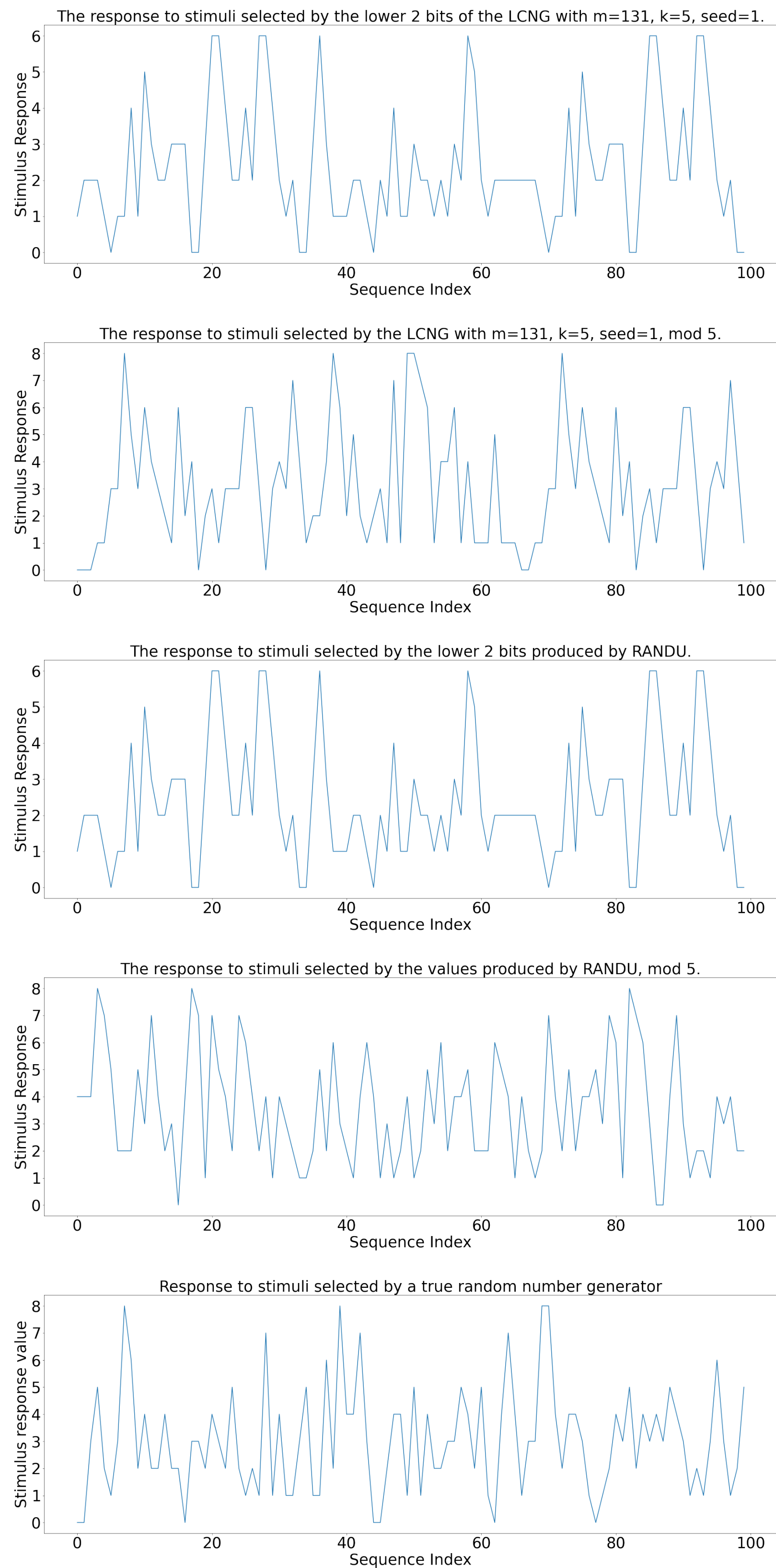


Figure B.7: This figure shows a visual representation of the first 100 stimuli responses, derived from the sequences produced by different RNGs as depicted in Figure B.6

Appendix C

Code Implementation

This section contains the Python 3 code for various models used within this research.

C.1 Linear Regression

This section contains the code used for all linear regression tasks. Note that, for readability-purposes, code to print statistics has been omitted.

```
import numpy as np
import matplotlib.pyplot as plt

import sys

class LinearRegression():

    def __init__(self):
        self._coef = None

    # Finds the analytical solution to a linear regression
    # problem and updates the coefficients
    def fit_analytical(self, X, y):
        self._coef =
            (np.dot(np.linalg.inv(np.dot(X.T,X)), X.T).dot(y))

    # Find the optimal coefficients using gradient descent
    # X --> Training samples
    # y --> Training labels
    # coef_init --> Initial coefficient value
    # l_rate --> Learning rate for gradient descent
    # epochs --> Number of epochs for gradient descent
    def fit(self, X, y, coef_init, l_rate = 0.001, epochs = 50):
        errors = []
        coefficients = np.array(coef_init)
        y = y.reshape(y.shape[0], 1)
        for epoch in range(epochs):

            yhat = self.predict(X, coef_init)
            yhat = yhat.reshape(yhat.shape[0], 1)
```

```
            loss = mse(y, yhat)
            gradient = mse_derivative(X, y, yhat)
            coef_init = coef_init - l_rate*gradient

            coefficients = np.c_[coefficients, coef_init]
            errors.append(loss)
        # save the coefficients in the class variable
        self._coef = coef_init
        return errors, coefficients

    # Predict the value for samples
    # X --> Samples
    # coef --> coefficients to use for prediction
    def predict(self, X, coef):
        return np.dot(X, coef)
```

Listing C.1: Linear Regression Implementation

C.2 Logistic Regression

```
import numpy as np

class LogisticRegression():

    def __init__(self):
        self._coef = None

    # Conduct gradient descent to approximate the optimal
    # coefficients of this problem.
    # X --> Training data
    # y --> Training labels
    # coef_init --> Initial coefficients
    # n_epoch --> Number of epochs for gradient descent
    # l_rate --> Learning rate for gradient descent
    def fit(self, X, y, coef_init, n_epoch, l_rate):
        for epoch in range(n_epoch):
            sum_error = 0
            for (row,label) in zip(X,y):
                prediction = self.predict(row, coef_init)
                error = label-prediction
                sum_error += error
                #Calculate the gradient of the log likelihood
                # with respect to the weights
                coef_init +=
                    l_rate*error*prediction*(1-prediction)*row

        self._coef = coef_init

    # Given samples and coefficients,
    # predict the label for the samples
    # X --> Samples to predict
    # coef --> Coefficients to use
    def predict(self, X, coef):
        prediction = 1.0/(1.0+np.exp(-(np.sum(X*coef, axis=1))))
```

```
        return prediction
```

Listing C.2: Logistic Regression Implementation

C.3 Random Forest

This section contains the Python 3 implementation of a Random Forest model as it is used in Section 3.6.

```
from sklearn.tree import DecisionTreeClassifier
import numpy as np

class RandomForest():

    # k --> Number of trees to grow
    def __init__(self, k):
        self.k = k

    # Selects the rows at index <rows> from
    # the samples and labels
    # rows --> array containing indexes of rows to select
    # X --> training samples
    # y --> training labels
    def pick_rows(self, rows, X, y):
        return X[rows].reshape((rows.shape[0], X.shape[1])), y[rows]

    # Create appropriate number of
    # decision trees, train each and save in
    # class variable
    # X --> Training samples
    # y --> Training labels
    # index_array --> array of array, contains the indexes
    # of the samples to use per tree
    def fit(self, X, y, index_array):

        self.forest = []
        for index in index_array:

            # Pick a subset of the data
            X_sub, y_sub = self.pick_rows(index, X, y)

            # Grow decision tree using scikit learn impl.
            dt = DecisionTreeClassifier()
            dt.fit(X_sub, y_sub)

            self.forest.append(dt)

    # Classify samples X
    # X --> Samples to classify
    def predict(self, X):

        # Ensure that the forest has been
        # trained beforehand, else notify user
        if self.forest == None:
```

```
            print("Please train your forest.")
            return

        predictions = np.zeros((X.shape[0],))

        for tree in self.forest:
            prediction = tree.predict(X)
            predictions+=prediction

        # If sum > k/2 return 1 else 0
        return predictions > self.k/2.0
```

Listing C.3: Random Forest Implementation