Evaluating The Predictability Of Pseudo-Random Number Generators Using Supervised Machine Learning Algorithms

by

Apprey-Hermann Joseph Kwame

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Computing and Information Systems

YOUNGSTOWN STATE UNIVERSITY

MAY 2020

Evaluating The Predictability Of Pseudo-Random Number Generators Using Supervised Machine Learning Algorithms

Apprey-Hermann Joseph Kwame

Signature		04/26/2020
	Apprey-Hermann Joseph Kwame, Student	Date
Approvals	Dr. John R. Sullins, Thesis Advisor	Date
	Dr. Alina Lazar, Committee Member	Date
	Dr. Abdu Arslanyilmaz, Committee Member	Date
	Dr. Salvatore A. Sanders, Dean of Graduate Studies	Date

ABSTRACT

Pseudo Random Number Generators (PRNG) are algorithms that help to create randomness in programs, but they are not truly random like the randomness obtain from physical processes. Despite this, PRNGs are widely used in applications that require random numbers, from gaming to security.

This research evaluates the use of machine learning algorithms to predict the numbers generated by PRNG. These experiments involve three commonly used PRNG from C++, Python, and Java, and uses two Machine Learning algorithms, Linear Regression and Artificial Neural Networks.

The outcome of the research confirms the possibility that machine learning algorithms can be trained to predict certain PRNGs. Even when trained with a small amount of data, there is evidence that machine learning algorithms can be used to predict the values created by pseudorandom number generators. Given that linear regression algorithm and a simple regression neural network were able to produce fairly good predictions with reasonable accuracy in our experiments, it is believable that more complex machine learning algorithms such as deep learning and recurrence algorithms might produce still better results.

ACKNOWLEDGMENT

I would like to express my special thanks of gratitude to my supervisor Dr. John S. Sullins, for the immense support to this particular project and in many other ways during the entire time of my graduate program at YSU. I will also like to thank the department chairperson Dr. Bayrak Coskun, my academic advisor Dr. Alina Lazar, and the entire faculty of the CSIS department of the school of computer science for their various contributions to the successful completion of my graduate program.

As an international student I can never be grateful enough for the continues support offered me by the IPO of YSU throughout my school year at YSU. The graduate school of YSU gave me the opportunity to pursue my academic aspiration in the USA, coming from a country like Ghana in Africa that opportunity is golden. And I will always be grateful.

Finally, I would also like to thank my family and friends in the USA and overseas for their help in diverse ways in my graduate education at YSU especially Ebo Fosu Edonu.

Dedication

I dedicate this work to my grandmother Mrs Mercy Awudu Atupra

Table of Contents

.

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Random number generation is a key component in most computing systems. Gaming applications, gambling systems, and security systems among others use randomization to achieve different objectives.

Randomization is used to implement AI in many gaming applications as it helps to reduce predictability, making games sustain their difficulty even after several attempts of play. In gambling, it could be used to ensure fairness in terms of chances and to depict the concept of luck. In security systems, randomization is used to generate keys and helps to increase the level of difficulty of unauthorized access in other ways. Unlike games and gambling systems where randomization can be said to be very useful, good random number generation is crucial for effective computer security.

Usually randomization is implemented with pseudo random number generators (PRNG) in computing systems. However, PRNG are not truly random, as they are not actual physical processes like flipping a coin. The fact that PRNG are not truly random raises serious concerns, especially their use in security systems like crypto PRNG where funds are involved.

Machine Learning (ML) is an area of artificial intelligence in which algorithms learn to perform predictions based on experience. A computer program is said to learn from experience E with respect to some class of task T and performance measure P, if its performance at tasks in T, as measured by P improves with experience E [26]. In other words, machine learning differs from the traditional approach of a computer program containing explicitly typed set of instructions to perform a task. Among some of the successful applications of Machine Learning are learning to recognize spoken words, learning to drive an autonomous vehicle, learning to classify new astronomical structures, and learning to play world-class backgammon [26]. Machine learning applications in cancer prognosis and prediction [27]. There are several algorithms for implementing different tasks in machine learning. The choice of an algorithm depends on the nature of problem and data available. Recurrent Learning, Reinforcement Learning, and Supervised Learning are reviewed in this research work. More specifically, linear regression and neural network.

This work seeks to evaluate the predictability of PRNGs using ML and AI. Specifically, we ask the question whether future values of certain PRNGs can be predicted by a machine learning algorithm based on the seeds used.

1.2 Contributions

The outcome of this research work can contribute knowledge related to the use of PRNGs, as well as how secure certain PRNGs are against an attack based on ML.

1.3 Research Questions

This deductive research work sought to test the theory that ML can be use to predict PRNGs by answering the main question how predictable is PRNG by ML algorithms? This major question further put into the following sub questions;

- Can ML learn from a seed or a sequence of PRNG random numbers and make predictions about future values?
- How accurate is the predictability of ML on certain PRNG?

1.4 Organization of Study

The entire work in presented in five major chapters. Chapter one introduces the work with the background to the study, which entails the motivation to the work, the contributions of the study as well as the specific questions that lead the study. Chapter two reviews related literature to the main concept of the study, randomization and specific areas in machine learning. Chapter three describes and demonstrates the simple linear regression algorithm and the neural network learning algorithm to be use in this research and the evaluation procedure as followed in the work. Chapter four focuses on the actual evaluation process for all the three PRNGs selected for this research work. Chapter five presents the testing and results from the evaluation of the selected PRNGs. Chapter six ends the study with the findings, recommendations, and conclusion.

2 Background and Related Work

2.1 Randomization

In computing, random values (mostly in the form of numbers) are generated by pseudorandom number generators (PRNG), also known as a deterministic random bit generators (DRBG). A PRNG is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers. The PRNG-generated sequence is not truly random, because it is completely determined by an initial value, called the PRNG's seed (which may include truly random values). Although sequences that are closer to truly random can be generated using hardware random number generators, pseudorandom number generators are important in practice for their speed in number generation and their reproducibility.

PRNGs are central in applications such as simulations (e.g. for the Monte Carlo method), electronic games (e.g. for procedural generation), and cryptography. Cryptographic applications require the output not to be predictable from earlier outputs, so more elaborate algorithms which do not inherit the linearity of simpler PRNGs are needed.

Good statistical properties including correlation and regression analysis are a central requirement for the output of a PRNG. In general, careful mathematical analysis is required to have any confidence that a PRNG generates numbers that are sufficiently close to random to suit the intended use.

Some widely used PRNGs include the linear congruential generator and the Mersenne twister.

The linear congruential generator is one of the most common and simplest PRNG. It uses the following recurrence to generate numbers.

$X_{n+1} = (aX_n + b) \bmod m$

In the formula above, a, b and m are large integers, and X_{n+1} is the next value of X in a series of pseudo-random numbers. The maximum number of values the formula can produce is one less than the modulus, m-1. To avoid certain non-random properties of a single linear congruential generator, several such random number generators with slightly different values of the multiplier coefficient, a, can be used in parallel, with a "master" random number generator that selects from among the several different generators.

Developed by Makoto Matsumoto and Takuji Nishimura in 1997 [25], the Mersenne Twister is by far the most widely used pseudorandom number generator (PRNG). It was developed to specifically rectify most of the flaws found in older PRNGs. “From a choice of parameters, the algorithm provides a super astronomical period of 219937 – 1 and 623 – dimensional equidistribution up to 32 bits accuracy, while consuming a working area of only 624 words” [25]. Its name derives from the fact that its period length is chosen to be a Mersenne prime. In mathematics, a Mersenne prime is a prime number that is one less than a power of two. That is, it is a prime number of the form $M_n = 2^n - 1$ for some integer n. The underlying implementation in C is both fast and thread safe. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

Most computer programming languages include functions or library routines that provide random number generators. They are often designed to provide a random byte or word, or a floating point number uniformly distributed between 0 and 1.

The quality of randomness of such library functions varies widely from completely predictable output, to cryptographically secure. The default random number generator in many languages, including C++, Python, Ruby, R, IDL and PHP is based on the Mersenne Twister algorithm and is not enough for cryptography purposes, as is explicitly stated in the language documentation. Such library functions suffer limitations like poor statistical properties, either generating values that are not evenly distributed, or having excessive duplication among values generated. A typical example is the Python random() function used in this research. These functions are often initialized using a computer's real time clock as the seed, since such a clock generally measures in milliseconds, far beyond the ability of an outside entity to predict. These functions may provide enough randomness for certain tasks (for example video games) but are unsuitable where high-quality randomness is required, such as in cryptography applications, statistics, or numerical analysis.

John von Neumann cautioned about the misinterpretation of a PRNG as a truly random generator, and joked that "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin” [31]. That is, true random number generation cannot be realized with PRNG, since with enough patience and enough computing resources the sequence can be determined, particularly with modern computers.

2.2 Pseudo random number generators (PRNG) considered in this research

While there are countless number of PRNG, three of the most used were considered in this research work. These PRNG are classes and methods used in the programming languages C++, Java, and Python.

2.2.1 The rand() and srand() functions of the C++ PRNG.

The rand() and srand() functions are used in C/C++11, to pseudo generate random numbers. The Engines and engine adaptors used by C++ PRNG include the, linear congruential engine, Mersenne twister engine, subtract with carry engine, discard block engine, independent bits engine, shuffle order engine.

The rand() function can be used alone to generate random numbers, but if a sequence of random numbers are generated with rand() function alone the same sequence of numbers are generated again and again every time the program runs.

The srand() function is used to solve this problem by setting the starting point sequence. If srand() is not called, the rand() seed is set as if srand(1) were called at program start. Any other value for seed sets the generator to a different starting point.

However, the PRNG is often only seeded once, before any calls to rand() at the start of the program. A standard way to set the seed to a random value is to use the result of a call to srand(time(0)) as the seed. The function time() returns a time_t value which varies every time it is called, which means the pseudo-random numbers will vary for every time the program runs. The relationship between srand() and rand() is that srand() sets the seed for the rand() function. The two were used together in this research.

2.2.2 The Java PRNG (java.util.Random class)

Java offers three ways to generate random numbers in the form of built-in methods and classes including the java.util.Random class, the Math.random method, and the ThreadLocalRandom class.

The java.util.Random class uses a 48-bit seed, which is modified using a linear congruential formula [34]. can generate random numbers in the form of intergers, doubles, longs and even Boolean.s An instance of the class must be constructed and invoked to generate the appropriate type using the methods nextInt(), nextDouble(), nextLong(). Passing some argument in some of the methods can influence their output; for example nextInt(6) will generate numbers in the range 0 to 5.

The Math class contains various methods for performing various numeric operations such as calculating exponentiation, logarithms etc. One of these methods is random(), which returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. The returned values are chosen pseudo-randomly.

The java.util.concurrent.ThreadLocalRandom class is introduced in Java 1.7 to generate random numbers of type integers, doubles, and Booleans .

For security sensitive application java provides SecureRandom which belongs to the java.security class package.

2.2.3 The random() Python PRNG

Python uses the random() module for implementing various pseudo-random numbers, including integers, number sequences, and random permutation random sampling among others.

Almost all module functions depend on the basic function random(), which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. The functions supplied by this module are actually bound methods of a hidden instance of the Random class. You can instantiate your own instances of Random to get generators that don't share state.

The Random class can also be sub classed if you want to use a different basic generator of your own devising. This is done by overriding the random(), seed(), getstate(), and setstate() methods. Optionally, a new generator can supply a getrandbits() method, which allows randrange() to produce selections over an arbitrarily large range.

The random module also provides the SystemRandom class which uses the system function os.urandom() to generate random numbers from sources provided by the operating system.

Among the commonly used methods, the random.random() and random.seed() were used in this research work. random.seed(x) initializes the basic random number generator. The optional argument x can be any hashable object. If x is omitted or None, current system time is used; current system time is also used to initialize the generator when the module is first imported. If randomness sources are provided by the operating system, they are used instead of the system time (see the os.urandom() function for details on availability). If x is not None or an int, hash(x) is used instead. If x is an int, x is used directly. random.random() returns the next random floating point number in the range (0.0, 1.0).

2.3 Supervised Learning

Supervised learning (SL) is an approach of machine learning in which the desired result of each training example is known. SL is intended to find patterns in data that can be applied to an analytics process. Basically, each data in a supervised learning approach is labeled based on its category. The goal of the supervised learning algorithm is to learn the unique features of each of the categories by studying all the items in each category. For example, a set of images of shapes (such as triangles, squares and circles) can be used. This data could be made up of 150 shapes which consist of 50 circles, 50 triangles and 50 squares, with each image is labeled accordingly. A fraction of the 150 (120 for instance) will be used as the training dataset and the remaining 30 used as the testing dataset. The labelling process might seem like a very difficult task especially when dealing with very large dataset, but it is the most important part of supervised learning as the labelling serves as a guide for the learning algorithm.

SL algorithms can be further broken into two categories: classification and regression. The main difference between these is in terms of their outputs. The output of classification algorithms are in the form of objects or some form of categories (for example shapes, rich or poor, cold or hot etc.) Regression algorithms outputs are related to figures or real values (for example amount, measures, etc.).

2.4 Linear Regression

Linear regression (LR) is a statistical method for finding the relationship between independent and dependent variables. Linear Regression is generally classified into two types: Simple Linear Regression and Multiple Linear Regression.

2.4.1 Simple Linear Regression

Simple Linear Regression is used to find the relationship between two variables, an independent variable and it corresponding dependent variable. An example might be a person's age and their average height. The age represents the independent variable while the height is the dependent variable.

A pair consisting of an independent variable and its dependent variable is referred to as an observation. Simple linear regression uses the equation of a straight line to evaluate the relationship between the variables of an observation. An equation that approximates the relationship between the variables of an observation is said to be a simple model. A simple model can be used to find or predict the dependent variable given the independent variable.

In some instances, there can be more than one independent variables. In such cases a different model known as multiple regression is used instead.

2.4.2 Simple Linear Regression Model

The following is an example of a simple linear regression model:

$y = \beta 0 + \beta 1x + \varepsilon$

The linear regression model contains an error term that is represented by ε. The error term is used to account for the variability in y that cannot be explained by the linear relationship between x and y. If ε were not present, that would mean that knowing x would provide enough information to determine the value of y.

There also parameters that represent the population being studied. These parameters of the model are represented by β0 and β1.

The simple linear regression equation is graphed as a straight line, where:

- β0 is the y-intercept of the regression line.
- β1 is the slope.
- E(y) is the mean or expected value of y for a given value of x.

A regression line can show a positive linear relationship, a negative linear relationship, or no relationship at all.

- No relationship: The graphed line in a simple linear regression is flat (not sloped). There is no relationship between the two variables.
- Positive relationship: The regression line slopes upward with the lower end of the line at the y-intercept (axis) of the graph and the upper end of the line extending upward into the graph field, away from the x-intercept (axis). There is a positive linear relationship between the two variables: as the value of one increases, the value of the other also increases.
- Negative relationship: The regression line slopes downward with the upper end of the line at the y-intercept (axis) of the graph and the lower end of the line extending downward into the graph field, toward the x-intercept (axis). There is a negative linear relationship between the two variables: as the value of one increases, the value of the other decreases.

2.4.3 The Estimated Linear Regression Equation

If the parameters of the population were known, the simple linear regression equation (shown below) could be used to compute the mean value of y for a known value of x.

$E(y) = \beta0 + \beta1x + \varepsilon$

In practice, however, parameter values generally are not known so they must be estimated by using data from a sample of the population. The population parameters are estimated by using sample statistics. The sample statistics are represented by β0 and β1. When the sample statistics are substituted for the population parameters, the estimated regression equation is formed.

The estimated regression equation is:

$(\hat{y}) = \beta0 + \beta1x + \varepsilon$

The graph of the estimated simple regression equation is called the estimated regression line.

- β0 is the y-intercept of the regression line.
- β1 is the slope.
- $(\hat{y})$ is the estimated value of y for a given value of x.

2.4.4 Limitations of Simple Linear Regression

Even the best data does not tell a complete story.

Regression analysis is commonly used in research to establish that a correlation exists between variables. But correlation is not the same as causation: a relationship between two variables does not mean one causes the other to happen. Even a line in a simple linear regression that fits the data points well may not guarantee a cause-and-effect relationship.

Using a linear regression model will allow you to discover whether a relationship between variables exists at all. To understand exactly what that relationship is, and whether one variable causes another, you will need additional research and statistical analysis.

2.5 Neural Networks

An artificial neural network is a network of artificial neurons or nodes connected together by weighted connections meant to simulate the synapses of the brain . The first research in artificial neural networks was the perceptron model developed in the 1950s and 1960s by the Frank Rosenblatt.[30]

In general, neural networks are a form of deep learning that maps inputs to outputs and finds correlations between them. It is known as a "universal approximator" because it can learn to approximate an unknown function f(x) = y between any input x and any output y, assuming they are related at all (by correlation or causation, for example). The "universal approximation theorem claims that the standard multilayer feed-forward networks with a single hidden layer that contains finite number of hidden neurons, and with arbitrary activation function2 are universal approximators in C(Rm)" [28]. These artificial networks may be used for predictive modeling, adaptive control and applications where they can be trained using a dataset. Self-learning resulting from experience can occur within networks, which can derive conclusions from a complex and seemingly unrelated set of information.

In most ANN, input signals to neural units are based on the following function:

$net_Y = \sum_i(w_i * x_i) + b$

In the above function, $x_1,x_2....x_n$ are input variables. $w_1,w_2....w_n$ are weights on the connections from those respective inputs. b is the bias, which is summed with the weighted inputs to form the net input net_Y. Bias and weights are both adjustable parameters of the neuron. The goal of learning is to adjust these parameters to produce the output y that corresponds to the desired value of y.

The output of a neuron can range from -infinity to +infinity, but is usually defined by it activation function, a mapping of the net input net_Y to the output value Y. The table below gives some common activation functions.

Table 1: Activation functions of ANN

Activation function	Operation
Identity function	Maps input to the same output value. It is a linear operator in vector space. Also known as straight line function where activation is proportional to the input.
Binary Step Function	Uses threshold to classify output into either true or false. It is very useful for classifiers.
Sigmoid Function	Approximates a binary step function, but has a smooth derivative. Two common sigmoid functions are the Binary Sigmoid Function where the output varies from 0 to 1, and the Bipolar Sigmoid Function where the output value varies from -1 to 1. Also known as Hyperbolic Tangent Function or tanh.
Ramp Function:	Maps negative inputs to 0 and positive inputs to the same output
ReLu	Stands for the rectified linear unit (ReLU). The most widely use function that solves

	and replaces the problematic sigmoid function.

A simple neural network model is adapted and used to predict PRNG in this research.

2.6 Reinforcement Learning

Reinforcement learning (RL) is version of machine learning that is motivated by a behaviorist psychology theory commonly referred to as “rewards and punishment”.

In his attempt to describe RL[26] explained reinforcement learning to address the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals. The reinforcement learning model (otherwise known as an agent) is made to interact with its environment with the attempt to solve a given problem in a way that can be best described as “trial and error”. The feedback from it actions is used to evaluate if it has made an error or not and then to reward or punish accordingly.

An agent in machine learning could be implemented in a self-driving car or a program playing chess that interacts with its environment, receiving a reward state depending on how it performs, such as driving to destination safely or winning a game. Conversely, the agent receives a penalty for performing incorrectly, such as going out of lane or off the road or being checkmated.

In an attempt to compare reinforcement learning to supervised and unsupervised learning, Shweta Bhatt of Youplus, in an online newsletter wrote: [35] “Though both supervised and reinforcement learning use mapping between input and output, unlike supervised learning where feedback provided to the agent is correct set of actions for performing a task, reinforcement learning uses rewards and punishment as signals for positive and negative behavior."

In reinforcement learning the goal is to find a suitable action model that would maximize the total cumulative reward of the agent.

Reinforcement learning can be implemented with several algorithms. [7] listed 29 different RL algorithms in an online article titled "reinforcement learning algorithms quick overview".

The popular ones among them include Q-learning, SARSA(State-Action-Reward-State-Action), Deep-Q-Networks, DDPG (Deep Deterministic Policy Gradient) and Monte Carlo.

These algorithms share some similarities mostly in the way they simulate the dynamics of their environment (model) and how they exploit it. Q-learning and SARSA are model-free and commonly used. Each of the algorithms use one of the three attributes for optimization. either model, value or policy.

Some of the algorithms are simple to implement like Q-learning and SARSA but they lack generality because they lack the ability to estimate values for unseen states.

Advanced algorithms such as Deep Q-Networks and DDPG (Deep Deterministic Policy Gradient) use Neural Networks to estimate Q-values. However, DQNs can only handle discrete, low-dimensional action spaces.

Table 2: List of learning Algorithms and their properties

Algorithm	**Description**	**Model**	**Policy**	**Action Space**	**State Space**	**Operator**
Monte Carlo	Every visit to Monte Carlo	Model-Free	Off-policy	Discrete	Discrete	Sample-means
Q-learning	State–action–reward–state	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA	State–action–reward–state–action	Model-Free	On-policy	Discrete	Discrete	Q-value
Q-learning – Lambda	State–action–reward–state with eligibility traces	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA – Lambda	State–action–reward–state–action with eligibility traces	Model-Free	On-policy	Discrete	Discrete	Q-value

DQN	Deep Q Network	Model-Free	Off-policy	Discrete	Continuous	Q-value
DDPG	Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
A3C	Asynchronous Advantage Actor-Critic Algorithm	Model-Free	On-policy	Continuous	Continuous	Advantage
NAF	Q-Learning with Normalized Advantage Functions	Model-Free	Off-policy	Continuous	Continuous	Advantage
TRPO	Trust Region Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
PPO	Proximal Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage

TD3	Twin Delayed Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
SAC	Soft Actor-Critic	Model-Free	Off-policy	Continuous	Continuous	Advantage

Reinforcement learning seems like a great idea, but faces challenges. "The excitement and PR hype behind reinforcement learning is a bit disproportionate relative to the economic value it is creating today. It turns out that reinforcement learning is a type of machine learning whose hunger for data is even greater than supervised learning. It is difficult to get enough data for reinforcement learning algorithms. There's more work to be done to translate this to business and practice" (Andrew Ng, cochairman and cofounder, Coursera) [1]

While reinforcement learning was not used for the research in this thesis, it might be a good approach for further research on this topic.

2.7 Previous Work in this Area

There has been extensive publications and research work on pseudo random numbers, but most focus on either evaluating, analyzing, or modifying the popular random number generator engines like the linear congruential algorithm and the Mersenne twister. However, two publications were found to be somewhat related to this work. F. Fan et al [36] used a simple neural network structure and a typical training procedure to demonstrate the learning and prediction power of the neural network in simple pseudo-random environments. Petr Savicky et al [35] also describe how dependencies between random numbers generated with some popular pseudo random number generators can be detected using general purpose machine-learning techniques. However, neither experiments with the PRNGs used in actual programming languages, nor do they use the same learning algorithms explored in this paper.

3.0 Methodology

In this research, simple linear regression and neural network learning is used to evaluate the predictability of a set of commonly used pseudorandom number generators (PRNG). This involved generation of observations for each PRNG (broken into training and testing datasets), training the model with the training dataset, and evaluating of the model with the testing datasets.

3.1 Generation of Observations

The PRNG is first used to generate a set of observations, a pair of a dependent and independent variables. For a PRNG, this will be a seed and a random number. A total of 2000 observations are generated for each of the selected PRNG. The first 1400 (70% of the total generated observations) were used as the training set, 15% were used for validating and the remaining 15% for testing the model. In the rattle application, the validation process which precedes the testing process uses a fraction of the dataset allocated for validation to evaluate the model parameters whilst it is still being tuned but not for the final unbiased estimate of error. Contrary to testing that evaluates the performance of the model over the testing data set, which is the remaining of the dataset not used for building and so will provide an unbiased estimate.

3.2 Training of the Model with The Datasets

The two machine learning models used for evaluating the PRNG in this research are the linear regression model and the neural network model. The two models are integrated into Rattle of the R programming language. Rattle is a free graphical user interface for Data Science, developed using R. R is a free software environment for statistical computing, graphics, machine learning and artificial intelligence. Together Rattle and R provide a sophisticated environment for data science, statistical analyses, and data visualization. Rattle version 5.3.0 was used, making it easy to train the models with different datasets validate and test them. The behavior of the models is explained in the following paragraphs.

3.2.1 Linear Regression Model

The training for simple linear regression is done by first studying similar pairs of variables in order to identify a relationship between them. The algorithm starts by assuming a relation between the pairs of variables (x and y) as shown below:

P(Y) = B0 + B1*X

Where:

- P(Y) = the predicted Y,
- B0 = first constant,
- B1 = second constant,
- X = actual x variables.

B0 and B1 represent constants in our function, and the objective of the algorithm is to adjust B0 and B1 such that they can eventually approximate a relationship between the X and Y variables. The algorithm starts the training process with B0= 0 and B1= 0, and adjusts them after every iteration during the training. The adjustment continues throughout the entire training process of the algorithm until the end. In Rattle the parameters that minimize the squared error loss are not estimated using gradient descent which depends on epochs to get better, they are computed exactly.

We illustrate the training process by walking through the sample dataset as in the Table 2 as follows;

Table 3: A sample data set for explain the training process of linear regression.

X	Y
0	1
1	5
2	9
4	13
6	17
8	21
10	25

From the dataset above, we first plug in the x variables to make a prediction for y. Initially B0 = 0 and B1 = 0, so we get: Y = B0 +B1 *X = 0 + 0*0 = 0

The algorithm compares this prediction to the actual values in the training set to calculate the error:

error (e) = Prediction(P(Y)) - actual Y from our training dataset (Y(i))

For the first pair in the training set, we have error () = 0-1 = - 1.

We must update/adjust the constants B0 and B1 in our function in order to reduce the error. This adjustment is done in small increments, usually a step size between 0 and 1. This is generally referred to as the learning rate. For this demonstration 0.01 is used as the learning rate.

The formula for adjusting B0 is as follows:

B0(t+1) = B0(t) – r*error

For this example that gives:

B0(t+1)=0-0.01*(-1)= 0.01

Since B1 directly interacts with the x variable its formula is:

*B1(t+1) = B1(t) – r * error * x*

For this example we get:

*B1(t+1) = 0 – 0.01 *(-1) * 0 = 0*

So after the example the constants B0 and B1 will be updated as *B0 = 0.01 and B1 = 0*

This process must be iterated for all the (x, y) observations in the training dataset, updating the constants each time. A complete iteration of all the observations in the training dataset is known as a pass or an epoch. Normally, several epochs are needed to reduce the error in attempt to improve the accuracy of the model's prediction. In rattle the iterations are experimental.

3.2.2 Neural Network Model

Artificial neural networks can be constructed to have similar principles as linear regression as demonstrated in Figure 1. In this model, X is an input. B1 is the weight on the connection from input X and B0 is the weight on the connection from the bias unit (always 1), which gives the output a value of B0 + B1X.

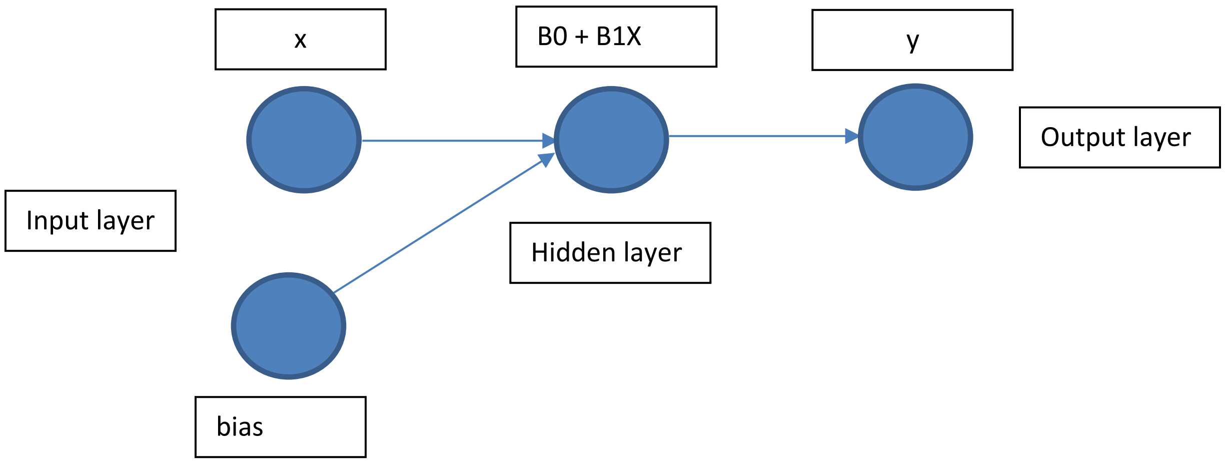


Figure 1. Comparing linear regression to neural network

Another thing neural networks have in common with linear regression model is how they calculate errors and adjust their weights (that is, change B0 and B1 to reduce error). The Rattle package uses the backpropagation learning algorithm, the most used supervised learning algorithm for feedforward neural networks.

A neural network was used in this experiment because relation between the random numbers and their seed might not necessary be a linear relationship. In such a case, it cannot be represented as a linear equation of the form B0 + B1X, and linear regression would not be effective. On the other hand, a neural network with at least one layer of hidden units can represent nonlinear relationships, and capable of learning them with algorithms such as backpropagation.

The Figure 2 represents the structure of the neural network used for in this work, with a single layer of hidden units containing 10 units. However, neural networks can have many more connections and weights, and can be even more complex with many hidden layers in the case of deep neural networks.

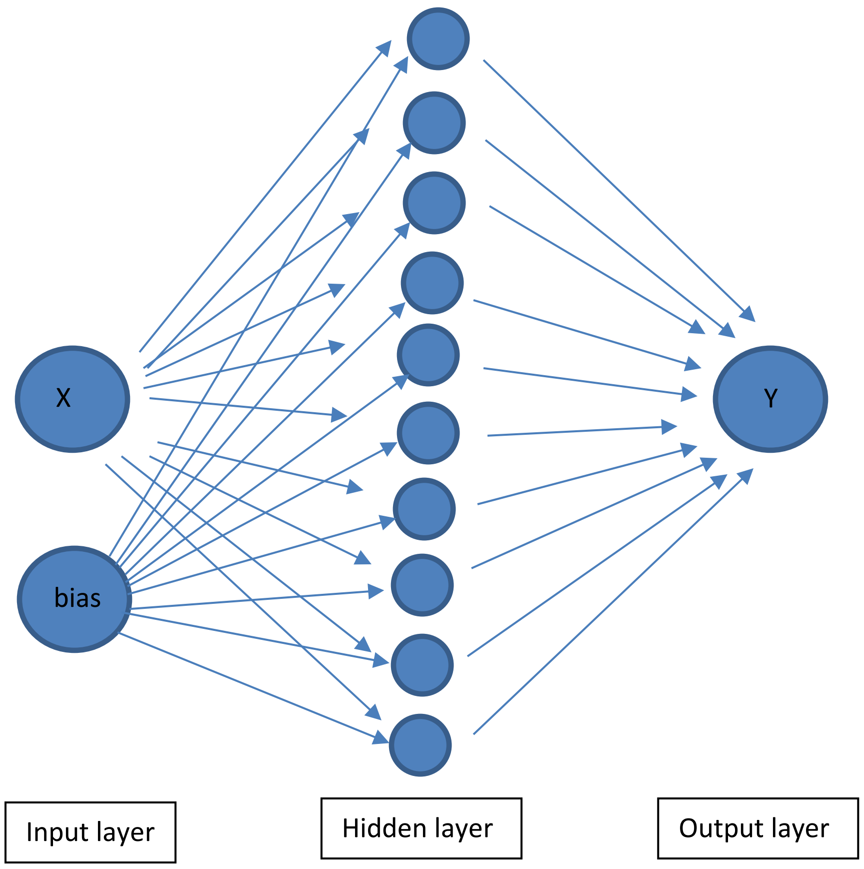
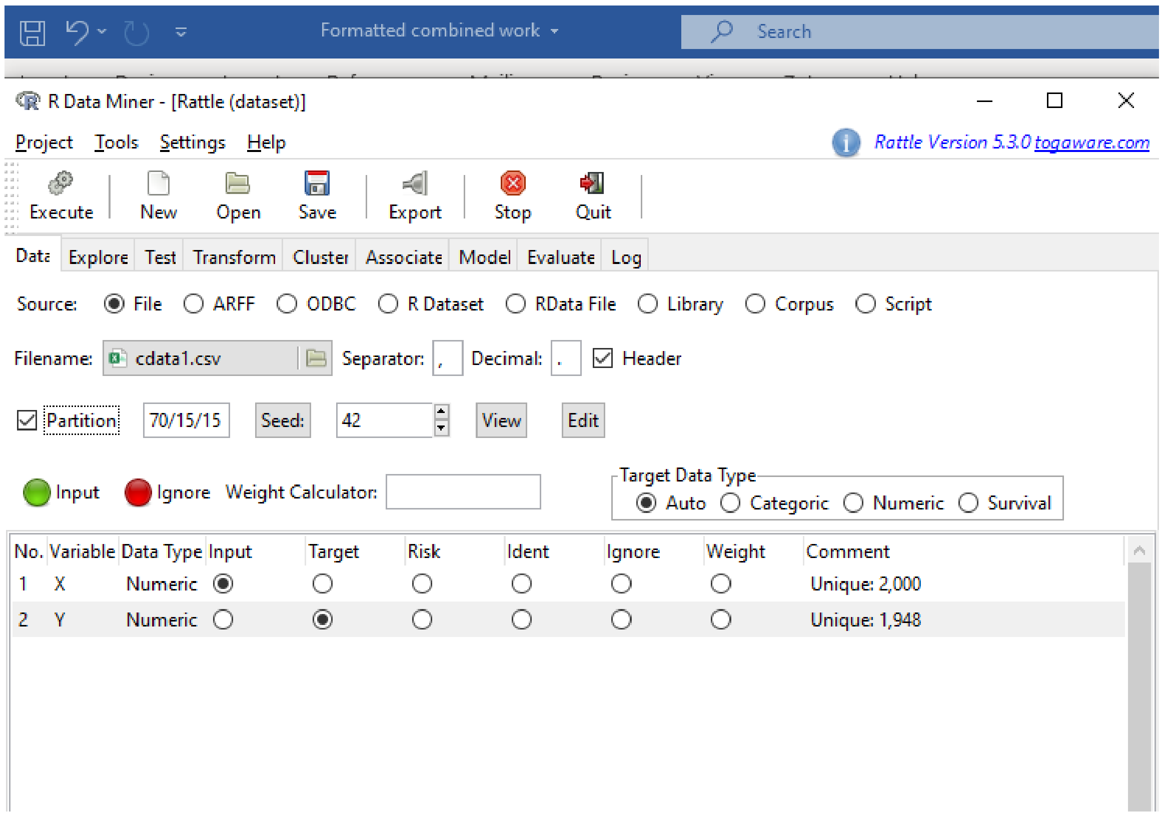


Figure 2. Structure of neural network used in this experiment.

3.2.3 The Rattle Application

The NNM of rattle presents a GUI from which optional features can be selected. The interface in Figure 2. Shows the available options that can be made to the model. A function to calculate the weight can be typed in weight calculation text box. A default partition of 70/15/15 meaning 70% of the dataset be use for training, 15% for validation and 15% for testing. of the dataset to be used for training, validation, and testing. nodes for the hidden layers. A default seed is set in other to keep the model consistent when it is train multiple times.


Formatted combined work
Search
R Data Miner - [Rattle (dataset)]
Project Tools Settings Help
Rattle Version 5.3.0 togaware.com
Execute New Open Save Export Stop Quit
Data Explore Test Transform Cluster Associate Model Evaluate Log

Source: [✓ File] ARFF ODBC R Dataset RData File Library Corpus Script

Filename: cdata1.csv Separator: , Decimal: . [✓ Header]

[✓ Partition] 70/15/15 Seed: 42 View Edit

Input Ignore Weight Calculator:

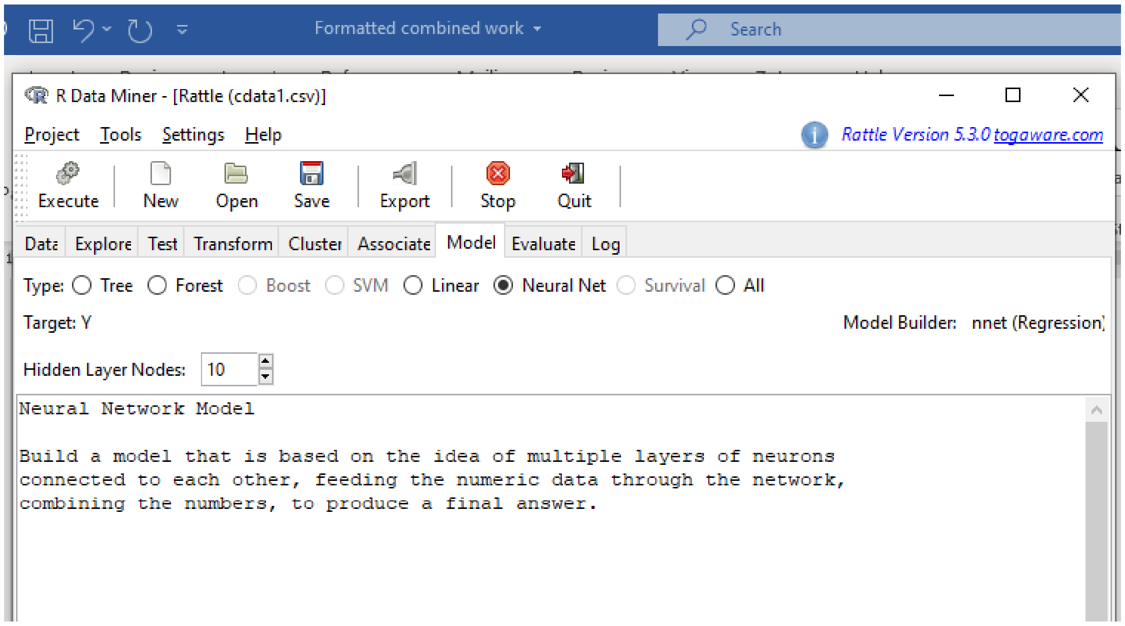
Target Data Type: [✓ Auto] Categoric Numeric Survival

No.	Variable	Data Type	Input	Target	Risk	Ident	Ignore	Weight	Comment
1	X	Numeric	●	○	○	○	○	○	Unique: 2,000
2	Y	Numeric	○	●	○	○	○	○	Unique: 1,948

Figure 3. Data tab of Rattle

The model tab as shown in figure 3. Gives the option of the type of model to be trained. The selection of a model depends on the kind of dataset input at the data tab. Only models that can be train on the dataset will be activated for selection. There is also the option to select “All” which will train the dataset with all activated models. Another option is to select an integer for the node . this set the number of nodes the hidden layers will have in the model. For this experiment 10 was selected because the dataset is small.

Other features including bias, epochs, and activation functions are experimental to the application. In the summary of training the model which can be seen at appendixes 14 to 24, it show that the activation functions used by rattle in building the models are skip-layer connections and linear output units.


Formatted combined work
Search
R Data Miner - [Rattle (cdata1.csv)]
Project Tools Settings Help
Rattle Version 5.3.0 togaware.com
Execute New Open Save Export Stop Quit
Data Explore Test Transform Cluster Associate Model Evaluate Log

Type: Tree Forest Boost SVM Linear [✓ Neural Net] Survival All

Target: Y

Model Builder: nnet (Regression)

Hidden Layer Nodes: 10

```
Neural Network Model

Build a model that is based on the idea of multiple layers of neurons
connected to each other, feeding the numeric data through the network,
combining the numbers, to produce a final answer.
```

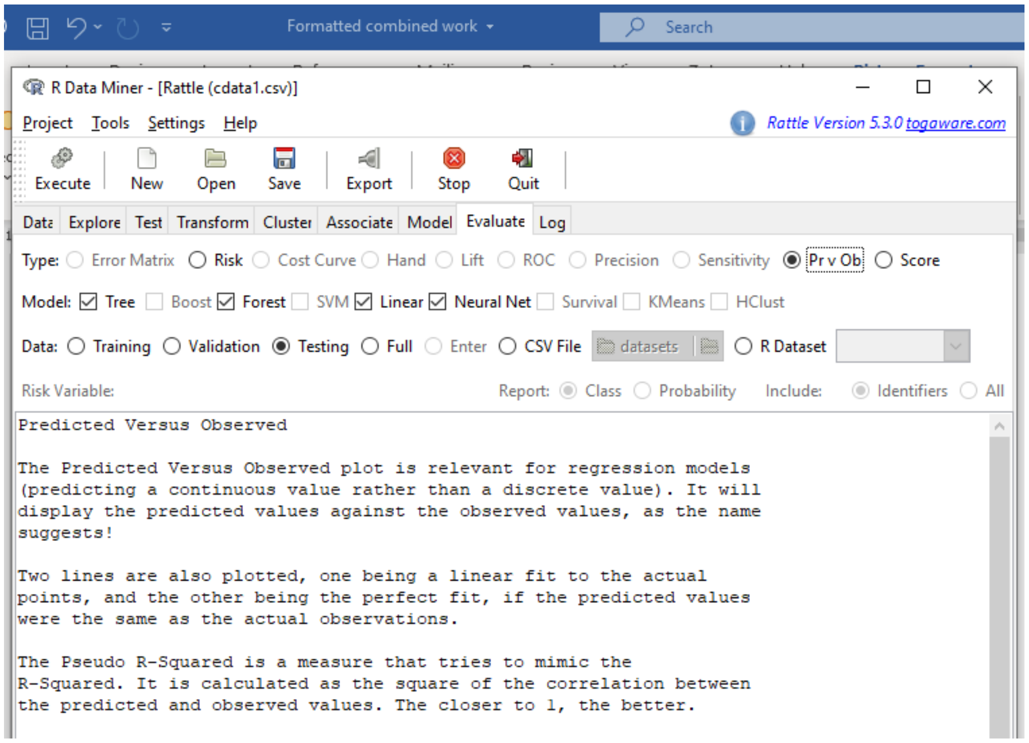
Figure 4. The rattle model tab.

3.3 Model Testing

After a successful training of the algorithm, there is the need to test the model.

The experiment with the models on all the six generated datasets is implemented in chapter four and the performance of the models are evaluated and analyzed with linear fits and R-square. This comes in a function in the rattle package and it presents the results in a “Predicted Versus Observed” plot. This graph is relevant for regression models because most the predictions are a continuous value rather than a discrete value. Figure 5 shows the evaluation tab as used in rattle.

More details about the evaluation is in chapter five.


Formatted combined work
Search
R Data Miner - [Rattle (cdata1.csv)]
Project Tools Settings Help
Rattle Version 5.3.0 togaware.com
Execute New Open Save Export Stop Quit
Data Explore Test Transform Cluster Associate Model Evaluate Log

Type: Error Matrix, Risk, Cost Curve, Hand, Lift, ROC, Precision, Sensitivity, [✓ Pr v Ob], Score

Model: [✓ Tree] Boost [✓ Forest] SVM [✓ Linear] [✓ Neural Net] Survival KMeans HClust

Data: Training, Validation, [✓ Testing], Full, Enter, CSV File, R Dataset

Risk Variable: Report: Class Probability Include: Identifiers All

```
Predicted Versus Observed

The Predicted Versus Observed plot is relevant for regression models
(predicting a continuous value rather than a discrete value). It will
display the predicted values against the observed values, as the name
suggests!

Two lines are also plotted, one being a linear fit to the actual
points, and the other being the perfect fit, if the predicted values
were the same as the actual observations.

The Pseudo R-Squared is a measure that tries to mimic the
R-Squared. It is calculated as the square of the correlation between
the predicted and observed values. The closer to 1, the better.
```

Figure 5. Evaluation tab in rattle.

4 Experimentation with Existing PRNGS

Three major PRNGs were considered in these experiments, those commonly used in the programming languages C++, Java and Python. Since most of these languages have more than one method and classes for generating pseudorandom numbers, the PRNG algorithm chosen in each language based on whether it returned at least two variables for use as an observation consisting of a dependent and an independent variable. As described in the previous chapter, in most cases the independent variable is the seed of the random number generator.

The procedure described in the previous chapter was used to evaluate each of the three PRNG in this section. Each part of the section follows the procedure to evaluate one of the selected PRNGs.

The datasets were analyzed for outliers, skewness, distribution, and correlation before training them on the models. To explore the datasets and possibly find any interesting relationship with the outcome. Correlation between the seed and the random numbers is very important to this experiment and the Pearson correlation test was used. This statistical analysis is necessary to identify any relationship between statistical properties and training outcomes. This however does not dispute the fact that correlation is not necessarily causative.

4.1 Part I: Analysis of C++ rand() and srand() datasets

In C++ the functions rand() and srand() were selected and used together to generate random numbers. The srand function was used to generate the seeds (independent variables) for the random number generation, and the rand() function was used to generate the actual random numbers (dependent variables). See Appendixes 7 and 8 for the C++ code used to generate the observations.

The code in Appendix 7 was used to generate 2000 random numbers based on the same seed 1 (that is, srand(1) was used in all cases), and the code in Appendix 8 was used to generate 2000 random numbers with each based on a different seed in the range 1 - 2000. The outputs are formatted into (X, Y) pairs, where the values of x are the seeds used as the independent variables while the values of y are the random numbers used as the dependent variables. See Appendix 1 and 2 for samples of labeled Dataset 1 and Dataset 2 respectively.

The following graphs were used for the analysis:

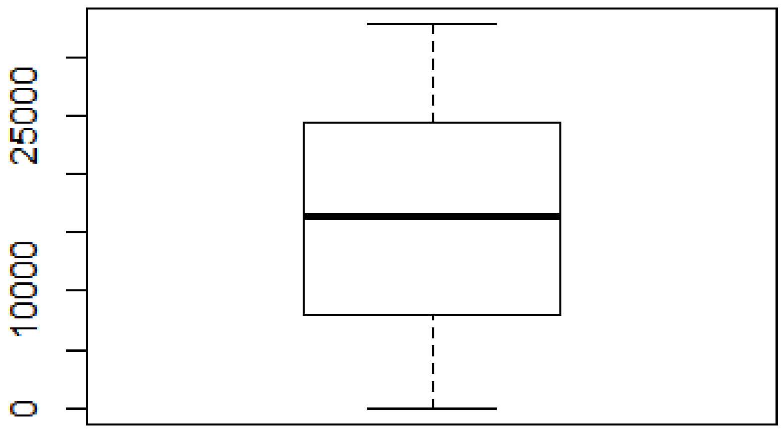


Figure 6. boxplot of dataset 1

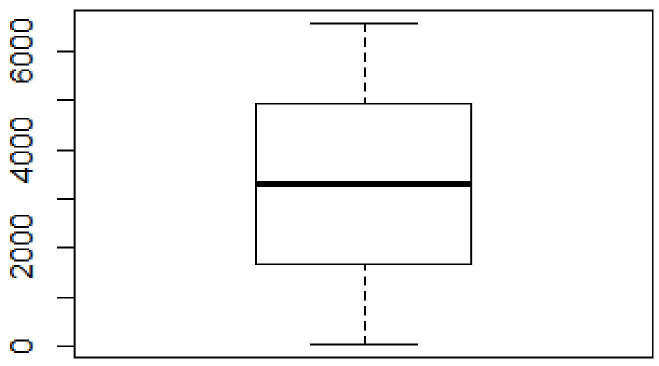


Figure 7 boxplot of dataset 2

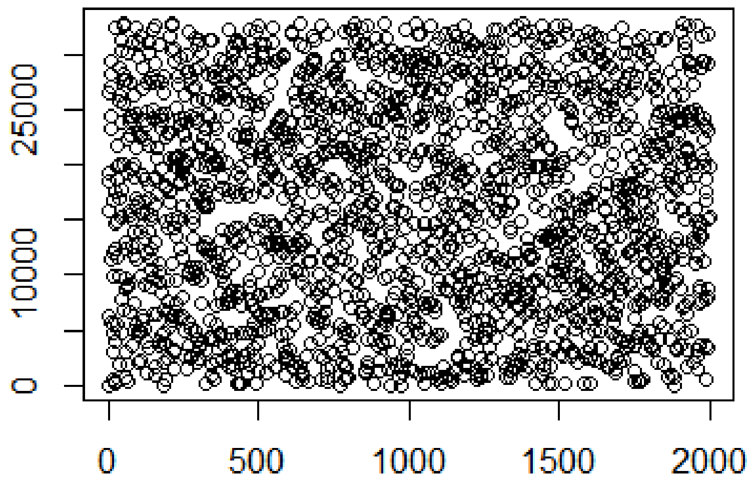


Figure 8 scatter plot of dataset 1

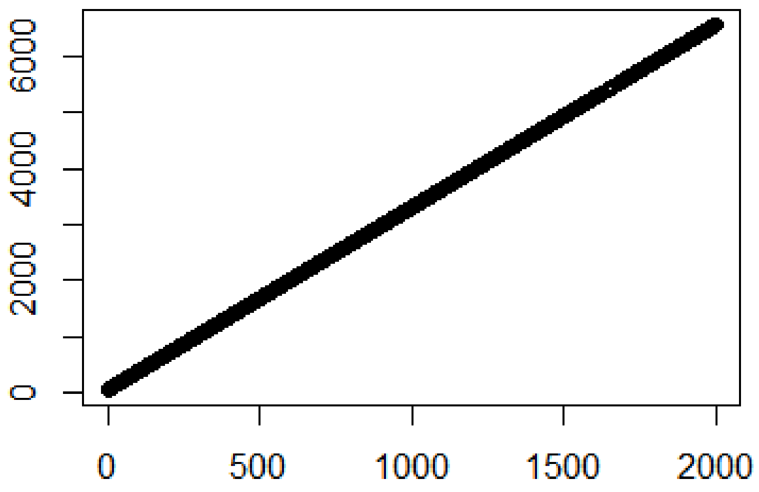


Figure 9 scatter plot of dataset 2

The boxplots in Figures 6 and 7 above show no outliers in the distribution of the random numbers in both datasets 1 and 2. The scatter graphs in Figures 8 and 9 also show an even distribution of the random numbers in both datasets; however, Figure 9 indicates a correlation between the random numbers and their seed. This is probably because the value of the seeds was directly proportional to their corresponding generated random numbers.

4.2 Part II: Analysis of Python Random() datasets

The random() and seed() methods of the Random module was used, as mentioned in Chapter two. The seed() function was used to set the seed for the basic generator and the random() function was used to return the appropriate random number. The random() method returns floating point random numbers, which were converted to 2 decimal places for convenience. Two sets of datasets labeled 'Dataset 3' and 'Dataset 4' each containing 2000 observations were generated for this section of the experiment. See Appendixes 3 and 4 for samples of Dataset 3 and Dataset 4, respectively. While all the random numbers in Dataset 3 were generated from the same seed of 1, for Dataset 4, every random number is generated with a different seed which were also in the range of 1-2000. Appendixes 9 and 10 show the Python code for generating Dataset 3 and Dataset 4, respectively.

Datasets 3 and 4 were analyzed with the following graphs:

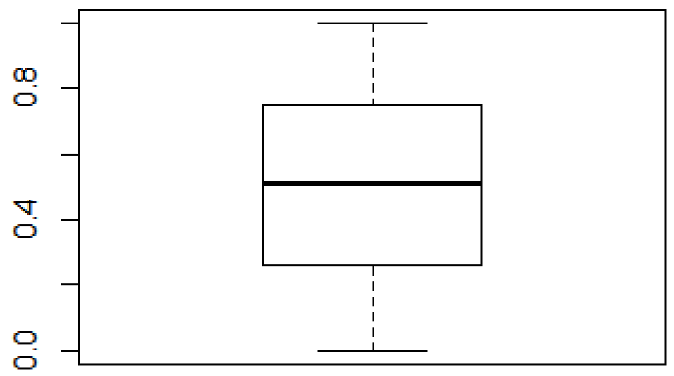


Figure 10 boxplot of dataset 3

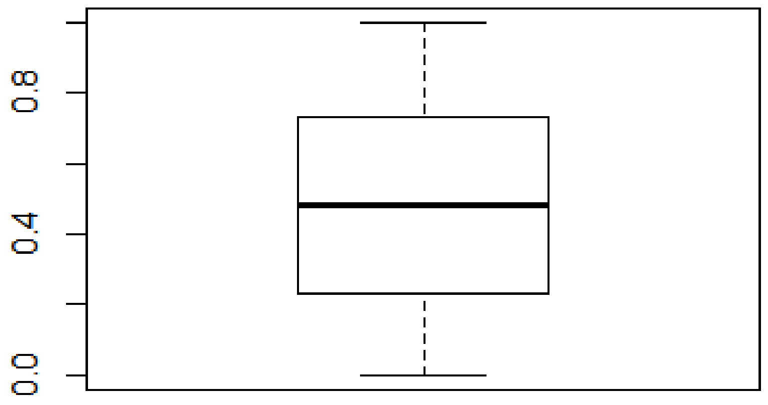


Figure 11 boxplot of dataset 4

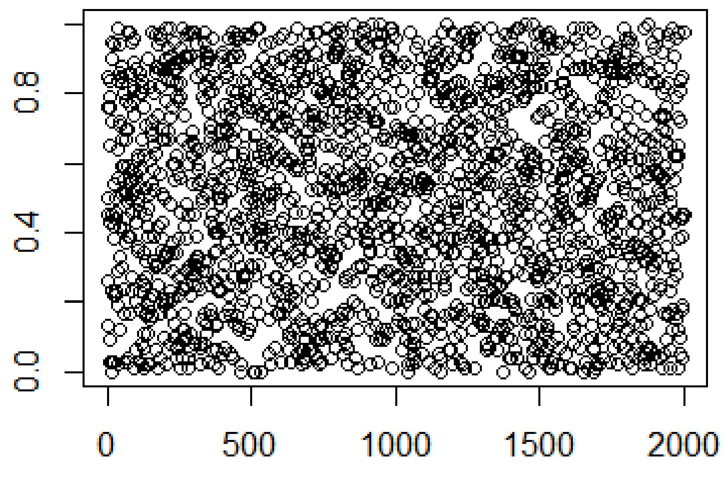


Figure 12 scattered plot of dataset 3

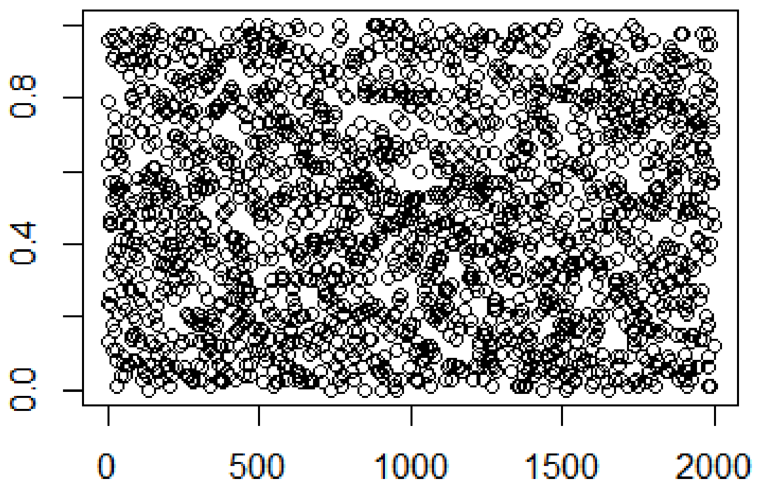


Figure 13 scattered plot of dataset 4

The boxplots in figures 10 and 11 above show no outliers in the distribution of the random numbers in both Datasets 3 and 4. The scatter graphs in figures 12 and 13 also show an even distribution of the random numbers in both datasets with no correlation between the random numbers and their seeds in both cases.

4.3 Part III: Analysis of Java java.util.random() dataset.

The random(), nextInt() and setSeed() methods were combined to generate two different sets of data, Dataset 5 and Dataset 6. The Java code for generating the two data sets is given in Appendix 11 for Dataset 5 and Appendix 12 for Dataset 6. For this section of the experiment each of the two datasets contains 2000 observations. All the random numbers in Dataset 5 were generated from the same seed of 1, while the random numbers in Dataset 6 have different seeds generated sequentially. Samples of Dataset 5 and Dataset 6 can be seen at Appendixes 5 and 6 respectively.

Datasets 5 and 6 were analyzed with the following graphs.

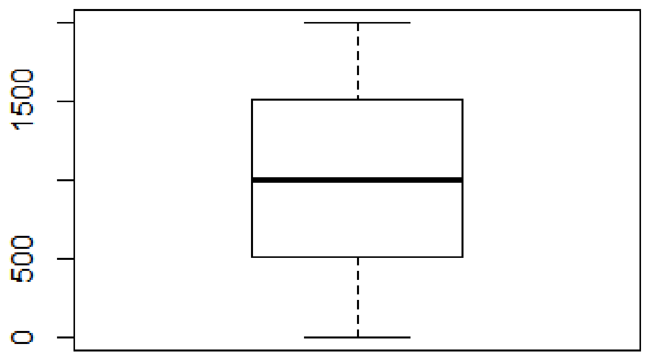
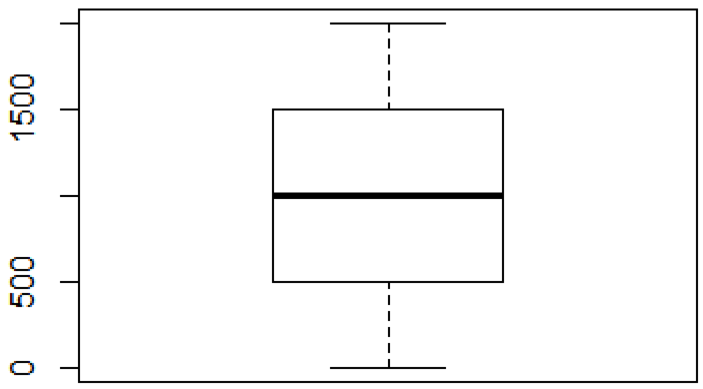


Figure 14 boxplot plot of dataset 5



.

Figure 15 boxplot plot of dataset 6

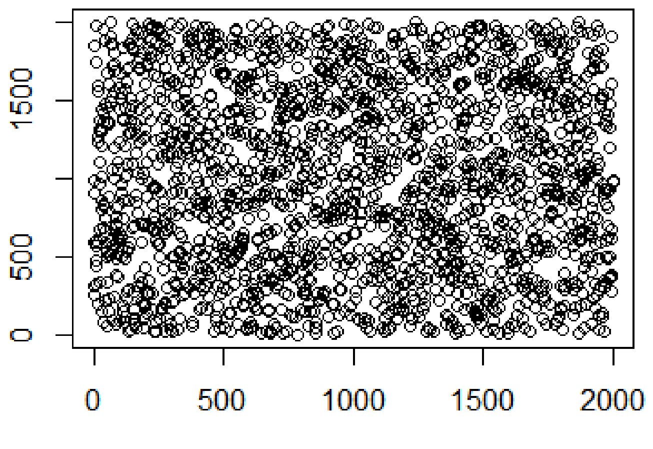


Figure 16 scattered plot of dataset 5

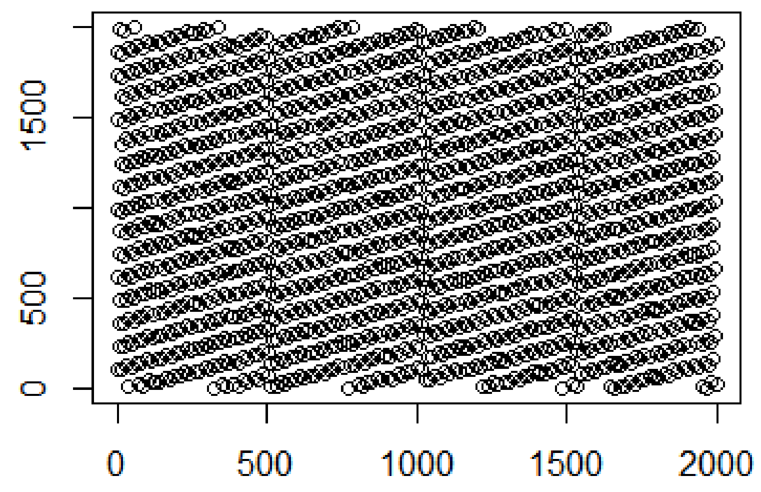


Figure 17 scattered plot of dataset 6

The boxplots in figures 14 and 15 above show no outliers in the distribution of the random numbers in both datasets 5 and 6. The scattered graphs in figures 16 and 17 also show an even distribution of the random numbers in both datasets. However, figure 17 indicates a pattern in the random number distribution which looks like a correlation between the random numbers and their seed. A closer look at dataset 6 reveals that there is an almost consistent increment and decrement over the about every 7 observations throughout the entire dataset. This could probability account for the pattern in figure 15.

4.3 Correlation Test on Data Sets

Correlation tests were used to see if there were relationships between the x and y values of the data sets used for this experiment, where the x values represent the independent variable while the y values are the dependent values. For Datasets 2, 4, and 6 the seeds were used as the x values. In Datasets 1, 3, and 5 the x value representing the sequence of the random number they correspond to. In simple terms, the x values in those datasets are just the serial numbers of the random numbers. PRNGs are highly dependent on their seed, and the hypothesis is that there is a relationship between a random number and it seed which would make it possible to perform regression functions.

A Pearson correlation test was used to all the datasets. The null hypothesis is that the two samples have no (i.e., 0) correlation. Pearson's product moment correlation coefficient was used.

If the p-value is less than 0.05 then we reject the null hypothesis and accept the alternative hypothesis that the samples are correlated, at the 95% level of confidence. The result from the correlation tests are presented in the following table;

Table 4 Correlation test result comparison.

Dataset	Correlation	P- value	R square
Dataset 1	-0.0139	0.5353	0.00019321
Dataset 2	1	< 2.2e-16	1
Dataset 3	-0.0222	0.3221	0.00049284
Dataset 4	0.0067	0.3221	0.00004489
Dataset 5	-0.0055	0.8047	0.00003025
Dataset 6	0.0019	0.9311	0.00000361

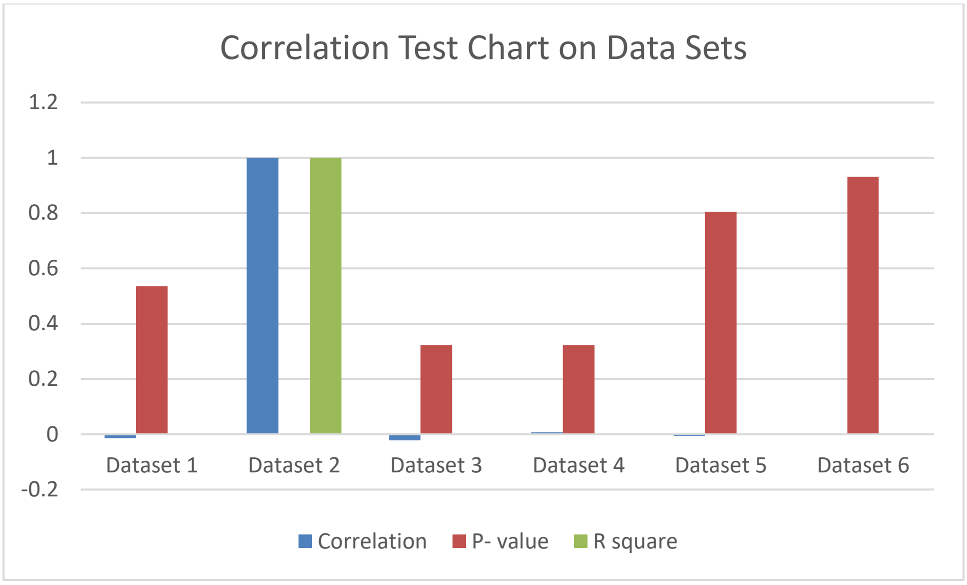


Figure 18. Correlation test chats of data sets.

We can see that only Dataset 2 shows a strong correlation between X and Y values.

Each of the six datasets were used to train the linear regression and neural network models in R.

The summary of the training on each of the data sets can be seen at appendixes 13 to 24. The models were tested and the results are presented in the chapter five.

5.0 Results and Evaluation

After training on each of the six datasets, validation was done using 15% of the dataset, the resulting models were tested for correctness using the corresponding testing sets. The outcomes of the testing are presented in graphs in this section. The section is made up of three parts, each corresponding to the three PRNGs used in this research.

The testing was done with 15% of the entire dataset for each of the six datasets, and the results of each is presented in a “Predicted Versus Observed” plot, using the R-data miner application. This graph is relevant for regression models because most the predictions are a continuous value rather than a discrete value. These graphs contain the following components:

- The plotted graphs for linear fit consist of two lines representing the actual and predicted data, where there is a perfect fit if the predicted values were the same as the actual observations.
- The Pseudo R-Squared value in each graph is a measure that mimics the standard R-Squared measure that represents the proportion of variance for a dependent variable, but is calculated as the square of the correlation between the predicted and observed values. The closer to 1 this measure is, the better the correlation.

5.1 Part 1: Test Analysis of C++ rand() and srand() PRNG.

5.1.1 Testing the Linear regression model created from Dataset 1

Figure 19 shows the results of testing the model created from Dataset 1 with the linear regression model. The R-square value of 0.002016 indicates that the predictions of the model are far from the actual observations. In other words, the model performed poorly. Figure 20, which gives the results of training the neural network shows similarly poor results. It may be the case that either the data set is too small for the models to learn from, or that there is actually no correlation between the independent and dependent variables, which was indicated in the scatter plot in Figure 5.

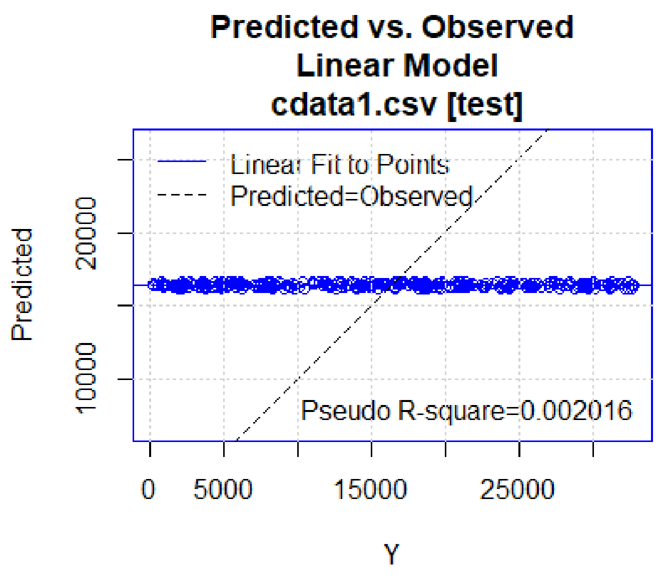


Figure 19. Predicted over Observed plot with LM for dataset 1

5.1.2 Testing the Neural Network model created from Dataset 1

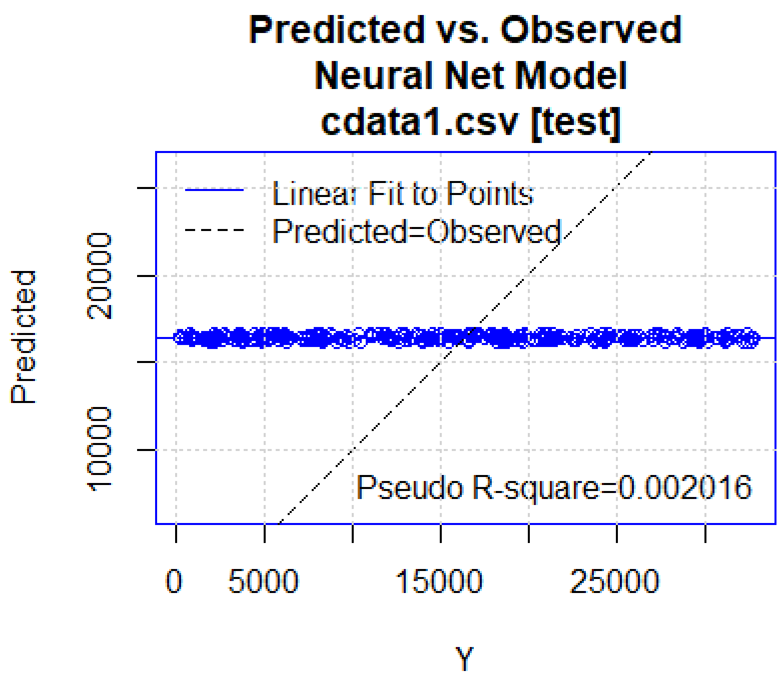


Figure 20. Predicted over Observed plot with NNM for dataset 1

5.1.3 Testing the Linear regression model created from Dataset 2

Figures 21 and 22 give test results for Dataset 2 for linear regression and neural network learning, respectively. The line showing the linear fit to the points and the line showing the mapping of the predicted values to the observed values both lie on the same plane. The R-square value of 1 indicates that both models made an accurate prediction of the observation. In Figure 4 there is a correlation between the independent and dependent variables, and that probably is the reason for the accuracy in the predictions.

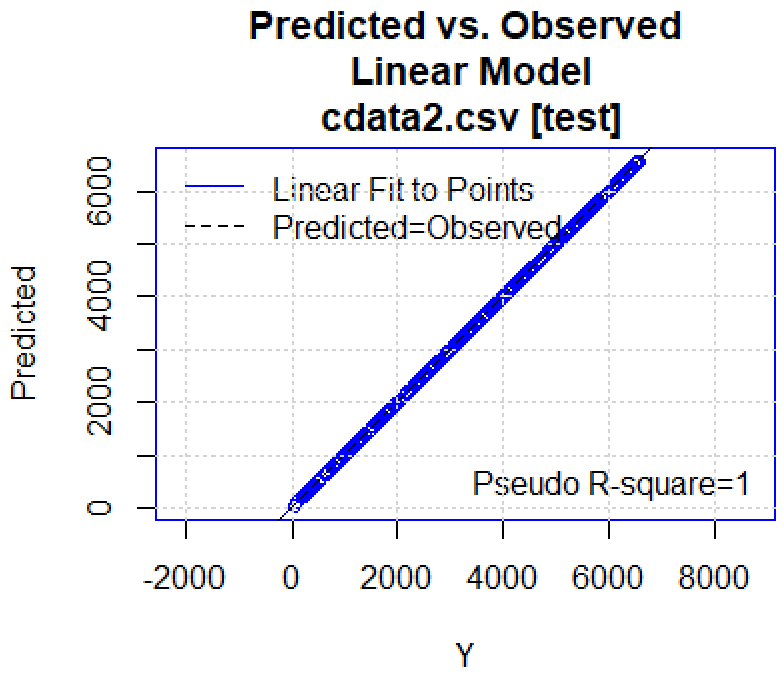


Figure 21 Predicted over Observed plot with LM for dataset 2

5.1.4 Testing the Neural Network model created from Dataset 2

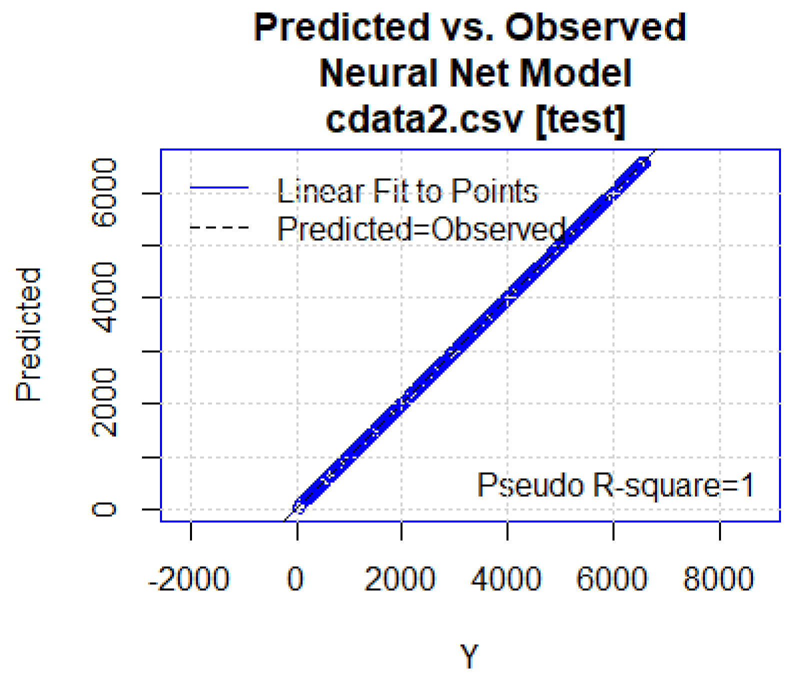


Figure 22. Predicted over Observed plot with NNM for dataset 2

5.2 Part 2: Test Analysis of Python Random() dataset.

5.2.1 Testing the Linear regression model created from Dataset 3

Figures 23 and 24 give test results for models created from Dataset 3 using linear regression and neural network learning, respectively. The R-square value of the linear model is 0.00155 and the R-square value of the neural network model is 0.0008148, indicating that they both performed poorly. However, the difference in their R-squares indicates that the LM performed better than the NNM in the prediction.

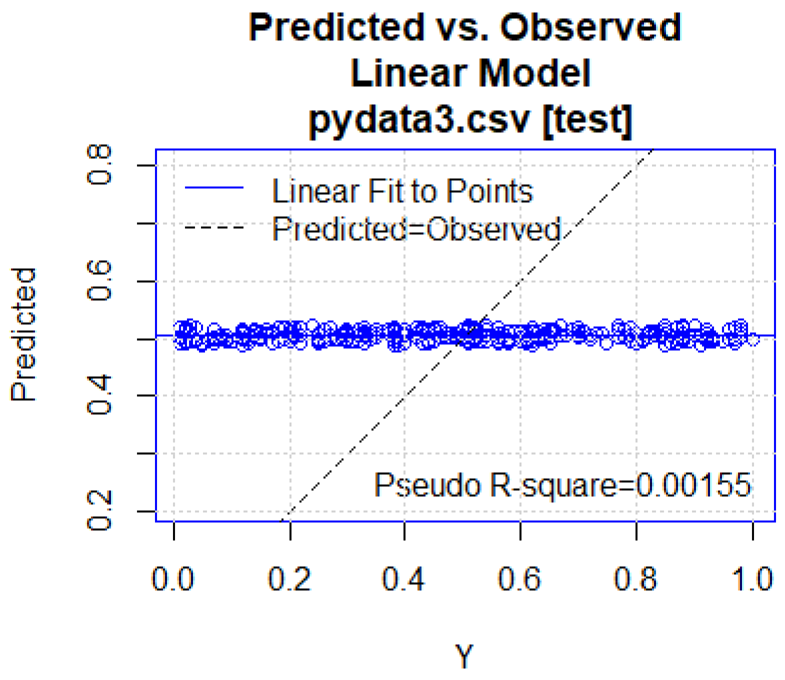


Figure 23. Predicted over Observed plot with LM for dataset 3.

5.2.2 Testing the Neural Network model created from Dataset 3

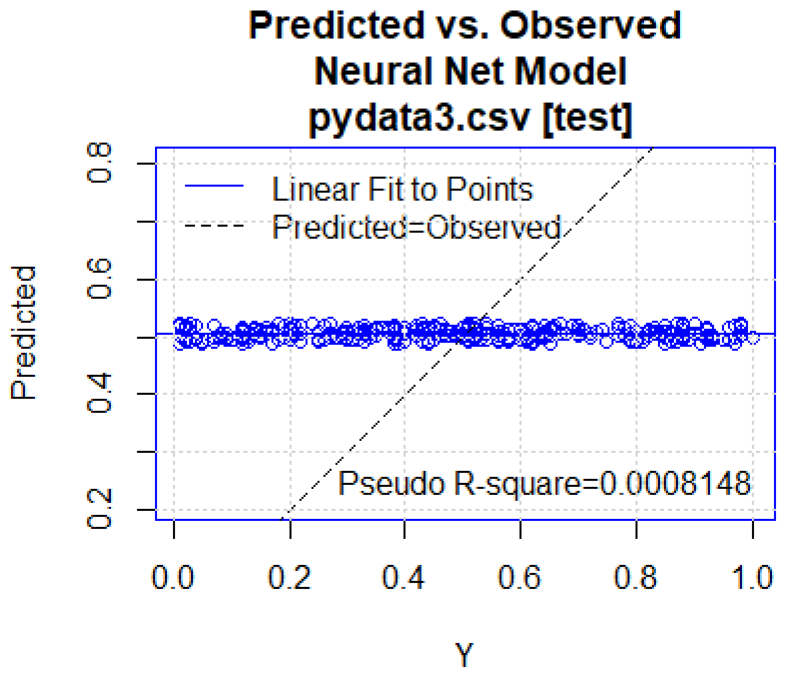


Figure 24. Predicted over Observed plot with NNM for dataset 3.

5.2.3 Testing the Linear regression model created from Dataset 4

Figures 25 and 26 give test results for models created from Dataset 4 using linear regression and neural network learning, respectively. While the R-square value of the LM is 0.0000004859, the R-square of the NNM is 0.008327. Again, both performed poorly. However, the difference in their R- square values indicates that the NNM performed much better than the LM in the predictions. This could be as a result of the complexity of the ANN, which has more parameters that linear regression. It could also be something inherent in the data, or it is an indication that the data is not enough. A larger training set might help to determine this. Finally, figure 26 shows a strange outlier in the results. It is not clear what caused that outlier.

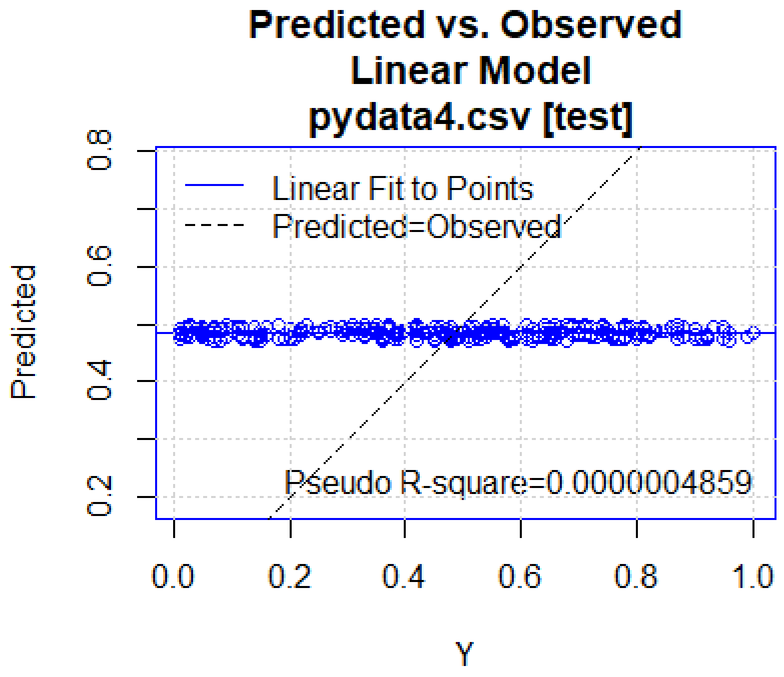


Figure 25. Predicted over Observed plot with LM for dataset 4.

5.2.4 Testing the Neural Network model created from Dataset 4

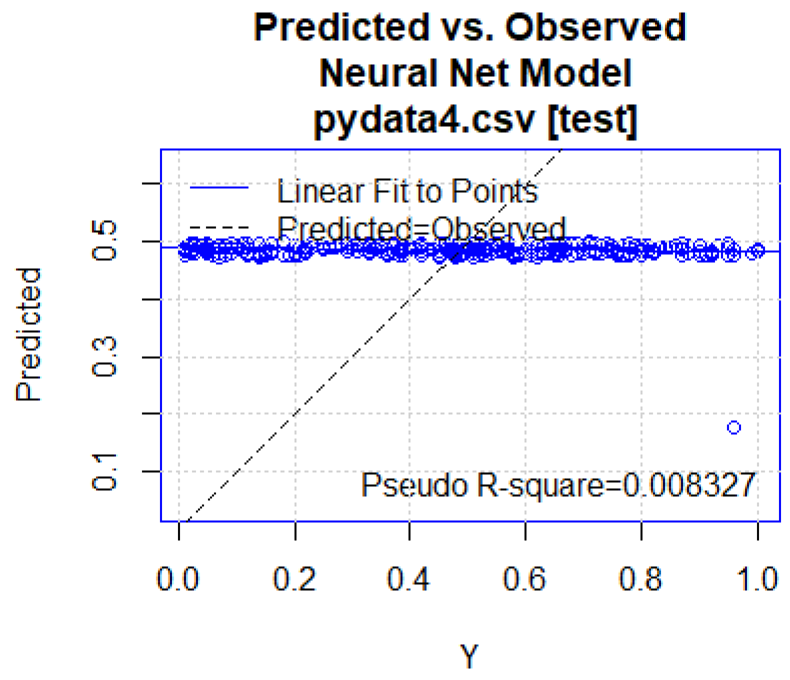


Figure 26. Predicted over Observed plot with NNM for dataset 4.

5.3 Part 3: Test Analysis Of Java java.util.random() dataset.

5.3.1 Testing the Linear regression model created from Dataset 5

Figures 27 and 28 give test results for models created from Dataset 5 using linear regression and neural network learning, respectively. Whiles The R-square value of the LM is 0.001212, the R-square of the NNM is 0.002146, showing that they both performed poorly.

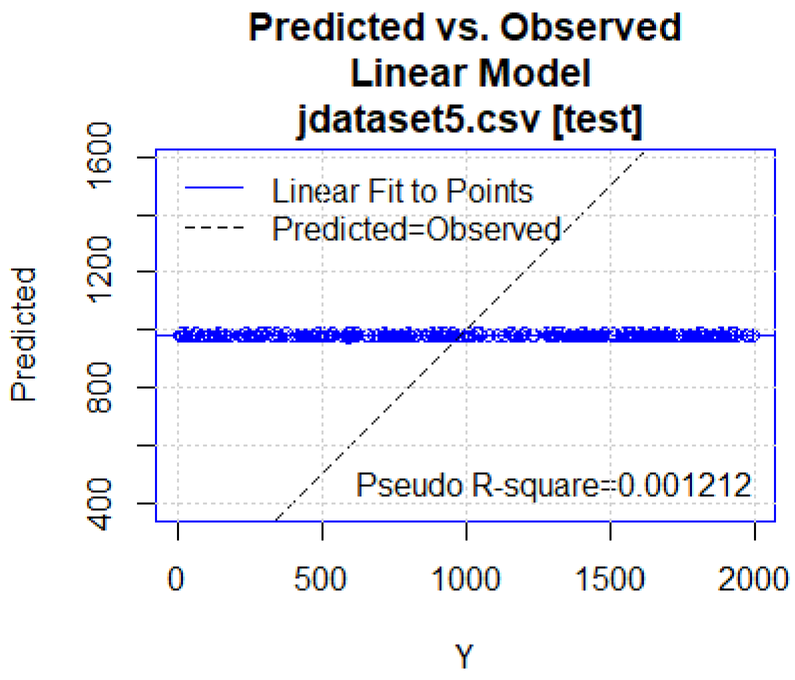


Figure 27. Predicted over Observed plot with LM for dataset 5.

5.3.2 Testing the Neural Network model created from Dataset 5

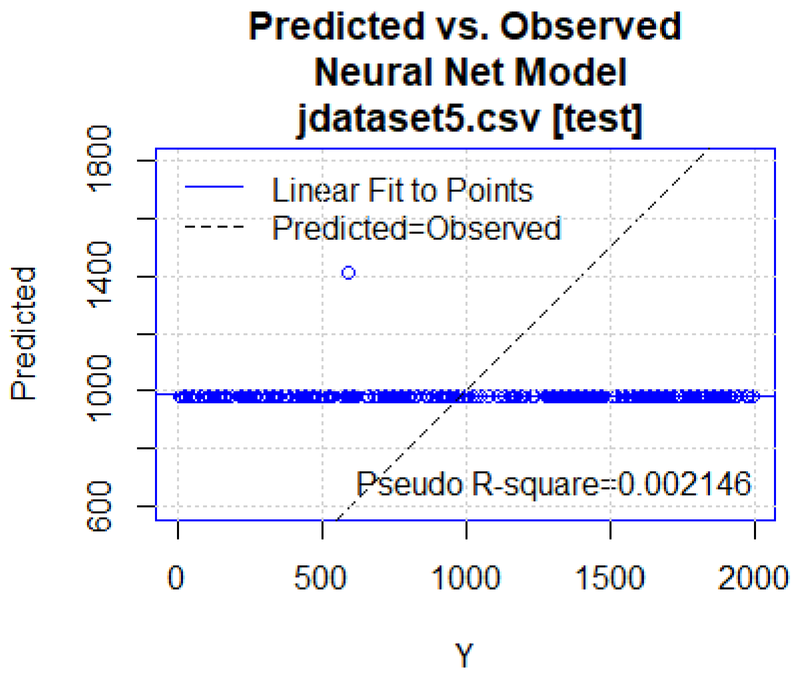


Figure 28. Predicted over Observed plot with NNM for dataset 5.

5.3.3 Testing the Linear regression model created from Dataset 6

Figures 29 and 20 give test results for models created from Dataset 6 using linear regression and neural network learning, respectively. The two models recorded the same R-square of 0.001319,

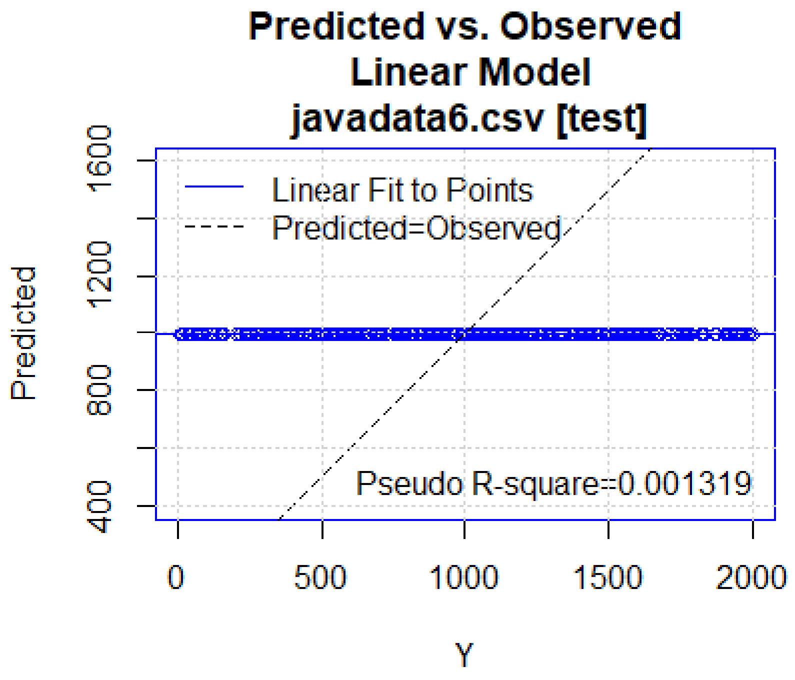


Figure 29. Predicted over Observed plot with LM for dataset 6.

5.3.4 Testing the Neural Network model created from Dataset 6

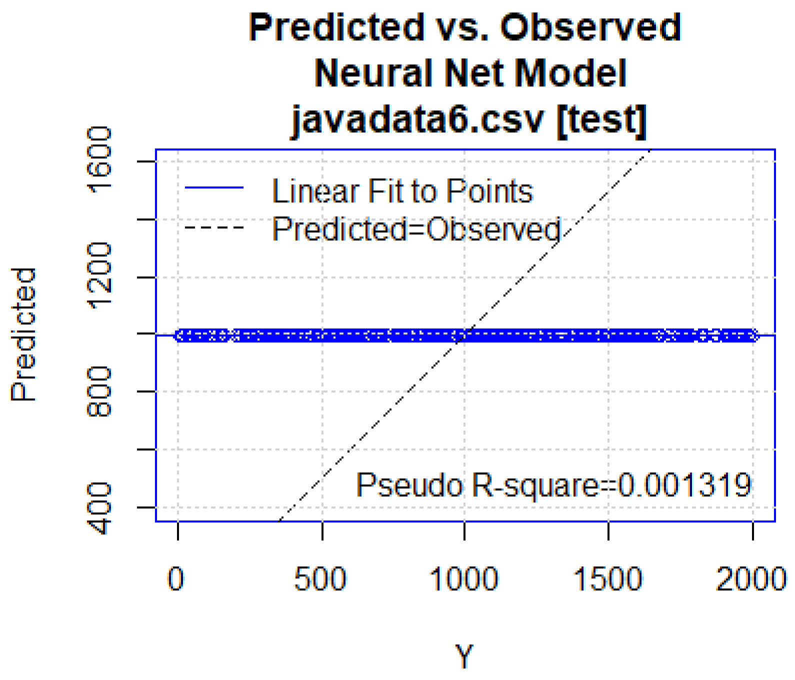


Figure 30. Predicted over Observed plot with NNM for dataset 6.

5.4 Comparing of Results Over All Datasets

In the following table the performance of the models on the various dataset is compared.

Table 5. Comparing model performance

Dataset	Programing language	Seed status	Linear model R-square	Neural Networks model R-square
Dataset 1	C++	Common	0.002016	0.002016
Dataset 2	C++	Unique	1	1
Dataset 3	Python	Common	0.00155	0.0008148
Dataset 4	Python	Unique	0.0000004859	0.008327
Dataset 5	Java	Common	0.001212	0.002146
Dataset 6	Java	Unique	0.001319	0.001319

The bar graph below compares the performance of the two models on the datasets

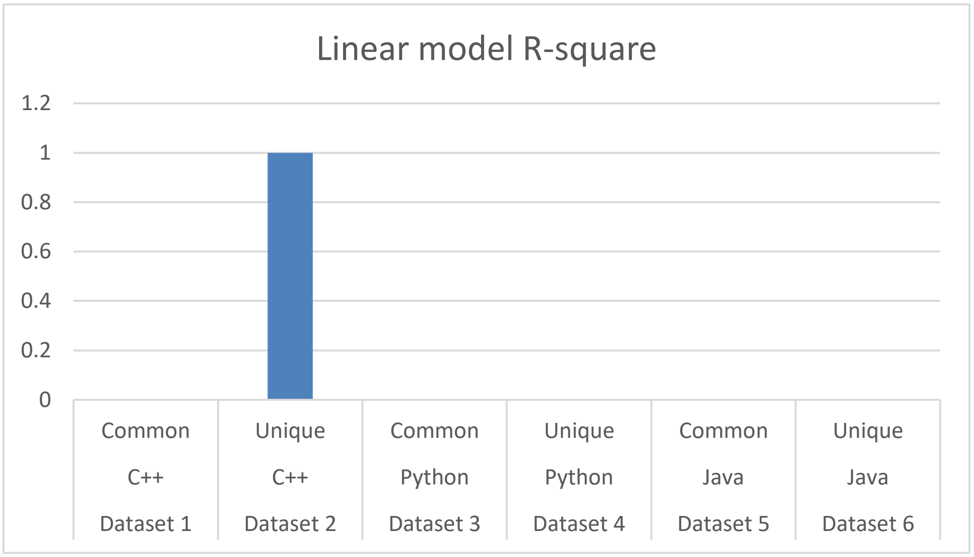


Figure 31. Chart of LM R-Squares of the datasets

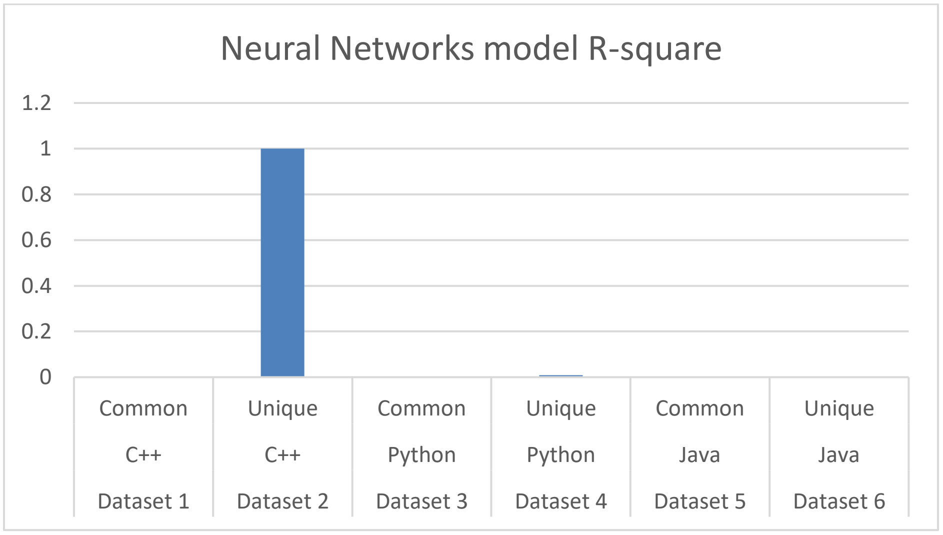


Figure 32. Chart of NNM R-Square of the data sets

The charts in Figures 31 and 32 show that the performance of the two models were the same on three of the datasets (datasets, 1, 2 and 6) but the overall performance of NNM performed better than the LM.

A closer look at the testing results reveals the following observations:

- There was a linear correlation between independent variables and dependent variables in Dataset 2. The random numbers increased proportionally to their seed. It was the only dataset that both models made the best predictions on. This could be as a result of the PRNG used in generating the values in Dataset 2.

6.0 Conclusion

There is evidence that machine learning algorithms can be used to predict the values created by pseudorandom number generators. Given that linear regression algorithm and a simple regression neural network were able to produce fairly good predictions with reasonable accuracy in some of our experiments, it is believable that more complex machine learning algorithms such as deep learning and recurrence algorithms might produce still better results.

It can also be concluded that reseeding before each random number generated is to be avoided, specifically with C++ as the learning algorithms were often more successful on datasets created using that procedure.

The C++ PRNG is predictable given the seed, but Python and Java PRNGs are not as predictable with or without the seed.

The neural network performed no better than simple linear regression in most cases. This finding was a little surprising, as neural networks are in general a more powerful representation.

6.1 Future Research

The following suggestions are made for future research work related to this topic:

- Tests should be performed on other PRNGs (including those with and without seeding) to further confirm the results of this thesis. This might include those related to secure random number generation (such as the ANSI X9.17 PRNG which uses 3DES).

- Tests should be performed using more sophisticated ML algorithms such as deep learning and recurrence algorithms.
- Larger amount of data should be used in the training.

7 References

[1.] Ben Lorica ,“Practical applications of reinforcement learning in industry”; An overview of commercial and industrial applications of reinforcement learning. December 14, 2017.

[2.] https://www.oreilly.com/radar/practical-applications-of-reinforcement-learning-in-industry/

[3.] Jens Kober, J. Andrew Bagnell, Jan Peters “Reinforcement learning in robotics: A survey” First Published August 23, 2013 Research Article.

[4.] https://doi.org/10.1177/0278364913495721

[5.] Steeve Huang “Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG),” , Jan 11, 2018.

[6.] https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287

[7.] Jonathan Hui, “Reinforcement Learning Algorithms Quick Overview,” Aug 29, 2018 ·

[8.] https://medium.com/@jonathan_hui/rl-reinforcement-learning-algorithms-quick-overview-6bf69736694d

[9.] https://en.wikipedia.org/wiki/Reinforcement_learning

[10.] Kaelbling, Leslie P.; Littman, Michael L.; Moore, Andrew W. "Reinforcement Learning: A Survey". Journal of Artificial Intelligence Research. 4: 237–285. arXiv:cs/9605103. doi:10.1613/jair.301. (1996). Archived from the original on 2001-11-20.

[11.] Van Otterlo, M.; Wiering, M. Reinforcement learning and markov decision processes. Reinforcement Learning. Adaptation, Learning, and Optimization. (2012). pp. 3–42. doi:10.1007/978-3-642-27645-3_1. ISBN 978-3-642-27644-6.

[12.] Gosavi, Abhijit Simulation-based Optimization: Parametric Optimization Techniques and Reinforcement. Operations Research/Computer Science Interfaces (2003). Series. Springer. ISBN 978-1-4020-7454-7.

[13.] Felix Dörre and Vladimir Klebanov, Karlsruhe Institute of Technology, Germany, CVE-2016-6313.

[14.] https://www.outsource2india.com/software/articles/businesses-benefits-machine-learning.asp

[15.] https://www.sas.com/en_us/insights/analytics/machine-learning.html

[16.] https://newsroom.ibm.com/IBM-helps-bring-supercomputers-into-the-global-fight-against-COVID-19

[17.] "The Legacy of DES - Schneier on Security". www.schneier.com. October 6, 2004.

[18.] National Bureau of Standards, Data Encryption Standard, FIPS-Pub.46. National Bureau of Standards, U.S. Department of Commerce, Washington D.C., January 1977

[19.] "Announcing the ADVANCED ENCRYPTION STANDARD (AES)" (PDF). Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26,

2001. Archived (PDF)from the original on March 12, 2017. Retrieved October 2, 2012.

[20.] Daemen, Joan; Rijmen, Vincent "AES Proposal: Rijndael"(PDF). National Institute of Standards and Technology. (March 9,
2003). p. 1. Archived (PDF) from the original on 5 March 2013. Retrieved 21 February 2013.

[21.] Anderson, D. R., Sweeney, D. J., and Williams, T. A. "Essentials of Statistics for Business and Economics (3rd edition)." Accessed January 8, 2020.

[22.] North Carolina State University. Using Cigarette Data for An Introduction to Multiple Regression. Journal of Statistics Education, 2(1). Accessed January 8, 2020.

[23.] Massachusetts Institute of Technology: MIT OpenCourseWare. "Statistics for Applications: Simple Linear Regression." Accessed January 8, 2020.

[24.] Mendenhall, W., and Sincich, T. (1992). "Statistics for Engineering and the Sciences (5th edition)." Accessed January 8, 2020.

[25.] M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”, ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

[26.] Tom M. Mitchell “Machine Learning” McGraw-Hill Companies, Inc. pp. 367 1997.

[27.] K. Kourou et al. / Computational and Structural Biotechnology Journal 13 (2015) 8–17

[28.] Balázs Csanád Csáji, Approximation with Artificial Neural Networks MSc thesis, Faculty of sciences ötvös Loránd, University Hungary.

[29.] Michael Nielsen Neural Networks and Deep Learning

[30.] Hopfield, J. J. (1982). "Neural networks and physical systems with emergent collective computational abilities". Proc. Natl. Acad. Sci. U.S.A.

[31.] Von Neumann, John (1951). "Various techniques used in connection with random digits" (PDF). National Bureau of Standards Applied Mathematics Series. 12: 36–38.

[32.] Cox, D. R., and E. J. Snell. 1989. *The Analysis of Binary Data*, 2nd ed. London: Chapman and Hall.

[33.] https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292

[34.] https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

[35.] Petr Savicky & Marko Robnik-Šikonja (2008) LEARNING RANDOM NUMBERS: A MATLAB ANOMALY, Applied Artificial Intelligence, 22:3, 254-265, DOI: 10.1080/08839510701768382

[36.] F. Fan and G. Wang, "Learning From Pseudo-Randomness With an Artificial Neural Network–Does God Play Pseudo-Dice?" in IEEE Access, vol. 6, pp. 22987-22992, 2018.

Appendix Pages

Appendix 1

Sample data set 1 generated with C++ rand() and srand() functions with the same seed 1.

X	Y
1	41
2	18467
3	6334
4	26500
5	19169
6	15724
7	11478
8	29358
9	26962
10	24464
11	5705
12	28145
13	23281
14	16827
15	9961
16	491
17	2995
18	11942
19	4827
20	5436
21	32391
22	14604
23	3902
24	153
25	292
26	12382
27	17421
28	18716
29	19718
30	19895
31	5447
32	21726

33	14771
34	11538

Appendix 2

Sample data set 2: generated from C++ rand() and srand() functions with different seeds.

X	Y
1	41
2	45
3	48
4	51
5	54
6	58
7	61
8	64
9	68
10	71
11	74
12	77
13	81
14	84
15	87
16	90
17	94
18	97
19	100
20	103
21	107
22	110
23	113
24	116
25	120
26	123
27	126
28	130
29	133
30	136

31	139
32	143
33	146

Appendix 3

Sample data set 3: generated from python random() with the same seed.

X	Y
1	0.13
2	0.85
3	0.76
4	0.26
5	0.5
6	0.45
7	0.65
8	0.79
9	0.09
10	0.03
11	0.84
12	0.43
13	0.76
14	0
15	0.45
16	0.72
17	0.23
18	0.95
19	0.9
20	0.03
21	0.03
22	0.54
23	0.94
24	0.38
25	0.22
26	0.42
27	0.03
28	0.22
29	0.44
30	0.5
31	0.23

32	0.23
33	0.22
34	0.46

Appendix 4

Sample data set 4: generated from python random() with different seeds.

X	Y
1	0.13
2	0.96
3	0.24
4	0.24
5	0.62
6	0.79
7	0.32
8	0.23
9	0.46
10	0.57
11	0.45
12	0.47
13	0.26
14	0.11
15	0.97
16	0.36
17	0.52
18	0.18
19	0.68
20	0.91
21	0.16
22	0.96
23	0.92
24	0.71
25	0.38
26	0.75
27	0.65
28	0.11
29	0.55
30	0.54
31	0.01

32	0.08
33	0.57
34	0.53

Appendix 5

Sample data set 5: generated from Java java.util.random() class with the same seed.

X	Y
1	985
2	588
3	1847
4	313
5	254
6	904
7	434
8	606
9	1978
10	1748
11	569
12	473
13	317
14	1263
15	1562
16	1234
17	1592
18	1262
19	596
20	189
21	1376
22	332
23	1310
24	1099
25	674
26	959
27	1298
28	153
29	1437
30	1302
31	1205

32	854
33	800
34	1006

Appendix 6

Sample data set 6: generated from Java java.util.random() class with different seeds.

X	Y
1	985
2	108
3	1734
4	1862
5	1487
6	611
7	236
8	364
9	1989
10	1113
11	738
12	866
13	492
14	1615
15	1241
16	1351
17	976
18	100
19	1725
20	1853
21	1478
22	602
23	227
24	355
25	1981
26	1104
27	730
28	857
29	483
30	1606
31	1232

32	1377
33	1003
34	126

Appendix 7

C++ code to generate random numbers based on the same seed one (1)

Using srand() and rand() functions. (for data set 1)

```
//driver code

int main()

{

    srand(1); // replaced vale with different your
preferred seed.

   for(int a = 0; a<2000; a++)

         {

              for(int i = 0; i<1; i++)

              {

                  cout<<a+1<<"--";


                  cout<<rand()<<endl;

              }

         }

   return 0;

}
```

Appendix 8

C++ program to generate random numbers based on the different sequential seed, from 1 to 2000. (for data set 2)

```
#include <iostream>

#include <iomanip>

#include <ctime>

using namespace std;


//driver code

int main()

{

  for(int a = 0; a<2000; a++)

        {

        srand(a+1);

            for(int i = 0; i<1; i++)

            {

                cout<<a+1<<"--";

                cout<<rand()<<endl;

            }

        }

  return 0;

}
```

Appendix 9

Python code for generating 2000 random numbers from the same seed of 1

For generation data set 3

```
import random

sn = 0

random.seed(1)

while(sn < 2000):


    sn = sn +1


    pyRand1 = random.random()

    print(sn, "-", round(pyRand1, 2))
```

Appendix 10

Python code for generating 2000 random numbers from the different seeds

For generating data set 4

```
import random

sn = 0

while(sn < 2000):

    sn = sn +1


    random.seed(sn)

    pyRand1 = random.random()

    print(sn, "-", round(pyRand1, 2))
```

Appendix 11

Java code for generating Dataset 5 (set of random numbers from the same seed)

```
package randomNumbers;

import java.util.*;

public class randomNumbers {
      public static void main(String args[]){


     // create instance of Random class
        Random rand = new Random();

     // set seed
        long s = 1;
        rand.setSeed(s);

      // Generate random integers in range 0 to 1999
        for(int i = 0; i < 2000; i++) {


            int rand_int1 = rand.nextInt(2000);

       // Print random integers
            System.out.println( i+1 +" - "+rand_int1);
            s +=1;
        }


    }

}
```

Appendix 12

Java code for generating Dataset 6 (set of random numbers from the different seeds)

```
package randomNumbers;

import java.util.*;

public class randomNumbers {
      public static void main(String args[]){



            // create instance of Random class
        Random rand = new Random();

        // set seed
        long s = 1;

        // Generate random integers in range 0 to 1999
        for(int i = 0; i < 2000; i++) {


        rand.setSeed(s);

        int rand_int1 = rand.nextInt(2000);

        // Print random integers
            System.out.println( s +" - "+rand_int1);
            s +=1;
        }


    }

}
```

Appendix 13

Summary of the Linear Regression model (built using lm): for training on Dataset 1

```
Call:

lm(formula = Y ~ ., data = crs$dataset[crs$train, c(crs$input,

    crs$target)])

Residuals:

     Min       1Q   Median       3Q      Max

-16467.6  -7947.9    169.9   8066.5  16540.1

Coefficients:

              Estimate Std. Error t value Pr(>|t|)

(Intercept) 16204.5062   500.8927  32.351   <2e-16 ***

X               0.2291     0.4312   0.531    0.595

Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9389 on 1398 degrees of freedom

Multiple R-squared:  0.0002019,  Adjusted R-squared:  -0.0005133

F-statistic: 0.2823 on 1 and 1398 DF,  p-value: 0.5953

==== ANOVA ====

Analysis of Variance Table

Response: Y

            Df       Sum Sq  Mean Sq F value Pr(>F)

X            1     24885734 24885734  0.2823 0.5953

Residuals 1398 123243239949 88156824

[1] "\n"

Time taken: 0.14 secs
```

Appendix 14

Summary of the Neural Net model (built using nnet): for training on Dataset 1.

```
A 1-10-1 network with 32 weights.

Inputs: X.

Output: Y.

Sum of Squares Residuals: 123243239949.4717.

Neural Network build options: skip-layer connections; linear
output units.

In the following table:

   b  represents the bias associated with a node

   h1 represents hidden layer node 1

   i1 represents input node 1 (i.e., input variable 1)

   o  represents the output node

Weights for node h1:

    b->h1    i1->h1

  -352.02  -2857.30

Weights for node h2:

    b->h2    i1->h2

   235.29     -0.11

Weights for node h3:

    b->h3    i1->h3

  -117.65   -663.48

Weights for node h4:

    b->h4    i1->h4

   483.94   3503.18

Weights for node h5:

    b->h5    i1->h5
  -338.53  -2929.99

Weights for node h6:

    b->h6    i1->h6

 -1022.99 -70932.06

Weights for node h7:

    b->h7    i1->h7

   331.39   1672.58

Weights for node h8:

    b->h8    i1->h8

   764.72   5342.82

Weights for node h9:

    b->h9    i1->h9

   105.57    746.97

Weights for node h10:

   b->h10   i1->h10

   978.82   7455.83

Weights for node o:

     b->o     h1->o     h2->o     h3->o     h4->o     h5->o 
h6->o     h7->o

  4330.84    -71.46   3112.99    626.78   2510.75   2587.90 
40608.05   1321.59

    h8->o     h9->o    h10->o     i1->o

  2795.53    962.91   1169.90      0.23

Time taken: 0.11 secs

Rattle timestamp: 2020-04-19 21:39:43
```

Appendix 15

Summary of the Linear Regression model (built using lm): for training dataset 2.

```
Call:
lm(formula = Y ~ ., data = crs$dataset[crs$train, c(crs$input,
    crs$target)])
Residuals:
     Min       1Q   Median       3Q      Max
-0.51332 -0.25354 -0.00405  0.24628  0.51102
Coefficients:
               Estimate  Std. Error t value Pr(>|t|)
(Intercept) 38.10442038  0.01534825    2483   <2e-16 ***
X            3.26559480  0.00001321  247161   <2e-16 ***
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 0.2877 on 1398 degrees of freedom
Multiple R-squared:      1,Adjusted R-squared:      1
F-statistic: 6.109e+10 on 1 and 1398 DF,  p-value: < 2.2e-16
==== ANOVA ====
Analysis of Variance Table
Response: Y
            Df     Sum Sq    Mean Sq     F value    Pr(>F)
X            1 5056419600 5056419600 61088418483 < 2.2e-16 ***
Residuals 1398        116          0
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
[1] "\n"
Time taken: 0.20 secs
Appendix 16
```

```
Summary of the Neural Net model (built using nnet): for training
on Dataset 2.

A 1-10-1 network with 32 weights.

Inputs: X.

Output: Y.

Sum of Squares Residuals: 115.7155.

Neural Network build options: skip-layer connections; linear
output units.

In the following table:

   b  represents the bias associated with a node

   h1 represents hidden layer node 1

   i1 represents input node 1 (i.e., input variable 1)

   o  represents the output node

Weights for node h1:

  b->h1  i1->h1

 104.02 1390.38

Weights for node h2:

  b->h2  i1->h2

  15.28  -33.17

Weights for node h3:

  b->h3  i1->h3

  -7.98  -33.46

Weights for node h4:

  b->h4  i1->h4

  64.17  342.45

Weights for node h5:

  b->h5  i1->h5
 -47.50 -851.74

Weights for node h6:

  b->h6  i1->h6

 -24.74 -413.77

Weights for node h7:

  b->h7  i1->h7

 105.75   56.61

Weights for node h8:

  b->h8  i1->h8

  72.27  357.09

Weights for node h9:

  b->h9  i1->h9

  30.08  136.44

Weights for node h10:

 b->h10 i1->h10

  90.21  505.85

Weights for node o:

   b->o   h1->o   h2->o   h3->o   h4->o   h5->o   h6->o   h7->o
h8->o   h9->o

 118.94  -98.53   65.37   34.72   15.08   20.73    4.34   14.16
30.16   11.45

 h10->o   i1->o

 -53.15    3.27

Time taken: 0.05 secs

Rattle timestamp: 2020-04-20 04:06:12
```

Appendix 17

Summary of the Linear Regression model (built using lm): for training on Dataset 3.

```
Call:
lm(formula = Y ~ ., data = crs$dataset[crs$train, c(crs$input,
    crs$target)])
Residuals:
     Min       1Q   Median       3Q      Max
-0.52197 -0.24346  0.00441  0.24817  0.50897
Coefficients:
               Estimate  Std. Error t value Pr(>|t|)
(Intercept)  0.52220320  0.01533102   34.06   <2e-16 ***
X           -0.00001664  0.00001320   -1.26    0.208
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 0.2874 on 1398 degrees of freedom
Multiple R-squared:  0.001135,   Adjusted R-squared:  0.0004207
F-statistic: 1.589 on 1 and 1398 DF,  p-value: 0.2077
==== ANOVA ====
Analysis of Variance Table
Response: Y
            Df  Sum Sq  Mean Sq F value Pr(>F)
X            1   0.131 0.131216  1.5888 0.2077
Residuals 1398 115.456 0.082586
[1] "\n"
Time taken: 0.05 secs
```

Appendix 18

Summary of the Neural Net model (built using nnet): for training on Dataset 3

```
A 1-10-1 network with 32 weights.

Inputs: X.

Output: Y.

Sum of Squares Residuals: 115.3024.

Neural Network build options: skip-layer connections; linear
output units.

In the following table:

   b  represents the bias associated with a node

   h1 represents hidden layer node 1

   i1 represents input node 1 (i.e., input variable 1)

   o  represents the output node

Weights for node h1:

 b->h1 i1->h1

-10.58 -11.37

Weights for node h2:

 b->h2 i1->h2

 -0.77  -4.78

Weights for node h3:

 b->h3 i1->h3

 -2.05  -3.69

Weights for node h4:

 b->h4 i1->h4

  2.32   7.18

Weights for node h5:

 b->h5 i1->h5
 -1.11  -3.35

Weights for node h6:

 b->h6 i1->h6

 -2.19  -6.15

Weights for node h7:

 b->h7 i1->h7

  0.05  -3.04

Weights for node h8:

 b->h8 i1->h8

  1.16   2.45

Weights for node h9:

 b->h9 i1->h9

 -1.21  -2.72

Weights for node h10:

 b->h10 i1->h10

   0.80    3.28

Weights for node o:

  b->o  h1->o  h2->o  h3->o  h4->o  h5->o  h6->o  h7->o  h8->o
h9->o h10->o  i1->o

  4.83 -25.27   0.64  -2.11 -14.53   0.34   3.62  -2.47   5.49  -
2.79   4.73   0.00

Time taken: 0.04 secs


Rattle timestamp: 2020-04-20 04:29:00
```

Appendix 19

Summary of the Linear Regression model (built using lm): for training on Dtataset 4.

```
Call:

lm(formula = Y ~ ., data = crs$dataset[crs$train, c(crs$input,

    crs$target)])

Residuals:

     Min       1Q   Median       3Q      Max

-0.49387 -0.25710 -0.00471  0.25500  0.52263

Coefficients:

              Estimate Std. Error t value Pr(>|t|)

(Intercept) 0.47084351 0.01561220  30.159   <2e-16 ***

X           0.00001395 0.00001344   1.038      0.3

Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2926 on 1398 degrees of freedom

Multiple R-squared:  0.0007696, Adjusted R-squared:  5.487e-05

F-statistic: 1.077 on 1 and 1398 DF,  p-value: 0.2996

==== ANOVA ====

Analysis of Variance Table

Response: Y

             Df  Sum Sq  Mean Sq F value Pr(>F)

X             1   0.092 0.092218  1.0768 0.2996

Residuals 1398 119.730 0.085644

[1] "\n"

Time taken: 0.02 secs

Rattle timestamp: 2020-04-20 04:34:26
```

Appendix 20

Summary of the Neural Net model (built using nnet): for training on Dataset 4.

```
A 1-10-1 network with 32 weights.

Inputs: X.

Output: Y.

Sum of Squares Residuals: 119.5464.

Neural Network build options: skip-layer connections; linear
output units.

In the following table:

   b  represents the bias associated with a node

   h1 represents hidden layer node 1

   i1 represents input node 1 (i.e., input variable 1)

   o  represents the output node

Weights for node h1:

 b->h1 i1->h1

 -5.38 -30.04

Weights for node h2:

 b->h2 i1->h2

  0.58  -0.42

Weights for node h3:

 b->h3 i1->h3

 -0.62  -0.12

Weights for node h4:

 b->h4 i1->h4

 -1.79  -3.86


Weights for node h5:
 b->h5 i1->h5

 -0.33  -0.15

Weights for node h6:

 b->h6 i1->h6

 -1.64  -5.57

Weights for node h7:

 b->h7 i1->h7

 -1.83 -31.41

Weights for node h8:

 b->h8 i1->h8

 -1.54   0.76

Weights for node h9:

 b->h9 i1->h9

 -0.09  -0.76

Weights for node h10:

 b->h10 i1->h10

  -2.03   -0.37

Weights for node o:

  b->o  h1->o  h2->o  h3->o  h4->o  h5->o  h6->o  h7->o  h8->o
h9->o h10->o  i1->o

 -0.66  -0.03   0.45   0.01   0.72   0.06   0.73  -4.97   1.13
0.70  -1.07   0.00

Time taken: 0.02 secs

Rattle timestamp: 2020-04-20 04:36:52
```

Appendix 21

Summary of the Linear Regression model (built using lm): for training on Dataset 5.

```
Call:

lm(formula = Y ~ ., data = crs$dataset[crs$train, c(crs$input,

    crs$target)])

Residuals:

    Min      1Q  Median      3Q     Max

-982.65 -492.74  -20.93  489.51 1018.00

Coefficients:

              Estimate Std. Error t value Pr(>|t|)

(Intercept) 977.722765  30.719710   31.83   <2e-16 ***

X             0.004241   0.026445    0.16    0.873

Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 575.8 on 1398 degrees of freedom

Multiple R-squared:  1.84e-05,   Adjusted R-squared:  -0.0006969

F-statistic: 0.02572 on 1 and 1398 DF,  p-value: 0.8726

==== ANOVA ====

Analysis of Variance Table

Response: Y

            Df    Sum Sq Mean Sq F value Pr(>F)

X            1      8528    8528  0.0257 0.8726

Residuals 1398 463562051  331589

[1] "\n"

Time taken: 0.01 secs

Rattle timestamp: 2020-04-20 04:42:21 herma

Appendix 22
```

```
Summary of the Neural Net model (built using nnet): for training
on Dataset 5.

A 1-10-1 network with 32 weights.

Inputs: X.

Output: Y.

Sum of Squares Residuals: 462334474.0702.

Neural Network build options: skip-layer connections; linear
output units.

In the following table:

   b  represents the bias associated with a node

   h1 represents hidden layer node 1

   i1 represents input node 1 (i.e., input variable 1)

   o  represents the output node

Weights for node h1:

  b->h1  i1->h1

   3.92   12.64

Weights for node h2:

  b->h2  i1->h2

  16.51   20.81

Weights for node h3:

  b->h3  i1->h3

 442.58 -112.82

Weights for node h4:

  b->h4  i1->h4

 371.92  303.84

Weights for node h5:

  b->h5  i1->h5
 364.36  -41.70

Weights for node h6:

  b->h6  i1->h6

  -0.41  -52.46

Weights for node h7:

  b->h7  i1->h7

  -1.47  -15.20

Weights for node h8:

  b->h8  i1->h8

 165.53  160.70

Weights for node h9:

  b->h9  i1->h9

  31.61   26.29

Weights for node h10:

 b->h10 i1->h10

  99.27  103.39

Weights for node o:

   b->o   h1->o   h2->o   h3->o   h4->o   h5->o   h6->o   h7->o
h8->o   h9->o

 204.26  126.79    5.15  896.84  201.40 -461.28   28.20   76.64
203.58   36.73

 h10->o   i1->o

 202.57    0.00

Time taken: 0.02 secs

Rattle timestamp: 2020-04-20 04:47:03
```

Appendix 23

Summary of the Linear Regression model (built using lm): for training on Dataset 6.

```
Call:
lm(formula = Y ~ ., data = crs$dataset[crs$train, c(crs$input,
    crs$target)])
Residuals:
    Min      1Q  Median      3Q     Max
-996.49 -512.22    5.87  499.61 1002.14
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) 995.784990  31.146065  31.971   <2e-16 ***
X             0.001374   0.026812   0.051    0.959
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 583.8 on 1398 degrees of freedom
Multiple R-squared:  1.879e-06,  Adjusted R-squared:  -0.0007134
F-statistic: 0.002626 on 1 and 1398 DF,  p-value: 0.9591
==== ANOVA ====
Analysis of Variance Table
Response: Y
            Df    Sum Sq Mean Sq F value Pr(>F)
X            1       895     895  0.0026 0.9591
Residuals 1398 476518789  340858
[1] "\n"
Time taken: 0.01 secs
Rattle timestamp: 2020-04-20 04:50:12
```

Appendix 24

```
Summary of the Neural Net model (built using nnet): for training
on Dataset 6.

A 1-10-1 network with 32 weights.

Inputs: X.

Output: Y.

Sum of Squares Residuals: 476518672.3179.

Neural Network build options: skip-layer connections; linear
output units.

In the following table:

   b  represents the bias associated with a node

   h1 represents hidden layer node 1

   i1 represents input node 1 (i.e., input variable 1)

   o  represents the output node

Weights for node h1:

   b->h1   i1->h1

 -780.84  4363.65

Weights for node h2:

   b->h2   i1->h2

   28.77  -413.11

Weights for node h3:

   b->h3   i1->h3

  -24.11   -71.17

Weights for node h4:

   b->h4   i1->h4

-1189.31 -1057.64

Weights for node h5:

   b->h5   i1->h5
  -54.70  -306.90

Weights for node h6:

   b->h6   i1->h6

  -95.92 -6664.04

Weights for node h7:

   b->h7   i1->h7

  500.10  3107.79

Weights for node h8:

   b->h8   i1->h8

 -269.60   143.85

Weights for node h9:

   b->h9   i1->h9

 -540.06 -8173.22

Weights for node h10:

  b->h10  i1->h10

 -194.84   380.37

Weights for node o:

    b->o    h1->o    h2->o    h3->o    h4->o    h5->o    h6->o
h7->o    h8->o

  545.54  -117.96   328.62    90.94   268.74   308.22  3872.81
416.53    10.76

   h9->o   h10->o    i1->o

  942.42   140.94     0.00

Time taken: 0.01 secs

Rattle timestamp: 2020-04-20 04:52:48
```