A Scalable and Efficient Outlier Detection Strategy for Categorical Data

by

Enrique G. Ortiz

A thesis submitted in partial fulfillment of the requirements
for the Honors in the Major Program
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
and in The Burnett Honors College
at the University of Central Florida
Orlando, Florida

Spring Term 2007

Thesis Chair: Michael Georgiopoulos, Ph.D.

Abstract

Outlier detection has received significant attention in many applications, such as credit card fraud detection and network intrusion detection. Most of the existing research efforts focus on numerical datasets and cannot be directly applied to categorical sets where there is little sense in ordering the data and calculating distances among data points. Furthermore, a number of the current outlier detection methods require quadratic time with respect to the dataset size and usually need multiple scans of the data; these features are undesirable when the datasets are large and scattered over multiple geographically distributed sites. In this paper, we focus and evaluate, experimentally, a few representative current outlier detection approaches (one based on entropy and two based on frequent itemsets) that are geared towards categorical sets. In addition, we introduce a simple, scalable and efficient outlier detection algorithm that has the advantage of discovering outliers in categorical datasets by performing a single scan of the dataset. This newly introduced outlier detection algorithm is compared with the existing, and aforementioned outlier detection strategies. The conclusion from this comparison is that the simple outlier detection algorithm that we introduce is more efficient (faster) than the existing strategies, and as effective (accurate) in discovering outliers.

Dedication

To God,

Who has filled my life with immeasurable blessings.

Acknowledgements

I acknowledge the NSF funding that supports my research and the Intelligent Systems Laboratory.

I thank the Honors College for providing me with funding and introducing me to all of the research opportunities available at UCF.

I also thank my thesis advisor, Dr. Michael Georgiopoulos, and my graduate student mentor, Anna Koufakou, for their support and direction as we completed this project.

I express great gratitude to Dr. Beatriz Roldán Cuenya, who provided me with my first undergraduate research experience and the foundation to continue doing research.

I acknowledge my friends Brian C. Becker and Edric Gonzalez for sticking with me through the difficulties of studying engineering.

I finally thank my family for always encouraging me to ask why.

Table of Contents

List of Figures

List of Tables

List of Terms

k – Target number of outliers

n – The number of data points

m – The number of attribute values

l – A specific attribute (ranging from 1 to *m*)

x – Data point or record

x_i^l – The *i*-th value of the *l*-th attribute of point *x*

$p(x_i^l)$ – Frequency of x_i^l in the dataset

X_l – The *l*-th attribute vector

V – Number of distinct attribute values per attribute

V_l – Number of distinct attribute values per attribute

D – Dataset

I – Itemset

F – Frequent Itemset

E(x) – Entropy

minsup – Minimum support for the identification of frequent patterns

FPS(D, minsup) – Set of all frequent patterns

Greedy – Greedy/Entropy Algorithm

AVF – Attribute-Value Frequency Algorithm

FPOF – Frequent Pattern Outlier Factor

FDOD – Fast Distributed Outlier Detection

Probability of Miss – the number of true outliers that were not detected (missed) over the total number of actual outliers in the dataset

Probability of False Alarm – the number of points detected as outliers that are truly non-outliers over the total number of non-outliers

CHAPTER 1 Introduction

Mining for outliers in data is an important research field with many applications in credit card fraud detection, discovery of criminal activities in electronic commerce, and network intrusion detection. Outlier detection approaches focus on discovering patterns that occur infrequently in the data, as opposed to traditional data mining techniques that attempt to find patterns that occur frequently in the data. One of the most widely accepted definitions of an outlier pattern is provided by Hawkins [1]: “An outlier is an observation that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism.”

Outliers are frequently treated as noise that needs to be removed from a dataset in order for a specific model or algorithm to succeed (e.g. points not belonging in clusters in a clustering algorithm). However, lately it has been acknowledged that outlier detection techniques can lead to the discovery of important information in the data, “one person’s noise is another person’s signal” [2]. On the other hand, any outlier detection strategy can also be used for the cleaning of the data before any traditional data-mining algorithm is applied on the data. Examples of data where the discovery of outliers is useful are irregular credit card transactions, indicating potential credit card fraud [3], or patients who exhibit abnormal symptoms due to their suffering from a specific disease or ailment [4].

Most of the research efforts in outlier detection strategies have focused on datasets that are comprised of numerical attributes or ordinal attributes that can be directly mapped into numerical values. Quite often, when we have data with categorical attributes, it is assumed that the categorical attributes could be easily mapped into numerical values. However, there are cases of categorical attributes where this mapping to numerical attributes is not a straightforward process, and the results greatly depend on the mapping that is used (e.g., the mapping of a marital status attribute (Married or Single) or a person's profession (engineer, financial analyst, etc.) to a numerical attribute).

Recently there has been some focus on data with categorical or mixed attributes (e.g. He et al.[5], [6],[7], and Otey et al. [8]). Yet, these efforts have not been contrasted to each other and they have been evaluated using different datasets. In this thesis, we explore some of these methods and evaluate them on the same datasets with regard to their efficiency (speed), scalability, and effectiveness (accuracy) in detecting outliers in categorical data.

Another issue that has only recently become a focus in the literature is related to the large and distributed nature of the datasets available today. With the explosion of technology, the size of data for a particular application has grown and will continue to grow. In addition, most of the data is distributed among different sites belonging to the same or different organizations. Transferring the data to a central location and then detecting outliers is usually impractical because of the size of the data and the expense of constantly moving it, without accounting for data ownership and control issues.

Hence, successful outlier detection strategies must perform well and be scalable as the size and dimensionality of the dataset grows. Furthermore, in order to deal with the distributed nature of the data, the communication overhead and synchronization between the different sites in which the data resides should be minimized; consequently, the passes over the data should be minimal. In this paper, we introduce a simple outlier detection strategy for categorical datasets called *Attribute-Value Frequency* (AVF), which performs and scales well, and would work efficiently for datasets that are geographically distributed over a number of sites. We compare this simple algorithm with He's [5-7] and Otey's [8] algorithms from the perspective of accuracy (in terms of finding the outliers) and speed (of finding the outliers in the data).

CHAPTER 2 Background

2.1 Literature Review

2.1.1 Statistical-Model-Based

The earliest approaches used to detect outliers were statistical-model-based, which assumed that a parametric model described the distribution of the data (e.g., normal distribution) and the data was mostly single-dimensional or univariate [9, 10]. The drawbacks of these approaches include the difficulty of finding a right model for each dataset and associated application, as well as their efficiency decreases as the dimensions of the dataset increases [8, 10]. Another issue with high dimensional datasets is that the dataset becomes less dense, which makes the convex hull harder to determine ("Curse of Dimensionality") [11]. There are some methods, like the Principal Component Analysis, that can help alleviate this problem. Another idea to handle higher dimensional datasets is to organize the data points in layers based on the idea that shallow layers tend to contain outliers more often than the deep layers (e.g. [12, 13]); in practice, however, these ideas are impractical for more than two dimensions.

2.1.2 Distance-Based

Distance-based approaches do not make any assumptions about the distribution of the data because they essentially compute the distances among points. However, this leads to high computational complexity (e.g. nearest neighbor approach that has quadratic complexity with respect to the dataset size). This renders them impractical for large datasets.

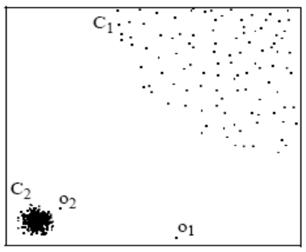

Figure 1 Distance-Based Method

Distance-based methods will have a problem finding local outlier o2 [14]

There have been improvements of the original distance-based algorithms, such as Knorr's et al. [2], where an outlier is defined as an object *O* in a dataset *T* that has at least a fraction *p* of the objects in *T* lying further than distance *D* from it. The complexity of their proposed approach, however, is still exponential on the number of nearest neighbors. Finally, Bay and Schwabacher [15] randomize the data for efficient pruning of the search space, so that even though the worst-case algorithmic complexity is quadratic, the algorithm in practice becomes closer to linear complexity.

2.1.3 Clustering

Clustering techniques can be used with the idea that the points that do not belong in the formed clusters are designated as outliers. Shekhar et al. [16] use a graph to reflect the connections between each point. However, this technique is applicable only when a graph of the data can be constructed. Clustering-based methods are focused on optimizing clustering measures of goodness, and not on finding the outliers in the data [2].

2.1.4 Density Based

Density-based methods focus on estimating the density distribution of the input space and identifying outliers as those lying in regions of low density. Breunig et al. [14] assign a *degree of outlier-ness* to each data point: they calculate a local outlier factor (*LOF*) for each point based on the ratios of the local density of the area around the point and the local densities of its neighbors. The size of the local neighborhood of a point is determined by the area containing a user-given minimum number of points (*MinPts*). Papadimitriou et al. in [17] present a similar technique called *LOCI* (Local Correlation Integral) which tackles the issue of choosing values for *MinPts* in the previous technique (*LOF*) by using statistical values based on the data itself. All density-based techniques have the advantage that they can detect outliers that are missed by techniques with a single, global criterion, as can be seen in Figure 1. However, in high-dimensional spaces the data is almost always sparse, which leads to problems with density-based methods [18].

2.1.5 Others

There have been other research efforts, such as [19], where the Support Vector Data Description (SVDD) is proposed. This method obtains a spherically shaped boundary around a dataset that can be made flexible by using a variety of kernel functions. In [20], Replicator Neural Networks (RNNs) are used to detect outliers. More recently, [21] looked into using only a few fixed reference points to rank the data points, resulting in algorithmic complexity of $O(R \cdot n \cdot \log n)$, where R is the number of reference points and n is the size of the dataset.

All the aforementioned methods are geared towards numerical data and thus are more applicable to numerical datasets or data where ordinal data can be easily transformed to suitable numerical values [11]. In the case of categorical datasets, there is not much sense in ordering the data and then mapping them to numerical values (e.g., distance between two values such as TCP Protocol and UDP Protocol [8]). Consequently, methods such as distance-based, density-based, etc. are deemed unsuitable. Moreover, a number of these methods are quadratic in complexity with respect to the data size, *n*, which would be unacceptable for very large datasets. In addition, if a distributed setting is assumed, employing algorithms that depend on pair-wise distance computations is infeasible, as the different sites would have to either exchange all of their local data points in order to calculate the distances or replicate all data points in every local database.

2.2 Related Prior Research

In this research, we implemented and experimented with three current outlier detection approaches directed towards categorical data. The first is proposed by He et al. [6] and is based on the idea of Entropy. The second technique by Otey et al. [8] focuses on datasets with mixed attributes (both categorical and numerical). The third technique is discussed in [7], another paper by He et al. The methods presented in [7] by He and in [8] by Otey are based on scoring each data point using the concept of frequent itemsets (see [22]). Wei et al. [18] also deal with outliers in categorical datasets and use frequent itemsets. In [18], the authors use hyperedges, which simply store frequent itemsets along with the data points that contain these frequent itemsets. Because their method is based on frequent itemsets and built on a premise quite similar to that in [7], it was not considered in our experiments. Finally, Xu et al. [23] use mutual reinforcement to discover outliers in a mixed attribute space. However, their method focuses on a slightly different outlier detection problem. Instead of discovering local outliers as noise, they identify local outliers in the center, where they are similar to some clusters on one hand and unique on the other.

CHAPTER 3 Algorithms

In this section, we describe the algorithms for outlier detection presented in [6], [7], and [8]. This chapter first explains the entropy-based algorithms presented in [6]. Subsequently, it discusses a more efficient algorithm derived from the entropy-based method. Then it discusses two scoring algorithms based on frequent itemset mining that are presented in [7] and [8]. In addition, we present a simple example to exhibit how each algorithm works to discover outliers.

3.1 Entropy-Based Algorithms

The Greedy algorithm takes as an input the desired number of outliers (k). All points in the set are initially non-outliers. We formulate the set of outliers by conducting k scans over the dataset to iteratively determine the top k outliers. During each scan, we remove every non-outlier individually from the dataset and recalculate the total entropy of the system. The data point that has the maximum impact on the total entropy is the point that lowers the entropy the most when removed.

3.1.1 Entropy

The entropy, $E(X_l)$ of an attribute X_l of a dataset, that describes the degree of "disorder" that this attribute contributes to the dataset, is defined as follows:

$$E(X_l) = -\sum_{i=1}^{V_l} p(x_i^l) \log_2(p(x_i^l))$$

where, in the above equation, V_l denotes the number of attribute values of attribute X_l, and $p(x_i^l)$ denotes the probability with which value x_i^l of attribute X_l is assumed. The entropy of a random variable (in our case the attribute X) is attributed to Shannon [24]. Shannon also defined the entropy of a multi-dimensional random variable (or the multiple attributes corresponding to a dataset). In the case where the attributes are independent, the entropy (disorder) of the multiple attributes of a dataset is equal to the sum of the entropies of each one of the attributes and is defined as follows:

$$E(X_1) + E(X_2) + \cdots + E(X_m)$$

There is a major overhead due to calculating the entropy and frequencies of each attribute value. We solve this time complexity using Google's open source hash table, which is highly optimized [25]. One hash table is necessary for each individual attribute, resulting in *m* hash tables using attribute values as hash keys and the frequency as the referred value.

3.1.2 Greedy Algorithm

He et al. use the above entropy definition to detect the outliers in the data. In particular, a Local-Search heuristic based Algorithm (LSA) is introduced in [5], and a Greedy Algorithm is introduced in [6], both relying on the entropy idea. Because the Greedy algorithm is an improvement on the LSA and is consequently superior, we will only focus on the Greedy algorithm. The Greedy algorithm takes as an input the desired number of outliers (k). All points in the set are initially designated as non-outliers. To formulate the set of outliers we conduct k scans over the dataset to determine the top k outliers. During each scan, we remove every non-outlier individually from the dataset and recalculate the total entropy of the system. The non-outlier data point that results in the maximum decrease for the entropy of the dataset is the outlier data-point that the algorithm removes.

The complexity of the Greedy algorithm is $O(n \cdot k \cdot m \cdot V)$, where n designates the size of the dataset, k is the number of outlier points, m is the number of attributes in the dataset, and V designates the number of distinct attribute values, per attribute. If the number of attribute values per attribute V is a small number, the complexity of Greedy becomes equal to $O(n \cdot k \cdot m)$. The pseudocode of the Greedy algorithm is provided in Figure 2.

```
Algorithm: Greedy
Input:      Dataset - D
            Target number of outliers - k
Output:     k detected outliers
---------------------------------------------------------------------
label all data points $x_1, x_2, \dots, x_n$ as non-outliers
calculate initial frequency of each attribute value and update hash
      table
calculate initial entropy
counter = 0
while ( counter != k ) do
      counter++
      while ( not end of database ) do
            read next record x labeled non-outlier
            label x as outlier
            calculate decrease in entropy
            if ( maximal decrease achieved by record o )
                  update hash tables using o
                  add x to set of outliers
            end if
      end while
end while
```

Figure 2 Pseudocdoe for the Greedy Algorithm

3.2 Attribute-Value Frequency (AVF) Algorithm

Although the algorithms discussed in the previous section scale linearly with respect to the number of data points, *n*, they still need *k* scans over the dataset to find *k* outliers, which is a disadvantage for very large datasets and/or datasets that are distributed among different sites. It is intuitive that outliers are those points which are infrequent in the dataset. Under the assumption of independent attributes, we could claim that the *infrequent-ness* of a data-point in the dataset is strongly correlated with the *infrequent-ness* of the value of every coordinate of this data-point. The *infrequent-ness* of the value of every coordinate of a data-point is calculated by computing how frequently this value is assumed by the corresponding attribute.

More specifically, for each attribute X_l, we compute the probability, $p(x_i^l)$, of each value x_i^l that attribute X_l can assume (note that this probability is a frequency-based definition of probability and this is why the name of this algorithm (AVF) was chosen as such). We then designate a score for each point x in the dataset as the average of all these probabilities, which we define below:

$$AVF\ Score(x = x_1, x_2, \dots, x_m) = \frac{1}{m}\sum_{l=1}^{m}\sum_{i}^{V^l} p\left(x_i^l \middle| x_i^l \in x\right)$$

We named this outlier detection algorithm *Attribute Value Frequency* (AVF) algorithm. With AVF, we first calculate the probability of each attribute value, then we calculate the average of these probabilities of each attribute value corresponding to the point of interest. Once the score of all the points is calculated, we designate the k points with the smallest score values as the k outliers. The complexity of AVF is $O(n \cdot m)$. As a reminder Greedy's complexity is $O(n \cdot k \cdot m)$. Using AVF, the outliers are identified after only one scan of the dataset, instead of k scans that are needed by Greedy in order to identify k outliers. The algorithm's pseudocode is presented in Figure 3.

```
Algorithm: AVF
Input:      Dataset - $D$
            Target number of outliers - $k$
Output:     $k$ detected outliers
----------------------------------------------------------------

label all data points $x_1, x_2, \dots, x_n$ as non-outliers
foreach point $x$
      foreach attribute $m$
            update frequency table
      end
end
foreach point $x$
      calculate average frequency for each attribute
end

identify top $k$ outliers
```

Figure 3 Pseudocode for the Attribute-Value Frequency (AVF) Algorithm

3.3 Frequent Itemset Mining Based Algorithms

In this section, we describe two algorithms that use the concept of frequent itemset mining in order to create an outlier score for each data point in the dataset. Using this score, they identify the top k outliers. These algorithms differ from the outlier detection algorithms described in sections 3.1 and 3.2, because they take into account the association of attributes with each other. *Frequent Itemset Mining* or *Frequent Pattern Mining* is part of *Association Rule Mining*, which has received considerable attention since the seminal paper on this subject by Agrawal and Srikant [22]. These are described below:

3.3.1 Frequent Itemset Mining

Given a dataset D and a set of r literals $S=\{i_1, i_2, \dots, i_r\}$ that are found in D, we can define an itemset I as a non-empty subset of S. For example, items in a supermarket could be “bread”, “milk”, etc; then a possible itemset I could be {“bread”, “milk”}.

Given a user-defined threshold called minimum support, *minsup*, a frequent itemset *F* is one that appears in the dataset at least *minsup* times. Frequent itemset mining in a dataset *D* with a threshold *minsup* results in a set of frequent itemsets or patterns, denoted as *FPS*(*D*, *minsup*). The support of an itemset *I*, designated as *support*(*I*), is the percentage of data points in *D* that contain itemset *I*.

3.3.2 Find Frequent Pattern Outlier Factor (FindFPOF)

He et al. in [7] observe that because frequent itemsets are "common patterns" that are found in many of the points of the dataset, outlier detection algorithms can be discovered by relying upon the concept of frequent itemsets. Hence, they define a Frequent Pattern Outlier Factor (FPOF) for every data point based on the support of the frequent itemsets contained in the data point. In addition, they use a *contradictness* score in order to describe the reasons why the identified outliers are abnormal, based on the itemsets that are not contained in the detected outlier data. The *contradictness* score is used to better explain the outliers and not to detect the outliers, so it is omitted from our discussion. The FPOF outlier score is calculated as follows:

$$FPOF\ Score(x) = \frac{\sum_{F \subseteq x, F \in FPS(D,minsup)} support(F)}{\|FPS(D, minsup)\|}$$

The FPOF score of a point is the summation of the support of all of the frequent subsets *F* of the data point over the total size of the *FPS (D, minsup)*, which is the number of all frequent sets in the dataset *D*. The idea is that data points with small FPOF values (small number of frequent subsets) are likely to be outliers. The outlier algorithm, FindFPOF, runs the *Apriori* algorithm first, to identify all of the frequent patterns in dataset *D* with a threshold *minsup*, designated by *FPS*(*D*, *minsup*). Then, the outlier score for each data point *x* is calculated. The algorithm uses the FPOF values to identify the top *k* outliers. The pseudocode for the FindFPOF algorithm is given in Figure 4.

```
Algorithm: FindFPOF
Input:      Dataset - D
            Minimum support - minsup
            Target number of outliers - k
Output:     k detected outliers
-----------------------------------------------------------------------
FPS(D, minsup) = Mine for frequent item sets in D
foreach data point x in D
      foreach frequent pattern F in FPS(D, minsup)
            if x contains F
                  outlier_score(x) +=  support(F) / sizeof(FPS(D,
                        minsup))
            endif
      end
end
identify top k outliers
```

Figure 4 Pseudocode for the FindFPOF Algorithm [7]

3.3.3 Fast Distributed Outlier Detection (FDOD)

The method by Otey et al. in [8] is also based on the concept of frequent itemsets. The authors detect outliers by assigning to each point an anomaly score inversely proportional to the support of its infrequent itemsets. The way to handle the continuous attributes is to maintain a covariance matrix for each itemset. We have not considered continuous attributes since our focus is on categorical data, so we omit this part from our discussion. Specifically, in [8] the authors calculate an anomaly score for each data point *x*, which is given below:

$$Otey's\ Score(x) = \sum_{I \subseteq x} \frac{1}{|I|} \quad | \quad support(I) \leq minsup$$

which can be explained as follows: for each point *x*, we find all subsets *I* of *x* which are infrequent, i.e. their support is less than *minsup*. Then, the anomaly score of *x*, will be equal to the sum of the inverse of the length of the infrequent itemsets. If a point has very few frequent patterns, its outlier factor will be high. Thus, the outliers are those *k* points with the maximum outlier score in the previous equation. This algorithm is very similar to FindFPOF in that it first mines the dataset *D* for the frequent itemsets and then an outlier score is calculated for each point in dataset *D*.

An example for the algorithm in Otey et al. [8] is given in Figure 5. Given the itemset lattice in the figure, P1 has a score of 0 because all its subsets are frequent, while P2 contains 2 infrequent subsets, {abd}, {bd}, so its anomaly score equals the sum of the inverse of their length, i.e. (1/3+1/2).

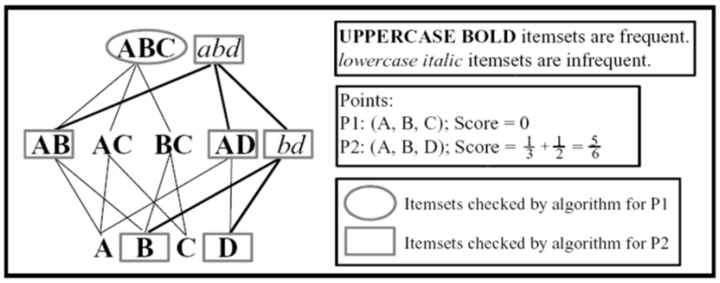
ABC
abd
AB
AC
BC
AD
bd
A
B
C
D
UPPERCASE BOLD itemsets are frequent.
lowercase italic itemsets are infrequent.
Points:
P1: (A, B, C); Score = 0
P2: (A, B, D); Score = $\frac{1}{3}+\frac{1}{2}=\frac{5}{6}$
Itemsets checked by algorithm for P1
Itemsets checked by algorithm for P2

Figure 5 Example for Otey's Algorithm [8]

The pseudocode for this algorithm is given in Figure 6. The authors state that the execution time is linear to the dataset size, *n*, but exponential to the number of categorical attributes. They also try to limit the lattice levels (i.e. the maximum levels in the tree that contains the frequent itemsets) that they search as part of their algorithm, in order to achieve better execution times. However, as their results in [8] show, this affects their detection rates in a negative way.

```
Algorithm: FDOD
Input:      Dataset - D
            Minimum support - minsup
            Target number of outliers - k
Output:     k detected outliers
-----------------------------------------------------------
FPS(D, minsup) = Mine for frequent item sets in D

foreach point x in D
      foreach itemset I in x
            if FPS(D, minsup) does not contain I
                  outlierScore(x) +=  1/length(I)
            endif
      end
end
identify top k outliers
```

Figure 6 Pseudocode for the FDOD Algorithm [8]

The most straightforward method of carrying out this algorithm is to find all of the combinations each point contains and searching for them in the set of all frequent itemsets. However, this proves to be very time expensive. Based on probability theory, we derive a quicker method. We can achieve the same outcome a lot faster if we compute the total combinations of length *j* we can possibly have for the *m* attributes and then find how many are infrequent if we decrease that number by the number of frequent subsets (itemsets) for that point.

First, we calculate the number of combinations for each possible outcome using the combination function. The combination function describes the number of ways of picking *k* outcomes from *n* possibilities:

$${}_nC_k = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

For example, $\binom{3}{2} = 3$ combinations of two elements from the set {1, 2, 3} → {1, 2}, {1, 3} and {2, 3}. We can then check if the current point contains any of the frequent itemsets, which is a faster operation. If a frequent itemset is found, the count of the number of subsets or itemsets for that size is decremented. In the end, we are left with the number of infrequent itemsets for each size, which is all the information we need to calculate the outlier score. The modified pseudocode is presented below in Figure 7.

```
Algorithm: Modified FDOD
Input:      Dataset - D
            Minimum support - minsup
            Target number of outliers - k
Output:     k detected outliers
------------------------------------------------------------------------
FPS(D, minsup) = Mine for frequent item sets in D
foreach transaction x in D
      calculate all combinations and store in vector v
      foreach frequent pattern F in FPS(D, minsup)
            if x contains F
                  update v - decrement number of combinations for the
                  size of F
            endif
      end

      foreach combination c in vector v
            outlier_score(x) +=  c * 1/position in vector
      end
end
identify top k outliers
```

Figure 7 Modified FDOD Pseudocode

3.4 Examples

This section uses the example originally presented in [7] to demonstrate how each algorithm works. The dataset shown in Table 1 consists of ten customers with the attributes *Age-range*, *Car*, and *Salary-level* and two attribute values each. The values distinguished with bold text (Customers 5, 6, 8, and 10) are the customers considered outliers, since they contain attribute-values occurring less frequently.

Customers	Age-Range	Car	Salary-Level
1	Middle	Sedan	Low
2	Middle	Sedan	High
3	Young	Sedan	High
4	Middle	Sedan	Low
5	**Young**	**Sports**	**High**
6	**Young**	**Sports**	**Low**
7	Middle	Sedan	High
8	**Young**	**Sports**	**Low**
9	Middle	Sedan	High
10	**Young**	**Sports**	**Low**

Table 1 Customer Information

3.4.1 Greedy

As stated previously, the Greedy algorithm uses entropy calculations to determine which points lower the entropy the most. Assume the target number of outliers is four. Table 2 illustrates the effect of removing each individual point during four passes over the dataset. In this example, during each pass over the dataset it is obvious which points minimize the entropy the most, thus revealing the outliers.

Initial Total Entropy:	***2.97***	***2.90***	***2.77***	***2.44***
	Entropy Impact			
Remove Customer	***First Pass***	***Second Pass***	***Third Pass***	***Fourth Pass***
1	2.97	2.95	2.83	2.49
2	2.97	2.91	2.83	2.57
3	2.97	2.86	2.71	2.30
4	2.97	2.95	2.83	2.49
5	**2.90**	-	-	-
6	2.90	**2.77**	-	-
7	2.97	2.91	2.83	2.57
8	2.90	2.77	**2.44**	-
9	2.97	2.91	2.83	2.57
10	2.90	2.77	2.44	**1.57**
Minimum:	**2.90**	**2.77**	**2.44**	**1.57**

Table 2 Entropy Calculations for Example in Table 1

3.4.2 Attribute-Value Frequency Based (AVF) Algorithm

The AFV algorithm first calculates the frequency of each attribute value, which is shown for this example in Table 3.

Attribute Values	Frequency
Middle	0.5
Young	0.5
Sedan	0.6
Sports	0.4
Low	0.5
High	0.5

Table 3 Frequency of Attribute Value for Example in Table 1

As seen in Table 4, we calculate the average frequency of attribute values contained by each individual customer.

Customers	Average Frequency
1	0.53
2	0.53
3	0.53
4	0.53
5	**0.47**
6	**0.47**
7	0.53
8	**0.47**
9	0.53
10	**0.47**

Table 4 Average Frequency of Attribute Values for Each Customer for Example in Table 1

As a result, it is obvious that customers 5, 6, 8 and 10 are outliers because of their low average frequency value.

3.4.3 Find Frequent Pattern Outlier Factor (FindFPOF)

The first step in the FindFPOF algorithm is the *Apriori* algorithm, which is the frequent itemset mining phase. This stage takes a minimum support threshold as an input. For the purpose of this example, the minimum support is 0.5. The execution of the *Apriori* algorithm results in the frequent patterns shown in Table 5.

Pattern	Support
Middle	0.5
Young	0.5
Sedan	0.6
Low	0.5
High	0.5
Middle, Sedan	0.5

Table 5 Frequent Itemsets for Example in Table 1

With the frequent itemsets detected, we are able to calculate the *FPOF* score for each customer. We calculate the scores shown in Table 6 by summing the support of each pattern the customer contains and dividing by the total number of frequent itemsets.

Customers	FPOF
1	0.35
2	0.35
3	0.27
4	0.35
5	**0.17**
6	**0.17**
7	0.35
8	**0.17**
9	0.35
10	**0.17**

Table 6 FPOF Calculations for Example in Table 1

Because of the FPOF score, it is obvious customers 5, 6, 8 and 10 are outliers because of their low score.

3.4.4 Fast Distributed Outlier Detection (FDOD)

As in the FindFPOF algorithm, the first step in the FDOD algorithm is the frequent itemset mining phase and the minimum support is 0.5. The execution of the *Apriori* algorithm results in the frequent patterns shown in Table 5

To calculate the outlier score using Otey's method, we sum one over the length of the infrequent itemsets each individual customer contains. For this example, the scoring results in Table 7.

Customers	Outlier Score
1	1.33
2	1.33
3	1.83
4	1.33
5	**2.83**
6	**2.83**
7	1.33
8	**2.83**
9	1.33
10	**2.83**

Table 7 Outlier Score Using Otey's Method for Example in Table 1

The outlier score easily identifies points 5, 6, 8 and 10 as the outliers because of their high score.

CHAPTER 4 Experiments and Results

4.1 Experimental Design

4.1.1 Hardware

We conduct all experiments on a workstation with a Pentium 4 2.6 GHz processor and 1.5 GB of RAM as the benchmark. We also ran the experiments on a 2.0 GHz Pentium Core Duo laptop with similar results.

4.1.2 Data Processing

We consider two issues with the datasets. First, most of our datasets have values that can be considered categorical; however, some have numeric (continuous) attributes. We discretize this data using the equal-frequency method in [26], and then treat these attributes as categorical. We handle missing values by either eliminating the missing values, or by considering the unknowns as an additional attribute value. In addition, for the frequent itemset-based algorithms, the minimum threshold is 0.1.

4.2 Datasets

We experimented with five real datasets from the UCI Machine Learning repository [27] and a set of artificially-generated datasets based on the work described in [5, 6] using Cristofor's software [28]. Using simulated data has the advantage that we can experiment with as many data points as needed. In addition, we can experiment with different values for the dimensionality of the data. We discuss these datasets in the following sections.

4.2.1 Wisconsin Breast Cancer

The Wisconsin breast cancer dataset contains 699 records with 9 attributes. For the purpose of this experiment, all attributes are considered categorical. There are 458 *benign* records of that dataset and 241 *malignant* records, which are 65.5% and 34.5% of that dataset respectively. All unknown values are removed from the dataset. Following the method used by [20], we only kept every sixth malignant record in order to create a more imbalanced distribution, resulting in 39 *malignant* outliers (8%) and 444 *benign* non-outliers (92%).

4.2.2 Lymphography

The purpose of the lymphography dataset is to detect abnormal lymph nodes. This dataset contains 148 instances and 19 attributes including the class label. Classes 2 (metastases) and 3 (malign lymph) make up about 96% of the dataset, while 1 (normal find) and 4 (fibrosis) account for approximately 4%. This dataset does not contain unknown values.

4.2.3 Post-operative Patients

The task of the post-operative dataset is to determine where to place patients after surgery, i.e. Intensive Care Unit, home, or general hospital floor. This dataset contains 90 instances and 9 attributes including the class label. There are three classes with 2, 24, and 64 instances respectively. We handle every attribute as categorical. There are three unknown values, which we handle as an extra attribute value because the dataset is so small.

4.2.4 Page Blocks

The purpose of the page blocks dataset is to separate text from graphic areas. This dataset contains 5,473 instances with 10 attributes. We reduce the size of the dataset by half to make it more imbalanced. To do this every other outlier is removed from the dataset. There are five classes: text, horizontal line, vertical line, graphic, and picture. Text makes up about 90% of dataset, while the rest make up 10%. We discretize its four continuous attributes using equal-frequency and a max value of 20. The rest of the attributes are handled as categorical values.

4.2.5 Simulated

The experiments conducted with the simulated data were used to display the efficiency (speed) and associated scalability of the algorithms that we are evaluating; not their detection rates/capabilities (effectiveness). The idea behind these experiments is to see how the performance of each algorithm under investigation changes as specific data parameters change (e.g., the size of the dataset, *n*). Another data parameter that we experimented with is *m*, which is the dimensionality of the dataset. For example, Greedy calculates the Entropy for each point and each attribute. In addition, the Frequent Itemset based algorithms will need to create and search through many more frequent itemsets as the dimensionality increases. Finally, it is worth mentioning that the Greedy algorithm is also dependent on the input number of outliers, *k*, while the performance of the other three algorithms should not change for different values of *k*.

We create simulated datasets based on the experiments described in [5, 6]. For all experiments, we used a random seed generator of 5. For the first three datasets, the input-*k* is a constant value of 30. The first dataset has 10 attributes, 10 distinct attribute values per attribute, and varies at 1k, 10k, 30k, 50k, 100k, 200k, 300k, 400k, 500k, 600k, 700k, and 800k data points. The second dataset contains 100,000 points, 10 distinct attribute values per attribute, and the number of attributes varies at 2, 5, 10, 20, and 30. The third dataset has 100,000 data points, 10 attributes, and the number of distinct attribute values varies at 5, 10, 20, 30, and 40. The fourth and final created dataset has 100,000 data points, 10 attributes, 10 attribute values per attribute, and input target *k* varies at 1, 10, 30, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1,000.

4.3 Results

The following section presents all of the results for the real datasets (Wisconsin Breast Cancer, Lymphography, Post-operative Patients, and Page Blocks) and simulated datasets (Varying Data Size, Varying Number of Attributes, Varying Number Attribute Values, and Varying Input-*k*).

4.3.1 Wisconsin Breast Cancer

Because of its imbalanced distribution, the Wisconsin breast cancer dataset is a good dataset to analyze the rate at which the algorithms discover the outliers. Table 8 shows the results (the number of actual outliers found by each algorithm) for different input-*k* values. The percentage of total outliers is the number of outliers detected out of the total number of outliers.

	Greedy		AVF		FindFPOF		FDOD	
k	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**
4	4	10.26	4	10.26	3	7.69	3	7.69
8	8	20.51	7	17.95	7	17.95	7	17.95
16	15	38.46	14	35.90	14	35.90	15	38.46
24	22	56.41	21	53.85	21	53.85	21	53.85
32	29	74.36	28	71.79	27	69.23	28	71.79
40	33	84.62	32	82.05	31	79.49	33	84.62
48	37	94.87	36	92.31	35	89.74	37	94.87
56	39	100.00	39	100.00	39	100.00	39	100.00

Table 8 Results (Outliers Detected) for All Outlier Detection Algorithms using the Wisconsin Breast Cancer

The algorithms differ from each other very slightly and are approximately equally effective at discovering the outliers. This fact is also demonstrated in Figure 8.

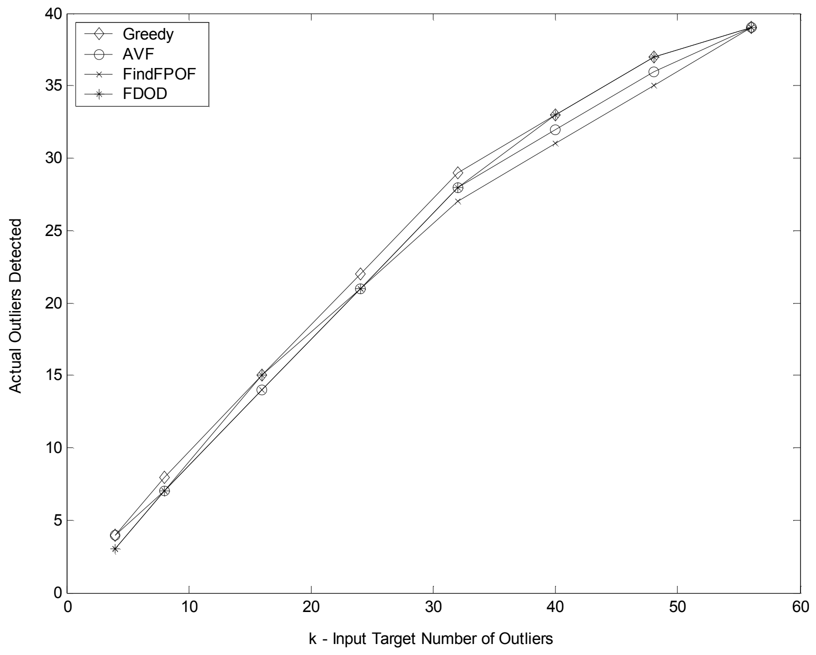

Figure 8 Input Target Outliers, *k*, vs. Actual Outliers Found for the Wisconsin Breast Cancer Dataset

All algorithms perform equivalently.

Table 9 displays the results presented in [6], which are on par with our results except for two values. Our version of the Greedy algorithm detects one more actual outlier with a *k* input of 48, while our FindFPOF implementation detects one fewer with *k* input of 32. Since they only differ by one outlier, these different results are given no importance. The differences can be justified by the calculation accuracy of each implementation.

k	**Greedy**	**FindFPOF**
4	4 (10.26%)	3 (7.69%)
8	7 (17.95%)	7 (17.95%)
16	15 (38.46%)	14 (35.90%)
24	22 (56.41%)	21 (53.85%)
32	27 (69.23%)	28 (71.79%)
40	33 (84.62%)	31 (79.49%)
48	36 (92.31%)	35 (89.74%)

k	Greedy	FindFPOF
56	39 (100%)	39 (100%)

Table 9 Results for Greedy and FindFPOF as Presented in the Original Papers

The literature's results match those in Table 8.

Table 10 presents the *Probabilities of False Alarm and Miss*. The *Probability of False Alarm* is the number of points detected as outliers that are truly non-outliers over the total number of non-outliers.

$$Probability\ of\ False\ Alarm = \frac{k - Number\ of\ Outliers\ Found}{Number\ of\ NonOutliers}$$

The *Probability of Miss* is the number of true outliers that were not detected (missed) over the total number of actual outliers in the dataset.

$$Probability\ of\ Miss = \frac{Total\ Number\ of\ Outliers - Number\ of\ Outliers\ Found}{Total\ Number\ of\ Outliers}$$

	Probability of False Alarm				*Probability of Miss*			
k	Greedy	AVF	FindFPOF	FDOD	Greedy	AVF	FindFPOF	FDOD
4	0.0	0.0	0.2	0.2	89.7	89.7	92.3	92.3
8	0.0	0.2	0.2	0.2	79.5	82.1	82.1	82.1
16	0.2	0.5	0.5	0.2	61.5	64.1	64.1	61.5
24	0.5	0.7	0.7	0.7	43.6	46.2	46.2	46.2
32	0.7	0.9	1.1	0.9	25.6	28.2	30.8	28.2
40	1.6	1.8	2.0	1.6	15.4	17.9	20.5	15.4
48	2.5	2.7	2.9	2.5	5.1	7.7	10.3	5.1
56	3.8	3.8	3.8	3.8	0.0	0.0	0.0	0.0

Table 10 Wisconsin Breast Cancer *Probability of False Alarm* and *Probability of Miss*

The *Probability of Miss* quickly approaches zero, while the *Probability of False Alarm* increases slightly.

To better illustrate the relationship between the *Probability of Miss* and the *Probability of False Alarm*, we present Figure 9. This figure represents the ideal relationship between the *Probability of False Alarm* and the *Probability of Miss*. The

Probability of False Alarm increases very little, while the *Probability of Miss* quickly approaches zero.

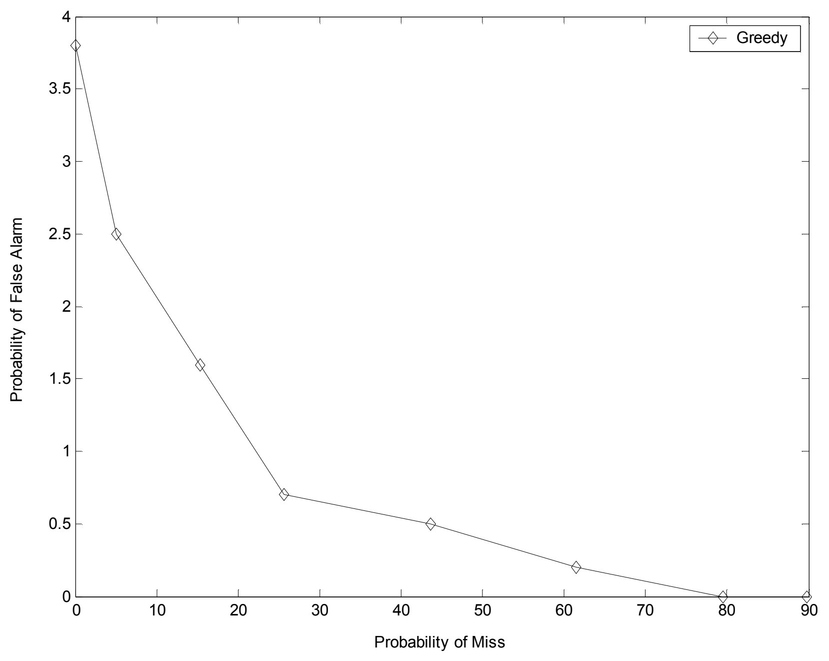

Figure 9 *Probability of False Alarm* vs. *Probability of Miss* for Greedy Algorithm using Breast Cancer Dataset

This figure illustrates that as the *Probability of Miss* converges to zero, the *Probability of False Alarm* increases. The *Probability of False Alarm* is negligible considering all outliers are detected.

4.3.2 Lymphography

Like the Wisconsin breast cancer dataset, the lymphography dataset is very imbalanced and results in the detection of all outliers. Table 11 presents the rates at which the algorithms converge on the outliers in the lymphography dataset.

	Greedy		AVF		FindFPOF		FDOD	
k	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**
2	2	33.33	2	33.33	2	33.33	2	33.33
4	4	66.67	4	66.67	4	66.67	4	66.67
6	5	83.33	4	66.67	4	66.67	4	66.67
8	6	100.00	5	83.33	5	83.33	5	83.33
12	6	100.00	6	100.00	5	83.33	5	83.33
13	6	100.00	6	100.00	5	100.00	6	100.00
15	6	100.00	6	100.00	6	100.00	6	100.00

Table 11 Results (Outliers Detected) for All Outlier Detection Algorithms using the Lymphography Dataset

The Greedy algorithm converges on the number of outliers the quickest, followed by AVF, FindFPOF and FDOD. Below, in Figure 10, it is apparent the algorithms converge at slightly different rates.

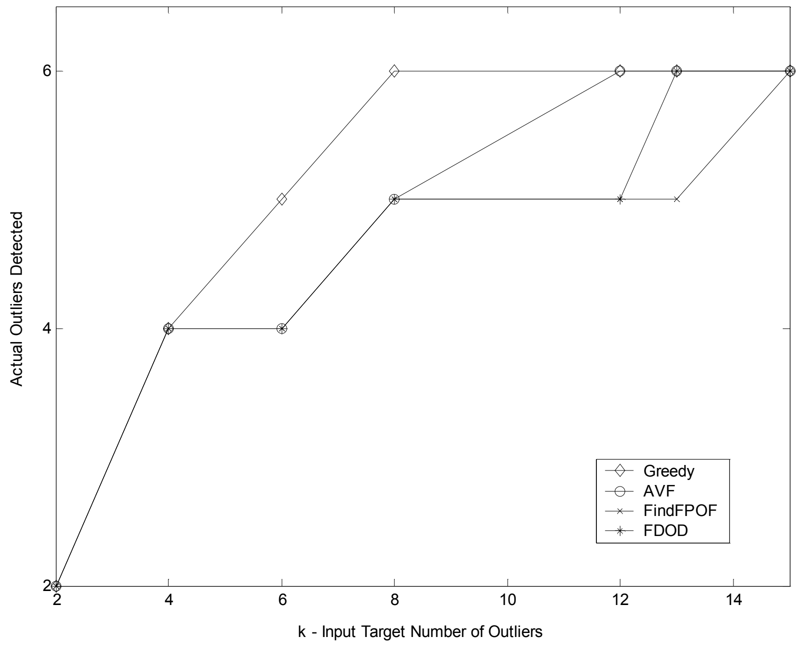

Figure 10 Input Target Outliers, *k*, vs. Actual Outliers Found for the Lymphography Datset

Greedy detects all the six outliers the quickest, followed by AVF, FindFPOF and FDOD.

The algorithms for this dataset also converge on the outliers with a very low *Probability of False Alarm*. Note that the presence of non-outliers is negligible when all outliers have been detected.

	Probability of False Alarm				*Probability of Miss*			
k	**Greedy**	**AVF**	**FindFPOF**	**FDOD**	**Greedy**	**AVF**	**FindFPOF**	**FDOD**
2	0.0	0.0	0.0	0.0	66.7	66.7	66.7	66.7
4	0.0	0.0	0.0	0.0	33.3	33.3	33.3	33.3
6	0.7	1.4	1.4	1.4	16.7	33.3	33.3	33.3
8	1.4	2.1	2.1	2.1	0.0	16.7	16.7	16.7
12	4.2	4.2	4.9	4.9	0.0	0.0	16.7	16.7
13	4.9	4.9	5.6	4.9	0.0	0.0	16.7	0.0
15	6.3	6.3	6.3	6.3	0.0	0.0	0.0	0.0

Table 12 Lymphography *Probabilities of False Alarm* and *Probability of Miss*

The *Probability of False Alarm* is negligible considering the *Probability of Miss* is zero.

4.3.3 Post-operative Patients

The post-operative patients dataset is not as imbalanced as the other datasets discussed earlier. In addition, the attributes do not define the data points into distinct classes, i.e. outliers and non-outliers. As a result, none of the algorithms detect all of the outliers. See Table 13 for more information.

	Greedy		AVF		FindFPOF		FDOD	
k	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**
10	4	15.38	3	11.54	3	11.54	1	3.85
20	7	26.92	7	26.92	7	26.92	7	26.92
30	8	30.77	10	38.46	9	34.62	9	34.62
40	12	46.15	11	42.31	10	38.46	10	38.46
50	13	50.00	12	46.15	12	46.15	13	50.00
60	20	76.92	16	61.54	17	65.38	18	69.23
70	21	80.77	21	80.77	21	80.77	21	80.77
80	24	92.31	24	92.31	24	92.31	24	92.31

Table 13 Results (Outliers Detected) for All Outlier Detection Algorithms using the Post-operative Patients Dataset

Though none of the algorithms converge on all of the outliers in the dataset, it is important to note that all of the algorithms detect the outliers at the same rate, as seen in Figure 11.

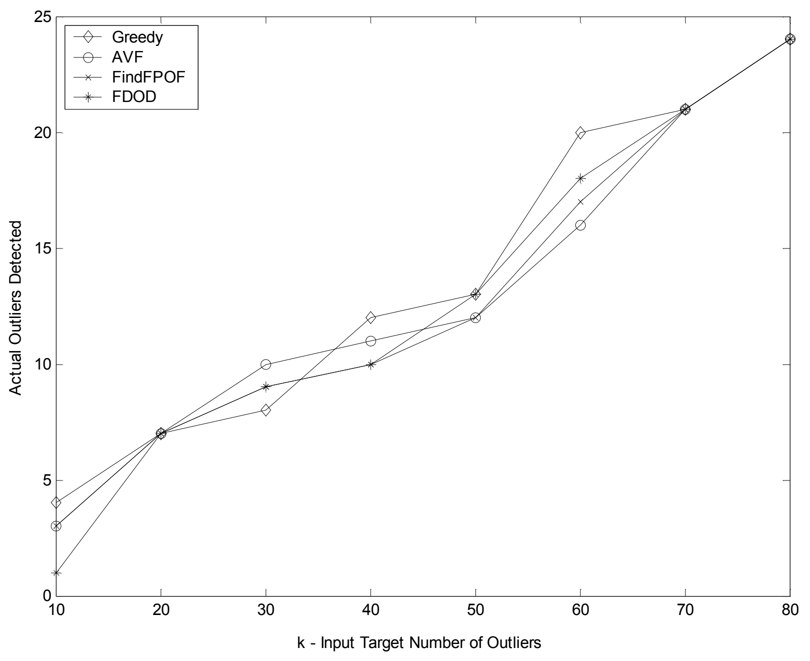

Figure 11 Input Target Outliers, *k*, vs. Actual Outliers Found for the Post-operative Patients Dataset

All algorithms have comparable results.

For this dataset, the ineptitude of the algorithms in detecting the outliers in this dataset is made evident in the rapid increase in the *Probability of False Alarm*. Table 14 shows the probabilities.

	Probability of False Alarm				***Probability of Miss***			
k	**Greedy**	**AVF**	**FindFPOF**	**FDOD**	**Greedy**	**AVF**	**FindFPOF**	**FDOD**
10	9.4	10.9	10.9	14.1	84.6	88.5	88.5	96.2
20	20.3	20.3	20.3	20.3	73.1	73.1	73.1	73.1
30	34.4	31.3	32.8	32.8	69.2	61.5	65.4	65.4
40	43.8	45.3	46.9	46.9	53.8	57.7	61.5	61.5
50	57.8	59.4	59.4	57.8	50.0	53.8	53.8	50.0
60	62.5	68.8	67.2	65.6	23.1	38.5	34.6	30.8
70	76.6	76.6	76.6	76.6	19.2	19.2	19.2	19.2
80	87.5	87.5	87.5	87.5	7.7	7.7	7.7	7.7

Table 14 Post-operative Patient *Probability of False Alarm* and *Probability of Miss*

The *Probability of Miss* decreases slowly since the dataset is not imbalanced enough. The *Probability of False Alarm* also increases too rapidly, which is not ideal.

4.3.4 Page Blocks

The algorithms do not fair well at detecting all the outliers for the page blocks dataset, like the post-operative patients dataset. Table 15 shows that the Greedy algorithm detects the outliers at the quickest rate followed by the AVF algorithm.

	Greedy		AVF		FindFPOF		FDOD	
k	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**	**No. Found**	**% of Total Outliers**
100	45	16.07	40	14.29	19	6.79	19	6.79
200	81	28.93	84	30.00	42	15.00	42	15.00
300	130	46.43	120	42.86	63	22.50	63	22.50
400	157	56.07	168	60.00	74	26.43	74	26.43
500	177	63.21	189	67.50	80	28.57	80	28.57
600	183	65.36	201	71.79	94	33.57	94	33.57
700	213	76.07	206	73.57	96	34.29	96	34.29
800	237	84.64	214	76.43	110	39.29	110	39.29
900	242	86.43	223	79.64	116	41.43	116	41.43
1000	242	86.43	233	83.21	121	43.21	121	43.21

Table 15 Results (Outliers Detected) for All Outlier Detection Algorithms using the Page Blocks Dataset

Greedy and AVF have comparable results in number of outliers found. FindFPOF and FDOD detect outliers at half the rate of Greedy and AVF.

Even though the AVF algorithm has the second quickest rate, Figure 12 shows the large difference between the total outliers detected by the Greedy and AVF algorithms.

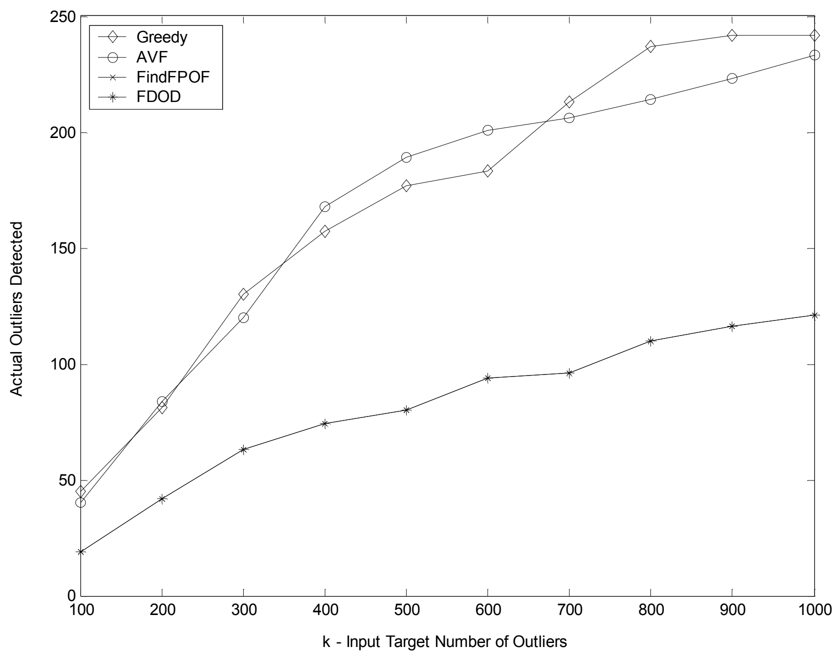

Figure 12 Input Target Outliers, *k*, vs. Actual Outliers Found for the Page Blocks Datset

The Greedy and AVF algorithms converge close to each other and have better accuracy than the FindFPOF and FDOD algorithms. FindFPOF has identical results as FDOD and is covered by the graph of FDOD.

For the Greedy and AVF algorithms, the *Probability of False Alarm* does not grow as rapidly as that for the post-operative patients dataset. The FindFPOF and FDOD algorithms do not converge on the outliers as quickly as the Greedy and AVF algorithms.

	Probability of False Alarm				*Probability of Miss*			
k	**Greedy**	**AVF**	**FindFPOF**	**FDOD**	**Greedy**	**AVF**	**FindFPOF**	**FDOD**
100	1.1	1.2	1.6	1.6	83.9	85.7	93.2	93.2
200	2.4	2.4	3.2	3.2	71.1	70.0	85.0	85.0
300	3.5	3.7	4.8	4.8	53.6	57.1	77.5	77.5
400	4.9	4.7	6.6	6.6	43.9	40.0	73.6	73.6
500	6.6	6.3	8.5	8.5	36.8	32.5	71.4	71.4
600	8.5	8.1	10.3	10.3	34.6	28.2	66.4	66.4
700	9.9	10.1	12.3	12.3	23.9	26.4	65.7	65.7
800	11.5	11.9	14.0	14.0	15.4	23.6	60.7	60.7
900	13.4	13.8	16.0	16.0	13.6	20.4	58.6	58.6
1000	15.4	15.6	17.9	17.9	13.6	16.8	56.8	56.8

Table 16 Page Block *Probability of False Alarm* and *Probability of Miss*

The *Probability of Miss* decreases quickly for the Greedy and AVF algorithm, while the *Probability of False Alarm* does not increase that quickly. For the FindFPOF and FDOD algorithms, the *Probability of Miss* does not decrease rapidly.

4.3.5 Simulated

The first generated dataset, as stated previously, has 10 attributes, 10 distinct attribute values per attribute and ranges from 1,000 to 800,000 points. For these simulations, the input *k* is kept constant at 30. Table 17 shows the timing results for each *k* input.

Data Size	Greedy	AVF	FindFPOF	FDOD
1,000	0.27	0.00	0.81	4.58
10,000	2.72	0.03	8.13	44.72
30,000	8.53	0.06	24.02	134.30
50,000	14.31	0.09	40.19	222.88
100,000	26.42	0.19	81.06	445.39
200,000	52.75	0.39	165.08	891.28
300,000	79.39	0.58	241.61	1337.06
400,000	106.14	0.80	323.97	1781.78
500,000	131.75	0.94	404.45	2233.74
600,000	158.70	1.16	484.00	2678.73
700,000	184.94	1.33	564.80	3127.22
800,000	212.08	1.56	667.55	3568.55

Table 17 Scalability of Varying Data Size (Time Measured in Seconds)

Notice how quickly the Greedy, FindFPOF, and FDOD algorithms are increasing in comparison to the others.

The outcome shows that the Greedy, FindFPOF and FDOD algorithms do not scale gracefully. The timing for all of the algorithms increases; however, the AVF algorithm grows slowly since it only has to do one pass over the dataset. The scalability of Greedy is heavily dependent on the size of the dataset. The FindFPOF and FDOD algorithms also slow down in execution time because of the *Apriori* algorithm. Figure 13 shows a better representation of the scaling.

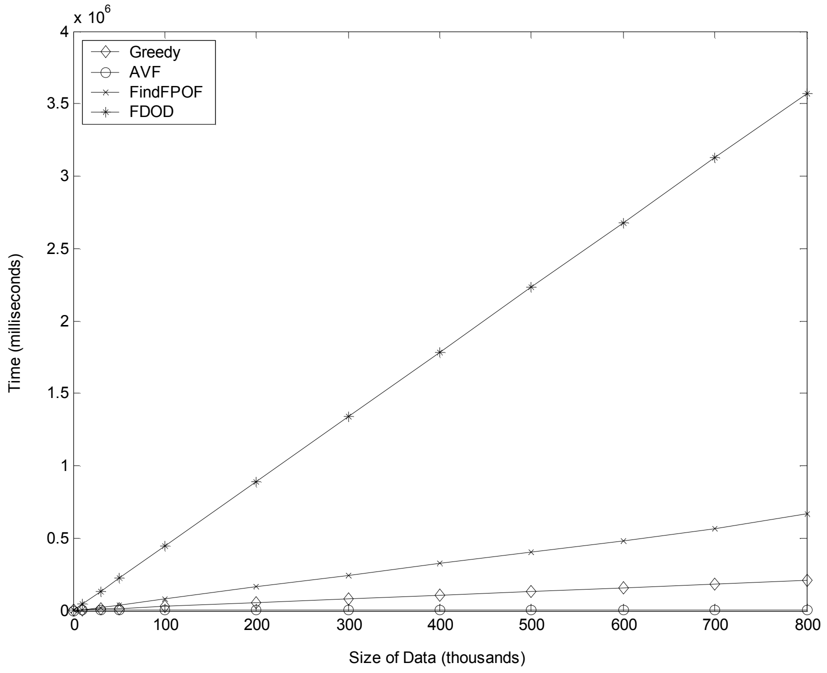

Figure 13 Graph of Varying Data Dimension (thousands) vs. Time (milliseconds)

In the figure above, all algorithms increase in a linear fashion. FDOD increases the quickest, while AVF increases at the slowest rate.

The second dataset contains 100,000 points, 10 distinct attribute values per attribute, and the number of attributes is varied from 2 to 30. The input *k* is kept constant at 30.

Number of Attributes	Greedy	AVF	FindFPOF	FDOD
2	5.81	0.06	1.08	1.88
5	13.61	0.11	2.83	5.45
10	27.02	0.19	5.36	13.17
20	53.06	0.38	10.31	34.44
30	80.17	0.59	15.83	67.50

Table 18 Varying Number of Attributes (Time Measured in Seconds)

The results in Table 18 show that Greedy and FDOD increase more quickly by a large factor compared to AVF and FindFPOF. The AVF algorithm remains under 1 second in execution time, while the FindFPOF algorithms grow slowly in execution time. Figure

14, below, shows a plot of the aforementioned results. All algorithms scale linearly, except for FDOD. FDOD grows exponentially, as was noted in the algorithms description. The Greedy and FDOD algorithms increase the fastest in execution time, whereas AVF grows the slowest. The number of attributes affects the Greedy algorithm because at each iteration it must compute the entropy for each attribute. We attribute the increase in run time for the FindFPOF and FDOD algorithms to the execution of the *Apriori* algorithm, which mines for frequent patterns. Once again, the AVF algorithm only performs one pass over the dataset, which is why the execution time only increases slightly as the number of attributes increases.

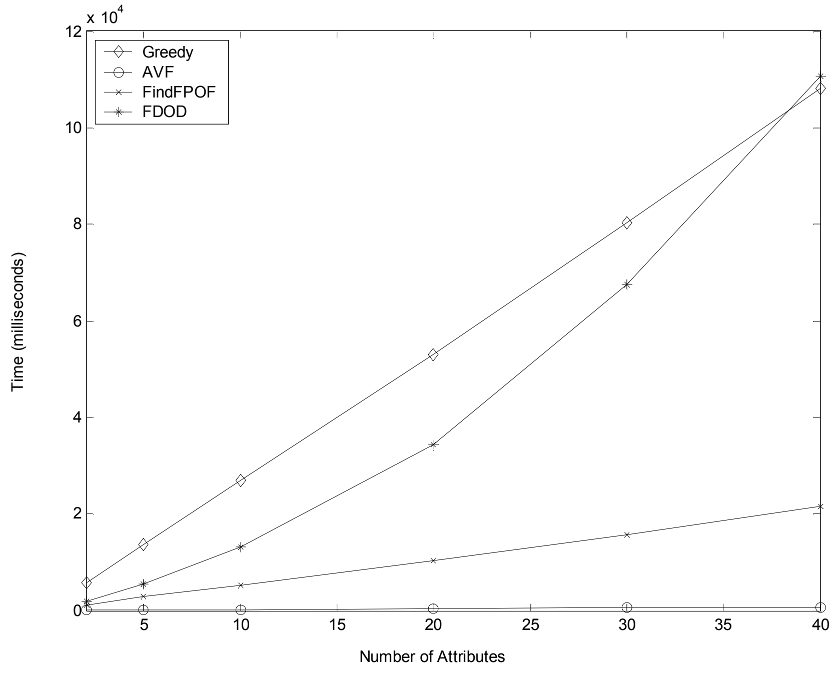

Figure 14 Number of Attributes vs. Time (milliseconds)

In this figure, the Greedy algorithm has the largest linear slope and FDOD algorithm increases exponentially. The AVF algorithm has smallest slope.

The third generated dataset has 100,000 points, 10 attributes, and the number of distinct attribute values per attribute varies from 5 to 40. Again, the input *k* is kept constant at 30. Table 19 shows the results of varying the number of attribute values per attribute.

Number of Attribute Values	Greedy	AVF	FindFPOF	FDOD
5	26.41	0.20	164.67	625.38
10	26.31	0.24	5.30	13.17
20	26.26	0.20	2.33	1.52
30	26.02	0.19	2.33	1.53
40	26.17	0.22	2.31	1.52

Table 19 Varying Number of Attribute Values per Attribute (Time Measured in Seconds)

For all algorithms the execution remains relatively constant, however the FDOD algorithm decreases quickly in execution time.

The plot of the results for various numbers of attribute values per attribute is shown in Figure 15. The Greedy and AVF algorithms have a relatively constant execution time. Interestingly, the FindFPOF and FDOD algorithms decrease in execution time, which seems to follow an inverse relationship. We attribute this to the fact that the number of attribute values per attribute, *V,* increases, while the number of attributes, *m*, and the size of the data, *n*, does not. As the number of attribute values per attribute, *V*, increases fewer and fewer attribute values occur frequently. This results in fewer frequent itemsets. The number of attribute values does not affect the Greedy or AVF because it is too small to affect the hashing of the frequencies and the subsequent calculations.

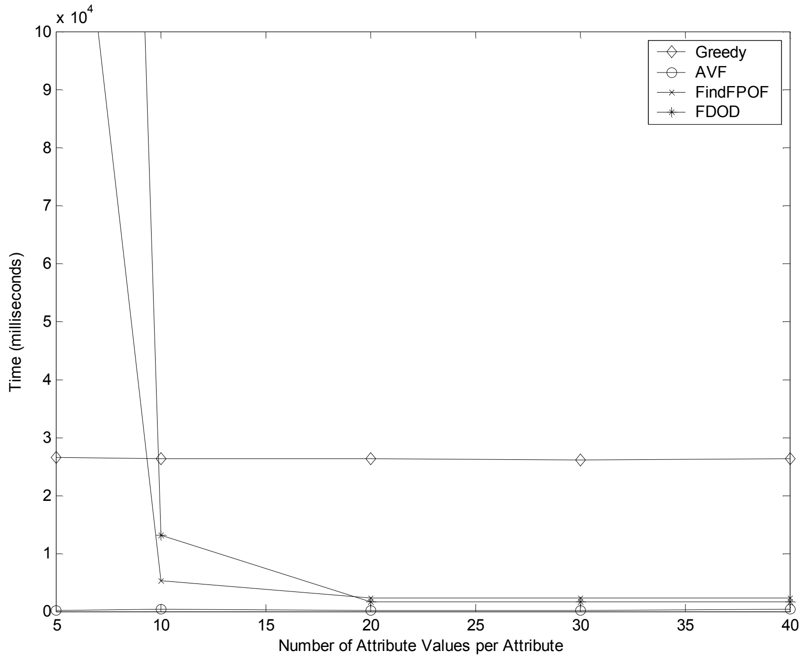

Figure 15 Number of Attribute Values vs. Time (seconds)

The Greedy and AVF algorithms have a slope close to zero; however, the FindFPOF and FDOD algorithms decrease in execution exhibiting an inverse relationship.

The fourth and final dataset contains 100,000 points, 10 attributes, and 10 attribute values. The *k* input value is varied from 1 to 1,000. The results are presented in Table 20.

***k*-input**	**Greedy**	**AVF**	**FindFPOF**	**FDOD**
1	0.97	0.19	81.39	445.74
10	9.02	0.17	80.83	446.25
30	26.39	0.19	81.31	445.56
50	43.81	0.20	80.86	446.36
100	87.59	0.22	80.89	447.55
200	175.14	0.28	81.53	445.91
300	262.34	0.31	81.03	445.84
400	349.63	0.38	80.80	445.30
500	436.84	0.42	80.56	445.81
600	526.63	0.47	80.83	446.14
700	611.02	0.52	83.47	445.88
800	697.86	0.58	82.39	446.16
900	784.92	0.63	81.53	448.74
1000	871.44	0.67	81.09	446.48

Table 20 Varying *k*-input (Time Measured in Seconds)

The AVF, FindFPOF, and FDOD algorithms are all relatively constant in execution time; however, the AVF algorithm performs the best with a constant time close to zero. The Greedy algorithm performs the worst, increasing in execution time as the *k* value increases. Figure 16 shows that Greedy has a steep slope compared to the other algorithms. The Greedy algorithm increases drastically in execution time as the input-*k* value grows because it must conduct *k* passes over the dataset to find *k* outliers. However, the AVF, FindFPOF, and FDOD algorithms simply find the top-*k* outliers in one pass over the dataset.

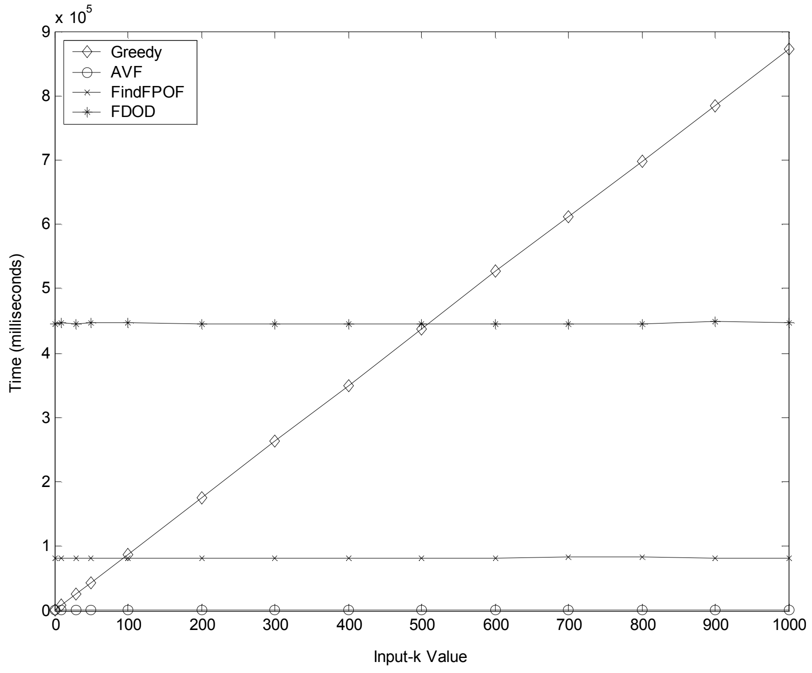

Figure 16 Input-k (thousands) vs. Time (milliseconds)

The Greedy algorithm has a sharp slope in contrast to the near zero slope of the other algorithms.

4.4 Discussion

As seen in results from the previous sections with real and artificially generated datasets, AVF approximates very well the outlier detection capabilities of the Greedy algorithm, while it performs very well for larger values of the size of the dataset (*n*) dataset dimensionality (*m*) and the target number of outliers (*k*). The Greedy algorithm becomes exceedingly slow for large values of *n*. This is because the entropy calculations become increasingly more expensive as *n* increases. FindFPOF and FDOD also become slower for larger datasets, which is because they need to search through all the possible subsets of each data point.

In addition, the Greedy algorithm slows down considerably for large values of *k* as well as for larger dimensionalities, *m*. In contrast, the AVF, FindFPOF and FDOD algorithms do not depend on *k*, so their performance remains constant for each different *k* value. However, higher dimensionalities, *m*, do slow down the FindFPOF and Greedy algorithms, because they result in the production of more frequent itemsets. The authors in [7, 8] discuss entering a MAXLEVEL parameter, in order to put an upper bound to the length of the possible itemsets created. For example if the user specifies a maximum itemset length of five, the itemsets created cannot contain more than five items. However, the MAXLEVEL usage in [8] negatively influences the accuracy of outlier detection in their algorithms.

The FindFPOF and FDOD algorithms depend on the minimum support threshold *minsup* entered by the user. The accuracy of FindFPOF and FDOD can be improved if different values for *minsup* are used. We ran several different tests for pageblocks varying the value of *minsup*. For example, for *k* = 200 and *minsup* = 0.04, FDOD detected 120 outliers (in contrast to the 42 in Table 15); however, for *minsup* = 0.03, FDOD detects 51 outliers. This is because a lower *minsup* value makes more subsets frequent, while a higher *minsup* value makes more subsets infrequent. This is an example of how challenging it can be to select the appropriate minimum support threshold specifically for each dataset and application.

AVF has the advantage that it does not create itemsets and assumes only a single pass over the entire dataset. Therefore, its complexity is only dependent on the size, *n*, and the dimensionality of the dataset, *m*. In addition, AVF eliminates the need to make difficult choices for any user-given parameters such as minimum support or maximum itemset length.

Revisiting the example discussed in Section 3.4 shows that an additional outlier, customer 3, exists. This is because point 3 contains the infrequent itemset {Young, Sedan}. All other instances of the attribute value “Young” are combined with “Sports”. Every algorithm, except for AVF, succeeds at detecting customer 3 as the next most likely outlier. Based on this knowledge we intend to experiment with techniques to increase the outlier detection accuracy of AVF without significantly deteriorating its performance.

CHAPTER 5 Conclusions and Future Plans

Outlier detection is a research area that has received much attention. However, most of the research has focused on numerical datasets, and not specifically on the detection of outliers in categorical data. In addition, the research that has focused on categorical data have not been contrasted to each other. We have focused on four outlier detection algorithms for categorical data, including one algorithm that we developed (our own innovation). The algorithms developed by other researchers include Greedy [6], FindFPOF [7], and FDOD [8]. The algorithm that we introduced, called AVF, has shown to be an effective (accurate) and efficient (scalable) technique, which lends itself to the nature of data today, as AVF's performance does not deteriorate with large datasets or for datasets with high dimensionalities.

Our experiments demonstrated that AVF is as effective (accurate) in detecting outliers, as existing and representative outlier detection strategies, reported earlier in the literature. Furthermore, our experiments have demonstrated that AVF is more efficient (faster), and quite often a lot more efficient than these representative existing outlier detection strategies. One of the limitations of AVF, mentioned at the end of the previous section, that was evident in the example presented in Section 3.4 and in the page blocks results when comparing Greedy and AVF, is the tradeoff of accuracy when detecting outlier one pass, which is much faster than the current algorithms. How to overcome this tradeoff is the topic of some of our future work. We also plan to modify and extend the ideas presented in this thesis to apply to a distributed setting.

List of References

[1] D. Hawkins, *Identification of Outliers*. London: Chapman and Hall, Reading, 1980.

[2] E. Knorr, R. Ng, and V. Tucakov, "Distance-based outliers: Algorithms and applications," *VLDB Journal*, 2000.

[3] R. J. Bolton and D. J. Hand, "Statistical fraud detection: A review," *Statistical Science*, vol. 17, pp. 235–255, 2002.

[4] K. I. Penny and I. T. Jolliffe, "A comparison of multivariate outlier detection methods for clinical laboratory safety data," *The Statistician, Journal of the Royal Statistical Society*, vol. 50, pp. 295–308, 2001.

[5] Z. He, X. Xu, and S. Deng, "An Optimization Model for Outlier Detection in Categorical Data," *Lecture Notes in Computer Science*, vol. 3644, pp. 400-409, 2005.

[6] Z. He, X. Xu, and S. Deng, "A Fast Greedy Algorithm for Outlier Mining," presented at the PAKDD 2006 Conference, Singapore, 2006.

[7] Z. He, X. Xu, J. Huang, and S. Deng, "FP-Outlier: Frequent Pattern Based Outlier Detection", Computer Science and Information System (ComSIS'05)," 2005.

[8] M. E. Otey, A. Ghoting, and and A. Parthasarathy, "Fast Distributed Outlier Detection in Mixed-Attribute Data Sets," *Data Mining and Knowledge Discovery*, 2006.

[9] V. Barnett and T. Lewis, *Outliers in Statistical Data*: John Wiley, 1994.

[10] P. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*: Pearson Addison-Wesley, 2005.

[11] V. Hodge and J. Austin, "A Survey of Outlier Detection Methodologies," *Artificial Intelligence Review*, vol. 22, pp. 85, 2004.

[12] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*. Berlin Heidelberg NY Springer, 1998.

[13] I. Ruts and P. Rousseeuw, "Computing depth contours of bivariate point clouds," *Comput. Stat Data Anal*, pp. 153-168, 1996.

[14] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying density-based local outliers," presented at ACM SIGMOD International Conference on Management of Data, 2000.

[15] S. D. Bay and M. Schwabacher, "Mining distance-based outliers in near linear time with randomization and a simple pruning rule," presented at ACM SIGKDD Int'l Conference on Knowledge Discovery and Data Mining, 2003.

[16] S. Shekhar, C. Lu, and P. Zhang, "Detecting Graph-Based Spatial Outliers: Algorithms and Applications," presented at 7th ACM SIGKDD Int'l Conference on Knowledge Discovery and Data Mining, 2001.

[17] S. Papadimitriou, H. Kitawaga, P. Gibbons, and C. Faloutsos, "LOCI: Fast outlier detection using the local correlation integral," presented at International Conference on Data Engineering, 2003.

[18] L. Wei, W. Qian, A. Zhou, and W. Jin, "HOT: Hypergraph-based Outlier Test for Categorical Data," presented at 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2003), 2003.

[19] D. Tax and R. Duin, "Support Vector Data Description," *Machine Learning*, pp. 45-66, 2004.

[20] S. Harkins, H. He, G. Williams, and R. and Baster, "Outlier Detection Using Replicator Neural Networks," presented at Fifth International Conference Data Warehousing and Knowledge Discovery, 2002.

[21] Y. Pei, O. Zaiane, and Y. Gao, "An Efficient Reference-based Approach to Outlier Detection in Large Dataset," presented at Sixth IEEE International Conference on Data Mining (ICDM'06), 2006.

[22] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," presented at International Conference on Very Large Data Bases VLDB, 1994.

[23] J. Xu Yu, W. Qian, H. Lu, and A. Zhou, "Finding centric local outliers in categorical/numerical spaces," *Knowledge Information Systems*, vol. 9, pp. 309–338, 2006.

[24] C. E. Shannon, *A Mathematical Theory of Communication*: Bell System Technical Journal, 1948.

[25] "Google Hash Table," http://google-sparsehash.googlecode.com/svn/trunk/docs/index.html

[26] H. Liu, F. Hussain, C. L. Tan, and M. Dash, "Discretization: An Enabling Technique," *Data Mining and Knowledge Discovery*, vol. 6, pp. 393-423, 2002.

[27] C. Blake and C. Merz, "UCI Machine Learning Repository," 1998.

[28] D. Cristofor, http://www.cs.umb.edu/~dana/GAClust/index.html

Appendix A

Build Instructions

1. Run Matlab
2. Entropy and Attribute-Value Frequency (AVF) Algorithms
   - From folder: Outlier Detection
     - mex -g outlierdetection.cpp
     - mex outlierdetection.cpp
3. Frequent Itemset Mining Based Algorithms
   - From folder: Frequent Itemset Outlier Detection\Matlab
     - mex -g ../Code/OutlierDetection/OutlierDetectionAlgorithms.cpp ../Code/Apriori/Myth/MythAprioriDataProvider.cpp ../Code/Apriori/Apriori.cpp ../Code/Apriori/AprioriTrie.cpp ../Code/Apriori/IAprioriDataProvider.cpp ../Code/Apriori/Trie.cpp
     - mex ../Code/OutlierDetection/outlierdetectionalgorithms.cpp ../Code/Apriori/Myth/MythAprioriDataProvider.cpp ../Code/Apriori/Apriori.cpp ../Code/Apriori/AprioriTrie.cpp ../Code/Apriori/IAprioriDataProvider.cpp ../Code/Apriori/Trie.cpp

Execution Instructions

Entropy and Attribute-Value Frequency (AVF) Algorithms

[outliers, time] = outlierdetection(data, k, algorithm)

Inputs:

- data – n (number of points) x m (number of attributes) dataset
- k – target number of outliers to be detected
- algorithm – which algorithm to run (1: LSA, 2: Greedy, and 3: AVF)

Outputs:

- outliers – labels for outliers and non-outliers (1 for outliers and 0 for non-outliers)
- time – execution time in milliseconds

Frequent Itemset Mining Based Algorithms

[flags, time(i)] = OutlierDetectionAlgorithms(data, minSup, k, algorithm);

Inputs:

- data – n (number of points) x m (number of attributes) dataset

- minSup – minimum support (frequency) threshold for frequent itemset mining
- k – target number of outliers to be detected
- algorithm – which algorithm to run (1: LSA, 2: Greedy, and 3: AVF)

Outputs:

- outliers – labels for outliers and non-outliers (1 for outliers and 0 for non-outliers)
- time – execution time in milliseconds

Appendix B

OutlierDetectionAlgorithms Class Reference

Contains all Frequent Itemset Mining based algorithms.

We use the MythAprioriDataProvider to access each data point (transaction) in the dataset. It also stores a vector of the frequent itemsets. The function we use to acquire each transaction is:

MythAprioriDataProvider::readLine(set<ItemType> &transaction.

It returns the itemset in a standard template set.

```
#include <OutlierDetectionAlgorithms.h>
```

Public Member Functions

- **OutlierDetectionAlgorithms** ()
  Constructor.

- **~OutlierDetectionAlgorithms** ()
  Destructor.

- double **RemoveOutliers** (**MythAprioriDataProvider** *mADP, double minSupp, int targetK, int option)
  Detects outliers in dataset using the algorithm chosen (option).

- bool **FindFPOF** (**MythAprioriDataProvider** *mADP, double minSup, int targetK)
  Runs He's technique using Frequent Pattern Outlier Factor (FPOF).

- bool **OteyAlternateApproach** (**MythAprioriDataProvider** *mADP, double minSup, int targetK)
  Runs faster version of Otey's technique using his outlier score. Algorithm called Fast Distributed Outlier Detection (FDOD).

- int * **GetFlags** ()
  Returns points to flags array.

- void **PrintFlags** ()
  Displays tagged outliers to screen.

- void **PrintOutlierness** ()
  Displays outlier score for each point to screen.

Protected Member Functions

- double **CalculateOteyOutlierFactor** (**MythAprioriDataProvider** *mADP, set< ItemType > *transaction, vector< **ItemSet** > *frequentItemSets, vector< int > combinationCount)
  Calculate outlier factor using Otey's function.

- void **RemoveFrequentItemsets** (**MythAprioriDataProvider** *mADP, set< ItemType > *transaction, vector< **ItemSet** > *frequentItemSets, vector< int > *combinationCount)
  Remove frequent itemsets from vector counting infrequent itemsets.

- int **CalculateCombinations** (int n, int k)
  Calculates the number of combinations; the number of ways of picking k unordered outcomes from n possibilities.

- double **Factorial** (int num)
  Calculates the factorial of a number.

Detailed Description

Contains all Frequent Itemset Mining based algorithms.

Implements several outlier detection using matlab matrices implemented using templates provided through **MYTH.h**. The user can input n data vectors with m dimensions which translates to n rows and m columns. An outlier detection algorithm can be ran to extract outliers which are stored in a matrix. Two algorithms are implementations based on Frequent Itemset Mining.

Definition at line 101 of file OutlierDetectionAlgorithms.h.

Member Function Documentation

double OutlierDetectionAlgorithms::RemoveOutliers (MythAprioriDataProvider * *data*, double *minSupp*, int *targetK*, int *option*)

Detects outliers in dataset using the algorithm chosen (option).

Parameters:

data Data provider for myth matrix.
minSupp Minimum support (threshold) value for frequent itemset mining.
targetK Target number of outliers to be detected.
option Value of algorithm wishes to execute.

Returns:

Returns true if algorithm successfully executed.

Definition at line 156 of file OutlierDetectionAlgorithms.cpp.

References FindFPOF(), and OteyAlternateApproach().

bool OutlierDetectionAlgorithms::FindFPOF (MythAprioriDataProvider * *mADP*, double *minSup*, int *targetK*)

Runs He's technique using Frequent Pattern Outlier Factor (FPOF).

The outlierness score is calculated based on if the point contains a frequent itemset. Therefore, the more frequent itemsets the point contains, the higher the score. Outliers are those with the lowest score.

Parameters:

mADP A data provider that extracts the points from a dataset.
minSup The minimum support used to mine frequent itemsets.
targetK Target number of outliers

Returns:

Returns true if algorithm is successfully executed.

Definition at line 201 of file OutlierDetectionAlgorithms.cpp.

References Apriori::Apriori_Algorithm(), MythAprioriDataProvider::DecodeValue(), MythAprioriDataProvider::GetDataMatrix(), MythAprioriDataProvider::GetItemSetVector(), MythAprioriDataProvider::ReadLine(), and MythAprioriDataProvider::Reset().

Referenced by RemoveOutliers().

bool OutlierDetectionAlgorithms::OteyAlternateApproach (MythAprioriDataProvider * *mADP*, double *minSup*, int *targetK*)

Runs faster version of Otey's technique using his outlier score. Algorithm called Fast Distributed Outlier Detection (FDOD).

The outlierness score is calculated based on if the point does not contain a frequent itemset. Therefore, the less frequent itemsets the point contains, the higher the score. Outliers are those with the highest score. The alternate approach to the Otey algorithm was created in response to the slowness of the original. This uses the combination function (nCk)to determine the number of combinations there are for each transaction and removing the ones that are frequent.

Parameters:

mADP A data provider that extracts the points from a dataset.
minSup The minimum support used to mine frequent itemsets.
targetK Target number of outliers to be detected.

Returns:

Returns true if algorithm is successfully executed.

Definition at line 377 of file OutlierDetectionAlgorithms.cpp.

References Apriori::Apriori_Algorithm(), CalculateCombinations(), CalculateOteyOutlierFactor(), MythAprioriDataProvider::DecodeValue(), MythAprioriDataProvider::GetDataMatrix(), MythAprioriDataProvider::GetItemSetVector(), MythAprioriDataProvider::ReadLine(), and MythAprioriDataProvider::Reset().

Referenced by RemoveOutliers().

int * OutlierDetectionAlgorithms::GetFlags ()

Returns points to flags array.

Returns:

Return pointer to flags array.

Definition at line 681 of file OutlierDetectionAlgorithms.cpp.double OutlierDetectionAlgorithms::CalculateOteyOutlierFactor (MythAprioriDataProvider * *mADP*, set< ItemType > * *transaction*, vector< ItemSet > * *frequentItemSets*, vector< int > *combinationCount*) `[protected]`

Calculate outlier factor using Otey's function.

Parameters:

mADP A data provider that extracts the points from a dataset.
transaction Transaction/itemset under consideration.
frequentItemSets Frequent itemsets of the whole datasets.
combinationCount The combinations for the different number of literals.

Returns:

Returns the outlier score.

Definition at line 532 of file OutlierDetectionAlgorithms.cpp.

References RemoveFrequentItemsets().

Referenced by OteyAlternateApproach().

void OutlierDetectionAlgorithms::RemoveFrequentItemsets (MythAprioriDataProvider * *mADP*, set< ItemType > * *transaction*, vector< ItemSet > * *frequentItemSets*, vector< int > * *combinationCount*) `[protected]`

Remove frequent itemsets from vector counting infrequent itemsets.

Parameters:

mADP A data provider that extracts the points from a dataset.
transaction Transaction/itemset under consideration.
frequentItemSets Frequent itemsets of the whole datasets.
combinationCount The combinations for the different number of literals.

Returns:

void

Definition at line 566 of file OutlierDetectionAlgorithms.cpp.

References MythAprioriDataProvider::DecodeValue().

Referenced by CalculateOteyOutlierFactor().

int OutlierDetectionAlgorithms::CalculateCombinations (int *n*, int *k*) `[protected]`

Calculates the number of combinations; the number of ways of picking k unordered outcomes from n possibilities.

Parameters:

n The number of possibilities.
k The number of unordered outcomes.

Returns:

The number of ways of picking k unordered outcomes from n possibilities.

Definition at line 644 of file OutlierDetectionAlgorithms.cpp.

References Factorial().

Referenced by OteyAlternateApproach().

double OutlierDetectionAlgorithms::Factorial (int *num*) `[protected]`

Calculates the factorial of a number.

Parameters:

num The number to be "factorialized."

Returns:

Return the factorial of num.

Definition at line 662 of file OutlierDetectionAlgorithms.cpp.

Referenced by CalculateCombinations().

Generated using *Doxygen*.

The documentation for this class was generated from the following files:

- M:/Documents/Research/Outlier Research/Code/Frequent Itemset Outlier Detection/Code/OutlierDetection/OutlierDetectionAlgorithms.h
- M:/Documents/Research/Outlier Research/Code/Frequent Itemset Outlier Detection/Code/OutlierDetection/OutlierDetectionAlgorithms.cpp

Appendix C

OutlierDetection Class Reference

Performs outlier detection techniques using data matrix provided by user. The user can choose the algorithm and the outliers and non-outliers will be outputted to their respective matrix.

```
#include <outlierdetection.h>
```

Public Member Functions

- **OutlierDetection** ()
  Constructor.

- **OutlierDetection** (const **MIOList**<> &IOList)
  Overloaded Constructor.

- **OutlierDetection** (const **InputMatrix** &data)
  Overloaded Constructor.

- **~OutlierDetection** ()
  Used in the creation of the object.

- bool **create** (const **MIOList**<> &IOList)
  Returns true if object created.

- bool **create** (**InputMatrix** &data)
  Function that calculates entropy.

- double **entropy** (int frequency, unsigned int qty)
  Chooses correct algorith depending option.

- double **removeOutliers** (int targetK, int option)
  Implements Local Search Algorithm described by He.

- bool **localSearchAlgorithm** (int targetK)
  This function calculates locates a number of outliers using LSA equivalent number of outliers to that chosen by user. The located outliers are stored as '1' in matrix of flags.

- bool **altLSA** (int targetK)
  Implements Greedy Algorithm described by He.

- bool **greedyAlgorithm** (int targetK)

Implements New Algorithm.

- bool **frequencyAlgorithm** (int targetK)
  Accessor functions.

- **InputMatrix** & **getData** ()
  Returns matrix of all data.

- void **setData** (const **MIOList**<> &IOList)
  Returns true if object created.

- void **setData** (**InputMatrix** &data)
  Returns true if object created.

- unsigned int **getM** ()
  Returns number of columns (attributes).

- unsigned int **getN** ()
  Returns number of rows (vectors/points).

- int * **getFlags** ()
  Returns pointer to flags array storing true for outlier and false for non-outlier.

- double **getTotalEntropy** ()
  Returns total entropy of current set of outliers and non-outliers or whole data depending if an outlier detection algorithm has been run.

Detailed Description

Performs outlier detection techniques using data matrix provided by user. The user can choose the algorithm and the outliers and non-outliers will be outputted to their respective matrix.

Definition at line 100 of file outlierdetection.h.

Member Function Documentation

bool OutlierDetection::create (const MIOList<> & *IOList*)

Returns true if object created.

Parameters:

IOList Input/output list from matlab.

Definition at line 165 of file outlierdetection.cpp.

References setData().

Referenced by OutlierDetection().

double OutlierDetection::entropy (int *frequency*, unsigned int *qty*)

Chooses correct algorith depending option.

Parameters:

frequency Frequency of a value.
qty Total number of values.

Returns:

Returns value of calculated entropy.

Definition at line 309 of file outlierdetection.cpp.

Referenced by greedyAlgorithm(), and localSearchAlgorithm().double OutlierDetection::removeOutliers (int *targetK*, int *option*)

Implements Local Search Algorithm described by He.

Parameters:

targetK Target number of outliers to be detected.
option Value of algorithm wishes to execute.

Returns:

Returns true if algorithm successfully executed.

Definition at line 331 of file outlierdetection.cpp.

References frequencyAlgorithm(), greedyAlgorithm(), and localSearchAlgorithm().

bool OutlierDetection::localSearchAlgorithm (int *targetK*)

This function calculates locates a number of outliers using LSA equivalent number of outliers to that chosen by user. The located outliers are stored as '1' in matrix of flags.

Parameters:

targetK Target number of outliers to be detected.

Returns:

Returns true if algorithm correctly executed.

Definition at line 375 of file outlierdetection.cpp.

References entropy().

Referenced by removeOutliers().

bool OutlierDetection::greedyAlgorithm (int *targetK*)

Implements New Algorithm.

Parameters:

targetK Target number of outliers to be detected.

Returns:

Returns true if algorithm correctly executed.

Definition at line 565 of file outlierdetection.cpp.

References entropy().

Referenced by removeOutliers().

void OutlierDetection::setData (const MIOList<> & *IOList*)

Returns true if object created.

Parameters:

IOList Input/output list from matlab.

Definition at line 203 of file outlierdetection.cpp.

References MIOList< t_uNumInputs, t_uDesiredNumOutputs >::Input(), and MRefMatrix< TYPE, Complexity >::PointTo().

Referenced by create().

void OutlierDetection::setData (InputMatrix & *data*)

Returns true if object created.

Parameters:

IOList Input/output list from matlab.

Definition at line 235 of file outlierdetection.cpp.

References MRefMatrix< TYPE, Complexity >::PointTo().

Generated using *Doxygen*.

The documentation for this class was generated from the following files:

- M:/Documents/Research/Outlier Research/Code/OutlierDetection/outlierdetection.h
- M:/Documents/Research/Outlier Research/Code/OutlierDetection/outlierdetection.cpp