A Cheating Detection Framework for Unreal Tournament III: a Machine Learning Approach

Luca Galli, Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi

Abstract— **Cheating reportedly affects most of the multiplayer online games and might easily jeopardize the game experience by providing an unfair competitive advantage to one player over the others. Accordingly, several efforts have been made in the past years to find reliable and scalable approaches to solve this problem. Unfortunately, cheating behaviors are rather difficult to detect and existing approaches generally require human supervision. In this work we introduce a novel framework to automatically detect cheating behaviors in Unreal Tournament III by exploiting supervised learning techniques. Our framework consists of three main components: (i) an extended game-server responsible for collecting the game data; (ii) a processing backend in charge of preprocessing data and detecting the cheating behaviors; (iii) an analysis frontend. We validated our framework with an experimental analysis which involved three human players, three game maps and five different supervised learning techniques, i.e., decision trees, Naive Bayes, random forest, neural networks, support vector machines. The results show that all the supervised learning techniques are able to classify correctly almost 90% of the test examples.**

I. Introduction

In the context of on-line gaming, cheating refers to using artificial systems to gain a competitive advantage over the other players. Unluckily, cheating is a widespread phenomenon among on-line multiplayer games (especially among first person shooters) that affects negatively the game experience of *honest* players. Accordingly, several efforts have been made in the past years to find reliable and scalable solutions to this problem. Unfortunately, cheating behaviors are rather difficult to detect. Thus, the most successful and used approaches still heavily relies on the players' collaboration as well as on the monitoring activity of the game-server administrators. In this scenario, machine learning is yet a poorly exploited technology and, so far, only few works [1], [2], [3], [4] focused on the application of machine learning techniques to this problem.

In this work we introduce a methodology to automatically detect cheating behaviors in Unreal Tournament III, a slightly old but still popular first person shooter game. In particular, our approach exploits supervised learning techniques to learn a cheating detection model from a labeled dataset. To this purpose, we first developed a rather sophisticated and customizable cheating system and, then, we designed a framework consisting of three main components: (i) an extended game-server, (ii) a processing backend, and (iii) an analysis frontend. The extended game-server is responsible for gathering the more relevant game data and sending them to the backend. The backend is in charge of preprocessing the collected data and applying the learned models to detect cheating behaviors. Finally, the frontend provides a user interface to perform both a real-time monitoring of the ongoing games and an off-line analysis of past games.

Finally, we performed an experimental analysis to validate our framework. We collected a rather large dataset from several matches which involved three human players and three different game maps. Then, we applied five methods of supervised learning to build a model for the cheating detection and we assessed their performance on a test set. The reported results are very promising. All the five supervised learning techniques tested were able to classify correctly almost 90% of the cheating behaviors. In particular, the support vector machines and the Naive Bayes classifiers provided the highest accuracy with only one classification error out of 39 examples.

The paper is organized as follows. In Section II we provide a brief overview of the related works in the literature. We introduce Unreal Tournament and the cheating behaviors respectively in Section III and in Section IV. In Section V we describe our framework while in Section VI we report the experimental results. Finally, we draw our conclusions in Section VII.

II. Related Work

Several works in the literature (e.g., see [5], [6], [7], [8]) investigated protocols and system architectures to either detect or prevent cheating in on-line gaming. However, these works mainly focused on role-playing games (RPG) or real-time strategy games (RTS), while we focus on first person shooters.

Concerning the application of machine learning techniques to detect cheating behaviors in first person shooters, only few works have been introduced so far in the literature. Yeung et al. [1] applied a dynamic Bayesian network to detect aimbot in Cube, an open source FPS game. In the Bayesian model designed by the authors, the aiming accuracy of the player only depends on the player's movements, on the distance from the target, and on the presence of cheating behaviors.

Luca Galli, Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi are with the Politecnitco di Milano, Dipartimento di Elettronica e Informazione, Milano; email: `luca.galli@mail.polimi.it`, `{loiacono,cardamone,lanzi}@elet.polimi.it`,

Fig. 1. A screenshot of Unreal Tournament 3.

Although this work focused on detecting a rather simple cheating behavior, the reported results are very promising.

More recently, Chapel et al. in [4] proposed a rather general probabilistic framework to detect cheaters only on the basis of the outcomes of the game. Despite being very effective in principle, this approach requires a prior knowledge of the players' rankings in order to estimate the cheating probabilities.

Finally, Pao et al. [3] applied supervised learning techniques, namely k-nearest neighbors and support vector machines, to detect game bots in Quake II. Although related to cheating detection, detecting game bots is a rather different and slightly simpler problem. In fact, cheating systems exploit game bots only for very specific tasks (e.g., aiming at the opponents) and are therefore much more difficult to detect.

III. Unreal Tournament

Unreal Tournament III (UT3) is the last title of a very popular series of commercial *first person shooters*. It is based on the *Unreal Engine*, a quite popular game engine, used by several commercial games in the past years. Besides its impressive rendering capabilities (see Fig. 1), the Unreal Engine also exploits the NVDIA's PhysX engine, to accurately simulate the physics and the dynamics of the game world. UT3 was developed using the *Unreal Script* programming language, a java-like scripting language interpreted by the *Unreal Engine*. This two-tier architecture allows the decoupling between the development of the underlying engine and the actual gameplay: any modification to the engine does not require a change to the scripts implementing the gameplay.

Most of the scripts developed for UT3 are publicly available and can be modified to change the game behavior. Therefore, although the source code of the Unreal Engine is not available, the game itself is still highly customizable through using the Unreal Script language.

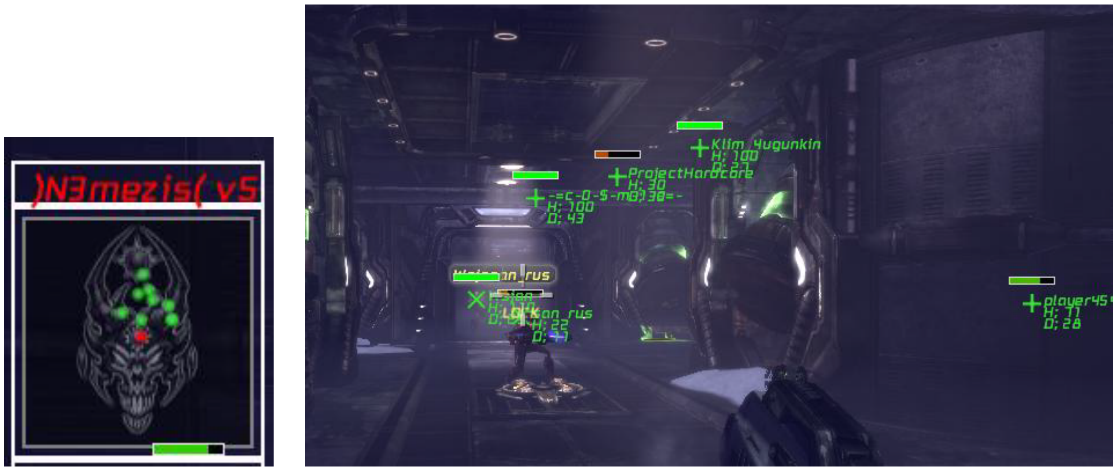
IV. Cheating in Unreal Tournament

Nowadays, several commercial cheating systems are available for all the popular first person shooters (e.g., see for example [9], [10], [11]). Cheating systems typically provide an *hacked* user interface to the player with artificial aids [12], [13], such as (i) automatic localization of the opponents' position, (ii) extended information about the status of the opponents (e.g., their health status, their current weapon, and their distance, etc.), (iii) aiming and firing support systems. Despite being generally customizable, commercial cheating systems are not provided with source code and, thus, cannot be neither extended nor modified. In particular, it might be difficult to automatically change and keep track of all the system settings during the game. In addition, the licensing policies of commercial cheating system might constrain the experimental setup used to collect the data. For these reasons we preferred to develop our own cheating systems. Following the documentation available on the dedicated on-line resources [12], [13], we developed a state-of-the-art cheating system which includes all the major features provided by commercial systems [9], [10], [11].

In the remainder of this section we provide a very brief description of the most important features included in our cheating system. All the described features might be enabled or disabled in-game.

Aiming and Firing Support System. Our cheating system features both an *aimbot* and a *triggerbot*. The aimbot is an automated support system in charge of assisting the player with the acquisition of targets. In particular, we implemented all the most advanced features available in commercial cheating systems, such as (i) a *slow aiming* feature, (ii) the *visibility angle*, and (iii) *bone aiming* feature. The slow aiming feature basically improves the player's experience by avoiding too fast changes of the aiming direction during the acquisition of targets. The visibility angle allows to constrain the target acquisition only to opponents in a specific range so that the cheating behavior would result more believable. The bone aiming feature allows to aim at a specific part of the opponents' body to fit different combat styles (e.g. aiming at the enemy's head when using a precision rifle, aiming at the enemy's feet when using a rocket launcher). The triggerbot is a rather simple mechanism that triggers a firing action as soon as a valid target falls under the crosshair of the player. Of course, the aimbot and the triggerbot can be used together resulting in a fully automated aiming and firing system.

Radar System. Most of the commercial cheating systems provide a radar, that is a system to show the position of all the other players in the map. Figure 2 shows two in-game pictures of our radar system, consisting of both a *2D radar* and a *3D radar*. The 2D radar (see Figure 2a) provides a small map of the gaming environment updated with the current position of all the players in the match. The 3D radar (see Figure 2b) provides a sort of augmented perspective,



(a) (b)

Fig. 2. In-game screenshots of the radar system implemented: (a) 2D radar and (b) 3D radar.

by rendering on the screen additional information about the location of the opponents and their status.

Warning System. We implemented a warning mechanism to signal the presence of opponents with the following three warning levels: (i) the *green* level, which signal that the player is not visible to any opponent; (ii) the *yellow* level, which signal that the player is visible to at least one opponent; (iii) the *red* level, which signal that at least one opponent is aiming at the player. During the game, the warning level is displayed at the top of the screen to keep the player informed about the current situation.

V. Our Framework

Our framework is composed of three components: (i) an extended game-server, (ii) a processing backend, and (iii) an analysis frontend. The extended game-server gathers the more relevant game data and sends them to the backend. After preprocessing the incoming data, the backend exploits the learned decision models to detect cheating behaviors. The frontend provides a user interface to perform either a real-time monitoring or an off-line analysis.

Keeping separated the game-server and the backend has two major advantages. First, it allows to setup a dedicated system for the backend without dealing with the limitations typically posed by the game-servers (e.g., in our case the game-server runs on a windows machine, while the backend runs on a linux machine). Second, it allows to perform real-time analysis without affecting the load-balancing of the game-server.

A. Extended Game-Server

Unreal Tournament III can be easily extended and customized through the Unreal Script technology. Thus, in this case, extending the game-server was straightforward and did not require any change to the source code of the engine. Our extended game-server automatically collects for each player all the game data that might be useful to identify cheating behaviors, such as (i) the coordinate reference systems and positioning in the map, (ii) aiming angles, (iii) current speed and acceleration, (iv) information about status (e.g., the health, the shield, the type of the weapon used, and whether the player is firing or jumping). Data is gathered on the game-server approximately 5 times per second and sent to the processing backend.

Table I illustrates the more relevant data collected at each timestep. In particular, Table I-A reports the data related to the player, while Table I-B reports the data related to the target opponent. The target opponent is defined as follows. When none of the opponents are visible, the target opponent is the closest one to the player. When at least one opponent is visible, the target opponent is the one the player is aiming more accurately, where the aiming accuracy is computed on the basis of the aiming direction and the position of the opponent.

B. Processing Backend

The backend is responsible for preprocessing the incoming data, building a decision model from a labeled dataset, and applying an existing decision model to classify (i.e., detect cheating behaviors) an unlabeled dataset. In our implementation, the backend was integrated with RapidMiner [14], a widely used data-mining framework with a large library of supervised methods.

Data Preprocessing. The game data logged by the server provide a very low-level description of the players' actions and need to be preprocessed to enable the detection of cheating behaviors with supervised learning methods. To this purpose, the collected data are clustered in *frames* corresponding approximately to 30 seconds of gameplay. Then, for each *frame*, a set of statistics is computed to summarize the behavior of the players within the

TABLE I

DESCRIPTION OF THE MOST RELEVANT DATA LOGGED DURING GAME: (A) ATTRIBUTES OF THE PLAYER AND (B) ATTRIBUTES OF THE TARGET OPPONENT.

Name	**Range**	**Description**
PlayerName	–	Nickname of the player.
HP	[0; +∞)	Health status, i.e., the value of *health points*, of the player.
shield	[0; +∞)	Shield status, i.e., the value of the *shield points*, of the player.
InstantWeapon	{True, False}	True if the player is currently using an instant weapon, i.e., a weapon that hits the target immediately as the player shoots.
Flying	{True, False}	True when the player is either jumping or falling.
Firing	{True, False}	True when the player is firing.
Speed	[0; +∞)	Speed of the player; values are in *Unreal Unit*.
Accel	[0; +∞)	Acceleration of the player; values are in *Unreal Unit*.
X	(-∞; +∞)	X position of the player on the map.
Y	(-∞; +∞)	Y position of the player on the map.
Z	(-∞; +∞)	Z position of the player on the map.
Pitch	(-∞; +∞)	Pitch angle of the player aiming direction; values are in *Unreal Unit*.
Yaw	(-∞; +∞)	Yaw angle of the player aiming direction; values are in *Unreal Unit*.
Roll	(-∞; +∞)	Roll angle of the player aiming direction; values are in *Unreal Unit*.
EnemyVisible	{True, False}	True when at least one opponent is visible to the player.
Timestamp	[0; +∞)	Time elapsed from the beginning of the match.
Cheating	–	A string with the complete parameters setting of the cheating system currently used.

(A)

Name	**Range**	**Description**
TargetName	–	Nickname of the target opponent.
TargetHP	[0; +∞)	Health status, i.e., the value of *health points*, of the *target opponent*.
AimingAccuracy	[0.0 ; 1.0]	Score of the aiming direction of the player with respect to the position of the target opponent: the higher is the score, the more accurate is the aiming direction.
Distance	[0; +∞)	Distance between the player and the target opponent.

(B)

frame. Table II reports the complete set of statistics computed for each *frame*. The statistics include the firing frequency of the player when the target opponent is visible (FiringOnVisible); the aiming accuracy of the player while firing (AimingScoreOnFiring); the type of weapons used by the player (avgInstantWeapon); the health status of the target enemy (avgΔHP$_{Target}$ and varΔHP$_{Target}$); the aiming actions of the player, i.e., the aiming angles selected, (avgPitch, varPitch, avgΔPitch, varΔPitch, avgYaw, varYaw, avgΔPitch, varΔYaw, avgRoll, varRoll, avgΔRoll, varΔRoll); the acceleration of the player (avgAccel, varAccel); the distance between the player and the target enemy (avgDist, varDist, avgDist$_X$, varDist$_X$, avgDist$_Y$, varDist$_Y$, avgDist$_Z$, varDist$_Z$, avgΔDist, varΔDist, avgΔDistInv, varΔDistInv). Finally, each *frame* is labeled as a cheater example if the player used the cheating system for at least 50% of the time concerned by the *frame*, i.e., for at least 15 seconds. Otherwise the example is labeled as an honest example.

Supervised Learning Methods. In the experiments reported in this paper, we considered five methods of supervised learning: (i) Naive Bayes classifiers [15], (ii) decision trees [16], (iii) Breiman's random forests [17], (iv) neural networks, trained using backpropagation [18], and (v) support vector machines [19]. Naive Bayes classifiers compute probabilistic classifiers based on the assumption that all the variables (the data attributes) are independent. Decision trees are a well-know approach which produce human-readable models represented as trees. Random forests [17] are ensembles of decision trees. They compute many decision trees from the same dataset, using randomly generated feature subsets and boostrapping, and generate a model by combining all the generated trees using voting. Neural Networks are a widely used [18] supervised learning method inspired by the early

TABLE II

DESCRIPTION OF THE STATISTICS COMPUTED FOR EACH FRAME DURING THE DATA PREPROCESSING.

Name	Description
FiringOnVisible	Fraction of time the player spent firing while the target opponent was visible.
AimingScoreOnFiring	Average score of the player's aiming accuracy while firing to the target opponent.
avgInstantWeapon	Fraction of time the player was using a instant weapon, i.e., a weapon that hits the target immediately as the player shoots.
avgΔHP$_{Target}$ (varΔHP$_{Target}$)	Average (variance) change of the target opponent's health points while the player is firing.
avgPitch (varPitch)	Average (variance) pitch angle of the player.
avgΔPitch (varΔPitch)	Average (variance) change of the pitch angle of the player.
avgYaw (varYaw)	Average (variance) yaw angle of the player.
avgΔYaw (varΔYaw)	Average (variance) change of the yaw angle of the player.
avgRoll (varRoll)	Average (variance) roll angle of the player.
avgΔRoll (varΔRoll)	Average (variance) change of the roll angle of the player.
avgAccel (varAccel)	Average (variance) acceleration of the player.
avgDist (varDist)	Average (variance) distance between the player and the target opponent.
avgDist$_X$ (varDist$_X$)	Average (variance) distance between the player and the target opponent along the X axis.
avgDist$_Y$ (varDist$_Y$)	Average (variance) distance between the player and the target opponent along the Y axis.
avgDist$_X$ (varDist$_Z$)	Average (variance) distance between the player and the target opponent along the Z axis.
avgΔDist (varΔDist)	Average (variance) change of the distance between the player and the target opponent.
avgΔDistInv (varΔDistInv)	Average (variance) change of the distance between the player and the target opponent when the target opponent is not visible.

models of sensory processing by the brain. Finally, support vector machines are a rather recent method of machine learning based on structural risk minimization that proved to be very successful in solving complex classification and regression problems.

C. Frontend

In this work we focused mainly on extending the game-server to collect the players' data and on implementing a fully integrated backend for processing the incoming data. Accordingly, to implement the frontend we followed a very straightforward approach. We developed a collection of script that might be easily executed from a terminal to (i) detect in real-time if there are some suspect cheating behaviors in an ongoing match, (ii) perform an offline analysis of a previously collected dataset from a past match, and (iii) build a new decision model from one or more previously collected datasets or update an existing one. Future works might include the development of a more sophisticated frontend with a GUI.

VI. EXPERIMENTAL ANALYSIS

To validate our framework we performed an experimental analysis which involved three different human players and was carried out on three different maps. In our analysis we compared the performance of different supervised learning techniques (i.e., support vector machines, neural networks, random forests, decision trees, and Naive Bayes) on a dataset including also a previously unseen human player and a previously unseen map.

A. Experimental Design

To build the training set, we collected the logs from 8 matches played on two different maps (namely, *Biohazard* and *Rising Sun*) and involving two human players (namely an expert player and a novice player). Each human player played two 20 minutes deathmatches for each map against an artificial intelligence controlled by the CPU. One of the match was played against a *novice* artificial intelligence (i.e., the easiest difficulty level of the game), the other against a *Godlike* artificial intelligence (i.e., the hardest difficulty level of the game). During each match the human players were allowed to switch on or to switch off the cheating system as they wish. However, they have been asked to play roughly the same time with the cheating system active and without. In all the experiments we used the same settings for the cheating system, i.e., the player were not allowed to change them; the aimbot was enabled with the *slow aim* active while the visibility angle and the *bone aiming* features were not active; the triggerbot, the 2D radar and the 3D radar were enabled, while the warning system was not enabled. The collected data, after the preprocessing step, lead to a dataset consisting of 250 examples almost evenly distributed among the two classes, i.e., `cheater` and `honest`.

To obtain an unbiased and more reliable assessment of the performance of the learned models, we built a test set from a completely new set of logs. In particular, we collected the logs of 39 matches played on three different maps (namely, *Biohazard*, *Sentinel* and *Rising Sun*) and involving three human players (namely an expert player, and average player, and a novice player). In this case, each match consisted of a two minutes deathmatch against an artificial intelligence controlled by the CPU; the players were not allowed to switch on or to switch off the cheating system; in particular, in 18 matches out of 39, the cheating system was active while in the others, i.e., 21 out of 39, it was not active; when active, the configuration of the cheating system was exactly the same used previously for building the training set. Finally, we preprocessed the collected data by using only one sequence of 30 seconds from each logged match, leading to a test set of 39 examples.

B. Results and Discussion

We used the previously collected training set to build five decision models with five different supervised learning methods, i.e., decision trees, Naive Bayes, random forests, neural networks, and support vector machines. Then, we compared the performance of the five learned models by applying them (off-line) to the previously collected test set. Both the training and the test of the decision models was carried out using the implementations provided with RapidMiner [14] [1]. For all the five supervised learning techniques used we used basically the default parameters settings. The decision tree was learned using the tree induction algorithm of RapidMiner, that works similarly to Quinlan's C4.5 [16], using the *gain ratio* as split criterion [20]. To train the Naive Bayes classifier we used the *Laplace correction* [15] to prevent high influence of zero probabilities. In the case of random forests, the learned model consisted of ten random trees and the *gain ratio* was used as split criterion [20]. Concerning the neural networks, we used a feed-forward neural network with a single hidden layer trained by a backpropagation algorithm; the number of hidden nodes was set as $(|A| + |C|)/2 + 1$, where $|A|$ is the number of attributes and C is the number of classes; all the nodes use a sigmoidal activation function. Finally, for the support vector machines we used *JMySVMLearner*, a Java implementation of *mySVM* [21] included in RapidMiner, with a *dot* kernel [19].

Table III compares the accuracy achieved by the five models on the test set. All the learned models predict correctly almost 90% of the examples in test set; Naive Bayes and support vector machines are the most accurate with only one wrong prediction; neural networks are, instead, the less accurate method with *only* 35 correct predictions out of 39. As expected, the most powerful and expensive technique, i.e., support vector machines, achieves the best results. However, surprisingly the Naive Bayes classifier achieves the same performance, also suggesting that the data attributes are not strongly interdependent.

TABLE III
COMPARISON OF THE ACCURACY ACHIEVED WITH FIVE DIFFERENT SUPERVISED LEARNING TECHNIQUES.

Technique	Correct Predictions	Accuracy
Decision Trees	36 out of 39	92.31%
Naive Bayes	**38** out of 39	**97.44%**
Random Forest	37 out of 39	94.87%
Neural Networks	35 out of 39	89.74%
Support Vector Machines	**38** out of 39	**97.44%**

Table IV reports the confusion matrix for each technique which provides more information about the distribution of wrong and correct predictions. In particular, it is worth noticing that only the Naive Bayes classifier does not misclassify any example of *honest* behavior (Table IV.B), i.e., no false positives are reported. In contrast, in the case of support vector machines (despite having the same overall accuracy) one false positive is reported by the confusion matrix (Table IV.E). From one hand, the performance of the two techniques are too close in our analysis to draw some general conclusions. On the other hand, with respect to the problem considered here, false positive might be considered more critical than false negative. In fact, the goal of a cheat detection system is to identify cheaters and prevent them from accessing either temporarily or permanently to the online gaming service. Accordingly, a false positive would result in preventing an honest player to use a service that he/she might have paid for.

VII. Conclusions

Cheating is a widespread phenomenon which compromise the players' experience in most of the recent multiplayer online games. Unfortunately, detecting cheating behaviors is a difficult task and the existing systems heavily rely on a time-consuming monitoring activity.

In this work we introduced a framework to automatically detect cheating behaviors in Unreal Tournament III by exploiting supervised learning techniques. Our framework mainly consists of three components : (i) an extended game-server responsible for collecting the game data; (ii) a processing backend in charge of preprocessing data and detecting the cheating behaviors; (iii) a frontend which provides a user interface for monitoring the game activity.

To validate our framework we performed an experimental analysis involving three human players, three game maps

[1]All the results reported in this work have been obtained using the version 5.1 of RapidMiner.

TABLE IV

CONFUSION MATRIX OBTAINED WITH FIVE DIFFERENT SUPERVISED LEARNING TECHNIQUES: (A) DECISION TREES, (B) NAIVE BAYES, (C) RANDOM FORESTS, (D) NEURAL NETWORKS, AND (E) SUPPORT VECTOR MACHINES.

	True Class	
Predicted Class	**Cheater**	**Honest**
Cheater	17	2
Honest	1	19

(A)

	True Class	
Predicted Class	**Cheater**	**Honest**
Cheater	17	0
Honest	1	21

(B)

	True Class	
Predicted Class	**Cheater**	**Honest**
Cheater	17	1
Honest	1	20

(C)

	True Class	
Predicted Class	**Cheater**	**Honest**
Cheater	18	4
Honest	0	17

(D)

	True Class	
Predicted Class	**Cheater**	**Honest**
Cheater	18	1
Honest	0	20

(E)

and five different supervised learning techniques. Our results show that our framework is able to detect effectively the cheating behaviors. In fact, the tested supervised learning techniques achieved an overall accuracy ranging from 89.74%, for neural networks, up to 97.44%, for the Naive Bayes classifier and the support vector machines.

REFERENCES

[1] S. Yeung and J. Lui, "Dynamic bayesian approach for detecting cheats in multi-player online games," *Multimedia Systems*, vol. 14, pp. 221–236, 2008, 10.1007/s00530-008-0113-5. [Online]. Available: http://dx.doi.org/10.1007/s00530-008-0113-5

[2] H. Kim, S. Hong, and J. Kim, "Detection of auto programs for mmorpgs," in *AI 2005: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, S. Zhang and R. Jarvis, Eds. Springer Berlin / Heidelberg, 2005, vol. 3809, pp. 1281–1284. [Online]. Available: http://dx.doi.org/10.1007/11589990_187

[3] H.-K. Pao, K.-T. Chen, and H.-C. Chang, "Game bot detection via avatar trajectory analysis," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 2, no. 3, pp. 162 –175, sept. 2010.

[4] L. Chapel, D. Botvich, and D. Malone, "Probabilistic approaches to cheating detection in online games," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, aug. 2010, pp. 195 –201.

[5] N. Baughman and B. Levine, "Cheat-proof playout for centralized and distributed online games," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1, 2001, pp. 104 –113 vol.1.

[6] P. Laurens, R. F. Paige, P. J. Brooke, and H. Chivers, "A novel approach to the detection of cheating in multiplayer online games," in *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 97–106. [Online]. Available: http://portal.acm.org/citation.cfm?id=1270390.1271088

[7] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin, "An efficient synchronization mechanism for mirrored game architectures," in *Proceedings of the 1st workshop on Network and system support for games*, ser. NetGames '02. New York, NY, USA: ACM, 2002, pp. 67–73. [Online]. Available: http://doi.acm.org/10.1145/566500.566510

[8] J. Goodman and C. Verbrugge, "A peer auditing scheme for cheat elimination in mmogs," in *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, ser. NetGames '08. New York, NY, USA: ACM, 2008, pp. 9–14. [Online]. Available: http://doi.acm.org/10.1145/1517494.1517496

[9] "Artificial Aiming," http://www.artificialaiming.net.

[10] "Illusion-Hacks," http://www.illusion-hacks.net.

[11] "Artificial Aiming," http://www.killahacks.com/.

[12] "Hack Forums," http://www.hackforums.net/.

[13] "Game Deception," http://www.gamedeception.net/.

[14] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler, "Yale: Rapid prototyping for complex data mining tasks," in *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, L. Ungar, M. Craven, D. Gunopulos, and T. Eliassi-Rad, Eds. New York, NY, USA: ACM, August 2006, pp. 935–940. [Online]. Available: http://rapid-i.com/

[15] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[16] J. R. Quinlan, *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[17] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[18] S. Haykin, *Neural Networks: A Comprehensive Foundation*. New York: Macmillan, 1994.

[19] B. Schölkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond (Adaptive Computation and Machine Learning)*, 1st ed. The MIT Press, Dec. 2001. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0262194759

[20] J. Han, *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[21] S. Rüping, *mySVM-Manual*, University of Dortmund, 2000, http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/.