IJST, Transactions of Electrical Engineering, Vol. 36, No. E2, pp 109-129
Printed in The Islamic Republic of Iran, 2012
© Shiraz University

A COST SENSITIVE LEARNING METHOD TO TUNE THE NEAREST NEIGHBOUR FOR INTRUSION DETECTION[*]

M. R. MOOSAVI,[**] M. ZOLGHADRI JAHROMI, S. GHODRATNAMA, M. TAHERI AND M. H. SADREDDINI
Dept. of Computer Science and Engineering, School of Electrical and Computer Engineering, Shiraz University, Shiraz, I. R. of Iran
Email: mmoosavi@cse.shirazu.ac.ir

Abstract– In this paper, a novel cost-sensitive learning algorithm is proposed to improve the performance of the nearest neighbor for intrusion detection. The goal of the learning algorithm is to minimize the total cost in leave-one-out classification of the given training set. This is important since intrusion detection is a problem in which the costs of different misclassifications are not the same. To optimize the nearest neighbor for intrusion detection, the distance function is defined in a parametric form. The free parameters of the distance function (i.e., the weights of features and instances) are adjusted by our proposed feature-weighting and instance-weighting algorithms. The proposed feature-weighting algorithm can be viewed as general purpose wrapper approach for feature weighting. The instance-weighting algorithm is designed to remove noisy and redundant training instances from the training set. This, in turn improves the speed and performance of the nearest neighbor in the generalization phase, which is quite important in real-time applications such as intrusion detection. Using the KDD99 dataset, we show that the scheme is quite effective in designing a cost-sensitive nearest neighbor for intrusion detection.

Keywords– Distance metric learning, feature-weighting, instance-weighting, intrusion detection systems, nearest neighbor, KDD99 dataset

1. INTRODUCTION

Nowadays, security plays a strategic role in modern computer network systems. Intrusion detection systems are effective security tools that look for known or potential threats in network traffic and/or audit data recorded by hosts [1]. Basically, an IDS analyzes user's behavior using the information from various sources such as audit trail, system table, and network usage data [2, 3]. The problem of intrusion detection has been studied extensively in computer security, and has received a lot of attention in the fields of machine learning and data mining [4-8].

In its basic form, the nearest neighbor (NN) rule, which is a non-parametric classification method, has been used for intrusion detection. The basic rationale for the NN rule is both simple and intuitive: patterns close in the feature space are likely to belong to the same class. Therefore, its performance relies on the locally constant class conditional probability [9]. A variety of distance measures has been used in NN classification and various methods have been proposed to adapt the distance measure to the application at hand.

The NN classifier has many advantages over other methods: It leads to a very simple approximation of the Bayes classifier. Therefore it is nearly optimal in the large sample limit. On the other hand, it can learn from a small set of examples. The state-of-the-art NN classifier uses local information, which can

[*]Received by the editors June 11, 2012; Accepted December 29, 2012.
[**]Corresponding author

yield highly adaptive behavior. Moreover, it performs well in real-world problems compared to many complex and costly implemented methods. As a lazy learner, new examples could be incrementally added as they become available. The classifier is interpretable and gives competitive performance with other methods such as decision trees or neural nets [10]. However, the NN classification method suffers from the following major problems:

1. The NN algorithm is very sensitive to the features used by the classification algorithm. Irrelevant features degrade the performance of the algorithm as they contribute equally in the distance function. Many feature selection/weighting algorithms are proposed in the literature that attempt to solve this problem by controlling the contribution of each feature in the distance function. These methods can be categorized into two main groups: filter and wrapper approaches [11-18].

2. The NN algorithm is very sensitive to the quality of the training samples. Noisy (i.e., mislabeled) training examples can cause misclassification of many test instances. To solve this problem, many algorithms have been proposed in the past research [19- 24].

3. The basic NN algorithm memorizes all of the training samples for use in the generalization phase. To classify a query pattern, its distance from all training examples should be calculated. This makes the algorithm slow when the number of training examples is large. Many algorithms proposed in the literature attempt to solve this problem by selecting a small subset of training data [25-28].

In the following, we briefly address some recent techniques proposed in the literature to tackle the problems mentioned above. These techniques define the distance function in a certain form to incorporate different kinds of information. For this purpose, the distance function is usually defined in a parametric form. The procedure of adapting the distance function (i.e., tuning the parameters), based on a set of training data, is usually called distance metric learning [29].

In [30], a scheme is proposed to learn weighted metrics to improve the generalization accuracy of the NN algorithm. The weights (i.e., the parameters of the distance measure) may be specified for each class, feature, or individual instance. To specify the parameters of the distance function, a learning algorithm is proposed that uses gradient descent to minimize a performance index that is an approximation to leave-one-out (LOO) classification error-rate.

In [19], an adaptive K-NN classification algorithm is proposed that is based on the concept of statistical confidence from hypotheses testing. This method takes into account the effective influence size of each training example and the statistical confidence with which the label of each training example can be trusted. In [20], a locally adaptive distance measure is used that is based on assigning a weight to each training instance. The parameters of the distance measure (i.e., the weights of the training instances) are specified by a simple heuristic. This scheme is shown to be effective in improving the performance of the basic NN, but it suffers from sensitivity to noise.

In [24], an adaptive NN classifier is proposed for noisy environments. This instance weighting algorithm attempts to consider class separability by minimizing entropy in the deciding area of each instance. In [18] an algorithm is proposed to tackle the first and second problem mentioned above. This was achieved by assigning a weight to each feature and each training instance. The weight parameters are tuned by means of a hill-climbing search method. In [21], an algorithm is proposed to tackle the second and third problems. The algorithm is designed to select a small subset of weighted prototypes from training data. To specify the weights of the prototypes, a learning algorithm is proposed that attempts to directly minimize the LOO classification error-rate of the given training set.

In this paper, based on our mentioned contributions [18, 21], a general method is proposed to simultaneously tackle all of the three problems of the NN algorithm mentioned above. The algorithm is designed to efficiently improve the performance of the NN algorithm in cost-sensitive problems such as intrusion detection.

Most learning algorithms used to tune a classifier attempt to minimize the error-rate of the classifier on the training data. These algorithms implicitly assume that class-to-class misclassification costs are the same. In many real-world applications such as medical diagnosis of a certain disease, this assumption is not true. It is obvious that misclassifying a patient as healthy (i.e., false negative) is much more costly than misclassifying a normal case as patient (i.e., false positive).

Intrusion detection is another typical problem in which the cost of different misclassifications are different [31, 32]. Obviously, failing to detect an intruder is more costly than misclassifying a normal user as intruder. Indeed, if a normal user's logon fails due to false-positive prediction, the imposed cost is not more than a further try by the user. On the other hand, granting the permission to an intruder may result in the breakdown of the security system. Therefore, cost-sensitive learning algorithms are applied to minimize the total cost of misclassifications by taking different misclassification costs into account [33, 34].

For an *m*-class cost-sensitive problem, with *n* training data, assume that an *m* by *m* cost matrix, *C*, is available. Each element $C_{i,j}$ of this matrix represents the cost of predicting an element of class *i* in class *j*. The performance of a classification algorithm can be represented by an *m* by *m* matrix denoted as Confusion Matrix (*CM*). Each element $CM_{i,j}$ of this matrix represents the number of elements (i.e., test instances) of class *i* predicted as class *j*. Given the cost and confusion matrices, the *CPE* measure is calculated as:

$$CPE = \frac{1}{n}\sum_{i=1}^{m}\sum_{j=1}^{m} CM_{i,j} \times C_{i,j} \tag{1}$$

where *n* is the number of examples used to test the classifier.

The 1998 DARPA Intrusion Detection Evaluation Program, managed by the MIT Lincoln Labs, prepared a standard dataset for the intrusion detection learning task [35]. The prepared dataset was originally used in the KDD Cup 99, International Knowledge Discovery and Data Mining Competition. The KDD Cup 99 dataset is the most widely used benchmark [8, 36, 37].

Various intrusion detection methods have been proposed in the literature. There are two major trends for intrusion detection: Signature based and anomaly based. The signature based, also called Misuse detection, treat intrusion detection as a classification problem. The KDD Cup 99 was a cost based misuse detection contest. In this trend, the attacks are classified based on perfectly learned patterns of abnormal usage or signatures [38]. The approach is reliable, economical and has low false-positive error rate [39].

Although classification algorithms have been extensively used for the intrusion classification problem [40], there are few reports about misuse detection methods that perform better than the winner of the KDD 99 contest. Moreover, many of the proposed methods require high computational or memory demands.

Researchers usually use the following techniques to alleviate the complexity of the problem:

1. Ensemble learning: combining different techniques in hybrid systems. Some of the proposed methods use different classification techniques to accurately classify different attack types [41-42] and some others try to combine the advantages of signature based and anomaly detection systems [43]. The best reported results to date are from hybrid systems of decision trees. In fact, the contest winner fused 50x10 C5 decision trees using cost-sensitive bagged boosting algorithm [44] and is yet the state of the art method for the KDD 99 dataset.

2. Evolutionary classifiers: Many fuzzy rule based classifiers are proposed for intrusion detection. While usual rule based techniques fail in the case of the KDD 99, major rule based IDS use genetic algorithms for introducing new rules into the population or tuning rule weights [42, 45]. Some of the well-known and state-of-the art evolutionary systems are too time consuming or use a huge rule base.

3. Sampling or data generation: Many of the intrusion detection systems reported their results on a subset of the prepared dataset (training and test datasets) or used a sampling method to extract two subsets for train and test from the original KDD dataset. Many others used datasets generated by their simulation or gathered data from their own network.

Our proposed method is obviously notable since it is a basic algorithm which improves the CPE without using these popular techniques. In fact, improving the CPE on the KDD test set is an arduous task. The basic methods could be further improved using the anomaly based paradigm [46], the online (incremental) learning problem [47] or data preprocessing techniques [37].

The proposed algorithm, in this paper, attempts to minimize the overall cost of misclassification in LOO classification of the given training set using NN rule. This is achieved by specifying the weights of features and training instances.

We propose two algorithms for this purpose: feature-weighting and instance-weighting. Both of these algorithms attempt to improve the performance of the NN algorithm in cost-sensitive problems by directly minimizing the *CPE* in LOO classification of the given training set.

The rest of the paper is organized as follows. In Section 2, the NN algorithm with weighted features and instances is presented to introduce the notation. In Section 3, we introduce the last-runner problem. Our feature-weighting algorithm makes use of the efficient solution that we provide for this problem. In Section 4, the details of our feature and instance weighting algorithms are presented. The results of our experiments on KDD99 intrusion detection dataset are presented in Section 5. Finally, conclusions are remarked in Section 6.

2. NEAREST NEIGHBOR CLASSIFICATION WITH WEIGHTED FEATURES AND INSTANCES

For an *m*-class problem, assume that a set of training examples of the form $T = \{(X_i, l_i) \mid i = 1, ..., n\}$ is given, where, $X_i = [x_{i_1}, x_{i_2}, ..., x_{i_d}]^T$ is a *d*-dimensional vector of attributes and $l_i \in \{1, 2, ..., m\}$ specifies the class label of X_i. To identify the NN of a query pattern, a variety of distance functions has been proposed in the literature [48]. The Euclidean distance function has often been used for this purpose. The Euclidean distance, d_E, between two patterns X_i and X_j can be expressed as:

$$d_E(X_i, X_j) = \sum_{k=1}^{d} (x_{ik} - x_{jk})^2 \tag{2}$$

It must be noticed that the square root of the distance function is removed in (2) without affecting the functionality of the NN algorithm. In this paper we use the following distance function to measure the distance between a query pattern Q, and a stored instance, X_i:

$$d(Q, X_i) = (1/u_i) \sum_{k=1}^{d} w_k (q_k - x_{ik})^2 \tag{3}$$

where, w_k is used to denote the weight assigned to the *k*-th feature and u_i denotes the weight assigned to the training instance X_i.

The weight assigned to an instance controls its influence in the feature space for classifying test instances. An instance X_p having a zero weight (i.e., u_p=0) appears to be far away from all instances, such that, it is not the NN of any query pattern. In this way, noisy and redundant training patterns can be effectively removed from the training set by setting their weights to zero. Also, in (3), the contribution of each feature in the distance function can be controlled by its weight. This way, irrelevant features can be easily removed from the feature space by setting their weights to zero.

The contribution of this paper is in the algorithms that we propose to learn the parameters of the distance function $W = \{w_k \,|\, 1 \le k \le d\}$ and $U = \{u_i \,|\, 1 \le i \le n\}$.

3. THE LAST-RUNNER PROBLEM

In this section, the last-runner problem is introduced. Our feature weighting algorithm presented in the next section makes use of the efficient solution that we propose for this problem.

a) Problem statement

Assume that n runners $R = \{r_i \,|\, i = 1, 2, .., n\}$ participate in a competition by running in a pre-specified path away from the point *O*. The initial distance of runners (i.e., at the time $t = 0$) from point *O* (denoted as offsets) are given as $\{a_i \,|\, i = 1, 2, .., n\}$ where, it is assumed that $a_1 < a_2 < ... < a_n$. We also know that all runners run at constant velocities $\{v_i \,|\, i = 1, 2, .., n\}$. An example of the last-runner problem with three runners is shown in Fig. 1.

As time (*t*) goes from zero to infinity, we are interested in tracking the last-runner (i.e., the runner that is behind all others). Our aim is to design an efficient algorithm to identify the last-runner as a function of time. The algorithm should output a list *S* expressed as: $S = [(r_1, s_1), (r_j, s_2), (r_k, s_3), ...]$. The first element (r_1, s_1) denotes that r_1 is the last-runner at the start of the competition (i.e., at $t = s_1 = 0$). The next two elements (r_j, s_2) and (r_k, s_3) denote that the r_j and r_k are the next last-runners occupying the last position at $t = s_2$ and $t = s_3$, respectively. In other words, r_1 and r_j are the last-runners in the time intervals $s_1 < t < s_2$ and $s_2 < t < s_3$, respectively.

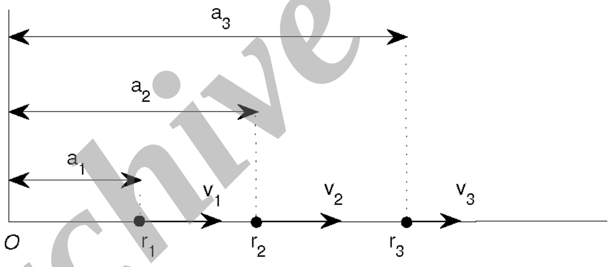

Fig. 1. An example of the last-runner problem

b) Solution

The distance of each runner r_i from point O as a function of time t can be expressed as:

$$d_i(t) = a_i + v_i t \tag{4}$$

In general, the time $t_{i,j}$ in which r_i passes r_j (assuming $i < j$) can be expressed as:

$$t_{i,j} = \frac{a_j - a_i}{v_i - v_j} \tag{5}$$

Note that in the above equation, if $v_i \le v_j$, the runner r_i cannot pass r_j. Therefore r_j can never occupy the last position. In this case, Eq. (5) gives a negative value for $t_{i,j}$. On the other hand, if $v_i > v_j$, r_i will definitely overtake r_j at a positive time $t_{i,j}$. It must be noted that r_i will not be the last-runner again ($\forall t > t_{i,j}$). In other words, each runner can only occupy the last position once during the competition.

We denote the times that the last-runner is changed as critical-times. The algorithm should produce the list of runners and their associated critical times. This list is denoted as $S=[(r_i,s_i)\,|\,r_i\in R, s_i\le s_j \forall j>i]$, where s_i is the time that r_i occupies that last position (i.e., critical-time for r_i).

Example 1. Consider the example shown in Fig. 2 with four runners $[r_i=(a_i,v_i)\,|\,i=1,2,3,4]$. The initial distance from point O and also the speed of runners are specified as: [(8,24),(15,25),(24,16),(28,12)]. In this example, r_2 and r_3 pass r_4 (at the time $t=1.0$) before r_1 passes r_4 (at the time $t=1.67$). The runner r_1 is the last runner for $t<1.67$, when it passes r_4. Hence, r_4 is the last-runner thereafter. In Fig. 2, runners r_2 and r_3 are displayed with dashed lines to denote that they never occupy the last position. For this example, the algorithm should produce the list $S=[(r_1,0),(r_4,1.67)]$ as output.

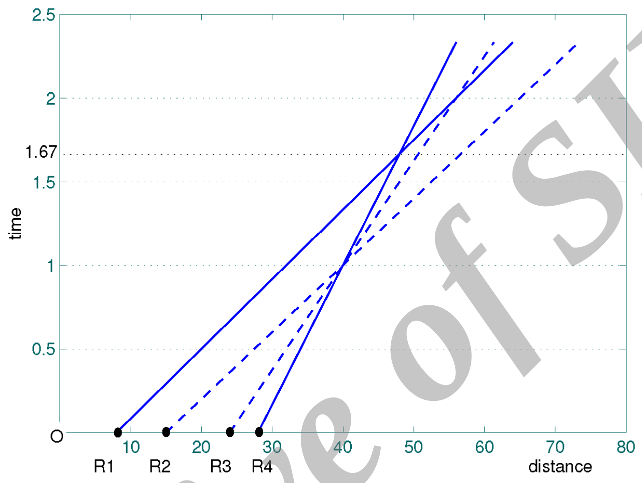

Fig. 2. A last-runner problem example

A simple solution for the last-runner problem can be expressed as follows. Initially (i.e., at $t=0$), r_1 is the last-runner. To find the next last-runner (and its associated critical-time) the time at which r_1 passes each of the other runners (i.e. $t_{1,i}\,\forall i>1$) can be easily calculated using Eq. (5). As mentioned before, a negative value of $t_{1,j}$ indicates that r_1 cannot pass r_j (since r_1 is slower than r_j). Ignoring the negative values, the time at which the last-runner is changed can be easily determined by finding the minimum of these times. Assuming that $t_{1,k}$ is the minimum of these times, the next last-runner is identified as r_k, and $t_{1,k}$ is the time that r_k becomes the last-runner (r_1 is the last-runner for time interval $0<t<t_{1,k}$). To find the next last-runner (for $t>t_{1,k}$) and its associated critical-time, we should repeat the above-mentioned procedure by finding the times at which r_k passes each of the other runners.

It is obvious that the worst-case time complexity of the above simple solution is $O(n^2)$, where n is the number of runners. In the following, we provide an efficient algorithm to solve this problem in $\theta(n)$. This algorithm consists of two steps: removing the fast-runners and removing the lucky-runners. The aim of these two steps is to identify (and remove from the list) those runners that can never be the last-runner. After these steps, each runner that is left in the reduced list will definitely be the last-runner at some time interval in the competition.

1. Removing the fast-runners: It is very easy to identify a group of runners (denoted as *fast-runners*) that can never be the last-runner during the competition. The aim of this step is to reduce the list of runners by removing the fast-runners from the list. A runner r_k is a fast-runner if:

$$v_k \ge \min_{i<k}(v_i) \tag{6}$$

Equation (6) states that a runner r_k can never be the last-runner if one of the runners that is located behind r_k (at the start of the competition) has a speed less than r_k. The list *L* that excludes all fast-runners can be easily constructed by one pass over all runners in the order that they appear in the list *R*. It must be noted that the runners in the constructed list *L* are now simultaneously sorted in descending order of their velocities ($v_i > v_j, \forall i < j$) and ascending order of their offsets ($a_i < a_j, \forall i < j$).

As the list of runners *R* is initially sorted in ascending order of the offsets, a simple way to construct the list L is to remove all runners from R that violate the condition: $v_1 > v_2 > ... > v_n$. Note that this is a one-pass algorithm with complexity of $\theta(n)$.

Example 2. As an example, consider the list of runners given below.

$$R = \left[(8,24),(15,25),(24,16),(28,12),(30,14),(31,13),(32,8),(36,6),(38,20),(41,4),(42,3)\right]$$

The list L that excludes all fast-runners is constructed by removing r_2, r_5, r_6 and r_9.

$$L = \left[(8,24),(24,16),(28,12),(32,8),(36,6),(41,4),(42,3)\right]$$

2. Removing the lucky-runners: A *lucky-runner* is referred to a runner that is not a fast-runner but can never be a last-runner (i.e., does not satisfy the condition given in Eq. (6) but still cannot occupy the last position). The aim of this phase is to identify and remove the lucky-runners from the list *L* (constructed in the previous section).

It is obvious that the necessary condition for a runner r_k to be the last-runner is that all runners located behind it at the start of the competition (i.e., $\{r_1, r_2, .., r_{k-1}\}$) pass r_k. Now, r_k, is said to be a lucky-runner if it can pass another runner before all the runners $\{r_1, r_2, .., r_{k-1}\}$ have passed it.

Example 3. To illustrate this situation, consider an example with three runners: $R = [r_1, r_2, r_3]$ where $a_1 < a_2 < a_3$ and $v_1 > v_2 > v_3$ (i.e. there is no fast-runner in the list R). Assuming that $t_{2,3} \le t_{1,2}$, the runner r_2 is a lucky-runner (and can never be the last-runner). This runner is lucky since by the time that r_1 overtakes r_2, the runner r_2 has already passed r_3. Therefore r_1 and r_3 will be the last-runners for $t < t_{1,3}$ and $t > t_{1,3}$, respectively. In this example, the only critical-time is $t_{1,3}$ (i.e., r_3 becomes the last-runner at $t = t_{1,3}$ and remains in the last position until the end of the competition).

Assume that the list L is constructed from R by excluding all fast-runners (as explained in the previous section). In general, we can state that for every three consecutive runners in L (i.e., r_{i-1}, r_i, r_{i+1}), r_i is a lucky-runner (and can be removed) if the condition $t_{i,i+1} \le t_{i-1,i}$ is satisfied. The algorithm for removing fast-runners makes use of this condition to identify and remove all lucky runners.

A general procedure for identifying the lucky-runners is a sequential processing of the list *L* and can be stated as follows. Starting with $i = 1$, we calculate $t_{i,i+1}$ and continue until the condition $t_{i,i+1} \le t_{i-1,i}$ is satisfied for some value of i (i.e., $i = k$). This means that r_k is a lucky-runner and must be removed. As the removal of r_k makes the three runners $r_{k-2}, r_{k-1}, r_{k+1}$ consecutive in the list, we need to check if r_{k-1} is a lucky-runner or not. To do this, we need to calculate $t_{k-1,k+1}$ and check to see if $t_{k-1,k+1} \le t_{k-2,k-1}$. If this condition is satisfied, r_{k-1} is a lucky-runner and must be removed. After removing r_{k-1}, the process continues by checking to see if r_{k-2} is a lucky-runner or not. This backward traversal continues until the examined runner is not a lucky-runner.

Example 4. To illustrate this procedure, identifying the lucky-runners of list L in example 2 is presented here. $t_{2,3} \le t_{1,2} \Rightarrow r_2\, is\, lucky; t_{3,4} \le t_{1,3} \Rightarrow r_3\, is\, lucky; t_{6,7} \le t_{5,6} \Rightarrow r_6\, is\, lucky; t_{4,5} \le t_{5,7} \Rightarrow r_5\, is\, lucky;$
$\Rightarrow S = \{(r_1, 0), (r_4, 1.5), (r_7, 2.0)\}$

After removing the lucky-runners, each runner in the resultant list will definitely be the last-runner at some stage during the competition. If we re-index the list of retained runners as $[r_i \,|\, 1 \le i \le m, m \le n]$, the inequality $t_{i-1,i} < t_{i,i+1}$ is valid for all of the runners in this list. In fact, the runners in this list are also sorted in ascending order of times $t_{i,i+1}$ for $1 \le i \le m-1$.

Initially, r_1 is the last-runner. The first change in ranking occurs when r_1 passes r_2 at $t = t_{1,2}$. This makes r_2 the last-runner. Therefore $t_{1,2}$ is the first critical-time. After this time the runner r_1 is in the front of r_2 forever and can be ignored. Similarly, the next change in ranking occurs at $t_{2,3}$ when r_2 passes r_3. In fact, the time interval that each runner occupies the last position is readily available, since we have already computed the values of critical-times $t_{i,i+1}$ (for $1 \le i \le m-1$) during the process of identifying the lucky-runners.

Figure 3 presents our proposed algorithm for finding the last-runners and corresponding critical times. In the algorithm of Fig. 3, $r_i.a$ and $r_i.v$ denote the offset and velocity of the runner r_i. Also, indices *first* and *last* are used to refer to the first and the last element of a list, respectively.

```
1:  function CRITICALTIMECOMPUTATION(R) returns S    ▷ Input: the list of runners: $R = [r_i = (a_i,v_i)|i = 1, 2,..n]$ where $a_i \le a_{i+1}$ (and $v_i < v_{i+1}$ if $a_i = a_{i+1}$ ). Output: the list of runners and their associated critical-times $S = [(r_i,s_i)|r_i \in R, \forall j > i, s_i \le s_j]$
2:      $L = [r_1]$
3:      for each $r \in R$ do
4:          if $r.v < L_{last}.v$ then
5:              add $r$ to $L$
6:      $S = [(r_1, 0)]$
7:      remove $L_{first}$ form $L$                              ▷ remove $r_1$
8:      for each $r \in L$ do
9:          $t = \frac{r.a - S_{last}.a}{S_{last}.v - r.v}$
10:         while $t \le S_{last}.s$ do
11:             remove $S_{last}$ from $S$
12:             $t = \frac{r.a - S_{last}.a}{S_{last}.v - r.v}$
13:         add(r , t) to S
        return S
```

Fig. 3. The algorithm for calculating the list of last-runners and their associated critical-time

To analyze the time complexity of the algorithm, suppose the list *R* includes *n* runners. In general, the procedure of identifying the lucky-runners (in a list *L* containing *n* runners) consists of two types of computation. One is the forward traversal of computing $t_{i,i+1}$ for $i = 1,...,n-1$ and the other is the backward traversal of computing the new times when a lucky-runner is spotted. The backward traversal consists of computing *p* new times (where $p < n$) and each time removing the visited runner from the list. The time taken by the list construction and pruning (lines 3 to 6) is $\theta(n)$. In the line 7, at most *n* elements exist in the list *L*, and consequently, lines 9 and 13 are iterated $O(n)$ times. There are at most *n-2* lucky runners (i.e. all of the runners excluding the first and the last one), hence *n-2* passes through the whole loop (lines 10 to 12). Therefore, in the worst-case using an amortized analysis [49], line 11 never executes more than line 13. It means that the second loop started at line 8 is of $\theta(n)$.

4. LEARNING THE DISTANCE FUNCTION PARAMETERS

In this section, we introduce our proposed method for specifying the parameters of the distance function expressed in (3). The parameters are specified in two steps. In the first step, the weights of features (i.e., $W = \{w_k \mid k = 1, 2, .., d\}$) are determined assuming that the weights of instances are fixed (i.e. $U = \{u_i = 1 \mid 1 \le i \le n\}$). In the second step, the weights of instances (i.e., $U = \{u_i \mid i = 1, 2, .., n\}$) are specified, assuming that the weights of features are given and fixed. The overall scheme consists of two algorithms: feature-weighing and instance-weighting. Both of these algorithms attempt to minimize the average cost in LOO classification of the given training set.

a) The proposed feature weighting algorithm

Our aim in this section is to propose an algorithm that attempts to minimize the *CPE* in LOO classification of the given training set by specifying the weights of features $\{w_f \mid f = 1, 2, ..., d\}$.

In its basic form, the proposed algorithm is a greedy search method. The algorithm starts with an initial solution to the problem (i.e., $\{w_k = 1 \mid k = 1, 2, ..., d\}$) and attempts to improve the solution by adjusting the weight of one feature in each iteration. The basic component of the learning scheme is an algorithm that provides the answer to the following question:

What is the optimal weight of feature k (i.e., w_k) assuming that the weights of all other features are given and fixed?

The weight w_k is optimal in the sense that it results in minimum *CPE* in LOO classification of the training data. In this way, the overall learning algorithm consists of visiting each feature in turn to adjust its weight. It must be noted that the weight specified for a feature is optimal if the weights of other features remain fixed. That is why the second pass and subsequent passes over the features can reduce the *CPE*. In experiments, we simply stop the search after a pre-specified number of passes over all features.

In the following, we explain how the proposed algorithm specifies the weight w_f of feature *f* (assuming that the weights of other features are fixed), as shown in Fig. 4.

As we increase w_f from 0 to ∞, the predicted class (and classification cost) of each training instance X_t may change several times in LOO test. This is due to the fact that the distance function used to find the nearest neighbor of X_t is a function of w_f. Obviously, the classification cost of X_t depends on its nearest neighbor in LOO test. We are interested in finding those values of w_f that change the nearest neighbor of X_t. As we increase w_f from 0 to ∞, all training instances move away from X_t. The situation is analogous to the last-runner problem discussed in Section 3. Training instances (analogous to runners) are moving away from X_t (point *O*) as w_f (analogous to time) is increased from 0 to ∞. Each training instance X_k moves away from X_t at constant velocity v_k specified by:

$$v_k = (1/u_k)(x_{kf} - x_{tf})^2 = d_f(X_t, X_k) \tag{7}$$

where x_{kf} and x_{tf} represent the values of feature *f* for instances X_k and X_t, respectively. The term $d_f(X_t, X_k)$ is used to denote the distance between the instances X_t and X_k in feature *f*. The initial distance of each training instance X_k from X_t (denoted as a_k) is the distance between X_k and X_t when w_f is set to zero:

$$a_k = (1/u_k) \sum_{i(i \ne f)} w_i \left(x_{ti} - x_{ki}\right)^2 = d_{\bar{f}}(X_t, X_k) \tag{8}$$

where $d_{\bar{f}}(X_t, X_k)$ denotes the distance between X_t and X_k ignoring feature *f* (i.e. their distance while $w_f = 0$).

In LOO classification of each training instance X_t, we can now use the solution that we provided for the last-runner problem to find the critical-values of w_f that cause a change in NN of X_t. Using these critical-values, we can easily construct the nearest neighbors list (NN-list) of X_t as w_f is increased from 0 to ∞. Having this list, for any value of w_f, we can calculate the cost of classifying X_t in LOO test.

As an example, assume that the NN-list of X_t is given by $S = [(X_k, c_1 = 0), (X_p, c_2), (X_m, c_3)]$). The first element of each pair identifies the NN and the second element gives the value of w_f. From the list, we can easily conclude that X_k, X_p and X_m are the nearest neighbors of X_t for intervals $0 < w_f < c_2$, $c_2 < w_f < c_3$ and $w_f > c_3$, respectively. As we know the true classes of X_t, X_k, X_p and X_m, we can use the cost matrix to calculate the cost of classifying X_t for any value of w_f.

To find the best weight of the feature under consideration (i.e., *f*) , the NN-list of all training instances are merged into a global list (denoted as GS_f). This list is then sorted in ascending order of the critical-values of w_f.

For any specific value of w_f, we know the NN of each training instance. Each critical-value in this list indicates a change in the NN of one of the instances in the training set. Assuming that GS_f has n'' critical-values (i.e., $w_{f1}, w_{f2}, .., w_{fn''}$), we need to check $n''+1$ thresholds to find the best value of w_f. The thresholds tested are $w_f = 0, (w_{f1} + \varepsilon), (w_{f2} + \varepsilon), .., (w_{fn''} + \varepsilon)$, where ε is a very small positive number. The best value of w_f is simply the threshold resulting in minimum cost.

```
1:  function FEATUREWEIGHTING($T$,$U$) returns $W$   ▷ Input: Training set $T = \{X_i | 1 \le i \le n\}$ , instance weights $U = \{u_i = 1 | 1 \le i \le n\}$   ▷ Output: Feature Weights $W = \{w_f | i = 1, 2,..,d\}$
2:    for iter=1 to no. of iterations do
3:    for each feature $f \in featurespace$ do
4:         let $w_f = 0$                                ▷ i.e. remove the feature $f$ from the feature space
5:         $GS_f$ = []
6:         for each instance $X_t \in T$ do
7:              construct list $L_t$ by sorting all training instances in ascending order of their distance from $X_t$
8:              $list_t$ = []
9:              for each instance $X_k \in L_t$ do
10:                  add ($d_f(X_t,X_k)$,$d_{\bar{f}}(X_t,X_k)$) to $list_t$
11:
12:             $S_t$ = CriticalTimeComputation($list_t$)
13:             add critical-times of $S_t$ to the global list $GS_f$
14:        sort $GS_f$ in ascending order of critical-times
15:        for each different threshold $th$ in $GS_f$ do          ▷ $th = 0, GS_f(i) + \varepsilon$
16:             find the overall CPE (assuming $w_f = th$)          ▷ using equation (1)
17:        $w_f = th$ with minimum overall CPE
    return $W$
```

Fig. 4. The feature-weighting algorithm

To analyze the time complexity, suppose the feature-weighting algorithm includes *i* iterations to assign weight to *d* features. The overall complexity of this algorithm is $O(idn^2 lg(n))$, described in the following. In each iteration for each feature, the loop on X_t starting at the line 6 iterates *n* times. This loop has a sorting function of $O(nlg(n))$ in line 7 and a complexity of $O(n)$ for lines 9-13. Thus, the overall complexity of the loop at line 6 is $O(n^2 lg(n))$. Then, line 14 sorts a list with maximum length n^2. This sorting procedure has a complexity of $O(n^2 lg(n))$. Finally the loop at line 15, which is a sequential pass over the list *GS* can be ignored in comparison with $O(n^2 lg(n))$.

In data mining applications with very large training set, using the feature-weighting algorithm presented in this section may not be feasible due to time constraints. One simple solution for this problem is to select a subset T_r of the full training set *T*. Using T_r, instead of *T*, in the feature-weighting algorithm of Fig. 4, the overall complexity of the algorithm is $O(idn'^2 lg(n'))$, where n' is the number of selected instances.

b) The instance weighting algorithm

The WDNN algorithm presented in [21] attempts to minimize the LOO error-rate of NN classifier by specifying the weight of each training instance. Given the class-to-class misclassification costs, this algorithm can be easily modified to minimize the cost in LOO classification of the training set. However, in applications with very large training set (such as KDD99), it is not feasible to specify the weight of all training instances due to time requirement. One simple solution to this problem is to select a small subset, T_r, from the full training set T as candidate prototypes and use the algorithm to specify the weights of instances in T_r rather than T. In other words, the instance-weighting algorithm proposed in this section attempts to minimize the total cost in LOO classification of the full training set by specifying the weights of the instances in $T_r = \{(X_i, l_i) \mid 1 \le i \le n', X_i \in T, n' \le n\}$. Obviously, in applications with small training set, the full training set can be selected as the prototype set (i.e, $T_r = T$). Therefore, in this paper, the terms instance-weighting and prototype-weighting are used interchangeably. In prototype-weighting, the weight of instances not present in T_r are assumed to be zero. It must be noted that after the application of instance-weighting algorithm, the number of prototypes left (i.e., having non-zero weight) is usually much less than n' since the algorithm sets the weight of redundant candidate prototypes to zero.

The algorithm that constructs the prototype set T_r from the full training set T is presented in section 5.2. Here, we assume that a prototype set $T_r = \{(X_i, l_i) \mid 1 \le i \le n', X_i \in T, n' \le n\}$ is available. Further, we assume that the weights of the features have been specified using the algorithm of Fig. 4. In the following, we present the prototype-weighting algorithm, which attempts to minimize the *CPE* in LOO classification of the full training data by specifying the weights of prototypes $U = \{u_i \mid i = 1, 2, ..., n'\}$.

For a problem with n' prototypes and known cost matrix, the algorithm starts with an initial solution to the problem (i.e., $\{u_k = 1 \mid k = 1, 2, ..., n'\}$) and attempts to improve the solution by adjusting the weight of one prototype in each iteration. Basically, this is a greedy optimization method and the *CPE* never increases during this process. The overall learning algorithm consists of passing over the entire prototype set for a pre-specified number of iterations or until no improvement is observed over previous iteration.

The prototype-weighting algorithm, presented in Fig. 5, starts by finding the *associates* of each prototype. The *associates* of a prototype X_l (denoted as $A(X_l)$) are those training instances that have X_l as their nearest prototype in LOO test. The algorithm keeps the associate list of each prototype in memory and updates them as the weights of prototypes change during the execution of the algorithm.

The prototype-weighting algorithm specifies the weight u_p of a typical prototype X_p, as follows. In the first step, X_p is removed from the instance space by setting its weight to zero (i.e., $u_p = 0$). This forces each of its associates (for example X_i) to use a new nearest neighbor (i.e., X_k). Then, the algorithm finds and stores the predicted class of each associate of X_p, which is used along with the true class of X_p and each of its associates to calculate its effect in classification cost. The best value of u_p is specified in such a way that classification cost of instances in the association list is minimum. To do this, in the next step, the Score s of each training instance, X_i in the association list of X_p is calculated with the following definition of Score:

$$s(X_i) = \frac{u_k d_\omega(X_p, X_i)}{d_\omega(X_k, X_i)} \tag{9}$$

where $d_\omega(X_p, X_i) = \sum_{k=1}^{d} w_k (x_{pk} - x_{ik})^2$.

```
1: function PROTOTYPEWEIGHTING($T,T_r,W$) returns $U$        ▷ Input: Training set $T = \{X_i|1 \le i \le n\}$, Prototype
   set (a subset of Training set) $T_r = \{X_i|1 \le i \le n'\}$, feature weights $W = \{w_i|1 \le i \le d\}$
                                                               ▷ Output: Prototype Weights $U = \{u_i|i = 1, 2,..,n'\}$
2:  for each training Instance $X_k \in T$ do
3:      find $X_l$, that is the nearest prototype of $X_k$
4:      add $X_k$ to $A(X_l)$
5:  for iter=1 to no. of iterations do
6:      for each prototype $X_p \in T_r$ do
7:          let $u_p = 0$                                      ▷ i.e., remove the instance from the feature space
8:          for each instance $X_i \in A(X_p)$ do
9:              find $X_k$ that is the nearest prototype of $X_i$
10:             $s(X_i) = u_k \frac{d_\omega(X_p, X_i)}{d_\omega(X_k, X_i)}$
11:         sort all instances in $A(X_p)$ in ascending order of their score, s
12:         for each different threshold $th = 0, th = s(p) + \epsilon, p = 1,..,n''$ do
                                                               ▷ $\epsilon$ is a very small positive number
13:             calculate the CPE (assuming $u_p = th$)        ▷using equation (1)
14:         $u_p = best\_th$                                   ▷$best\_th$ is the $th$ resulting in minimum CPE
15:         for each instance $X_i \in A(X_p)$ do             ▷updating the associate list of $X_p$
16:             if $s(X_i) > best\_th$ then
17:                 find $X_k$, the nearest neighbor of $X_i$
18:                 remove $X_i$ from $A(X_p)$
19:                 add $X_i$ to $A(X_k)$
    return $U$
```

Fig. 5. The prototype-weighting algorithm

The score of an instance defined in (9) has an interesting property. For an instance having score $s(X_i) = a$, if we choose $u_p > a$, the instance X_i will stay in associate list of X_p. Otherwise (i.e., if $u_p < a$), it moves to the associate list of another instance (that we have already found and stored in the previous step). Using this, the predicted classes of any training instance $X_i \in A(X_p)$ for the two cases of $u_p < s(X_i)$ and $u_p > s(X_i)$ are known. As we know the true class of X_i, we can easily calculate the cost of classifying X_i for any value of u_p.

For instance, X_i (in the associate list) with $s(X_i) = a$, assume that L is the true class of X_i (i.e., $L = l_i$), T is the predicted class for $u_p < a$ and P is the predicted class for $u_p > a$ (i.e., $P = l_p$, the class label of X_p). Then, the cost of classifying X_i can be expressed as:

$$Cost(X_t) = \begin{cases} C_{L,T} & if\, u_p < a \\ C_{L,P} & if\, u_p > a \end{cases} \tag{10}$$

where, $C_{I,J}$ is used to represent the cost of classifying an instance of class I in class J, as given in the cost matrix.

Having the relation between a certain value of u_p and the corresponding cost of classifying each associate of X_p, the best value for u_p can be easily found by sorting the associates in ascending order of their scores (i.e., $s(X_1) < s(X_2) < ... < s(X_{n''})$), assuming the list contains n'' instances. Considering any value of u_p between $s(X_j)$ and $s(X_{j+1})$, the first j instances in the list will stay in associate list of X_p and the rest will have a new nearest neighbor. In this way, n'' + 1 different values (i.e., $\{0, s(X_1)+\varepsilon, s(X_2)+\varepsilon, .., s(X_{n''})+\varepsilon\}$) should be examined to find the best value of u_p. After specifying the best weight u_p of X_p, the associate lists should be updated. This is done by moving those associates of X_p whose score are greater than u_p to their new neighbor's list.

The worst-case time complexity of the prototype-weighting algorithm can be expressed as follows. Suppose that the distances between all pairs of the training instances are calculated before execution of the algorithm. Also, let λ_p and *t* denote the number of associates of X_p and the number of iterations, respectively. In each iteration, the loop starting at line 6 repeats n' times. The statements at lines 8 and 12 are executed λ_p times. The sorting procedure of line 11 has a complexity of $O(\lambda_p lg(\lambda_p))$. Updating the associate list of X_p at line 15 has a worst-case time complexity of $O(\lambda_p \times n')$. Therefore, the overall complexity can be expressed as:

$$O(t\sum_{p=1}^{n'}(\lambda_p + \lambda_p \lg(\lambda_p) + \lambda_p + \lambda_p n')) \;\Rightarrow\; O(t\sum_{p=1}^{n'}(\lambda_p \lg(\lambda_p) + \lambda_p n'))$$

$$\Rightarrow\; O(t\sum_{p=1}^{n'}(\lambda_p \lg(n) + \lambda_p n')) \;\Rightarrow\; O(t\left(\lg(n)\sum_{p=1}^{n'}\lambda_p 1 + n'\sum_{p=1}^{n'}\lambda_p\right))$$

As updating the weight of each prototype could change the length of associate lists, the summation $\sum_{p=1}^{n'}\lambda_p$ is greater than *n*. However, as the NN of each instance does not change many times, the summation is linearly related with parameter *n*. Therefore, the time complexity can be simply expressed as $O(tn\lg(n))$.

5. EXPERIMENTS

Most techniques are evaluated based on KDD Cup 1999 intrusion detection dataset [8, 38]. The experiments on KDD99 dataset are reported in this section to evaluate the effectiveness of the proposed feature and prototype weighting algorithms.

a) KDD99 intrusion detection dataset

The KDD99 intrusion detection dataset is based on the 1998 DARPA initiative, which provides designers of intrusion detection systems (IDS) with a benchmark on which to evaluate different methodologies [50, 51, 52]. To build the dataset, a simulation was made including three 'target' machines running various operating systems and services. To simulate network traffic, three additional machines were used to spoof different IP addresses. Finally, a sniffer was used to record all network traffic using the TCP dump format [35, 53]. The data set consists of 4,898,430 connection records. Each record has 41 attributes and a label indicating the status of the records as either normal or a specific attack type. These features have different forms of continuous, discrete, and symbolic, with significantly varying ranges and ability to separate various classes. There are four groups of features: Basic Features, Content Features, Time-based Traffic Features and Host-based Traffic Features. The training set contains various attack types. Each attack falls into one of the following major categories: *Denial of Service (DOS)*, *Remote to Local (R2L)*, *User to Root (U2R)* and *Probe*. More details about the dataset could be found in intrusion detection literature [54, 55, 56]. The cost matrix used to score entries in KDD99 contest is given in Table 1. This cost matrix is used to evaluate the performance of various IDS schemes proposed in the literature [57].

b) Experimental setup

The KDD99 dataset is used as a benchmark to compare different intrusion detection methods [36], but the dataset suffers from some quality problems [58, 59]. Some pre-process procedures, such as data cleaning and prototype selection, could be used to improve the quality of the dataset. Figure 6 shows the block diagram of various steps used in our experiments. The details of each step are presented in the following.

Table 1. The cost matrix used to evaluate the performance of various IDS schemes on the KDD99 dataset

Cost Matrix		Predicted Class				
		Normal	DOS	U2R	R2L	Probe
Actual Class	Normal	0	2	2	2	1
	DOS	2	0	2	2	1
	U2R	3	2	0	2	2
	R2L	4	2	2	0	2
	Probe	1	2	2	2	0

Data cleaning and normalization: In this step, all duplicate training instances were removed from the training set to solve some of the problems concerning the original dataset [58]. Moreover, each categorical feature was replaced by *P* binary features, where *P* is the number of values that the feature can assume.

Data normalization must correct the bias in favour of features having large values and can be regarded as a way of assigning weights to different features. In this way, a normalization method that is suitable for one feature may not be suitable for another. In KDD99 dataset, features 4 and 5 (namely src_bytes and dst_bytes) are spanned over a very large range in the interval [0,1.3 billion]. The value of these features for most data samples is in a small range, while a very few samples have very large values. To normalize these two features, a logarithmic measure was used as follows [60]:

$$x_{if}^{normalized} = log(x_{if} - \min_i(x_{if}) + 1) \Big/ log(\max_i(x_{if}) - \min_i(x_{if}) + 1) \tag{11}$$

where, $\min_i(x_{if})$ is used to denote the minimum value of feature *f*. Other features were simply normalized using their mean and standard deviation:

$$x_{if}^{normalized} = (x_{if} - \underset{j}{mean}(x_{jf})) \Big/ (8 \times \underset{j}{stdev}(x_{jf})) + 0.5 \tag{12}$$

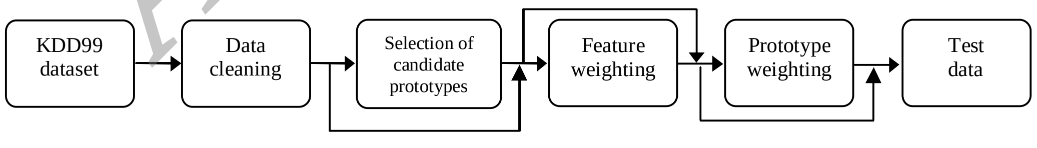

Fig. 6. Block diagram of various steps in the experiments

Selection of candidate prototypes: A simple heuristic was used to select a small candidate prototype subset T_r from the full training set *T*. The prototype selection algorithm starts with $T_r = T$. For each train data X_i in *T*, the algorithm considers each of its *k* nearest neighbors for removal. An instance X_j in the neighborhood of X_i is removed from T_r if both of the following conditions are satisfied: 1) Its distance from X_i is less than a predetermined threshold *r* and 2) Both X_i and X_j have the same class label. It must be noted that this is a one pass algorithm and we used the Euclidean distance measure and *k* = 3000 and *r* = 0.15 in experiments reported in this paper.

c) Experimental results

In experiments reported in this section, duplicate training instances were first removed from the training set. Following this, all features were normalized as described in Section 5.2. In the next step, our prototype selection algorithm was used to select a small subset of the training set. The application of this algorithm selects 3314 instances from training data as prototype subset. Table 2 show the distribution of data in different classes for different data sets used in the experiments. Using this prototype set, our feature-weighting algorithm of Section 4.1 was used to specify the weights of the features. Then, the instance-weighting algorithm of Section 4.2 was used to specify the weights of the prototypes. Using prototype and instance weights, the performance of our method was evaluated by classifying the KDD99 test data.

Table 2. Distribution of data in different classes at various stages of the experiments

Dataset	Normal	DOS	U2R	R2L	Probe	Total
Original train set (10% KDD)	97277	391458	52	1126	4107	494020
Duplicate training instances removed	87831	54572	52	999	2131	145585
Selected prototype subset	2556	372	41	103	242	3314
Test set (Corrected KDD)	60593	229853	70	16347	4166	311029
Full KDD dataset	972780	3883370	52	1126	41102	4898430

To compare the outcome, we first report the results of the KDD99 contest [57]. The results of the basic 1-NN classifier on KDD99 test data is shown in Table 3. Also, Table 4 shows the results obtained by the winner of the KDD99 contest [44], which was discussed in the introduction section. Considering *CPE* as the major evaluation criterion, the basic 1-NN classifier, ranked ninth in the contest, achieves an average cost of 0.2523, while the average cost for the winner of the contest is 0.2331.

Table 3. Results of the basic 1-NN classifier on KDD99 test data

Confusion Matrix	Predicted Class						
Actual Class		Normal	DOS	U2R	R2L	Probe	%Correct
	Normal	60322	57	1	1	212	99.55
	DOS	6144	223633	0	0	76	97.29
	U2R	209	1	8	5	5	3.51
	R2L	15785	1	0	95	308	0.59
	Probe	697	342	0	2	3125	75.01
	CPE=0.2523						

Table 4. Classification results obtained by the winner of KDD99 contest

Confusion Matrix	Predicted Class						
Actual Class		Normal	DOS	U2R	R2L	Probe	%Correct
	Normal	60262	78	4	6	243	99.45
	DOS	5299	223226	0	0	1328	97.12
	U2R	168	0	30	10	20	13.16
	R2L	14527	0	8	1360	294	8.40
	Probe	511	184	0	0	3471	83.32
	CPE=0.2331						

In Table 5, we report the classification results when the selected prototypes are used to classify the KDD99 test data (i.e., nearest prototype classification, without weighting).

Comparing the results of Tables 3 and 5, we observe that reducing the number of training instances did not have a drastic effect on classification results. In fact, we observe a small drop in average cost when using the prototype subset instead of the full training set.

Table 5. Classification of the KDD99 test data using the selected prototype subset

Confusion matrix	Predicted class						
Actual class		Normal	DOS	U2R	R2L	Probe	%Correct
	Normal	60254	70	14	4	251	99.44
	DOS	5857	223929	0	0	67	97.42
	U2R	68	0	29	9	122	12.72
	R2L	15735	0	100	352	2	2.17
	Probe	894	206	0	1	3065	73.57
	CPE=0.2480						

In Table 6, we report the classification results of the nearest prototype classifier after the application of feature-weighting algorithm for 2 iterations. Comparing the results of Tables 5 and 6, we observe that our feature-weighting algorithm could improve the performance by reducing the *CPE* from 0.2480 to 0.2309 (6.9% relative improvement). In fact, the KDD99 test data comes from a distribution that is very different from the training data. Our feature weighting algorithm is expected to reduce the average cost by a larger amount when test data have the same distribution as the training data.

Table 6. Nearest prototype classification results of KDD99 data, after feature-weighting

Confusion matrix	Predicted class												
		LOO classification of training data						Test results					
Actual class	nPts=3314	Normal	DOS	U2R	R2L	Probe	%Correct	Normal	DOS	U2R	R2L	Probe	%Correct
	Normal	87806	0	2	15	8	99.97	60158	86	4	12	333	99.28
	DOS	26	54381	0	0	165	99.65	6023	222780	0	0	1050	96.92
	U2R	0	0	52	0	0	100.00	39	1	45	8	135	19.74
	R2L	31	0	0	968	0	96.90	12743	4	2713	725	4	4.48
	Probe	0	3	0	0	2128	99.86	333	448	0	68	3317	79.62
		CPE=0.0027						CPE=0.2309					

It must be noted that our feature-weighting algorithm reduced the number of features from 41 to 30. This is due to the fact that the algorithm removes redundant/irrelevant features by setting their weights to zero.

Figure 7 shows the value of *CPE* during the first pass of our feature-weighting algorithm. As seen, the algorithm has reduced the *CPE* from 0.0498 to 0.0232 in LOO classification of the prototype set. The *CPE* is monotonically decreasing. This is due to the fact that our algorithm is a greedy optimization method.

In the last step, we used the algorithm of Section 4.2 to specify the weights of selected prototypes. In this step, the weights of features are assumed to be fixed and set to the values specified by the feature-weighting algorithm. Table 7 gives the classification results when the instance weighting algorithm is applied for 3 iterations. Comparing the results of Tables 6 and 7, we observe that the average cost is further reduced from 0.2309 to 0.1967 (14.8% relative improvement).

The number of prototypes is reduced from 3314 to 236. This is because our instance-weighting algorithm removes redundant prototypes (and noisy instances) by setting their weights to zero. The reduction of prototypes can significantly reduce the classification time which is an important issue in on-line intrusion detection systems.

In Table 8, a summary of the results obtained at various stages of our experiments is presented. In this table we report the number of stored instances (i.e. prototypes) for classifying test data, the number of features used and the *CPE*.

Table 7. NN classification results on KDD99 data after the application of feature-weighting and prototype-weighting algorithms

Confusion matrix	Predicted class												
		LOO classification of training data						Test results					
	nPts=236	Normal	DOS	U2R	R2L	Probe	%Correct	Normal	DOS	U2R	R2L	Probe	%Correct
Actual class	Normal	87815	3	1	6	6	99.9818	60217	67	5	20	284	99.3795
	DOS	3	54448	0	0	121	99.7728	5359	224007	0	0	487	97.4566
	U2R	3	0	49	0	0	94.2308	157	0	44	9	18	19.2982
	R2L	12	1	2	984	0	98.4985	8879	0	146	1279	5885	7.9004
	Probe	10	4	1	1	2115	99.2492	243	576	4	1	3342	80.2208
		CPE=0. 0016						CPE=0. 1967					

Table 8. Summary of the NN results obtained on KDD99 at different stages

Stage	no. of prototypes	no. of features	train CPE	test CPE
Using the KDD full training set (duplicate instances removed)	145585	41	0.0018	0.2484
Using the prototype subset	3314	41	0.0039	0.2480
After application of the feature-weighting algorithm	3314	30	0.0027	0.2309
After application of feature & prototype-weighting algorithms	236	30	0.0016	**0.1967**

To evaluate the effectiveness of our algorithms, we compared the results with other basic and major methods in Table 9. In [66] a feature selection method is proposed for LSSVM, which uses a least square cost function and RBF kernel. Although the interpretability of the system is good, its improvement is due to sampling the dataset. Moreover, LSSVM perform binary classification and lead to high computation demand as the number of attack types increases.

Table 9. Comparison of the results with other major methods

Method	CPE	Method	CPE
1-NN	0.2523	Multi-classifier [61]	0.2285
5-NN	0.2459	MOGFIDS [62]	0.2317
C4.5	0.2426	XCSR [63]	0.2660
SVM	0.2474	ESC-IDS [64]	0.1579
PNrule [65]	0.2371	PLSSVM [66]	0.1807
KDD Cup Runner up [67]	0.2356		
KDD Cup Winner [44]	0.2331	The proposed method	0.1967

In [64] a neuro-fuzzy classifier was proposed. Different ANFIS networks are used for different intrusion classes. They have also used subtractive clustering to determine the number of rules and initial locations for membership functions. At last, a genetic algorithm is used to optimize the system. Even though using fuzzy rules has a good comprehensibility, tuning the membership functions and using a complex decision making engine decrease the interpretability of the system in comparison to the size of our prototype set. Moreover, training several ANFIS networks and using genetic algorithm leads to heavy computational challenges. XCSR [63] is also a rule based system which uses GA to generate new rules. Finally, MOGFIDS fuzzy rule-based system is evolved from an agent based evolutionary framework and can act as a genetic feature selection wrapper [62].

Overall, our proposed method could reduce the *CPE* from 0.2523 (using original training set and NN algorithm) to 0.1967. This is a 22% relative improvement, which is quite significant in the KDD99 cost-sensitive classification problem.

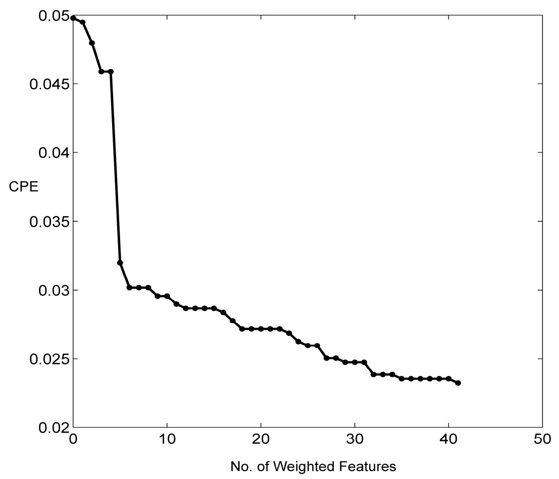

Fig. 7. The CPE on train data during the application of the feature-weighting algorithm (the first pass)

6. CONCLUSION

In this paper, we proposed a method of adapting the nearest neighbor classifier for cost-sensitive problems. For this purpose, the distance function was defined in a parametric form. The free parameters of the distance function (weights of features and instances) are used for tuning the NN classifier for cost sensitive problems. Using the given cost matrix, the proposed feature and instance weighting algorithms attempt to minimize the average cost in leave-one-out classification of the training data.

Using KDD99 intrusion detection dataset, we showed that the scheme is successful in reducing the average cost of classification on previously unseen data. Apart from this, the scheme removes redundant features and instances by setting their weights to zero. In other words, the scheme not only reduces the average cost of classification in comparison with basic NN, but also it can significantly improve the classification time of basic NN by removing redundant features and instances.

REFERENCES

1. Giacinto, G., Roli, F. & Didaci, L. (2003). A modular multiple classifier system for the detection of intrusions in computer networks. *Multiple Classifier Systems*, 2709, pp. 161–161.
2. Bace, R. & Mell, P. (2001). Intrusion detection systems. US Dept. of Commerce, Technology Administration, National Institute of Standards and Technology.
3. Cho, S. B. (2002). Incorporating soft computing techniques into a probabilistic intrusion detection system, Systems, Man, and Cybernetics, Part C: Applications and Reviews, *IEEE Transactions on*, Vol. 32, pp. 154 – 160.
4. Axelsson, S. (2000). Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Dept. of Computer Engineering, Chalmers University of Technology, Sweden.
5. Lane, T. D. (2000). Machine learning techniques for the computer security domain of anomaly detection, Ph.D. thesis, Department of Electrical and Computer Engineering, Purdue University.
6. Saniee Abadeh, M., Habibi, J., Barzegar, Z. & Sergi, M. (2007). A parallel genetic local search algorithm for intrusion detection in computer networks. *Engineering Applications of Artificial Intelligence*, Vol. 20, pp. 1058–1069.
7. Patel, A., Qassim, Q. & Wills, C. (2010). A survey of intrusion detection and prevention systems. *Information Management & Computer Security*, Vol. 18, pp. 277–290.
8. Gogoi, P., Bhattacharyya, D., Borah, B. & Kalita, J. (2011). A survey of outlier detection methods in network

anomaly identification. *The Computer Journal*, Vol. 54, pp. 570–588.

9. Peng, J., Heisterkamp, D. & Dai, H. (2004). Adaptive quasiconformal kernel nearest neighbor classification, Pattern Analysis and Machine Intelligence. *IEEE Transactions on*, Vol. 26, pp. 656–661.
10. Fayyad, U., Piatetsky-Shapiro, G., Smyth, P. & Uthurusamy, R. (1996). Advances in knowledge discovery and data mining. Vol. 15, MIT press Cambridge, MA.
11. Domeniconi, C., Peng, J. & Gunopulos, D. (2002). Locally adaptive metric nearest-neighbor classification, Pattern Analysis and Machine Intelligence. *IEEE Transactions on*, Vol. 24, pp. 1281–1285.
12. Hastie, T. & Tibshirani, R. (2002). Discriminant adaptive nearest neighbor classification, *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, Vol. 18, pp. 607–616.
13. Tahir, M., Bouridane, A. & Kurugollu, F. (2007). Simultaneous feature selection and feature weighting using Hybrid Tabu Search/K-nearest neighbor classifier. Pattern Recognition Letters 28 pp. 438–446.
14. Wettschereck, D., Aha, D. & Mohri, T. (1997). A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms, Artificial Intelligence Review 11, pp. 273–314.
15. Ricci, F. & Avesani, P. (1999). Data compression and local metrics for nearest neighbor classification. Pattern Analysis and Machine Intelligence, IEEE Transactions on 21, pp. 380–384.
16. Short, I. & Fukunaga, R. K. (1981). The optimal distance measure for nearest neighbor classification. Information Theory, IEEE Transactions on 27, pp. 622–627.
17. Kumar, G., Kumar, K. & Sachdeva, M. An empirical comparative analysis of feature reduction methods for intrusion detection. *International Journal of Information and Telecommunication Technology*, Vol. 1, pp. 44–51.
18. Ghodratnama, S., Moosavi, M. R., Taheri, M., Zolghadri Jahromi, M. (2010). A cost sensitive learning algorithm for intrusion detection, in: Electrical Engineering (ICEE). *18th Iranian Conference on, 10778, IEEE, IEEE*, pp. 559–565.
19. Wang, J., Neskovic, P. & Cooper, L. (2006). Neighborhood size selection in the k-nearest-neighbor rule using statistical confidence. *Pattern Recognition*, Vol. 39, pp. 417–423.
20. Wang, J., Neskovic, P. & Cooper, L. (2007). Improving nearest neighbor rule with a simple adaptive distance measure, *Pattern Recognition Letters*, Vol. 28, pp. 207–213.
21. Jahromi, M., Parvinnia, E. & John, R. (2009). A method of learning weighted similarity function to improve the performance of nearest neighbor. *Information sciences*, Vol. 179, pp. 2964–2973.
22. Ferri, F., Albert, J. & Vidal, E. (1999). Considerations about sample-size sensitivity of a family of edited nearest-neighbor rules, Systems, Man, and Cybernetics, Part B: Cybernetics. *IEEE Transactions on 29*, pp. 667–672.
23. Wilson, D. L. (1972). Asymptotic properties of nearest neighbor rules using edited data, Systems, Man and Cybernetics. *IEEE Transactions on 2*, pp. 408–421.
24. Moosavi, M., Javan, F., Jahromi, Z. & Sadreddini, M. (2010). An adaptive nearest neighbor classifier for noisy environments, in: Electrical Engineering (ICEE). *18th Iranian Conference on, IEEE*, pp. 576–580.
25. Jankowski, N. & Grochowski, M. (2004). Comparison of instances seletion algorithms I. Algorithms survey, Artificial Intelligence and Soft Computing-ICAISC 2004 3070, pp. 598–603.
26. Grochowski, M. & Jankowski, N. (2004). Comparison of instance selection algorithms II. Results and comments, Artificial Intelligence and Soft Computing-ICAISC 2004 3070, pp. 580–585.
27. Wilson, D. & Martinez, T. (2000). An integrated instance-based learning algorithm. *Computational Intelligence*, Vol. 16, pp. 1–28.
28. Cano, J., Herrera, F. & Lozano, M. (2003). Using evolutionary algorithms as instance selection for data reduction in kdd: an experimental study. *Evolutionary Computation, IEEE Transactions on 7*, pp. 561–575.
29. Weinberger, K. Q. & Saul, L. K. (2009). Distance metric learning for large margin nearest neighbor classification. *Journal of Machine Learning Research*, Vol. 10, pp. 207–244.
30. Paredes, R., Vidal, E. (2006). Learning weighted metrics to minimize nearest-neighbor classification error. *IEEE*

Transactions on Pattern Analysis and Machine Intelligence, Vol. 28, pp. 1100–1110.

31. Lee, W. & Stolfo, S. (2000). A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security (TISSEC)*, Vol. 3, pp. 227–261.
32. Lee, W., Stolfo, S. & Mok, K. (1999). A data mining framework for building intrusion detection models, in: Security and Privacy, 1999. *Proceedings of the 1999 IEEE Symposium on, IEEE*, pp. 120–132.
33. Ling, C. & Sheng, V. (2008). Cost-sensitive learning and the class imbalanced problem. Encyclopedia of Machine Learning. C. Sammut.
34. Liu, X. & Zhou, Z. (2006). The influence of class imbalance on cost-sensitive learning: An empirical study, in: Data Mining, 2006. ICDM'06. *Sixth International Conference on, IEEE*, pp. 970–974.
35. Stolfo, S., Fan, W., Lee, W., Prodromidis, A. & Chan, P. (2000). Cost-based modeling and evaluation for data mining with application to fraud and intrusion detection: Results from the jam project, in: *Proceedings of the DARPA Information Survivability Conference*, pp. 1–15.
36. Horng, S. J., Su, M. Y., Chen, Y. H., Kao, T. W., Chen, R. J., Lai, J. L. & Perkasa, C. D. (2011). A novel intrusion detection system based on hierarchical clustering and support vector machines. *Expert Systems with Applications*, Vol. 38, pp. 306–313.
37. Davis, J. J. & Clark, A. J. (2011). Data preprocessing for anomaly based network intrusion detection : A review. *Computers & Security*, Vol. 30, pp. 353–375.
38. Chen, Z., Zhang, Y., Chen, Z. & Delis, A. (2009). A digest and pattern matching-based intrusion detection engine. *The Computer Journal*, Vol. 52, pp. 699–723.
39. Biermann, E., Cloete, E. & Venter, L. (2001). A comparison of Intrusion detection systems. *Computers & Security*, Vol. 20, pp. 676–683.
40. Mitrokotsa, A. & Dimitrakakis, C. (2012). Intrusion detection in manet using classification algorithms: The effects of cost and model selection. Ad Hoc Networks.
41. Pan, Z., Chen, S., Hu, G. & Zhang, D. (2003). Hybrid neural network and C4. 5 for misuse detection, in: Machine Learning and Cybernetics. *2003 International Conference on*, Vol. 4, IEEE, pp. 2463–2467.
42. Wu, S. X. & Banzhaf, W. (2010). The use of computational intelligence in intrusion detection systems: A review. *Applied Soft Computing*, Vol. 10, pp. 1–35.
43. Hwang, K., Cai, M., Chen, Y. & Qin, M. (2007). Hybrid intrusion detection with weighted signature generation over anomalous internet episodes. *Dependable and Secure Computing, IEEE Transactions on*, Vol. 4, pp. 41–55.
44. Pfahringer, B. (2000). Winning the KDD99 classification cup: bagged boosting. *ACM SIGKDD Explorations Newsletter*, Vol. 1, pp. 65–66.
45. Shafi, K., Kovacs, T., Abbass, H. A. & Zhu, W. (2009). Intrusion detection with evolutionary learning classifier systems. *Natural Computing*, Vol. 8, pp. 3–27.
46. Chandola, V., Banerjee, A. & Kumar, V. (2012). Anomaly detection for discrete sequences: a survey. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 24, pp. 823 –839.
47. Jiang, F., Sui, Y. & Cao, C. (2011). An incremental decision tree algorithm based on rough sets and its application in intrusion detection. *Artificial Intelligence Review*, pp. 1–14.
48. Wilson, D. & Martinez, T. (2000). Reduction techniques for instance-based learning algorithms. *Machine Learning*, Vol. 38, pp. 257–286.
49. Cormen, T. (2001). *Introduction to algorithms*. The MIT press, 2 edition, pp. 405–430.
50. Kdd cup 1999 intrusion detection dataset, (2007). tp://kdd.ics.uci.edu/databases/kddcup99/task.html, Visited July 2009.
51. UCI Machine Learning Repository: KDD Cup 1999 Data Set, (2009). http://archive.ics.uci.edu/ml/databases/kddcup99, 1999. Visited Sep 2009.
52. Elkan, C. (2001). The foundations of cost-sensitive learning. *International Joint Conference on Artificial Intelligence*, volume 17, Citeseer, pp. 973–978.

53. Lippmann, R., Haines, J., Fried, D., Korba, J. & Das, K. (2000). The 1999 DARPA off-line intrusion detection evaluation, *Computer Networks*, Vol. 34, pp. 579–595.
54. Chou, T. , Yen, K. & Luo, J. (2008). Network intrusion detection design using feature selection of soft computing paradigms. *International Journal of computational intelligence*, Vol. 4, pp. 196–208.
55. Lippmann, R., Fried, D., Graf, I., Haines, J., Kendall, K., McClung, D., Weber, D., Webster, S., Wyschogrod, D., Cunningham, R. & Zissman, M. (2000). Evaluating intrusion detection systems: the 1998 darpa off-line intrusion detection evaluation. *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, Vol. 2, pp. 12 –26.
56. Sabhnani, M. & Serpen, G. (2004). Why machine learning algorithms fail in misuse detection on KDD intrusion detection data set. *Intelligent Data Analysis*, Vol. 8, pp. 403–415.
57. Elkan, C. (2000). Results of the KDD'99 classifier learning. *ACM SIGKDD Explorations Newsletter*, Vol. 1, pp. 63–64.
58. Tavallaee, M., Bagheri, E., Lu, W. & Ghorbani, A. A. (2009). A detailed analysis of the kdd cup 99 data set. *Proceedings of the Second IEEE international conference on Computational intelligence for security and defense applications, CISDA'09*, IEEE Press, Piscataway, NJ, USA, pp. 53–58.
59. McHugh, J. (2000). Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security (TISSEC)*, Vol. 3, pp. 262–294.
60. Venkatchalam, V. & Selvan, S. (2008). Performance comparison of Intrusion detection system classifiers using various feature reduction techniques. *International journal of simulation*, Vol. 9, pp. 30–39.
61. Sabhnani, M. & Serpen, G. (2009). Application of machine learning algorithms to kdd intrusion detection dataset within misuse detection context. *Proceedings of International Conference on Machine Learning: Models, Technologies, and Applications*, Vol. 1, pp. 209–215.
62. Tsang, C. H., Kwong, S. & Wang, H. (2007). Genetic-fuzzy rule mining approach and evaluation of feature selection techniques for anomaly intrusion detection. *Pattern Recognition*, Vol.0, 40, pp. 2373–2391.
63. Behdad, M., Barone, L., French, T. & Bennamoun, M. (2012). On xcsr for electronic fraud detection. *Evolutionary Intelligence*, Vol. 5, pp. 139–150.
64. Toosi, A. N. & Kahani, M. (2007). A new approach to intrusion detection based on an evolutionary soft computing model using neuro-fuzzy classifiers. *Computer Communications*, Vol. 30, pp. 2201–2212.
65. Agarwal, R. & Joshi, M. V. (2000). PNrule: A new framework for learning classi er models in data mining, Techinical Report TR 00-015, IBM Research Report, Computer Science/Mathematics.
66. Amiri, F., Rezaei, M., Lucas, C., Shakery, A. & Yazdani, N. (2011). Journal of Network and Computer Applications Mutual information-based feature selection for intrusion detection systems. *Journal of Network and Computer Applications*, Vol. 34, pp. 1184–1199.
67. Levin, I. & Street, H. M. (2000). KDD-99 Classifier Learning Contest: LLSoft's Results Overview. *SIGKDD explorations*, Vol. 1, pp. 67–75.