Modelling Evolving User Behaviours

Jose A. Iglesias, Plamen Angelov, Agapito Ledezma and Araceli Sanchis

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 behaviour profile of a computer user is presented. In this case, a computer user behaviour is represented as the sequence of the commands (s)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 behaviour. Also, because of 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 on-line scheme. We also develop further the recursive formula of the potential of a data point to become a cluster centre 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 behaviour modelling where it can be represented as a sequence of actions and events. It has been evaluated on several real data streams.**

I. Introduction

RECOGNIZING the behaviour of others in real-time is a significant aspect in many different environments. Specifically, the recognition of computer users can be very beneficial for assisting them, predicting their future actions or detecting masqueraders. This recognition needs the creation of a *user profile* that contains information that characterizes the usage behaviour of a computer user. The construction of effective user profiles is a difficult problem for different aspects: human behaviour is usually 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.

In recent years, significant work has been carried out for profiling users; however, most of the user profiles do not change according to the environment and new goals of the user. In this research, it is proposed an adaptive approach for creating behaviour profiles and recognizing users. We call this approach *EvABCD* (*Evolving Agent Behaviour Classification based on Distributions of relevant events*). It is based on representing the behaviour of an *agent* (user) as an adaptive distribution of her/his relevant atomic behaviours in an evolving way.

For evaluating *EvABCD*, the UNIX operating system environment is used. Some research on this environment [1], [2] 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 from a sequence of commands and classifies a new user 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 of we use an evolving classifier, it is constantly learning and adapting the existing classifier structure to accommodate the newly observed emerging behaviours. Once a user is classified, relevant actions can be done, however this task is not addressed in this paper.

The creation of UNIX user profiles from a sequence of UNIX commands should consider the sequentiality 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 behaviour classification; if we do not know the features that influence the behaviour 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 behaviours of the user because the user behavior evolves 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 behaviour itself may also evolve (we assume that each behaviour is described by one or more fuzzy rules). Therefore, we use an evolving (fuzzy) system that allows for the user behaviours to be dynamic, to evolve.

This paper is organized as follows: Section 2 provides a brief overview of the background and related work relevant to this research. Our approach (*EvABCD*) is explained in detail in section 3. Section 4 describes the construction of the user behaviour 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.

II. Background and Related Work

Different methods have been used to find out relevant information under the human behaviour in many different areas: Macedo et al. [3] propose a system (*WebMemex*) that provides recommended information based on the captured history of navigation from a list of known users. Pepyne et al. [4] describe a method using queuing theory and logistic regression modeling methods for profiling computer users based on simple temporal aspects of their behaviour. Gody and Amandi [5] present a technique to generate readable user

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

Plamen Angelov is with the Department of Communication Systems, InfoLab21, Lancaster University, UK. E-mail:p.angelov@lancaster.ac.uk

Jose A. Iglesias, Agapito Ledezma and Araceli Sanchis are with the CAOS Group, Carlos III University of Madrid, Spain. E-mails:{jiglesia,ledezma,masm}@inf.uc3m.es.

profiles that accurately capture interests by observing their behaviour on the Web.

In the computer intrusion detection problem, Coull et al. [6] propose an effective algorithm that uses pair-wise sequence alignment to characterize similarity between sequences of commands. Schonlau et al. [1] investigate a number of statistical approaches for detecting masqueraders. Angelov and Zhou propose in [7] to use evolving fuzzy classifiers for computer intrusion detection.

Although there is a lot of work focusing on user profiling in a specific environment, it is not clear that they can be transferred to other environments. However, *EvABCD* can be used in any domain in which a user behaviour can be represented as a sequence of actions or events. Because sequences are very relevant 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 unlabelled sequential data that discover meaningful patterns of sequential behaviour 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.

It should be emphasized that all of the above approaches ignore the fact that user behaviours can change or their description can change and evolve. To the best of our knowledge this is the first publication where user behaviour is considered, treated and modelled as a dynamic and evolving phenomenon. This is the most important contribution of this paper.

III. The proposed Approach EvABCD

This section introduces the proposed approach for automatic clustering, classifier design and classification of the behaviour profiles of users. Although the proposed approach can be applied for any behaviour represented by a sequence of events, we detail it using the UNIX Commands environment. Therefore, a behaviour profile is created from the commands a UNIX user types during a period of time. In addition, a novel evolving user behaviour classifier based on *Evolving Fuzzy Systems* is presented which takes into account the fact that the behaviour of any user is not fixed, but is rather changing, evolving.

In order to classify an observed behaviour, our approach, as many other agent modeling methods [11] creates a library which contains the different expected behaviours. However, in our approach this library is not a pre-fixed one, but is evolving, learning from the observations of the users real behaviours and moreover it starts to be filled in 'from scratch' by assigning temporarily to the library the first observed user as a prototype. That means that the library is continuously changing, evolving influenced by the changing user behaviours observed in the environment. This evolving library (called *evolving-profile-library (EPLib)*) is created and updated by the *Evolving user classifier*.

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

1) **Learning and Update of the Classifier:** This action involves in itself two sub-actions:
   a) **Update User Behaviour Profiles**. This sub-action analyzes the sequences of commands typed by different UNIX users (data stream) on-line and creates and updates the corresponding profiles. This process is detailed in Section 4.
   b) **Evolving the Classifier**. This sub-action includes on-line learning and update of the classifier, including the potential of each behaviour to be a prototype, and update of the *evolving-profile-library (EPLib)*. This whole process is explained in Section 5.
2) **User Classification:** The user profiles created in the previous action are associated to 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.

IV. Construction of the User Behaviour Profile

In order to construct a user behaviour profile in on-line mode from a data stream we have to extract an ordered sequence of recognized atomic behaviours. For this purpose we consider that the atomic behaviours of a user are represented by the commands (s)he types in.

The commands typed by an agent are inherently sequential, and this sequentiality is considered in the modeling process (when a user types a command, it usually depends on the previous typed commands and it is related to the following commands). According to this aspect, in order to get the most representative set of subsequences from a sequence, we propose the use of a *trie* data structure [12]. This *trie* data structure was also used in [13] and in [14] where a team behaviour was learned as well as in [15] to classify the behaviour patterns of a *RoboCup* soccer simulation team.

The construction of a user profile from a single sequence of commands is done by a three steps process: 1. Segmentation of the sequence of commands, 2. Storage of the subsequences in a *trie*, and 3. Creation of the user profile. These steps are detailed in the following 3 subsections.

In order to clarify this process for creating, lets consider the following sequence as an example: $\{ls \rightarrow date \rightarrow ls \rightarrow date \rightarrow cat\}$.

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

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

B. Storage of the subsequences in a trie

The subsequences of commands are stored in a *trie* in a way that all possible subsequences are accessible and explicitly represented. In a *trie*, a node represents a command, and its *children* represent the commands that follow it. Also, each node keeps track of the number of times a command has been inserted on to it. When a new subsequence is inserted into the *trie*, existing nodes of the *trie* are modified and/or new nodes are created. Moreover, 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* (Figure 1 a). Each node is labeled with the number 1 which indicates that the command has been inserted in the node once (in Figure 1, this number is enclosed in square brackets). Then, the suffixes of the subsequence ($\{date \rightarrow ls\}$ and $\{ls\}$) are also inserted (Figure 1 b). Finally, after inserting the 3 subsequences and its corresponding suffixes, the completed *trie* is obtained (Figure 1 c).

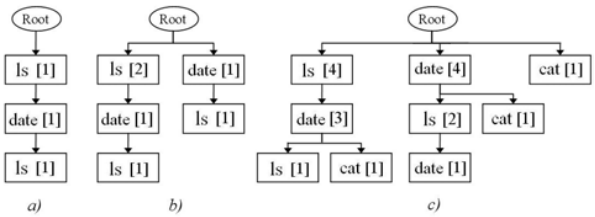

Fig. 1. Steps of creating an example trie.

C. Creation of the user profile

Once the *trie* is created, the subsequences that characterize the user profile and its relevance are obtained by traversing the *trie* (where a subsequence is a path from the *root* node to any other node of the *trie*). For this purpose, **frequency-based methods** can be used. In particular, in *EvABCD*, to evaluate the relevance of a subsequence, its relative frequency or support [16] 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* to the total number of subsequences of equal size inserted. Because of the frequency of a command is always higher than the frequency of two consecutive commands, it is important to calculate the support according to the subsequences of equal size.

Thus, in this step the *trie* can be transformed into a set of subsequences labeled with its support value. In *EvABCD* this set of subsequences is represented as a distribution of relevant subsequences.

In the previous example, the *trie* consists of 9 nodes; therefore, the profile consists of 9 different subsequences which are labeled with its support. Figure 2 shows the distribution of these subsequences.

Once a user behaviour profile has been created, it is classified by the classifier as explained in the next section.

V. 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 (a distribution represents a user behaviour and has been calculated as explained in the previous subsection). 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 behaviour which is one of the prototypes of the *evolving-profile-library (EPLib)*. **The prototypes are not fixed and evolve** (dynamically change) taking into account the new information collected on-line from the data stream - this is what makes the classifier *Evolving*. The number of these prototypes is not pre-fixed but it depends on the homogeneity of the observed behaviours. The whole classifier is detailed in the following sub-sections.

A. User behaviour representation

EvABC receives observations in real-time from the environment to analyze. In our case, these observations are the commands typed by a user. These observations are converted into the corresponding distribution on-line. In order to classify UNIX user behaviours 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 observed. It means that we should know all the different subsequences of the environment *a priori*. However, these subsequences could be unknown and the creation of this data space from the beginning is not efficient. For this reason, in *EvABCD* the dimensions of the data space also evolves (is incrementally growing) according to the different subsequences that are represented in it.

Figure 3 explains graphically this novel idea. In this example, the distribution of the first user consists of 5 subsequences of commands (*ls, date, ls-date, cat* and *vi*), therefore we need a 5 dimensional data space to represent this distribution (each different subsequence is represented by one dimension). If we consider the second user, we can see that 2 of the 5 previous subsequences have not been typed by this user (*ls-date* and *cat*). It is important to consider that this value is not available so it can not be represented by the number 0. Also, there are 2 new subsequences (*emacs* and *rm*) so the representation of this value in the same data space needs to increase the dimensionality of the data space from 5 to 7. To sum up, the dimensions of the data space represent the different subsequences typed by the users and they

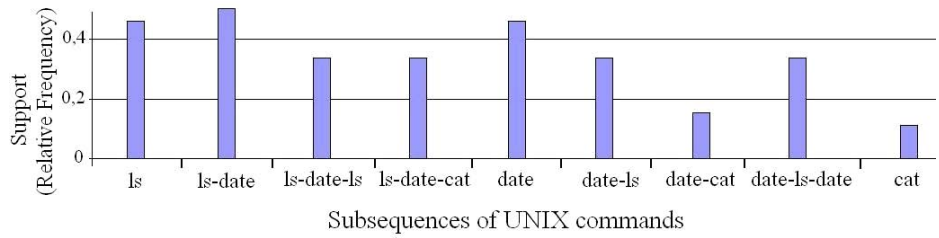

Fig. 2. Distribution of subsequences.

will increase according to the different new subsequences obtained.

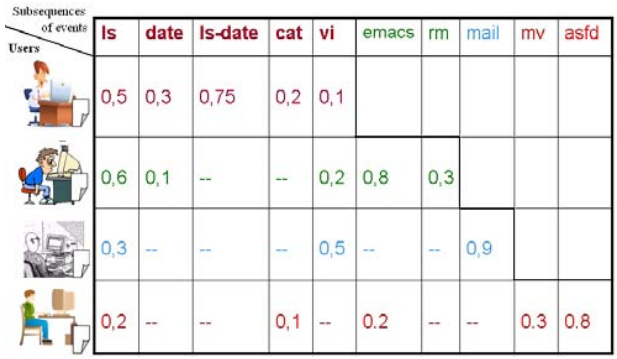

Fig. 3. Distributions of subsequences of events in evolving systems - Example

B. Calculating the potential of a data sample

As in [7], a prototype is a data sample (a behaviour represented by a distribution of subsequences of events) that represents several samples and has been selected from the available data by an unsupervised learning. The classifier is initialized with the first data sample (which is stored in *EPLib*). After this, each data sample is classified to one of the prototypes (classes) defined in the classifier. Then, based on the *potential* of the new data sample to become a prototype [17], it could form a new prototype or replace an existing one.

The potential of a data sample (z_k) is calculated by the equation (1) which represents a function of the accumulated distance between a sample and all the other samples in the data space *per class*. The result of this function represents the *density* of the data that surrounds a certain data sample.

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

where z_k denotes the k^{th} data sample and *distance* represents the distance between two samples in the data space.

In [18] the potential is calculated using the euclidean distance and in [7] it is calculated using the cosine distance. Cosine distance has the advantage that it tolerates different samples to have different number of attributes (subsequences labeled with its support value). Cosine distance also tolerates if 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 behaviours.

$$cosDist(z_k, z_p) = 1 - \frac{\sum_{j=1}^{n} z_{kj} z_{pj}}{\sqrt{\sum_{j=1}^{n} z_{kj}^2 \sum_{j=1}^{n} z_{pj}^2}} \tag{2}$$

where z_k and z_p represent the two samples to measure its distance and n represents the number of different attributes (subsequences) in both samples.

Note that the expression in the equation (1) requires all the accumulated data which contradicts to the requirement for real-time and on-line application needed in the proposed approach. For this reason, a recursive expression of the potential in which is not needed to store the history of all the data was developed in [17] [18] using euclidean distance and in [7] using cosine distance.

As it is explained in the Appendix, to get recursively the value of the potential of a sample using the equation (1) is necessary to calculate *nxn* 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}). As it is detailed in the Appendix, the result of this derivation is:

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

However, in our particular application of user behaviour modelling the data represent support values and are thus positive. Thus, to simplify the expression (1) one can use simply the distance instead of square of the distance. For this reason, we use equation (4) instead of (1).

$$P(z_k) = \frac{1}{1 + \frac{\sum_{i=1}^{k-1} cosDist(x_k, x_i)}{k-1}} \quad (4)$$

Using the equation (4), we develop a recursive expression similar to the recursive expressions derived in [18] and [7]. This formula is as follows:

$$P_k(z_k) = \frac{1}{2 - \frac{1}{k-1}\frac{1}{\sqrt{\sum_{j=1}^{n}(z_k^j)^2}} B_k}; k = 2, 3, ...; P_1(z_1) = 1$$
$$where \quad B_k = \sum_{j=1}^{n} z_k^j b_k^j \; ; \; b_k^j = b_{(k-1)}^j + \sqrt{\frac{(z_k^j)^2}{\sum_{l=1}^{n}(z_k^l)^2}}$$
$$and \quad b_1^j = \sqrt{\frac{(z_1^j)^2}{\sum_{l=1}^{n}(z_1^l)^2}} \; ; \; j = [1, n+1] \quad (5)$$

C. Creating new prototypes

The proposed evolving user behaviour classifier *EvABCD* can start '*from scratch*' (without prototypes in the library) in a similar manner as *eClass* evolving fuzzy rule-based classifier proposed in [18], used in [19] for robotics and further developed in [7]. The potential of each new data sample (user behaviour represented by a distribution of subsequences) is calculated *recursively* and the potential of the other prototypes is updated. After that, 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 equation (6).

$$\exists i, \; i = [1, NumPrototypes]; \; P(z_k) > P(Prot_i) \quad (6)$$

Thus, if the new data sample is not relevant, the overall structure of the classifier is not changed. Otherwise, the classifier evolves by adding new prototypes which represent a part of the observed data samples.

D. 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 [7]. By *well* we mean that the value of the membership function that describes the closeness to the prototype is a Gaussian bell function due to its generalization capabilities (equation (7)):

$$\exists i, \; i = [1, NumPrototypes]; \; \mu_i(z_k) > e^{-1} \quad (7)$$

For this reason, we calculate the membership function between a data sample and a prototype which is defined as (8):

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

where cosDist($z_k, Prot_i$) represents the cosine distance between a data sample (z_k) and the i^{th} prototype ($Prot_i$); σ_i represents the spread of the membership function, which also represents the radius of the zone of influence of the prototype. This spread is determined based on the scatter [20] of the data. The equation to get the spread of the k^{th} data sample is defined in (9):

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

where *k* is the number of data samples inserted in the **same class**; *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 (as shown in [18]) recursively by equation (10):

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

E. Classification Method

In order to classify a new data sample, we compare it with all the prototypes stored in the *evolving-profile-library (EPLib)*. This comparison is done using cosine distance and the smallest distance determines the closest similarity. This aspect is considered in equation (11).

$$Class(x_z) = Class(Prot^*);$$
$$Prot^* = MIN_{i=1}^{NumProt}(cosDist(x_{Prototype_i}, x_z)) \quad (11)$$

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 are acceptable for real-time applications (in order of milliseconds per data sample).

F. Structure of the EvABCD

Once explained the different parts in which the proposed classifier can be divided, we show the structure of this classifier. The input of the proposed classifier is a behaviour stream, where each behaviour is represented as a distribution of subsequences of events. Therefore, once the distribution has been created from the stream, it is processed by the classifier. The structure of the proposed classifier is as follows:

1) **Classify** the new sample in a class (represented by a prototype) using (11).
2) **Calculate the *potential*** of the new data sample to be a prototype using the recursive formula (5).
3) **Update all the prototypes** considering the new data sample (using (5)). It is done because the *situation* of the data space changes with the insertion of each new data sample.
4) **Insert** the new data sample as a new prototype **if needed** (if (6) holds).

5) **Remove** any prototype **if needed** (if (7) holds).

Therefore, as we can see, the classifier does not need to be configured (the classifier can start '*from scratch*') according to the environment where it is used. Also, the relevant information of the obtained samples is necessary to update the library, but it is not necessary to store all the information in it.

G. Supervised and unsupervised learning

The proposed classifier can be used for both supervised and unsupervised learning.

- **Supervised learning:** The data samples that are observed can have a label assigned to them *a priori*. In this case, a specific class (label) is represented by several prototypes (the number of prototypes depends on how heterogeneous are the samples of the same class). This techniques is used for example in eClass1 [18] and [7];
- **Unsupervised learning:** The observed data samples do not have labels. In this case, the classes are created based on the prototypes and, thus, any prototype represents a different class (label). Such technique is used for example in eClass0 [18] and [7] which is a clustering-based classification.

VI. 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 4 different groups. Therefore, in these experiments we will use ***supervised learning***.

A. Data Set

For evaluating *EvABCD* in the UNIX environment, we have used the command-line data collected by Greenberg [21] using UNIX *csh* command interpreter. In these data, four target groups were identified, representing a total of **168** male and female **users** with a wide cross-section of computer experience and needs. Salient features, the size of the data stream (the number of people observed) and command lines 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. Sample: 55 Users and 77423 command lines.
- *Experienced Programmers*: In this group, the members were senior computer science undergraduates, expected to have a fair knowledge of the Unix environment. Sample: 36 Users and 74906 command lines.
- *Computer Scientist*: This group had varying experience with Unix, although all were experts with computers. Sample: Sample: 52 Users and 125691 command lines.
- *Non-programmers*: Document preparation was the dominant activity of the members of this group. Knowledge of Unix was the minimum necessary to get the job done. Sample: 25 Users and 25608 command lines.

B. Experimental Design

In order to measure the performance of the proposed classifier using the above data, the well-established technique of cross-validation is used. For this research, **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 the proposed *EvABCD* does not need necessarily to work in this mode. This is done mainly in order to have comparable results with the established off-line techniques. In reality the proposed *EvABCD* classifier can work on a per sample and per user basis.

The number of UNIX commands analyzed per user is very relevant for the result of the classification. Using *EvABCD* in a real application, after a user has typed a particular number of commands, its behaviour can be classified and the evolving behaviour library updated. However, in order to use all the data we have, in this experiment all the commands the user has typed during a long period of time are used. For this resason, a distribution (which represent a behaviour) is represented by a very large number of subsequences. And if the number of users increases, the number of different subsequences increases, too.

In the phase of behaviour model creation, the length of the subsequences in which the original sequence is segmented (used for creating the *trie*) is a relevant parameter: using longer subsequences, the time consumed for creating the *trie* and the number of relevant subsequences in the corresponding distribution increase drastically. In the experiments presented in this paper, the ***subsequence length*** value was selected to be **3**.

C. Results

Although the sequence length is small (3 commands), the number of commands typed per user is large; thus, the number of different subsequences of commands created per user is very large. The number of diffeerent subsequences is shown per group as follows; Novice Programmers: 25614, Experienced Programmers: 43049, Computer Scientists: 66490, Non-Programmers: 10572. Also, the number of different subsequences of commands typed by the 168 users is **135317**.

According to this data, after applying *EvABCD* using the explained experimental design, the percentage of users correctly **classified into its corresponding group** is: **100%** on validation data! Therefore, this result shows that the proposed classifier works excellent in this kind of environments.

The number of prototypes created per group is important, too. As we have used 10-fold cross validation, the number of different prototypes created in each of the 10 runs is shown in table VI-C. This number varies depending on the heterogeneity of the data.

The result obtained in this experiment shows that the proposed classifier can be very useful to classify user behaviours in a dynamic environment for the example of the UNIX user profiles with a great amount of data (in this case, commands

TABLE I

EVABCD: NUMBER OF PROTOTYPES CREATED PER GROUP USING 10-FOLD CROSS-VALIDATION

	Prototypes in each of the 10 runs									
Group	1	2	3	4	5	6	7	8	9	10
Novice Progr.	3	2	7	2	2	2	3	4	2	2
Exp. Progr.	2	3	3	2	2	2	2	2	2	2
Comp. Scientists	2	2	1	2	1	1	1	1	3	3
Non-Progr.	2	2	1	2	2	2	2	2	2	2

per user). In order to compare these results we consider two well established classifiers - the *algorithm* ***C4.5*** used to generate a decision tree and the *k-nearest neighbor algorithm* ***(k-NN)*** used to classify objects based on closest training examples in the feature space. However, this comparison could not be done in this experiment because of the big amount of attributes per sample to consider. This obstructed both *algorithm C4.5* and *k-nearest neighbor algorithm (k-NN)* could not run because of the memory overload. Note, that our proposed approach does not need to store entire data stream in the memory and disregards any sample after being used. Thus, based on the experiment size and dimensions as described so far, the proposed approach *EvABCD* was the only working alternative.

However, in order to make a comparison, we reduced the number of subsequences of commands per user using its support value. In this case, we consider that the subsequences with a higher support are more *relevant*. The percentage of subsequences reduced is very high and only around the 3% of the initial data were used. It means that the total number of different subsequences considered was in this reduced dimensionality second experiment equal to **3531**. In this reduced dimension experiment, again the proposed *EvABCD* evolving classifier outperformed the well established off-line classifiers and the results are tabulated in table VI-C.

TABLE II

COMPARATIVE RESULTS

Classifier	Rate of unknown users correctly classified
EvABCD	**81,54 %**
C4.5	73,80 %
3-Nearest Neighbor	44,64 %

Note that this reduction in the dimensionality of the experiment as well as the 10-fold cross-validation is needed only for the sake of comparison. Inevitably, the reduction of the raw data leads to a lower performance, but nevertheless, the proposed evolving classifier *EvABCD* outperforms significantly the well established off-line classifiers. Moreover, it is computationally more simple and efficient as it is recursive and one pass (works on a per sample basis).

VII. CONCLUSIONS

In this paper we propose a generic approach to user behaviours modelling and consider the specific example of users of Unix computer command sequences. The proposed evolving classifier *EvABCD* is one pass, non-iterative, recursive and therefore, computationally very efficient and fast.

The test results with a data sequence of 168 real users of Unix demonstrates that it is also able to outperform significantly the well established off-line classifiers in terms of correct classification on validation data. Although it is not addressed in this paper, the proposed method can be also used to monitor, analyze and detect abnormalities based on a time varying pattern 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

In this appendix the expression of the potential is transformed in a recursive expression in which the potential 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 (equation (1)) represented by a vector of elements and the distance cosine expression (2).

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

where z_k denotes the k^{th} sample inserted in the space data. Each sample is represented by a set of values represented by a number: the i^{th} attribute (element) of the k_z 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 equation (12) which is named as $den.P(z_k)$.

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

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

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

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 attributes value of the sample. Therefore, deriving (13), we obtain:

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

In order to obtain an expression for the potential from (14), we rename as follows:

$$
\begin{aligned}
&den.P_k(z_k) = 2 + [\frac{1}{h(k-1)}[[-2B_K] + [\frac{1}{h}D_k]]] \\
&where: B_k = \sum_{i=1}^{k-1} \frac{f_i}{g_i};\ D_k = \sum_{i=1}^{k-1} (\frac{f_i}{g_i})^2
\end{aligned} \tag{15}
$$

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

$$
B_k = \sum_{i=1}^{k-1} \frac{\sum_{j=1}^{n} z_k^j z_i^j}{\sqrt{\sum_{j=1}^{n} (z_i^j)^2}} = \sum_{j=1}^{n} z_k^j \sum_{i=1}^{k-1} \sqrt{\frac{(z_i^j)^2}{\sum_{l=1}^{n} (z_i^l)^2}} \tag{16}
$$

If we define b_k^i, each attribute of the sample b, we get:

$$
b_k^j = \sum_{i=1}^{n} \sqrt{\frac{(z_i^j)^2}{\sum_{l=1}^{n} (z_i^l)^2}} \tag{17}
$$

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

$$
\begin{aligned}
&B_k = \sum_{j=1}^{n} z_{kj} b_k^j\ ;\ b_k^j = b_{(k-1)}^j + \sqrt{\frac{(z_k^j)^2}{\sum_{l=1}^{n} (z_k^l)^2}} \\
&b_1^j = \sqrt{\frac{(z_1^j)^2}{\sum_{l=1}^{n} (z_1^l)^2}}
\end{aligned} \tag{18}
$$

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

$$
\begin{aligned}
&D_k = \sum_{i=1}^{k-1} (\frac{\sum_{j=1}^{n} z_k^j z_i^j}{\sqrt{\sum_{j=1}^{n} (z_i^j)^2}})^2 \\
&D_k = \sum_{j=1}^{n} z_k^j \sum_{p=1}^{n} z_k^p \sum_{i=1}^{k-1} z_k^{ij} z_k^{ip} \frac{1}{\sum_{l=1}^{n} (z_i^l)^2}
\end{aligned} \tag{19}
$$

If we define d_k^i, each attribute of the sample d, we get:

$$
d_k^{jp} = \sum_{i=1}^{k-1} z_k^{ij} z_k^{ip} \frac{1}{\sum_{l=1}^{n} (z_i^l)^2} \tag{20}
$$

Therefore:

$$
\begin{aligned}
&D_k = \sum_{j=1}^{n} z_k^j \sum_{p=1}^{n} z_k^p d_k^{jp};\ d_k^{jp} = d_{(k-1)}^{jp} + \frac{z_k^j z_k^p}{\sum_{l=1}^{n} (z_k^l)^2}; \\
&d_1^{1j} = \frac{z_1^j z_1^1}{\sum_{l=1}^{n} (z_1^l)^2}\ ; j = [1, n+1]
\end{aligned} \tag{21}
$$

Finally:

$$
\begin{aligned}
&P_k(z_k) = \frac{1}{2 + [\frac{1}{h(k-1)}[[-2B_K] + [\frac{1}{h}D_k]]]} \\
&k = 2, 3...; P_1(z_1) = 1
\end{aligned} \tag{22}
$$

where B_k is obtained as in (18), and D_k is described in (21).

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*x*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}).

REFERENCES

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

[2] R. A. Maxion and T. N. Townsend, "Masquerade detection using truncated command lines," in *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 219–228.

[3] A. A. Macedo, K. N. Truong, J. A. Camacho-Guerrero, and M. da GraÇa Pimentel, "Automatically sharing web experiences through a hyperdocument recommender system," in *HYPERTEXT 2003*. New York, NY, USA: ACM, 2003, pp. 48–56.

[4] D. L. Pepyne, J. Hu, and W. Gong, "User profiling for computer security," in *Proc. American Control Conference*, 2004, pp. 982–987.

[5] D. Godoy and A. Amandi, "User profiling for web page filtering," *IEEE Internet Computing*, vol. 9, no. 4, pp. 56–64, 2005.

[6] S. E. Coull, J. W. Branch, B. K. Szymanski, and E. Breimer, "Intrusion detection: A bioinformatics approach." in *ACSAC*, 2003, pp. 24–33.

[7] P. Angelov and X. Zhou, "Evolving fuzzy rule-based classifiers from data streams," *IEEE Transactions on Fuzzy Systems: Special issue on Evolving Fuzzy Systems*, vol. 16, no. 6, p. to appear, 2008.

[8] J. Anderson, *Learning and Memory: An Integrated Approach*. New York: 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," in *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 1998, pp. 150–158.

[11] P. Riley and M. M. Veloso, "On behavior classification in adversarial environments," in *DARS*, pp. 371–380.

[12] E. Fredkin, "Trie memory," *Comm. A.C.M.*, vol. 3, no. 9, pp. 490–499, Sept. 1960.

[13] J. A. Iglesias, A. Ledezma, and A. Sanchs, "Sequence classification using statistical pattern recognition." in *IDA*, ser. LNCS, vol. 4723. Springer, 2007, pp. 207–218.

[14] G. A. Kaminka, M. Fidanboylu, A. Chang, and M. M. Veloso, "Learning the sequential coordinated behavior of teams from observations," in *RoboCup*, ser. Lecture Notes in Computer Science, vol. 2752. Springer, 2002, pp. 111–125.

[15] J. A. Iglesias, A. Ledezma, and A. Sanchis, "A comparing method of two team behaviours in the simulation coach competition," in *MDAI*, ser. LNCS, vol. 3885. Springer, 2006, pp. 117–128.

[16] R. Agrawal and R. Srikant, "Mining sequential patterns," in *International Conference on Data Engineering*, Taipei, Taiwan, 1995, pp. 3–14.

[17] P. Angelov and D. Filev, "An approach to online identification of takagi-sugeno fuzzy models," *Systems, Man, and Cybernetics, Part B, IEEE Transactions on*, vol. 34, no. 1, pp. 484–498, Feb. 2004.

[18] P. Angelov, X. Zhou, and F. Klawonn, "Evolving fuzzy rule-based classifiers," *Computational Intelligence in Image and Signal Processing, 2007. CIISP 2007. IEEE Symposium on*, pp. 220–225, April 2007.

[19] X. Zhou and P. Angelov, "Autonomous visual self-localization in completely unknown environment using evolving fuzzy rule-based classifier," *Computational Intelligence in Security and Defense Applications, 2007. CISDA 2007. IEEE Symp.*, pp. 131–138, April 2007.

[20] P. Angelov and D. Filev, "Simpl_ets: a simplified method for learning evolving takagi-sugeno fuzzy models," *Fuzzy Systems, 2005. FUZZ '05. The 14th IEEE International Conference on*, pp. 1068–1073, May 2005.

[21] S. Greenberg, "Using unix: Collected traces of 168 users," Master's thesis, Department of Computer Science, University of Calgary, Alberta, Canada, 1988.