

OUTLIER SEQUENCE DETECTION USING LZW

by

Miki J. Ushijima

ProQuest Number: 10794995

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10794995

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code  
Microform Edition © ProQuest LLC.

ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 – 1346

T 6230

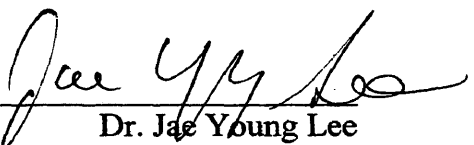
C.2

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematical and Computer Sciences).

Golden, Colorado

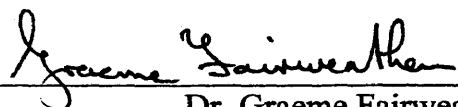
Date 1/17/2007

Signed:   
Miki J. Ushijima

Approved:   
Dr. Jae Young Lee  
Thesis Advisor

Golden, Colorado

Date 1/22/2007

  
Dr. Graeme Fairweather  
Professor and Head  
Department of Mathematical and  
Computer Sciences

## ABSTRACT

Outlier detection of sequence data consists of finding sequences that have abnormalities or uncommon characteristics from the rest of the population. These sequences can represent data ranging from strings of DNA to written text. Being able to find outlier sequences is beneficial towards applications such as noise removal to fraud detection. Unfortunately, it is difficult to find a universal outlier detection method that is efficient yet accurate. Accuracy and precision is often times gained through complex and exhaustive methods, but that becomes a problem with the continuing growth of the amount of data stored in databases. It has become imperative to explore methods that can detect outliers both accurately and efficiently.

To achieve such a goal, we propose a new technique that incorporates a data compression algorithm called LZW. A series of modifications and adaptations are made to the simple yet effective LZW algorithm to transform it from a data compression algorithm to a vital process in outlier detection. We show the method's ability to accurately detect outliers through a series of experiments. One of the experiments is designed so that our method can be fairly compared to another method recently published. The final result is an effective outlier sequence detection method that is computationally inexpensive relative to exhaustive outlier detection methods while maintaining a comparable accuracy.

## TABLE OF CONTENTS

ABSTRACT.....	iii
LIST OF FIGURES .....	vi
LIST OF TABLES.....	vii
ACKNOWLEDGEMENTS.....	ix
CHAPTER 1 INTRODUCTION.....	1
1.1 Overview of Outlier Detection.....	1
1.1.1 Challenges of Outlier Detection.....	1
1.1.2 General Approaches to Outlier Detection.....	2
1.2 Research Overview .....	4
CHAPTER 2 PREVIOUS AND RELATED WORK.....	7
2.1 Simple Statistical Approaches .....	7
2.2 Distance-Based Approaches .....	8
2.3 Deviation-Based Approaches.....	9
2.4 Other Techniques and Areas of Research.....	11
2.5 Application towards Sequence Data .....	13
CHAPTER 3 OVERVIEW OF DATA COMPRESSION AND LZW .....	15
3.1 Outlier Detection and Data Compression .....	15
3.2 Brief Discussion of Data Compression.....	16
3.2.1 Lossless versus Lossy Compression .....	17
3.2.2 Common Lossless Compression Techniques.....	17
3.3 LZW Algorithm .....	23
3.4 Characteristics of the Patterns Found by LZW .....	27

CHAPTER 4	THE OUTLIER SEQUENCE DETECTION USING LZW(OSDUL) PROCESS .....	31
4.1	Applying the LZW Algorithm to Sequence Data .....	31
4.1.1	Modifications to the LZW Algorithm.....	31
4.1.2	The Source of the Pattern.....	32
4.1.3	The Trimming Process.....	34
4.2	Consideration of Memory .....	41
4.3	Score Assignment to Patterns .....	43
4.3.1	Method 1: Assigning Bit Length .....	45
4.3.2	Method 2: Scaled Scores.....	47
4.4	Calculating a Compression Ratio of a Sequence .....	49
4.5	Outlier Detection Using the Compression Ratio .....	52
4.5.1	Method 1: K-Means Clustering .....	52
4.5.2	Method 2: Deviation-Based Method.....	54
4.6	The OSDUL Process.....	57
4.7	Computational Complexity.....	57
CHAPTER 5	EXPERIMENTS AND RESULTS.....	63
5.1	Single Family Outlier Detection .....	64
5.2	Varying the Outlier Percentage in the Dataset.....	75
5.3	Outliers from Various Families .....	76
5.4	Supervised Outlier Detection.....	76
CHAPTER 6	FUTURE WORK .....	83
REFERENCES	.....	85

## LIST OF FIGURES

3.1 The LZW algorithm .....	25
4.1 The LZW algorithm modified for OSDUL.....	33
4.2 The OSDUL-LZW algorithm with source consideration .....	35
4.3 Algorithm of the trimming process.....	39
4.4 OSDUL-LZW algorithm with TRIM.....	42
4.5 Algorithm to find matching patterns.....	53
4.6 K-Means clustering algorithm .....	55
4.7 Deviation-based algorithm.....	56
4.8 The OSDUL Process.....	58
4.9 Example pattern tree created by OSDUL-LZW algorithm .....	61
5.1 Ratio vs Frequency of the Dataset HCV(380) and RubellaE1(20).....	67
5.2 Ratio vs Frequency of the Dataset RubellaE1(380) and HCV(20).....	68
5.3 Ratio vs Frequency of the Dataset HCV(380) and TetRN(20).....	69
5.4 Ratio vs Frequency of the Dataset RubellaE1(380) and TetRN(20) .....	70
5.5 Ratio vs Frequency of the Dataset TetRN(380) and HCV(20).....	71
5.6 Ratio vs Frequency of the Dataset TetRN(380) and RubellaE1(20) .....	72
5.7 Ratio vs Frequency of the Dataset TetRN(380) and HCV(20) No Trimming .....	73
5.8 Ratio vs Frequency of the Dataset TetRN(380) and RubellaE1(20) No Trimming .....	74
5.9 Distribution of the HCV Family .....	78

## LIST OF TABLES

3.1	An example Huffman encoding table .....	19
3.2	Adjusted Huffman encoding table .....	20
3.3	Initial translation table .....	24
3.4	The resulting translation table.....	26
3.5a	Resulting table of Data 1 .....	29
3.5b	Resulting table of Data 2.....	29
4.1a	Initial translation table.....	36
4.1b	Translation table after processing the first sequence data: abcabcabc.....	36
4.1c	Translation table after processing the second sequence data: abcabcabc .....	37
4.1d	Translation table after processing the third sequence: bbcabb .....	38
4.2	Modified Table 4.1d due to trimming.....	40
4.3	An example of a sorted translation table.....	44
4.4	An example of a sorted translation table with scores based on Method 1 .....	47
4.5	An example of a sorted translation table with scores based on Method 2.....	49
5.1	HCV(380) and RubellaE1(20) average accuracy measurements.....	67
5.2	RubellaE1(380) and HCV(20) average accuracy measurements.....	68
5.3	HCV(380) and TetRN(20) average accuracy measurements.....	69
5.4	RubellaE1(380) and TetRN(20) average accuracy measurements .....	70
5.5	TetRN(380) and HCV(20) average accuracy measurements.....	71
5.6	TetRN(380) and RubellaE1(20) average accuracy measurements .....	72
5.7	Varying the composition of RubellaE1 sequences against HCV.....	75
5.8	Varying the composition of TetRN sequences against HCV.....	75
5.9	Results for having outlier sequences from two families.....	76

5.10 The percentage of the sequence from each family that were 3 standard deviations from the mean, results from [35] ..... 79

5.11 The percentage of the sequence from each family that were 1.5\*IQR away from the third quartile..... 80

## ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude and thanks to Dr. Jae Young Lee for giving me the opportunity to research this topic, and for his patience and guidance throughout the process. I would also like to extend my thanks to the committee members Dr. William Hoff and Dr. Cyndi Rader for their time and input in helping to develop this thesis to its fullest potential. Finally, I would like to thank my manager Jim Crowell for allowing me to take some time to work on this thesis, and my family and friends for all of their support.

## CHAPTER 1

### INTRODUCTION

#### 1.1 Overview of Outlier Detection

Outlier detection is a data mining task that attempts to identify data within a dataset that does not conform to common behaviors of the entire dataset. In many cases, these outliers are considered undesirable noise, and outlier detection methods can be used to remove these noises to purify the dataset. Other times, the outlier data can be very informative. If applied correctly, outlier detection can effectively detect uncommon and suspicious patterns that lead to network intrusion detection, credit card usage fraud detection, DNA mutation detection, and much more [1].

##### 1.1.1 Challenges of Outlier Detection

Significant challenges must be addressed in order for outlier detection to be effective. Any outlier detection method can be defined to have two simply stated but surprisingly difficult subtasks: define what makes a data point in the dataset an outlier and to efficiently find those outliers from the dataset [2].

The first subtask of outlier detection is to define attributes that make an item in the data notably different from the rest of the dataset. The user of the outlier detection method must carefully assess the dataset, and have significant knowledge of the content of the data. What attributes of the data can we quantify and how many of those attributes

do we need to consider when we are trying to detect outliers? Selecting too many attributes often leads to significant increase in computational complexity [1], but selecting too few attributes runs the risk of decreasing accuracy of the outlier detection.

The second subtask is as equally challenging as the first. With a list of attributes at hand, the method must effectively calculate those attributes and assign a numerical value which the computer will be able to compare. This becomes a daunting task when the attribute is not naturally numerical. For example, if a path of an object in motion is determined to be one of the attributes to compare, then one must find a way to effectively quantify the path. The same is true for attributes such as shape, text, and credit card use behavior. This becomes very difficult and computationally expensive when there are multiple attributes to compare. Once the attributes are quantified and compared, the final part is to determine the outliers. The outlier detection must differentiate the outliers from the common data efficiently and accurately.

These outlier detection challenges are no different for sequences. A sequence is data that is encoded in a string of characters or symbols. English text and DNA are examples of sequences. No matter the original content of the data, comparing a string of characters and trying to detect outliers is no easy task.

### **1.1.2 General Approaches to Outlier Detection**

Most outlier detection methods can be categorized into three approaches: statistical-based, distance-based and deviation-based outlier detection. As the name suggests, statistical-based outlier detection uses statistical means to detect outliers. It assumes that the population follows a certain known distribution or a probability model such as the normal distribution [1]. This requires significant knowledge of the dataset prior to outlier detection. From this knowledge, statistical parameters such as mean and variance are calculated and used for hypothesis testing [3]. A hypothesis test considers

two hypotheses where a hypothesis is a “statement about a population parameter.” The two hypotheses are called the null hypothesis and the alternative hypothesis, and generally, these two hypotheses complement each other. If the null hypothesis states that the mean,  $\mu$ , is equal to 0, then the alternative hypothesis will often state that  $\mu$  is not equal to 0 [3]. In the context of outlier detection, the null hypothesis states that data come from the assumed distribution, and the alternative hypothesis states that the data comes from a different distribution. If the null hypothesis is true for a set of data from the dataset, then that set of data is considered non-outliers of the dataset. On the other hand, if a set of data from the dataset was rejected by the null hypothesis, then the alternative hypothesis becomes true. This indicates that the rejected data do not belong in the initial distribution and are considered outliers of the dataset.

The significant limitation of statistical-based approaches to outlier detection is the requirement of knowing, with confidence, that the population follows a certain known distribution. If that assumption cannot be made with confidence, then statistical-methods cannot guarantee accurate outlier detection. To avoid such a limitation, other outlier detection methods are introduced such as the distance-based outlier detection [4, 5]. Distance-based outlier detection relies on methods to calculate a distance between data using the data’s parameters or attribute values. Once we know the distances between all points in the dataset, we can calculate how “close” the data points are to each other. The data points can congregate to one cluster, or they can congregate to several different clusters. If there is a data point that is significantly far from the clusters populated by the majority of the dataset, then that data point is considered an outlier. Though this does not require as much knowledge of the population distribution as the statistical approaches require, it does require intelligent decisions in deciding how to calculate the distance and how far is far enough for a data to become an outlier. These decisions are often made through experimentation.

The third approach is the deviation-based outlier detection first introduced in [6]. This method attempts to detect data that deviates from the general population. It often

requires a dissimilarity function which is a function that, given a set of data, returns a value indicating how “dissimilar” the data is within this set. A low dissimilarity value means that the items in the data are similar to each other. A subset that, when removed from the original set of data, reduces the overall dissimilarity value is called the exception set. The items in this exception set can then be considered outliers of the original dataset, because by removing them, the dataset becomes more consistent. Much like the distance-based approach, this method does not require the knowledge of the population distribution, but an effective dissimilarity function must be chosen.

In addition to these methods, there are simpler approaches if the data is univariate, meaning that there is only one dimension for the data. These simple approaches use basic statistical measures such as mean and standard deviation. For data with a normal distribution, a data point that is three standard deviations away from the mean can be considered an outlier of that distribution [7]. Calculating the interquartile range, IQR, as a measure for outliers is also a simple and commonly used technique. The IQR is the difference between the first quartile and the third quartile of the data. Any data point that is beyond  $1.5 \cdot \text{IQR}$  away from the first or the third quartile can be considered outliers [8, 9].

There is no one method that claims to be better than the others. The methods have their strengths and weaknesses, and the appropriate approach must be chosen based on the information available about the context of the data.

## **1.2 Research Overview**

As discussed in the previous section, many outlier detection techniques rely on or require prior knowledge of the context of the data. This requirement limits the ability of one outlier detection method to be used towards other applications in other contexts. An outlier detection method used to detect DNA mutation may not be suitable for detecting

unusual data packets sent through the internet. The goal of this research concentrates on finding an outlier detection method specific to sequence data that does not require prior knowledge. Any data that can be represented in a sequence of characters or symbols should be able to use this method to detect outliers.

In specific terms, this research introduces a context-free outlier detection method for sequence data by integrating a data compression technique called LZW. A brief discussion about data compression and how data compression, specifically the LZW algorithm, fits outlier sequence data detection is in Chapter 3 of this document. The rest of the document will discuss previous and related work in Chapter 2, our proposed method in Chapter 4, its experimental results in Chapter 5 and future work in Chapter 6.



## CHAPTER 2

### PREVIOUS AND RELATED WORK

#### 2.1 Simple Statistical Approaches

Many approaches have been developed over time for outlier detection. Some of the earliest developments were simple statistical-based outlier detection for univariate or single dimension cases such as the Wright's method [7]. Wright's method labels a data point as an outlier if it is more than three standard deviations away from the mean. With a normal distribution, it is expected that 99% of the data lies within three standard deviations of the mean. Tukey introduces a different method in which he uses the box-plot as a visual aid in detecting outliers [8 , 9]. Tukey establishes "fences" that define boundaries which set the range of normally behaving data. Any data that are not within these boundaries are considered outliers. The fences are calculated by multiplying the interquartile range by a factor of 1.5 or 3 and subtracting it from the first quartile or adding it to the third quartile. Multiplying the interquartile range by 1.5 creates a fence that rejects "outside" data points while multiplying the interquartile range by 3 a fence that rejects "far out" data points. Using 1.5 as the multiplying factor is used commonly in practice [9].

## 2.2 Distance-Based Approaches

Distance-based outlier detection has been explored by many in recent years. It can be argued that Knorr and Ng were among the first to introduce an efficient method that is optimized to find outliers [4, 5]. Before their research, clustering algorithms such as the CLARANS[10], DBSCAN[11] and BIRCH [12] indirectly led to outlier detection, but that was not their main objective. Knorr and Ng use a measure called DB( $p$ ,  $D$ )-outlier. A data point  $o_1$  is an outlier if the ratio of data points that have a distance of  $D$  or more from  $o_1$  is greater than  $p$ . To detect outliers that are more than three standard deviations away from the mean of normally distributed data, the values for  $p$  and  $D$  are 0.9988 and  $0.13\sigma$  respectively, where  $\sigma$  is the standard deviation of the distribution [4]. To calculate this DB( $p$ ,  $D$ )-outlier measurement, Knorr and Ng develop algorithms such as the index-based algorithm and the nested-loop algorithm [5].

The index-based algorithm uses standard multidimensional indexing structures such as R-trees [13] and k-d trees[14, 15] to search for neighbors that are within a radius  $D$  of the data point. If we set  $M = N(1-p)$ , then  $M$  is the minimum number of data points required to be within a radius of  $D$  for the data point to be labeled as a non-outlier. If the algorithm cannot find at least  $M$  neighbors with distance  $D$  from a data point, then that data point is an outlier. Doing this for all data points in the dataset produces a list of outliers and non-outliers based on DB( $p$ ,  $D$ ) [5].

The nested-loop algorithm is in concept the same as the index-based algorithm. The difference is in the implementation. The nested-loop algorithm avoids potentially computationally intensive procedures by dividing the data into smaller blocks and iterating through them in a nested-loop form. During the iteration, the algorithm compares the distance of two data points directly, and keeps a count of how many neighbors it was able to find. It first compares data points within the same block. Data points are removed from the block if there are more than  $M$  neighbors within the same block calculated by the DB( $p$ ,  $D$ ) measure. Those data points that were not removed are

then compared to data points from different blocks. After processing all data blocks, those remaining in the data block are considered outliers [5].

Ramaswamy, Rastogi, and Shim point out shortcomings of the algorithms introduced by Knorr and Ng [5] and address them in their approach [16]. Their approach does not require the user to specify the distance  $D$  required by Knorr and Ng's measurement,  $DB(p, D)$ . Instead, they measure the distance from point  $m$  to the  $k$ th nearest neighbor, denoted  $D^k(m)$ , to determine a point's likelihood of being an outlier. A large  $D^k(m)$  indicates that points around  $m$  are sparse. They use the distance  $D^k(m)$  to find the top  $n$  outliers, assuming that the user is only interested in those outliers. In their perspective, a data point  $m$  is an outlier if "no more than  $n-1$  other points in the data set have a higher value for  $D^k$ " than  $m$ . This approach indirectly ranks each of the outliers; the  $n$  outliers the algorithm reports are the  $n$  highest ranking outliers, or outliers that are the furthest from the rest of the dataset [16].

In calculating the  $D^k$  value, Ramaswamy et al. also develop an index-based and a nested-loop based algorithm. Additionally, they provide a partition-based algorithm that partitions the input prior to calculating  $D^k$  values by running it through a clustering algorithm. They calculate a lower and upper value of  $D^k$  for points in each partition. By comparing the lower and upper boundaries of the partitions, they remove those partitions that are guaranteed not to have the top  $n$  outliers. The remaining partitions become candidate partitions and an index-based approach is used to detect the top  $n$  outliers.

### **2.3 Deviation-Based Approaches**

Unlike the distance-based approaches, deviation-based approaches such as the one by Arning, Agrawal, and Raghavan [6] do not require a metrical distance function to find the difference between two data points. Instead, their algorithm requires a function that calculates change in "dissimilarity" of the dataset due to changes in the composition of

the data. This dissimilarity function,  $D$ , combined with a cardinality function,  $C$ , yields a smoothing factor,  $SF$  of the set of items  $I$  and  $I_j$  where  $I_j \subseteq I$  and  $I_j = \{i_1, i_2, i_3, \dots, i_j\}$ .

$$SF(I_j) = C(I - I_j) * (D(I) - D(I - I_j)) \quad (2.1)$$

A set of similar items will have a high smoothing factor. On the other hand, if the set contained a lot of dissimilar items, that set will have a low smoothing factor. Thus, the smoothing factor of a set will increase as we remove outliers from the set and at the point when the smoothing factor reaches a maximum, the set is free of outliers. This means that the items of  $I$  that are not in  $I_j$  can be considered outliers of the dataset  $I$ . This set of items,  $I - I_j$ , is called the exception set [6].

The dissimilarity function  $D$  for the deviation-based method does not have to be metrical. Any function that returns a low value when data points in  $I$  are dissimilar and high values when data points in  $I$  are similar can be used as the dissimilarity function  $D$ . The cardinality function  $C$  must return a lower value for  $C(I_j)$  in relation to  $C(I)$ . The simplest form of the cardinality function  $C$  is the actual size of the dataset. For example, if  $I$  contains 10 data points and  $I_j$  contains 7 data points, then  $C(I) = 10$  and  $C(I_j) = 7$ . Arning et al. suggest another cardinality function  $C$ .

$$C = \frac{1}{|I_j| + 1} \quad (2.2)$$

Here,  $|I_j|$  is the size of the set  $I_j$ . This cardinality function  $C$  favors small exception size, and is suggested by the authors because the outlier set tends to be small in size compared to the entire data set.

Another approach to deviation-based outlier detection is the OLAP Data Cube Technique introduced by Sarawagi, Agrawal, and Megiddo [17] and uses multidimensional databases [18, 19]. These multidimensional databases are also called data cubes and they consist of two types of attributes called measures and dimensions [20]. Measurements are numerical attributes while dimensions are a set of non-numerical attributes that together form a key. Using this data cube, the method can take either a hypothesis-driven exploration approach or a discovery-driven exploration approach [17].

The hypothesis-driven exploration approach is a drill-down approach; the algorithm typically starts from the highest hierarchy of the data cube and works its way down to find abnormalities based on the aggregated values and visible data in the lower hierarchies. If the search down the data cube does not detect outliers, then the algorithm rolls up to a higher hierarchy and follows a different path. Unfortunately, as Colliat illustrates in [21], the search space can get very large, so this approach is not efficient.

Rather than hypothesis-driven exploration, Sarawagi et al. suggest the discovery-driven exploration where the algorithm finds outliers based on indicators at various levels of the data cube. It does so by computing an anticipated value of the data point based on its position, and combinations of trends along the different dimensions surrounding the data point. This algorithm requires similar data scan techniques as with cube aggregate computation discussed in [22].

## **2.4 Other Techniques and Areas of Research**

Statistical-based, distance-based, and deviation-based outlier detection techniques are all basic forms of outlier detection. Other techniques have been developed using different approaches for different types of data. One popular field of outlier detection research is outlier detection of visual images such as the one explained in [23]. It uses the Bayesian model to detect outlier edges that are not aligned to the dominant structures

in the image. Image-based outlier detection may not be successful when visualizing the data set is limited by screen resolution, human eye resolution or computational power. Wegman in [24] proposes strategies to overcome this problem unique to visual data mining techniques.

Outlier detection of objects in motion is also a very popular field of study, especially of data coming from surveillance cameras. Studies [25, 26, 27, 28, 29] are only a sample of the many efforts to quantify and recognize motion for outlier detection.

Neural networks have also been used for outlier detection. Hawkins, He, Williams and Baxter present a generic outlier detection of large multivariate databases using replicator neural networks [30], while Dannenberg, Thom and Watson provide a neural network implementation specific to music recognition [31].

This thesis explores yet another technique: the integration of data compression into outlier detection. As Cilibrasi and Vitanyi express in a recent paper [32], the use of data compression towards outlier detection has not been explored as much in the past. There have been a few, such as the one introduced by Benedetto, Caglioti, and Loreto[33], but they use data compression as a means to calculate entropy. The difference in entropy of a set of data to another becomes their basis of outlier detection rather than evolving the data compression technique toward outlier detection. Similar approaches towards proteins sequences are summarized in [34] illustrating the potential of data compression towards outlier detection and other data mining techniques.

There is one study documented in [35] by Sun, Chawla, and Arunasalam that does not use data compression techniques but implements an outlier detection method similar to the one presented in this thesis. Sun et al. propose an outlier detection method that uses a Probabilistic Suffix Tree, PST. A PST is “a compact representation of a variable order Markov chain, which uses a suffix tree as its index structure.” The PST is created in such a way that the outlier sequences are at the root, and each node records a probability distribution vector of the node. The probability distribution vector corresponds to the conditional probabilities of the symbol of the node immediately

following the chain of symbols leading to this node. These probability distribution vectors are used to calculate measures that compare the similarity of sequences, called similarity measures. If a similarity measure is beyond three standard deviations from the mean of similarity measures, then it is considered an outlier. In Chapter 5, we will compare the results and the computational complexity of this implementation against the one proposed by this thesis.

## 2.5 Application Towards Sequence Data

Each method is effective for different situations and different types of data, and the user must carefully choose which outlier detection method to use. Statistical methods are solidly backed by mathematical theory, but the user must know the distribution of the data prior to using these methods. There may not be a well defined statistical distribution that fits the dataset, or the user may not know enough about the data to accurately assume a distribution. Furthermore, statistical methods cannot naturally handle data such as sequences. Sequences will have to be modified and represented in a numerical way in order to run statistical-based outlier detection.

The same is true for distance-based outlier detection such as the one by Knorr and Ng [5]. The  $DB(p,D)$  measurement is useful when the user does not know the distribution of the information, but the user must be able to calculate a distance from one data point to another. “Distance” is not a natural measurement between two sequences. Even if the user is successful at determining how to calculate a distance between two sequences, it is difficult to choose an effective  $D$  value; often the only way to choose an effective  $D$  value is through experimentation. The method in [16] avoids having to specify  $D$  by using the  $D^k(p)$  measurement, but this method only finds the top  $n$  outliers. If the user does not know how many outliers there are in the dataset, it is difficult to determine the  $n$  value.

Deviation-based approaches face similar challenges when applied toward sequences. One advantage the deviation methods have over distance-based methods is that the dissimilarity function does not have to be metrical, but that does not eliminate the challenge to determine how to calculate a dissimilarity of sequences. Other forms of deviation based approaches tend to have high computational complexities and this becomes a serious problem when databases get large.

The general approaches to outlier detection do not present themselves to be effective means to detect sequence outliers. More recent development in compression based measures for outlier detection is used specifically for sequence data, and this thesis is an extension of these developments. While recent development has concentrated on using data compression as an entropy measurement, this thesis concentrates on modifying a data compression algorithm to achieve outlier detection. It addresses many of the challenges of statistical, distance, and deviation-based methods when detecting sequence data outliers. The thesis presents a sequence data outlier detection method that can be used for any sequence data without having to know the data distribution or having to specify a distance to base outlier detection all the while maintaining a reasonable computational complexity.

## CHAPTER 3

### OVERVIEW OF DATA COMPRESSION AND LZW

#### 3.1 Outlier Detection and Data Compression

Sequence data can be formed from many different sources, but they all have the same format: a string of characters. Each sequence can vary in length, where length is the number of characters in the sequence, and vary in the number of unique characters used in the entire dataset, but at least they are strings of characters. So, what makes strings of characters different from each other? The answer lies in the patterns, which are substrings of characters found within a sequence. For example, the sequence 'outlier' has patterns such as 'ou', 'ut', 'tl'. These patterns do not necessarily have to be the same length. In addition to the patterns mentioned before, 'outlier' has patterns such as 'out', 'lier', 'outli', and 'outlier'. Now compare the string 'outsider' with 'outlier'. They both share the pattern 'out' and 'er'. If we add yet another string 'shout', all three strings share the pattern 'out'. In contrast, there are patterns that are only unique to each of the strings: 'l', 'li', 'si', 'sid', and 'h' and 'sh'. It is these uncommon patterns that help us detect if the strings are different from each other and how different they are from each other. The more a sequence has uncommon and unshared patterns, the more likely it is an outlier of the dataset. Finding these uncommon patterns effectively is the key to detecting outlier sequences.

Directly searching for uncommon patterns is a difficult task. Fortunately there is a different approach that indirectly finds uncommon patterns. The different approach is to find the common patterns and then to assume that those patterns that are not common

must be uncommon. If we find common patterns, then patterns that are not found are uncommon. Techniques to find common patterns have been well developed, especially through data compression. By exploiting data compression techniques, we can find common patterns and, at the same time, identify uncommon patterns.

### **3.2 Brief Discussion of Data Compression**

The goal of data compression is to take the existing data and represent them in a more efficient way. The efficiency of the data compression is often expressed in a ratio of the original data size and the compressed data size. To reach a high compression ratio, many compression techniques attempt to find commonality in the dataset and encode this common pattern with a smaller code. The better the compression scheme can find these common patterns in the data, the better the overall compression.

Of course, not all data compression schemes are appropriate for outlier detection of sequences. In this section, we will briefly discuss common compression techniques to find an appropriate compression algorithm to fit our strategy.

Recall that the strategy is to identify common patterns through compression techniques where a pattern is a string of characters. If the pattern reoccurs often in the sequence, then that pattern is considered common. Likewise, if the pattern does not reoccur frequently in the data, then that pattern is considered uncommon. If we are able to identify common patterns through compression techniques, we can also identify uncommon patterns. These uncommon patterns are the key in detecting outliers.

With this strategy in mind, we now narrow down the countless data compression techniques to one that fits our model the best.

### 3.2.1 Lossless versus Lossy Compression

Data compression can be divided into two large categories: lossless and lossy compression. Lossless compression compresses the data without losing any information. Compressing and then decompressing the data will result in the exact image of the original data. Data compression of text and other data where data integrity is important use lossless compression. Lossy compression on the other hand, loses some information while compressing data. Therefore, compressing the data and decompressing the data does not result in the exact image of the original data. Lossy compression is mostly used in image data compression because images can retain much information even if they partially lose their integrity [36].

Since our method is based on finding common patterns in sequences, it is important to retain information and integrity of the patterns and of the sequence; thus, the method calls for lossless compression schemes rather than lossy compression schemes.

### 3.2.2 Common Lossless Compression Techniques

#### *Huffman Coding*

The Huffman coding is one of the basic lossless compression techniques. It is based on the idea of replacing symbols that have a high probability of occurrence with shorter codes than those symbols with a low probability of occurrence. The code development follows this idea and one other requirement: the two least frequently occurring symbols have codes of the same length. Based on these two requirements, the algorithm builds codes which are often in a form of a binary tree [37].

In the general case, the Huffman coding process must know the probabilities or frequencies of each unique character in the data. With this information, the method builds

codes that meet the two requirements. These codes are then used to replace the symbols or characters for an overall smaller file size.

Unfortunately, the frequency of each unique symbol in the data may be information that is not available. In such a situation, the Huffman method can be modified to calculate the frequency of each character as the method encounters them during the encoding process, an idea first developed independently in [38] and [39]. In order to ensure that they follow the two requirements of the Huffman codes, the codes must adapt according to the recalculated frequencies. This way, the method can closely follow the true Huffman coding algorithm even without knowing the frequencies prior to the coding process. Though this modification to the Huffman coding process is convenient, the updating and adapting process adds significant overhead to an otherwise very minimal algorithm.

The Huffman coding technique can be extended in other ways too. Typically, the Huffman codes use binary codes, but this can be easily extended to non-binary codes for a broader code base. Another possible extension to the Huffman coding method is to extend from encoding one character at a time to a sequence of characters. In other words, instead of creating a code based on the frequencies of single symbols, the method can generate codes based on the frequency of a sequence of symbols. Generating codes for a sequence of symbols increases compression efficiency, because the method is able to replace multiple symbols with one code. However, the Huffman coding process must examine every possible combination of symbols in order to generate codes for them. This leads to an exponential increase in the number of codes to be generated and stored by the algorithm. This exponential growth quickly makes this extension an impractical choice.

### *Predictive Coding*

Predictive coding applies a different methodology to data compression. The process purposefully transforms the data sequence based on its history so that the newer

sequences have a desirable characteristic for data compression. The desirable characteristic is where the frequencies of symbols are severely skewed. Skewing the frequencies means that a select few symbols have high frequencies while the rest have low frequencies. This results in better compression [36].

For example, if the original dataset contained four characters A, B, C, D with equal probabilities, a possible Huffman encoding table will look like the following:

Table 3.1 An example Huffman encoding table

Character	Probability	Codeword
A	.25	00
B	.25	10
C	.25	11
D	.25	010

Since the probabilities are equal, it does not matter which letter receives which code length. If the original dataset contained 25 of each character for a total of 100 characters, the total number of bits to encode it will be

$$2 * 25 + 2 * 25 + 2 * 25 + 3 * 25 = 225bits$$

Now, if a predictive function can be applied so that A, B, C and D now have probabilities 0.09, 0.09, 0.02 and 0.8 respectively, then the Huffman encoding table will be adjusted to look like the following:

Table 3.2 Adjusted Huffman encoding table

Character	Probability	Codeword
A	.09	10
B	.09	11
C	.02	010
D	.8	00

This time, we make sure that the shortest codewords are assigned to the characters that are most frequent. If we again assume 100 total characters, then there are 9 As, 9 Bs, 2 Cs and 80 Ds. When encoded, this comes out to be

$$2 * 80 + 2 * 9 + 2 * 9 + 3 * 2 = 202bits$$

A smaller number of bits are required than previously to compress the data.

The key component in this method is the transformation of the data to have a more skewed distribution of symbols. These transformations of data are based on correctly predicting the behavior of the data. If the prediction is correct, then all the decoder needs to reconstruct a mirror image of the sequence is to calculate it through a prediction function. If the prediction is incorrect, the method somehow must indicate and record the inconsistency between the result of the prediction function and the actual value. Thus, the more accurate the prediction function, the better the compression becomes.

The predictive coding method bases its prediction on the history of the data. The history can be as short or as long as the prediction function allows it to be. Varying the size or length of the history context affects the accuracy of the prediction function, but it also could affect the spatial and the computational complexity of the method.

Additionally, there are numerous predictive functions. The function could be a simple arithmetic calculation based on the history, or it can be a complicated manipulation. Either way, the most important factor in using predictive coding is the requirement to know substantial information about the data prior to compression. We must “find the prediction approach that is best suited to the particular data we are dealing with” [36]. A prediction function for one dataset may not be appropriate for another. There are generic predictive functions available, but they will do no good if the accuracy of the prediction is mediocre.

### *Dictionary Techniques*

Another approach that is different from the previous two is the dictionary technique. This technique builds a dictionary of patterns found in the data. Patterns found in the dictionary are assigned efficient codes and patterns not found in the dictionary are encoded with less efficient codes. With this method, we are able to split, “the input into two classes, frequently occurring patterns and infrequently occurring patterns” [36]. For the method to be effective, the dictionary must be selective in what it decides to put and store in the dictionary. If the dictionary does not store enough frequent patterns, then the frequent patterns that deserve efficient codes will not receive the advantage. Similarly, if the dictionary stores too many patterns, then patterns that are not as frequent will also be assigned efficient codes.

As with the other two methods, this method has many variations. First, we can choose whether the dictionary remains static or changes and adapts to the data. Static dictionaries are effective when we have prior knowledge of what sequence of symbols will appear frequently. Data with a set format, for example, will have recurring tags and identifiers. By anticipating that they will appear frequently, we can choose short codes to represent them. If we do not have this prior knowledge, then we shift to the adaptive methods, one of which is the Lempel-Ziv-Welch, LZW, algorithm [40]. Adaptive dictionary techniques build the dictionary while encoding the data. This way, we can

build a dictionary that closely represents the distribution of the data without prior knowledge of the data.

Another factor that a dictionary method must consider is the size of the dictionary. A dictionary that is too small may not be able to capture all of the frequent patterns, but a dictionary that is too large will become less selective. Physical memory restrictions may limit the size of the dictionary as well. To cope with these factors, the dictionary techniques must define rules and requirements for how patterns are entered and removed from the dictionary, and these will factor into how well the dictionary method performs as a means of compression [36].

All of these methods are effective when they are used appropriately for the context. The key for all of these methods is to choose the method correctly. That is no different for our purpose as well.

Looking through the three methods discussed, one method stands out from the others as a comfortable fit for our goals. The Huffman coding bases its method on the frequencies of single symbols. This can be extended to frequencies of sequences of symbols at the expense of increased computational complexity. The biggest downfall of the Huffman encoding is the need to know the frequencies of symbols, and this information is not always available. Also, if there are many unique symbols and they are evenly distributed, then the Huffman coding becomes ineffective.

The predictive coding method has its own challenges as well. The effectiveness of the predictive coding depends on the accuracy of the prediction function. Unfortunately, no single prediction function works well for all possible situations. One set of sequence data may behave completely differently than another set of sequences. The prediction function must be tailored and altered to fit each model. This is very inconvenient. Moreover, the predictive coding method does not keep a record of common patterns. It alters the original data to create common patterns. This is sufficient

for compression needs, but it does not satisfy our need of knowing what patterns were common and uncommon.

That leaves us with the dictionary method, and fortunately, this method meets the requirements. First of all, dictionary methods are able to categorize data into common and uncommon patterns. This distinction is needed to identify outlier sequences, because outlier sequences are those with many uncommon patterns. In addition to the ability to categorize patterns to common and uncommon patterns, the method naturally provides a mechanism to keep track of common patterns for future reference. Furthermore, the adaptive dictionary method does not require prior knowledge of the data; the dictionary method is able to do all of this without compromising the computational complexity. All of these characteristics of the dictionary method, specifically the LZW algorithm, make it the method of choice for our strategy.

### 3.3 LZW Algorithm

The LZW compression algorithm is surprisingly simple and fast. The algorithm takes one pass of the data to create a dictionary called the translation table where patterns are mapped to a shorter code or index. The translation tables are often implemented as hash maps, or in our case a dictionary tree. The patterns are found by identifying the longest recognized pattern and putting them in the table to be assigned a code. In many implementations, the table index itself is the code. Figure 3.1 illustrates the general LZW algorithm.

Initially, the translation table is populated with all of the unique characters of the data. Then, the algorithm examines the data one character at a time. The newly read character,  $c$ , is concatenated to the *prefix*, and if that character pattern is not found in the translation table, then the index of this *prefix* is outputted. The newly read character is then concatenated to the *prefix* and added to the translation table. If the *prefix* +  $c$  pattern

is found in the translation table, *c* is added to the end of the *prefix* and the *prefix* becomes longer by one letter [40].

The following is an example of how the translation table will be populated with the given string of characters. Assume that there are only 4 possible unique characters: a, b, c, and d for this dataset and the sequence we want to encode is ‘abcabcabcabddaddabc’.

Table 3.3 illustrates the initial translation table.

Table 3.3 Initial translation table

Index	Pattern
1	a
2	b
3	c
4	d

The first step is to read in the first character, *a*, and set *prefix* = *a*. Next, we read in the following character, *b*. The concatenation of *a* and *prefix* is ‘ab’. Since ‘ab’ is not in the translation table we output 1 and enter ‘ab’ into index 5 of the translation table. Once ‘ab’ is entered in the translation table, *prefix* is reset to the last character read in from the dataset and the process is repeated. As the LZW algorithm reads more data, the patterns stored in the translation table tend to get longer. LZW achieves data compression by representing long patterns in a single index value. Table 3.4 is the resulting translation table from processing the entire sequence ‘abcabcabcabddaddabc’. The final compressed data is ‘1235765441128’.

Decompressing the LZW-compressed data is equally as simple. All the decoder needs to know is the original state of the translation table, Table 3.3, and it is able to build the exact same translation table, Table 3.4, as it is decoding the data. For the purpose of this research, we are only interested in using the LZW encoding process to

Figure 3.1 The LZW algorithm

---

LZW

```
1  Initialize the translation table to have all unique characters in the data
2  prefix = first input character
3  While there is input to read
4    c = next input character
5
6    If c does not exist                                //input exhausted
7      output index(prefix)
8      exit loop
9
10   If the concatenated string prefix+c exists in the translation table,
11     prefix = prefix+c
12     return to loop                                    // to line 3
13   else
14     output index(prefix)
15     put string prefix+c into the translation table with new code
16     prefix = c
17     return to loop                                    // to line 3
```

---

Table 3.4 The Resulting Translation Table

Index	Pattern
1	a
2	b
3	c
4	d
5	ab
6	bc
7	ca
8	abc
9	cab
10	bca
11	abd
12	dd
13	da
14	ad
15	dda

find common patterns. This does not require us to actually encode and then decode the data. As such, a detailed discussion of the decoding process is beyond the scope of this thesis and will not be covered any further.

### 3.4 Characteristics of the Patterns Found by LZW

The patterns found by the LZW compression algorithm have the following characteristics.

1. All prefixes of a pattern in the translation table are also found in the table.

For a pattern of length  $n$  to be entered into the table, the prefix of the pattern, with length  $n-1$ , must be in the table. This is true for all patterns. Therefore, it can be argued that long patterns in the translation table are common patterns in the dataset. For example, if the pattern 'abcdef' is entered in the table, patterns 'abcde', 'abcd', 'abc', 'ab', and 'a' are also in the table. This means that the pattern 'a' was found at least six times, 'ab' was found at least five times, 'abc' at least four times, 'abcd' at least three times, and 'abcde' at least two times. To generalize, for a pattern of length  $n$  to be found, the first letter of the pattern must be found at the very least  $n$  times, the pattern composed of the first two letters must have been found at least  $n-1$  times, and so on.

2. The resulting translation table is sensitive to the order of the data.

Since the LZW algorithm builds the translation table by scanning one character at a time, the LZW algorithm is sensitive to the order of the data. Scanning the following data

abcdabcdabcdabcdabcd

(Data 1)

will result in a slightly different translation table than scanning

bcdabcdabcdabcdabcdabcd

(Data 2)

See Tables 3.5a and 3.5b for the resulting translation tables.

3. Patterns in the translation table vary in length.

Unlike many compression schemes, the LZW algorithm finds patterns of varying length. Additionally, the algorithm inserts patterns of varying length at varying times. For example, in Table 3.5a, a pattern of length 4 is inserted in the table before all possible patterns of length 3 are even considered. If the pattern is common enough to grow to longer lengths, the algorithm does not have to wait until all possible patterns of one length are found to go on. This way, common patterns will continue to grow, and uncommon patterns will remain short.

4. Patterns in the translation table reflect only the patterns found in the data.

Since the patterns stored in the translation table are created from reading the data one character at a time and concatenating those characters together, the patterns in the translation table can only reflect those patterns that are actually present in the data.

5. Not all possible patterns are detected.

With the simplicity of the LZW algorithm, not all possible patterns are guaranteed to be detected. For example, Table 3.4 represents the translation table resulting from running the LZW algorithm against the sequence 'abcabcabcabddaddabc'. If we look at

Table 3.5a Resulting table of Data 1

a
b
c
d
ab
bc
cd
da
abc
cda
abcd
dab
bcd
dabc
cdab

Table 3.5b Resulting table of Data 2

a
b
c
d
bc
cd
da
ab
bcd
dab
bcda
abc
cda
abcd
dabc

Table 3.4, the pattern 'add' is not stored in the translation table, even though the pattern 'add' appears in the string. Although this seems to be an undesirable characteristic resulting directly from the LZW algorithm, the idea is that if the pattern 'add' is common enough, the pattern will eventually be detected and inserted into the translation table.

## CHAPTER 4

### THE OUTLIER SEQUENCE DETECTION USING LZW(OSDUL) PROCESS

#### 4.1 Applying the LZW Algorithm to Sequence Data

The LZW algorithm does well with sequence data such as text where it can identify common patterns. Unfortunately, the LZW algorithm, as it is, is a compression algorithm. A few modifications must be made in order for it to be applied towards outlier detection.

##### 4.1.1 Modifications to the LZW Algorithm

The important goal in using the LZW algorithm is to find common patterns. As discussed earlier, longer patterns indicate common patterns, but this information is not enough to determine which patterns are truly the most common. There could be many patterns of length  $n$ , but one of these patterns may appear more frequently than the others. To detect this difference, we must keep track of how often the LZW algorithm comes across the pattern. This can be done by simply storing frequency information along with the corresponding pattern, and then incrementing this frequency as the LZW algorithm finds the patterns again. Note that this frequency does not represent the true number of times the pattern appears in the data. As discussed earlier, the algorithm does not recognize all possible patterns, and since the frequency will only be incremented when

the LZW algorithm finds the pattern, this frequency will be slightly less than the actual frequency of the pattern in the dataset. Again, the idea is that if the pattern is frequent enough, this difference will be insignificant.

Another modification made to the LZW algorithm is the removal of the encoding steps. We are not interested in the actual encoding of the common patterns; we are merely interested in extracting them. Therefore, it is unnecessary to concern ourselves with encoding the sequence. Figure 4.1 is the pseudocode of the LZW algorithm with these modifications, and will be referred to as the OSDUL-LZW algorithm for the rest of this document. The OSDUL-LZW algorithm is capable of reading through the dataset and storing patterns in a transition table along with its frequency.

#### **4.1.2 The Source of the Pattern**

At this point, we introduce one alteration to this method. The alteration deals with repetitive patterns within a sequence data. With the current OSDUL-LZW algorithm, it does not take into account the source of the pattern. If a sequence has a repetitive pattern within itself, then the algorithm will continue to count these patterns and increase the frequency and length. A high frequency and long pattern length indicates common patterns, so this repetitive behavior will identify the pattern as a common pattern. While this may sound correct, take into consideration the possibility that this repetitive pattern only occurs in an outlier sequence. In other words, this repetitive pattern is not found anywhere else but this particular sequence.

Without the alteration, the OSDUL-LZW algorithm will count each repeated pattern that is unique to this single sequence. If frequent enough, this pattern that only repeats in one sequence will become “common” according to the OSDUL-LZW algorithm. This is certainly not the ideal outcome, since we would like the algorithm to identify this odd sequence as an outlier.

Figure 4.1 The LZW algorithm modified for OSDUL

---

### OSDUL-LZW

```

1  Initialize the translation table to have all possible unique characters and set
    frequencies=1
2  For each sequence
3    prefix = first input character
4    While there are more characters to read for this sequence
5      c = next input character
6      If c does not exist                                // input exhausted
7        increment frequency of prefix
8        exit loop
9
10     If the concatenated string prefix+c exists in the translation table,
11       increment frequency of prefix+c
12       prefix = prefix+c
13       return to loop                                  // to line 4
14     else
15       put string prefix+c into the translation table and set its frequency=1
16       prefix = c
17       increment frequency of prefix
18       return to loop                                  // to line 4

```

---

To remedy this situation, we alter the OSDUL-LZW algorithm in Figure 4.1 to store the source of the pattern in addition to the pattern and its frequency. The algorithm will still allow the pattern length to grow, but will only increment the frequency if the pattern was detected from a different sequence. For example, if the pattern ‘aaa’ was last detected by the OSDUL-LZW algorithm in sequence 1, then even if the OSDUL-LZW algorithm finds pattern ‘aaa’ in sequence 1 again, the frequency of this pattern will not be increased. If the pattern ‘aaa’ is found in a different sequence, sequence 2 for example, then the frequency is incremented.

Figure 4.2 modifies the algorithm of Figure 4.1 to accommodate this alteration. With this, the OSDUL-LZW algorithm now will build a translation table, keep track of how frequent it encounters each pattern, and avoid counting repetitive patterns within the same sequence, all in a single pass of the data.

Tables 4.1a through d show an example of how the translation table is populated with the modifications described in sections 4.1.1 and 4.1.2. The example will process the following three sequences in order: abcabcabc, abcabcabc and bbcabb.

### **4.1.3 The Trimming Process**

Considering the source sequence prevents frequencies of repetitive patterns of outlier sequences from getting excessively large. Unfortunately, it does not prevent these patterns from getting excessively long. If all of the outlier sequences of the dataset have the same repetitive patterns, those patterns are going to get as long as, or even surpass, the true common patterns. This can cause the algorithm to label these repetitive but uncommon patterns to be as “common” as the other legitimately common patterns. To prevent this situation, we introduce the trimming process.



Table 4.1a Initial translation table

Index	Pattern	Frequency
1	a	0
2	b	0
3	c	0

Table 4.1b Translation table after processing the first sequence: abcabcabc

Index	Pattern	Frequency
1	a	1
2	b	1
3	c	1
4	ab	1
5	bc	1
6	ca	1
7	abc	1
8	cab	1

Table 4.1c Translation table after processing the second sequence: abcabcabc

Index	Pattern	Frequency
1	a	2
2	b	2
3	c	1
4	ab	2
5	bc	2
6	ca	1
7	abc	3
8	cab	1
9	abca	1
10	abcab	1

Table 4.1d Translation table after processing the third sequence: bbcabb

Index	Pattern	Frequency
1	a	3
2	b	3
3	c	1
4	ab	3
5	bc	3
6	ca	1
7	abc	3
8	cab	1
9	abca	1
10	abcab	1
11	bb	1
12	bca	1
13	abb	1

The trimming process as described in Figure 4.3 is done on a regular basis to the translation table while the translation table is being built. It goes through the entries of the translation table and finds the leaf entries of the table, where leaf entries are considered those entries that are not a prefix of another entry in the translation table. For all of these leaf entries, we decrement its frequency by 1. If, as a result of this reduction, the frequency becomes 0, then that entry is removed from the translation table. The only exception to this rule is if the leaf entry is one of the initial entries of the translation table. Table 4.2 illustrates a trimmed version of the translation table illustrated in Table 4.1d.

Figure 4.3 Algorithm of the trimming process

TRIM

- 1 L = all the leaf entries in the translation table
- 2 For all entries  $L_i$  in L
- 3     frequency of  $L_i$  = frequency of  $L_i - 1$
- 4     If frequency of  $L_i = 0$
- 5         remove  $L_i$  from the translation table if  $L_i$  is not one of the initial entries
- 6         of the translation table.

Trimming introduces one parameter to an otherwise parameter-free OSDUL-LZW algorithm. The parameter is called the trimming interval,  $\tau$ , and it indicates the interval at which the trimming process is called. If the trimming interval is 25, then for every 25th sequence read in by the OSDUL-LZW algorithm, the OSDUL-LZW algorithm calls the trimming process.

Table 4.2 Modified Table 4.1d due to trimming

Index	Pattern	Frequency
1	a	3
2	b	3
3	c	1
4	ab	3
5	bc	3
6	ca	1
7	abc	3
9	abca	1
11	bb	1

The trimming process decrements the frequency of all leaf entries. This includes those that are from common patterns, but the trimming process does not affect the common patterns as much as it does the uncommon patterns. Since the trimming process is not run all the time, common patterns have the time to recover from the trimming process; during the trimming interval, it can be assumed that the common patterns will be frequently recognized by the algorithm and its length and frequency will increase. On the other hand, uncommon patterns will not be frequently recognized by the algorithm, and its frequency and length will not change during the trimming intervals. This means that the uncommon patterns will not have the time to recover. Over time, the trimming process will keep removing uncommon patterns. If the interval is too long or too short, the effectiveness of the trimming process is lost. If the interval is too long, then the uncommon patterns will not be removed enough; similarly, if the trimming interval is too short then no pattern will have the opportunity to develop and become “common.”

Overall, the effect of the trimming process is minimal to common patterns while removing uncommon patterns. This is beneficial in all cases, but this is especially important for the case where outlier sequences have repetitive patterns. Repetitive patterns from outlier sequences will eventually fade away from the translation table.

## **4.2 Consideration of Memory**

The memory usage by the translation table is always an item of concern in every dictionary based algorithm such as the LZW. The translation table will grow as long as there is data to read and new patterns are found. Without any restrictions, the translation table will continue to grow.

There are a few different strategies to restrict the size of the translation table. One is to have no restriction and allow the translation table to grow as large as it needs to get. This method avoids the risk of losing information by restricting the size of the translation

Figure 4.4 OSDUL-LZW algorithm with TRIM

---

 OSDUL-LZW(  $\tau$  )

```

1  Initialize the translation table with all unique characters, set frequencies=1 and
source = 0
2  For each sequence
3    prefix = first input character
4    While there are more characters to read for this sequence
5      c = next input character
6      If c does not exist                                // input exhausted
7        If source of prefix+c is different than current sequence
8          increment frequency of prefix
9        If sequence count is equal to  $\tau$ 
10         TRIM()
11         sequence count = 0
12       else
13         sequence count = sequence count + 1
14       exit loop
15     If the concatenated string prefix+c exists in the translation table
16       If source of prefix+c is different than current sequence
17         increment frequency of prefix+c
18       prefix = prefix+c
19       return to loop                                  // to line 4
20     else
21       put string prefix+c into the translation table and set frequency=1 and
source-sequence = current sequence ID
22       prefix = c
23       increment frequency of prefix
24       return to loop                                  // to line 4

```

---

table, but it carries the risk of storing too much information, such as short and infrequent patterns.

Other strategies all require some sort of replacement guidelines, such as replacing the last pattern inserted or the least frequent pattern. Replacing the last pattern inserted into the translation table is easy to implement, but it is impractical for many cases because it may be overwriting a frequent pattern that we would like to keep in our translation table. A better approach is to replace the least frequent pattern.

For the experiments in this thesis, we assume that there will be infinite space available for the translation table. Depending on the size of the data, it may be impossible to have such an option, in which case the replacement strategies must be considered. The trimming process does have the potential of removing entries from the translation table, but it does not guarantee removal so it is not considered a restrictive strategy.

### **4.3 Score Assignment to Patterns**

Now that we have a process to build the translation table, we must find a way to use this information towards outlier detection. Before proceeding, reviewing the important characteristics about this translation table and the information contained in it will help determine a method to detect outliers.

The translation table is a table of patterns found by the OSDUL-LZW algorithm. For a pattern of length  $n$  to be found by the OSDUL-LZW algorithm, its prefix, characters 0 through  $n-1$  must also have been found previously. This suggests that the length of the pattern is an indicator of how common the pattern is in the data; the longer the pattern, the more it appears in the data.

The translation table is also set to store approximate frequency information. The frequency is incremented every time the OSDUL-LZW algorithm encounters the pattern.

Therefore, common patterns in the data that appear frequently will have a high frequency count.

By combining these two characteristics of the translation table, we can define what it means to be a common pattern. We will define a common pattern as a pattern with a high frequency and a long length. For example, a pattern with frequency  $m$  can be said to be a more common pattern than patterns with frequency less than  $m$ . Similarly, a pattern with length  $n$  can be said to be a more common pattern than patterns with length less than  $n$ . This means that we can sort the entries in the translation table according to pattern frequency and length to see which patterns in the translation table are the most common and which patterns are the least common. Specifically, all the patterns are first sorted by their frequency. Then, patterns with the same frequency will be sorted by the pattern length. Thus, the longest pattern with the highest frequency will be the most common pattern of the data found by the algorithm.

Table 4.3 An example of a sorted translation table

Pattern	Frequency
a	30
b	20
c	9
aba	4
ba	3
abaa	1
bb	1

With this sorted translation table, we now have a relative scale of common patterns. If we were to pick out two patterns from the translation table, we can say which pattern is more common than the other by observing its pattern length and frequency.

But, how much more common is another question. Just being able to tell which one is more common is not sufficient. To quantify how common a pattern is, we now look into assigning a concrete numerical value, called score, to each pattern according to the pattern's length and frequency.

### **4.3.1 Method 1: Assigning a Bit Length**

This method completes the idea of using compression methods for outlier detection by thinking of the score as a number of bits required to “encode” this pattern. Data compression is achieved by replacing common patterns with a shorter code to represent the patterns. Obviously, the larger the difference between the original pattern and the code, the more the data can be compressed.

Good data compression is achieved when the most common patterns are given the shortest code lengths. We extend this idea to our application by simply assigning the shortest bit lengths to the most common patterns and the highest bit lengths to the least common patterns.

Now, recall that the translation table is already sorted by pattern frequency and then pattern length where the pattern with the highest frequency and the longest length is considered the most common. This method goes down this sorted translation table, from most common to least common, and assigns a bit length.

The bit length is an integer representing the number of bits needed to encode this pattern, ignoring all complications of the decoding process. The bit length starts at 1 and increases by 1 in increments of powers of two. For example, the first two patterns will be assigned a bit length of 1, the next four patterns will be assigned a bit length of 2, the next eight patterns will be assigned a bit length of 3 and so on. It is incremented by powers of two, because each added bit increases the amount of possible values it can represent and the amount it increases by is in increments of powers of two. For example,

a single bit can represent two different values, 0 and 1. An addition of a second bit will increase this to four different values: 00, 01, 10, and 11. Furthermore, adding a third bit will allow for eight different values: 000, 001, 010, 011, 100, 101, 110, and 111.

This method naively depicts processes of a true compression scheme. In a true compression scheme, complications arise in this simple application because it is difficult to differentiate whether the bit string 000 is the concatenation of 0 and 00, or if it is a single code of 000. Fortunately, we do not have to concern ourselves with this complication because our intent is not to encode the patterns in the translation table. Any complications related to the process of encoding and decoding the data can be ignored.

As a result of this method, the two most common patterns in the translation table are assigned the value of 1, the next four most common patterns in the translation table are assigned the value of 2, and so on. This achieves the goal of assigning small numerical values to the common patterns, and larger numerical values to those less common.

The final step is to reset the scores of those patterns that have a length of 1. Again, think of scores as the number of bits required to encode the data. If we are replacing a single letter character with a single letter pattern, we have not achieved compression. To retain this relationship, we will reset the scores of patterns with length of 1 to the minimum number of bits originally required to encode the data. That is, if the original data has five unique characters, then 3 bits are required to represent all five characters. Thus, the scores of the single letter patterns are reassigned to be 3. Table 4.4 is an example of a sorted translation table with scores. In this case, we have three unique characters a, b and c, so their scores are reassigned to 2. Patterns a and b were originally assigned a score of 1 because they were the two most common patterns.

Table 4.4 An example of a sorted translation table with scores based on Method 1

Pattern	Frequency	Score (Method 1)
a	30	2
b	20	2
c	9	2
aba	4	2
ba	3	2
abaa	1	2
bb	1	3

### 4.3.2 Method 2: Scaled Scores

Although method 1 completes the idea of compression, it is not always a fair quantification of how common a pattern is compared to the others. Because the bit lengths are increased by 1 in increments of powers of two, a lot of patterns receive the same bit length as the number of patterns in the translation table gets large. For example, a translation table has 1000 patterns. Then the bit length will have to reach to 9 for all patterns to be assigned a score. This means all patterns from the 511<sup>th</sup> pattern to the 1000<sup>th</sup> pattern will be assigned a bit length of 9. If the frequency and pattern length of the 511<sup>th</sup> pattern is drastically higher than the 1000<sup>th</sup> pattern, then the algorithm is not assessing the 511<sup>th</sup> pattern fairly; the 511<sup>th</sup> pattern should be quantified to be more common than the 1000<sup>th</sup> rather than to be scored the same value.

Method 2 addresses this issue by scaling each frequency to the highest frequency in the translation table and then taking the log to dampen the difference between the highest frequency and the lowest frequency. In method 2, the score of pattern P is calculated in the following way:

$$Score(P) = \log\left(\frac{\max(frequencies)}{frequency(P)}\right) \quad (4.1)$$

where  $\max(frequencies)$  is the maximum of the frequencies found in the translation table and  $frequency(P)$  is the frequency of pattern  $P$ .

With this function, the patterns with the highest frequencies will be recorded as a frequency of 0, because  $\log(1) = 0$ . A low frequency of  $P$  results in a high score. Thus, we have a scheme where common patterns receive low scores and uncommon patterns receive high scores. As with method 1, patterns of length of 1 are reset to the number of bits originally required to represent all unique characters for the same reason described in Section 4.3.1.

This function does not take into account the actual length of the pattern. The method assigns patterns the same score if they have the same frequency, which means that even if the patterns differ in length, they still have the same score. Even so, longer patterns with the same score as shorter pattern have an advantage. Consider the score to be analogous to the number of bits required to encode the pattern. If two different length patterns were encoded with the same number of bits, then the longer pattern of the two is compressed more efficiently than the shorter pattern. For example, in Table 4.5, patterns “abaa” and “bb” have the same frequencies and scores, but pattern “abaa” is four characters long while pattern “bb” is two characters long. Originally pattern “aaba” was encoded using 2 bits per character, so it would have been 8 bits long. Now, the score is 3.40, which is more than half the original size. The original number of bits to encode pattern “bb” was 4 bits long but now the score is 3.40. This is not nearly a reduction in the number of bits as it is for pattern “aaba.” In conclusion, long patterns with the same score as short patterns provide better compression than short patterns, indicating that longer patterns are more common relative to short patterns.

Table 4.5 An example of a sorted translation table with scores based on Method 2

Pattern	Frequency	Score (Method 2)
a	30	2
b	20	2
c	9	2
aba	4	$\log(30/4) = 2.01$
ba	3	$\log(30/3) = 2.30$
abaa	1	$\log(30/1) = 3.40$
bb	1	$\log(30/1) = 3.40$

#### 4.4 Calculating a Compression Ratio of a Sequence

Up to this point, the focus has been on finding, defining and quantifying the common patterns in the translation table created from the data. Now, with the two methods to calculate a pattern's score, we can use it towards detecting outliers.

Both methods of calculating the score have one crucial detail in common. The value of the score is inversely proportional to how common the pattern is relative to the other patterns in the translation table. If the pattern's score is low, it means that the pattern is a common pattern relative to the other patterns. Conversely, if the pattern's score is high, the pattern is an uncommon pattern relative to the other patterns.

To determine outlier sequences in the data, we extend this score idea from the patterns to the sequences. Sequences can be represented by a concatenation of patterns listed in the translation table. Since the translation table is initially filled with all possible unique characters in the data, the translation table is guaranteed to have a set of patterns to match the entire sequence. In an ideal situation, common patterns found in the

translation table will map well to a common sequence. An outlier sequence, on the other hand, will not have many of these common patterns identified in the translation table and will have a harder time finding patterns to match.

If a sequence can be represented in a concatenation of patterns, and these patterns each have a score, then these scores can be used to calculate one numerical value to represent the sequence. This will make it easier to compare the sequences to each other. This value will be calculated with a function based on the sum of the scores of the individual patterns found in the sequence. Since common patterns are assigned small scores, the sum of the scores of a common sequence with many common patterns should also be low. This logic applies for the opposite. Uncommon patterns are assigned large scores, so the sum of the scores of an outlier sequence with many uncommon patterns should be high.

To find the matching patterns of the sequence, we again use the OSDUL-LZW algorithm. By using the same OSDUL-LZW algorithm to map the sequence with patterns, we are being consistent in the way we find patterns. The algorithm will first read the first character of the sequence. If that character is in the translation table, then the algorithm sets that as a *prefix* and reads in the next character. Again, the algorithm checks whether the concatenation of the *prefix* and the newly read character is in the translation table. The algorithm continues this process until it is not able to find the pattern in the translation table. Then, the score of the recognized pattern, the *prefix*, is added to the overall score of the sequence. For example, if Table 4.5 is the translation table and the sequence is “abaabbababc”, then the patterns recognized are “abaa”, “bb”, “aba”, “bb”, and “c”. The sum of their scores is 10.81. This is called the FIND\_MATCHING\_PATTERNS process and its detailed outline is described in Figure 4.5.

In FIND\_MATCHING\_PATTERNS, the overall ratio is calculated by dividing the sum by the length of the sequence. By dividing the sum by the length of the sequence, length variations of sequences become obsolete.  $k$  is a constant representing the minimum

number of bits needed to represent all the unique, single-letter characters present in the data. For example, if the data were allowed the values 0 to 9, then there are a total of ten unique single characters. To represent ten unique values, you need at least 4 bits. Thus, for this example,  $k$  is 4. The product of this constant,  $k$ , and the number of characters in the sequence represents how many bits were needed originally to represent this sequence. By dividing the sum of the scores by  $k \cdot \text{length}(\text{sequence})$ , we are calculating a compression ratio. A compression ratio of 1 means the number of bits before and after compression is the same. If the compression ratio is greater than 1, then the compression scheme actually increased the bit size of the data, and if the compression ratio is less than 1, then the compression was able to decrease the bit size. A small compression ratio indicates efficient data compression.

In actuality, a compression ratio often expresses the percentage of the data reduced using the compression technique [36]. In other words, a compression ratio is  $1 - (\text{sum} / (k \cdot \text{length}(\text{sequence})))$ , but for simplicity, we will continue calling the value  $(\text{sum} / k \cdot \text{length}(\text{sequence}))$  the compression ratio.

In the context of our goal to detect outlier sequences, the compression ratio represents how well the sequence “compressed” against the translation table of common patterns. If the sequence contains a lot of the common patterns registered in the translation table, then it means that the sequence is common. Common patterns in the translation table are assigned low score values; therefore, the sum of common patterns in the translation table will also be low relative to sequences with many uncommon patterns. As a result, the compression ratio of the sequence will be small if the sequence contains a lot of common patterns. In contrast, sequences with uncommon patterns will have high compression ratios. Unusually high compression ratio indicates that the sequence had many uncommon patterns, which implies an outlier sequence.

## 4.5 Outlier Detection Using the Compression Ratio

We know that sequences with unusually high compression ratios are likely to be outlier sequences. The challenge now is to determine at which threshold to determine what values are unusually high relative to the other compression ratios of the sequences.

Assuming that the dataset contains many common sequences and less outlier sequences, there are going to be more sequences with low compression ratios than those sequences with high compression ratios. To determine what it means to be an unusually high compression ratio, we use two different methods that exploit this uneven distribution of the compression ratios.

### 4.5.1 Method 1: K-Means Clustering

This method uses a simple method of outlier detection called the K-Means clustering algorithm. The K-Means clustering algorithm partitions the data into clusters. It identifies clusters by grouping data points that lie around similar values. Similarity between each data point is calculated by how close it is to the cluster's mean. When compared with the other clusters' means, the data must be closest to the mean of the cluster it is assigned to. Using an iterative process, the algorithm calculates a mean to each cluster, reassigns the data to each cluster, and recalculates the mean. Once the data stabilizes, the K-Means algorithm has identified clusters of data that are similar to each other [41].

The K-Means clustering algorithm takes in the number of clusters that the algorithm is to find as an input. In many cases, implementations of the K-Means clustering algorithm adds a limit to the number of times the algorithm iterates to avoid infinite loops.



We use this K-Means clustering algorithm to group the compression ratios of sequences. As stated before, we assume that there are more common sequences than there are outlier sequences. Furthermore, we know that the outlier sequences are those with the highest compression ratios. If we were to correctly identify and separate the outlier sequences from the common sequences, then the mean of the compression ratios of the outlier sequences will be higher than those of the common sequences. So, if we assign one cluster to have a low mean and another cluster to have a high mean, then sequences that fall into the cluster with the high mean will be our outlier sequences.

To make sure that we are not labeling common sequences with relatively high compression ratios as outliers, we identify three means: a very low mean, a mid-level mean, and a high mean. Initially, these means are set to the quartile values of the compression ratios. The algorithm will go through and adjust these means to best fit the data. At the end, the sequences will be divided into common sequences with low compression ratios, common sequences with relatively high compression ratios, and outlier sequences with very high compression ratios.

#### **4.5.2 Method 2: Deviation-Based Method**

This method exploits the fact that outliers are only at the high end of the compression ratios, and they are usually considerably higher than the highest scoring common sequence. It compares the smoothing factor of a set of sequences as discussed in Chapter 2 [6]. Since we are assuming that there are more common patterns, we can expect to see most of the compression ratios of the sequences to be around the same range. Outliers, on the other hand, will have high compression ratios, and they will be sparse because they are not as numerous as the common sequences. Therefore, by removing the outlier sequences from the dataset, the smoothing factor should significantly increase the similarity and uniformity of the data. In other words, the

Figure 4.6 K-Means clustering algorithm

---

**K-MEANS(k)**

- 1 Arbitrarily choose k ratios from the dataset as the initial cluster means
  - 2 For each ratio
  - 3     Calculate the difference/distance from the ratio to each of the cluster means
  - 4     assign/reassign the sequence to the cluster in which the difference from its ratio to the mean was the smallest
  - 5     For each cluster
  - 6         recalculate the mean of the cluster
  - 7         assign the recalculated value as the mean of the cluster
  - 8     If the cluster mean values changed
  - 9         return to loop // to line 2
  - 10    else
  - 11        exit loop
-

smoothing factor will show significant improvement when all the outliers are removed from the dataset. By comparing the smoothing factors at each point, we can find a point in the dataset that divides outliers from the common sequences.

If we set  $I = \{a_1, a_2, \dots, a_n\}$  where  $a_1 \geq a_2 \geq \dots \geq a_n$ ,  $I_j \subseteq I$ ,  $D(I_i)$  = variance of  $I_j$  and  $C(I_j)$  = cardinality of  $I_j$  ( $= |I_j|$ ) then the pseudocode for the deviation based method is illustrated in Figure 4.7. Of the calculated SF values from the deviation based algorithm, we identify the largest one. Then, all sequences in  $I_j$  are outliers. That is, all sequences up until the  $j$ th highest compression ratios are all outliers.

Figure 4.7 Deviation-based algorithm

---

#### DEVIATION\_BASED

- 1 Sort the ratios in non-increasing order
  - 2 For  $j=1$  to  $n-1$
  - 3      $I_j = \{a_1, a_2, \dots, a_j\}$
  - 4      $SF(I_j) = C(I - I_j) * (D(I) - D(I - I_j))$
- 

This method works well for our application because it is easy to compute the variance of the compression ratios,  $D(I)$ . If the sequences were to be measured in different ways, calculating the variance can become a challenge all on its own. For instance, if the sequences' measurements were based on more than one factor, then it becomes more difficult to quantify a variance of the sequence. Furthermore, this algorithm only considers one tail of the distribution, namely the high end. This also works well with our application because our outliers all are at the high end of the spectrum.

## 4.6 The OSDUL Process

The entire OSDUL process is a four step process. It first builds the translation table using the modified and altered OSDUL-LZW algorithm. It then assigns a score to each of the patterns stored in the translation table. Using these score values, the OSDUL process calculates a compression ratio for each sequence, and where a high compression ratio indicates that the sequence did not compress well against the overall population. The final step is to accurately separate the outlier sequences from the non-outlier sequences. Each of these processes is conceptually simple and easy to implement, and when they are put together, they create a method for outlier sequence detection.

From the experiments conducted in Chapter 5, scaled scoring discussed in Section 4.3.2 combined with K-Means discussed in Section 4.4.1 reported consistently better results. Thus, for the rest of this thesis, we will use scaled scoring to score the patterns and K-Means clustering to detect outliers.

## 4.7 Computational Complexity

One of the motivations to use the LZW algorithm for outlier detection was its simplicity. As a result, the OSDUL process has a relatively low computational complexity.

A key factor that makes any LZW algorithm efficient is how the translation table is implemented. The translation table is commonly implemented as a hash table or a prefix tree. For this thesis, we implemented it using a prefix tree and we will refer to it as the pattern tree. In this tree, there is a root node without any information. When

Figure 4.8 The OSDUL Process

---

```

OSDUL(Dataset D,  $\tau$ )
1  Open Dataset D to be accessible by OSDUL-LZW
2  OSDUL-LZW( $\tau$ ) = translation table T                                // Figure 4.4
3
4  // Scaled Score Assignment
5  Find largest frequency in T = max_frequency
6  For each entry p in T, score = Score(p)                            // Eq. 4.1
7
8  Reset Dataset D to be read from the start
9  FIND_MATCHING_PATTERN                                                // Figure 4.5
10
11 Sort each sequence based on its ratio
12 run K-MEANS(3) using the ratios calculated from FIND_MATCHING_PATTERN
13 Sequences with ratios in the cluster with the highest mean are outliers

```

---

initializing the translation table, one unique character or symbol in the dataset is inserted as a child of this root node. All other characters are linked to this child as its sibling. Thus, each node in the pattern tree has a parent, and can potentially have a sibling. If a node does not have a child, then it is considered a leaf node. Figure 4.1 is an illustration of an example pattern tree created by OSDUL-LZW. The letters represent the characters of the pattern and the numbers beside it represent frequency. Figure 4.9 shows that ‘aa’ has a frequency of 2, ‘ab’ has a frequency of 1, ‘abc’ has a frequency of 1, ‘baba’ has a frequency of 1, and so on. Leaf nodes are the last nodes of patterns ‘aaaa’, ‘aaab’, ‘aab’, ‘abc’, ‘baba’, ‘baa’, and ‘cb’.

This data structure is important to efficiently search the translation table for an existing pattern. By using a pattern tree, we can make sure that the search will only take a maximum of  $O(A)$  where  $A$  is the size of the alphabet: the number of unique characters in the dataset. As the OSDUL-LZW algorithm reads in characters from the dataset, it must see if the concatenation of the *prefix* and the new character is in the tree. From the previous iteration, we know that the *prefix* is in the tree because the OSDUL-LZW found it or inserted it into the tree. With the pattern tree, we can also save the location of the tail node of this prefix. This allows us to avoid having to search the tree for the concatenation of the *prefix* and the new character; rather, we have already found the *prefix* and all we have to do now is to see if the new character follows the *prefix*, which will take  $O(A)$  time. For example, we have a pattern tree as illustrated in Figure 4.1, the *prefix* is 'ab' and the newly ready character is 'c'. From the previous iteration, we know that pattern 'ab' is in the pattern tree. Furthermore, we have a pointer pointing to the 'b' node of the 'ab' pattern. Now, when we search for the concatenation of 'ab' and 'c', or 'abc', all we have to search for is the new character 'c' following the 'b' node. In Figure 4.1, the *prefix* 'ab' does have a 'c' node following it, and the search returns true. If the search returns false, then the algorithm will add a new node under the 'ab' *prefix*. The maximum number of nodes any parent node, and any *prefix* can have is the number of unique single letter characters in the dataset, or  $A$ . Thus, each search process of the OSDUL-LZW is guaranteed to be  $O(A)$ .

Using this pattern tree, we can now analyze the computational complexity of the OSDUL process. Set  $S$  equal to the number of sequences in the dataset,  $A$  to be the size of the alphabet, and  $m$  to be the average length of a sequence  $s$  in  $S$ . To build the pattern tree using the OSDUL-LZW algorithm, it must read in  $m*S$  characters. Upon reading in the characters, it must search the pattern tree, which is  $O(A)$ . The rest of the operations are trivial. Thus, building the tree takes  $O(A*m*S)$ .

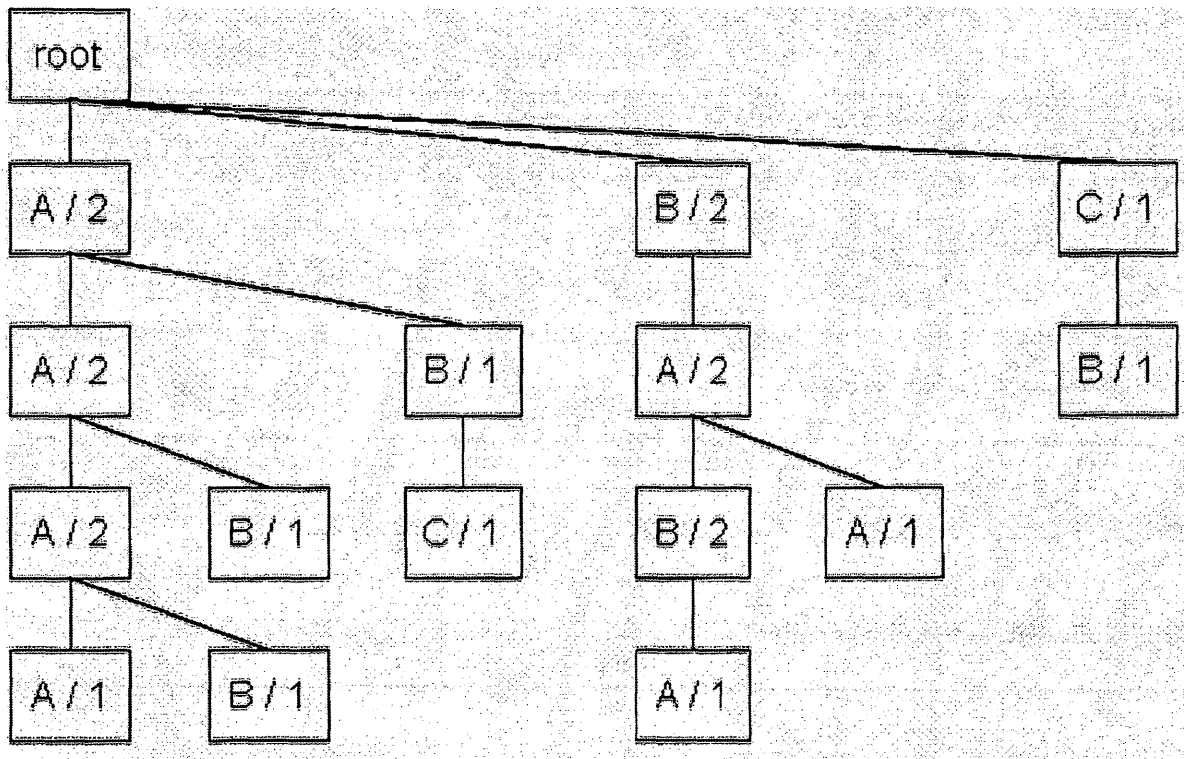
This does not take into consideration the trimming process. The trimming process is done at every trimming interval,  $\tau$ . To trim, we must go through the entire pattern tree

to find leaf nodes. Since the trimming process is done while the pattern tree is being built, the tree size is variable. We will set  $\alpha$  to be the proportion of the tree size at any point. The tree size is also proportional to the number of characters the OSDUL-LZW algorithm reads in. Thus, the trimming process takes  $\alpha*m*S$  number of operations, and it is done  $S/\tau$  times during OSDUL-LZW. Combining all this together, the computational complexity of OSDUL-LZW is  $O(\gamma*m*S^2)$  where  $\gamma$  is a constant equal to  $\alpha/\tau$ .  $\gamma$  is a small value since  $\alpha$  is less than 1 and  $\tau$  is greater than 1.

After completing the OSDUL-LZW procedure, we now must find the highest frequency within the pattern tree. This is relative to the tree size,  $O(\alpha*m*S)$ , but can be reduced to  $O(A)$  knowing that all parent nodes have equal or higher frequencies than their children. This is because the pattern tree is a prefix tree and for a pattern to be discovered, the prefix must also have been discovered. Consequently, the highest frequency will be at the highest level of the pattern tree, and there will be at most  $A$  nodes at the highest level. Since  $O(A)$  is insignificant in relation to the computational complexity of OSDUL-LZW, we will ignore it. FIND\_MATCHING\_PATTERN algorithm is very similar to OSDUL-LZW, thus the computational complexity is the same as building the tree. Sorting the ratio values of the sequence can be done in  $O(S*\log*S)$  using the quicksort algorithm [42]. This too is insignificant when compared to the complexity of OSDUL-LZW that is dominated by the  $S^2$  term and can be ignored.

The final step in the OSDUL process is the K-Means algorithm. It must look at all the ratios of the sequences and compare them with  $k$  mean values. This repeats  $n$  times until the clusters do not change, resulting in a computational complexity of  $O(k*S*n)$ . For the OSDUL process, we set  $k$  to 3 for low, medium, and high means. The average  $n$  value was less than 15 iterations for experiments conducted in Chapter 5, indicating that the K-Means process has a reasonably constant time complexity. Again, compared to the OSDUL-LZW process, this can be ignored.

Figure 4.9 Example pattern tree created by OSDUL-LZW algorithm.



As a result, the computational complexity of the OSDUL process can be said to be  $O(A*m*S) + O(\gamma*m*S^2)$ .

## CHAPTER 5

### EXPERIMENTS AND RESULTS

To test the effectiveness of the OSDUL process, we conducted several experiments using protein sequences as our data. These protein sequences are from the Pfam database publicly available through the internet [43]. The Pfam database is a large multiple alignments of common domains and families of proteins. The proteins in this database are sorted into families using protein sequence alignment processes discussed briefly in Chapter 2. Having the proteins classified into families make it an excellent dataset to test the effectiveness of the algorithm, because we can safely assume that a protein from a one family will be considered an outlier compared to proteins from a different family. A sample protein that has been preprocessed for OSDUL's use may look like the following: P I P K A R R P E G K T W A Q P G Y P W P L Y G N E G C G W A G W L L S P R G S R P S W G P T D P R R R .

For the experiments covered in this chapter, we will refer to three protein families collected from the Pfam database: HCV, RubellaE1, and TetRN. The protein sequences found in these families differ in average length and characteristics. The protein sequences used from the HCV family have an average of 109 characters per sequence, but they range from 37 characters to 114 characters long. Sequences from the RubellaE1 family average 398 characters per sequence, and range from 41 to 496 characters. TetRN family sequences average 46 characters per sequence and range from 15 to 50 characters long. Though the average lengths differ, the range is wide enough that looking at the length of the sequence itself is not a good indicator. Each family has its own set of common patterns, some of which are easy to see just by glancing at the data, and

accurately identifying these patterns is the key to successfully run outlier detection on the data.

Since the data is already sorted out into their families, the data can be considered to be free of outliers. We will mix portions of each of the families together to create a controlled dataset where we know exactly which sequences in the dataset are outliers. Having this control is important when we are assessing the accuracy and efficiency of the OSDUL process.

For all experiments in this chapter, except for the experiment described in Section 5.4, the trimming interval was set at 10. This value was used as it resulting in the best results for this set of experiments.

## 5.1 Single Family Outlier Detection

For the first experiment, we create 10 datasets containing 380 sequences from Family A and 20 sequences from Family B. These sequences are selected from the families randomly without replacement. For each dataset, we will run the OSDUL process and see if all 20 of the sequences from Family B are detected as outliers in relation to Family A. We create these 10 dataset for all combinations of HCV, RubellaE1 and TetRN. Through this experiment, we will be able to see if the OSDUL process can indeed find outliers from a dataset.

To assess the performance of the OSDUL process, we calculated four accuracy measurements: sensitivity, specificity, precision, and accuracy. Sensitivity measures how well the method recognizes outliers, specificity measures how well the method recognizes non-outliers, and precision is the percentage of the method correctly identifying outliers. Finally, accuracy is the overall ratio of correctly identified outlier and non-outlier proteins. If we set the number of outliers in the dataset as *positive*, the number of non-outliers as *negative*, number of outliers correctly identified by the method

as  $T\_positive$ , the number of non-outliers correctly identified as non-outliers by the method as  $T\_negative$ , and the number of non-outliers falsely identified as outliers by the method as  $F\_positive$ , then sensitivity, specificity, precision and accuracy can be calculated with the following equations [44].

$$sensitivity = \frac{T\_positive}{positive} \quad (5.1)$$

$$specificity = \frac{T\_negative}{negative} \quad (5.2)$$

$$precision = \frac{T\_positive}{T\_positive + F\_positive} \quad (5.3)$$

$$accuracy = \frac{T\_positive + T\_negative}{positive + negative} \quad (5.4)$$

For outlier detection, the sensitivity and accuracy are the most important of these measurements, because they assess how well the process was able to correctly detect outliers. If, in addition to a high sensitivity and accuracy, the process has a high specificity and precision, the process is superior in correctly detecting outliers while and non-outliers.

Figures 5.1 through 5.6 are graphical representations of the cumulative results. For cases when HCV and RubellaE1 are Family A, the dominant family, the divide between proteins with very low compression ratios and proteins with very high compression ratios is very definitive. Tables 5.1 through 5.4 report the average results which indicate that the OSDUL process was able to detect 100% of the outliers for both cases. Furthermore, the process was able to correctly label the rest of the proteins as non-outliers.

When the combination contains the TetRN family, the results are not as definitive. Figures 5.3 and 5.4 show that when TetRN is not of the dominant family, the frequencies of the higher ratios are very spread apart. On the other hand, Figures 5.5. and 5.6 show that when TetRN is the dominant family, the frequencies do not have a distinct separations between low and high ratios. This happens because the TetRN family has a lot of variation within its own family. Thus, when TetRN proteins are compared against patterns found in the other dominant family, the ratios of the TetRN proteins vary a lot. Similarly, when TetRN proteins are used to create the translation table, the OSDUL process cannot find quality patterns and TetRN proteins struggle to “compress” against its own patterns. Nonetheless, the results in Tables 5.5 and 5.6 show that the OSDUL process is able to handle this situation well.

Datasets like the TetRN family pose a strong argument for the trimming procedure of the OSDUL process. Figures 5.7 and 5.8 are the results from the same experiments as the ones in Figures 5.5 and 5.6, but without trimming the translation table. Notice in Figures 5.5 and 5.6 that there still is a visible distinction between the middle ranged ratio values and the high ranged ratio values, but in Figures 5.7 and 5.8, there is no distinction at all. As a result, the OSDUL process was not able to detect any of the outliers correctly and it labeled over half of the non-outlier sequences as outliers. This improvement justifies the trimming process’s added complexity.

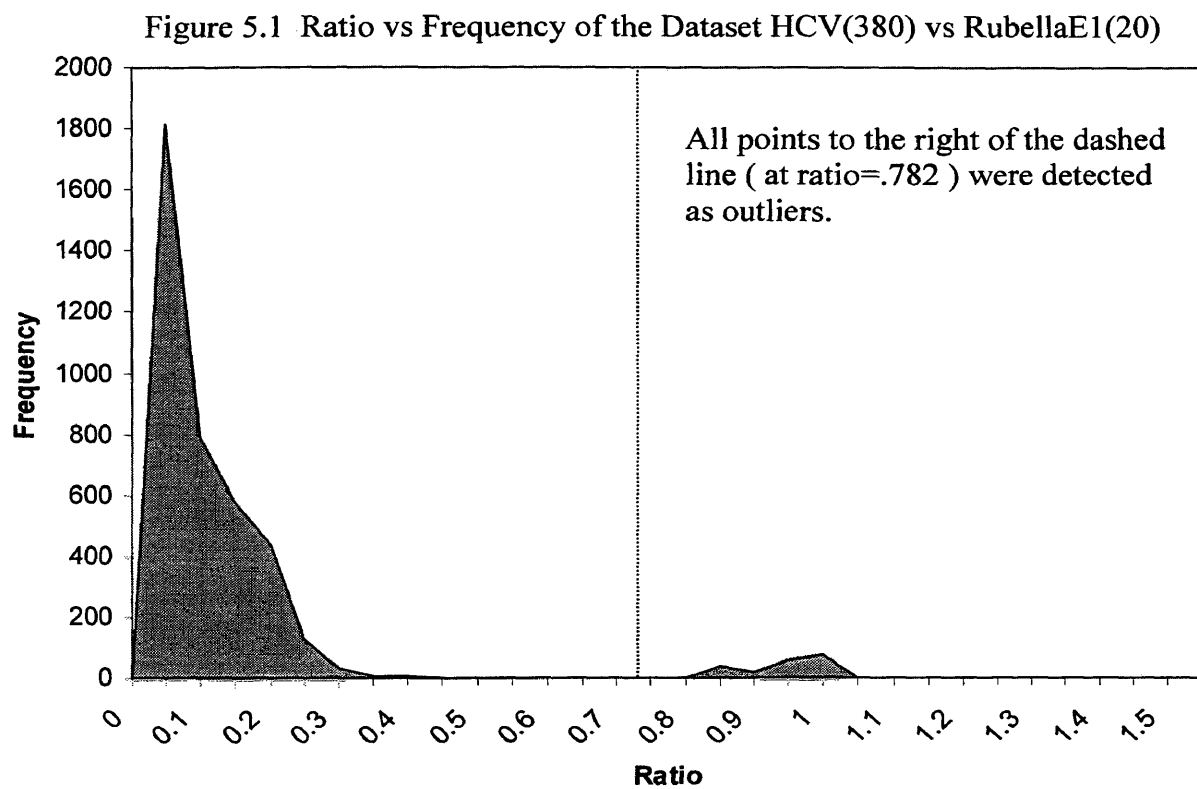


Table 5.1 HCV(380) and RubellaE1(20) average accuracy measurements

sensitivity	1.000
specificity	1.000
precision	1.000
accuracy	1.000

Figure 5.2 Ratio vs Frequency of the Dataset RubellaE1(380) vs HCV(20)

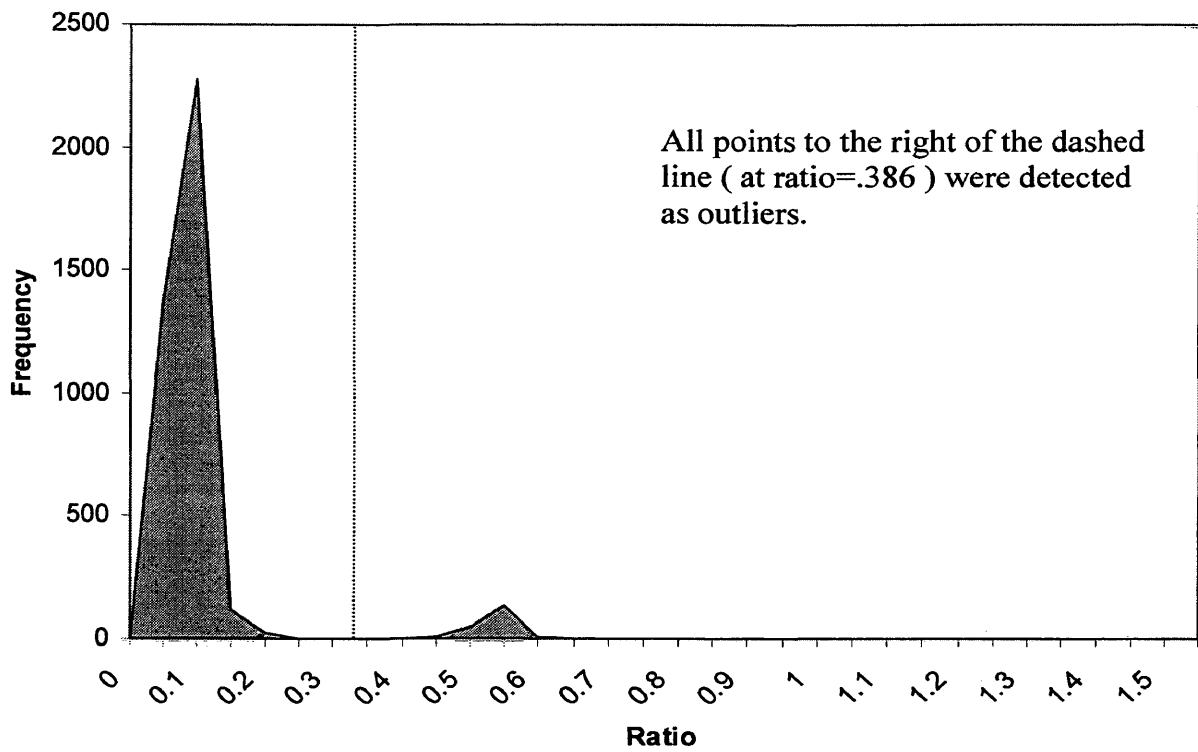


Table 5.2 RubellaE1(380) and HCV(20) average accuracy measurements

sensitivity	1.000
specificity	1.000
precision	1.000
accuracy	1.000

Figure 5.3 Ratio vs Frequency of the Dataset HCV(380) vs TetRN(20)

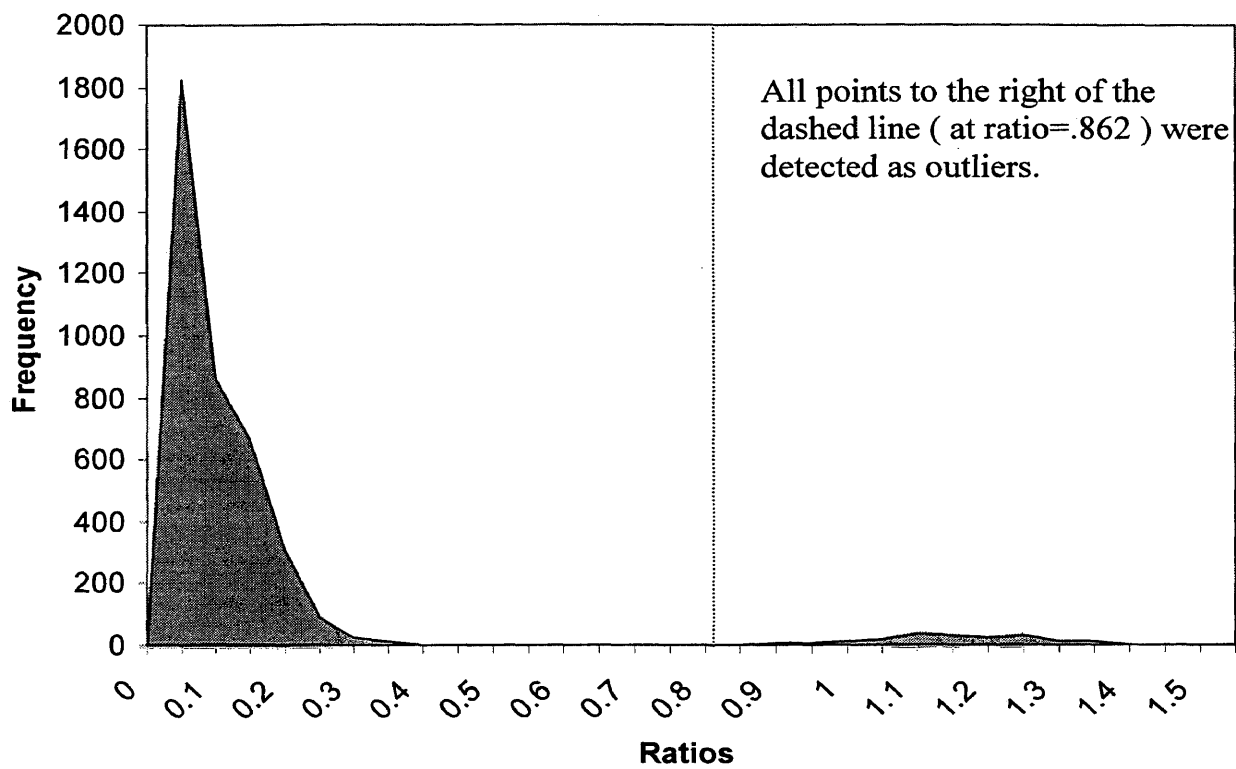


Table 5.3 HCV(380) and TetRN(20) average accuracy measurements

sensitivity	1.000
specificity	1.000
precision	1.000
accuracy	1.000

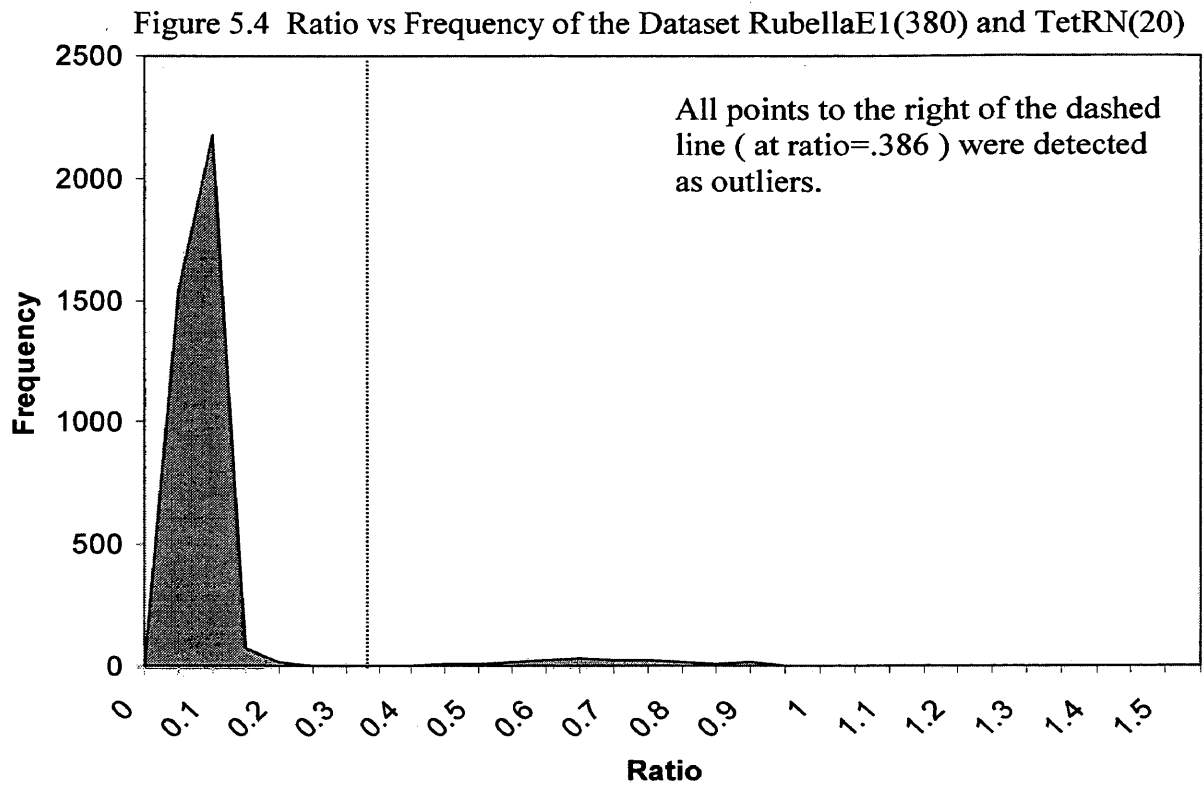


Table 5.4 RubellaE1(380) and TetRN(20) average accuracy measurements

sensitivity	0.970
specificity	1.000
precision	0.970
accuracy	0.999

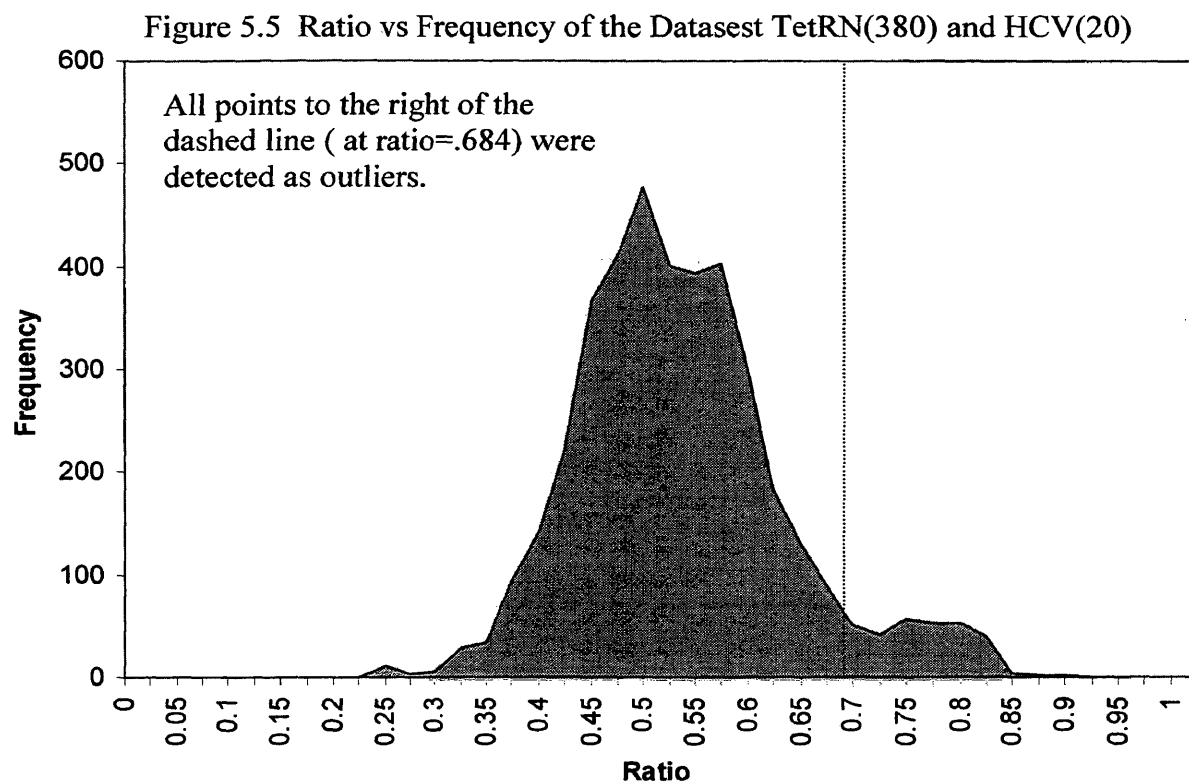


Table 5.5 TetRN(380) and HCV(20) average accuracy measurements

sensitivity	0.970
specificity	0.986
precision	0.970
accuracy	0.985

Figure 5.6 Ratio vs Frequency of the Dataset TetRN(380) and

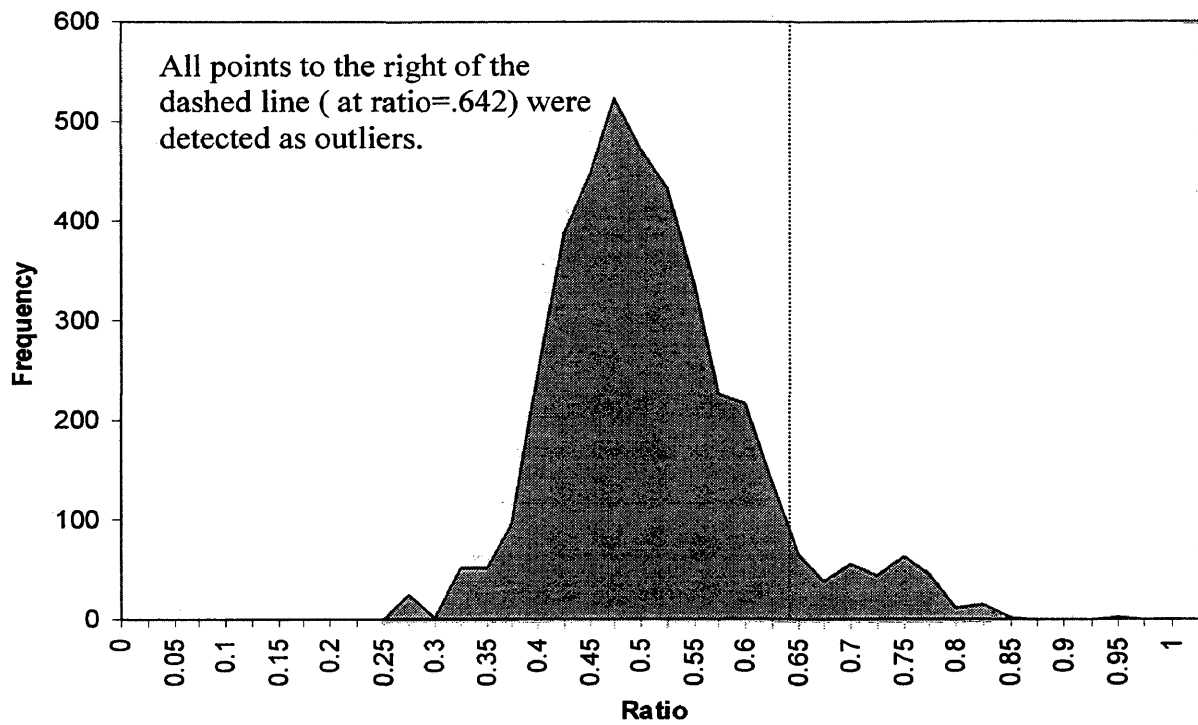


Table 5.6 TetRN(380) and RubellaE1(20) average accuracy measurements

sensitivity	1.000
specificity	0.977
precision	1.000
accuracy	0.978

Figure 5.7 Ratio vs Frequency of the Dataset TetRN(380) and HCV(20)  
No Trimming

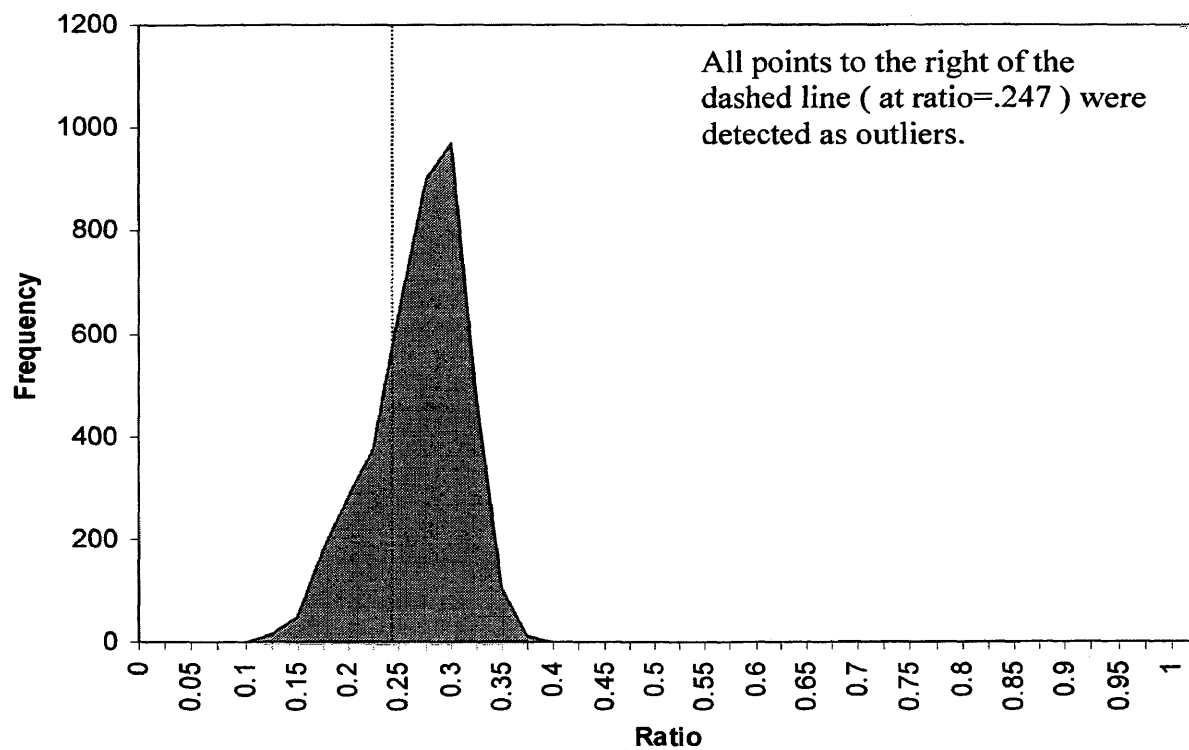
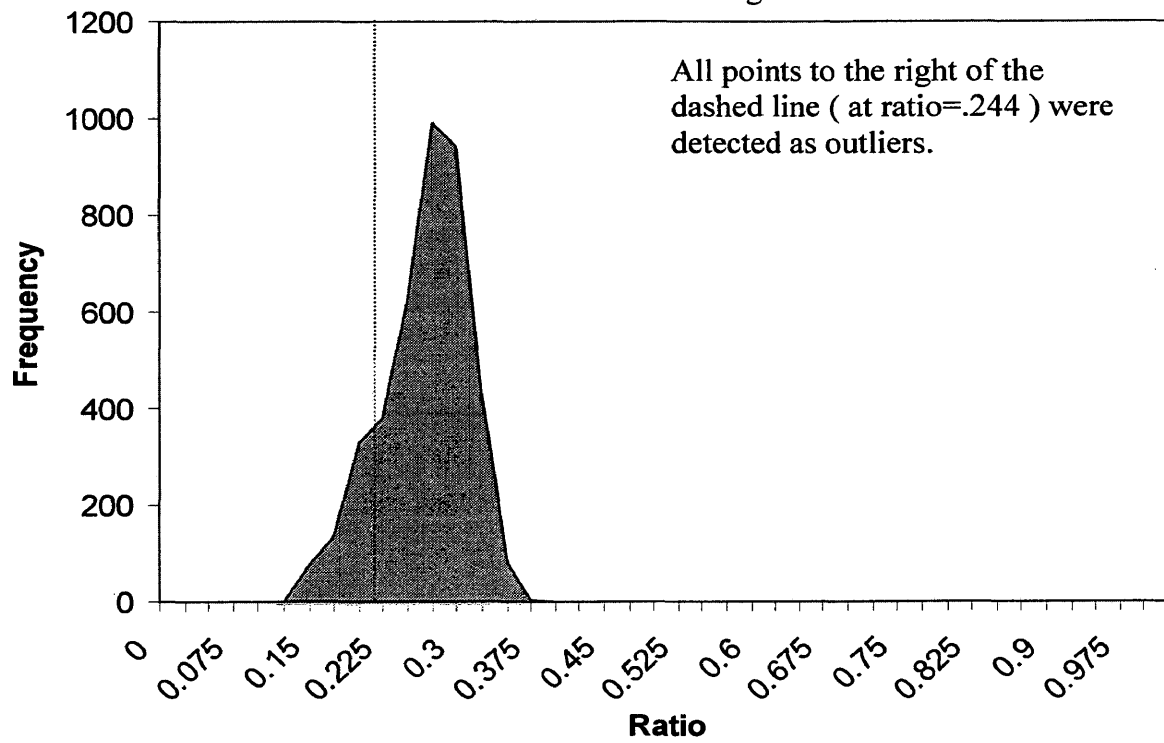


Figure 5.8 Ratio vs Frequency of the Dataset TetRN(380) and RubellaE1(20)  
No Trimming



## 5.2 Varying the Outlier Percentage in the Dataset

For the first experiment, we kept the outlier and non-outlier ratio to the same at 5%. For this experiment, we will alter the outlier and non-outlier ratio to 1, 10, 15, and 20% to see how the OSDUL process behaves with lower and larger amounts of outliers. For this experiment, we will use HCV as the dominant family, non-outlier, and RubellaE1 and TetRN as the outlier families. 10 datasets of 400 sequences were created with different ratios of outliers and non-outlier sequences, and the results were averaged. The following are the results for each of the ratios.

Table 5.7 Varying the composition of RubellaE1 sequences against HCV

	1%	5%	10%	15%	20%
sensitivity	1.000	1.000	1.000	0.978	0.746
specificity	0.984	1.000	0.984	0.995	0.981
precision	1.000	1.000	1.000	0.978	0.746
accuracy	0.984	1.000	0.996	0.992	0.934

Table 5.8 Varying the composition of TetRN sequences against HCV

	1%	5%	10%	15%	20%
sensitivity	1.000	1.000	0.973	0.847	0.866
specificity	0.980	1.000	1.000	1.000	1.000
precision	1.000	1.000	0.973	0.847	0.866
accuracy	0.980	1.000	0.997	0.977	0.973

As the percentage of outliers increase in the dataset, the accuracy tends to fall. This is an expected result since outlier detection is based on the assumption that the number of outliers is small relative to the number of items in the entire dataset. Even then, the OSDUL process is still able to keep reasonable accuracy at 20%. On the other hand, it seems counter intuitive that the accuracy of the 1% ratio would fall lower than

that of the 5% ratio, but when there are too few outliers, there is a risk of assessing non-outliers as outliers. The slight decline in specificity visible in the 1% tests supports this concept.

### 5.3 Outliers from Various Families

So far, all the outliers in the datasets that we ran the OSDUL process against only came from one family. The third experiment will have Family A, B, and C in the dataset, where Family A will have 380 proteins, Family B and C will each have 10 proteins. The goal is to see whether the OSDUL process can detect all proteins from Family B and C. Again, we create 10 datasets of each combination and average the results.

Table 5.9 Results for having outlier sequences from two families

	HCV as Family A	RubellaE1 as Family A	TetRN as Family A
sensitivity	1.000	0.995	1.000
specificity	1.000	1.000	0.997
precision	1.000	0.995	1.000
accuracy	1.000	0.999	0.997

The OSDUL still has very high accuracy even with multiple outlier protein families.

### 5.4 Supervised Outlier Detection

The final experiment will compare the OSDUL process to a similar approach introduced in [35] by conducting a similar experiment. Much like the OSDUL process, the method introduced in [35] creates a Probabilistic Suffix Tree, PST, and uses it to see

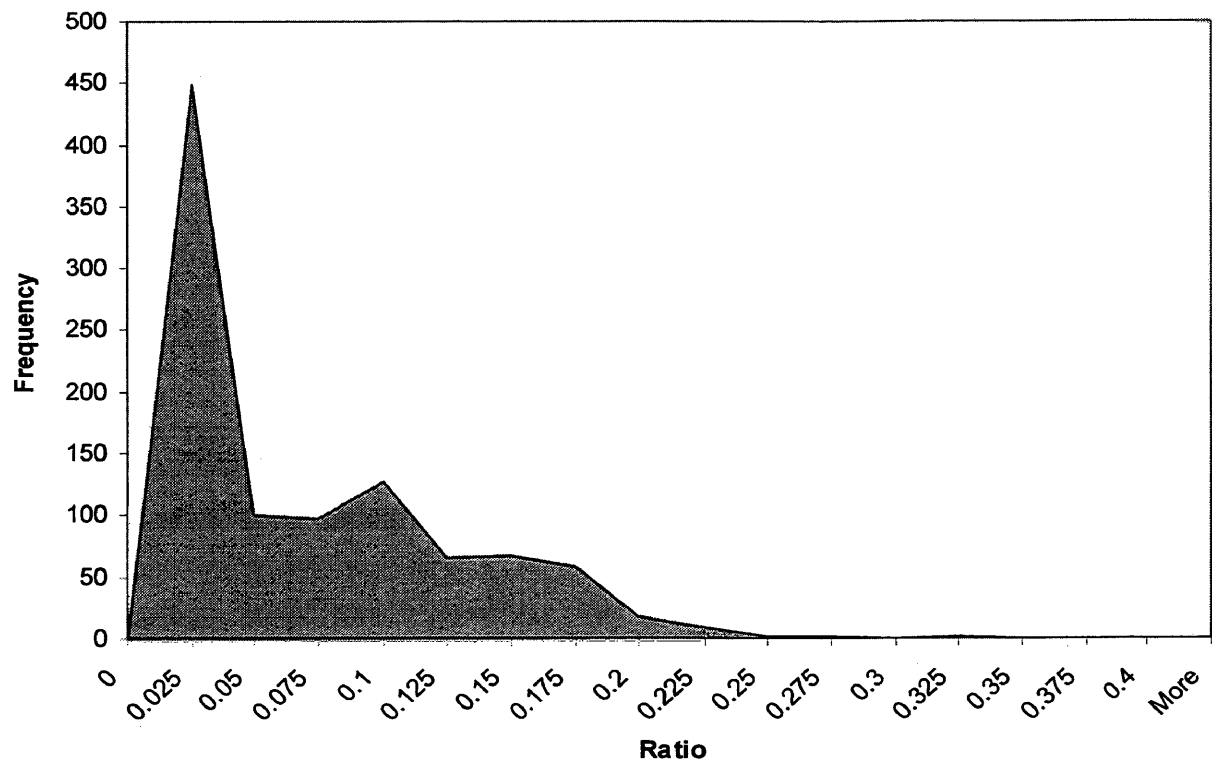
whether or not a data point is an outlier based on how it compares against the tree. We will refer to this method as the PST method for this section. One significant difference between the PST method and the OSDUL process is that the PST method requires the original data to be outlier free. In other words, the tree is first created only by reading in proteins from one family. Then, if a set of data comes from the same family, it should compare well against the tree. If a set of data comes from a different family, it should not compare well against the tree. This is different than the OSDUL process. The OSDUL process does not require the data to come from a single family. In fact, OSDUL's ability to determine outliers within the given dataset is an advantage of the OSDUL process over the PST method.

To fairly compare the OSDUL process against the PST method, we will design the experiment so that the OSDUL process will first create a tree from a single family, Family A. A tree will be created from 1000 protein sequences from Family A. Then, the OSDUL process will go through and assign scores to each pattern as usual. With these scores, we will calculate a compression ratio for each of the 1000 sequences used to create the tree. Figure 5.9 is a histogram of the calculated compression ratios of the 1000 proteins against its own tree when Family A is HCV.

At this point, the original OSDUL process goes through the 1000 sequences to see if there are any outliers using the K-Means algorithm. For this experiment, we know that all of these sequences come from the same family, so we do not need to run the K-Means step. Instead, the process will now evaluate 1000 proteins from Family A, the family that created the tree, and 1000 proteins each from the two other families, Families B and C. The goal is to identify all proteins from Family A as non-outliers and all proteins from Families B and C as outliers.

To do so, the PST method calculates a mean and standard deviation of the ratios calculated from the original sequences. They then read in an unknown sequence and calculate its ratio against the tree. If the sequence's ratio is beyond three deviations away

Figure 5.9 Distribution of the HCV Family



from the mean, then that sequence is detected as an outlier. Table 5.10 is part of their results reported in [35]; only the families relevant to this thesis are reported here. Note that [35] does not specify the exact family names so it can only be speculated that we are both working on similar datasets. Also note that the results in [35] do not show the percentage of sequences from the original family, Family A, that were mistakenly labeled as outliers. The PST can be set to have a certain tree depth which affects the accuracy of the PST method. Their results show that a tree depth of 4 is when their results converge.

Table 5.10 The percentage of the sequences from each family that were 3 standard deviations from the mean, results from [35]

PST	Tree Depth	HCV	RUB	TET
HCV	1	-	89.19	82.43
	2	-	95.68	99.84
	3	-	99.09	100.00
	4	-	99.63	100.00
	5	-	99.63	100.00
	6	-	99.63	100.00
RUB	1	0.00	-	0.00
	2	2.58	-	3.04
	3	98.99	-	99.73
	4	99.65	-	99.95
	5	99.70	-	100.00
	6	99.70	-	100.00

For a fair comparison, the OSDUL process will also determine outlier sequences by using statistical methods instead of the K-Means method. Unfortunately, Figure 5.9 clearly illustrates that the ratios calculated by the OSDUL process are not guaranteed to have a normal distribution. This makes the three standard deviation test inappropriate to use. Instead, we will use a similarly simple and widely used measure to statistically find outliers that uses the inter quartile ratio, IQR. Any protein that is  $1.5 \cdot \text{IQR}$  interval

beyond the third quartile will be considered an outlier [8]. The results are shown in Table 5.11.

Table 5.11 The percentage of the sequences from each family that were  $1.5 \cdot \text{IQR}$  away from the third quartile

	HCV Pattern Tree	RubellaE1 Pattern Tree	TetRN Pattern Tree
HCV	0.3	100	100
RubellaE1	100	2.6	100
TetRN	99.9	100	0

When a tree was built from the HCV family, 0.3% of its own sequences were considered outliers based on the IQR test, and all of the proteins from RubellaE1 and TetRN were outliers. This indicates that the OSDUL process was able to distinguish proteins from its own family and proteins from different families very accurately. The same goes for building a tree from the RubellaE1 and the TetRN families. In both cases, all or almost all proteins from its own family were not considered outliers, and nearly all proteins from different families were considered outliers. These results are comparable, if not better, than those reported in [35].

Finally, we will compare the computational complexities of these two methods. As explored in Chapter 4, the computational complexity of the OSDUL process is  $O(A \cdot m \cdot S) + O(\gamma \cdot m \cdot S^2)$ , but for this experiment, we can omit the trimming process. We can do so because trimming the pattern tree is effective in separating the outliers from the non-outliers within a single dataset. In this experiment, the given datasets are already separated and we are creating a pattern tree based on a single family. Trimming the pattern tree created from a single family does not add value.

Omitting the trimming process reduces the computational complexity of the OSDUL process to  $O(A*m*S)$  from  $O(A*m*S) + O(\gamma*m*S^2)$ . This is comparable to  $O(L*m*S) + O(A^\alpha*L)$  from [35] where  $L$  is the depth of their PST, and  $\alpha$  is a fixed constant based on the pruning parameters of the algorithm in [35]. Experiments in [35] illustrate that  $L=4$  and  $\alpha \leq 4$  are common values. For our protein sequences, the alphabet size  $A$  is at most 26. If we replace the variables with these constant values, the OSDUL process has a complexity of  $O(26*m*S)$  and the computational complexity of the PST method is  $O(4*m*S) + O(26^4*4)$ . The second term of the computational complexity of the PST method can be considered a constant, but it is a large constant value. Moreover, the growth of the second term can become significant if the alphabet size increases or if the  $\alpha$  value increases. On the other hand, the OSDUL process has a high value for the first term compared to the PST method, but it does not have the second term. An increase in alphabet size will not significantly affect the OSDUL process's computational complexity. These similar, if not better, results and computational complexity as the method in [35] shows the effectiveness of the OSDUL process.



## CHAPTER 6

### FUTURE WORK

As the authors in [34] suggest, the integration of data compression techniques for outlier detection and other data mining techniques is certainly an area that deserves more attention. The experimental results of the thesis as well as experiments by [32] and [33] show strong indication that using data compression techniques can be an efficient, effective, and accurate method of outlier detection. This thesis only considers the LZW algorithm but that does not mean that the LZW algorithm is the only algorithm applicable toward outlier detection. There are many more compression schemes available today. Replacing the OSDUL-LZW algorithm with a different compression scheme may provide interesting results. Exploring other methods and algorithms for the different portions of the OSDUL process, such as the trimming process and the scoring function, may also lead to improved results or better computational complexity.

Furthermore, compression is not limited to sequence data. Researching the possibility of using other compression schemes such as JPEG, MPEG, and other multimedia-related technologies may lead to outlier detection of visual data, sound files, and other complex data. Combining well-developed compression technologies with well-developed outlier detection methodologies can increase application of outlier detection toward large and complex data.



## REFERENCES

- [1] V.J. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22 (2), pp. 85-126, 2004.
- [2] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA, pp. 381 – 388, 2001.
- [3] G. Casella and R. Berger. *Statistical Inference Second Edition*. Duxbury, Pacific Grove, CA, pp. 373-374, 2002.
- [4] E. Knorr and R. Ng. A unified notion of outliers: Properties and computation. In: *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining*, Newport Beach, CA, pp. 392-403, 1997.
- [5] E. Knorr and R. Ng. Algorithms for Mining Distance-Based Outliers in Large Datasets. In: *Proc. VLDB Conference*, New York, NY, pp. 392-403, 1998.
- [6] A. Arning, R. Agrawal and P. Raghavan. A linear method for deviation detection in large databases. In: *Proc. 1996 Int. Conf. Data Mining and Knowledge Discovery*, Portland, OR, pp. 164-169, 1996.
- [7] V. Barnett, and T. Lewis. *Outliers in Statistical Data*. John Wiley & Sons, New York, NY, 1978.

- [8] J. Tukey. *Exploratory Data Analysis*. Addison-Wesley, Reading, MA, 1977.
- [9] D. Hoaglin. John W. Tukey and Data Analysis. *Statistical Science*, 13 (3), pp. 311-318, 2003.
- [10] R. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In: *Proc. VLDB*. pp. 144-155, 1994.
- [11] M. Ester, H. Kriegel, J. Sander and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In: *1996 Int. Conf. Data Mining and Knowledge Discovery*, Portland, OR, pp. 226-231, 1996.
- [12] T. Zhang, R. Ramakrishnan and M. Livny. BIRCH: an efficient data clustering method for very large databases. In: *Proc. ACM SIGMOD*, pp 103-114, 1996.
- [13] R. Guttman. A dynamic index structure for spatial searching. In: *Proc. ACM SIGMOD*. pp. 47-57, 1984.
- [14] J. Bentley. Multidimensional binary search trees used for associative searching. In: *CACM*, 18(9). pp. 509-517, 1975.
- [15] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley, 1990.
- [16] S. Ramaswamy, R. Rastogi and K. Shim. Efficient Algorithms for Mining Outliers from Large Data Sets. In: *Proc. ACM SIGMOD Conference on Management and Data*. Dallas, TX. pp. 427-438, 2000.

- [17] S. Sarawagi, R. Agrawal and N. Megiddo. Discovery-driven exploration of OLAP data cubes. In: *Proc. Int. Conf. Very Large Data Bases*. Bombay, India. pp. 544-555, 1998.
- [18] The OLAP Council. *MD-API the OLAP Application Program Interface Version 0.5 Specification*, 1996.
- [19] R. Agrawal, G. Ashish and S. Sarawagi. Modeling multidimensional databases. In: *Proc. 13th Int. Conf. of Data Engineering*, Birmingham, UK, 1997.
- [20] J. Gray, A. Bosworth, A. Layman and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals. In: *Proc. 12th Int. Conf. on Data Engineering*. pp. 152-159, 1996.
- [21] G. Colliat. OLAP, relational, and multidimensional database systems. Technical report, Arbor Software Corporation, Sunnyville, CA, 1995.
- [22] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan and S. Sarawagi. On the computation of multidimensional aggregates. In: *Proc. 22nd Int. Conf. on Very Large Databases*, Bombay, India, pp. 506-521, 1996.
- [23] J. Coughlan and A. Yullie. Manhattan world: orientation and outlier detection by Bayesian inference. *Neural Computation*, 15. pp. 1063-1088, 2003.
- [24] E. Wegman. Visual Data Mining. *Statistics in Medicine*, 22(9). pp. 1383-1397, 2003.

- [25] P. Torr and D. Murray. Outlier Detection of Motion Segmentation. In: *Proc. SPIE 2059*. pp. 432-443, 1993.
- [26] N. Johnson and D. Hogg. Learning the Distribution of Object Trajectories for Event Recognition. *Image and Vision Computing*, 14. pp. 609-615, 1996.
- [27] J. Owens and A. Hunter. Application of the Self-Organizing Map to Trajectory Classification. In: *Proc. Third IEEE Int. Workshop on Visual Surveillance*. pp. 77, 2000.
- [28] C. Mouza and P. Rigaux. Multi-scale Classification of Moving Object Trajectories. In: *Proc. 16th Int. Conf. on Scientific and Statistical Database Management*. pp. 307, 2004.
- [29] J. Lou, Q. Liu, T. Tan and W. Hu. Semantic Interpretation of Object Activities in a Surveillance System. In: *Proc. 16th Int. Conf. on Pattern Recognition*, 3. pp. 30777, 2002.
- [30] S. Hawkins, H. He, G. Williams and R. Baxter. Outlier Detection Using Replicator Neural Networks. *Neural Networks*. pp. 170-180, 2002.
- [31] R. Dannenberg, B. Thom and D. Watson. A machine learning approach to musical style recognition. In: *Proc. Int. Computer Music Conference*. pp. 344-347, 1997.
- [32] R. Cilibrasi and P. Vitanyi. Clustering by Compression. *IEEE Transactions on Information Theory*, 51(4). pp. 1523-1545, 2005.

- [33] D. Benedetto, E. Caglioti and V. Loreto. Language trees and zipping. *Physical Review Letters*, 88:4, 2002.
- [34] A. Kocsor, A. Kertesz-Farkas, L. Kajan and S. Pongor. Application of compression-based distance measures to protein sequence classification: a methodological study. *Bioinformatics*, 22(4). pp. 407-412, 2005.
- [35] P. Sun, S. Chawla and B. Arunasalam. Mining for Outliers in Sequential Databases. In: *Proc. 6th SIAM Int. Conf. on Data Mining*. Bethesda, MD, 2006.
- [36] K. Sayood. *Introduction to Data Compression Second Edition*. New York, NY, Morgan Kaufmann Publishers, 2000.
- [37] D. Huffman. A Method for the Construction of Minimum Redundancy Codes. In: *Proc. IRE*, 40. pp.1098 – 1101, 1951.
- [38] N. Faller. An Adaptive System for Data Compression. In: *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, Piscataway, NJ, pp. 593-597, 1973.
- [39] R. Gallager. *Information Theory and Reliable Communications*. John Wiley and Sons, New York, NY, 1968.
- [40] T. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, pp. 8-19, 1984.

- [41] J. MacQueen. Some methods for classification and analysis of multivariate observations. In: *Proc. 5th Berkeley Symp. Math. Statist, Prob.*, 1. pp. 281-297, 1967.
  
- [42] C. Hoare. Quicksort. *Computer Journal*, 5(1). pp 10-15, 1962.
  
- [43] A. Bateman, L. Coin, R. Durbin, R. Finn, V. Hollich, S. Griffiths-Jones, et. al. The Pfam Protein Families Database. *Nucleic Acids Research*. Database Issue 32:D138-D141, 2004.
  
- [44] W. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Princeton Hall, Englewood Cliffs, NJ, 1992.