Synthetic Data Generator For User Behavioral Analytics Systems

by

Asad Narayanan

A Thesis submitted to
the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of
the requirements for the degree of
Master of Computer Science with Data Science Specialization

Ottawa-Carleton Institute of Computer Science
Department of Computer Science
Carleton University
Ottawa, Ontario, Canada
September 2017

Abstract

Many of the User Behavioral Analytics (UBA) applications rely on the distributions and baselines of individual users and are sensitive to the changes in these patterns. These applications identify anomalies and threats by observing the deviation of user behavior from their baselines. Development and testing of these applications depend heavily on synthetic data as the availability of the real data is scarce in most of the cases. Synthetic data generated has to follow these patterns, or else it could result in noisy results. Through this work, we present a synthetic data generation technique, which could be utilized by UBA applications for their development and testing. The proposed system for data generation extracts the distribution of attributes, considering the dependencies between these attributes provided by the user. The extracted patterns are stored and can be used any number of time to generate synthetic data. Additionally, we also generate synthetic users, whose behaviors and baselines are similar to that of real users. The scalable system developed will help the development of UBA applications, especially during performance testing. The experiments conducted showed that the synthetic data captured the required patterns and relations from the real data. The experiments also showed that the process of data generation we follow can be scaled up linearly to the available number of processors.

Acknowledgments

First of all, I am deeply grateful to my supervisor Dr. Anil Maheshwari for his valuable guidance and constant support during this work. Completing this work without his help would have been impossible. His directions on both research and career always helped me to take the best decisions. I would also like to thank Dr. John Oommen for dedicating his valuable time to impart the knowledge about areas such as pattern extraction, that became critical part of this work.

I would like to express my gratitude to the Thesis defense committee (Dr. Amiya Nayak, Dr. Anil Maheshwari, Dr. John Oommen and Dr. Michiel Smid) for their thought provoking questions and invaluable comments.

Its my pleasure to take this opportunity to thank Shaun Pilkington, Maria Pospelova, and Stephan Jou from Interset for helping me with the crucial task of narrowing down my thesis problem. I would like to express my sincere thanks to Shaun Pilkington for introducing me to this problem and providing valuable ideas throughout the course of this work.

I would like to thank Ajitha Marath for keeping aside her valuable time to proof-read and edit my report. I would also like to thank Philip Benjamin George for his sincere efforts in reviewing my work.

I would like to thank my parents for always providing me with the moral support and backing my decisions. I would also like to thank Aparna, Gopakumar, Haripriya, Sathi, and Sreepriya for helping and supporting me in many ways.

Table of Contents

List of Tables

List of Figures

List of Acronyms

Acronyms	Definition
CPT	Conditional Probability Table
KDE	Kernel Density Estimation
NRMSE	Normalized Root Mean Square Error
PDF	Probability Density Function
PMML	Predictive Model Markup Language
PSDG	Parallel Synthetic Data Generator
RMSE	Root Mean Square Error
SDDL	Synthetic Data Description Language
SIEM	Security Information and Event Management
UBA	User Benahioral Analytics
VPN	Virtual Private Network
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation

Over the past few years, we have been witness to a rapid growth of digital data. Advancements in the Internet of Things and an increasing number of connected people across the globe are the reason for this exponential growth [2]. According to the International Data Corporation, the amount of digital data in the world will reach 44 Zettabytes (ZB) by 2020 - which is ten times that of 2013, and by 2025 this is expected to cross 180 ZB [2]. With the increased volume of data, storing and processing them to extract information has become more and more challenging. Big data analytics is an area that focuses on developing tools and techniques to process this huge volume of data, with the goal of extracting useful information in an efficient manner.

One of the main challenges of building such big data analytics tools is evaluating their endurance in processing the large volume of data in production environment. This is due to the fact that these analytics tools might behave unexpectedly when subjected to a huge amount of data [3]. Thus, to make sure the developed system is predictable and fault-tolerant, it has to go through different levels of performance testing. Testing the extremities of analytic softwares requires a huge

amount of good quality data. Unfortunately, the availability of the real data is limited due to several factors. Many organizations are unwilling to share their data with third parties due to privacy concerns, the cost of data transfer, as well as the unavailability of such a huge amount of data [3, 4].

Consequently, in the absence of real data, the development and testing of the analytics softwares rely on the synthetic data. For credible results during development and testing, it is important for the synthetic data to capture the characteristics of the real data [3]. Thus synthetic data generators play a key role in the development of successful analytics softwares for big data analysis. Synthetic data generators must be expressive and scalable in order to generate large quantities of realistic synthetic data in a short period of time. In the past, there have been several attempts made to generate synthetic data for different scenarios, and some of which are discussed in Chapter 2. In this thesis we propose a scalable synthetic data generator that is capable of generating synthetic data for User Behavioral Analytics applications.

User Behavioral Analytics (UBA), is an emerging area in data analytics that analyzes the behavior of individual users of an organization and use this knowledge to find anomalies or users with malicious intent. I stumbled upon this area while working at Interset during my co-op work term. Interset is an Ottawa based startup, which is pioneering the development of User Behavioral Analytics softwares. They provide their customers with an intuitive way to identify Intellectual Property theft and several other internal threats, which may not have spotted otherwise. Interset is backed by powerful concepts of machine learning, and patented techniques based on predictive analytics and risk based scoring [5]. With several hundreds of such anomaly models, testing the software developed by companies such as that of Interset is a challenging process. Real data obtained from the customers is insufficient for any high-quality performance testing. This forces these

companies to use of synthetic data for testing their systems.

Synthetic data generation for UBA systems is a cumbersome task. This is because, it is important for the synthetic data to maintain the behavioural baselines of individual users, based on which the models are designed. For example, in some cases of UBA, the working hour patterns for individual users are analyzed. Each user may have working hour patterns that are different from other users. If the synthetic data does not preserve these patterns, it may negatively impact the development and testing of analytic models that rely on these patterns. Low-quality synthetic data would also affect the validity of the benchmark results obtained during performance tests. Thus, it is necessary for the synthetic data to mimic the properties of its real world counterpart.

Apart from development and testing, synthetic data is also used to demo various products. This is because, in most of the cases, the real data may not be sufficient to exhibit all the features of a product. Moreover, there might also exist privacy issues in publicly displaying the real data. Synthetic data can be carefully designed to demonstrate all the features of the product without any fear of disclosure of confidential information.

1.2 User Behavioral Analytics (UBA)

User Behavioral Analytics(UBA) is an emerging area in the information security, that aims at proactively identifying potential vulnerabilities to an organization such as insider threats, targeted attacks, and financial frauds. These vulnerabilities are identified by running machine learning and statistical algorithms on historical data from authentication logs, VPN logs and other security logs that are collected by log management systems such as SIEM [6, 7]. Unlike other systems, a UBA system does not throw alerts just by looking at individual events. Instead, the potential risks are analyzed by assessing the behavioral variations from the baselines, of the entities involved in the event [8]. For example, before flagging a user as malicious, the UBA systems consider several factors such as the risk of entities involved with the user, timestamp of the associated events, job role of the user, activities of users in his/her peer group etc. It is important to identify such contextual information of events as this can reduce the false positives drastically. For example, the event of a developer in an organization, accessing its financial records at midnight can be considered as anomalous, but when an accountant of the same organization does the same, it might indicate their usual behavior. UBA systems are capable of capturing such contextual differences precisely which in turn makes the life of information security professionals easy [8].

1.3 Research Question

UBA systems like that of Interset, are used by organizations, with more than hundreds and thousands of employees. These systems are capable of analyzing millions of events generated by these users and learn their individual behavioral baselines. The goal of this work is to come up with a scalable system, that is capable of generating synthetic data for development and testing of such UBA systems.

The data that we are focusing through this work is security log file, which record the activities of individual users of an organization. Some examples of these logs are:

1. **Web proxy logs:** Web proxy log captures details of browsing activities in an organization. Such logs, in most cases include attributes such as user name or host name, source IP address, destination IP address, URL of the website accessed, bytes of data sent and received, HTTP method, HTTP response, etc. These are mostly used by network forensics to identify abnormal browsing behaviors of users [9].

2. **Active directory logs:** Active directory is a directory service provided by Microsoft that runs on domain controllers. It manages an organization's users, computers, shared resources etc. and also protects these network objects from unauthorized access. Active directory records the activities of users in the form of event logs, which are used by security personnels of an organization to detect malicious users. Active directory log captures information related to the events such as timestamp, username, destination host, event id (a unique number representing the type of event the user has triggered, for example, authentication success, authentication failure, etc.) and many other attributes [10].

3. **Repository logs:** Repository log records activities of users on different project repositories. It monitors and records actions each that each user performs on projects of an organization. The common attributes of this log are timestamp, user name, project name, action.

The system designed through this work captures the distributions and patterns of attributes, from the given sample log data and use these information to generate much larger log files. For example, using a log file containing one week activities of 10 users, this system can generate 1 year log data for 100 users. The main characteristics of the proposed system are discussed below.

1. Synthetic data generated should capture the dependencies and distributions of real data, including patterns such as active hours of individual users.

2. Generate additional synthetic users using the extracted patterns, to mimic the behaviors of users in real data.

3. Develop a scalable data generation process to generate huge amount of data in a parallel environment.

1.4 Overview of the results

Through this thesis we develop a system that is used for generating synthetic data for testing, development and demo of analytics softwares. The main focus is towards generating data for testing UBA systems such as that of Interset. This system first extracts the necessary patterns of the given real data and store it in a file. This file, which occupies much less memory compared to the real data, can be reused any number of times to generate data of desired magnitude. Pattern extraction is a one time process and extracts distribution from attributes of different types such

as categorical, numeric (float or integer) and time series. Pattern extraction also considers dependencies between the attributes. We tested our system using three different datasets, two web proxy log datasets and Iris data. Our experiments showed that the generated data preserved the dependencies and distribution of real data. We also showed that the data generation process is a linearly scalable to the number of processors.

1.5 Organization of the Thesis

In Chapter 2, we discuss some of the preliminary concepts and relevant works in the area of synthetic data generation. This include some of the methods used by them to extract and represent the distribution of categorical, numerical and timestamp data. In Chapter 3, we explain the methodology that we follow for extracting, and storing the patterns from real data, and generating synthetic data from these patterns. In this chapter we also discuss the methodology we follow to generate synthetic user profiles and the algorithm used for parallel data generation. Finally we discuss some of the constraints and assumptions that we follow while developing the system. Chapter 4 describes the experiments and analysis of the results. Chapter 5 includes the conclusion and possible future extensions.

Chapter 2

Preliminaries

This chapter provides some background knowledge related to the area of synthetic data generation. The first section of this chapter discusses some of the previous systems that are developed for generating synthetic data. In the second section, we explain some existing approaches for extracting patterns from the different attribute types of data such as *Categorical*, *Numeric* and *Timestamp*. The final section explains the Kernel Density Estimation (KDE) method that we use to extract patterns from numerical attributes of the data.

2.1 Previous works in synthetic data generation

Synthetic data is artificially generated data that preserves some of the selected properties from the original data [11]. It is widely used among developers and researchers for developing, testing and validating analytic software and models [11–13]. A lot of research has been performed in the area of synthetic data generation. They vary with respect to the requirements they address and in the degree of closeness of the generated data to the real data. In this section, we discuss some of the earlier approaches used for synthetic data generation.

2.1.1 Parallel Synthetic Data Generator (PSDG)

One of the main contribution to the area of synthetic data generation was by Hoag et al. through their work in Parallel Synthetic Data Generation (PSDG) [12]. Their approach decouples the process of data generation from data definition. In their approach the constraints describing the data are defined and stored in a Synthetic Data Description Language (SDDL) file. The process of data generation uses the power of cluster computing to generate a large amount of synthetic data that follows the constraints specified in the SDDL file [12].

SDDL standardizes the process of defining and storing the rules and constraints of the data. Later this approach of using SDDL for defining the rules and constraints of data has by adopted by other researchers [11,12]. SDDL uses a modular language similar to XML to store the description of the data. The attribute values of the data is generated by following constraints defined for that attribute. The different types of constraints supported by SDDL are min/max constraint, distribution constraint, formula constraint, and iteration constraint. Min/max constraint for an attribute defines the range of values that can be assigned to that attribute. A distribution constraint for an attribute consists of a collection of min/max constraints, each with an assigned probability of getting selected [12]. The formula constraint allows the user to define a formula that can be used for generating values for an attribute. Finally, the iteration constraint allows us to pick a value for an attribute by iterating through a set of values [12].

PSDG uses the information stored in SDDL to generate synthetic data in a parallel environment. This approach is capable of generating data deterministically across multiple processors. In a deterministic data generation approach, data can be consistently reproduced independent of the underlying architecture. Using this method they were able to generate data at the rate of 500,000 records per

second [12].

One of the limitations with the SDDL approach is the difficulty to accommodate complex statistical patterns. SDDL is also incapable of including the dependency between the attributes of the data [12]. As the dimensionality of the data increases the complexity in defining the constraints in the SDDL file increases. This process of defining constraints can be simplified if we can automate it by extracting them from a given data sample.

2.1.2 Realistic patterns from data mining models

Eno et al. in their work on "Generating synthetic data to match data mining patterns" attempted to generate synthetic data that captures more realistic patterns [11]. This approach is based on a data mining model which extracts and captures insights about the sample data used for its training. This approach allows us to capture hidden characteristics of the data set used for training. The model representing the data is stored as a PMML (Predictive Model Markup Language) file. This PMML file in turn is translated to constraints and definitions of an SDDL file used for data generation [11].

PMML is an open standard developed by the data mining group [11]. It defines a way of representing the data mining models such that it can be transferred across multiple software packages [11]. It stores different attributes of data and their data types in the form of a data dictionary. The distribution information of these fields are captured and stored as part of data mining model. PMML supports a variety of models such as decision trees, association rules, regression, naïve bayes, neural networks, etc. [11].

The authors use decision tree model learned from the data to define constraints and rules of the data. The decision tree model is stored in a PMML file. The PMML

to SDDL conversion takes places by performing a depth-first search on the tree model, extracting necessary information about the data. This information is used for defining the elements and constraints of the SDDL file [11]. Subsequently, using this SDDL file and the techniques of data generation in PSDG [12], they generate more realistic synthetic data.

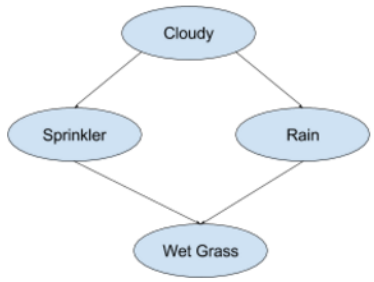
This approach opens the door for several other possibilities of generating more realistic data. Currently, their work only uses decision tree classifier, the closeness between synthetic data and real data can be improved further by using other classifiers such as neural networks or Naïve Bayes. There is also another possibility of extracting more complex patterns by combining patterns obtained from multiple models [11].

2.1.3 Bayesian Networks

Another interesting approach for synthetic data generation is using the principles of Bayesian Networks [14]. Bayesian Network is a powerful statistical tool that is capable of storing complex information about the data.

P(C=F)	P(C=T)
0.5	0.5

C	P(R=F)	P(R=T)
F	0.8	0.2
T	0.2	0.8



C	P(R=F)	P(R=T)
F	0.8	0.2
T	0.2	0.8

S R	P(W=F)	P(W=T)
F F	1.0	0.0
T F	0.1	0.9
F T	0.1	0.9
T T	0.01	0.99

Figure 1: Example of Bayesian Network [1]

A Bayesian Network is a way of representing probabilistic models in the form of a directed acyclic graph capturing all dependencies and probabilities of the model. It consists of two components: *the structure* and *the Conditional Probability Distribution (CPT)* table. The structure encodes the dependencies between random variables (Nodes). The predecessor nodes are the cause, and successor nodes are the effect. As shown in Figure 1, a directed edge from *cloudy* to *rain* indicates that *cloudy* is the cause, and *rain* is the effect [1].

CPTs encode the probability of effect given the cause. It can be represented either as tables with multinomial probabilities or can be extracted from a parameterized distribution [1]. In the example shown in Figure 1, the CPT of node rain gives us the probability of the rain given the presence or absence of the clouds. Probabilities in the root node CPT are obtained from the relative counts of its outcomes in the data. CPTs enable the Bayesian networks to model any complex relationships between a node and its parents [1,15].

Jim et. al. in [14] uses the Bayesian network to generate synthetic data with a primary goal to reduce the risk of disclosure of the real data. Their approach construct a Bayesian Network by extracting its structure and CPTs from the data. The structure is learned by searching and evaluating all possible network structures of the data. The CPTs for all the nodes (attributes of the data) are extracted from the given data [14]. The CPTs of the discrete attributes are represented as multinomial probabilities observed from the data [14]. In the case of continuous attributes of the data, the attributes are discretized before extracting their conditional probabilities. This is because probabilistic inferences are not possible if continuous attributes have discrete children [14].

The Bayesian network gives us an interesting way of representing relationships between the attributes. The dependencies between the attributes can be represented as a directed acyclic graph with independent attributes at the root level and the level of each attribute coming after the level of their parent attributes. The conditionally dependent patterns of each attribute can be stored in the form of CPT of that attribute. We use these two aspects of a Bayesian Network (Structure and CPT) to represent the patterns in UBA datasets. We construct the structure from the dependency information provided by user and CPTs are extracted from given data.

2.2 Pattern extraction from different types of data

This section discusses some of the approaches considered in the literature, to extract patterns from different attribute types of data. The main focus is on three different types of attributes: categorical, numeric(discrete or continuous) and timestamp. The attributes of the data analyzed in this thesis fall into these three categories.

2.2.1 Extracting patterns from categorical data

A straightforward approach of representing the patterns of a categorical attribute of the data, is by specifying all possible values of the attribute along with their observed probabilities in the real data. At the time of data generation, values for the attribute are chosen randomly from this list of values based on their probabilities [3, 13]. For example, if an attribute contains three distinct values and they occur in real data with a probability of 0.5, 0.3 and 0.2 respectively. Then, the same proportion of the attribute values are maintained in the synthetic data generated of any size. However, specifying this information manually becomes a cumbersome task when the size of data grows or if there are conditional dependencies between the attributes of the data [13].

Some researchers make the process of manually listing the labels easy, by allowing the users to provide a file containing the attribute values and their observed frequencies as input. The data generator automatically extracts probabilities for each attribute value by normalizing their frequencies to 1 [13].

In this work, the frequencies of the values in a categorical attribute are extracted automatically from the data. These frequencies are used to find their corresponding probabilities of getting selected. Our approach also allows categorical attributes to be conditionally dependent on other categorical attributes and stores

these conditional probabilities in a conditional probability table.

2.2.2 Extracting patterns from numerical data

Numerical attribute types are common in UBA datasets. These attributes may hold important information such as the number of pages printed, bytes sent and bytes received, etc. UBA applications might use these attributes to define baselines for individual users. This makes it necessary for us to retain the distribution of these numeric attributes in the synthetic data. Next, we discuss different approaches that are used to represent these distributions.

There are several ways of extracting and representing the patterns from the numerical attributes of the data. Based on the requirements being addressed, the complexity of these methods may vary. Jim et al. in [14] simplifies the process by treating the numerical attributes like *age* by discretizing it into groups of equal size. Hoag et al. gives the user more flexibility by providing a variety of ways to represent the numerical data [12]. The easiest among them is to pick the numerical attribute values from a uniform distribution specified by minimum and maximum values. In the case of more complex data, the user can also specify a set of such uniform distribution, each with a probability of getting selected. At the time of data generation, first, a minimum and maximum value set is selected randomly from this list using the probabilities. Then, the attribute value is selected uniformly between the chosen minimum and maximum value. The user also has a provision for representing numeric data by specifying a formula that generates random values [12]. Alexandrov et al. generates numeric data from parametric distributions such as Pareto and Gaussian which is specified by the user [3].

Ada et al. in their work extends KNIME to generate synthetic data [13]. KNIME is an open source data exploration platform equipped with a variety of capabilities

such as data cleansing, modeling, mining, etc. Their work supports the generation of numeric data using four different parametric distributions namely: uniform, Gaussian, beta, and gamma [13]. They also allow numerical attributes to be conditionally dependent on other categorical attributes. In the case of conditional dependence, values of these dependent attributes are picked from a distribution whose parameters are decided based on values of categorical attributes that they depend on [13]. In the case of an independent numeric attribute, values for all records are generated from the same distribution using the same parameter [13].

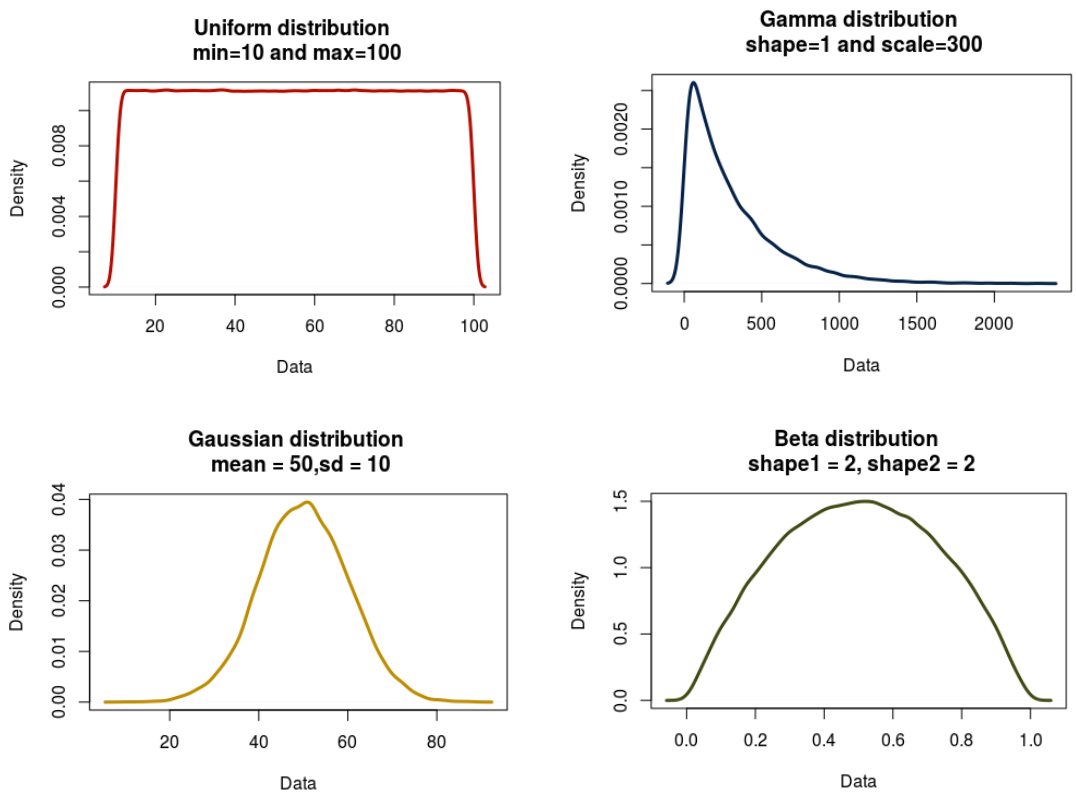


Figure 2: Density of different distributions

The precision of numerical data patterns can be improved by representing them in the form of parametric distributions [3, 12, 13]. Each parametric distribution has different properties, making it suitable for various situations. In the case of a

uniform distribution, the user has to specify minimum and maximum values as parameters. At the time of data generation, values between them is chosen uniformly at random. Gaussian distribution generates continuous values based on the specified mean and standard deviation. Although, a problem with the Gaussian distribution is that it is not a bounded distribution [13]. One way to overcome this is by specifying a global maximum and minimum values. However, doing this might change the mean and standard deviation of the generated data from the specified values. Gamma distribution is one of the flexible distributions among the other parametric distributions. The Gamma distribution is bounded on one side and can be transformed easily into other distributions such as exponential, log and beta. The authors make the life of users easy by allowing them to specify minimum, maximum and peak values as parameters. The shape and scale parameters of gamma distribution is calculated automatically using these values. On the other hand, in case the users want a distribution bounded on both the sides, they can use beta distribution. The parameters of the beta distribution are also calculated using the minimum, maximum and peak values specified by the user [13].

In this work, we make use of the kernel density estimation (KDE) method to represent the patterns of the numeric attributes. KDE is a non-parametric distribution which is capable of representing complex and uneven distribution patterns. KDE is learned from the data provided by the user and it is in turn used to generate data that resembles the real data. This method is explained in more detail in the Section 2.3.

2.2.3 Extracting patterns from timestamp data

In UBA systems, the timestamp is an attribute that is often a potential source of behavioral information. This is because in many cases, the behavior of user may vary on different hours of the day and different days of the week. For example, an employee may have a routine working hours pattern (such as 9 AM to 5 PM) and the intensity of user activities outside this working hours is possibly very less. Similarly, activity intensity of an employee might be more on Wednesday mornings, compared to Friday evenings. It is necessary for UBA systems to identify such patterns to baseline a user's normal behavior. There are several methods used by researchers to generate timestamp attributes in synthetic data. However, it is important for us to select a method that is capable of preserving such behavioral patterns.

One of the straightforward approaches for timestamp data generation includes selecting dates randomly from a set of dates. Alexandrov et. al. generates dates by selecting them randomly from a set of dates with predefined probability [3]. Timestamp data can also be generated by adding a time difference value to a starting timestamp. The time difference value is taken from a distribution of observed time differences [16].

A relevant approach is used by Omari et al. to generate temporal data for market basket data analysis [17]. Through this work, the authors generate synthetic market basket data by considering shopping hours patterns and peak shopping hours. The authors consider two shopping scenarios for data generation: supermarket and e-commerce. Timestamp attributes in these two scenarios are modeled separately. In the case of a supermarket, the timestamps generated is from 8 AM to 8 PM every day (Working hours of the supermarket). Whereas for e-commerce sites, purchases mostly happen during after office hours. Cases like peak hours are

modeled separately for both scenarios. In the case of the supermarket, the peak hours they observed are from Thursday to Saturday, whereas for e-commerce, it is from Friday to Sunday. These constraints and conditions can be specified through parameters before data generation. The number of events over different time intervals is determined from a Poisson distribution [17]. Poisson distribution is a discrete distribution that generates a number of events in a given time interval using an average number of events provided. It assumes that the events are independent of each other [17].

As mentioned earlier, the timestamp is a critical attribute for analysis in many UBA applications. So similar to market basket data analysis, it is necessary for us to extract the patterns of the timestamp attribute carefully. In our work, we follow an approach for timestamp generation which is similar to that followed by Omari et al. [17]. We identify the timestamp patterns individually for each entity (for example users) that are directly dependent on the timestamp. Complex parameters such as the probability of a user being active on a particular day, the number of events for each user per time interval, etc. is extracted and stored from the sample data provided automatically. The detailed algorithm is given in Section 3.1.4.

2.3 Kernel Density Estimation (KDE)

One way to represent the patterns of a numerical attribute is using parametric distributions. Numerical attributes can be represented parametrically by approximating the actual distribution to an existing statistical distribution, with parameters obtained from the sample data. However, in many cases, the distributions of the numerical attributes in a UBA dataset can be multimodal, i.e. distribution with more than one peak. Approximating these multimodal distributions to a parametric distribution could interfere with the user baselines and make the data generated

a noisy one. Trying to fit the exact attribute distribution using multiple parametric distributions is a challenging task. On the other hand, nonparametric distributions avoid any assumptions about the underlying data and estimate the density function of the actual distribution [18].

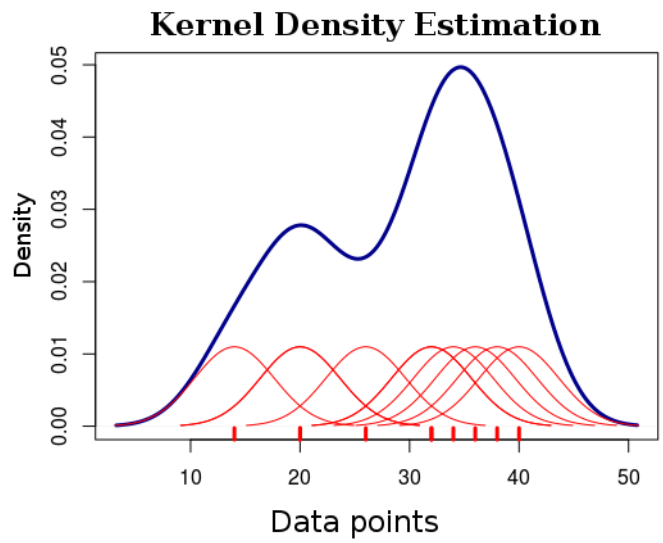


Figure 3: Kernel density estimation

Kernel density estimation (KDE) is a generalized nonparametric technique, that is used to estimate the underlying density, without the need to store the entire data [19]. Probability density function (PDF) using KDE for a set of values $(X_1, X_2, \ldots, X_n)$ is given by

$$f_{KDE}(x) = \frac{1}{n}\sum_{i=1}^{n} K(\frac{x - X_i}{h}) \tag{1}$$

In the above equation, K is the kernel function, which is a symmetric function centered at X_i [20]. Among the several candidates for the kernel function, we use Gaussian kernel for all our experiments. From Equation (1), we can see that the PDF is obtained by averaging the effect of kernel functions centered at each data

point X_i (Figure 3) [18]. The smoothness of KDE function is determined by the chosen bandwidth h. The choice of h has more impact on the shape of density function than the type of kernel used [20]. The equation of the Gaussian kernel we use [18] is given by:

$$K(u) = \frac{1}{\sqrt{2\pi}\sigma} e^{(-\frac{u^2}{2})} \tag{2}$$

The method we use for extracting the distribution of numerical data using KDE is given in Algorithm 1. It takes a list of values X, whose density needs to be estimated, as input. The first step in the process of estimating the density using KDE is to determine the optimal bandwidth, that can best fit the given data. We use Silverman's rule of thumb to estimate the optimal bandwidth. Silverman's rule of thumb is one of the popular technique used for bandwidth estimation, which is also deployed in various tools such as SAS and R [20]. The bandwidth value using Silverman's rule of thumb is given by

$$h = 0.9An^{-1/5} \tag{3}$$

In the above equation, h is the bandwidth, and A is the minimum of the *Standard deviation* and $0.746 * IQR$ (Inter Quartile Range) of the given data sample [20]. From the Equation (3), we can see that the bandwidth asymptotically converges to zero for higher values of n.

The patterns of a numerical data is stored as a set of bins with probability densities estimated using KDE, and the *bin_width*. As the next step, we generate a set of points representing the bins, starting from the minimum value of the data

sample to its maximum value. The distance between the points representing each bin or the *bin_width* is provided by the user. If the user doesn't provide value for *bin_width*, it is chosen to be same as the estimated bandwidth h. The *bin_width* decides the trade-off between accuracy of the pattern extracted and memory used to store this pattern. As the *bin_width* value decreases, the accuracy of the extracted pattern increases. However, this consequently increases the memory required to store this pattern.

Algorithm 1: kernel_density_estimate (X, bin_width)

Input : X - list of numeric values to estimate distribution

Output: h - estimated bandwidth,

densities - density of each bin value

```
#Standard deviation
σ_x := standard_deviation(X);
#Interquartile range
iqr := getQuartileRange(X) ;
A := min(σ_x, 0.746*iqr);
#Silverman's rule of thumb to get bandwidth
h := 0.9 * A * |X|^(-1/5);
if h = 0 then
    return h, {sum(X)/n → 1};
end
#Generate bins
if bin_width is None then
    bin_width := h;
end
bin_values := generate bins of size bin_width from minimum to maximum of
  values of X;
#Get density for each bins
densities := empty map;
foreach _bin in bin_values: do
    densities[_bin] = get_density(value_list,bandwidth,_bin);
end
return bin_width,densities;
```

To calculate the density of the points representing each bin, we use Gaussian kernels centered at each data point of the given data. The standard deviation for each Gaussian kernels is given by the estimated bandwidth h. The probability

density at each bin is obtained by taking the average effect of the kernels at each data point of the sample data (Algorithm 2). Once the densities for all the bins are obtained, this function returns the set of bins with densities and the estimated bandwidth.

Algorithm 2: get_density (X, h, x)

Input : X - list of numeric values to estimate distribution
h - standard deviation of values,
x - the bin point

Output: p - estimated density of the point x

```
prob_sum := 0;
foreach X_i in X do
    u := (x - X_i)/h;
    prob_sum += e^(-0.5*(u^2)) / (sqrt(2*π)*h);
end
p := prob_sum/|X|;
return p ;
```

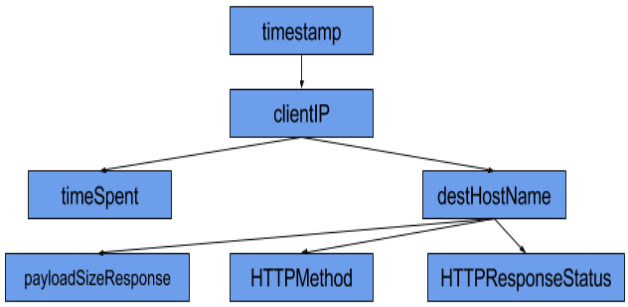
Chapter 3

Methodology

Usability of a synthetic data generator strongly depends on its ability to preserve the patterns of the original data in the synthetic data. Hence, pattern extraction plays a crucial role in the design of synthetic data generator. The first section of this chapter explains the details of the pattern extraction component of the synthetic data generator. The second and third sections give the details of the approach followed in generating synthetic data using the extracted patterns. The second section focuses on a sequential approach followed for data generation, while the third section discusses the approach for generating data in a parallel environment. Following this section, we discuss a special case of UBA where we generate patterns for new synthetic users using the extracted patterns. This allows the data generator to generate synthetic data for new users. For example, one can use synthetic data generator to generate data for 100 users using extracted patterns from a sample data of 10 users. In the last section of this chapter, we discuss the constraints and the assumptions based on which this synthetic data generator is developed.

3.1 Pattern extraction

The pattern extraction component of the data generator takes a real data sample and a list of attribute dependencies as input. The sample data consists of a set of rows and columns, where each row represents separate records of the data, and each column specifies an attribute that defines a feature of each record. The quality of the extracted patterns strongly depends on the sample data, and hence it is important to use proper sample data for pattern extraction. The dependencies between the attributes are provided as a set of *parent-child* relations through the dependency list. The dependency list maps each attribute of the data to its parent attribute(s). The parents of an attribute (child) are the attributes of the given data which it depends on. Table 2 gives attribute dependencies of a sample dataset discussed in Section 4.1.3. Attributes of the data without a parent attribute are considered to be independent attributes.



Attribute	Parent
timestamp	NA
clientIP	timestamp
timeSpent	clientIP
destHostName	clientIP
payloadSizeResponse	destHostName
HTTPMethod	destHostName
HTTPResponseStatus	destHostName

Table 2: Sample attribute dependencies

The process of pattern extraction consists of four important steps. In the first step, we assign levels for each attribute in the data, based on the attribute dependencies collected from the user. During data generation, these levels decide the order of processing these attributes. The order of processing ensures that the value for an attribute is generated only after assigning a value to its parent attribute. After assigning a level to each attribute, in the next three steps we extract patterns

and distributions of each attribute from the real data. The attributes of the dataset are assumed to be in one of the three types: categorical, numeric, or timestamp. Patterns for attributes belonging to each of these types are extracted separately from the real data.

3.1.1 Identification of attribute levels

The dependencies provided as input by the user is used to tag each attribute with a level number. This level number determines the order of processing of these attributes during data generation. Independent columns of the data are assigned to *level 0* or the root level. Each child attribute is assigned a level that comes after the level of its parent attributes. Such an ordering of the levels allows us to generate the data such that an attribute's value is generated only after generating the value(s) for the attribute(s) that it depends on.

Algorithm 3 gives the details of the steps involved in extracting the levels of attributes from the *dependency_list*. As the first step, *dependency_list* provided by the user is used to initialize *attribute_list* - a list of attributes of the given dataset. The level number of the attributes are updated in *levelMap* by iterating through this *attribute_list* multiple times. To avoid endless loop situations, we limit the number of iterations to *max_iter*, which is initialized to the size of *attribute_list*. During each iteration through the *attribute_list*, we check if all the parents of each attribute have a level assigned in the *levelMap*. If an attribute has levels assigned for all its parents, then its level is determined as the highest parent level + 1. Attributes with no parents are assigned level zero. When an attribute is assigned a level, it is removed from the *attribute_list* and is not considered for the future iterations. The iterations are continued until the *attribute_list* is empty (all attributes are assigned a level in *levelMap*) or *max_iter* has reached.

We can see from the algorithm that after each complete iteration through *attribute_list*, the algorithm assigns a level to at least one of the attributes. Hence, after n^{th} iteration through *attribute_list*, at least n attributes will have an associated level. In the worst case scenario, the algorithm takes n iterations for the *attribute_list* of size n resulting in $O(n^2)$ running time. If the *dependency_list* has a cyclic relation (i.e., when a child attribute is parent of a parent attribute), then the algorithm returns error once maximum iteration limit is reached.

Algorithm 3: Level Extraction (*dependency_list*)

Input : *dependency_list*: maps every attribute to its parents set

Output: *level_map*: maps every attribute to its level

```
attribute_list := dependency_list.keys();
max_iter := attribute_list.length();                    # Maximum iterations
iter_count := 0;
level_map := emptymap;
while !(isEmpty(attribute_list)) and iter_count < max_iter do
    foreach attribute in attribute_list do
        parents := dependency_list[attribute];
        if isEmpty(parents) then
            level_map[attribute] := 0;                  # Assign root level
            delete attribute_list[attribute];
        else
            if parents in level_map.keys() then
                level_map[attribute] = get_max_level(parents,level_map) + 1;
                delete attribute_list[attribute];
            end
        end
        iter_count := iter_count + 1;
    end
end
if isEmpty(attribute_list) then
    return level_map;
else
    return NULL;                                        # ERROR: failed to extract levels
end
```

3.1.2 Extraction of patterns from categorical attributes

Extracting patterns from categorical attributes of the data is one of the fundamental steps of our pattern extraction process. The patterns for each categorical attributes are stored in the form of Conditional Probability Tables (CPT). Given its parent attribute(s) value(s), the CPT of an attribute gives the probability of selecting each of its values. The CPT maps each parent values observed in real data, to the probability of occurrence for each value of the attribute. For example, Table 3 below gives the CPT for *HTTPResponseStatus* attribute of Web proxy II dataset (discussed in Section 4.1.3). Given a *destHostname* value (parent attribute), this table provides the probability of selecting an *HTTPResponseStatus* value (child attribute).

Web proxy II data		**HTTPResponseStatus**			
		200	**301**	**302**	**304**
destHostName	**www.nbc.com**	0.93	0.01	0.02	0.04
	tailorednews.com	0.57	0.00	0.214	0.214
	www.ic-live.com	0.50	0.00	0.50	0.00
	ultimatetvshows.com	1.00	0.00	0.00	0.00
	a.tribalfusion.com	0.99	0.00	0.01	0.00

Table 3: Sample Conditional Probability Table (parent attribute:destHostName, child attribute: HTTPResponseStatus)

The probability of occurrence for each value of a categorical attribute is calculated using the relative frequency of that value in the real data. For example, in the given real data, *destHostname* www.ic-live.com occurs 50 times with *HTTPResponseStatus* 200 and 50 times with *HTTPResponseStatus* 302, giving the

probability of selecting these $HTTPResponseStatus$ values as:

$$P(\frac{HTTPResponseStatus = 200}{destHostnames = www.ic - live.com}) = 0.5$$

and

$$P(\frac{HTTPResponseStatus = 302}{destHostnames = www.ic - live.com}) = 0.5$$

The algorithm used for extracting the categorical patterns (described in Algorithm 4) takes real data, a list of categorical columns, dependency list, and level map as inputs. This algorithm can be divided into three parts. As the first step, we create objects of the type $categoricalColumn$ for each categorical attribute. These objects store information related to the attribute such as the attribute name, *parents* of the attribute, *level* number, CPT, and *parent_count* map. The *parent_count* map of an attribute stores the count of each distinct parent value of that attribute in real data. The *parent_count* map is used to obtain the probability of each attribute value from its relative frequency. Initially, the CPT and *parent_count* map of the attribute objects are empty. The *level* and *parents* of each attribute are obtained from the input parameters.

In the next step, the CPT and *parent_count* of all the categorical attribute objects are updated through a single iteration over the real data. The CPT keeps track of the number of occurrences of each attribute value along with its parent value, for each attribute object. In the case of attributes at the root level (no parent), the CPT keeps track of the frequency of its values from the real data. For the rest of the attributes, the CPT records the frequency of occurrence of all observed parent-child value combinations.

In the final step, these frequencies in the CPTs are normalized to obtain corresponding probabilities. To normalize, each frequency value is divided by the count of occurrence of the corresponding parent values (stored in *parent_count* of the attribute object). For the root level attributes, we divide each frequency value by the total number of records in the real data.

Algorithm 4: Categorical pattern extraction(*data*, *categorical_attrs*)

Input : *dependency_list*: maps every attribute to its parent set
data: given data for training
categorical_attrs: list of categorical columns
levelMap: maps every attribute to its level

Output: *categorical_data*: list of categorical attribute objects with their patterns

```
categorical_data := emptylist;
                                                    # Initialize column objects
foreach column in categorical_attrs do
    new_node := new
    CategoricalNode(column,levelMap[column],dependency_list[column]);
    categorical_data[column] = new_node;
end
foreach row in data do
    foreach column in categorical_attrs do
        if isRoot(categorical_data[column]) then
            categorical_data[column].CPT[row[column]] += 1;
        else
            categorical_data[column].CPT[row[parents]][row[column]] += 1;
            categorical_data[column].parent_count[row[parents]] += 1;
        end
    end
end
        # Divide each value in CPT by the count of respective parent
foreach column in categorical_attrs do
    if isRoot(categorical_data[column]) then
        categorical_data[column].CPT = Divide values by the total number
        records;
    else
        categorical_data[column].CPT = Divide the values by the count of the
        parent value;
    end
end
return categorical_data;
```

3.1.3 Extraction of patterns from numerical attributes

This section explains the method we follow for pattern extraction from numerical attributes of the data. Similar to the pattern extraction from categorical attributes discussed in the previous section, the first step is to create objects representing each numerical attribute of the data. Each of these objects store information related to the attribute such as the *name* of the attribute, *level*, *parents*, *bandwidth*, and *CPT*. The CPT and bandwidth of an attribute object gives the information about the distribution of that attribute. This method extracts the distribution of the attribute for each of its parent values, rather than an overall distribution. For instance, in the case of a credit card activity log, where every user has their own spending pattern, this method captures spending pattern for each user.

In the next step, for each distinct parent value of an attribute, we extract a list of all values for that attribute from the real data. This list of values is gathered for all parent values of all numerical attributes in the real data, through a single iteration and stored in *DistinctParentsData* map.

Algorithm 5: Numerical data extraction($dependency\_list, numericcols, data, levelMap$)

Input : *dependency_list*: maps every attribute to its parents set,
numericcols : list of numeric columns in given data
data: Given data
levelMap: maps every attribute to its level

Output: *numeric_data*: list of numeric attribute objects with their patterns

```
numeric_data := emptymap
foreach column in numericcols do
    newcol := new
    numericnode(column, levelMap[column], dependency_list[column]);
    numeric_data[column] := newcol;
end
foreach row in data do
                                    # Obtain the list of values for each parents
    foreach col in numeric_data do
        col_parents := numeric_data[col].parents;
        parents_combination_value := row[col_parents];
        DistinctParentsData[col][parents_combination_value].add(row[col]);
    end
end
foreach col in DistinctParentsData do
    foreach parents_value in DistinctParentsData[col] do
                                                # Extract KDE for given values
        bandwidth, kdevals := kde(DistinctParentsData[col][parents_value]) ;
        numeric_data[col].bandwidth[parents_value] := bandwidth;
        numeric_data[col].CPT[parents_value] := kdevals;
    end
end
return numeric_data
```

As the final step, for each numerical attribute in the dataset, we extract the distribution for its parent value combinations. This is done by passing the list of values for each parent combination from *DistinctParentsData* to a *Kernel Density Estimate* (KDE) function. KDE is a non-parametric method of estimating the numerical data distribution. The *bandwidth* and *kdevals* returned by this function can be used to generate data that follows the same distribution. The *bandwidth* defines the smoothness of the underlying distribution while *kdevals* maps a set of bins to their probabilities. The steps involved in KDE and *bandwidth* estimation are explained in Section 2.3.

For each attribute object, the CPT and bandwidth values are updated by the *kdevalues* and *bandwidth* values respectively, returned for each parent value combination. Finally, this function returns the *numeric_data* - the list of attribute objects, where each object represents a numerical attribute of the data.

3.1.4 Extraction of patterns from timestamp attribute

The timestamp is an important attribute for most of the UBA applications. In the case of *market basket data* analysis, the analysis of transaction timestamps gives us the knowledge of the buying habits of the customers, such as the peak hours of their purchases [17]. In the same way, analyzing the timestamps of user activities in a UBA application gives us an understanding of the temporal patterns of each user. This includes information such as the hours each user is most likely to be active, which in turn can be used to detect their abnormal behaviors.

In our current approach we try to extract the timestamp patterns of the attribute that is directly dependent on the timestamp attribute (child attribute). For example, in the case of Web proxy I dataset explained in Section 4.1.2, our goal is to extract the timestamp patterns for each values of the *host* attribute (child attribute).

The CPT of the timestamp attribute object stores the probability of occurrence of each child value in different time intervals (hours of the week, minutes of the week, etc.). For all our experiments, we consider the hours of the week as the time interval.

As the first step of the timestamp pattern extraction process, we create an object for the timestamp attribute of the data (See Algorithm 6). The CPT and $eventsPH$ are the main components of a timestamp attribute object. The CPT stores the probability of each child value being active in a given time interval., while the $eventsPH$ stores the average number of events/records in that interval for each child value. At the time of data generation, the average number of events from $eventsPH$ is used to select the number of events for an interval using Poisson distribution.

Before we extract timestamp patterns from the data, we convert the timestamp values to week hours. A week hour of a timestamp value represents the hour of the week, starting from 0 (Sunday 12:00-12:59 AM) to 167 (Saturday 11:00-11:59 PM).

As the next step, we calculate the frequency of week hours for each distinct values of the child attribute. The probability of the child values being active during each week hour is obtained by dividing the frequencies of these week hours by the corresponding count of week day in the given data. For example, if a user (child of timestamp) was active two times on a Sunday morning 10-11 AM, and in the date range of the given data, Sunday occurs eight times, then the probability of this user being active on Sunday morning 10-11 AM is 0.25.

Finally, for each distinct value of the child attribute, we calculate the average of the number of records in each week hour. This gives information regarding the density of records in different hours of the week- for example, the patterns such as a user is more active on Monday morning at 11 AM than Friday evening at 6 PM.

Algorithm 6: Timestamp data extraction $(timestampcol, data, childcols)$

```
1  ] Input  : timestampcol: timestamp column
             childcols: attributes dependent on timestamp column
             data: given data
   Output: timestamp_data: timestamp_data containing the distribution
2  timestamp_data := timestampNode(timestampcol,childcols,CPT,EventsPH);
3  days_counts := count of each day of week in the date range of the given data;
4  distinct_children := distinct value of child attribute;
5  data[weekhour] := get_hour_of_the_week(data[timestamp]);
6  foreach value in distinct_children: do
7      subdata := data[child_columns = value];
                                          # Extract frequency of week hours
8      timestamp_data.CPT[value] :=
       extract_probsbility(subdata[weekhour],days_count);
                                 # Extract distribution of event counts in each hour
9      eventsPH_Avg := Average no. of records per week hour;
10     timestamp_data.EventsPH[value] := eventsPH_Avg;
11     return timestamp_data;
12 end
```

3.1.5 Storing the extracted patterns

For each attribute in the real data we have a corresponding attribute object representing the extracted pattern. By storing these attribute objects in a file format we

use extracted patterns to generate data any number of times. This makes the pattern extraction a one-time process. In the current approach, we store the attribute objects in a JSON file. JSON is supported in a variety of programming languages making it easy to store and retrieve attribute objects. Since the data inside the attribute objects stored are in the form of maps, the attribute objects can be easily reconstructed from the JSON file (See Appendix B.1 for attribute patterns stored in JSON file).

The patterns stored in the JSON file captures information stored in each attribute objects such as its name, data type, level, CPT, and parents. The numerical attributes have an additional parameter for storing *bandwidth* from KDE in the JSON file. Similarly, for timestamp attributes the JSON file stores the $eventsPH$.

The pattern file in JSON file is extremely small compared to the real data, but it acts as a its blueprint. Hence, compared to original data, it can be transported easily to different locations to generate synthetic data with the properties of the original data.

The JSON file holding the patterns of the real data can be modified in a variety of ways for different requirements. One such requirement could be to increase the number of users in the synthetic data. For example, from the real data of 100 users, we can generate synthetic data with 10,000 users by modifying the JSON file. This can be achieved by using the patterns of the users from the JSON file to generate additional synthetic users (refer Section 3.4 for details). The pattern file can also be modified to induce anomalous users for testing the analytic models, by altering the distribution of the users.

3.2 Data generation using extracted model

The pattern file extracted from the data is a collection of attribute objects with patterns for each attribute. At the time of synthetic data generation, we reconstruct these attribute objects for each attribute of the data from the pattern file. During the process of data generation, the attribute values of the records are generated in the order of their level numbers. The process of data generation starts from the level 0 which consists of the independent attributes. These attributes at the root level determine the number of records in the synthetic data.

Depending upon the presence and absence of a timestamp attribute at the root level, the input given by the user and the process of generating the root level data differ. In the case where the root level does not include a timestamp attribute, the user provides the number of records that need to be generated as input, where as if the root level contains a timestamp attribute, the user provides the start and end timestamps of the synthetic data as input.

In the case where the timestamp attributes are absent at the root level, we generate n (number of records given as input) partially filled records with values in the root level attributes and other attributes as empty. The root attribute values for each of these records are selected randomly using the values and probabilities in the CPT of the attribute objects.

The process of generating root level data involves more steps if the root level includes a timestamp attribute. The first step, in this case, is to generate all timestamp values between the given start and the end timestamps. In the next step, for each timestamp value generated we identify the values of child attribute that are active during that week hour of the timestamp. A child attribute value during a week hour is decided to be active or inactive based on their probability in

the timestamp attribute CPT. For example, in the case of Web proxy I dataset (discussed in section 4.1.2) where the *host* attribute is dependent on the timestamp, for each timestamp value, we generate all the users who are active on that week hour based on their observed probabilities from the CPT of the timestamp attribute.

Finally, for each child value, active during a timestamp, we find the possible number of records using $eventsPH$ of the timestamp attribute. The number of records for each child value is selected from a Poisson distribution with mean chosen from the $eventsPH$. Based on the number of records for a set of timestamp and active child value, we generate an equal number of partial records with the same timestamp and child attribute values. As the next step, for each partial record, we populate the values for the non-timestamp attributes at the root level.

In the later steps, we complete each of these partial records by populating the values of other attributes using their parent values. The columns are processed in the order of their levels, thereby assuring that the parent values are generated before generating the child attribute values.

Algorithm 7: Data generation($start\_date, end\_date, n$)

Input : *start_date*, *end_date*; start and end dates (if timestamp is a root),
n ; number of records(if there is no timestamp in root)
Output: *bandwidth*; estimated bandwidth, *densities*; density of each bin value

```
                                                    # Check if root has timestamp
if root contains timestamp then
    ts_root := timestamp attribute object of the root;
    time_intervals := generate_timeinterval(start_date, end_date);
    records := list(empty map);
    foreach t in time_intervals: do
        foreach child_val in ts_root.CPT.keys: do
            if child_val is active during dateminute(t) then
                #events := get number of events from ts_root.eventPH;
                for  i from 0 to #events do
                                                    # Generate partial records
                    records.append([ts_root → t, ts_root.child → child_val]);
                end
            end
        end
    end
                                        # Generate values for non-timestamp roots
    foreach r in non-timestamp root do
        foreach rec in records do
            rec[r] := Generate_random_value(r.CPT);
        end
    end
    next_level := 2;
else
    records := Generate n partial records with root level attribute values
    chosen from their CPT;
    next_level := 1;
end
foreach rec in records: do
    foreach level from next_level to leaf do
        foreach column in level do
            rec[column] := Generate_random_value(column.CPT,
            column.parents);
        end
    end
end
generate_csv(records);
```

Algorithm 8: generate_random_value $(CPT, type, parents = None)$

Input : CPT : Extracted CPT from data,
$type$: type of column,
parents : In case of non-root node

Output: $value$: value extracted from CPT

```
if parents = None: then
    value := generate value randomly from CPT using probabilities
else if type = categorical then
    value := generate value randomly from CPT[parents] using probabilities
else
    bandwidth := bandwidth[parents];
    bin_val := generate value randomly from CPT[parents] using
    probabilities;
    value := generate value randomly between bin_val and
    bin_val + bandwidth;
return value
```

The categorical attribute values are generated using the CPT of the corresponding categorical attribute object. Depending upon the parent values, a value is selected randomly for the child attribute from a list of values with their probabilities. Each value of the parents has a separate list of values in the CPT of the attribute object. In the case of the numerical attributes, based on the parent attribute values, a bin value and bandwidth is selected respectively from the CPT and the bandwidth parameters of the associated attribute object. Its final value is chosen uniformly randomly from bin value to bin + bandwidth value.

3.3 Parallel data generation

The synthetic data can be generated in parallel in a natural manner as the generation of each record of the synthetic data is independent of the other records. The only constraint that needs to be considered is the attribute dependencies, i.e. the parent attributes must be generated before generating the child attributes.

We designed a parallel approach that allows the process of data generation to scale with the available number of processors. This parallel process is designed following the work-time framework, enabling it to be formulated as a sequence of time blocks, independent of the processors [21]. The steps within each time block are executed concurrently. In our case, the process of data generation involves a sequence of l time blocks where l is the number of levels. Data for attributes at the same level is generated in parallel for all the records, using the available processors.

In the process of parallel data generation, first, we generate data for attributes at level 0. This step involves two cases: root level without a timestamp attribute and root level with a timestamp attribute. In the first case, n (number of records given as input) partial records are generated with values populated for the root attributes. The values for these attributes are randomly generated using their corresponding attribute object CPTs.

Algorithm 9: Parallel data generation $(START\_DATE, END\_DATE, n)$

```
Input  : START_DATE : Start date (In case root is timestamp)
         END_DATE: Start date (In case root is timestamp)
         n: Required record count (In case root is not timestamp)
         model: Map with levels as keys and attributes in the level as value
Output: records; synthetic data records
1  root := model[0];
2  if root contains timestamp attribute: then
3      tmp_records = generate_timestamps(START_DATE,END_DATE);
       pardo :
               foreach timestamp in tmp_records: do
                   records := create partial records with timestamp and
           generate active child values;
                   foreach row in records: do
                       update row by randomly generating other root attribute
               values;
                   end
               end
4      current_level := 1;
5  else
6      records := emptylist;
7      for count from 1 to n: do
8          row := generate values for each attribute in root randomly;
9          records.append(row);
10     end
11     current_level := 0
                                   # Generate values of other attributes for partial records
12 foreach level greater than current_level: do
       pardo :
               foreach row in records do
                   row := row.append(get values for columns in level)
               end
13 end
```

In the case where the root level contains a timestamp attribute, first, all possible values of timestamps are generated between given start date and end date. Next, using the same approach discussed in Section 3.2, we generate partial records with values for attributes in level 0 and level 1. Once we have the partial records, we generate the values for the attributes at subsequent levels. Within each level, the records are processed in parallel using the available number of processors. The levels are processed in an increasing order to assure that the parent values are generated before the generating the values of child columns. The completed records are written to CSV files directly from each processor thereby avoiding unnecessary network operations.

Let the total amount of work involved in generating synthetic data with n records is $W(n)$. If each record of the data has c attributes and assuming each attribute can be generated in a constant time, the total time required to complete the work $W(n)$ is $O(nc)$. Our parallel approach divides this work into a set of l time units, where l is the number of levels. Using p processors, each level $i < l$ can be executed in $O(\frac{W_i(n)}{p})$ time, where $W_i(n)$ is total work in i^{th} level. Using the work-time scheduling principle, the total time for completing W(n) using p processors can be expressed as

$$
\begin{aligned}
T_p(n) &\leq \sum_{i=1}^{l} \left\lceil \frac{W_i(n)}{p} \right\rceil + k \\
&\leq \frac{W(n)}{p} + k + e \\
&\leq O(\frac{nc}{p} + k)
\end{aligned}
$$

The value of k incorporates time taken for initialization, synchronization, and

other intermediate processes. From the above, we can see that the speed of data generation increases linearly with the number of processors. This shows that the data generation process can be scaled depending upon the available number of processors. The experiments conducted to verify this are explained in Section 4.3

3.4 Generating additional users

This section explains a special use case of UBA where the goal is to add additional users to synthetic data than those that are present in the given data. For example, suppose the given real data have only information about ten users, and we need to test the performance of our system for handling data of 1000 users or more. Such cases demand the creation of synthetic users whose behavior blends with that of other users. This use case can be addressed with our system in multiple ways.

Random sampling of user profiles

A straightforward approach is to add additional users by uniformly randomly picking users from our extracted model and duplicating the user data. This leads to the generation of synthetic users following behaviors of existing users. For example, if we want to generate 100 new *clientIP*s from the four *clientIP* of Web proxy II dataset, we create each new *clientIP* by selecting a *clientIP* randomly from the four *clientIP*s and copying the distribution of *timeSpent*, *destHostName*, and *timestamp* for the new *clientIP* from the corresponding distributions of the selected *clientIP*. At the time of data generation, since data for each *clientIP* is generated randomly using the extracted probabilities, it is less likely for two *clientIP*s to have same data. This will indeed provide better metric during performance testing than just duplicating the records of the data.

Combine user behaviors

The second approach is to create new synthetic users by uniformly randomly selecting the behavioral patterns (distribution of related attributes) from all the users. That is instead of replicating the behavior of a single user, we construct a user by

randomly selecting distributions of each directly related attributes from the distributions of all the users. For example, in the case of Web Proxy II dataset (discussed in Section 4.1.3) each $clientIP$ is directly related to the $timestamp$, $timeSpent$, and $destHostName$ attributes. A new clientIP can be created from three existing clientIPs by selecting the $timestamp$ distribution from the first, $timeSpent$ distribution from the second and the distribution of $destHostName$ from the third clientIPs. This approach allows us to generate users with n^a different behaviors, where n is the number of users in the original data and a is the number of related attributes.

Generate users from prior distribution

This approach is based on learning the prior distribution of the dependent attributes of a user. The parameters of the attribute distribution of new users is selected from the prior distribution of that attribute. For example, instead of storing the distributions of events per hour for each user as bins and probabilities, we can assume that the numbers for each user follow a Poisson distribution and store the average number of events per hour. From the average number of events per hour of all the users, we can learn the mean and the standard deviation of a Gaussian distribution (prior distribution of Poisson). While generating a new user, we can select the average number of events per hour from the Gaussian distribution.

3.5 Constraints and Assumptions

In this section some of the assumptions and constraints that we followed while developing the synthetic data generator are described.

1. In our current approach, to keep the process simple, we are assuming that no attributes depend on any of the numerical attributes.

2. If the timestamp column is present, we assume it to be present at the root. This is because, the events are typically synchronized with the timestamp.

3. We assume that the end user provides enough data to extract the required patterns. The quality of the generated data is directly impacted by the quality of the given data.

Chapter 4

Experiments and results

This chapter discusses the experiments conducted to test and validate our data generation approach. The main objective of these experiments is to verify that the generated synthetic data preserves the patterns present in the original data fed to the synthetic data generator. The first section of this chapter gives the details of the experimental setup, and the three datasets used in our experiments. The next section compares the distributions of these datasets with their synthetic counterparts, generated using the synthetic data generator. In the last section, the scalability of the data generation process is discussed.

4.1 Experimental Setup

All the experiments in this chapter involved three steps: pattern extraction, data generation, and data validation. As the first step of the experiment, the distributions and patterns were extracted from the real data and were stored in a pattern file in JSON format, using the algorithms discussed in Chapter 3. As the next step, we generated the synthetic data using this pattern file. Finally, the distributions and patterns extracted from the generated data were validated against that of real data.

This system for data generation, with all the algorithms for pattern extraction and data generation discussed in Chapter 3, was implemented using Python 2.7. The experiments were executed on a Linux platform (Intel core i5 processor, 8G RAM). The parallel synthetic data generation experiments were performed on multiple machines from Amazon Elastic Compute Cloud (Section 4.3.2). The visualization and validation of the synthetic data was implemented in R language (v3.2.5).

Our data generation approach was validated using three different datasets: Iris dataset, NASA Apache web log, and a publicly available Web proxy log dataset. The characteristics of these datasets are discussed in detail in the following subsections.

4.1.1 Iris flower dataset

The Iris flower dataset is a publicly available dataset that is widely used by statistics and machine learning researchers. This dataset is used to test the distributions of generated numerical data and to show that this method of data generation is not limited to UBA logs. The Iris dataset contains values of *sepal length*, *petal length*, *sepal width*, and *petal width*, for different classes of Iris flowers. This dataset has 150 observed samples each with five attributes- the four features of the flower (mentioned above) and a *class* column indicating the class of the flower. Among these five attributes, *class* is a categorical attribute, while the rest all are numerical. Before running experiments on this dataset, we assumed that each attribute (*sepal length, sepal width, petal length*, and *petal width*) is dependent on the *class* of the flower (Figure 4). Any dependencies or correlation between numeric attributes were ignored as it is not within the scope of this work.

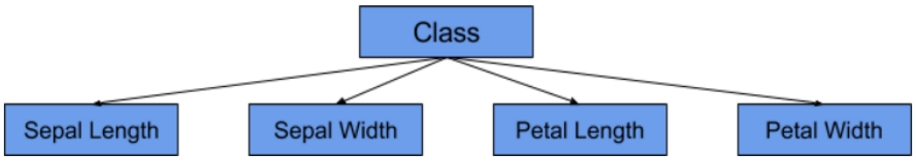


Figure 4: Attribute dependencies - Iris flower dataset

class	sepal_length	sepal_width	petal_length	petal_width
Iris-setosa	5.1	3.5	1.4	0.2
Iris-versicolor	7	3.2	4.7	1.4
Iris-virginica	6.3	3.3	6	2.5
Iris-versicolor	5.6	3	4.1	1.3
Iris-setosa	5	3.5	1.6	0.6

Table 4: Sample Iris dataset (lengths and widths are in cm)

4.1.2 Web proxy dataset I

Web proxy log data is an integral part of the network forensics. It is used to identify a variety of threats such as malicious insiders, anomalous browsing activities, etc. The first web proxy dataset we used for our experiments is public 1995 NASA Apache web log data [22]. This data consists of browsing activity logs for a period of one month (01-07-1995 to 28-07-1995). Each record describes an activity and consists of a *host* that makes the request, *time* attribute representing timestamp of the activity (in epoch time format), *url* of the request, *method* (HTTP method used for the request), *response* (HTTP response code) and *bytes* downloaded. Each record in the original data also contains a *logname* attribute with no value set and

hence discarded from our experiments. The primary goal of these experiments was to test the synthetic data generation process and analyze the patterns in synthetic data. Hence, we considered only a subset of the original data for our experiments. From the original data, we filtered the activities of five different *hosts*, and used this data subset for our experiments.

time	host	url	bytes	method	response
804571902	ottgate2.bnr.ca	/shuttle/countdown/count.html	73231	GET	200
804572077	ottgate2.bnr.ca	/shuttle/countdown/countdown.html	0	GET	304
804574705	bettong.client.uq.oz.au	/images/USA-logosmall.gif	234	GET	200
804576484	bettong.client.uq.oz.au	/cgi-bin/imagemap/countdown?108,145	96	GET	302
804588159	romulus.ultranet.com	/history/apollo/apollo-13/apollo-13.html	0	GET	304
804837529	cliffy.lfwc.lockheed.com	/cgi-bin/imagemap/countdown?102,175	110	GET	302

Table 5: Sample Web proxy I log dataset

In the Web proxy I dataset records (as shown in Table 5), the *time* decided activities of each *host* in the network. The *url* and the *bytes* attributes were assumed to be directly related to the *host* values. This was based on the assumption that a typical *host* will have a set of *urls* that it frequently requests. The *method* and *response* attributes were decided by the *url* of a given record. Figure 5 depicts the dependency diagram considering these relations between the attributes within a record that we have discussed.

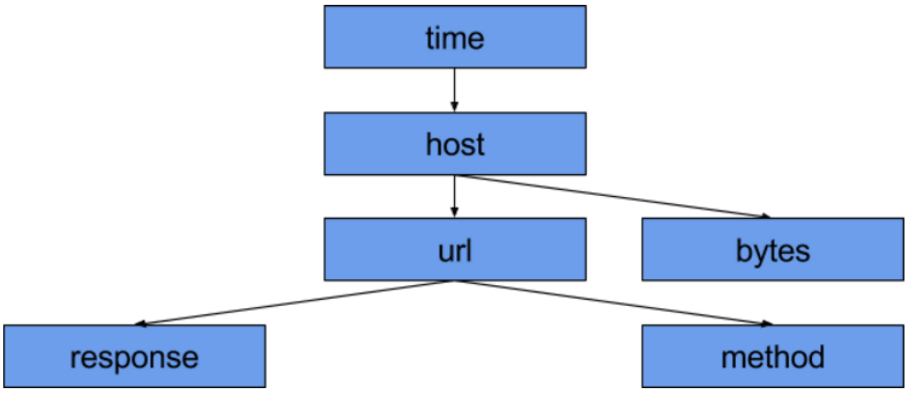


Figure 5: Attribute dependencies - Web Proxy I dataset

4.1.3 Web proxy dataset II

The second log dataset that we used for our experiments is a publicly available Web proxy log dataset [9]. With the prime focus on testing our synthetic data generation methods, we took a subset of this dataset for our experiments. From the set of different attributes of this dataset, we identified the following attributes as the characteristic features of this dataset: *timestamp*, *timeSpent*, *clientIP*, *HTTPResponseStatus*, *payloadSizeResponse*, *HTTPMethod*, *destHostName*. We assumed *clientIP* to be the key subject whose activities are dependent on *timestamp*. The other attributes described the activities of the user such as the destinations he/she accessed, bytes downloaded, HTTP method used, etc. Also, in order to reduce the noise, instead of selecting all the *clientIP*s, we focused on a set of four *clientIP*s who were active for a week between April 27, 2005, and May 6, 2005.

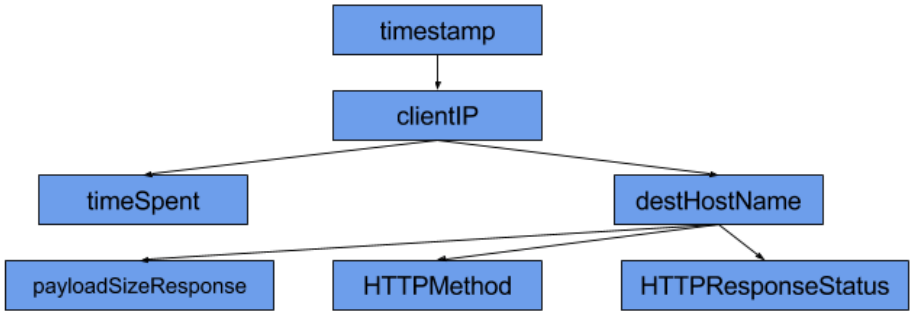


Figure 6: Attribute dependencies - Web Proxy II dataset

Figure 6 depicts the relations in this dataset that we considered for our experiments. *clientIP* was assumed to be dependent on *timestamp*, and each *clientIP* was assumed to have their own distribution of *timeSpent* and *destHostName*. Finally *payloadSizeResponse*, *HTTPMethod* and *HTTPResponseStatus* were assumed to be dependent on the *destHostName* accessed.

timestamp	timeSpent	clientIP	httpResponseStatus	payloadSizeResponse	HTTPMethod	destHostName
2005-04-28T20:27:34	12636	45.0.0.93	200	341	GET	pgq.yahoo.com
2005-04-28T15:15:02	150	45.0.0.191	200	267	HEAD	download.windowsupdate.com
2005-04-28T15:17:03	3	45.0.0.191	503	1736	GET	toolbar.google.com
2005-04-28T15:17:34	3	45.0.0.191	503	1736	GET	toolbar.google.com

Table 6: Sample Webproxy II log dataset

4.2 Results and Analysis

In this section, we analyze the synthetic data generated using the above three datasets, and compare their distribution to their original counterparts. The distributions are analyzed separately for the different types of attributes, *viz.* numerical, categorical and timestamp. First, we analyze the distribution of numerical attributes of all the three datasets, then the distribution of categorical attributes is compared. Finally, we study the distribution of timestamp for the Web proxy log datasets.

4.2.1 Analysis of the numerical data distribution

Let us begin by analyzing the numerical attributes of synthetic data generated for Iris dataset. As stated before, each of the four numerical attributes of the flower (length and width of sepal and petal) are assumed to be dependent on the class of the flower. There are three distinct class of flowers in the real data, namely, *Iris-setosa*, *Iris-versicolor* and *Iris-virginica*. The combination of these three classes and four attributes resulted in the twelve different distributions that we have captured from the real data. These distributions were extracted using Algorithm 5 of Chapter 3 (i.e. distribution of each attribute for each class of the flower). This algorithm extracted the non-parametric distributions for each of these twelve combinations from the data, using KDE. The extracted distribution of an attribute mapped each parent value (*class* of Iris flower) to a set of bin values with probabilities and a bandwidth. For generating a value for a numerical attribute, based on the value of its parent attribute, a bin was selected at random using the extracted probabilities. Once a bin was chosen, a value between bin and bin + bandwidth was selected randomly, and assigned to the attribute.

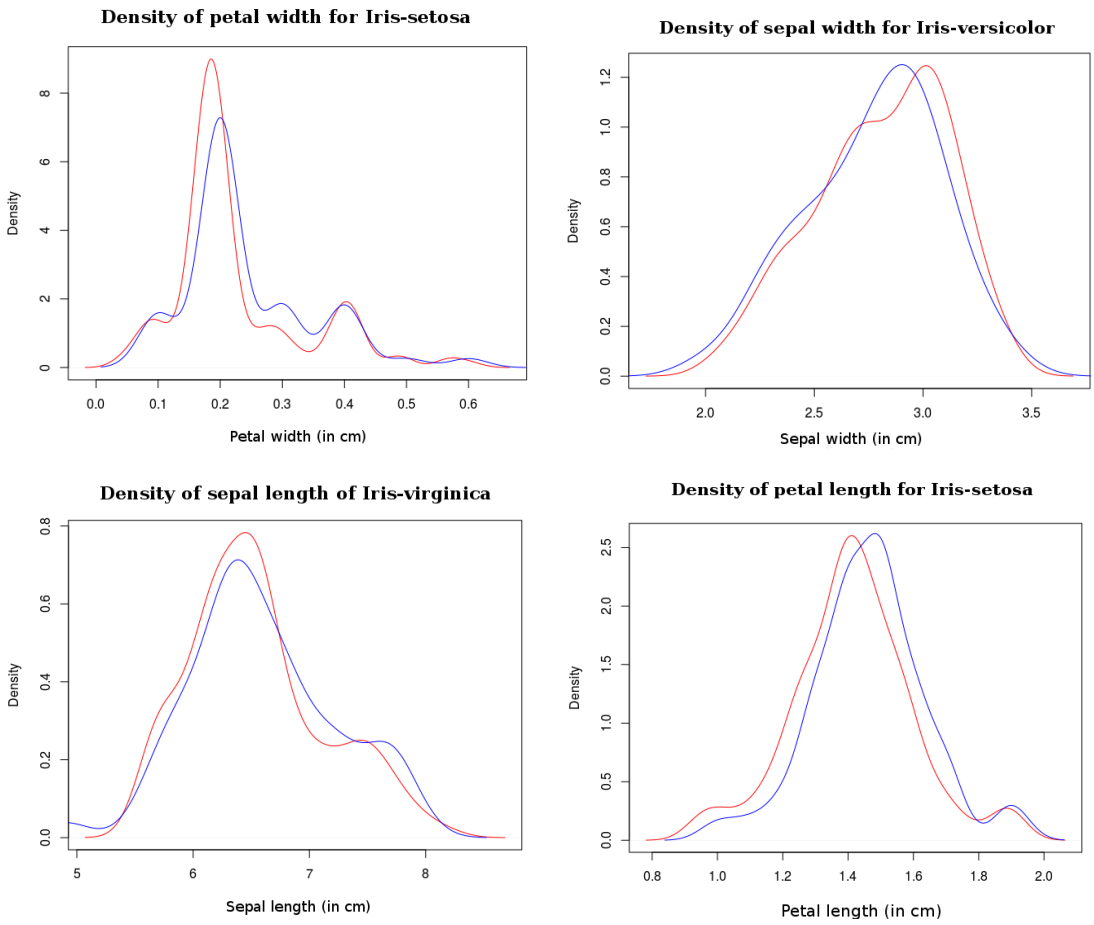


Figure 7: Sample distribution of Iris data *(Red: Synthetic, Blue: Real)* It can be observed that the synthetic data is able to reproduce the distributions from real data

Figure 7 shows the distributions of real (blue) data and synthetic (red) data, of four sample *attribute - class* combinations. It can be observed that the synthetic data closely follows the distribution of the real data. Refer Appendix A.1 for more graphs demonstrating the remaining attributes and classes.

As the next step, let us focus on the parameters characterizing the distribution of the numerical data. These parameters are used to compare the distribution of numerical attributes in the real and synthetic data. The parameters we selected are the mean and standard deviation of the data along with the quartiles. The mean

and standard deviation give an idea about the overall spread of the data, while the quartiles give us the information about the skewness of the data and regions with maximum densities. For example, Q3 or the third quartile gives the value at 75th percentile, that divides lowest 75% of the attribute values from the highest 25%. If a column has values ranging from 0 to 100 and the value of Q3 resides at 10, this means that 75% of data is less than 10 and the distribution is skewed towards 0.

Class	Attributes	Min		Q1		Q2		Q3		Max		Mean		SD	
		Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn
Iris-versicolor	petal length	3.00	3.29	4.00	3.97	4.35	4.38	4.60	4.63	5.10	5.14	4.26	4.30	0.47	0.44
Iris-versicolor	petal width	1.00	0.91	1.20	1.17	1.30	1.34	1.50	1.49	1.80	1.85	1.33	1.33	0.20	0.22
Iris-versicolor	sepal length	4.90	4.65	5.60	5.51	5.90	5.80	6.30	6.19	7.00	7.18	5.93	5.86	0.52	0.53
Iris-versicolor	sepal width	2.00	2.02	2.53	2.59	2.80	2.83	3.00	3.04	3.40	3.39	2.77	2.80	0.31	0.31
Iris-setosa	petal length	1.00	0.94	1.40	1.30	1.50	1.40	1.58	1.50	1.90	1.90	1.46	1.41	0.19	0.17
Iris-setosa	petal width	0.10	0.05	0.20	0.17	0.20	0.19	0.30	0.26	0.60	0.60	0.24	0.23	0.35	0.36
Iris-setosa	sepal length	4.30	4.15	4.80	4.73	5.00	4.96	5.20	5.19	5.80	5.85	5.00	4.96	0.35	0.36
Iris-setosa	sepal width	2.30	2.24	3.13	3.09	3.40	3.35	3.68	3.74	4.40	4.44	3.42	3.42	0.39	0.38
Iris-virginica	petal length	4.50	4.45	5.10	5.03	5.55	5.48	5.87	5.79	6.90	7.13	5.55	5.47	0.55	0.54
Iris-virginica	petal width	1.40	1.32	1.80	1.82	2.00	1.99	2.30	2.19	2.50	2.54	2.03	2.00	0.27	0.26
Iris-virginica	sepal length	4.90	5.50	6.23	6.16	6.50	6.49	6.90	6.85	7.90	8.10	6.59	6.55	0.64	0.58
Iris-virginica	sepal width	2.20	2.04	2.80	2.73	3.00	2.96	3.17	3.15	3.80	3.78	2.97	2.94	0.32	0.30

Table 7: Comparison of distribution - real vs. synthetic Iris data. This table provides the mean, standard deviation, minimum, maximum and three quartiles of each class-feature combination. It can be observed that synthetic data values are closely aligned with that of the real data.

Table 7 shows these values for each attribute-class relation. It can be seen from this table that these values are very close to each other for all the parameters indicating that the synthetic data preserved the distribution of numerical attributes present in the real data.

Next let us analyze the distribution of bytes downloaded by each host from Web proxy I dataset. The parent values of *bytes* attribute are the five distinct values of *host* in Web proxy I data.

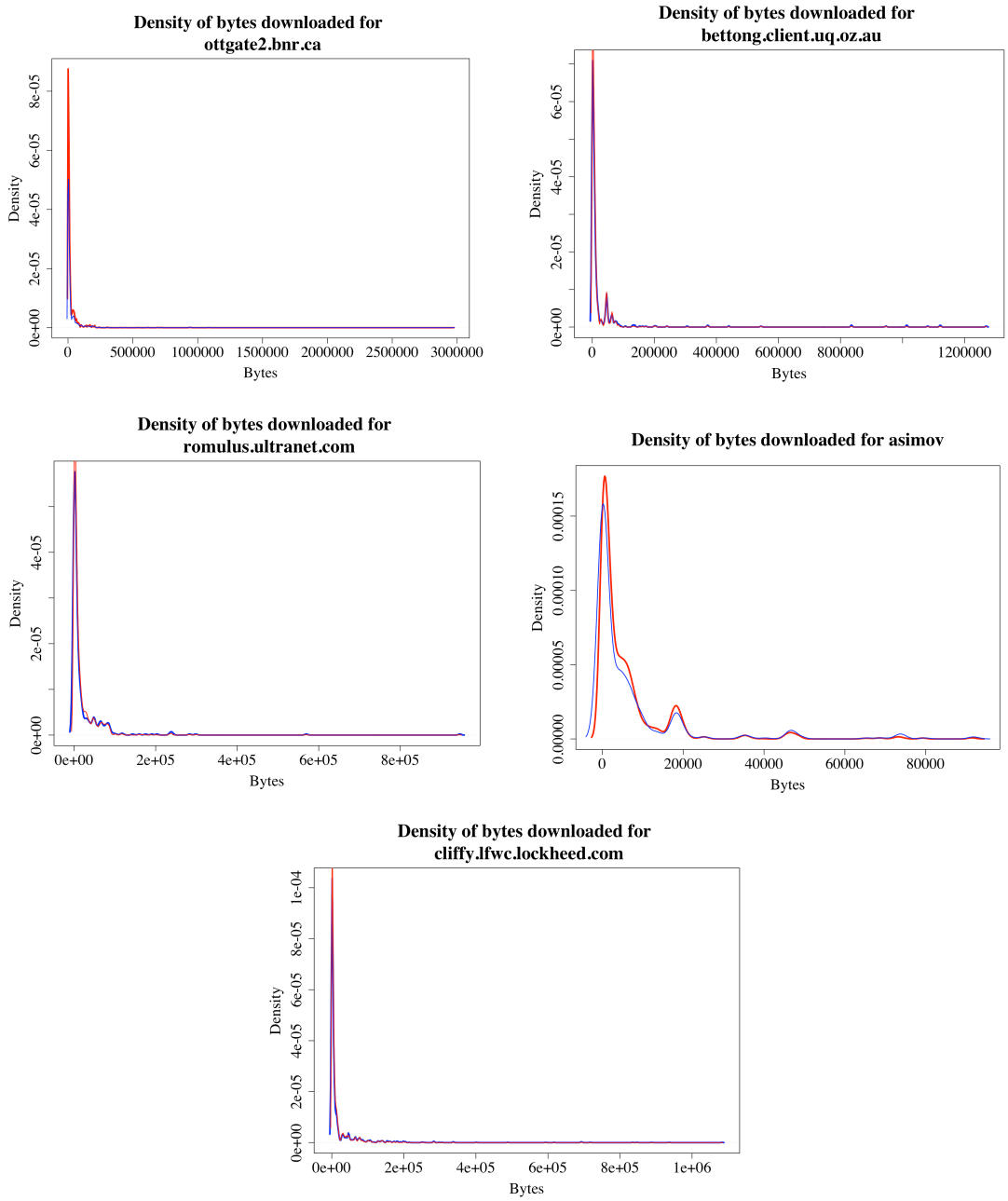


Figure 8: Distribution of *bytes* by each *host (Red: Synthetic, Blue: Real)*

Host	Min		Q25		Q50		Q75		Max		Mean		SD	
	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn
ottgate2.bnr.ca	0	0	1204	2011	4843	5792	17459	16252	2973350	2971190	29614	21936	113596	66928
bettong.client.uq.oz.au	0	0	786	1555	4535	4873	12229	10830	1269716	1267818	33612	18851	133372	76516
romulus.ultranet.com	0	0	111	146	4179	5284	18114	17007	946425	943662	22764	17273	63290	36501
asimov	0	0	0	163	1016	2179	6777	6751	91829	91786	6737	6147	14116	10866
cliffy.lfwc.lockheed.com	0	0	527	970	2146	3060	12008	9928	1082916	1081381	23981	15932	77925	47708

Table 8: Distribution parameters of *bytes* for *host* values of Webproxy I data

Table 8, compares the parameters characterizing the distribution of *bytes* attribute of Web proxy I dataset. From this table, we can see that the values for each parameter in the real and synthetic data are very close, considering the range of bytes downloaded by each host.

Next let us study the distribution of *timeSpent* attribute of Web proxy II dataset. Based on the assumption that the time spent is dependent on the *clientIP*, let us compare the distribution of the *timeSpent* in real and synthetic data for each *clientIP*. The distribution data of time spent is shown in Figure 9 and Table 9. Please note that, since the distribution was skewed towards zero, the plots display densities of the *timeSpent* that is less than 1000.

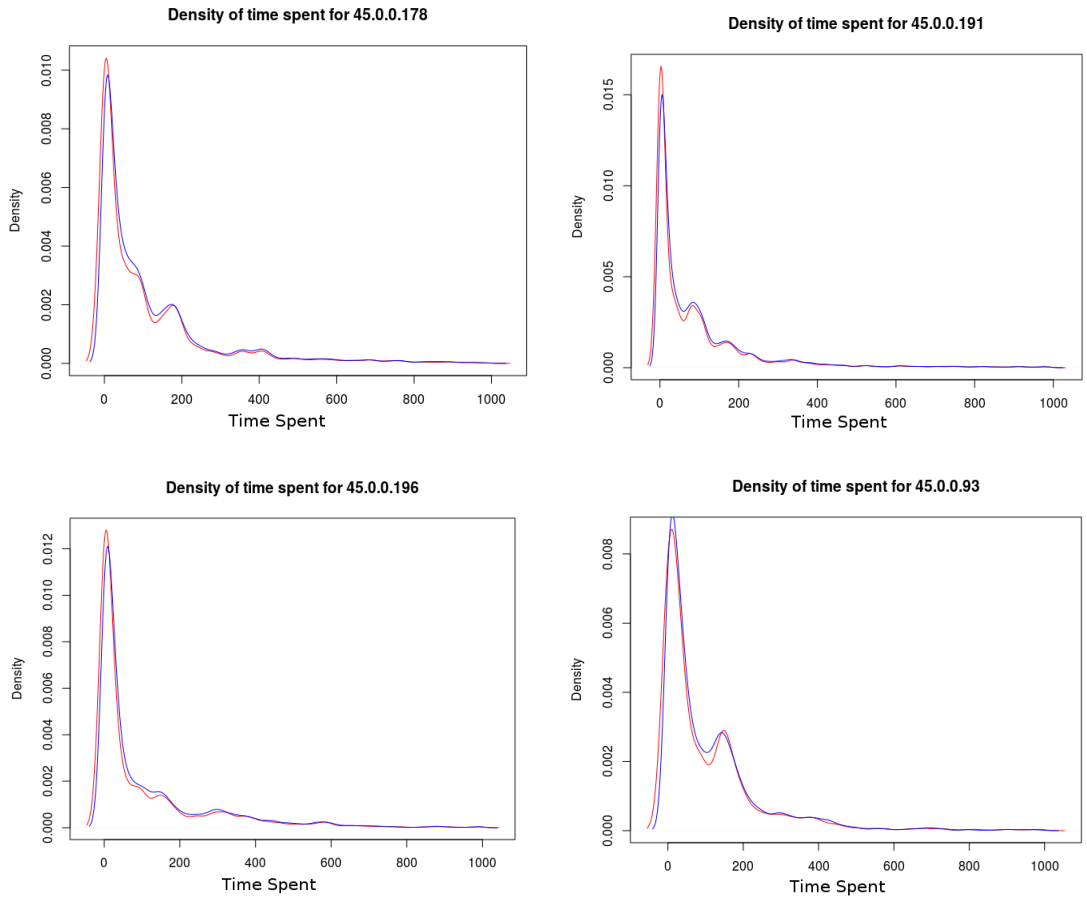


Figure 9: Distribution of *timeSpent* by each *clientIP (Red: Synthetic, Blue: Real)* Since time spent is skewed towards zero, the plot displays distribution of time spent <1000

Client IP	Min		Q1		Q2		Q3		Max		Geometric Mean		SD	
	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn
45.0.0.178	1	0	2	13	50	62	164	172	360060	360044	26.08	41.78	6,609.56	4,038.13
45.0.0.191	1	0	2	8	34	45	126	134	359790	359792	21.02	31.86	8,354.93	6,212.62
45.0.0.196	1	0	2	8	23	34	146	157	360093	360098	18.87	30.95	14,559.16	10,339.94
45.0.0.93	1	0	2	16	45	57	155	155	360012	360009	32.19	43.22	27,661.38	22,044.20

Table 9: Distribution parameters of *timeSpent* (in seconds) for different *clientIP*s of Webproxy II data

From the above plots (Figure 9), we can see that the generated data is able to capture the distribution of the *timeSpent* very closely. From Table 9, we can see that the quartiles for each client IP are close to each other, given the fact that the values range from 0 to 360,000. This shows that the synthetic data has successfully captured the distribution pattern of *timeSpent* present in the real data.

The *payloadResponseSize* is another numeric attribute of Web proxy II data. The *payloadResponseSize* was assumed to be depended on the value of *destHostName* attribute. Since number of unique values in the *destHostName* attributes were very high, there were several parent-child combinations. Hence we took a subset of parent values and corresponding parent-child combinations from the real data for our analysis.

Please note that in the case of payload response size of Web proxy II data, the data ranged from zero to order of ten thousands. However, it was skewed with 75% of data less than 2000 for most of the destinations. Considering this skewness and the range of the data, Table 10 indicates that the parameters of distribution are very close to each other for both real and synthetic data.

Destination	Min		Q1		Q2		Q3		Max		Geometric Mean		SD	
	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn	Orig	Syn
webmail.att.net	0.00	0.00	369.00	357.00	436.00	343.00	794.00	570.00	59,434.00	59,209.00	631.19	525.60	9,818.67	6,815.24
i.a.cnn.net	0.00	0.00	231.00	224.00	232.00	249.00	678.00	392.00	64,456.00	64,905.00	464.88	363.22	4,245.89	3,134.92
a111.g.akamai.net	226.00	187.00	485.00	526.00	1,282.00	1,101.00	1,905.00	1,873.00	58,181.00	58,287.00	1,198.93	862.89	5,797.87	2,836.50
64.4.55.45	186.00	21.00	1,714.00	1,680.00	1,948.00	1,938.00	2,254.00	2,204.00	28,618.00	28,480.00	2,048.48	1,946.60	5,500.39	4,527.28

Table 10: Distribution parameters of *payloadSizeResponse* for sample *destination*s of Webproxy II data

The above analysis of obtained results clearly indicates that the distribution

of numerical attributes in the real and synthetic data are similar, for each parent value. In other words the synthetic data generated preserved the numerical patterns present in the given original data. Hence, it is reasonable to conclude that our approach is able to capture more realistic patterns from numerical data attributes.

4.2.2 Analysis of categorical data distribution

Next, let us analyze the distribution of categorical attributes in synthetic data and compare it with the same for the real data. We begin our analysis with categorical attribute of the Iris dataset. In the case of the Iris dataset, the only categorical attribute is its class. According to our assumption, the class is an independent attribute and all other attributes are dependent on the class values. So we compared the proportion of record counts for each class value in the synthetic data with that of the real data.

Class	Record count in real data	Expected count	Record count in synthetic data	Percentage difference
Iris-setosa	50	166.67	152	9.2%
Iris-versicolor	50	166.67	174	4.3%
Iris-virginica	50	166.67	174	4.3%

Table 11: Record counts in real and synthetic Iris data

Comparison of record counts in real and synthetic Iris data for each class of flower. Number of records in class in real and synthetic data

The real data had a total of 150 records, with 50 records from each class of the

Iris flower. The distributions of categorical attributes were extracted using Algorithm 4 of Chapter 3, in which the frequencies of distinct values of an attribute were extracted from the real data and converted into probabilities. These distributions extracted from real Iris data were then used to generate random class values for 500 synthetic records. The expected count of each class in 500 records is 166.67, obtained by scaling the number of records per class in the original data to the size of the synthetic data. As we can see from Table 11 this actual count of class values in the synthetic data is close to this expected value. The deviation of the actual count from the expected count is less than 10% for all three classes.

Next, let us study the distribution of categorical attributes in Web proxy I and Web proxy II datasets. The values of categorical attributes are conditionally dependent on other attributes in these datasets. Hence, it is necessary to compare the counts of values in each parent-child relations of real and synthetic data for these datasets. Sample values in *host-url* relations of Web proxy I and *clientIP-destHostName* relations of Web proxy II datasets are shown in Table 12 and Table 13 respectively. Since the number of unique combinations of values in parent and child attributes of these datasets were high, we calculated the normalized root mean squared error (NRMSE) for these attributes. NRMSE gives an estimate of the differences in distributions, by comparing the observed and expected counts. The observed count of a value in our context is the frequency of that value in the synthetic data, and the expected count is calculated by scaling the frequency of the same value in the real data to the size of synthetic data. Table 12 and Table 13 show the observed and expected counts of sample *host-url* relations of Web proxy I and *clientIP-destHostName* relations of Web proxy II datasets.

host	url	Observed Count	Expected Count
cliffy.lfwc.lockheed.com	/images/KSC-logosmall.gif	454	519.5
cliffy.lfwc.lockheed.com	/images/NASA-logosmall.gif	361	390.8
ottgate2.bnr.ca	/images/NASA-logosmall.gif	279	282.0
bettong.client.uq.oz.au	/shuttle/countdown/	105	113.8

Table 12: Observed and Expected counts of sample values in host-url relation (Web proxy I dataset)

clientIP	destHostName	Observed Count	Expected Count
45.0.0.191	news.google.com	2056	2028.6
45.0.0.178	www.comics.com	798	782.9
45.0.0.191	espn.go.com	333	338.1
45.0.0.178	www.nytimes.com	77	78.2

Table 13: Observed and Expected counts of sample values in clientIP-destHostName relation (Web proxy II dataset)

Next let us calculate the normalized RMSE for all categorical parent-child relations in both the datasets. Normalized RMSE is calculated using the following formula [23]:

$$NRMSE = \frac{\sqrt{\frac{\sum_{i=1}^{n}(X_o^i - X_e^i)^2}{n}}}{max(X_o) - min(X_o)} \times 100 \tag{4}$$

Here, n is the total number of distinct combinations of a parent-child relation.

For example, in the case of *url* attribute of Web proxy I, n is the total number of unique *host-url* combinations (*host* is the parent of *url*). X_o^i is the observed count of ith parent-child combination in synthetic data, and X_e^i is the expected count of the same calculated from the real data. The normalized RMSE calculated for all the categorical attribute relations of Web proxy I and Web proxy II, are provided in Table 14.

Parent-child relation	Normalized RMSE
host,url	1.2%
url,method	0.61%
url,response	0.59%

Parent-child relation	Normalized RMSE
destHostName, clientIp	0.3%
httpMethod,destHostName	0.29%
httpResponseStatus, destHostName	0.4%

Table 14: Normalized RMSE for each categorical relations in Webproxy I (left) and Webproxy II (right) datasets.

As we can see from this table the maximum NRMSE among all the relations in these datasets is 1.2%. These results clearly indicate that our approach for synthetic data generation is capable of preserving the patterns followed by categorical attributes in the real data.

4.2.3 Analysis of timestamp data distribution

Next, let us analyze the distribution of the records in the synthetic data across different values of its timestamp attribute. Among the three datasets, timestamp attributes are present in Web proxy I and Web proxy II datasets. While analysing the timestamp attribute, we need to make sure that for each value of its child, the distribution of the records across the timestamp is preserved. For example, in the case of Web proxy I data, for each host (child of timestamp attribute), the distribution of the activities across the time intervals should match the corresponding distributions in the real data.

Distributions of timestamp attributes were extracted from the real data using Algorithm 6 of Chapter 3. The extracted patterns included the probabilities of each value of its child column (e.g., each host in case of Web proxy I) being active during each hour of the week. The patterns also included the average number of records in each of those hours for every instance of its child. During the process of data generation, as the first step, all the time intervals between the given start date and the end date were generated. Then for each time interval, we randomly generated the values of its child node (for example hosts), who were possibly active during that period, using the extracted probabilities of that interval.

For example, let's consider the requirement to generate Web proxy I data for two days. The first step is to generate 48 unique timestamp values, where each value corresponds to a distinct hour between the two days. Next, for each of these timestamps, we generate all the hosts who are active during that period. For instance, if we want to generate the hosts who are active on a Monday at 7 AM and assume that among the five different hosts, according to the extracted probabilities, three hosts have 90% chance of working on a Monday at 7 AM and rest two hosts have only 20% chance. Subsequently, each of the three hosts are selected

randomly with a probability of 0.9 and rest of the two hosts are chosen with a probability of 0.2. Once we have a set of hosts for a given timestamp, each host is combined with the timestamp value to form separate records. Finally, multiple copies of each record are generated, according to the distribution of the number of events for the hosts of the record, in that hour.

Web proxy I synthetic dataset consisted of proxy log data for 5 hosts, over a period of four month. This was generated using the extracted patterns from one month of real Web proxy I data. Similarly, the patterns from one week of original Web proxy II dataset were used to produce four weeks of synthetic data.

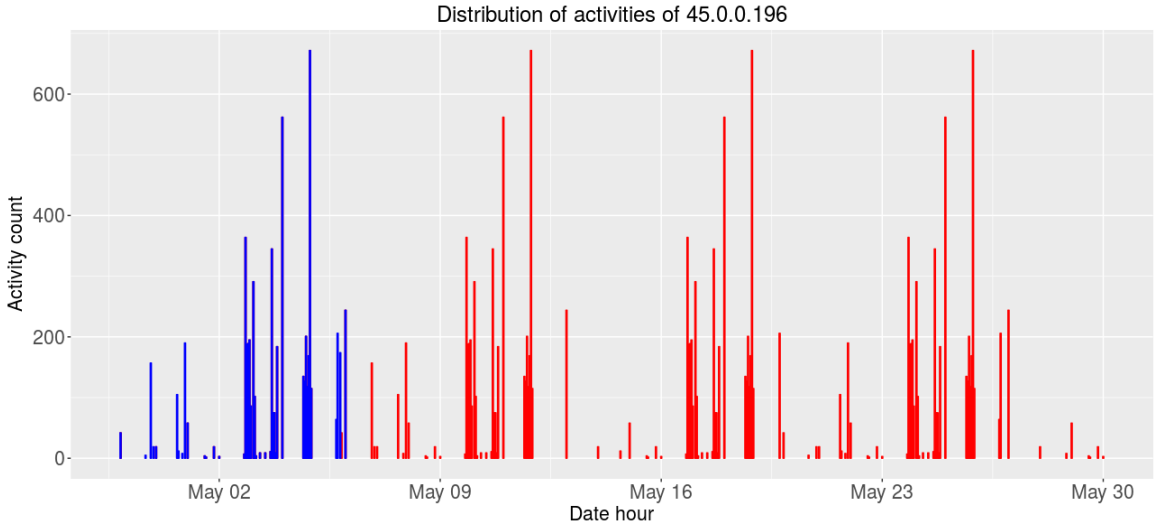


Figure 10: Pattern of activity counts in real and synthetic data for source IP 45.0.0.196 of Webproxy II dataset(*Red: Synthetic, Blue: Real*)

Figure 10 compares the activity count of a *clientIP*, in each date hour of real and synthetic Web proxy II datasets. The date hour in this figure represents the timestamp values rounded to the nearest hour, and activity count is the number of records for a *clientIP* in each of those date hours. It is visible from this figure that the synthetic data (red) is maintaining the patterns extracted from one week of real data (blue) in rest of its three weeks.

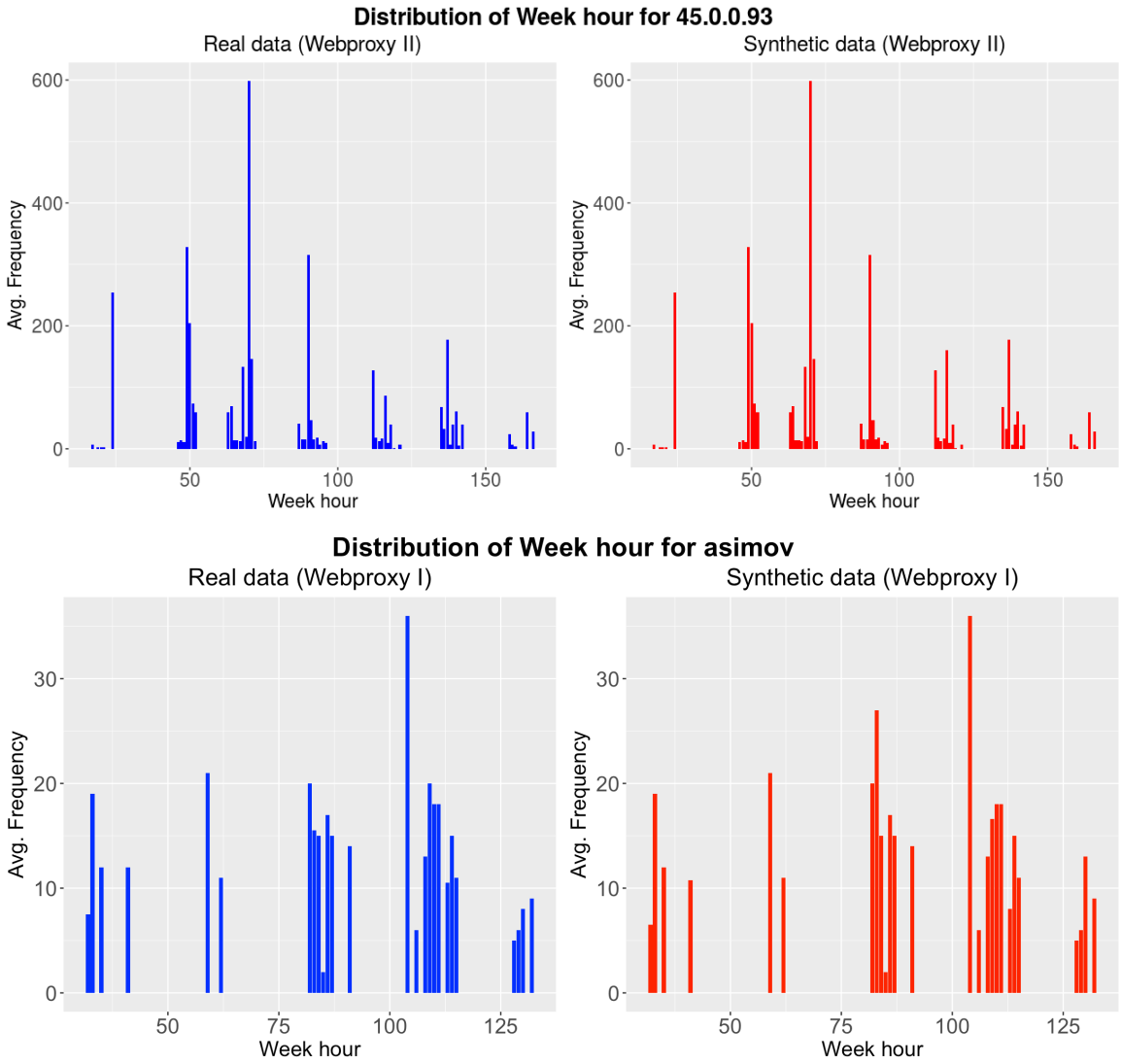


Figure 11: Week hour *vs.* Avg. Frequency (average no. of record) of Web proxy I and Web proxy II datasets (*Red: Synthetic, Blue: Real*)

When we consider the timestamp attribute, the synthetic data generated should be able to preserve the average number of records in each time interval when compared with the real data. Figure 11, compares the average number of records in each hour of the week, of real and synthetic data for both Web proxy I and Web proxy II datasets.

Week Hour	Observed count	Expected Count
104	324	324
109	166	200
41	140	156
82	140	140
114	135	135
33	133	133
108	117	117
113	106	126
110	90	90
111	90	90

Week Hour	Observed count	Expected Count
0	3127	3130
50	2966	2960
44	2623	2618
125	2615	2610
45	2556	2556
22	1715	1715
40	1660	1664
24	1492	1485
26	1282	1292
163	1264	1264

Table 15: Observed and expected record counts for sample week hours of a host from Webproxy I (left) and a client IP from Webproxy II (right)

To summarise the similarity between the timestamp attributes, we calculated the normalised RMSE (NRMSE) for each host of Web proxy I and each client IP of Web proxy II. To calculate NRMSE for a host or client IP, we took the number of records in each hour of the week from synthetic data as the observed values. The record counts in corresponding week hours of the real data were scaled to the size of synthetic data to obtain the expected values. NRMSE for each individual was calculated by applying these observed (X_o) and expected counts (X_e), along with the total number of hours in a week (n) to Equation 4. Table 16 gives the NRMSE values of sample hosts of Web proxy I, and client IPs of Web proxy II.

host	Normalized RMSE
ottgate2.bnr.ca	5.81%
bettong.client.uq.oz.au	4.39%
romulus.ultranet.com	2.03%
asimov	4.19%
cliffy.lfwc.lockheed.com	8.70%

Client IP	Normalized RMSE
45.0.0.178	0.963%
45.0.0.191	0.613%
45.0.0.196	0.001%
45.0.0.93	1.271%

Table 16: Normalized RMSE for sample hosts of Webproxy I dataset (Left) and client IPs of Webproxy II dataset (Right)

We can see from the table that the maximum error for Web proxy I dataset is 8.7% and that for Web proxy II is 1.27%. The reason for the variation in results between these two datasets is because, there is more variation in the number of events per week hour for the hosts of Web proxy I dataset than that of Web proxy II dataset. These results clearly indicate that our approach for synthetic data generation is capable of preserving the patterns followed by timestamp attributes in the real data.

4.3 Parallel data generation - Performance analysis

This section discusses the experiment conducted to test the scalability of synthetic data generator, by following the parallel data generation approach discussed in Section 3.3. We begin by explaining the datasets we used for our experiments and then describe the environment we used for conducting the experiments. Finally, we analyze and discuss the experimental results.

4.3.1 Experiment data

The two datasets we used for our experiments were Web proxy I and Web proxy II. Instead of repeating the process of pattern extraction once again, we reused the pattern files of both the datasets, obtained from the previous experiments. Pattern file of Web proxy I data consisted of the distribution information of five hosts, extracted from one month of data. This pattern data was extended by adding patterns for 100 synthetic *host*s, using the approach discussed in Section 3.4. We used this modified pattern file to generate 12 years synthetic data for all experiments with Web proxy I dataset.

The Web proxy II pattern file contained the distribution information of 4 different *sourceIP*s, extracted from one week's data. Similar to Web proxy I dataset, this pattern data is extended by adding patterns for 100 synthetic *sourceIP*s, using the approach discussed in Section 3.4. This modified pattern file was used to generate one years' synthetic data for all experiments with Web proxy II dataset.

4.3.2 Environment

Aws instance name	No. of processors	Memory (GB)
c3.large	2	3.75
c3.xlarge	4	7.5
c3.2xlarge	8	15
c3.4xlarge	16	30
c3.8xlarge	32	60

Table 17: Specifications of machines used for experiments

The experiments were performed on five machines from the Amazon Elastic Compute Cloud (EC2), with different CPU counts (refer to Table 17). In each experiment, we generated synthetic data independently using each of these five machine configurations. The data was generated using Algorithm 9 of Section 3.3, implemented on Apache Spark framework of Python.

4.3.3 Performance comparison

During each experiment, we generated synthetic Web proxy I and Web proxy II datasets using one of the five machines. We began our experiments using the machine with two processors, and each experiment doubled the number of processors from the previous one. Experiments using Web proxy I data generated two years of data with ~15 million records occupying 1.07 G of memory. Similarly, every experiment using Web proxy II generated one-year data for hundred users with ~41 million records, which occupied 2.5 G of memory (Table 18).

	Webproxy I	Webproxy II
Number of hosts in original data	4	4
Number of hosts in generated data	100	100
Date range of original data	01-07-1995 to 28-05-1995	28-04-2005 to 06-05-2005
Date range of generated data	01-07-1995 to 01-07-2007	28-04-2005 to 06-05-2006
Number of records in generated data	$\sim$ 15 million	$\sim$ 41 million
Size of generated data	1.07 G	2.5 G

Table 18: Properties of generated synthetic data

Figure 12: No. of Processors *vs.* Records per second: Linear increase in the rate of data generation with increase in number of processors for both Web proxy I and Web proxy II datasets.

When the performance of data generation was analyzed for each experiment, it was observed that the number of processors has a direct influence on the speed of data generation. As can be seen from Figure 12, the rate at which records were generated increased approximately linearly with increase in the number of processors.

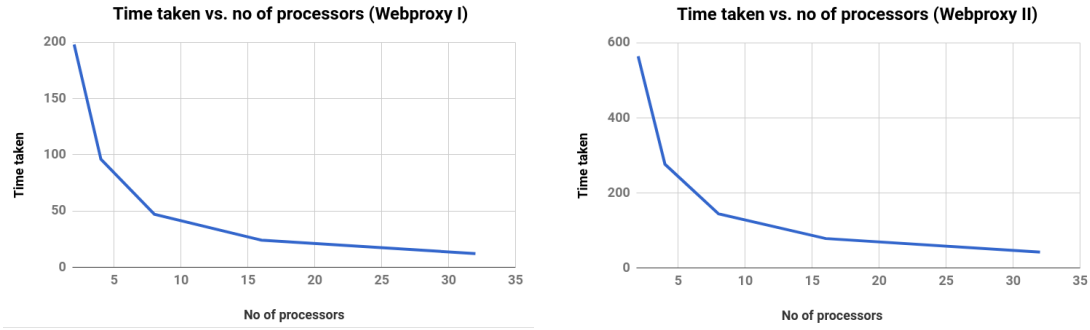


Figure 13: No. of Processors *vs.* time taken for data generation: Decrease in time taken for data generation with increase in number of processors

Similarly, we can see from Figure 13, that the time taken for generating the same amount of data was halved when the number of processors used for the data generation was doubled. With two processors, it took 564 minutes to generate 41 million records of Web proxy II data. With 32 processors, the same amount of data took only 42 minutes to generate (Figure 13). The same trends followed in the case of Web proxy I data (Table 19), but it took comparatively less time for generating the same number of records. This difference in time between two datasets were due to the differences in complexity between both the datasets such as the number of attributes, number of distinct values for each attribute, type of each attribute and number of dependency levels.

Aws instance name	No. of processors	Web proxy I		Web proxy II	
		Time taken (Mins)	Records per second	Time taken (Mins)	Records per second
c3.large	2	190	3,945	564	1,216
c3.xlarge	4	96	2,603	276	2,485
c3.2xlarge	8	47	5,314	144	4,763
c3.4xlarge	16	24	10,437	78	8,792
c3.8xlarge	32	12	20,830	42	16,329

Table 19: Time taken and record per second for data generation on machines with different number of CPUs

Hence it can be concluded that the approach we followed for data generation can be easily scaled up to the available number of processors. With more optimization and approximation, the total time required for execution can be further reduced significantly.

Chapter 5

Conclusion and Future Work

Through this work we formulated a system that can be used to generate synthetic data for User Behavioral Analytics applications. This system learns the patterns for specified attributes from a sample real data, potentially for a limited time period. Using these patterns, the system can generate synthetic data for a larger time period, for more number of users, that closely follows the behavioral pattern of the real data. This will help the development and testing of UBA applications, especially during the situations that demand large amount of data, such as scale and performance testing.

The system takes a sample real data, and a set of dependencies in the real data, as input. The dependency list consists of relationships between the attributes that the user care about, in the data. This system is capable of extracting patterns from four different data types: categorical, timestamp, integer, and double. These extracted patterns are stored in a JSON file, and used for data generation. The process of data generation can be linearly scaled to the available number of processors. This system can also generate additional synthetic users using the patterns of users that are extracted from the sample real data. Thus the system can be used to generate data for several new users, using patterns extracted from a real data of limited users.

To evaluate the system, we conducted experiments on three different datasets: Iris flower dataset, and two web proxy log datasets. These experiments and analysis showed that the generated synthetic data followed the distribution patterns present in the real data. The numerical attributes of the generated data were able to maintain distributions that were conditionally dependent on values of the parent attribute. Experiments conducted also showed that the timestamp attribute of the generated data followed the same density of events on different hours of the week as that of the real data. The peak hours and inactive hours of the users in real data were correctly captured and reproduced in the synthetic data. Results obtained showed a low normalized RMSE between the observed and expected counts of timestamp and categorical attributes, indicating close similarity between real and synthetic data.

Experiments were also performed to evaluate the scalability of the developed system. Through the experiments conducted on five different machines with different number of processors, we saw a linear increase in the rate of data generation with the increase in the number of processors.

Future Work

Next let us discuss some of the possible ways to extend and enhance the developed synthetic data generation system, in future.

- This work focused only on parallelizing the data generation part since pattern extraction is a one-time process. In future we would like to parallelize the pattern extraction process. This will allow the end user to extract patterns from large and complex datasets and use these patterns, thereby being able to generate more realistic data.

- The dependencies supported by our current system are limited and is subject to constraints. For instance, our system does not support dependencies between numerical attributes. It does not allow numerical attributes to be the parent of a categorical attribute. Extending the dependency support could generalize the system and make it suitable to address synthetic data generation for more complex real datasets.

- The main focus of this work is to generate synthetic data for the development, testing, and demonstration of UBA systems. We would like to study the inferences that can be gathered from the generated synthetic data.

- In this work, we have followed a Maximum Likelihood Estimation approach to extract the patterns from the given real data. In future we would like to try the Bayesian Estimation approach for pattern extraction. This may allow us to obtain a posterior distribution of attributes instead of a point value obtained from the currently used Maximum Likelihood Estimation approach.

- Our current approach depends on the attribute dependencies given by the end user for pattern extraction. In future we are planning to infer these dependencies from the data using methods such as tree dependence, thereby

eliminating the need for collecting attribute dependencies in the real data from end user.

- Instead of keeping the distributions extracted as static, we would like to explore the approaches using time variant distributions and drift analysis that allows these distributions to change over time.

- In some applications, maintaining the anonymity of the data without disturbing the underlying distributions and patterns is important. To protect privacy of individuals in synthetic data generated using our approach, we would like to explore efficient ways by which we can add noise into the generated data, thereby removing the possibility of tracing the data back to individuals.

List of References

[1] K. Murphy. "A brief introduction to graphical models and Bayesian networks." http://www.cs.ubc.ca/~murphyk/Bayes/bnintro.html (1998).

[2] M. Kanellos. "152,000 smart devices every minute in 2025: IDC outlines the future of smart things." https://www.forbes.com/sites/michaelkanellos/2016/03/03/152000-smart-devices-every-minute-in-2025-idc-outlines-the-future-of-smart-things/ (2016).

[3] A. Alexandrov, K. Tzoumas, and V. Markl. "Myriad: Scalable and expressive data generation." *Proc. VLDB Endow.* **5**(12), 1890–1893. ISSN 2150-8097 (2012).

[4] M. Harvey, N. McCarroll, and Y. Shuf. "Generating synthetic data." US Patent App. 14/056,179, https://www.google.com/patents/US20140115007 (2014).

[5] L. Corporation. "Lockheed Martin commercial cyber announces insider threat detection solution with interset." http://www.prnewswire.com/news-releases/lockheed-martin-commercial-cyber-announces-insider-threat-detection-solution-with-interset-300283340.html (2017).

[6] M. Rouse. "What is user behavior analytics (UBA)? - definition from whatis.com." http://searchsecurity.techtarget.com/definition/user-behavior-analytics-UBA.

[7] M. Tierney. "Veriato blog." https://www.veriato.com/company/blog/veriato-blog/2015/12/15/the-rise-of-user-behavior-analytics (2015).

[8] J. T. Johnson. "User behavioral analytics tools can thwart security attacks." http://searchsecurity.techtarget.com/feature/User-behavioral-analytics-tools-can-thwart-security-attacks.

[9] A. Chuvakin. "Public security log sharing." http://log-sharing.dreamhosters.com/.

[10] "Active directory architecture." `https://technet.microsoft.com/en-us/library/bb727030.aspx`.

[11] J. Eno and C. W. Thompson. "Generating synthetic data to match data mining patterns." *IEEE Internet Computing* **12**(3), 78–82. ISSN 1089-7801 (2008).

[12] J. E. Hoag and C. W. Thompson. "A parallel general-purpose synthetic data generator." *SIGMOD Rec.* **36**(1), 19–24. ISSN 0163-5808 (2007).

[13] I. Adä and M. R. Berthold. "The new Iris data: Modular data generators." In "Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining," KDD '10, pages 413–422. ACM, New York, NY, USA. ISBN 978-1-4503-0055-1 (2010).

[14] J. Young, P. Graham, and R. Penny. "Using Bayesian networks to create synthetic data." *Journal of Official Statistics* **25**(4), 549 (2009).

[15] A. Oniśko, M. J. Druzdzel, and H. Wasyluk. "Learning Bayesian network parameters from small data sets: Application of noisy-or gates." *International Journal of Approximate Reasoning* **27**(2), 165–182 (2001).

[16] K. Houkjær, K. Torp, and R. Wind. "Simple and realistic data generation." In "Proceedings of the 32nd international conference on Very large data bases," pages 1243–1246. VLDB Endowment (2006).

[17] A. Omari, R. Langer, and S. Conrad. "Tartool: A temporal dataset generator for market basket analysis." In "International Conference on Advanced Data Mining and Applications," pages 400–410. Springer (2008).

[18] A. Elgammal, R. Duraiswami, D. Harwood, and L. S. Davis. "Background and foreground modeling using nonparametric kernel density estimation for visual surveillance." *Proceedings of the IEEE* **90**(7), 1151–1163 (2002).

[19] "How kernel density works." `http://pro.arcgis.com/en/pro-app/tool-reference/spatial-analyst/how-kernel-density-works.htm` (2017).

[20] S. J. Sheather *et al.* "Density estimation." *Statistical Science* **19**(4), 588–597 (2004).

[21] J. JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA. ISBN 0-201-54856-9 (1992).

[22] "Imhotep - data analytics platform by indeed." http://opensource.indeedeng.io/imhotep/docs/sample-data/.

[23] "Statistics cirp wiki." http://cirpwiki.info/wiki/Statistics#Normalization.

Appendix A

Iris distributions

A.1 Distributions of Iris-setose

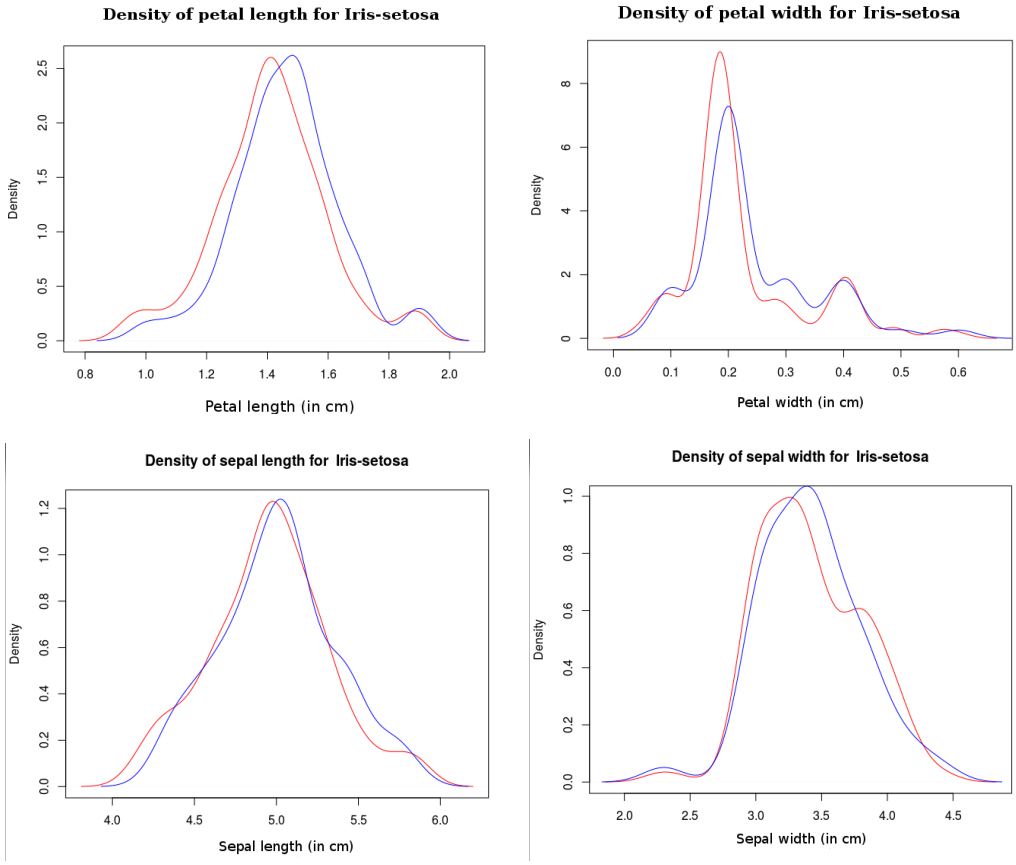


Figure 14: Distribution of the attributes of Iris-setose

A.2 Distributions of Iris-virginica

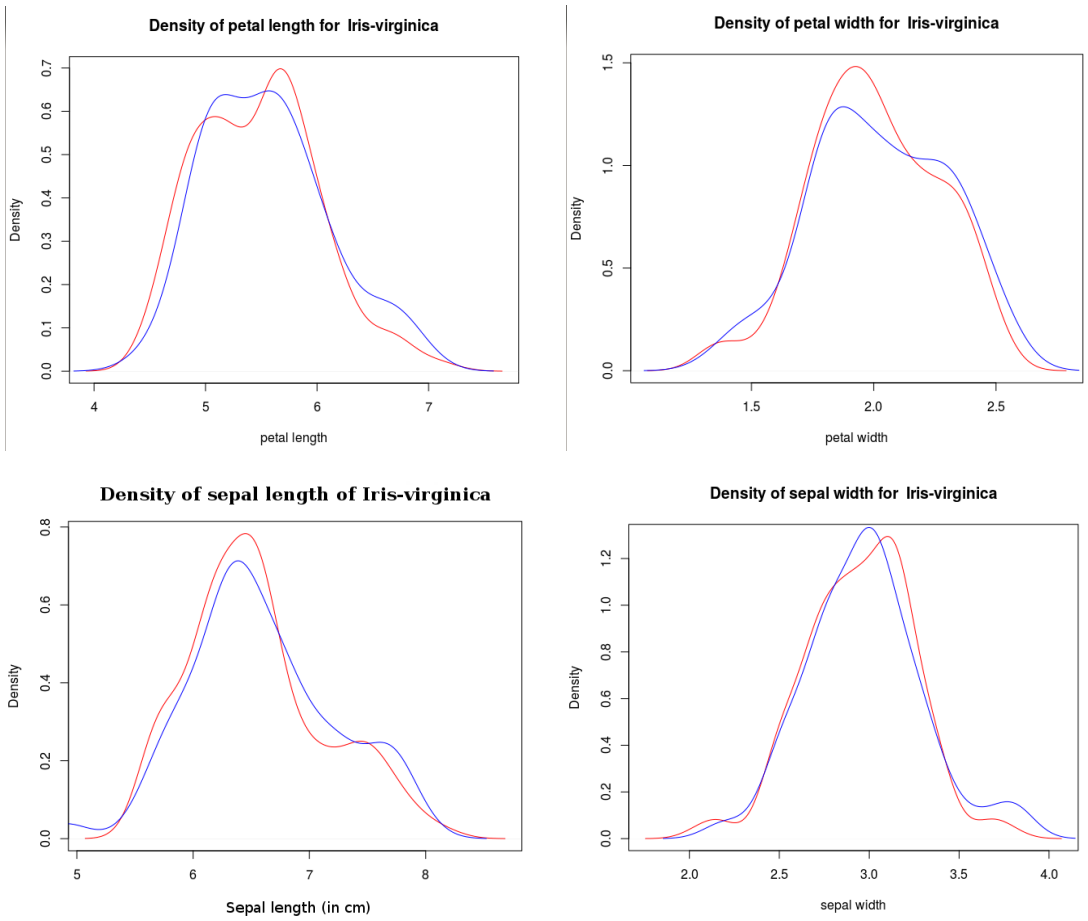


Figure 15: Distribution of the attributes of Iris-virginica

A.3 Distributions of Iris-versicolor

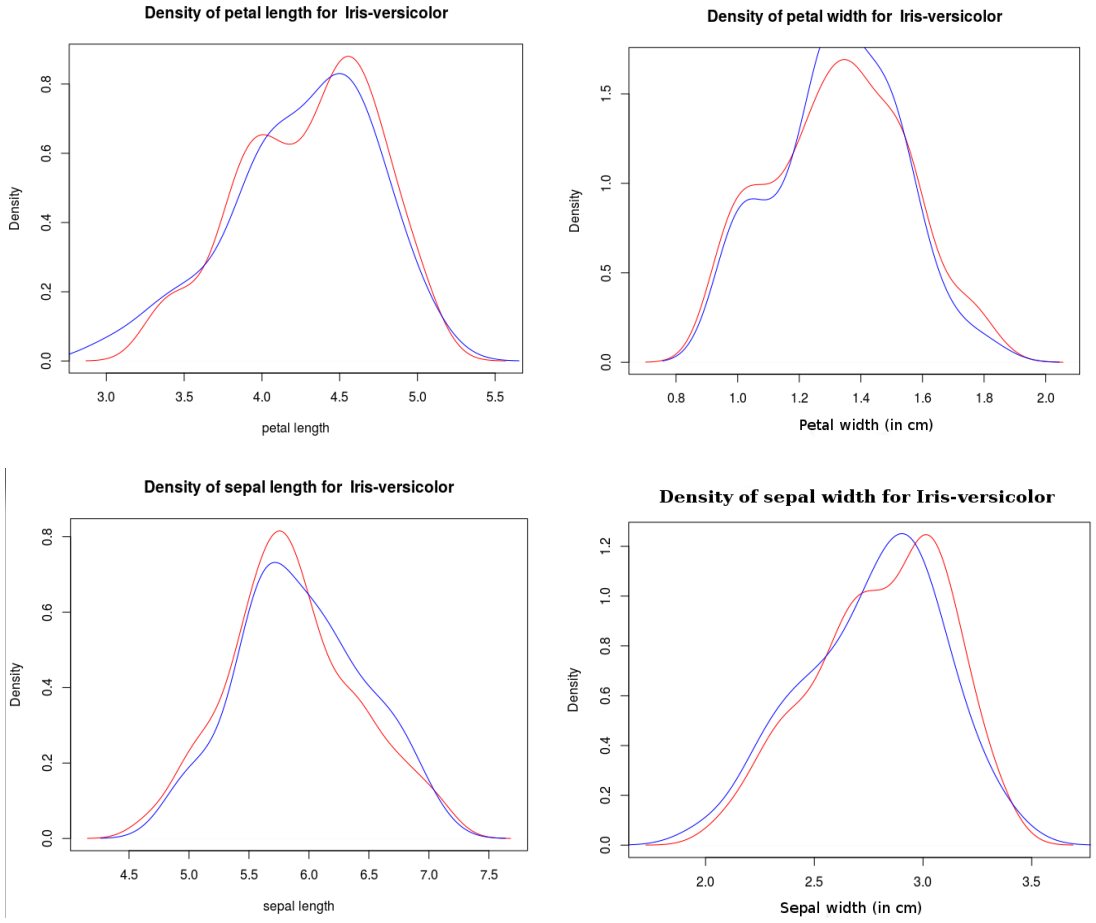


Figure 16: Distribution of the attributes of Iris-versicolor

Appendix B

Sample JSON file

B.1 JSON file


```
{
   "name": "httpResponseStatus",
   "level": 3,
   "c_p_t": {
      "fpdownload.macromedia.com": {
         "304": 1
      },
      "images.ea.com": {
         "200": 1
      },
      "uf2.svrni.ca": {
         "200": 1
      },
      "news.bbc.co.uk": {
         "200": 0.5,
         "301": 0.5
      },
      "pgq.yahoo.com": {
         "200": 1
      },...
   },
   "is_root": "No",
   "parents": ["destHostName"],
   "col_type": "varchar"
}
```

```
{
   "name": "payloadSizeResponse",
   "level": 3,
   "c_p_t": {
       "members.lycos.co.uk": {
              "5907": 1
       },
       "images.ea.com": {
              "0.0": 2.8248167E-5,
              "69486.2": 1.082612E-5,...
       },
       "uf2.svrni.ca": {
              "28052.541": 8.7698376E-5,
              "17263.102": 8.047894E-5,...
       },
       "news.bbc.co.uk": {
              "0.0": 1.10893925E-4,
              "13647.093": 1.1127061E-4,
              "11941.206": 5.226901E-5,...
       },
      "pgq.yahoo.com": {
         "342.83563": 0.51028556,
         "340.82907": 0.41615093,
         "341.97568": 0.29967806,...
      }
   }
    "bandwidth": {
       "members.lycos.co.uk": 0,
       "images.ea.com": 9926.601,
       "uf2.svrni.ca": 2157.8877,
       "news.bbc.co.uk": 1705.8866,
       "pgq.yahoo.com": 0.28665185,...
    },
    "is_root": "No",
    "parents": ["destHostName"],
    "col_type": "int"
}


{
       "name": "timestamp",
       "level": 0,
       "time_probs":
       {"clientIp":
              {"45.0.0.93": {3840: 1.0, 0: 1.0, 6660: 0.5, 3900: 1.0, ...},
              "45.0.0.191": {0: 1.0, 9600: 1.0, 6660: 0.5, ...},
              "45.0.0.178": {5760: 1.0, 0: 1.0, 6660: 0.5, ...},}}
       "time_bucket": "weekhour",
       "is_root": "Yes",
       "parents": [],
       "col_type": "timestamp",
       "number_eventsPH":
       {"clientIp": {
              "45.0.0.93": {0: 254, , 3840: 16,  2820: 146: 1, 5640: 39: 1, ...},
              "45.0.0.191": {0: 1, 9600: 126, 900: 643, 4320: 2, 3720: 182, ...},
              "45.0.0.178": {0: 297,  5760: 91, 900: 193, 5640: 17, ... },...
       "parentscount": None,
       "children": ["clientIp"],
       "ts_format": "%Y-%m-%dT%H:%M:%S"}}
}
```