Unsupervised Anomaly Detection Using an Optimized K-Nearest Neighbors Algorithm

Michael J. Prerau, Eleazar Eskin

Abstract

Unsupervised anomaly detection has great utility within the context of network intrusion detection system. Such a system can work without the need for massive sets of pre-labelled training data and has the added versatility of being free of the overspecialization that comes with systems tailored for specific sets of attacks. Thus, with a system that seeks only to define and categorize normalcy, there is the potential to detect new types of network attacks without any prior knowledge of their existence. This paper discusses the creation of such a system that uses a k-nearest neighbors algorithm to detect anomalies in network connections, as well as the optimization necessary to make the algorithm feasible for a real-world system.

1 Unsupervised Anomaly Detection

In the *Unsupervised Anomaly Detection* (UAD) problem, we are given a large data set where most of the elements are normal, and there are intrusions buried within the data set. Unsupervised anomaly detection algorithms have the major advantage of being able to process unlabeled data and detect intrusions that otherwise could not be detected. In addition, these types of algorithms can semi-automate the manual inspection of data in forensic analysis by helping analysts focus on the suspicious elements of the data.

UAD algorithms make two assumptions about the data which motivate the general approach. The first assumption is that the number of normal instances vastly outnumbers the number of anomalies. The second assumption is that the anomalies themselves are qualitatively different from the normal instances. The basic idea is that since the anomalies are both different from normal and are rare, they will appear as outliers in the data which can be detected. An example of an intrusion that an unsupervised algorithm will have a difficulty detecting is a *syn-flood* DoS attack. The reason is that often under such an attack there are so many instances of the intrusion that it occurs in a similar number to normal instances. Thus, UAD algorithms may not label these instances as an attack because the region of the feature space where they occur may be as dense as the normal regions of the feature space.

UAD algorithms are limited to being able to detect attacks only when the assumptions hold over that data, which is not always the case. For example, these algorithms will not be able to detect the malicious intent of someone who is authorized to use the network and who uses it in a seemingly legitimate way. The reason is that this intrusion is not qualitatively different from normal instances of the user. In the framework, these instances would be mapped very close to each other in the feature space and the intrusion would be undetectable.

2 A Geometric Framework for Unsupervised Anomaly Detection

The key to the framework used in the system is mapping the records from the audit stream to a feature space. The feature space is a vector space typically of high dimension. Inside this feature space, we assume that some probability distribution generated the data. We wish to label the elements that are in low density regions of the probability distribution as anomalies are labelled as anomalies. However, in general the probability distribution is unknown. Instead points that are in sparse regions of the feature space. For each point, we examine the point's location within the feature space and determine whether or not it lies in a sparse region of the feature space.

2.1 Feature Spaces

Data is collected from some audit stream of the system. Without loss of generality, this data is split into data elements $x_1, ..., x_l$. The space of all possible data elements is defined as the *input (instance) space* X. Exactly what the input space is depends on the type of data that is being analyzed. The input space can be the space of all possible network connection records, event logs, system call traces, etc.

The elements of the input space are mapped to points in a *feature space* Y. In this framework a feature space is typically a real vector space of some high dimension d, $\Re^d$, or more generally a Hilbert space. The main requirement for the system in the feature space is that we can define a dot product between elements of the feature space.

A *feature map* is defined to be a function that takes as input an element in the input space and maps it to a point in the feature space. In general, we use ϕ to define a feature map and we get

$$\phi : X \to Y \quad . \tag{1}$$

The term *image* of a data element x is used to denote the point in the feature space $\phi(x)$.

Since the feature space is a Hilbert space, for any points y_1 and y_2 their dot product $< y_1, y_2 >$ is defined. Any time there is a dot product, it can also be defined as norm on the space as well as a distance.

The norm of a point y in the feature space $||y||$ is simply the square root of the dot product of the point with itself, $||y|| = \sqrt{< y, y >}$. Using this and the fact that a dot product is a symmetric bilinear form we can define the distance between two elements of the feature space y_1 and y_2 with

$$\begin{aligned} ||y_1 - y_2|| &= \sqrt{< y_1 - y_2, y_1 - y_2 >} \\ &= \sqrt{< y_1, y_1 > -2 < y_1, y_2 > + < y_2, y_2 >} \quad . \end{aligned}$$

Using this framework, the feature map can be used to define relations between elements of the input space. Given two elements in the input space x_1 and x_2, we can use the feature map to define a distance between the two elements as the distance between their corresponding images in the feature space. We can define the distance function d_ϕ as

$$\begin{aligned} d_\phi(x_1, x_2) &= ||\phi(x_1) - \phi(x_2)|| \qquad (2) \\ &= \sqrt{< \phi(x_1), \phi(x_1) > -2 < \phi(x_1), \phi(x_2) > + < \phi(x_2), \phi(x_2) >} \quad . \end{aligned}$$

For notational convenience, the subscript is often dropped from d_ϕ.

If the feature space is $\Re^d$, this distance corresponds to standard Euclidean distance in that space.

3 Normalization

When dealing with unsupervised learning for very large datasets with high dimensionality, there is little practical feasibility in performing learning algorithms so as to determine weight values for the attribute vectors. Therefore, it is necessary to cut one's losses by performing normalization on the data, which will reduce unwanted artificial weighting based on the ranges of the features. This should alleviate the problem of biasing the learning for a particular attribute with a naturally larger magnitude than the other attributes.

For the datasets utilized in these experiments, we encountered three forms of attribute types: continuous, discrete or categorical, and binary. The goal was to affect different methods of specialized normalization for each attribute type such that the aggregate effect produced a sense of homogeneity for the entire vector.

3.1 Continuous

The continuous data was normalized by replacing each attribute value with its distance to the mean of all the values for that attribute in the instance space. In order to do this, the mean and standard deviation vectors must be calculated:

$$mean[j] = \frac{1}{n} \sum_{i=1}^{n} instance[j] \qquad (3)$$

$$standard[j] = \sqrt{\left(\frac{1}{n-1}\right)\sum_{i=1}^{n}(instance[j] - mean[j])^2} \tag{4}$$

From this, the new instances can be calculated by dividing the difference of the instances with the mean vector by the standard deviation vector:

$$newinstance[j] = \frac{instance[j] - mean[j]}{standard[j]} \tag{5}$$

This results in rendering all continuous attributes comparable to each other in terms of their deviation from the norm, which is exactly what we are seeking in anomaly detection.

3.2 Discrete

For discrete or categorical data, two possible means of normalization were considered. The first method was based on the thought that categorical data can be conceptualized as a set of equidistant points, thus creating a spacial representation that equally favors each category. Therefore, one can create an n dimensional binary vector for a discrete attribute with n possible values. Each element in the vector would represent a unique discrete value and would have a value of either zero or a constant.

This works well in many cases. For the KDD-cup data set, however, since we were dealing with attributes with many values, this ended up tripling the size of the instance vectors and became unfeasible in respect to loading in and working with the entire 4.9 million instances in the set.

Thus, a comparable alternative had to be sought. It was found that it was possible to achieve similar results not by modifying the data, but by modifying the distance metric employed by the system. The system still employed a Euclidean distance metric, however for the discrete attributes it would give a distance of zero if the values differed, or it would give a constant, which was inversely proportional to the number of possible values if they were the same. This resulted in a fairly good method of normalization which was highly superior in space and memory usage to the vector method.

3.3 Binary

For the binary data, it is impossible to use the standard deviation metric as the result was often an extremely large number of standard deviations from the mean if there were many zero values and a few one values. Therefore, the binary values were left alone and incorporated into distance metric as is.

3.4 Scaling

At one point there was an attempt to bring further cohesiveness to the data by scaling all values between zero and one. This removed all sense of artificial weighting. The down side to such a scaling was that there was a great loss of

precision that occurred with some attributes. An attribute with a very anomalous instance would end up squashing all the normal values within an extremely small range, thus losing often important distinctions between instances. For this reason, the algorithms were applied to the data set without scaling.

4 K-Nearest Neighbor Algorithm

The K-Nearest neighbors determines whether or not a point lies in a sparse region of the feature space by computing the sum of the distances to the k-nearest neighbors of the point. We refer to this quantity as the kNN score for a point.

Intuitively, the points in dense regions will have many points near them and will have a small kNN score. If the size of k exceeds the frequency of any given attack type in the data set and the images of the attack elements are far from the images of the normal elements, then the kNN score is useful for detecting these attacks.

5 Optimization

Though a k-nearest neighbor value will give yield an excellent sense of how closely a new instance fits in with the rest of the data, it is extremely costly to calculate, with a complexity of $O(n^2)$. This problem is accentuated with with the complex data used for intrusion detection, as a large amount of data-points are required for a solid cross-section of the data. Thus, it is necessary to find computational shortcuts that will allow us to perform anomaly detection with greater celerity. This is especially vital in intrusion detection systems because the damage caused by an attack may often be relative to the time in which it takes for it to be discovered.

In this system, *clustering* is used as a means of breaking down the search space into smaller subsets so as to remove the necessity of checking every data point. This is a variation of agglomerate-neighbor algorithms, however it only uses agglomeration(clustering) as a tool to reduce the time of finding the k neighbors, not as the metric of calculation itself. It should also be noted that clustering itself can also be an affective means of UAD.

Kulling, which eliminates data in linear time from cluster information, is another form of optimization on the k-nearest neighbor algorithm. Kulling exploits the thresholding that happens after the values are computed and invaluable in the application of this algorithm to massive datasets.

5.1 Cluster Optimization

Since we are interested in only the k-nearest points to a given point, we can significantly speed up the algorithm by using a technique similar to canopy clustering. Canopy clustering is used as a means of breaking down the space into smaller subsets so as to remove the necessity of checking every data point. We

use the clusters as a tool to reduce the time of finding the k-nearest neighbors. Clusters (Fig. 1)are defined as hyperspheres in the instance space which contain data points from the training set. This way, a new instance can be quickly tested against a small number of similarly located instance sets before the actual point to point calculation begins. This is done in a way that makes it possible to eliminate clusters without looking at their contents, thus drastically reducing the calculation time.

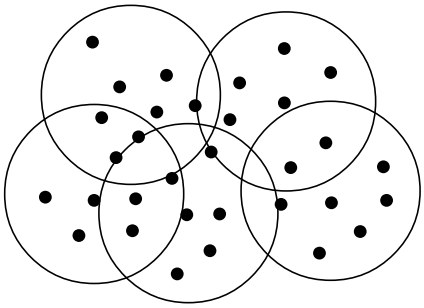

Figure 1: Clusters

We first cluster the data using the fixed-width clustering algorithm of the previous section with a variation where we place each element into only one cluster. Once the data is clustered with width w, we can compute the k-nearest neighbors for a given point x by taking advantage of the following properties.

We denote as $c(x)$ the point which is the center of the cluster that contains a given point x. For a cluster c and a point x we use the notation $d(x, c)$ to denote the distance between the point and the cluster center. For any two points x_1 and x_2, if the points are in the same cluster

$$d_\phi(x_1, x_2) \leq 2w \tag{6}$$

and in all cases

$$d_\phi(x_1, x_2) \leq d_\phi(x_1, c(x_2)) + w \tag{7}$$

$$d_\phi(x_1, x_2) \geq d_\phi(x_1, c(x_2)) - w \quad . \tag{8}$$

The algorithm uses these three inequalities to determine the k-nearest neighbors of a point x.

Let C be a set of clusters. Initially C contains all of the clusters in the data. At any step in the algorithm, we have a set of points which are potentially among the k-nearest neighbor points. We denote this set P. We also have a set of points that are in fact among the k-nearest neighbor points. We denote this set kNN. Initially kNN and P are empty. We precompute the distance from x to each cluster. For the cluster with center closest to x, we remove it from C and add all of its points to P. We refer to this operation as "opening"

the cluster. The key to the algorithm is that we can obtain a lower bound the distance from all points in the clusters in set C using equation (7). We define

$$d_{\min} = \min_{c \in C} d(x, c) - w \quad . \tag{9}$$

The algorithm performs the following. For each point in $x_i \in P$, we compute $d(x, x_i)$. If $d(x, x_i) < d_{\min}$, we can guarantee that x_i is closer point to x then all of the points in the clusters in C. In this case, we remove x_i from P and add it to kNN. If we can not guarantee this for any element of P (including the case that if P is empty), then we "open" the closest cluster by adding all of its points to P and remove that cluster from C. Notice that when we remove the cluster from C, $d_{\min}$ will increase. Once kNN has k elements, we terminate.

Most of the computation is spent checking the distance between points in D to the cluster centers. This is significantly more efficient than computing the pairwise distances between all points.

The choice of width w does not affect the k-NN score, but instead only affects the efficiency of computing the score. Intuitively, we want to choose a w that splits the data into reasonably sized clusters.

5.1.1 Clustering Algorithm

The clusters are created in linear time with the following algorithm:

1. Set $C = \phi$
2. $\forall\ x\ \epsilon\ D$:
   (a) If $\exists\ c_i\ \epsilon\ \mathbf{C}$ s.t. $[d_i < \mathbf{W}]$, add $\mathbf{x}$ to c_i as the center
   (b) Else if $\mathbf{C} = \phi$ or $\neg\exists\ c_i\ \epsilon\ \mathbf{C}$ s.t. $[d_i < \mathbf{W}]$, create a new cluster c_{new} and add x to it as the center

Where:

- $\mathbf{D}$ = The training set
- $\mathbf{x}$ = The given point for which to calculate the k nearest neighbors
- $\mathbf{C}$ = Set of clusters
- $\mathbf{W}$ = The cluster radius
- $\mathbf{D}_i$ = The distance from $\mathbf{x}$ to the center of $\mathbf{C}_i$

First an empty cluster set $\mathbf{C}$ is created. Then each point in the training set $\mathbf{D}$ is looked at in succession. If a $\mathbf{x}$ point falls within the radius of a cluster $\mathbf{W}$, it is added to the cluster. If the cluster set is empty or if the point doesn't fall into any preexisting cluster within $\mathbf{C}$, a new cluster is created with $\mathbf{x}$ as the centroid and add it to this set. In this method, the entire training set can be clustered in a single pass.

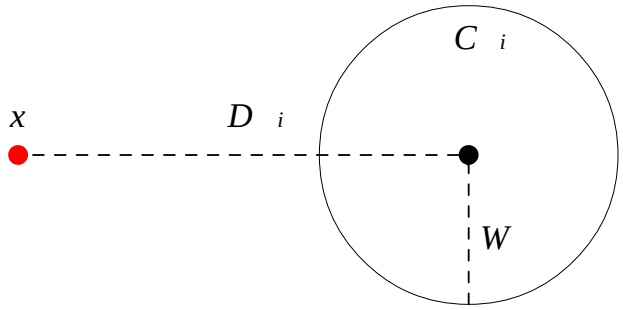

Figure 2: A Cluster

5.1.2 The Cluster-Optimized K-Nearest Neighbor Algorithm in Greater Detail

The pseudocode for the algorithm is as follows:

1. For $\mathbf{x}$, calculate d_i for each $c_i \epsilon \mathbf{C}$
2. Sort $\mathbf{C}$ by d_i
3. While $|\mathbf{kNN}| < \mathbf{k}$
   (a) Calculate distance to all points in P_0 and put them in $\mathbf{P_c}$
   (b) If $[d_1 - \mathbf{W} < \mathbf{I}]$, set $\mathbf{I} = d_1 - \mathbf{W}$
   (c) Else, set $\mathbf{I} = d_0 + \mathbf{W}$
   (d) For each $p_i \epsilon \mathbf{P_c}$ where $distance(\mathbf{x}, p_i) < \mathbf{I}$
      i. If $|\mathbf{kNN}| < \mathbf{k}$
         A. Move p_i from $\mathbf{P_c}$ to $\mathbf{kNN}$
      ii. Else, if $|\mathbf{kNN}| = \mathbf{k}$ and $distance(x, p_i) < kNN_{max}$
         A. Move p_i from $\mathbf{P_c}$ to $\mathbf{kNN}$ replacing kNN_{max}
   (e) Remove c_0 from $\mathbf{C}$

Where:

- $\mathbf{k}$ = Number of nearest neighbors needed
- $\mathbf{x}$ = The given point for which to calculate the $\mathbf{k}$ nearest neighbors
- $\mathbf{C}$ = Set of clusters
- $\mathbf{W}$ = The cluster radius
- $\mathbf{I}$ = Closest point interval
- $\mathbf{kNN}$ = Set of k-nearest neighbors

- $\mathbf{P_c}$ = Set of candidate points
- A cluster $c \in \mathbf{C}$ is made up of the pair (P, d) where:
  - P = The set of points p within the cluster ($p \in P$)
  - d = The distance from $\mathbf{x}$ to the center of c
- kNN_{max} = Largest element in **kNN**

To find the k-nearest neighbors, the distance from a given point $\mathbf{x}$ to every cluster center must first be calculated. Based on these distances, the cluster set $\mathbf{C}$ is sorted. The cluster in $\mathbf{C}$ with the smallest distance from $\mathbf{x}$ is called c_0, and the next closest is called c_1, etc. Next, the distance from $\mathbf{x}$ to all the points in P_0 is calculated, after which those points are moved to the list of candidate points, $\mathbf{P_c}$.

The rest of this algorithm is concerned with the creation of an interval $\mathbf{I}$ and its modification in terms of cluster boundaries. The modification is done with the goal that $\mathbf{I}$ should include the minimum number of points needed to calculate the k-nearest neighbors, given the current cluster configuration.

When the clusters are initially sorted, a partial ordering is created. The reason that this is only a partial ordering is that there can be overlap between two clusters. Since the goal of this system is to guarantee that only the points that *must* be checked are checked, it is necessary to ensure when a point is added to **kNN** based on $\mathbf{I}$, it is impossible that a point outside that interval is closer than it to $\mathbf{x}$. Thus, it is vital to initially exclude the parts of a cluster that are overlapped by another cluster, as that section could contain points from the overlapping cluster that are closer to $\mathbf{x}$ than those in the cluster with the closer center.

When the interval is first created, c_1 is checked to see if it overlaps c_0. If there is no overlap, that is if the distance to the farthest point in the closest cluster ($d_0 + \mathbf{W}$) is less than the closest point of the next closest cluster ($d_1 - \mathbf{W}$), then all points are all necessary candidates and the interval is set to $d_0 + \mathbf{W}$. If there does exist an overlapping cluster, the interval is set to $d_i - \mathbf{W}$, the point of intersection closest to $\mathbf{x}$, so that a given point within this new interval is guaranteed to be closer than any point in the overlapping cluster.

Once the interval for this iteration has been determined, all points in $\mathbf{P_c}$ that are within $\mathbf{I}$ can be moved to **kNN**. If **kNN** is full, then if the distance of a given point is less than that of the largest element in **kNN**, kNN_{max} is replaced by that point. We now remove c_0 from $\mathbf{C}$, making c_1 the new c_0, c_2 the new c_1, etc. If **kNN** has $\mathbf{k}$ elements in it, the process can halt. If **kNN** has less than $\mathbf{k}$ elements, the process begins again with the new c_0 as the closest cluster.

5.2 Kulling

In order to determine whether an instance is an attack or a normal connection, its kNN score is compared to a threshold. If it is greater than the threshold

then it is a considered an attack, if it is less than the threshold then it is considered normal. It turns out that it is possible to utilize the threshold to quickly elminate a large portion of the data in linear time.

If the entire dataset is clustered with a $\mathbf{W}$ of $\frac{\theta}{2k}$, where θ is the threshold value and k is the number of nearest neighbors that will be summed for the kNN, then any cluster that has more than k elements inside it can automatically be disgarded. The reason for this is that assuming the worst case scenario, where the instance being checked is on the opposite end of the cluster as all of the other cluster elements, if the $\mathbf{W} = \frac{\theta}{2k}$ and there more than k elements in the cluster, its kNN value will be $k * \frac{\theta}{k} \leq \theta$. Therefore, in any cluster with a diameter of $\frac{\theta}{k}$ that has more than k elements in it, all of its elements will have a kNN value of less than the threshold (Fig. 3). As a result, the entire cluster can be removed because all the elements within it are automatically classified normal connections. This entire process can be done in linear time, as clustering is linear and the counting and removal of clusters can be done in one pass.

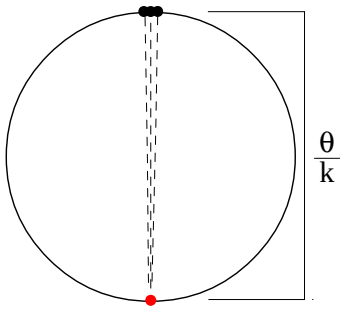

Figure 3: Kulling Threshold

Kulling has worked excellently in practice because in the KDD-Cup dataset, for example, all the normal connections are very similar to each other and grouped within just a few clusters. This algorithm consistantly removes 95% or more of the datapoints when a threshold is used that gives favorable detection results. Kulling therefore saves an enormous amount of time and allows datasets to be used for k-nearest neighbor that would have otherwise been unfeasible.

6 Experiments

Experiments were performed over two different types of data: network connection records, and system call traces.

6.1 Performance measures

To evaluate the system, two major indicators of performance were of interest: the *detection rate* and the *false positive* rate. The detection rate is defined as the number of intrusion instances detected by the system divided by the total

number of intrusion instances present in the test set. The false positive rate is defined as the total number of normal instances that were (incorrectly) classified as intrusions divided by the total number of normal instances. These are good indicators of performance, since they measure what percentage of intrusions the system is able to detect and how many incorrect classifications it makes in the process. These values are calculated over the labelled data to measure performance.

The trade-off between the false positive and detection rates is inherently present in many machine learning methods. By comparing these quantities against each other, there can be an evaluation of the performance invariant of the bias in the distribution of labels in the data. This is especially important in intrusion detection problems because the normal data outnumbers the intrusion data by a factor of $100 : 1$. The classical accuracy measure is misleading because a system that always classifies all data as normal would have a 99% accuracy.

ROC (Receiver Operating Characteristic) curves are plotted depicting the relationship between false positive and detection rates for one fixed training/test set combination. ROC curves are a way of visualizing the trade-offs between detection and false positive rates.

6.2 Data Set Descriptions

The network connection records we used was the KDD Cup 1999 Data, which contained a wide variety of intrusions simulated in a military network environment. It consisted of approximately 4,900,000 data instances, each of which is a vector of extracted feature values from a connection record obtained from the raw network data gathered during the simulated intrusions. A connection is a sequence of TCP packets to and from some IP addresses. The TCP packers were assembled into connection records using the Bro program modified for use with MADAM/ID. Each connection was labelled as either normal or as exactly one specific kind of attack. All labels are assumed to be correct.

The simulated attacks fell in one of the following four categories : DOS - Denial of Service (e.g. a syn flood), R2L - Unauthorized access from a remote machine (e.g. password guessing), U2R - unauthorized access to superuser or root functions (e.g. a buffer overflow attack), and Probing - surveillance and other probing for vulnerabilities (e.g. port scanning). There were a total of 24 attack types.

The extracted features included the basic features of an individual TCP connection such as its duration, protocol type, number of bytes transferred, and the flag indicating the normal or error status of the connection. Other features of an individual connection were obtained using some domain knowledge, and included the number of file creation operations, number of failed login attempts, whether root shell was obtained, and others. Finally, there were a number of features computed using a two-second time window. These included - the number of connections to the same host as the current connection within the past two seconds, percent of connections that have "SYN" and "REJ" errors, and the number of connections to the same service as the current connection

within the past two seconds. In total, there are 41 features, with most of them taking on continuous values.

The KDD data set was obtained by simulating a large number of different types of attacks, with normal activity in the background. The goal was to produce a good training set for learning methods that use labeled data. As a result, the proportion of attack instances to normal ones in the KDD training data set is very large as compared to data that one would expect to observe in practice.

Unsupervised anomaly detection algorithms are sensitive to the ratio of intrusions in the data set. If the number of intrusions is too high, each intrusion will not show up as anomalous. In order to make the data set more realistic we filtered many of the attacks so that the resulting data set consisted of 1 to 1.5% attack and 98.5 to 99% normal instances.

The system call data is from the BSM (Basic Security Module) data portion of the 1999 DARPA Intrusion Detection Evaluation data created by MIT Lincoln Labs. The data consists of 5 weeks of BSM data of all processes run on a Solaris machine. We examined three weeks of traces of the programs which were attacked during that time. The programs we examined were *eject*, and *ps*.

Each of the attacks that occurred correspond to one or more process traces. An attack can correspond to multiple process traces because a malicious process can spawn other processes. The attack is considered detected if one of the processes that correspond to the attack is detected.

6.3 Experimental Setup

For each of the data sets, the data was split into two portions. One portion, the training set, was used to set parameters values for the algorithm and the second, the test set, was used for evaluation.

The parameters were set based on the training set. Then for each of the methods over each of the data sets, the detection threshold was varied and at each threshold computed the detection rate and false positive rate. For each algorithm over each data set we obtained a ROC curve.

For the KDD cup data, the value of k was set to $10,000$ of the data set. For the *eject* data set, $k = 2$ and for the *ps* data set, $k = 15$. The k is adjusted to the overall size of the data.

6.4 Experimental Results

the approach to unsupervised anomaly detection performed very well over both types of data.

In the case of the system call data, the each of the algorithms performed perfectly. What this means is that at a certain threshold, there was at least one process trace from each of the attacks identified as being malicious without any false positives. An explanation for these results can be obtained by looking at exactly what the feature space is encoding. Each system call trace is mapped to a feature space using the spectrum kernel that contains a coordinate for

each possible sub-sequence. Process traces that contain many of the same sub-sequences of system calls are closer together than process traces that contain fewer sub-sequences of system calls.

The results are not surprising when we consider previous approaches to anomaly detection over system call traces. In the original work in this area, sub-sequences of system calls were the basis of the detection algorithm. The supervised anomaly detection algorithm presented in recorded a set of sub-sequences that occurred in a normal training set and then detected anomalies in a test set by counting the number of unknown sub-sequences that occur within a window of 20 consecutive sub-sequences. If the number of previously unseen sub-sequences is above a threshold, then the method would determine that the process corresponds to an intrusion. The results of their experiments suggest that there a many sub-sequences that occur in malicious processes that do not occur in normal processes. This explains why in the feature space defined by the spectrum kernel, the intrusion processes are significantly far away from the normal processes. Also, in the experiments, the normal processes clumped together in the feature space. This is why the intrusion processes were easily detected as outliers.

For the network connections, the data is not nearly as regular as the system call traces. From the experiments, it was found that there were some types of attacks that the system was able to detect well and other types of attacks that it were not able to detect. This is reasonable because some of the attacks using the feature space were in the same region as normal data. Although the detection rates are lower than what is typically obtained for either misuse or supervised anomaly detection, the problem of unsupervised anomaly detection is significantly harder because we do not have access to the labels or have a guaranteed clean training set.

Detection Rate	False Positive Rate
91%	8%
23%	6%
11%	4%
5%	2%

The ROC curve (Fig. 4) shows the performance of the kNN algorithm over the KDD Cup 1999 data. Figure 1 shows the Detection Rate and False Positive Rate for some selected points from the ROC curve.

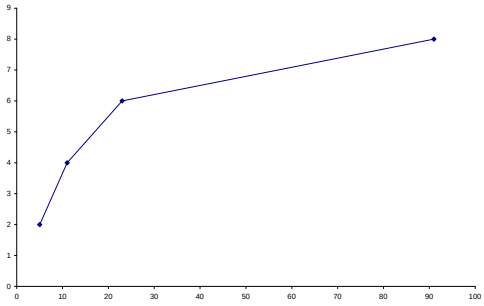

Figure 4: ROC Curve