# Creating Evolving User Behavior Profiles Automatically

Jose Antonio Iglesias, *Member*, *IEEE Computer Society*,
Plamen Angelov, *Senior Member*, *IEEE Computer Society*,
Agapito Ledezma, *Member*, *IEEE Computer Society*, and
Araceli Sanchis, *Member*, *IEEE Computer Society*

**Abstract** Knowledge about computer users is very beneficial for assisting them, predicting their future actions or detecting masqueraders. In this paper, a new approach for creating and recognizing automatically the behavior profile of a computer user is presented. In this case, a computer user behavior is represented as the sequence of the commands she/he types during her/his work. This sequence is transformed into a distribution of relevant subsequences of commands in order to find out a profile that defines its behavior. Also, because a user profile is not necessarily fixed but rather it evolves/changes, we propose an evolving method to keep up to date the created profiles using an *Evolving Systems* approach. In this paper, we combine the evolving classifier with a trie-based user profiling to obtain a powerful self-learning online scheme. We also develop further the recursive formula of the potential of a data point to become a cluster center using cosine distance, which is provided in the Appendix. The novel approach proposed in this paper can be applicable to any problem of dynamic/evolving user behavior modeling where it can be represented as a sequence of actions or events. It has been evaluated on several real data streams.

**Index Terms** Evolving fuzzy systems, fuzzy-rule-based (FRB) classifiers, user modeling.

## 1 Introduction

Recognizing the behavior of others in real time is a significant aspect of different human tasks in many different environments. When this process is carried out by software agents or robots, it is known as *user modeling*. The recognition of users can be very beneficial for assisting them or predicting their future actions. Most existing techniques for user recognition assume the availability of handcrafted user profiles, which encode the a-priori known behavioral repertoire of the observed user. However, the construction of effective user profiles is a difficult problem for different reasons: human behavior is often erratic, and sometimes humans behave differently because of a change in their goals. This last problem makes necessary that the user profiles we create *evolve*.

There exists several definitions for user profile [1]. It can be defined as the description of the user interests, characteristics, behaviors, and preferences. User profiling is the practice of gathering, organizing, and interpreting the user profile information. In recent years, significant work has been carried out for profiling users, but most of the user profiles do not change according to the environment and new goals of the user. An example of how to create these static profiles is proposed in a previous work [2].

In this paper, we propose an adaptive approach for creating behavior profiles and recognizing computer users. We call this approach *Evolving Agent behavior Classification based on Distributions of relevant events* (EVABCD) and it is based on representing the observed behavior of an *agent* (computer user) as an adaptive distribution of her/his relevant atomic behaviors (events). Once the model has been created, EVABCD presents an evolving method for updating and evolving the user profiles and classifying an observed user. The approach we present is generalizable to all kinds of user behaviors represented by a sequence of events.

The *UNIX* operating system environment is used in this research for explaining and evaluating EVABCD. A user behavior is represented in this case by a sequence of *UNIX* commands typed by a computer user in a command-line interface. Previous research studies in this environment [3], [4] focus on detecting masquerades (individuals who impersonate other users on computer networks and systems) from sequences of *UNIX* commands. However, EVABCD creates evolving user profiles and classifies new users into one of the previously created profiles. Thus, the goal of EVABCD in the *UNIX* environment can be divided into two phases:

1. Creating and updating user profiles from the commands the users typed in a *UNIX* shell.
2. Classifying a *new* sequence of commands into the predefined profiles.

Because we use an evolving classifier, it is constantly learning and adapting the existing classifier structure to accommodate the newly observed emerging behaviors. Once a user is classified, relevant actions can be done; however, this task is not addressed in this paper.

- *J.A. Iglesias, A. Ledezma, and A. Sanchis are with the CAOS Group, Carlos III University of Madrid, Spain.*
  *E mail: {jiglesia, ledezma, masm}@inf.uc3m.es.*
- *P. Angelov is with the Department of Communication Systems, InfoLab21, Lancaster University, United Kingdom.*
  *E mail: p.angelov@lancaster.ac.uk.*

The creation of a user profile from a sequence of *UNIX* commands should consider the consecutive order of the commands typed by the user and the influence of his/her past experiences. This aspect motivates the idea of automated sequence learning for computer user behavior classification; if we do not know the features that influence the behavior of a user, we can consider a sequence of past actions to incorporate some of the historical context of the user. However, it is difficult, or in general, impossible, to build a classifier that will have a full description of all possible behaviors of the user, because these behaviors evolve with time, they are not static and new patterns may emerge as well as an old habit may be forgotten or stopped to be used. The descriptions of a particular behavior itself may also evolve, so we assume that each behavior is described by one or more fuzzy rules. A conventional system do not capture the new patterns (behaviors) that could appear in the data stream once the classifier is built. In addition, the information produced by a user is often quite large. Therefore, we need to cope with large amounts of data and process this information in real time, because storing the complete data set and analyzing it in an offline (batch) mode, would be impractical. In order to take into account these aspects, we use an evolving fuzzy-rule-based system that allows for the user behaviors to be dynamic, to evolve.

This paper is organized as follows: Section 2 provides an overview of the background and related work. The overall structure of our approach, EVABCD, is explained in Section 3. Section 4 describes the construction of the user behavior profile. The evolving *UNIX* user classifier is detailed in Section 5. Section 6 describes the experimental setting and the experimental results obtained. Finally, Section 7 contains future work and concluding remarks.

## 2 Background and Related Work

Different techniques have been used to find out relevant information related to the human behavior in many different areas. The literature in this field is vast; Macedo et al. [5] propose a system (*WebMemex*) that provides recommended information based on the captured history of navigation from a list of known users. Pepyne et al. [6] describe a method using queuing theory and logistic regression modeling methods for profiling computer users based on simple temporal aspects of their behavior. In this case, the goal is to create profiles for very specialized groups of users, who would be expected to use their computers in a very similar way. Gody and Amandi [7] present a technique to generate readable user profiles that accurately capture interests by observing their behavior on the web.

There is a lot of work focusing on user profiling in a specific environment, but it is not clear that they can be transferred to other environments. However, the approach we propose in this paper can be used in any domain in which a user behavior can be represented as a sequence of actions or events. Because sequences play a crucial role in human skill learning and reasoning [8], the problem of user profile classification is examined as a problem of sequence classification. According to this aspect, Horman and Kaminka [9] present a learner with unlabeled sequential data that discover meaningful patterns of sequential behavior from example streams. Popular approaches to such learning include statistical analysis and frequency-based methods. Lane and Brodley [10] present an approach based on the basis of instance-based learning (IBL) techniques, and several techniques for reducing data storage requirements of the user profile.

Although the proposed approach can be applied to any behavior represented by a sequence of events, we focus in this research in a command-line interface environment. Related to this environment, Schonlau et al. [3] investigate a number of statistical approaches for detecting masqueraders. Coull et al. [11] propose an effective algorithm that uses pairwise sequence alignment to characterize similarity between sequences of commands. Recently, Angelov and Zhou propose in [12] to use evolving fuzzy classifiers for this detection task.

In [13], Panda and Patra compared the performance of different classification algorithms—Naive Bayesian (NB), C4.5 and Iterative Dichotomizer 3 (ID3)—for network intrusion detection. According to the authors, ID3 and C4.5 are robust in detecting new intrusions, but NB performs better to overall classification accuracy. Cufoglu et al. [14] evaluated the classification accuracy of NB, IB1, SimpleCART, NBTree, ID3, J48, and Sequential Minimal Optimization (SMO) algorithms with large user profile data. According to the simulation results, NBTree classifier performs the best classification on user-related information.

It should be emphasized that all of the above approaches ignore the fact that user behaviors can change and evolve. However, this aspect needs to be taken into account in the proposed approach. In addition, owing to the characteristics of the proposed environment, we need to extract some sort of knowledge from a continuous stream of data. Thus, it is necessary that the approach deals with the problem of classification of streaming data. Incremental algorithms build and refine the model at different points in time, in contrast to the traditional algorithms which perform the model in a batch manner. It is more efficient to revise an existing hypothesis than it is to generate hypothesis each time a new instance is observed. Therefore, one of the solution to the proposed scenario is the incremental classifiers.

An incremental learning algorithm can be defined as one that meets the following criteria [15]:

1. It should be able to learn additional information from new data.
2. It should not require access to the original data, used to train the existing classifier.
3. It should preserve previously acquired knowledge.
4. It should be able to accommodate new classes that may be introduced with new data.

Several incremental classifiers have been implemented using different frameworks

- Decision trees. The problem of processing streaming data in online has motivated the development of many algorithms which were designed to learn decision trees incrementally [16], [17]. Some examples of the algorithms which construct incremental decision trees are: ID4 [18], ID5 [19], and ID5R [20].

- Artificial neural networks (ANN). Adaptive Resonance Theory (ART) networks [21] are unsupervised ANNs proposed by Carpenter that dynamically determine the number of clusters based on a vigilance parameter [22]. In addition, Kasabov proposed another incremental learning neural network architecture, called Evolving Fuzzy Neural Network (EFuNN) [23]. This architecture does not require access to previously seen data and can accommodate new classes. A new approach to incremental learning using evolving neural networks is proposed by Seipone and Bullinaria [24]. This approach uses an evolutionary algorithm to evolve some MLP parameters. This process aims at evolving the parameters to produce networks with better incremental abilities.
- Prototype-based supervised algorithms. Learning Vector Quantization (LVQ) is one of the well-known nearest prototype learning algorithms [25]. LVQ can be considered to be a supervised clustering algorithm, in which each weight vector can be interpreted as a cluster center. Using this algorithm, the number of reference vectors has to be set by the user. For this reason, Poirier and Ferrieux proposed a method to generate new prototypes dynamically. This incremental LVQ is known as Dynamic Vector Quantization (DVQ) [26]. However, this method lacks the generalizing capability, resulting in the generation of many prototype neurons for applications with noisy data.
- Bayesian. Bayesian classifier is an effective methodology for solving classification problems when all features are considered simultaneously. However, when the features are added one by one in Bayesian classifier in batch mode in forward selection method, huge computation is involved. For this reason, Agrawal and Bala [27] proposed an incremental Bayesian classifier for multivariate normal distribution data sets. Several incremental versions of Bayesian classifiers are proposed in [28]
- Support Vector Machine (SVM). A Support Vector Machine performs classification by constructing an N-dimensional hyperplane that optimally separates the data into two categories. Training an SVM "incrementally" on new data by discarding all previous data except their support vectors, gives only approximate results. Cauwenberghs et al. consider incremental learning as an exact online method to construct the solution recursively, one point at a time. In addition, Xiao et al. [29] propose an incremental algorithm which utilizes the properties of SV set, and accumulates the distribution knowledge of the sample space through the adjustable parameters.

However, as this research focus in a command-line interface environment, it is necessary an approach able to process streaming data in real time and also cope with huge amounts of data. Several incremental classifiers do not considered this last aspect. In addition, the structure of the incremental classifiers is assumed to be fixed, and they can not address the problem of so-called concept *drift* and *shift* [30]. By *drift*, they refer to a modification of the concept over time, and *shift* usually refers to a sudden and abrupt changes in the streaming data. To capture these changes, it is necessary not only tuning parameters of the classifiers, but also a change in its structure. A simple incremental algorithm does not evolve the structure of the classifier. The interpretation of the results is also an important characteristic in a classifier, and several incremental classifiers (such as ANN or SVM) are not good in terms of interpretation of the results. Finally, the computational efficiency of the proposed classifier is very important, and some incremental algorithms (such SVM) have to solve quadratic optimization problems many times.

Taking all these aspects into account, we propose in this paper an evolving fuzzy-rule-base system which satisfies all of the criteria of the incremental classifiers. However, the approach has also important advantages which make it very useful in real environments

1. It can cope with huge amounts and data.
2. Its evolving structure can capture sudden and abrupt changes in the stream of data.
3. Its structure meaning is very clear, as we propose a rule-based classifier.
4. It is noniterative and single pass; therefore, it is computationally very efficient and fast.
5. Its classifier structure is simple and interpretable.

Thus, an approach for creating and recognizing automatically the behavior profile of a computer user with the time evolving in a very effective way. To the best of our knowledge, this is the first publication where user behavior is considered, treated, and modeled as a dynamic and evolving phenomenon. This is the most important contribution of this paper.

## 3 The Proposed Approach

This section introduces the proposed approach for automatic clustering, classifier design, and classification of the behavior profiles of users. The novel evolving user behavior classifier is based on *Evolving Fuzzy Systems* and it takes into account the fact that the behavior of any user is not fixed, but is rather changing. Although the proposed approach can be applied to any behavior represented by a sequence of events, we detail it using a command-line interface (*UNIX* commands) environment.

In order to classify an observed behavior, our approach, as many other agent modeling methods [31], creates a library which contains the different expected behaviors. However, in our case, this library is not a prefixed one, but is evolving, learning from the observations of the users real behaviors and, moreover, it starts to be filled in "from scratch" by assigning temporarily to the library the first observed user as a prototype. The library, called *Evolving-Profile-Library (EPLib)*, is continuously changing, evolving influenced by the changing user behaviors observed in the environment.

Thus, the proposed approach includes at each step the following two main actions:

1. **Creating and evolving the classifier.** This action involves in itself two subactions:

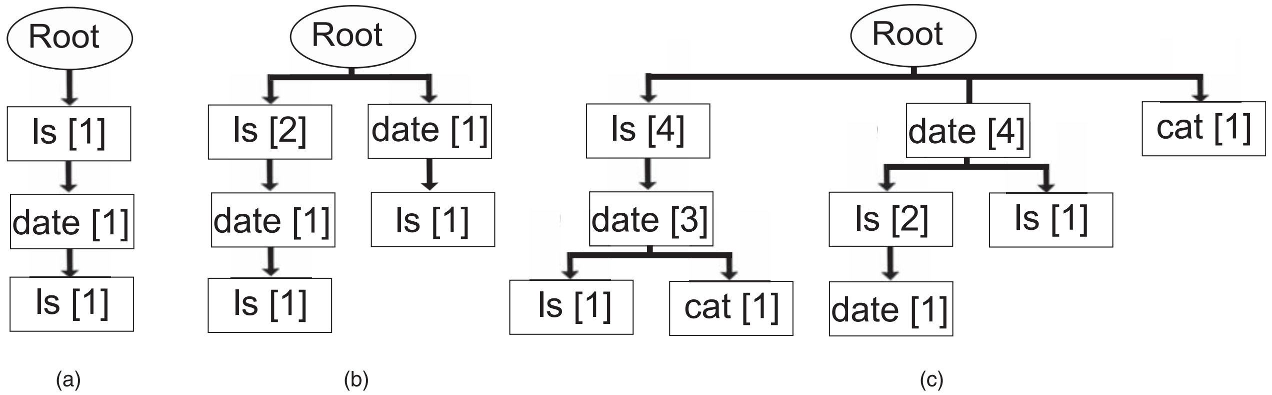


Fig. 1. Steps of creating an example trie.

a. **Creating the user behavior profiles**. This subaction analyzes the sequences of commands typed by different *UNIX* users online (data stream), and creates the corresponding profiles. This process is detailed in Section 4.

b. **Evolving the classifier**. This subaction includes online learning and update of the classifier, including the potential of each behavior to be a prototype, stored in the *EPLib*. The whole process is explained in Section 5.

2. **User classification.** The user profiles created in the previous action are associated with one of the prototypes from the *EPLib*, and they are classified into one of the classes formed by the prototypes. This action is also detailed in Section 5.

## 4 Construction of the User Behavior Profile

In order to construct a user behavior profile in online mode from a data stream, we need to extract an ordered sequence of recognized events; in this case, UNIX commands. These commands are inherently sequential, and this sequentiality is considered in the modeling process. According to this aspect and based on the work done in [2], in order to get the most representative set of subsequences from a sequence, we propose the use of a *trie* data structure [32]. This structure was also used in [33] to classify different sequences and in [34], [35] to classify the behavior patterns of a *RoboCup* soccer simulation team.

The construction of a user profile from a single sequence of commands is done by a three step process:

1. Segmentation of the sequence of commands.
2. Storage of the subsequences in a *trie*.
3. Creation of the user profile.

These steps are detailed in the following three sections. For the sake of simplicity, let us consider the following sequence of commands as an example: $\{ls \rightarrow date \rightarrow ls \rightarrow date \rightarrow cat\}$.

### 4.1 Segmentation of the Sequence of Commands

First, the sequence is segmented into subsequences of equal length from the first to the last element. Thus, the sequence $A = A_1A_2 \ldots A_n$ (where $n$ is the number of commands of the sequence) will be segmented in the subsequences described by $A_i \ldots A_{i+length}$ $\forall i,\ i = [1, n - length + 1]$, where *length* is the size of the subsequences created. In the remainder of the paper, we will use the term *subsequence length* to denote the value of this length. This value determines how many commands are considered as dependent.

In the proposed sample sequence ($\{ls \rightarrow date \rightarrow ls \rightarrow date \rightarrow cat\}$), let 3 be the subsequence length, then we obtain

$\{LS \rightarrow DATE \rightarrow LS\}$, $\{date \rightarrow ls \rightarrow date\}$, $\{ls \rightarrow date \rightarrow cat\}$.

### 4.2 Storage of the Subsequences in a *trie*

The subsequences of commands are stored in a *trie* data structure. When a new model needs to be constructed, we create an empty *trie*, and insert each subsequence of events into it, such that all possible subsequences are accessible and explicitly represented. Every trie node represents an event appearing at the end of a subsequence, and the nodes children represent the events that have appeared following this event. Also, each node keeps track of the number of times a command has been recorded into it. When a new subsequence is inserted into a *trie*, the existing nodes are modified and/or new nodes are created. As the dependencies of the commands are relevant in the user profile, the subsequence suffixes (subsequences that extend to the end of the given sequence) are also inserted.

Considering the previous example, the first subsequence ($\{ls \rightarrow date \rightarrow ls\}$) is added as the first branch of the empty *trie* (Fig. 1a). Each node is labeled with the number 1 which indicates that the command has been inserted in the node once (in Fig. 1, this number is enclosed in square brackets). Then, the suffixes of the subsequence ($\{date \rightarrow ls\}$ and $\{ls\}$) are also inserted (Fig. 1b). Finally, after inserting the three subsequences and its corresponding suffixes, the completed *trie* is obtained (Fig. 1c).

### 4.3 Creation of the User Profile

Once the *trie* is created, the subsequences that characterize the user profile and its relevance are calculated by traversing the *trie*. For this purpose, **frequency-based methods** are used. In particular, in EVABCD, to evaluate the relevance of a subsequence, its relative frequency or support [36] is calculated. In this case, the support of a subsequence is defined as the ratio of the number of times the subsequence has been inserted into the *trie* and the total number of subsequences of equal size inserted.

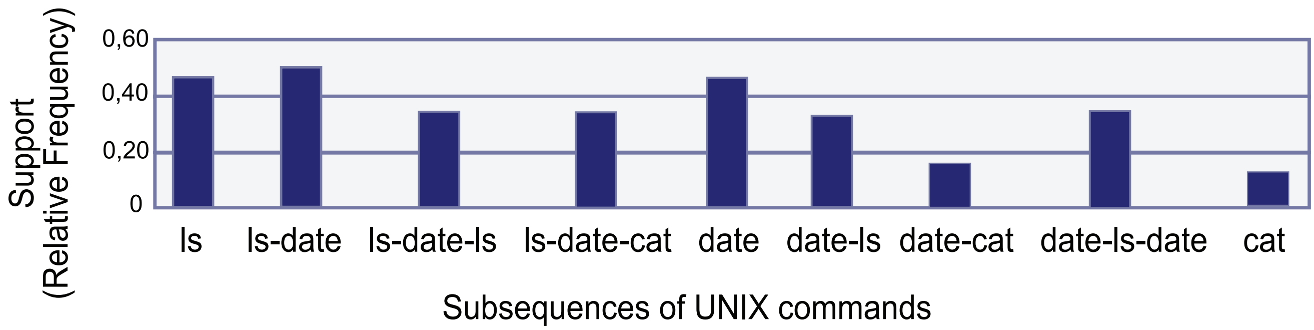


Fig. 2. Distribution of subsequences of commands   Example.

In this step, the *trie* can be transformed into a set of subsequences labeled by its support value. In EVABCD, this set of subsequences is represented as a distribution of relevant subsequences. Thus, we assume that user profiles are n-dimensional matrices, where each dimension of the matrix will represent a particular subsequence of commands.

In the previous example, the *trie* consists of nine nodes; therefore, the corresponding profile consists of nine different subsequences which are labeled with its support. Fig. 2 shows the distribution of these subsequences.

Once a user behavior profile has been created, it is classified and used to update the *Evolving-Profile-Library*, as explained in the next section.

## 5 Evolving UNIX User Classifier

A classifier is a mapping from the feature space to the class label space. In the proposed classifier, the feature space is defined by distributions of subsequences of events. On the other hand, the class label space is represented by the most representative distributions. Thus, a distribution in the class label space represents a specific behavior which is one of the prototypes of the *EPLib*. **The prototypes are not fixed and evolve** taking into account the new information collected online from the data stream—this is what makes the classifier *Evolving*. The number of these prototypes is not prefixed but it depends on the homogeneity of the observed behaviors. The following sections describes how a user behavior is represented by the proposed classifier, and how this classifier is created in an evolving manner.

### 5.1 User Behavior Representation

EVABCD receives observations in real time from the environment to analyze. In our case, these observations are UNIX commands and they are converted into the corresponding distribution of subsequences online. In order to classify a *UNIX* user behavior, these distributions must be represented in a data space. For this reason, each distribution is considered as a data vector that defines a *point* that can be represented in the data space.

The data space in which we can represent these *points* should consist of $n$ dimensions, where $n$ is the number of the different subsequences that could be observed. It means that we should know all the different subsequences of the environment a priori. However, this value is unknown and the creation of this data space from the beginning is not efficient. For this reason, in EVABCD, the dimension of the data space also **evolves**, it is incrementally growing according to the different subsequences that are represented in it.

Fig. 3 explains graphically this novel idea. In this example, the distribution of the first user consists of five subsequences of commands (*ls, ls-date, date, cat*, and *date-cat*); therefore, we need a five-dimensional data space to represent this distribution because each different subsequence is represented by one dimension. If we consider the second user, we can see that three of the five previous subsequences have not been typed by this user (*ls-date, date,* and *date-cat*), so these values are not available. Also, the values of the two new subsequences (*cp* and *ls-cp*) need to be represented in the same data space; thus, it is necessary to increase the dimensionality of the data space from five to seven. To sum up, the dimensions of the data space represent the different subsequences typed by the users and they will increase according to the different new subsequences obtained.

### 5.2 Structure of the EVABCD

Once the corresponding distribution has been created from the stream, it is processed by the classifier. The structure of this classifier includes

1. **Classify the new sample** in a class represented by a prototype. Section 5.6.
2. **Calculate the *potential*** of the new data sample to be a prototype. Section 5.3.
3. **Update all the prototypes** considering the new data sample. It is done because the *density* of the data space surrounding certain data sample changes with the

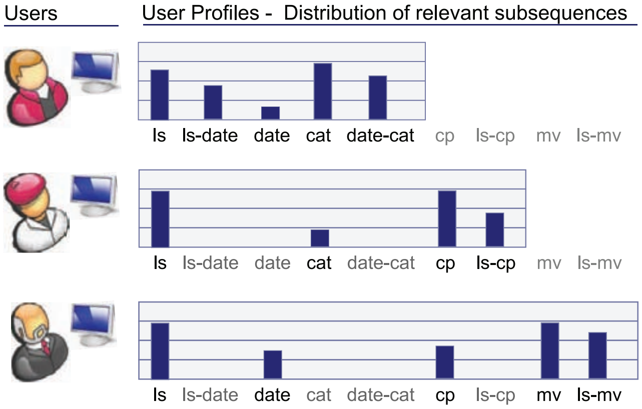


Fig. 3. Distributions of subsequences of events in an evolving system approach   Example.

insertion of each new data sample. **Insert the new data sample** as a new prototype if needed. Section 5.4.

4. **Remove** any prototype if needed. Section 5.5.

Therefore, as we can see, the classifier does not need to be configured according to the environment where it is used because it can start *"from scratch."* Also, the relevant information of the obtained samples is necessary to update the library, but, as we will explain in the next section, there is no need to store all the samples in it.

## 5.3 Calculating the Potential of a Data Sample

As in [12], a prototype is a data sample (a behavior represented by a distribution of subsequences of commands) that represents several samples which represent a certain class.

The classifier is initialized with the first data sample, which is stored in *EPLib*. Then, each data sample is classified to one of the prototypes (classes) defined in the classifier. Finally, based on the *potential* of the new data sample to become a prototype [37], it could form a new prototype or replace an existing one.

The potential (P) of the $k$th data sample $(x_k)$ is calculated by (1) which represents a function of the accumulated distance between a sample and all the other k − 1 samples in the data space [12]. The result of this function represents the *density* of the data that surrounds a certain data sample

$$P(x_k) = \frac{1}{1 + \frac{\sum_{i\ 1}^{k\ 1} distance^2(x_k, x_i)}{k\ 1}}, \tag{1}$$

where *distance* represents the distance between two samples in the data space.

In [38], the potential is calculated using the euclidean distance and in [12] it is calculated using the cosine distance. Cosine distance has the advantage that it tolerates different samples to have different number of attributes; in this case, an attribute is the support value of a subsequence of commands. Also, cosine distance tolerates that the value of several subsequences in a sample can be null (null is different than zero). Therefore, EVABCD uses the cosine distance (*cosDist*) to measure the similarity between two samples, as it is described below

$$cosDist(x_k, x_p) = 1 - \frac{\sum_{j\ 1}^{n} x_{kj} x_{pj}}{\sqrt{\sum_{j\ 1}^{n} x_{kj}^2 \sum_{j\ 1}^{n} x_{pj}^2}}, \tag{2}$$

where $x_k$ and $x_p$ represent the two samples to measure its distance and $n$ represents the number of different attributes in both samples.

### 5.3.1 Calculating the Potential Recursively

Note that the expression in (1) requires all the accumulated data sample available to be calculated, which contradicts to the requirement for real time and online application needed in the proposed approach. For this reason, we need to develop a recursive expression of the potential, in which it is not needed to store the history of all the data.

In (3), the potential is calculated in the input-output joint data space, where $z = [x, Label]$; therefore, the $k$th data sample $(x_k)$ with its corresponding label is represented as $z_k$. Each sample is represented by a set of values; the value of the $i$th attribute (element) of the $z_k$ sample is represented as $z_k^i$.

As it is detailed in the Appendix, this derivation is proposed as a novel contribution and the result is as follows:

$$\begin{aligned}
&P_k(z_k) = \frac{1}{2 + \left[\frac{1}{h(k\ 1)}\left[\left[-2B_K\right] + \left[\frac{1}{h} D_k\right]\right]\right]} \\
&k = 2, 3 \ldots; P_1(z_1) = 1 \\
&where: \\
&B_k = \sum_{j\ 1}^{n} z_k^j b_k^j;\ b_k^j = b_{(k\ 1)}^j + \sqrt{\frac{\left(z_k^j\right)^2}{\sum_{l\ 1}^{n}\left(z_k^l\right)^2}} \\
&b_1^j = \sqrt{\frac{\left(z_1^j\right)^2}{\sum_{l\ 1}^{n}\left(z_1^l\right)^2}};\ j = [1, n+1] \\
&D_k = \sum_{j\ 1}^{n} z_k^j \sum_{p\ 1}^{n} z_k^p d_k^{jp};\ d_k^{jp} = d_{(k\ 1)}^{jp} + \frac{z_k^j z_k^p}{\sum_{l\ 1}^{n}(z_k^l)^2} \\
&d_1^{j1} = \frac{z_1^j z_1^1}{\sum_{l\ 1}^{n}\left(z_1^l\right)^2};\ j = [1, n+1],
\end{aligned} \tag{3}$$

where $d_k^{ij}$ represents this accumulated value for the $k$th data sample considering the multiplication of the $i$th and $j$th attributes of the data sample. Thus, to get recursively the value of the potential of a sample using (1), it is necessary to calculate $k \times k$ different accumulated values which store the result of the multiplication of an attribute value of the data sample by all the other attribute values of the data sample.

However, since the number of needed accumulated values is very large, we simplify this expression in order to calculate it faster and with less memory.

### 5.3.2 Simplifying the Potential Expression

In our particular application, the data are represented by a set of support values and are thus positive. To simplify the recursive calculation of the expression (1), we can use simply the distance instead of square of the distance. For this reason, we use in this case (4) instead of (1)

$$P_k(z_k) = \frac{1}{1 + \frac{\sum_{i\ 1}^{k\ 1} cosDist(x_k, x_i)}{k\ 1}}. \tag{4}$$

Using (4), we develop a recursive expression similar to the recursive expressions developed in [37], [38] using euclidean distance and in [12] using cosine distance. This formula is as follows:

$$\begin{aligned}
&P_k(z_k) = \frac{1}{2 - \frac{1}{k\ 1}\frac{1}{\sqrt{\sum_{j\ 1}^{n}\left(z_k^j\right)^2}} B_k}; k = 2, 3, \ldots; P_1(z_1) = 1 \\
&where: \\
&B_k = \sum_{j\ 1}^{n} z_k^j b_k^j; b_k^j = b_{(k\ 1)}^j + \sqrt{\frac{\left(z_k^j\right)^2}{\sum_{l\ 1}^{n}\left(z_k^l\right)^2}} \\
&b_1^j = \sqrt{\frac{\left(z_1^j\right)^2}{\sum_{l\ 1}^{n}\left(z_1^l\right)^2}}; j = [1, n+1],
\end{aligned} \tag{5}$$

Using this expression, it is only necessary to calculate $(n+1)$ values where $n$ is the number of different obtained subsequences; this value is represented by $b$, where $b_k^j, j = [1, n]$ represents the accumulated value for the $k$th data sample.

### 5.4 Creating New Prototypes

The proposed evolving user behavior classifier, EVABCD, can start *"from scratch"* (without prototypes in the library) in a similar manner as *eClass* evolving fuzzy rule-based classifier proposed in [38], used in [39] for robotics and further developed in [12]. The potential of each new data sample is calculated *recursively* and the potential of the other prototypes is updated. Then, the potential of the new sample ($z_k$) is compared with the potential of the existing prototypes. A new prototype is created if its value is higher than any other existing prototype, as shown in (6)

$$\exists i,\ i = [1, NumPrototypes]:\ P(z_k) > P(Prot_i). \tag{6}$$

Thus, if the new data sample is not relevant, the overall structure of the classifier is not changed. Otherwise, if the new data sample has high descriptive power and generalization potential, the classifier evolves by adding a new prototype which represents a part of the observed data samples.

### 5.5 Removing Existing Prototypes

After adding a new prototype, we check whether any of the already existing prototypes are described *well* by the newly added prototype [12]. By *well*, we mean that the value of the membership function that describes the closeness to the prototype is a Gaussian bell function chosen due to its generalization capabilities

$$\exists i,\ i = [1, NumPrototypes]:\ \mu_i(z_k) > e^{\ 1}. \tag{7}$$

For this reason, we calculate the membership function between a data sample and a prototype which is defined as

$$\mu_i(z_k) = e^{\ \frac{1}{2}[\frac{cosDist(z_k, Prot_i)}{\sigma_i}]},\quad i = [1, NumPrototypes], \tag{8}$$

where cosDist($z_k, Prot_i$) represents the cosine distance between a data sample ($z_k$) and the $i$th prototype ($Prot_i$); $\sigma_i$ represents the spread of the membership function, which also symbolizes the radius of the zone of influence of the prototype. This spread is determined based on the scatter [40] of the data. The equation to get the spread of the $k$th data sample is defined as

$$\sigma_i(k) = \sqrt{\frac{1}{k}\sum_{j\ 1}^{k} cosDist(Prot_i, z_k)};\ \sigma_i(0) = 1, \tag{9}$$

where $k$ is the number of data samples inserted in the data space; *cosDist($Prot_i, z_k$)* is the cosine distance between the new data sample ($z_k$) and the $i$th prototype.

However, to calculate the scatter without storing all the received samples, this value can be updated recursively (as shown in [38]) by

$$\sigma_i(k) = \sqrt{[\sigma_i(k-1)]^2 + \frac{1}{k}[cosDist^2(Prot_i, z_k) - [\sigma_i(k-1)]^2]}. \tag{10}$$

### 5.6 Classification Method

In order to classify a new data sample, we compare it with all the prototypes stored in *EPLib*. This comparison is done using cosine distance and the smallest distance determines the closest similarity. This comparison is shown as

$$\begin{aligned} Class(x_z) &= Class(Prot^*); \\ Prot^* &= MIN_{i\ 1}^{NumProt}(cosDist(x_{Prototype_{\mathrm{i}}}, x_z)). \end{aligned} \tag{11}$$

The time consumed for classifying a new sample depends on the number of prototypes and its number of attributes. However, we can consider, in general terms, that both the time consumed and the computational complexity are reduced and acceptable for real-time applications (in order of milliseconds per data sample).

### 5.7 Supervised and Unsupervised Learning

The proposed classifier has been explained taking into account that the observed data samples do not have labels; thus, using unsupervised learning. In this case, the classes are created based on the existing prototypes and, thus, any prototype represents a different class (label). Such a technique is used, for example, in eClass0 [12], [38] which is a clustering-based classification.

However, the observed data samples can have a label assigned to them a priori. In this case, using EVABCD, a specific class is represented by several prototypes, because the previous process is done for each different class. Thus, each class has a number of prototypes that depends on how heterogeneous are the samples of the same class. In this case, a new data sample is classified in the class which belongs the closest prototype. This technique is used, for example, in eClass1 [12], [38].

## 6 Experimental Setup and Results

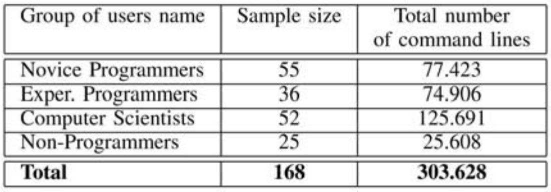
In order to evaluate EVABCD in the *UNIX* environment, we use a data set with the *UNIX* commands typed by 168 real users and labeled in four different groups. Therefore, in these experiments, we use ***supervised learning***. The explained process is applied for each of the four group (classes) and one or more prototypes will be created for each group. EVABCD is applied in these experiments considering the data set as pseudo-online streams. However, only using data sets in an offline mode, the proposed classifier can be compared with other incremental and nonincremental classifiers.

### 6.1 Data Set

In these experiments, we use the command-line data collected by Greenberg [41] using *UNIX csh* command interpreter. This data are identified in four target groups which represent a total of 168 male and female users with a wide cross section of computer experience and needs. Salient features of each group are described below

- ***Novice programmers***. The users of this group had little or no previous exposure to programming, operating systems, or *UNIX*-like command-based interfaces. These users spent most of their time learning how to program and use the basic system facilities.

TABLE 1
Sample Group Sizes and Command Lines Recorded



| Group of users name | Sample size | Total number of command lines |
|---|---|---|
| Novice Programmers | 55 | 77.423 |
| Exper. Programmers | 36 | 74.906 |
| Computer Scientists | 52 | 125.691 |
| Non-Programmers | 25 | 25.608 |
| **Total** | **168** | **303.628** |

- ***Experienced programmers***. These group members were senior Computer Science undergraduates, expected to have a fair knowledge of the *UNIX* environment. These users used the system for coding, word processing, employing more advanced *UNIX* facilities to fulfill course requirements, and social and exploratory purposes.
- ***Computer scientist***. This group, graduates and researchers from the Department of Computer Science, had varying experience with *UNIX*, although all were experts with computers. Tasks performed were less predictable and more varied than other groups, research investigations, social communication, word processing, maintaining databases, and so on.
- ***Nonprogrammers***. Word processing and document preparation was the dominant activity of the members of this group, made up of office staff and members of the Faculty of Environmental Design. Knowledge of *UNIX* was the minimum necessary to get the job done.

The sample sizes (the number of people observed) and the total number of command lines are indicated in Table 1.

### 6.2 Comparison with Other Classifiers

In order to evaluate the performance of EVABCD, it is compared with several different types of classifiers. This experiment focuses on using the well-established batch classifiers described as follows:

- The **C5.0** algorithm [42] is a commercial version of the C4.5 [43] decision-tree-based classifier.
- The **Naive Bayes** classifier (**NB**) is a simple probabilistic classifier based on applying Bayes theorem with strong (naive) independence assumptions [44]. In this case, the numeric estimator precision values are chosen based on analysis of the training data. For this reason, the classifier is no-incremental [45].
- The **K Nearest Neighbor** classifier (**kNN**) is an instance-based learning technique for classifying objects based on closest training examples in the feature space [46]. The parameter k affects the performance of the kNN classifier significantly. In these experiments, the value with which better results are obtained is $k = 1$; thus, a sample is assigned to the class of its nearest neighbor. However, using this value, the corresponding classifier may not be robust enough to the noise in the data sample.
- The **AdaBoost** (*adaptive boosting*) classifier [47] combines multiple weak learners in order to form an efficient and strong classifier. Each weak classifier uses a different distribution of training samples. Combining weak classifiers take advantage of the so-called *instability* of the weak classifier. This instability causes the classifier to construct sufficiently different decision surfaces for minor modifications in their training data sets. In these experiments, we use a decision stump as a weak learner. A decision stump is advantageous over other weak learners because it can be calculated very quickly. In addition, there is a one-to-one correspondence between a weak learner and a feature when a decision stump is used [48].
- The **Support Vector Machine** classifier relies on the statistical learning theory. In this case, the algorithm *Sequential Minimal Optimization* is used for training support vector machines by breaking a large quadratic programming (QP) optimization problem into a series of smallest possible QP problems [49]. In addition, in order to determine the best kernel type for this task, we have performed several experiments using polynomial kernel function with various degrees and radial basis function. In this case, the polynomial kernel function results performed best in all experiments. Therefore, we use polynomial kernel function of degree 1 for proposed experiments.
- The **Learning Vector Quantization** classifier is a supervised version of vector quantization, which defines class boundaries based on prototypes, a nearest-neighbor rule and a winner-takes-it-all paradigm. The performance of LVQ depends on the algorithm implementation. In these experiments, it is used the enhanced version of LVQ1, the OLVQ1 implementation [50].

In addition, we compare EVABCD with two incremental classifiers which share a common theoretical basis with batch classifiers. However, they can be incrementally updated as new training instances arrive; they do not have to process all the data in one batch. Therefore, most of the incremental learning algorithms can process data files that are too large to fit in memory. The incremental classifiers used in these experiments are detailed as follows:

- **Incremental naive bayes**. This classifier uses a default precision of 0.1 for numeric attributes when it is created with zero training instances.
- **Incremental kNN**. The kNN classifier adopts an instance-based approach whose conceptual simplicity makes its adaptation to an incremental classifier straightforward. There is an algorithm called incremental kNN as the number of nearest neighbors k needs not be known in advance and it is calculated incrementally. However, in this case, the difference between Incremental kNN and nonincremental kNN is the way all the data are processed. Unlike other incremental algorithms, kNN stores entire data set internally. In this case, the parameter k with which better results are obtained is $k = 1$.

In these experiments, the classifiers were trained using a feature vector for each of the 168 users. This vector consists of the support value of all the different subsequences of commands obtained for all the users; thus, there are

TABLE 2
Total Number of Different Subsequences Obtained

| Number of commands per user | Subsequence Length | Number of different subsequences |
|---|---|---|
| 100 | 3 | 11.451 |
| | 5 | 34.164 |
| 500 | 3 | 39.428 |
| | 5 | 134.133 |
| 1.000 | 3 | 63.375 |
| | 5 | 227.715 |

TABLE 3
EvABCD: Number of Prototypes Created per Group
Using 10-Fold Cross Validation

| | Number of prototypes in each of the 10 runs | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Group | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Novice Programmers | 4 | 5 | 3 | 4 | 4 | 3 | 2 | 3 | 4 | 5 |
| Exp. Programmers | 1 | 1 | 1 | 1 | 2 | 3 | 2 | 1 | 1 | 2 |
| Computer Scientists | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| Non-Programmers | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

subsequences which do not have a value because the corresponding user has not typed those commands. In this case, in order to be able to use this data for training the classifiers, we consider this value as 0 (although its real value is *null*).

### 6.3 Experimental Design

In order to measure the performance of the proposed classifier using the above data set, the well-established technique of **10-fold cross validation** is chosen. Thus, all the users (training set) are divided into 10 disjoint subsets with equal size. Each of the 10 subsets is left out in turn for evaluation. It should be emphasized that EVABCD **does not need to work in this mode**; this is done mainly in order to have comparable results with the established offline techniques.

The number of *UNIX* commands typed by a user and used for creating his/her profile is very relevant in the classification process. When EVABCD is carried out in the field, the behavior of a user is classified (and the evolving behavior library updated) after she/he types a limited number of commands. In order to show the relevance of this value using the data set already described, we consider different number of *UNIX* commands for creating the classifier: **100, 500,** and **1.000 commands per user**.

In the phase of behavior model creation, the length of the subsequences in which the original sequence is segmented (used for creating the *trie*) is an important parameter: using long subsequences, the time consumed for creating the *trie* and the number of relevant subsequences of the corresponding distribution increase drastically. In the experiments presented in this paper, the ***subsequence length*** varies **from 2 to 6**.

Before showing the results, we should consider that the number of subsequences obtained using different streams of data is often very large. To get an idea of how this number increases, Table 2 shows the number of different subsequences obtained using different number of commands for training (100, 500, and 1.000 commands per user) and subsequence lengths (3 and 5).

Using EVABCD, the number of prototypes per class is not fixed, it varies automatically depending on the heterogeneity of the data. To get an idea about it, Table 3 tabulates the number of prototypes created per group in each of the 10 runs using 1.000 commands per user as training data set and a subsequence length of 3.

### 6.4 Results

Results are listed in Table 4. Each major row corresponds to a training-set size (100, 500, and 1.000 commands) and each such row is further subdivided into experiments with

TABLE 4
Classification Rate (Percent) of Different Classifiers in the UNIX Users Environments Using Different Size of Training Data Set and Different Subsequence Lengths

| | | Classifier and Classification Rate (%) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Incremental Classifiers | | Non-Incremental Classifiers | | | | | |
| Number of commands for training (per user) | subsequences length | EvABCD | Incremental Naive Bayes | K-NN Incremental (k=1) | Naive Bayes | K-NN (k=1) | C5.0 | AdaBoost | SVM | LVQ |
| 100 | 2 | 65,5 | 77,3 | 38,3 | 79,1 | 42,8 | 73,9 | 59,5 | 82,1 | 68,4 |
| | 3 | 64,9 | 76,1 | 36,5 | 79,7 | 39,8 | 69,6 | 59,5 | 82,1 | 65,5 |
| | 4 | 64,5 | 72,0 | 34,1 | 74,4 | 39,2 | 74,6 | 59,5 | 77,4 | 64,3 |
| | 5 | 67,9 | 73,2 | 32,3 | 75,0 | 33,3 | 68,6 | 59,5 | 72,6 | 65,5 |
| | 6 | 64,3 | 73,2 | 32,3 | 77,3 | 32,7 | 70,1 | 59,5 | 69,6 | 63,1 |
| 500 | 2 | 58,3 | 77,9 | 33,9 | 82,1 | 41,9 | 73,9 | 60,2 | 83,3 | 69,6 |
| | 3 | 59,5 | 73,8 | 36,9 | 77,3 | 39,8 | 74,6 | 60,2 | 82,7 | 71,4 |
| | 4 | 59,2 | 70,8 | 39,2 | 76,7 | 38,9 | 73,9 | 60,2 | 79,2 | 69,0 |
| | 5 | 66,7 | 71,4 | 35,1 | 76,1 | 37,3 | 73,6 | 60,2 | 76,2 | 66,1 |
| | 6 | 70,8 | 72,6 | 35,7 | 75,5 | 36,9 | 75,6 | 60,2 | 75,0 | 65,5 |
| 1.000 | 2 | 60,1 | 78,5 | 43,6 | 85,1 | 44,1 | 73,9 | 61,3 | 83,9 | 73,2 |
| | 3 | 65,5 | 77,9 | 44,0 | 81,5 | 47,3 | 74,6 | 61,3 | 81,5 | 75,0 |
| | 4 | 61,3 | 77,3 | 43,5 | 79,1 | 46,1 | 73,9 | 61,3 | 80,4 | 71,4 |
| | 5 | 70,2 | 76,7 | 42,5 | 78,5 | 44,0 | 73,6 | 61,3 | 79,2 | 69,7 |
| | 6 | 72,0 | 76,1 | 41,9 | 77,9 | 44,6 | 74,6 | 61,3 | 77,4 | 68,5 |

different subsequence lengths for segmenting the initial sequence (from two to six commands). The columns show the average classification success using the proposed approach (EvABCD) and the other (incremental and nonincremental) classification algorithms.

According to these data, SVM and Nonincremental Naive Bayes perform slightly better than the Incremental NB and C5.0 classifiers in terms of accuracy. The percentages of users correctly classified by EVABCD are higher to the results obtained by LVQ and lower than the percentages obtained by NB, C5.0, and SVM. Incremental and Nonincremental kNN classifiers perform much worse than the others. Note that the changes in the subsequence length do not modify the classification results obtained by AdaBoost. This is due to the fact that the classifier creates the same classification rule (*weak hypotheses*) although the subsequence length varies.

In general, the difference between EVABCD and the algorithms NB and SVM is considerable for small subsequence lengths (two or three commands), but this difference decreases when this length is longer (five or six commands). These results show that using an appropriate subsequence length, the proposed classifier can compete well with offline approaches.

Nevertheless, the proposed environment needs a classifier able to process streaming data in online and in real time. Only the incremental classifiers satisfy this requirement, but unlike EVABCD, they assume a fixed structure. In spite of this, taking into account the incremental classifiers, it can be noticed that the difference in the results is not very significant; besides, EVABCD can cope with huge amounts of data because its structure is open and the rule-base evolves in terms of creating new prototypes as they occur automatically. In addition, the learning in EVABCD is performed in single pass and a significantly smaller memory is used. Spending too much time for training is clearly not adequate for this purpose.

In short, EVABCD **needs an appropriate subsequence length** to get a classification rate similar to the obtained by other classifiers which use different techniques. However, EVABCD does not need to store the entire data stream in the memory and disregards any sample after being used. EVABCD is one pass (each sample is proceeded once at the time of its arrival), while other offline algorithms require a batch set of training data in the memory and make many iterations. Thus, EVABCD is **computationally more simple and efficient** as it is **recursive** and **one pass**. Unlike other incremental classifiers, EVABCD does not assume a prefixed structure and it changes according to the samples obtained. In addition, as EVABCD uses a recursive expression for calculating the potential of a sample, it is also computationally very efficient. In fact, since the number of attributes is very large in the proposed environment and it changes frequently, EVABCD is the most suitable alternative. Finally, the EVABCD structure is simple and interpretable.

### 6.5 Scenario: A *New* Class Appears

In addition to the advantage that EVABCD is an online classifier, we want to prove the ability of our approach to adapt to new data; in this case, new users or a different behavior of a user. This aspect is especially relevant in the proposed environment since a user profile changes constantly. For this purpose, we design a new experimental scenario in which the number of users of a class is incrementing in the training data in order to detect how the different classifiers recognize the users of a *new* class.

In this case, EVABCD is compared with three nonincremental classifiers and 10-fold cross validation is used. This experiment consists of four parts:

1. One of the four classes is chosen to detect how EVABCD can adapt to it. The chosen class is named as *new class*.
2. Initially, the training data set is composed by a sample of the *new class* and all the samples of the other three classes. The test data set is composed of several samples of the four classes.
3. EVABCD is applied and the classification rate of the samples of the *new class* is calculated.
4. The number of samples of the *new class* in the training data set is increasing one by one and step 3 is repeated. Thus, the classification rate of the *new class* samples is calculated in each increase.

Fig. 4 shows the four graphic results of this experiment considering in each graph one of the four classes as the *new class*:

- *x-axis* represents the number of users of the *new class* that contains the training data set.
- *y-axis* represents the percentage of users of the *new class* correctly classified.

In the different graphs, we can see **how quickly** EVABCD **evolves and adapts** to the *new class*. If we consider the class *Novice Programmers*, it is remarkable that after analyzing three users of this class, the proposed classifier is able to create a *new* prototype in which almost 90 percent of the *test users* are correctly classified. However, the other classifiers need a larger number of samples for *recognizing* the users of this *new class*. Similar performance has been observed for the other three classes. Specially, C5.0 needs several samples for creating a suitable decision tree. As we can see in the graph which represents the class *Computer scientist* as the new class, the percentages of users correctly in the 1-NN classifier is always 0 because all the users of this class are classified in the *Novice programmers* class. The increase in the classification rate is not perfectly smooth because the new data bring useful information but also noise.

Taking into account these results, we would like to remark that the proposed approach is able to adapt to a new user behavior extremely quick. In a changing and real-time environment, as the proposed in this paper, this property is essential.

## 7 Conclusions

In this paper, we propose a generic approach, EVABCD, to model and classify user behaviors from a sequence of events. The underlying assumption in this approach is that the data collected from the corresponding environment can be transformed into a sequence of events. This sequence is segmented and stored in a *trie* and the relevant subsequences are evaluated by using a frequency-based method. Then, a distribution of relevant subsequences is

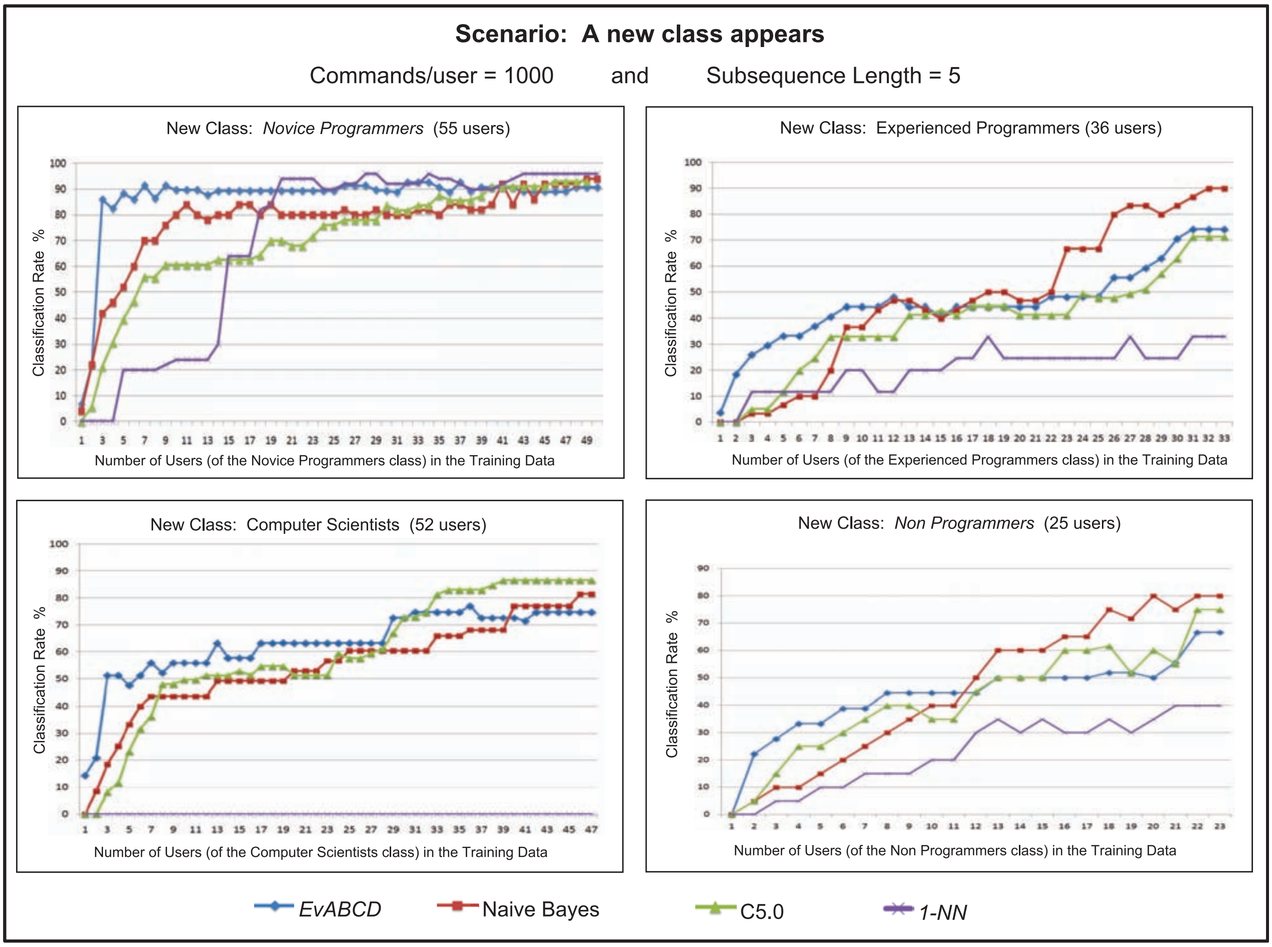


Fig. 4. Evolution of the classification rate during online learning with a subset of *UNIX* users data set.

created. However, as a user behavior is not fixed but rather it changes and evolves, the proposed classifier is able to keep up to date the created profiles using an *Evolving Systems* approach. EVABCD is one pass, noniterative, recursive, and it has the potential to be used in an interactive mode; therefore, it is computationally very efficient and fast. In addition, its structure is simple and interpretable.

The proposed evolving classifier is evaluated in an environment in which each user behavior is represented as a sequence of UNIX commands. Although EVABCD has been developed to be used online, the experiments have been performed using a batch data set in order to compare the performance to established (incremental and nonincremental) classifiers. The test results with a data set of 168 real *UNIX* users demonstrates that, using an appropriate subsequence length, EVABCD can perform almost as well as other well-established offline classifiers in terms of correct classification on validation data. However, taking into account that EVABCD is able to adapt extremely quickly to new data, and that this classifier can cope with huge amounts of data in a real environment which changes rapidly, the proposed approach is the most suitable alternative.

Although, it is not addressed in this paper, EVABCD can also be used to monitor, analyze, and detect abnormalities based on a time-varying behavior of same users and to detect masqueraders. It can also be applied to other type of users such as users of e-services, digital communications, etc.

## Appendix A

## Calculation of the Using Cosine Distance Potential Recursively

In this appendix the expression of the potential is transformed into a recursive expression in which it is calculated using only the current data sample ($z_k$). For this novel derivation we combine the expression of the potential for a sample data (1) represented by a vector of elements and the distance cosine expression (2).

$$P_k(z_k) = \frac{1}{1 + \left[\frac{1}{k\ \ 1}\sum_{i\ \ 1}^{k\ \ 1}\left[1 - \frac{\sum_{j\ \ 1}^{n} z_k^j z_i^j}{\sqrt{\sum_{j\ \ 1}^{n}\left(z_k^j\right)^2 \sum_{j\ \ 1}^{n}\left(z_i^j\right)^2}}\right]^2\right]} \qquad (12)$$

where $z_k$ denotes the $k$th sample inserted in the data space. Each sample is represented by a set of values; the value of the $i^{th}$ attribute (element) of the $z_k$ sample is represented as $z_k^i$.

In order to explain the derivation of the expression step by step; firstly, we consider the denominator of the (12) which is named as $den.P(z_k)$.

$$den.P_k(z_k) = 1 + \left[\frac{1}{k-1}\sum_{i\ 1}^{k\ 1}\left[1-\frac{\sum_{j\ 1}^{n} z_k^j z_i^j}{\sqrt{\sum_{j\ 1}^{n}(z_k^j)^2\sum_{j\ 1}^{n}(z_i^j)^2}}\right]\right]^2$$

$$den.P_k(z_k) = 1 + \left[\frac{1}{k-1}\sum_{i\ 1}^{k\ 1}\left[1-\frac{f_i}{h\ g_i}\right]^2\right]$$

where:

$$f_i = \sum_{j\ 1}^{n} z_k^j z_i^j,\ h = \sqrt{\sum_{j\ 1}^{n}\left(z_k^j\right)^2}\ and\ g_i = \sqrt{\sum_{j\ 1}^{n}\left(z_i^j\right)^2}. \qquad (13)$$

We can observe that the variables $f_i$ and $g_i$ depend on the sum of all the data samples (all these data samples are represented by $i$); but the variable $h$ represents the sum of the attribute values of the sample. Therefore, $den.P_k(z_k)$ can be simplified further into:

$$den.P_k(z_k) = 1 + \left[\frac{1}{(k-1)} + \sum_{i\ 1}^{k\ 1}\left[1-\left[\frac{-2f_i}{hg_i}\right]+\left[\frac{f_i}{hg_i}\right]^2\right]\right],$$

$$den.P_k(z_k) = 1 + \left[\frac{1}{(k-1)}\left[(k-1)+\sum_{i\ 1}^{k\ 1}\left[\frac{-2f_i}{hg_i}+\frac{f_i^2}{h^2g_i^2}\right]\right]\right]. \qquad (14)$$

And finally, into:

$$den.P_k(z_k) = 2 + \left[\frac{1}{h(k-1)}\left[\left[-2\sum_{i\ 1}^{k\ 1}\frac{f_i}{g_i}\right]+\left[\frac{1}{h}\sum_{i\ 1}^{k\ 1}\left(\frac{f_i}{g_i}\right)^2\right]\right]\right]. \qquad (15)$$

In order to obtain an expression for the potential from (15), we rename it as follows:

$$den.P_k(z_k) = 2 + \left[\frac{1}{h(k-1)}\left[[-2B_K]+\left[\frac{1}{h}D_k\right]\right]\right]$$
$$where\text{: } B_k = \sum_{i\ 1}^{k\ 1}\frac{f_i}{g_i}\ and\ \ D_k = \sum_{i\ 1}^{k\ 1}\left(\frac{f_i}{g_i}\right)^2. \qquad (16)$$

If we analyze each variable ($B_k$ and $D_k$) separately (considering the renaming done in (13)):

Firstly, we consider $B_k$

$$B_k = \sum_{i\ 1}^{k\ 1}\frac{\sum_{j\ 1}^{n} z_k^j z_i^j}{\sqrt{\sum_{j\ 1}^{n}\left(z_i^j\right)^2}} = \sum_{j\ 1}^{n} z_k^j \sum_{i\ 1}^{k\ 1}\sqrt{\frac{\left(z_i^j\right)^2}{\sum_{l\ 1}^{n}\left(z_i^l\right)^2}}. \qquad (17)$$

If we define each attribute of the sample $B_k$ by:

$$b_k^j = \sum_{i\ 1}^{k\ 1}\sqrt{\frac{\left(z_i^j\right)^2}{\sum_{l\ 1}^{n}\left(z_i^l\right)^2}}. \qquad (18)$$

Thus, the value of $B_k$ can be calculated as a recursive expression:

$$B_k = \sum_{j\ 1}^{n} z_k^j b_k^j;\ b_k^j = b_{(k\ 1)}^j + \sqrt{\frac{\left(z_k^j\right)^2}{\sum_{l\ 1}^{n}\left(z_k^l\right)^2}}$$
$$b_1^j = \sqrt{\frac{\left(z_1^j\right)^2}{\sum_{l\ 1}^{n}\left(z_1^l\right)^2}}. \qquad (19)$$

Secondly, considering $D_k$ with the renaming done in (13), we get:

$$D_k = \sum_{i\ 1}^{k\ 1}\left(\frac{\sum_{j\ 1}^{n} z_k^j z_i^j}{\sqrt{\sum_{j\ 1}^{n}\left(z_i^j\right)^2}}\right)^2 = \sum_{i\ 1}^{k\ 1}\frac{1}{\sum_{l\ 1}^{n}\left(z_i^l\right)^2}\left[\sum_{j\ 1}^{n} z_k^j z_i^j\right]^2$$
$$D_k = \sum_{j\ 1}^{n} z_k^j \sum_{p\ 1}^{n} z_k^p \sum_{i\ 1}^{k\ 1} z_k^{ij} z_k^{ip}\frac{1}{\sum_{l\ 1}^{n}\left(z_i^l\right)^2}. \qquad (20)$$

If we define $d_k^{jp}$ as each attribute of the sample $D_k$, we get:

$$d_k^{jp} = \sum_{i\ 1}^{k\ 1} z_k^{ij} z_k^{ip}\frac{1}{\sum_{l\ 1}^{n}\left(z_i^l\right)^2} \qquad (21)$$

Therefore:

$$D_k = \sum_{j\ 1}^{n} z_k^j \sum_{p\ 1}^{n} z_k^p d_k^{jp};\ d_k^{jp} = d_{(k\ 1)}^{jp} + \frac{z_k^j z_k^p}{\sum_{l\ 1}^{n}\left(z_k^l\right)^2};$$
$$d_1^{1j} = \frac{z_1^j z_1^1}{\sum_{l\ 1}^{n}\left(z_1^l\right)^2};\ j=[1,n+1]. \qquad (22)$$

Finally:

$$P_k(z_k) = \frac{1}{2+[\frac{1}{h(k\ 1)}[[-2B_K]+[\frac{1}{h}D_k]]]}$$
$$k=2,3\ldots;\ P_1(z_1)=1, \qquad (23)$$

where $B_k$ is obtained as in (19), and $D_k$ is described in (22).

Note that to get recursively the value of $B_k$, it is necessary to calculate $n$ accumulated values (in this case, $n$ is the number of the different subsequences obtained). However, to get recursively the value of $D_k$ we need to calculate $n \times n$ different accumulated values which store the result of multiply a value by all the other different values (these values are represented as $d_k^{ij}$).

## Acknowledgments

This work is partially supported by the Spanish Government under project TRA2007-67374-C02-02.

## References

[1] D. Godoy and A. Amandi, "User Profiling in Personal Information Agents: A Survey," *Knowledge Eng. Rev.*, vol. 20, no. 4, pp. 329 361, 2005.

[2] J.A. Iglesias, A. Ledezma, and A. Sanchis, "Creating User Profiles from a Command Line Interface: A Statistical Approach," *Proc. Int'l Conf. User Modeling, Adaptation, and Personalization (UMAP)*, pp. 90 101, 2009.

[3] M. Schonlau, W. Dumouchel, W.H. Ju, A.F. Karr, and Theus, "Computer Intrusion: Detecting Masquerades," *Statistical Science*, vol. 16, pp. 58 74, 2001.

[4] R.A. Maxion and T.N. Townsend, “Masquerade Detection Using Truncated Command Lines,” *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* pp. 219 228, 2002.

[5] A. Alaniz Macedo, K.N. Truong, J.A. Camacho Guerrero, and M. Graca Pimentel, “Automatically Sharing Web Experiences through a Hyperdocument Recommender System,” *Proc. ACM Conf. Hypertext and Hypermedia (HYPERTEXT '03),* pp. 48 56, 2003.

[6] D.L. Pepyne, J. Hu, and W. Gong, “User Profiling for Computer Security,” *Proc. Am. Control Conf.,* pp. 982 987, 2004.

[7] D. Godoy and A. Amandi, “User Profiling for Web Page Filtering,” *IEEE Internet Computing,* vol. 9, no. 4, pp. 56 64, July/ Aug. 2005.

[8] J. Anderson, *Learning and Memory: An Integrated Approach.* John Wiley and Sons, 1995.

[9] Y. Horman and G.A. Kaminka, “Removing Biases in Unsupervised Learning of Sequential Patterns,” *Intelligent Data Analysis,* vol. 11, no. 5, pp. 457 480, 2007.

[10] T. Lane and C.E. Brodley, “Temporal Sequence Learning and Data Reduction for Anomaly Detection,” *Proc. ACM Conf. Computer and Comm. Security (CCS),* pp. 150 158, 1998.

[11] S.E. Coull, J.W. Branch, B.K. Szymanski, and E. Breimer, “Intrusion Detection: A Bioinformatics Approach,” *Proc. Ann. Computer Security Applications Conf. (ACSAC),* pp. 24 33, 2003.

[12] P. Angelov and X. Zhou, “Evolving Fuzzy Rule Based Classifiers from Data Streams,” *IEEE Trans. Fuzzy Systems: Special Issue on Evolving Fuzzy Systems,* vol. 16, no. 6, pp. 1462 1475, Dec. 2008.

[13] M. Panda and M.R. Patra, “A Comparative Study of Data Mining Algorithms for Network Intrusion Detection,” *Proc. Int'l Conf. Emerging Trends in Eng. and Technology,* pp. 504 507, 2008.

[14] A. Cufoglu, M. Lohi, and K. Madani, “A Comparative Study of Selected Classifiers with Classification Accuracy in User Profiling,” *Proc. WRI World Congress on Computer Science and Information Eng. (CSIE),* pp. 708 712, 2009.

[15] R. Polikar, L. Upda, S.S. Upda, and V. Honavar, “Learn++: An Incremental Learning Algorithm for Supervised Neural Networks,” *IEEE Trans. Systems, Man and Cybernetics, Part C (Applications and Rev.),* vol. 31, no. 4, pp. 497 508, http:// dx.doi.org/10.1109/5326.983933, Nov. 2001.

[16] D. Kalles and T. Morris, “Efficient Incremental Induction of Decision Trees,” *Machine Learning,* vol. 24, no. 3, pp. 231 242, 1996.

[17] F.J. Ferrer Troyano, J.S. Aguilar Ruiz, and J.C.R. Santos, “Data Streams Classification by Incremental Rule Learning with Parameterized Generalization,” *Proc. ACM Symp. Applied Computing (SAC),* pp. 657 661, 2006.

[18] J.C. Schlimmer and D.H. Fisher, “A Case Study of Incremental Concept Induction,” *Proc. Fifth Nat'l Conf. Artificial Intelligence (AAAI),* pp. 496 501, 1986.

[19] P.E. Utgoff, “Id5: An Incremental Id3,” *Proc. Int'l Conf. Machine Learning,* pp. 107 120, 1988.

[20] P.E. Utgoff, “Incremental Induction of Decision Trees,” *Machine Learning,* vol. 4, no. 2, pp. 161 186, 1989.

[21] G.A. Carpenter, S. Grossberg, and D.B. Rosen, “Art2 a: An Adaptive Resonance Algorithm for Rapid Category Learning and Recognition,” *Neural Networks,* vol. 4, pp. 493 504, 1991.

[22] G.A. Carpenter, S. Grossberg, N. Markuzon, J.H. Reynolds, and D.B. Rosen, “Fuzzy Artmap: A Neural Network Architecture for Incremental Supervised Learning of Analog Multidimensional Maps,” *IEEE Trans. Neural Networks,* vol. 3, no. 5, pp. 698 713, Sept. 1992.

[23] N. Kasabov, “Evolving Fuzzy Neural Networks for Supervised/ Unsupervised Online Knowledge Based Learning,” *IEEE Trans. Systems, Man and Cybernetics Part B: Cybernetics,* vol. 31, no. 6, pp. 902 918, Dec. 2001.

[24] T. Seipone and J.A. Bullinaria, “Evolving Improved Incremental Learning Schemes for Neural Network Systems,” *Proc. IEEE Congress on Evolutionary Computation,* pp. 2002 2009, 2005.

[25] T. Kohonen, J. Kangas, J. Laaksonen, and K. Torkkola, “Lvq pak: A Program Package for the Correct Application of Learning Vector Quantization Algorithms,” *Proc. IEEE Int'l Conf. Neural Networks,* pp. 725 730, 1992.

[26] F. Poirier and A. Ferrieux, “Dvq: Dynamic Vector Quantization An Incremental Lvq,” *Proc. Int'l Conf. Artificial Neural Networks,* pp. 1333 1336, 1991.

[27] R.K. Agrawal and R. Bala, “Incremental Bayesian Classification for Multivariate Normal Distribution Data,” *Pattern Recognition Letters,* vol. 29, no. 13, pp. 1873 1876, http://dx.doi.org/10.1016/ j.patrec.2008.06.010, 2008.

[28] K. M, A. Chai, H.L. Chieu, and H.T. Ng, “Bayesian Online Classifiers for Text Classification and Filtering,” *Proc. Int'l Conf. Research and Development in Information Retrieval (SIGIR),* pp. 97 104, 2002.

[29] R. Xiao, J. Wang, and F. Zhang, “An Approach to Incremental SVM Learning Algorithm,” *Proc. IEEE Int'l Conf. Tools with Artificial Intelligence,* pp. 268 278, 2000.

[30] G. Widmer and M. Kubat, “Learning in the Presence of Concept Drift and Hidden Contexts,” *Machine Learning,* vol. 23, pp. 69 101, 1996.

[31] P. Riley and M.M. Veloso, “On Behavior Classification in Adversarial Environments,” *Proc. Int'l Symp. Distributed Autonomous Robotic Systems (DARS),* pp. 371 380, 2000.

[32] E. Fredkin, “Trie Memory,” *Comm. ACM,* vol. 3, no. 9, pp. 490 499, 1960.

[33] J.A. Iglesias, A. Ledezma, and A. Sanchis, “Sequence Classification Using Statistical Pattern Recognition,” *Proc. Int'l Conf. Intelligent Data Analysis (IDA),* pp. 207 218, 2007.

[34] G.A. Kaminka, M. Fidanboylu, A. Chang, and M.M. Veloso, “Learning the Sequential Coordinated Behavior of Teams from Observations,” *Proc. RoboCup Symp.,* pp. 111 125, 2002.

[35] J.A. Iglesias, A. Ledezma, and A. Sanchis, “A Comparing Method of Two Team Behaviours in the Simulation Coach Competition,” *Proc. Int'l Conf. Modeling Decisions for Artificial Intelligence (MDAI),* pp. 117 128, 2006.

[36] R. Agrawal and R. Srikant, “Mining Sequential Patterns,” *Proc. Int'l Conf. Data Eng.,* pp. 3 14, 1995.

[37] P. Angelov and D. Filev, “An Approach to Online Identification of Takagi Sugeno Fuzzy Models,” *IEEE Trans. Systems, Man, and Cybernetics, Part B,* vol. 34, no. 1, pp. 484 498, Feb. 2004.

[38] P. Angelov, X. Zhou, and F. Klawonn, “Evolving Fuzzy Rule Based Classifiers,” *Proc. IEEE Symp. Computational Intelligence in Image and Signal Processing (CIISP '07),* pp. 220 225, 2007.

[39] X. Zhou and P. Angelov, “Autonomous Visual Self Localization in Completely Unknown Environment Using Evolving Fuzzy Rule Based Classifier,” *Proc. IEEE Symp. Computational Intelligence in Security and Defense Applications (CISDA),* pp. 131 138, 2007.

[40] P. Angelov and D. Filev, “Simpl ets: A Simplified Method for Learning Evolving Takagi Sugeno Fuzzy Models,” *Proc. IEEE Int'l Conf. Fuzzy Systems (IEEE FUZZ),* pp. 1068 1073, 2005.

[41] S. Greenberg, “Using Unix: Collected Traces of 168 Users,” master's thesis, Dept. of Computer Science, Univ. of Calgary, Alberta, Canada, 1988.

[42] J. Quinlan, “Data Mining Tools See5 and c5.0,” http:// www.rulequest.com/see5 info.html, 2003.

[43] J.R. Quinlan, *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[44] R.O. Duda and P.E. Hart, *Pattern Classification and Scene Analysis.* John Wiley & Sons, 1973.

[45] G. John and P. Langley, “Estimating Continuous Distributions in Bayesian Classifiers,” *Proc. Conf. Uncertainty in Artificial Intelligence,* pp. 338 345, 1995.

[46] T. Cover and P. Hart, “Nearest Neighbor Pattern Classification,” *IEEE Trans. Information Theory,* vol. 13, no. 1, pp. 21 27, Jan. 1967.

[47] Y. Freund and R.E. Schapire, “A Decision Theoretic Generalization of On Line Learning and an Application to Boosting,” *J. Computer and System Sciences,* vol. 55, no. 1, pp. 119 139, 1997.

[48] J.H. Morra, Z. Tu, L.G. Apostolova, A. Green, A.W. Toga, and P.M. Thompson, “Comparison of Adaboost and Support Vector Machines for Detecting Alzheimer's Disease through Automated Hippocampal Segmentation,” *IEEE Trans. Medical Imaging,* vol. 29, no. 1, pp. 30 43, Jan. 2010.

[49] J. Platt, “Machines Using Sequential Minimal Optimization,” *Advances in Kernel Methods Support Vector Learning,* B. Schoelkopf, C. Burges, and A. Smola, eds., MIT Press, 1998.

[50] *Self Organizing Maps,* T. Kohonen, M.R. Schroeder and T.S. Huang, eds. Springer Verlag, 2001.