	Compte-Rendu Final PIAD Détection de triche dans les jeux en temps réel		
Activité	PIAD de Master d'Informatique	**Type**	CR
Objet	Compte-Rendu Final PIAD : Détection de triche dans les jeux en temps réel		
Destinataires	M. Pierre-Henri Wuillemin	**Pages**	22

Version	Statut	Auteur	Date	Description
1.0	Création	Larroque Stephen	06/05/2013	Création
1.0	MAJ	Larroque Stephen	09/05/2013	MAJ de nombreuses choses (oublié de changer la version, c'est la version finale soumise par mail)
1.1	MAJ	Larroque Stephen	10/05/2013	Corrections d'erreurs mineures

Table des matières

1 MOTIVATION

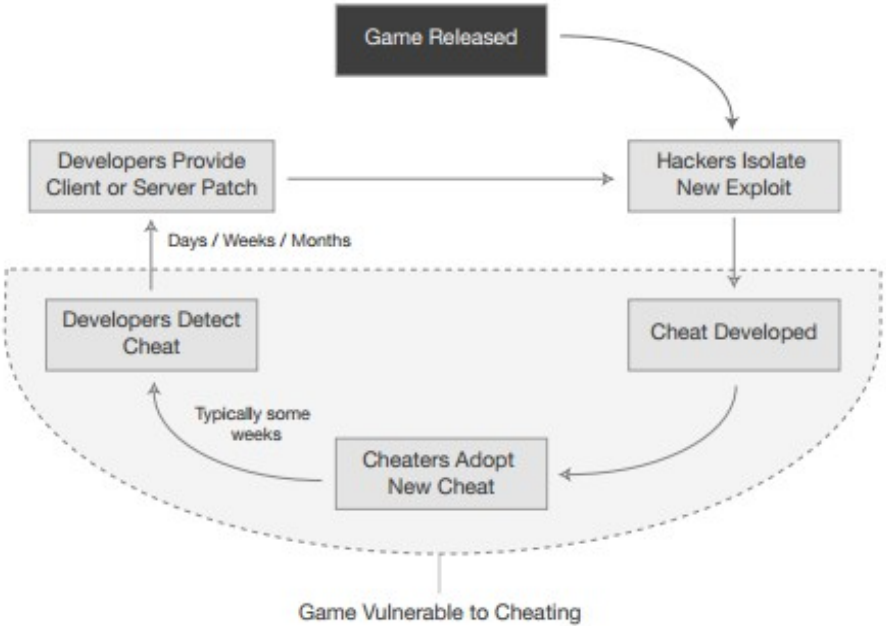
Les jeux en lignes représente une importante part de l'industrie vidéoludique et une importante source de revenus.

Outre les problèmes de gestion du réseau, le problème principal que rencontre les concepteurs de jeux en ligne est la triche.

Les tricheurs sont des joueurs qui utilisent un procédé afin de gagner un avantage décisif et inéquitable avec les autres joueurs, en général de façon non légitime par rapport aux règles du jeu (sauf pour les triches spécifiques au jeu[1] qui restent une zone floue et où la légitimité dépend des conventions éthiques entre joueurs et/ou du concepteur du jeu).

Les méthodes traditionnelles de détection des tricheurs nécessitent de lourdes infrastructures, sont peu efficaces[2], posent des problèmes concernant la vie privée des joueurs[2][4] et ils nécessitent un cycle de maintenance triche-défense réactionnaire par nature et très coûteux en temps et en personnel[2][5].

D'autre part, les tricheurs ont un avantage critique : ils ont un accès total à la machine qu'ils utilisent[4]. Les méthodes traditionnelles de détection des tricheurs, reposant essentiellement sur l'analyse de la machine cliente, sont nécessairement vouées à l'échec sur le long terme[2][4].



1. Illustration: cycle triche-défense, extrait du document [5]

Dans ce cadre, une approche complètement côté serveur semble être la plus appropriée. L'apprentissage machine semble être également un choix judicieux puisqu'il est très facile de générer des données en grand nombre, avec de nombreux critères, alors qu'il est difficile de définir explicitement les comportements de triche par une fonction ou des règles.

Certaines expérimentations antérieures à ce projet ont d'ailleurs été menées dans ce sens et ont été encourageantes[5][6][7], mais d'une part, la qualité, le nombre et les paramètres des jeux de test utilisés dans ces expérimentations ne sont pas suffisant pour permettre une évaluation précise des performances et de la viabilité de ces systèmes[5][6][7], et d'autre part, ces systèmes avaient beaucoup de contraintes notamment sur les types de triche qu'ils pouvaient détecter[5][6], et enfin, les faux positifs[5][6][7] (joueur honnête détecté comme tricheur) qui doivent à tout prix être évités. Enfin, la plupart de ces recherches ont été menées dans le but de réduire les triches par collusion dans les MMORTS (jeux de stratégie)[2][4][8], qui ont une dynamique tout à fait différente des jeux en temps réel à la première personne (FPS), et aucun code source n'est disponible.

D'autres types de systèmes de détection ont été proposé, notamment un système utilisant le langage Event-B afin de définir des règles logiques et procéder à un *model checking* comportemental[3], et un système d'analyse par machine learning du binaire du client de jeu en mémoire et de ses dépendances/hooks[4], automatisant ainsi le travail usuel des développeurs de signatures pour les systèmes anti-triche basé sur des base de signatures.

Ces deux systèmes ont tous deux des approches très intéressantes, mais ils ont néanmoins des lacunes flagrantes : le premier nécessite de définir manuellement des règles comportementales, ce qui est d'une part très difficile, très coûteux en temps humain et enfin peu robuste puisque modélisant uniquement les anomalies et triches connues ; le deuxième système, bien que théoriquement robuste car la détection se fait coté serveur, est en pratique peu fiable car il nécessite que les données coté client lui soient envoyés (ce qui peut être modifié par l'utilisateur de la machine cliente)[4] et ne peut détecter que les systèmes de triches de type « hook » qui s'attache au binaire du jeu, ignorant ainsi totalement les système de triche tierce tels que les *color bots* (triche qui tire automatiquement lorsqu'une couleur définie passe au milieu de l'écran et du curseur du joueur) et qui ne nécessitent aucune attache au binaire du jeu.

Une autre composante importante de ce problème et souvent ignorée est l'aspect communautaire: les tricheurs sont souvent des développeurs expérimentés, et les tricheurs forment de grandes communautés où ils partagent leurs connaissances et expérience. Il est donc vain de vouloir solutionner ce problème avec seulement une équipe restreinte de développeur, et avec un système « one-size-fits-all » (un seul système statique pour tous les problèmes de ce type). La déduction logique est que pour qu'un système anti-triche puisse être efficace sur le long terme, il doive lui aussi permettre une collaboration communautaire à grande échelle.

Il n'y a donc à l'heure actuelle aucune solution efficace pour résoudre le problème de la triche.

Dans le cadre de ce projet, nous nous intéresserons aux systèmes de triche basés sur la technologie : les systèmes de triche traditionnels d'une part (eg : *aimbot, wallhack, autoshoot*) et les *togglers* (tricheur activant leur système de triche de manière intermittente ou à des moments clés) d'autre part, qui sont beaucoup moins étudiés dans la littérature.

2 SOLUTION THÉORIQUE

Afin d'aborder tous les problèmes sus-mentionnés, nous posons comme hypothèse que les tricheurs montrent un ***comportement significativement différent*** des joueurs honnêtes, et qu'**un expert est capable de discriminer de manière fiable** entre un joueur honnête et un tricheur. Si c'est le cas, les tricheurs pourraient être identifiables quelque-soit la méthode utilisée pour tricher.

On émet donc l'hypothèse que ce problème de détection de triche est assimilable au **problème de détection d'anomalie**.

En sus, si l'on considère **chaque joueur comme une variable aléatoire**, on peut donc considérer que le mélange des comportements de tous les joueurs **convergera en loi vers une loi normale**, de par le **théorème central limite**.

Nous avons par ailleurs choisi une approche par **Système Immunitaire Artificiel par Sélection Négative (AIS)**, lesquel est une famille d'algorithmes de **détection d'anomalie (anomaly detection)** en modélisant le comportement des joueurs honnêtes, plutôt qu'en modélisant les comportements de triches ou les deux (misuse detection).

Les algorithmes de détection d'anomalie sont en fait très similaires au Naïve Bayes, à ceci près qu'ici nous faisons l'hypothèse que la **classe d'exemples normaux est significativement majoritaire par rapport à a la classe d'anomalies**.

Dans le cas du Naive Bayes, nous obtenons la vraissemblance d'un exemple nouveau par rapport à une classe avec l'équation suivante :

Pour la classe H des joueurs honnêtes :
$p(X|H) = p(x1|H) * p(x2|H) * \ldots * p(xn|H)$

Pour la classe C des joueurs tricheurs :
$p(X|C) = p(x1|C) * p(x2|C) * \ldots * p(xn|C)$

Enfin, on calcule le Max de Vraissemblance entre p(C|X) et p(H|X), et on affecte X à la classe où la vraissemblance est maximum. Naive Bayes est donc un algorithme exclusivement génératif, et qui modèle entièrement toutes les classes que l'algorithme doit détecter.

Dans le cas des algorithmes de détection d'anomalie, nous faisons l'hypothèse qu'il n'y a qu'une seule classe et valeur, et nous modélisons notre modèle uniquement pour cette classe.

Au lieu d'avoir ceci :
$p(X|H) = p(x1|H) * p(x2|H) * \ldots * p(xn|H)$

Nous simplifions par H et nous obtenons ceci :
$p(X) = p(x1) * p(x2) * \ldots * p(xn)$

En vérité la classe H est toujours là, mais il n'est plus utile de la faire apparaître, car on exclut l'existence de C dans notre jeu de données (en vérité cela fonctionne quand même avec quelques exemples appartenant à la classe C, mais il faut que H soit largement majoritaire).

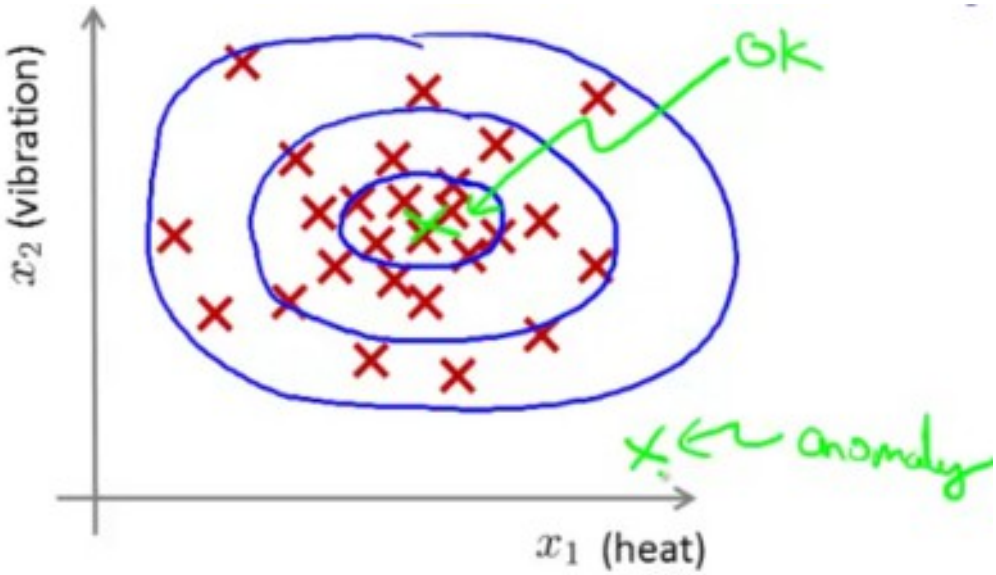


Fig2 : Exemple de détection d'anomalie sur un jeu 2D utilisant une gaussienne 2D

Enfin, pour affecter l'exemple X à la classe H ou C, on applique un seuil Z à p(X) : si p(X) >= Z, X est de la classe H. En revanche, si p(X) < Z, on considère que l'exemple X est de la classe C.

En effet, on peut avoir l'intuition que p(X) représente la vraisemblance qu'un joueur soit honnête. Lorsque X est honnête, p(X) → 1. En revanche, si X est un tricheur, p(X) → 0, car l'exemple X va sortir du modèle que nous avons généré pour la classe H.

Les algorithmes de détection d'anomalie sont donc des algorithmes **semi-génératifs** (on modélise la classe H des joueurs honnêtes), et **semi-discriminatifs** (les anomalies qui semblent ne pas correspondre à ce modèle sont probablement des tricheurs C).

L'avantage principal de ces algorithmes est qu'ils sont robustes : on détecte les joueurs honnêtes, pas les joueurs tricheurs. On est donc en mesure de détecter n'importe quel nouveau type de triche.

L'inconvénient est qu'on ne peut pas connaître la classe exacte de triche détectée puisque nous ne détectons jamais la triche, mais seulement les comportements honnêtes (les joueurs humains).

Ils sont d'autre part relativement computationnellement efficaces, et surtout une fois l'apprentissage des paramètres effectués (paramètres de la loi gaussienne), la

prédiction/détection est quasiment en temps constant, ce qui les rend pratique à l'analyse de données en temps réel.

Nous avons décidé d'implémenter trois algorithmes AIS que nous allons ici détailler.

2.1 Gaussienne univariée

L'algorithme de Gaussienne univariée applique une hypothèse très forte d'indépendance entre les variables.

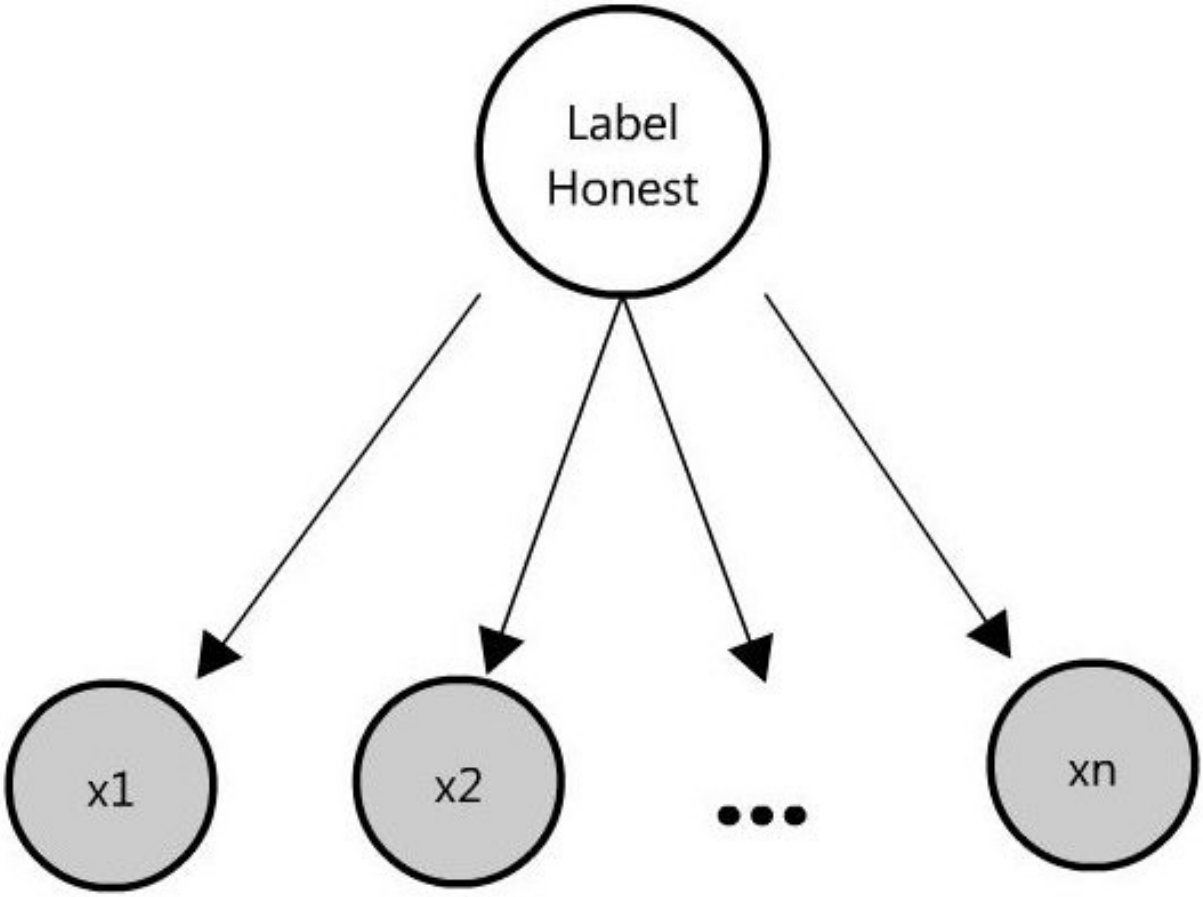


Fig3 : Graphe probabiliste pour les modèles Gaussien Univariés

Ce modèle considère qu'il n'y a jamais aucune corrélation entre les différents critères de comportement (plus d'informations sur les critères dans le chapitre 3 Implémentation). Ce modèle est très similaire au Naive Bayes, sauf qu'il n'y a qu'une seule classe modélisée (classe Honnête).

Nous avons donc :
P(X) = p(x1) * p(x2) * ... * p(xn)

Lors de l'apprentissage, il suffit de calculer les paramètres de la gaussienne multidimensionnelle (un vecteur de moyennes Mu et un vecteur de variance Sigma2 pour tous les critères).

Lors de la détection, pour calculer la « vraisemblance » d'un nouvel exemple X, il suffit de calculer la densité de probabilité de X par rapport aux paramètres appris :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Où f(x) = p(x).

Il suffit ensuite de définir un seuil Z en-dessous duquel un exemple est vraisemblablement une anomalie (un tricheur). La définition du seuil peut se faire par le calcul d'une courbe ROC ou d'un score F1.

Cet algorithme fournit en général des résultats moins précis à cause de son hypothèse naive, mais il fournit néanmoins de bons résultats et surtout il est extrêmement rapide et simple à calculer.

2.2 Gaussienne multivariée

L'algorithme Gaussien Multivarié pour les détections d'anomalie prend la forme d'une gaussienne multivariée, c'est-à-dire que les critères peuvent être corrélés selon la matrice des covariances.

Une gaussienne multivariée où la matrice de covariance ne possède des nombres que sur la diagonale, et 0 partout ailleurs, est équivalent à une gaussienne univariée.

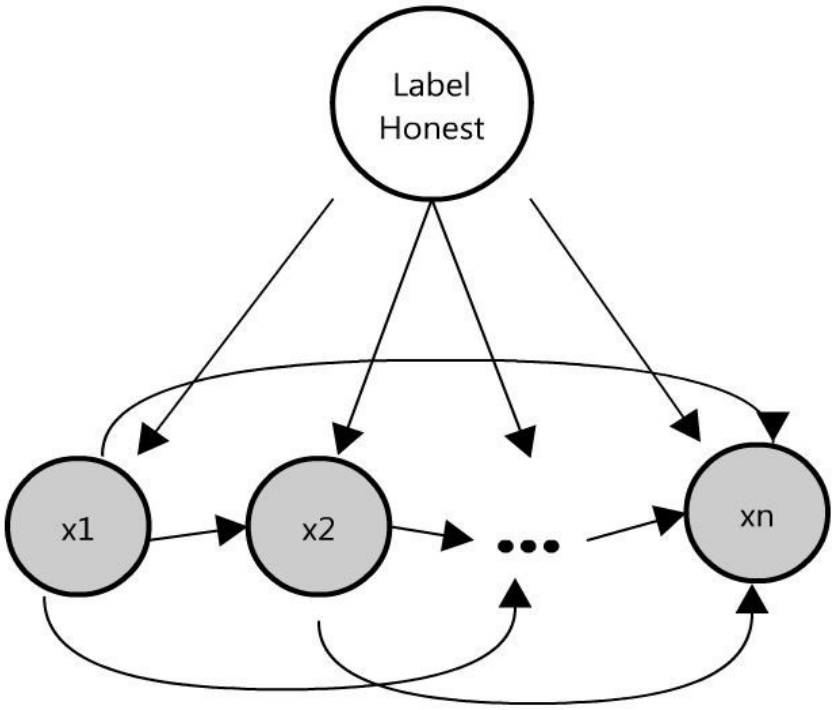


Fig4 : Graphe probabiliste pour les modèles Gaussien Multivariés

En multivarié, la matrice de covariance nous indique toutes les corrélations entre tous les critères.

Pour apprendre ce modèle, il est nécessaire d'apprendre le vecteur des moyennes Mu pour chaque critère, et la matrice de covariance Sigma2 :

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)(x^{(i)} - \mu)^T$$

À la détection, la « vraisemblance » est calculée avec l'équation de densité multidimensionnelle suivante :

$$p(x) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

Enfin, comme avec le modèle Univarié, nous définissons et utilisons un seuil Z en-dessous duquel X sera classé comme tricheur.

L'avantage de ce modèle est qu'il est très précis et capture toutes les corrélations entre critères.
Néanmoins l'inconvénient principal est qu'il est très lent à l'apprentissage à cause du calcul de la matrice de covariance est de complexité O((m^2)/2) où m est le nombre de critères.

2.3 CANS

CANS (Cluster-Augmented Negative Selection algorithm) est un tout nouvel algorithme que nous avons conçu et implémenté pour les besoins de ce projet.

L'algorithme CANS est à mi-chemin entre l'algorithme Gaussien Univarié et Gaussien Multivarié. C'est aussi une réduction de l'algorithme TAN (Tree Augmented Naive Bayes).

L'algorithme CANS calcule l'information mutuelle entre les critères, et agrège ensemble dans des « clusters » les critères qui sont le plus corrélés. Le nombre de clusters est variable et configurable.

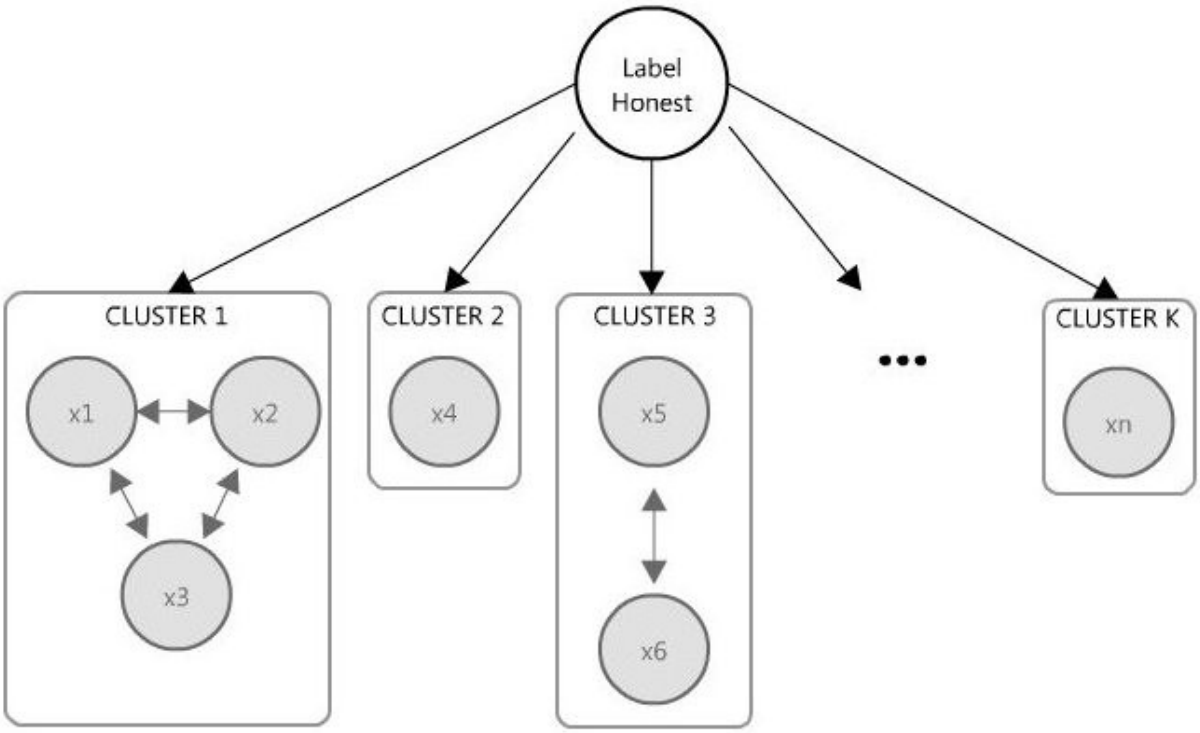


Fig5 : Graphe probabiliste pour le modèle CANS

CANS apprend les paramètres de la façon suivante :

- Calcule le vecteur des moyennes Mu de chaque critère
- Calcule le vecteur des variances Sigma2 de chaque critère
- Calcule la corrélation entre les critères
- Agrège les critères en clusters selon le nombre CMAX fixé
- Calcule les matrices de covariances pour chaque cluster où il y a plus qu'un seul critère contenu.

Pour calculer la corrélation entre les critères, CANS calcule l'information mutuelle entre chaque couple de critères.

Enfin, CANS agrège les critères en utilisant un algorithme similaire à Kruskal, qui agrège en priorité les critères qui sont les plus corrélés et en respectant la contrainte CMAX (CMAX étant le nombre maximum de critères dans un cluster).

À la détection, CANS calcule la vraisemblance de chaque cluster de la façon suivante :

- Soit en calculant la vraisemblance univariée pour ce critère seul s'il n'y a qu'un seul critère dans le cluster ;
- Soit en calculant la vraisemblance multivariée s'il y a plusieurs critères en utilisant la matrice de covariance correspondante à ce cluster et calculée précédemment pendant l'apprentissage.

L'intérêt de cet algorithme est qu'on peut fixer le nombre maximum de critères par cluster, et donc fixer le temps de calcul.

En effet, la complexité au pire cas de cet algorithme est en O((CMAX^2)*K) où CMAX est le nombre maximum de critères par cluster (CMAX <= m le nombre de critères au total) et K le nombre total de clusters.

Cet algorithme est donc linéaire pour un CMAX fixé, tandis que l'algorithme Gaussien Multivarié est de complexité quadratique.

Cluster 1
lastmouseeventtime
lastcommandtime

Cluster 2
ducked
reactiontime
selfdamagecount
powerup_quad
speedratio
movementdirection
midair
commandtime_reactiontime
weaponstate
angleinaframe
selfdamageeventcountacc

Cluster 3
mouseeventtime_reactiontime

Cluster 4
scoreacc

Fig6: Exemple d'un clustering effectué par l'algorithme CANS

3 IMPLÉMENTATION

3.1 Vue d'ensemble de l'implémentation

Les algorithmes ne sauraient résoudre à eux seuls le problème de triche qui a aussi d'autres dimensions (sociales notamment).

Nous avons donc supplémenté ces algorithmes par la réalisation d'un framework permettant de rajouter d'autres possibilités importantes pour la concrétisation et l'efficacité d'un système anti-triche opensource.

Ce framework, complètement opensource, est un système résidant côté serveur et composé de briques logicielles :

- interchangeables : chaque partie du système (interface jeu, algorithmes, détection) peut être changée aisément ;
- génériques et réutilisables : entre plusieurs jeux, il suffit de générer des données ;
- collaboratives : esprit open source, les paramètres de détection peuvent être partagés et améliorés itérativement dans une communauté à grande échelle.

Ce système fonctionne en offline (les données sont collectionnées et traitées plus tard) pour l'apprentissage (ce qui permet de réaliser plusieurs apprentissages avec plusieurs algorithmes et paramètres et de régler les paramètres d'apprentissage) et en mode online (les données sont traitées au vol) lors de la détection d'anomalie.

D'autre part, les données d'apprentissage collectées n'ont pas besoin d'être identifiées, ce qui protège la vie privée des joueurs.

Les spécifications de ce système ont de nombreux avantages :

- Côté serveur : totalement transparent pour les utilisateurs, allège les ressources client et serveur puisque le système peut être placé sur un autre serveur, impossible à désactiver ni outrepasser, mise à jour facile par les administrateurs, etc.
- Préemptif: plus de cycle triche-défense, le système de détection de triche détecte les tentatives de triches même si les méthodes de triche ne sont pas encore connues (*zero-day* ou *private*).
- La méthode de triche importe peu, car tant que le comportement induit est significativement différent d'un joueur normal, la triche sera détecté.
- Le(s) développeur(s) peut travailler à son rythme pour étendre et améliorer le système de détection, et même agir en avance, plutôt que de ne pouvoir que seulement réagir après qu'une tentative de triche se soit déroulée.
- Aisé d'étendre le système puisque chaque brique logicielle est interchangeable.
- Générique et réutilisable : l'interface jeu et le système de détection de triche sont découplés, il est donc possible de réutiliser le système entre plusieurs jeux très différents, en n'ayant seulement qu'à programmer une nouvelle interface jeu pour le jeu cible.

- Open source : le système pourra être développé et amélioré par de nombreuses personnes, puisque chacun peut avoir accès au code source librement.
- Collaborative : puisque le système est open source et que les données collectées n'ont pas d'identifiants,il est donc possible de partager les jeux de données d'apprentissage, ou simplement les paramètres résultant de l'apprentissage, entre un groupe d'utilisateurs, ou même sur le web entier.
- Amélioration itérative généralisée : les points précédents pourront donc donner lieu à une amélioration itérative du système par un groupe restreint de développeurs, mais qui profitera à tous les utilisateurs puisque le système est complètement ouvert.

3.2 Implémentation de la solution

Pour l'implémentation, nous avons utilisé le jeu FPS (jeu de tir en temps réel à la première personne) multijoueurs et opensource OpenArena, qui est basé sur Quake 3 Arena et dont le code source est disponible et est bien documenté.

Le framework implémenté est composé des éléments suivants :

- **Un serveur de jeu OpenArena étendu**, comprenant :
  - l'ajout facile des critères d'apprentissage/détection,
  - l'interface avec le système de détection de triche (enregistrement des critères dans un fichier log dédié ; génération de la table des joueurs pour identification à la phase de détection),
  - des commandes spéciales pour déclarer un joueur comme honnête ou tricheur (pour l'étiquetage des données).
- **OACS** : Le logiciel de détection d'anomalies, complètement externe au serveur de jeu et donc indépendant. OACS apprend à partir du jeu de données fourni par le jeu (on peut donc collecter ces données au long du temps), mais il l'utilise également ensuite lors de la détection.

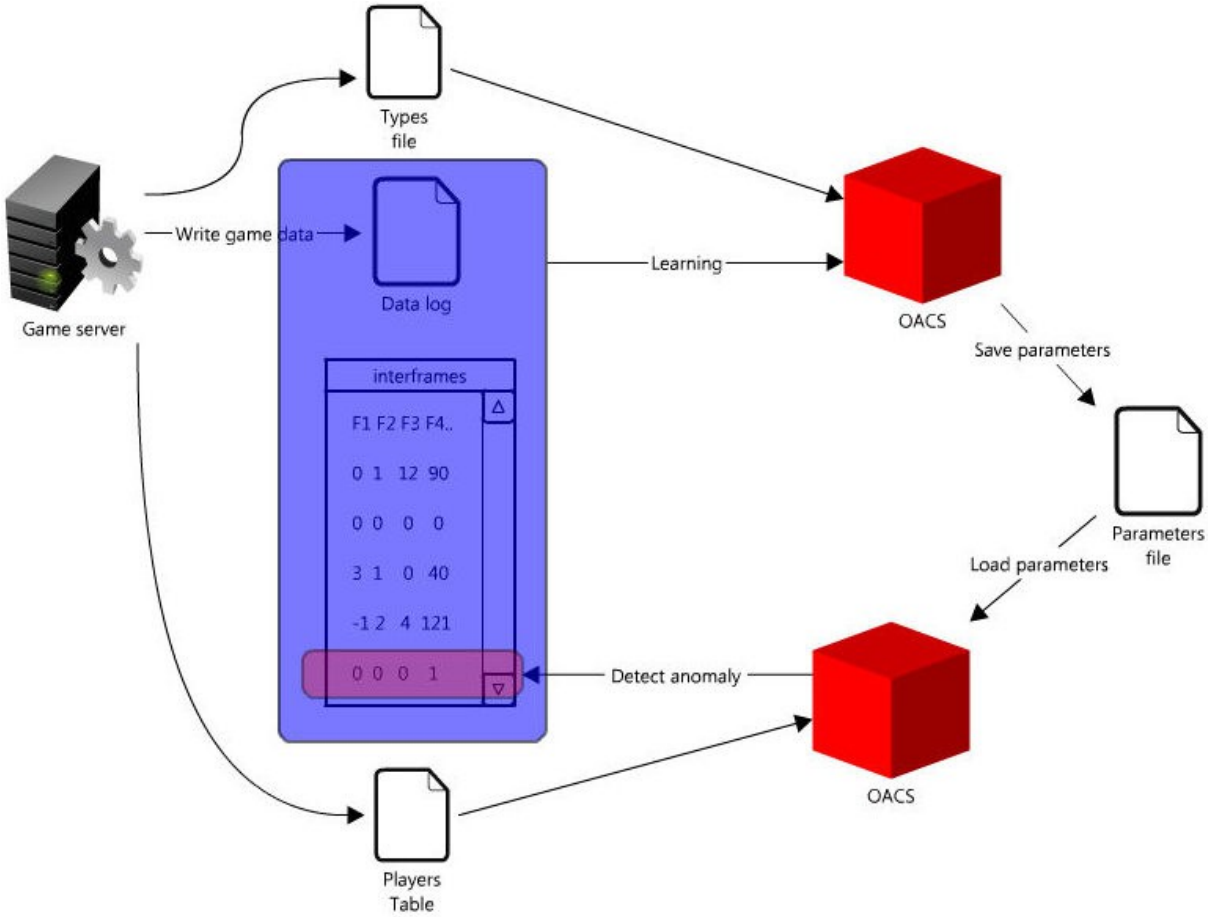


Fig7 : Fonctionnement global du système anti-triche

OACS est composé de nombreux modules et a été conçu de façon totalement modulaire. Chaque type de modules contient de nombreux sous-modules et il est très facile d'en créer de nouveaux :

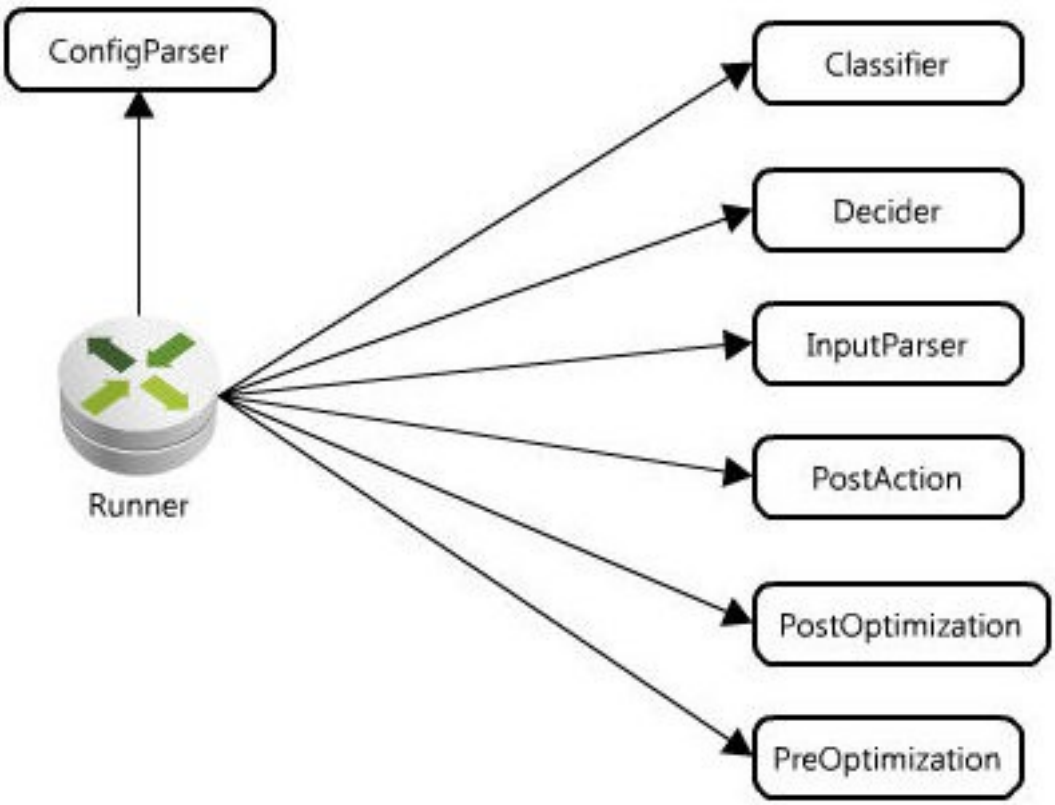


Fig8 : Types de modules disponibles dans OACS actuellement

En standard OACS fonctionne de la manière suivante :

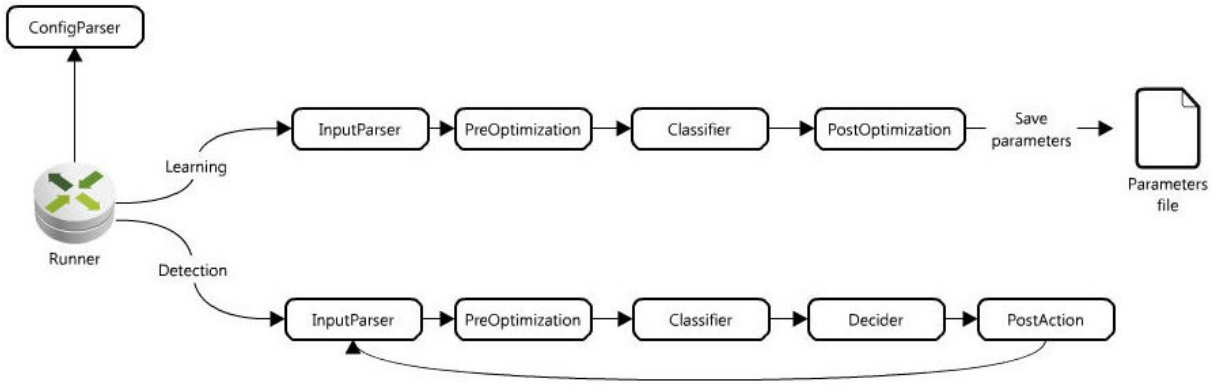


Fig9 : Fonctionnement standard d'OACS

Néanmoins le fonctionnement est totalement modulaire et configurable simplement dans le fichier de configuration. On peut donc imaginer des configurations complexes comme du chainage de classifiers :

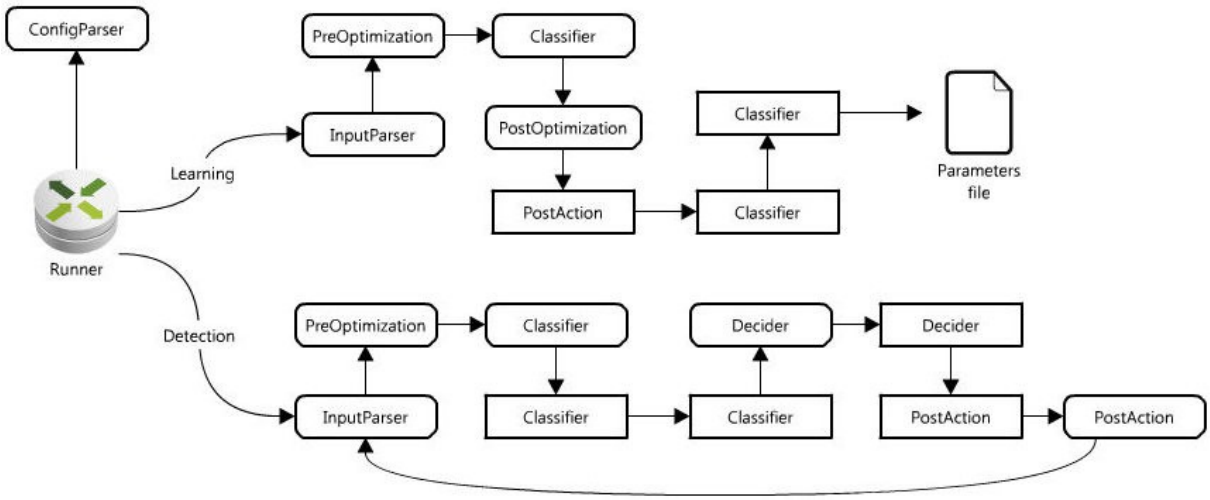


Fig10 : Configuration plus complexe de la routine d'exécution

3.3 Données de jeu et interframes

Les données de jeu sont composées d'un certain nombre de critères (tels que la position des joueurs, l'arme utilisée, etc... - Plus d'informations dans *Types de critères*).

Plus précisément, les données sont extraites des *interframes*, c'est-à-dire de la différence entre deux *frames* (état du monde dans le jeu) consécutives.

Ces données seront enregistrées dans un fichier texte (ou *log*), et seront utilisées à la fois pour l'apprentissage de nos modèles, et leur évaluation dans le cadre de la détection (biensûr, les données seront différentes).

3.4 Niveau d'abstraction des critères

Dans notre implémentation, nous avons choisi des critères d'apprentissage et de prédiction/détection de ***bas niveau*** (non traités). Éventuellement, des critères de haut niveau (agrégés et pré-traités) peuvent être également utilisé si les critères bas niveaux ne sont pas suffisants (en utilisant un module de « preoptimization » dans le logiciel OACS).

3.5 Types de critères

Pour une efficacité et interopérabilité maximale de notre système, nous avons défini une échelle de quatre types de critères :

IDENTIFIEURS : ce type de critère permet d'identifier des qualités de l'état du monde dans le jeu, mais qui ne sont pas nécessaires à l'apprentissage, mais seulement à la détection. Par exemple : identifieur aléatoire du joueur, identifieur aléatoire de la partie jouée, numéro de frame, timestamp, etc...

Note : ce sont des identifieurs qui renseignent sur l'état du jeu, mais ne permettent pas d'identifier les joueurs (les identifieurs joueurs sont générés aléatoirement pour toute la durée de la connexion au serveur).

Note2 : ce type est le seul type de critère à ne pas être utilisé à l'apprentissage. À la détection, ils ne sont pas donnés en argument aux algorithmes de détection (qui prennent les mêmes données en détection qu'à l'apprentissage), mais par la suite pour implémenter la tolérance temporelle (voir plus bas) et enregistrer la description des événements de triche.

SPÉCIFIQUE HUMAIN : ces critères sont utilisé à l'apprentissage, et sont caractéristiques d'un comportement humain, et qui sont invariables aux règles du jeu (voire au jeu joué). Par exemple : le temps de réaction après avoir vu un adversaire, la fréquence que la mire de visée reste sur l'adversaire, la précision de la visée, etc...

SPÉCIFIQUE AUX RÈGLES DU JEU : ces critères d'apprentissage et détection définissent des paramètres de jeu qui peuvent changer si les règles du jeu changent. Par exemple : La quantité de dégâts infligés avec une arme, numéro de l'arme utilisée, mort conséquente au tir, etc...

SPÉCIFIQUE AUX LIMITES (PHYSIQUES) DU JEU : ces critères définissent des paramètres qui sont théoriquement limités par la physique du jeu. Nous pensons que ce type de critère n'est pas fiable et beaucoup trop variable car dépendant de limites théoriques qui peuvent être (et ont souvent été) dépassé par les joueurs sans aucune triche et de façon totalement légitime (comme les « tricks » et « trickjumping », qui sont des mouvements spéciaux qui permettent de dépasser les limites physiques théoriques du système). Ce type de critère n'est donc pas utilisé dans notre implémentation. Exemples : vitesse du joueur, acceleration du joueur, etc...

La liste des critères est la suivante :

```
"playerid",
  "timestamp",

  "svstime",
  "reactiontime",
  "svtime",
  "lastcommandtime",
  "commandtime_reactiontime",
  "angleinaframe",
  "lastmouseeventtime",
  "mouseeventtime_reactiontime",
  "movementdirection",


  "score",
  "scoreacc",
  "hits",
  "hitsacc",
  "death",
```

```
"deathacc",
"captures",
"capturesacc",
"impressivecount",
"impressivecountacc",
"excellentcount",
"excellentcountacc",
      "defendcount",
"defendcountacc",
      "assistcount",
"assistcountacc",
      "gauntletfragcount",
"gauntletfragcountacc",

"frags",
"fragsinarow",

"selfdamageeventcount",
"selfdamageeventcountacc",

"ducked",
"midair",

"weapon",
"weaponstate",
"weaponinstanthit",

"powerup_quad",
"powerup_battlesuit",
"powerup_haste",
"powerup_invisibility",
"powerup_regeneration",
"powerup_flight",
"powerup_scout",
"powerup_guard",
"powerup_doubler",
"powerup_ammoregen",
"powerup_invulnerability",

"powerup_persistant_powerup",

"hasflag",
"holyshit",
"rank",
"enemyhadflag",

"health",
"max_health",
"armor",
"speed",
```

```
"speedratio",
"selfdamagecount",

"framerepeat",
"cheater"
```

Fig11 : Liste des critères utilisés dans OACS pour ioquake3 actuellement

3.6 Temporalité du système

Les concepteurs de ce système ont le sentiment que le manque de temporalité, limitation bien connue des algorithmes qui ont été implémentés, empêchent de prendre en compte des données qui pourraient être bien plus significatives si elles étaient replacées dans leur contexte temporel.

Afin de pallier à ce manque de temporalité, les concepteurs se proposent d'implémenter deux solutions simples :

- Des ***critères accumulateurs*** à l'apprentissage : ce sont des critères qui, au lieu de ne simplement contenir la valeur de l'événement en cours dans l'interframe, gardent en mémoire les événements passés et s'incrémentent.

  Par exemple, au lieu d'avoir un critère *"a tué un joueur"* on peut avoir *"a tué x joueurs d'affilée"* ou encore au lieu de *"a touché un adversaire"*, on aura *"a touché x fois un adversaire d'affilée sans rater"*

- Un ***concept de tolérance temporelle*** à la détection : puisque les algorithmes ne peuvent évaluer que des événements séparés (une interframe à la fois), il serait intéressant de pouvoir faire le lien entre plusieurs détection : si un joueur n'est détecté qu'une seule fois, cela ne devrait pas pour autant le sanctionner car cela peut être un événement rare isolé, mais tout à fait normal.
  À l'inverse, un joueur qui fréquemment procède à des événements qui sont détectés comme de la triche devrait être sanctionné.

  Le concept de tolérance temporelle induit ce comportement en gardant à tout moment une liste des joueurs et l'historique de leurs détections passées, comprenant le taux/probabilité de triche et la date de chaque détection. Ensuite, lors d'une nouvelle détection d'anomalie, le taux de cette détection est pondérée avec les taux des détections passées par rapport à l'intervalle de temps. Si le taux résultant dépasse un certain seuil fixé, alors le joueur est déclaré tricheur. Au final, plus l'intervalle entre les détection est grand, moins le joueur a de chance d'être un tricheur. Au contraire, plus l'intervalle entre les détection est petit, plus le joueur a de chance d'être un tricheur.

Ces deux concepts sont simples à implémenter et peu coûteux en terme de temps de réalisation et de computation.

3.7 Table des joueurs

Afin de permettre le partage opensource des résultats et la protection de la vie privée, les données de jeu ne contiennent pas de données d'identification (les seules données d'identification sont des identifieurs aléatoires qui ne permettent en aucune façon d'inférer l'identité du joueur à partir seulement de ces données).

Néanmoins, s'il est impossible d'inférer l'identité d'un tricheur, il est alors impossible de prendre des décisions.

Pour remédier à cela, le serveur de jeu étendu générera une *table des joueurs*, qui contiendra les données d'identification des joueurs (pseudonyme, adresse IP, date de connection) ainsi que la correspondance entre ces joueurs et leur identifieur aléatoire (*playerid*). Cette table pourra être optionnellement utilisée à la détection pour ajouter des informations supplémentaires dans les résultats.

3.8 Résultat de détection et décision

Le résultat de la détection sera par défaut le simple enregistrement dans un fichier texte de la description d'un événement de triche, avec l'identification du joueur si la table des joueurs est disponible (liste de tous les joueurs et paramètres d'identification, puisque les données d'apprentissage n'en possèdent pas). Un administrateur pourra alors vérifier les événements, consulter les ressources associées (capture d'écran, demo, log de jeu, etc...) afin de prendre une action à l'encontre de ce joueur.

Alternativement, le système pourrait également décider d'une action automatiquement, avec potentiellement une *gratification décalée* (l'action est prise avec un délai plus ou moins long après la détection) afin de détecter plus de tricheurs avant de les sanctionner tous à la fois, mais ceci ne sera pas implémenté dans le cadre de ce projet, la prise de décision étant en dehors du cadre de la détection de triche qui est fixé pour ce projet.

3.9 Interface graphique IPython Notebook

Une interface graphique interactive est également disponible.

Celle-ci est basé sur IPython Notebook et vous fournit une interface dans laquelle vous pouvez facilement expérimenter et commenter vos essais.

Avec cette interface, des instructions vous sont données pour vous accompagner et vous pouvez juste cliquer sur « Lancer » pour lancer les différentes fonctionnalités de l'application.

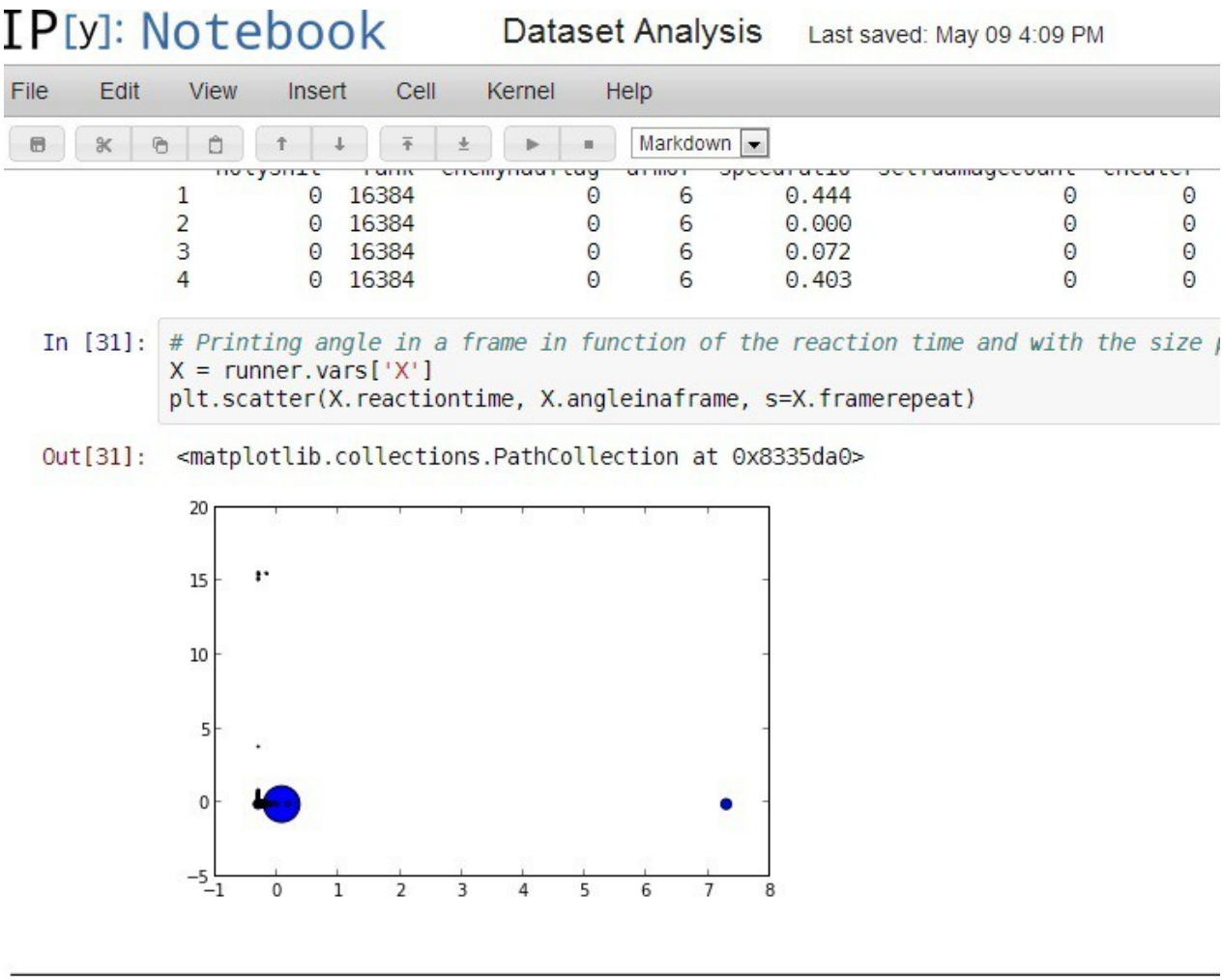


Fig12 : Capture d'écran de l'interface GUI via IPython Notebook

Cette interface est également très pratique et extrêmement conseillée pour les utilisateurs expérimentés.

3.10 Caractéristiques techniques

- Le serveur de jeu étendu (interface avec le système de détection de triche) a été programmé en C/C++ à partir des sources du projet[9].
- Le logiciel OACS de détection de triche repose sur Python[10] et Pandas[11].
- Une interface graphique interactive est proposée si IPython est installé.

Le système OACS fonctionne sous tous les systèmes d'exploitation supportant Python et les librairies nécessaires (principalement Pandas et Numpy), dont Windows, Linux et MacOSX.

4 COMPOSITION DU DÉLIVRABLE

À la conclusion de ce projet, les fichiers et documents suivant seront livrés :

- L'exécutable implémentant toutes les fonctionnalités prévues.
- Le code source entièrement commenté.
- La documentation complète du projet, à savoir :
  - Cahier des charges
  - Plan de développement
  - Dossier d'Analyse et de Conception
  - Rapport de tests décrivant les protocoles de test, leurs résultats complets et commentés
  - Rapport final
- Manuel d'installation
- Manuel de l'utilisateur
- Manuel du développeur

5 RÉFÉRENCES

[1] : 'Cheating in Online Video Games', by Savu-Adrian Tolbaru, 15/08/2011
[2] : 'Cheating and Virtual Crimes in Massively Multiplayer Online Games' by Rahul Joshi, Technical Report, RHUL-MA-2008-06, 15 January 2008
[3] : Tian, H., Brooke, P.J. and Bosser, A-G. (2012) 'Behaviour-based cheat detection in multiplayer games with Event-B', Lecture Notes in Computer Science, 7321/2012, Heideberg: Springer, pp.206-220
[4] : Fides: Remote Anomaly-Based Cheat Detection Using Client Emulation, by Edward Kaiser, Wu-chang Feng, Travis Schluessler, November 2009
[5] : Laurens, P.; Paige, R.F.; Brooke, P.J.; Chivers, H.; , "A Novel Approach to the Detection of Cheating in Multiplayer Online Games," Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on , vol., no., pp.97-106, 11-14 July 2007
[6] : Galli, L.; Loiacono, D.; Cardamone, L.; Lanzi, P.L.; , "A cheating detection framework for Unreal Tournament III: A machine learning approach," Computational Intelligence and Games (CIG), 2011 IEEE Conference on , vol., no., pp.266-272, Aug. 31 2011-Sept. 3 2011
[7] :Chapel, L.; Botvich, D.; Malone, D.; , "Probabilistic approaches to cheating detection in online games," Computational Intelligence and Games (CIG), 2010 IEEE Symposium on , vol., no., pp.195-201, 18-21 Aug. 2010
[8] :Mitterhofer, S.; Kruegel, C.; Kirda, E.; Platzer, C.; , "Server-Side Bot Detection in Massively Multiplayer Online Games," Security & Privacy, IEEE , vol.7, no.3, pp.29-36, May-June 2009
[9] :http://ioquake3.org/
[10] :http://www.python.org/
[11] :http://pandas.pydata.org/
[12] :https://forge.lip6.fr/projects/pyagrum
[13] :https://forge.lip6.fr/projects/agrum