uc3m | Universidad Carlos III de Madrid

Grado Universitario de Ingeniería Informática
2018-2019

Trabajo Fin de Grado

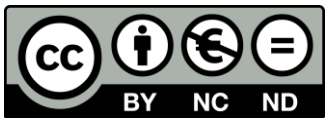
"Detección de trampas en el videojuego CS:GO utilizando técnicas de Inteligencia Artificial"

Daniel Fernández Aldea

Tutor/es

José Antonio Iglesias Martínez

Leganés, Marzo de 2019



AGRADECIMIENTOS

Este proyecto es la culminación de toda mi estancia en el grado, en el cual, a pesar de algunas dificultades por cuestiones personales, he obtenido más conocimientos aparte de los impartidos en el propio grado.

Por ello, en primer lugar, me gustaría agradecer a la universidad que ha hecho esto posible y a su gente en general, por la buena calidad de enseñanza.

Además, me gustaría agradecer particularmente a los profesores de la rama de computación o inteligencia artificial, por despertar en mí un interés en este campo tan bonito de la informática.

Agradecer también a mi tutor por todo el apoyo y motivación mostrado a lo largo del desarrollo del trabajo.

Gracias también a mis compañeros de clase, por hacer más amena la estancia, y haberme enseñado a trabajar en equipo.

Por último, agradecer a mis padres por el gran apoyo demostrado durante el desarrollo del trabajo, y a todos mis amigos que se han prestado a recolectar instancias para el proyecto.

ABSTRACT

Introduction and motivation

This project is developed using the videogame CS:GO (*Counter Strike: Global Offensive*) as its base.

This is a first person shooter (FPS) videogame developed by Valve Corporation, which is the owner of the most important digital distribution platform in the world right now, *Steam*.

It is the forth videogame of the series *Counter Strike*, being launched officially at August 21st, 2012.

This series, along with other series like *Call of Duty* or *Battlefield*, is the most important in the world of FPS videogames, since it is the one with the highest number of online players.

The most important rules (the ones to understand the project) of the competitive mode of the game are the following ones:

- The game is divided in rounds with an estimated time of 2 minutes each one of them. In each round, players respawn at their respective bases.
- There are two teams, terrorists and counter-terrorists. Terrorist's objective is to get into a place where it is possible to plant the bomb (there are two places in each map), to plant it and let the timer go on so the bomb explodes. On the other hand, counter-terrorist's objective is to prevent terrorists to get the bomb planted or to defuse it if it is planted. Also, any of the teams win the round if they kill every enemy on the opposite team (except if every terrorist is dead but the bomb explodes).
- Each team plays 15 rounds in one side, and after that they change roles. The first team who wins 16 rounds, wins the game.
- The game is hosted in a server where the information about the game gets updated 64 or 128 times (ticks) per second, depending on the server.
- To kill an enemy, a player will have to move his mouse so his *crosshair* places on an enemy, and shoot pressing the left click.

Moving the mouse does nothing but changing the angle the player is aiming to. That angle will have X and Y coordinates and will be dynamically stored in the memory of the process.

There are competitions in a lot of videogames, where the players can prove who is better, but, precisely, in this one, the amount of money invested is very high. For example, in

2018, there were at least more than 11 million of dollars invested (information collected from the most important competitions).

This amount of money gives importance to the videogame which is used as base in this project.

However, such amount attracts players, and not all of them are honest and honoured. There are players who use **cheats** (self-made or bought ones from professional coders) called **cheaters,** to have an advantage over the **skilled** players.

These cheats, sometimes are good enough to get over the **anticheat** security which the videogame or the competitions provide. There are cases of professional players being banned by the anticheat after winning prizes in important competitions, and the money is not taken back. This is one of the problems that has to be solved.

Also, CS:GO has a **competitive mode**, like explained before. There is an average of 600.000 games played per day. This mode is accessible by anyone who has bought the game or played it and got into a certain level, and the fact of ranking up each time a player wins a game encourages the use of cheats, since they can achieve results "easily".

As the game has 10 players, it gets easier to have a cheater in a game as the number of cheaters in the videogame increases. For example, supposing there is a 5% of cheaters in the CS:GO, there would be a 40% of chances of a player getting paired up with a cheater in his game.

The number of cheaters in the game is unknown, but it is known it is high enough to be the **worst** problem in the videogame. About 96% of the emails the developers receive are from people complaining about the cheaters. The reason of this, is that CS:GO has a "poor" anticheat called Valve Anti-Cheat (VAC).

This is one more problem that has to be solved, since cheaters are ruining skilled player's experience in the videogame.

Goals and objectives

As explained before, there are cheaters in the game who use external programs to gain any advantage over the enemy players.

There are different types of anti-cheats depending on how they work. VAC uses software signature recognition, but since **players want no intrusion** into the files of their computers, VAC does not scan files, and it only scans into game's allocated memory at computer's RAM, and that is the first reason of why VAC is poor.

The second reason is that cheat developers react fast when VAC updates using reverse engineering and discovering the updates.

There is a system where players can review cases of a player suspected of being cheater and give their verdict. This system is called **Overwatch**. The problem of Overwatch is

that in more than 75% of the cases the reviewed players are not cheaters (or there are not enough proofs for stating it), since the cases come from in-game reports sent by players (and there is a lot of subjectivity in them).

There are different types of cheats as well, depending on the help they provide. The ones that are more common in competitions are aim-assistant ones (called **aimbot**), since they do not show anything in the player's screen. Also, they are used in the competitive mode of the game.

Because of this, the **main goal** of this project is to create a system which is capable of distinguish aimbot from humanized aiming. Also, this system will have to respect the privacy of the players and not be intrusive. This will be the second goal.

For constructing the system, Artificial Intelligence will be used, and more precisely, supervised learning. In supervised learning, the system will build a function from training data, which will be data with pre-assigned classes.

Final model will be chosen after doing many tests, where the algorithm used and the dataset will be changed. It will be the model with the highest percentage of correct classified instances and has less false positives. The different models will be generated with Weka and RapidMiner, so .arff files will need to be generated.

The **instances** of the system will be the kills that a player does in a game (each time he kills an enemy). This information will have to be obtainable from the game somehow, during the game or after it.

The **classes** of the instances will be *cheater* or *skilled*. These classes belong to kills, so if a player gets a kill with *cheater* as its class, he will be considered as a cheater.

The attributes of each instance will have to provide enough information about the kill, to determine the class of the instance, i.e. determining if the mouse movement is humanized or not.

The system will need to have instances from cheaters, so a cheat will be developed. This cheat will be a software, which, through operations of reading and writing into CS:GO process memory, will be able to assist a player while trying to aim an enemy.

All the instances will be taken from a player trying to kill someone by a headshot, with the same weapon.

Since there can be, and it is almost safe to say there will be, a case where a skilled player gets classified as a cheater (false positive), the system will not be able to take the decision of banning a player from the videogame. In this case, the system will send the case of the **suspected** player to Overwatch.

And this is the last goal, minimizing the number of false positives.

State of art

By researching other projects, it is concluded that there is a high importance on keeping the videogames out of cheaters, since it is told in every introduction and conclusion of the works which treat this.

The other important aspect to highlight is the transcendence of the Artificial Intelligence in videogames, since it is present in almost every one of them.

Implementation

Aim-assistant

As explained before, a cheat, and more specifically an aimbot, has to be made. It will be developed in C++ language since it will be a program that will run infinite loops, where there will be reading and writing operations, and it will need to be as fast as possible to get the best results.

In this case, an external cheat will be the one developed, but there are other types of cheats (in regards of how it operates with the dynamic allocated memory of the game). This type of cheat is an external executable that has nothing to be with the game, that creates a **handle** to the process where it is required to modify the memory, without "sticking" to the process.

The next move is locating the memory addresses of what is needed to read or modify. It is known, because of the way the game is developed, that every memory address has the following format:

Initial memory address of the module "client_panorama.dll" (or "engine.dll") + offset of the data or structure that is needed

The problem is that the address of the module changes each time CS:GO is opened, since the image of the process (*csgo.exe*) is dynamically allocated in the system, as well as the modules of the process.

However, it can be solved by attaching the process with a handle, and searching for the module inside the process by its name. When the module is found, it will be possible to obtain its address.

Once the address of the module is known, somehow, the offsets of every single structure that are needed to code the program will have to be obtained.

To obtain these offsets, there would have to do reverse engineering with the game opened and seek for them. Nonetheless, there is an open code tool called ***hazedumper*** that can obtain every offset in the game with only being ran with CS:GO opened.

Now that all the addresses can be obtained, there is the need to set the functions that will be used to read and write into the process memory. In this case, *ReadProcessMemory* and *WriteProcessMemory* functions from Windows will be used.

In this point, all the things needed to code the cheat are had. The next step is coding the aimbot, which, in a short explained way, will do the following:

- The cheat will read a file named "*options.txt*" that allows the player to change the options of the cheat. The options can be changed in game if the file is changed and the key F8 is pressed.
- It will store the team, remaining life and position of every player in the game, including the owner of the cheat.
- It will search for the closest enemy (checking that the player is not from the same team) to the crosshair of the player, making mathematical operations with the position of the players and the view-angles of the owner of the cheat. In this calculations, the distance of the enemy player will be considered.
- The angle needed to add to the player's actual view-angle to place the crosshair in the enemy's head will be calculated, again, with mathematical operations, and taking in consideration that the angles in this game are not displayed from 0º to 360º, but from -180º to 180º.
- It will be checked if the key which activates the aimbot is pressed, and if it is, it will move the crosshair to the enemy's head at the speed that is set in the options. Also, the recoil of the weapon will be calculated.
- To finish cheat's process, the player will have to press the key F6.

It is possible that this system has troubles with some antivirus because the external process (cheat) is trying to access to the game's memory, and it can be detected as a threat. In this case, the executable of the cheat would have to be added to the exclusions of the antivirus.

Demo to csv converter

Now that the cheat is coded, there is the need to have a system that obtains the instances that the model will use in a format it can understand (*arff* format).

For that, *demo* files (file that contain the visual repetition of the game) will be recorded with an in-game command used as "*record nameofdemo*".

Then, a program capable of extracting the information of that file will be used with a few modifications. In this case, the company of the videogame itself, Valve, provides a tool for this task called "*demoinfogo*".

This information will be extracted to a *.csv* which will contain thousands of lines and will need to be simplified and formatted so Weka or RapidMiner can understand the data.

Csv to arff converter

In the *csv* file, among all the information, there are structures which contain the updates of the angle in every tick (or they do not, if there is no angle update in the tick). Also, the file contains the moment when a player is killed, and all the details from the kill.

This information is the one that will be used for coding the program that will be converting this *csv* file to an *arff* file. The output *arff* file will have the following modifiable properties:

- The number of angles stored before a kill used in the program.
- Raw angles or just the angles updates.
- The type of output instance (angles or velocity of the movements).
- The number of kills in an instance.

Arff manager

The last program coded will be a software that will make easier to operate with the produced *arff* files. It will have two options:

- Printing the header of an *arff* file to another file.
- Appending the data of an *arff* file to another file.

Also, a visual interface will be developed for each program (except the cheat) in C# language using *Windows Forms*, so they are easier to interact with.

Testing and results

Functionality check

First, the tests that will determine that the system works correctly (all the programs) are ran.

The obtained results show that all is working correctly and the system is ready to start processing the data that is given.

Data collection

Before running the system, the data (demos of the games) must have been collected. For that, six players with different level of skill (but with high skill level predominantly) will kill 500 bots in a map made for training the aim with bots called “Aim botz”. These players will not use any external program, only their aim.

By this, 3000 skilled instances will be collected, so, to make it fair, 3000 cheater instances will have to be collected as well.

For this task, the developed cheat will be used with two different settings. The first settings will try to adjust a little bit to what a human movement of the mouse looks like (higher smooth, slower moves). These setting will be called *legit cheating*, and will be harder for the system to recognize. The second settings will be blatant, so the system is able to recognize them more easily than the legit ones (lower smooth, faster moves). These settings will be called *rage cheating*.

As the behaviour of the cheat will always be the same (or almost the same), regardless the player who uses it, the 3000 instances (kills) will be recorder by the same player, the author of this project.

The system, as far as possible, will have to be able to classify both types of cheating correctly.

Tested algorithms

The different algorithms that will be used to test the system will be the following ones: *BayesNet*, *Naive Bayes*, *SimpleLogistic*, *SMO*, *Ibk*, *PART*, *J48*, *RandomTree*, *RandomForest* and *MLP* (MultiLayer Perceptron).

Each of these algorithms is available in Weka, so it will be the program to run these tests. The parameters of the algorithms will be the default ones, except in MLP, where a learning rate of 0.15 will be used, with 3 different number of cycles: 250, 500 and 1000. So, as MLP has 3 tests, there is a total of "12 algorithms".

Lastly, a few tests will be ran in RapidMiner using Deep Learning. Due to the high time that has to be used to compute this algorithm, few tests will be ran.

Datasets tested

There will be a total of 192 datasets, where the properties of each one will be the following ones:

- ***Angle type***. There are two different forms of collecting the angle:
  - Raw angles.
  - Angles updates.
- ***Instance type***. There are four type of instance:
  - Raw angles.
  - Raw angles and total speed of the movement.
  - Velocity between each movement, and total velocity and deviation.
  - Partial velocities and deviations, and total velocity and deviation.
- ***Kills per instance***. From 1 to 4 kills per instance.
- ***Angles before a kill***. The closed set {*5*, *10*, *15*, *20*, *25*, *30*} will be used.

Testing method

As there are 12 different algorithms and 192 different datasets, there will be, at least, 2304 tests, i.e. the number of combinations that are possible.

These tests will be ran in the tool called *Experimenter* that Weka has. This tool allows the user to run tests simultaneously and making comparatives easily, with different options.

Each test will train the model with 75% of the instances, and test it with the other 25%. The instances will be randomized by Weka.

If one of the models shows a possible improvement, further tests will be done, trying to maximize the results.

Also, best models with only 1 kill will be shown and analysed individually, since they are more versatile.

Cross-validation with 10 folds will be done with the best models. This validation method is meant to be more effective determining how good a model is, but it requires more computation.

Lastly, a test with 1000 instances, having 500 of each class will be done with the best models. These instances are totally new collected ones, and there will be 300 of them with even more blatant cheat options and 200 with even more human-like cheat options, again, toggling the movement smooth. With this, the behaviour of the system with data that is not tested with, is meant to be seen.

Global results

Firstly, to see roughly which algorithms and datasets seem to give the best results, global comparisons are made.

In these comparisons, the datasets with instances that show angles and not velocities seem to work better, as well as instances with a higher number of angles used in the conversion process or instances with a higher level of kills. Nonetheless, a model with only one kill per instance would be preferred since there could be games where a player only gets one kill.

In the other hand, the best algorithms are *RandomForest* and *BayesNet*, with results over 90% of correct classification, followed by *J48* and *PART*, obtaining a results near this percentage.

However, these global results are just indicators of the obtained results on the tests, and will not determine the behaviour of each algorithm with the different datasets, and vice versa. Despite this, they seem to be promising, since obtaining such high average percentages, means that there will be even higher ones to compensate the lower ones.

Once global results are shown, the following task will be finding the best individual models (combinations of algorithms and datasets).

Best models

The best individual results are shown by *BayesNet*, *RandomForest* and *DeepLearning*, as expected.

In these results, *DeepLearning* was able to hit a 96,34% of correct classification, 97,63% by *BayesNet*, and *RandomForest* was able to hit 97,38% and 98,17 with two different datasets. Again, the following best algorithms are *J48* and *PART*, obtaining around 95% of correct classification.

In these results, the global conclusions obtained in the previous section are not fully met, since half of the datasets used by these models use velocities instead of angles, and the half that use velocities use raw angles instead of angles updates (which are used by the other half).

However, all of them use 4 kills and more than 20 angles per instance, as expected in the global results.

Then, cross-validation with 10 folds is applied to these models, obtaining 98,10% with *DeepLearning*, 95,54% with *BayesNet*, and 95,15% and 95,54% with *RandomForest*.

Once best models are shown, best models with only 1 kill per instance will try to be obtained, since, as explained, models with 1 kills are preferred over more kills because there could be games where a player gets only 1 kill.

The best results with only 1 kill, again, are obtained by *BayesNet* and *RandomForest*, with 95,75% and 95,74% of correct classification, respectively, using in both cases cross-validation with 10 folds. Also, there is a case where the dataset with only 5 angles used obtains a good result (which is against the predicted with global conclusions).

A possible improvement is shown with both of these results, since they tend to be better as the number of angles used in the conversion is increased. So, the procedure will be increasing this number until the correct percentage of classification stops increasing as well.

With *RandomForest*, the best result after the increase is shown at 40 angles, with 95,88% of correct classification, which is almost the same than the best results with more than 1 kill per instance.

However, *BayesNet* was able to get the best result with 99,40% using 256 angles, which represent the aiming of the player 2 or 4 seconds before a kill (since the servers have 64 or 128 ticks).

Also, this last model is able to get 99,80% of correct classification training the system with all the instances, and using the dataset with 1000 new instances (as explained in the "Testing method" section) as test. This indicates that the system works perfectly with instances that was not trained with. The only 2 instances that failed to be classified, are from the new even more humanized settings of the cheat. This would probably be fixed if the model was trained with all the possible settings of the cheat.

Future works

In this work, a series of simplifications have been made, which, although it has been shown in previous sections that they do not influence the development and behaviour of the system, they make easier in some way to obtain better results due to the fact that the testing environment is closer.

The biggest different that this made, was that a fewer number of instances was needed to represent better the total set of possible cases in the data, ergo, it could be possible to obtain the same results (or similar ones) if these simplifications were not made, and more instances were collected.

These are the simplifications that were made:

- Using the same weapon to obtain the data.
- Trying to aim only to the head.
- Using the same map to obtain the data.

Once these restrictions are deleted, the only remaining process would be the harvest of a really big number of instances. For this, instances with every single weapon, aiming to every possible angle and in every map would have to be obtained. Also, percentage would increase if all the space of instances is represented.

Conclusion

The first and the most important thing to conclude, is that all goals and objectives were fulfilled after building the system.

This means that the system is able to detect cheaters correctly, with a hit rate higher than 99%. Also, this system uses only demos from recorded games, and because of that, the system is not intrusive at all, prioritizing the privacy of the players, being this another of the established goals.

Another objective was obtaining a low number of false positives, incriminating unjustly skilled players who do not use cheats. This was met with all the best models, where the majority of the fails in classification were identifying cheaters as skilled players.

However, the system is not and could not be perfect, and because of that, if a player gets classified as a cheater, the case of that player (the demo of the game) would be sent to Overwatch (the platform where players review the cases of suspected players).

Also, the best model uses one kill and there are other really good models that uses also only one. These are the best results that could have been obtained, since the models that use only one kill are more versatile than the other ones.

Lastly, it is considered that the student has been capable of applying the knowledge acquired throughout the degree to a problem of a scientific nature, and perfectly applicable to a real system. In addition, it has been necessary to carry out a prior

investigation and an acquisition of new knowledge (such as reading and writing in the memory of a process), thus demonstrating a self-taught capacity, which is considered very relevant in the computer field.

Keywords – cheat, competitive, ban, memory, artificial intelligence, supervised learning, videogame, angle.

RESUMEN

En el presente proyecto de fin de grado se ha desarrollado un sistema de detección de trampas o, en términos internacionales, *cheats*, en el videojuego de disparos en primera persona llamado *Counter-Strike: Global Offensive* (CS:GO).

El uso de trampas en los videojuegos arruina la experiencia de los jugadores que no las utilizan, ya que los que sí lo hacen, obtienen una ventaja visual o de ayuda de asistencia en el control del ratón sobre los jugadores que únicamente utilizan su habilidad.

Además, el presente videojuego está dotado de una gran importancia a nivel internacional, siendo uno de los más presentes en competiciones de deportes electrónicos. En dichas competiciones están presentes *cheats* realizados por programadores profesionales que son capaces de saltarse, en gran medida, la seguridad impuesta por las mismas mediante los sistemas *anti-cheat*. Cabe destacar que la suma total de dinero a repartir en las competiciones anuales, superó los 11 millones de dólares en 2018, de ahí la importancia de mantener una escena competitiva limpia, donde los jugadores ganen de forma lícita el dinero.

Este proyecto, enfoca su objetivo en crear un sistema capaz de detectar mediante inteligencia artificial, y, concretamente, aprendizaje supervisado, los *cheats* de asistente de puntería, o, en términos internacionales, *aimbot*. Dicho asistente de puntería permite al jugador que lo utilice obtener una ayuda en el control de movimiento del ratón. Esto se produce mediante operaciones de lectura y escritura de memoria, en las que se obtiene información del videojuego que se almacena de forma dinámica en la memoria del computador, y se sobrescribe dicha información con la nueva, aquella que otorga ventaja al jugador.

Para ello, se crean y se utilizan programas que captan la información más relevante de una partida, y convierten dicha información relevante en un formato legible para su tratamiento en programas de generación de modelos de aprendizaje supervisado.

Finalmente, se consiguen varios modelos que consiguen clasificar más de un 95% de las instancias correctamente.

Palabras clave – trampa, competitición, memoria, inteligencia artificial, aprendizaje supervisado, videojuego.

ÍNDICE DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

En este capítulo se describe la introducción del trabajo de fin de grado. En particular, se contempla la motivación del mismo, la descripción del problema planteado, el objetivo que se persigue, el marco regulador y el entorno operacional sobre el cual se realiza el trabajo. Por último, se detalla la organización del documento y se incluye una lista de todos los acrónimos utilizados a lo largo del trabajo, así como otra lista de las definiciones de los términos más relevantes para la comprensión del documento.

1. 1. Motivación

El presente trabajo se desarrolla utilizando como base el videojuego CS:GO (*Counter Strike: Global Offensive*).

El CS:GO es un videojuego de disparos en primera persona desarrollado por Valve Corporation (compañía propietaria de la plataforma de distribución de videojuegos más importante a nivel mundial actualmente, *Steam*), en cooperación con Hidden Path Entertainment. Concretamente, este es el cuarto videojuego de la saga Counter-Strike, con fecha de lanzamiento oficial el 21 de agosto de 2012, habiendo conseguido optimizar la jugabilidad a la par que los gráficos frente a sus predecesores.

Esta saga, ha sido siempre pionera en los videojuegos de disparos en primera persona, junto con otras sagas como *Call of Duty* o *Battlefield*, consiguiendo por norma general una media de jugadores *online* (en línea) mayor a estas, debido a que estas últimas, sacan al mercado nuevas versiones de sus juegos más asiduamente (por ejemplo, *Call of Duty* estrena un juego al año).

En el mes de noviembre de este año, el CS:GO ha obtenido una media de 310.085,4 jugadores *online*, y un pico de 546.031 [1] . Estas cifras, no son representativas del número de jugadores que poseen el videojuego, dado que hay jugadores que lo juegan más puntualmente, pero son orientativas en cuanto a la repercusión del mismo.

En este videojuego, al igual que en otros muchos, se realizan competiciones en las que los jugadores, en equipos de 5 personas, intentan vencer al resto de equipos en la modalidad llamada "demolición". En esta modalidad, el desarrollo y las reglas de la partida son los siguientes:

- La partida se divide en rondas con un tiempo de duración de aproximadamente 2 minutos. En cada ronda todos los jugadores reaparecen en sus respectivas bases.
- Existen dos bandos en la partida, donde cada bando queda representado por uno de los equipos. Uno de los bandos, será el de los terroristas, y el otro, el de los antiterroristas.
- El objetivo del bando terrorista, será conseguir adentrarse a una de las dos posibles localizaciones de las que dispone cada mapa para poder plantar una bomba, conseguir plantarla y que se acabe el tiempo restante antes de la explosión de la misma (bien

porque no quedan antiterroristas vivos, o porque ha explotado la bomba). Además, también ganarán la ronda si matan al equipo contrario antes del tiempo de ronda.

- Por el contrario, el objetivo del bando antiterrorista, será conseguir detener las intenciones de los terroristas, es decir, o exterminar al bando contrario, o impedir que planten la bomba antes de que se venza el tiempo de la ronda, o desactivar la misma una vez esté plantada.
- La base del bando antiterrorista está localizada más cerca de las localizaciones del plantado de la bomba, de tal forma que pueden llegar antes a las mismas para defenderlas.
- Los equipos jugarán 15 rondas en un bando, y después cambiarán los roles. Ganará la partida el primer equipo en llegar a las 16 rondas ganadas.
- Las partidas transcurren en *ticks*, existiendo servidores de 64 y de 128 *ticks* por segundo. En cada *tick* de la partida, se actualiza la información del servidor de juego.

Hay otros factores más difíciles que transcurren durante la partida, como la economía de cada jugador, así como la tipología de las armas disponibles, pero en este caso no se consideran relevantes para el entendimiento del documento.

A continuación, en la Figura 1, se muestra una captura de pantalla del videojuego, con la finalidad de explicar los elementos visuales más relevantes, habiendo numerado dichos elementos en la captura de pantalla.

Figura 1. Captura de pantalla de CS:GO

1. Cada jugador dispone de una vida y una armadura, que tendrán un valor por defecto de 100. Cada arma, realiza un determinado daño al jugador enemigo, que hace que ambas unidades disminuyan en relación al mismo.

2. El arma sostenida por el jugador, y en la esquina inferior derecha, el número de balas de las que dispone en el cargador, y el número de balas de las que dispone en reserva.
3. Jugador del bando terrorista.
4. Jugador del bando antiterrorista.
5. *Crosshair* (puntero). Este elemento dependerá del ángulo de visión del jugador, estando siempre en el centro del mismo. El ángulo de visión se controlará moviendo el ratón, y para matar a un enemigo, se deberá intentar poner el *crosshair* encima del enemigo, y disparar el arma con el control pertinente (por defecto será el click izquierdo del ratón). Si el puntero se posiciona sobre la cabeza de un enemigo, el arma infligirá un valor de daño más alto al ser disparada. Además, dependiendo del tipo de arma, se adoptará un método de disparo u otro, pero en este caso, únicamente emplearemos el modo automático, en el que, si se mantiene la tecla de disparo del arma, se vaciará el cargador de la misma (teniendo que controlar el retroceso del arma mientras esta se vacía).

Aunque las competiciones se realizan en muchos videojuegos, en este en concreto, la cantidad de dinero invertida en premios cada año es muy elevada. En la Tabla 1, se muestra una recopilación de las competiciones más importantes de CS:GO en 2018 [30]:

Tabla 1. Premios de las competiciones de CS:GO en 2018

Competición	Fecha	Premio ($)
World Electronic Sports Games World Electronic Sports Games 2017	Mar 13 - 18, 2018	1.500.000
FACEIT Major FACEIT Major: London 2018	Sep 5 - 23, 2018	1.000.000
ELEAGUE ELEAGUE CS:GO Premier 2018	Jul 21 - 29, 2018	1.000.000
ELEAGUE ELEAGUE Major: Boston 2018	Jan 12 - 28, 2018	1.000.000
ESL Pro League ESL Pro League Season 8 - Finals	Dec 4 - 9, 2018	750.000
ESL Pro League ESL Pro League Season 7 - Finals	May 15 - 20, 2018	750.000
Esports Championship Series Esports Championship Series Season 6	Nov 22 - 25, 2018	660.000
Esports Championship Series Esports Championship Series Season 5	Jun 8 - 10, 2018	660.000
Intel Extreme Masters Intel Extreme Masters XII - World Championship	Feb 27 - Mar 4, 2018	500.000
StarSeries i-League StarSeries & i-League CS:GO Season 6	Oct 7 - 14, 2018	300.000
ESL One: Cologne 2018	Jul 3 - 8, 2018	300.000
StarSeries i-League StarSeries & i-League CS:GO Season 5	May 28 - Jun 3, 2018	300.000
StarSeries i-League StarLadder & i-League StarSeries Season 4	Feb 17 - 25, 2018	300.000
EPICENTER EPICENTER 2018	Oct 23 - 28, 2018	295.000
Intel Extreme Masters Intel Extreme Masters XIII - Chicago	Nov 6 - 11, 2018	250.000
ESL One: New York 2018	Sep 26 - 30, 2018	250.000
DreamHack Masters DreamHack Masters Stockholm 2018	Aug 29 - Sep 2, 2018	250.000
Intel Extreme Masters Intel Extreme Masters XIII - Sydney	May 1 - 6, 2018	250.000
DreamHack Masters DreamHack Masters Marseille 2018	Apr 18 - 22, 2018	250.000
ESL One: Belo Horizonte 2018	Jun 13 - 17, 2018	200.000
ESL Pro League ESL Pro League Season 8 - Europe	Oct 2 - Nov 14, 2018	105.000
ESL Pro League ESL Pro League Season 7 - Europe	Feb 13 - Apr 26, 2018	105.000
ESL Pro League ESL Pro League Season 8 - North America	Oct 2 - Nov 14, 2018	87.000
ESL Pro League ESL Pro League Season 7 - North America	Feb 13 - Apr 26, 2018	87.000
Total		**11.149.000**

Como se puede ver, tanto el número de competiciones, como el premio de las mismas (siendo un total de aproximadamente **11 millones de dólares**), dotan de gran importancia al videojuego sobre el que se realiza el trabajo.

Con esta enorme cantidad de torneos y de premios, a algunos jugadores profesionales les surge la idea de contratar los servicios de algún programador de ***cheats*** (trampas), que sean capaces de burlar la seguridad impuesta por los programas ***anticheat*** (anti-trampas) de los que dispone el videojuego y/o competición (dado que la competición puede aportar sus propios *anticheat*). Prueba de ello, son algunos de los jugadores profesionales baneados (expulsados) durante o después de alguna competición importante, con premios como los expuestos en la *Tabla 1*.

Podemos encontrar el último caso, donde Nikhil "forsaken" Kumawat fue **baneado** durante la competición *eXTREMESLAND ZOWIE Asia CS:GO 2018* [2], la cual estaba dotada con un premio de 100.000$. Y, el caso más famoso, el del jugador Hovik "KQLY" Tovmassian siendo baneado del juego por el *anticheat* VAC (Valve Anti-Cheat) después de ganar la competición DreamHack Stockholm 2014, con un premio de 30.000$ [3].

Este es el primer motivo por el cual se necesita una alta eficacia por parte de los sistemas ***anticheat*** en su tarea de encontrar ***cheaters*** (jugadores tramposos), dado que, si no fuera así, los jugadores que participen en estas competiciones, podrían obtener con mayor facilidad los premios de las mismas, de forma injusta, ilícita e inmoral.

El segundo motivo por el cual se requieren sistemas eficaces de detección de *cheats*, es el uso constante de dichas trampas en el videojuego. Los desarrolladores del juego han admitido en varias ocasiones que el número de *cheaters* en el CS:GO es uno de los mayores problemas (si no el mayor) a los que enfrentan en este momento. Tal es así, que entre el 95 y el 97% de los correos que recibe Valve sobre quejas de CS:GO son sobre la cantidad de *cheaters* que hay en el juego [4].

Según la información proporcionada por Valve, en 2018, se jugaron una media de **600.000 partidas competitivas** (modalidad del videojuego para cualquier jugador que posea el mismo) al día [4]. Dicha modalidad de juego, al disponer de un factor competitivo donde el jugador busca subir su rango (nivel), fomenta de forma indirecta el uso de *cheats*.

Al ser una modalidad de juego que fomenta el uso de *cheats* para avanzar, y en la que un jugador disputa una partida con otros 9 jugadores adicionales (es decir, 10 en total), las probabilidades de que en una partida se encuentre un *cheater* aumentan gradualmente cuanto mayor es el porcentaje de *cheaters* respecto a la población activa de jugadores totales en el videojuego, como se puede observar en la Figura 2, donde se ilustra la probabilidad de compartir partida con uno de ellos teniendo en cuenta el porcentaje que puede haber en el videojuego.

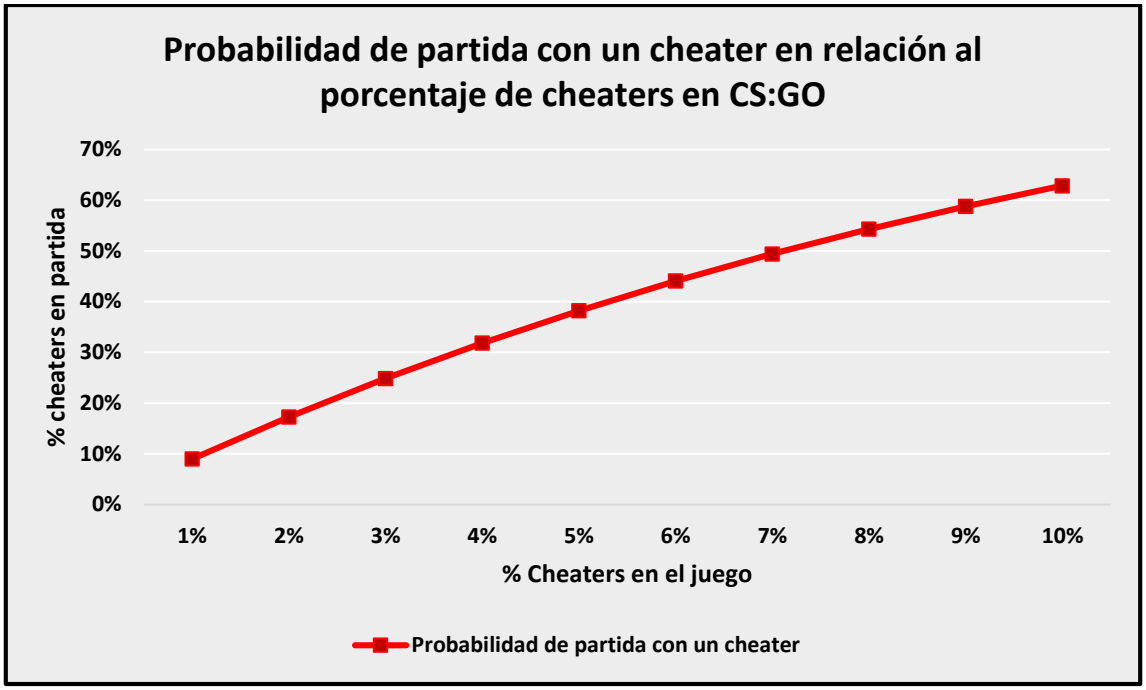


Figura 2. Probabilidad de partida con un cheater en relación al porcentaje de cheaters en CS:GO.

Es decir, si se supone que el porcentaje de *cheaters* en el juego es del 1%, aproximadamente en una de cada 10 partidas el jugador en cuestión tendrá que jugar con uno. No obstante, se estima que este número es mayor, con lo que, aunque el videojuego tan solo disponga de un 5% de *cheaters*, las probabilidades de que un jugador se cruce con uno en una partida son cercanas al 40%, y, por tanto, en aproximadamente 4 de cada 10 partidas tendrá que lidiar con uno de ellos.

Se sabe que el porcentaje de *cheaters* en el juego, a pesar de desconocerse la cifra exacta, es bastante elevado (o al menos lo suficiente para que muchos jugadores se quejen de la situación del videojuego), ya que, como se ha dicho, es uno de los problemas más importantes del CS:GO. Por ello, la cantidad de partidas competitivas en las que un jugador tiene que compartir partida con un *cheater*, ya sea a favor o en contra, es demasiado elevada, y es algo por lo que el juego recibe repetidamente quejas.

A esto se suma que, en diciembre de 2018, Valve decidiera que el videojuego sea accesible de forma gratuita [5], pasando a ser este un F2P (Free to Play –gratis–), en lugar de costar sobre los 15 euros como costaba antes de esta decisión. Debido a esto, la accesibilidad al videojuego es más fácil, así como lo será utilizar *cheats* en él sin temor a ser baneado, puesto que, si esto llegara a ocurrirle a un jugador, podría adquirir otra copia del videojuego de forma gratuita.

Por lo tanto, tanto por motivos de competiciones profesionales, como por motivos de reducir el número de *cheaters* en el juego, y, más concretamente, el número de *cheaters* en las partidas del modo competitivo (dado que en otras modalidades de juego es menos común), es necesario encontrar algún método que haga frente a este problema.

1. 2. Descripción del problema

Por lo explicado en el epígrafe anterior, el problema principal de CS:GO son los *cheats* y la detección de los mismos. A pesar de que erradicarlos es una tarea imposible, puesto que todos los sistemas de seguridad tienen fallas, es necesario una mejora del que dispone el videojuego, un *anticheat* que sea efectivo encontrando y baneando a los *cheaters*. El *anticheat* utilizado por CS:GO es el Valve Anti-Cheat.

Actualmente existen cuatro métodos para la detección de *cheats* en los videojuegos [6]:

1. **Búsqueda de firma de software**. Se escanea la memoria del juego, así como los procesos activos y los archivos contenidos en el ordenador del jugador, y se busca si existe una firma de software que esté contenida en la base de datos de firmas de *cheats* conocidas y recogidas. Es el método de detección de *cheats* más eficaz si el programador del *anticheat* conoce el funcionamiento de los mismos. El problema de este método, es que es muy intrusivo, puesto que para que sea completo y eficaz, se necesitan los permisos necesarios para acceder a los archivos contenidos en el ordenador del jugador. En el caso del VAC, no escanea los archivos del jugador, por lo que no es muy eficaz. Por otra parte, *anticheats* como el de ESEA (E-Sports Entertainment Association League), sí que lo hacen, y, particularmente este mismo, es conocido por su eficacia a la hora de mantener los *cheaters* a raya.
2. **Detección por estadística**. Se recogen las estadísticas de un jugador (tasa de disparos a la cabeza, precisión de los disparos, tiempo de reacción, etc), y se comparan con la información predeterminada por la que se considera a un jugador limpio (sin *cheats*).
3. **Revisión por parte de otros jugadores**. Si un jugador es considerado sospechoso, será juzgado y dictaminado *cheater* o no por otros jugadores. El CS:GO cuenta con un sistema como este, llamado Overwatch. El método principal por el que un jugador es determinado sospechoso o no, son los reportes (quejas) que recibe un jugador durante una partida de otros jugadores, y es lo que hace que este sistema sea menos eficaz, dada la subjetividad a la que incurre un jugador en este tipo de casos. Por ejemplo, existen un par de casos en los que jugadores profesionales (los cuales no utilizan ni utilizaron *cheats*), fueron baneados injustamente en cuentas secundarias en las que no se conocía que eran ellos [7].
4. **Limitaciones en el videojuego**. Se programan limitaciones en el juego, de tal forma que se previene que los *cheaters* tomen ventaja de ciertos aspectos a los que no deberían tener acceso. Por ejemplo, se limita la información recibida de otros jugadores si no están a una distancia a la que el videojuego considera necesaria para visualizar a un enemigo, o, es el servidor el que procesa y calcula la información del retroceso del arma (ya que antes era procesada por el cliente, y podía ser editada mediante lectura y escritura en memoria).

Hay dos razones por las que el VAC no es eficaz a la hora de encontrar y penalizar a los *cheaters*. La primera, se debe a que los jugadores quieren que se respete su privacidad, y por lo tanto no se puede buscar entre los archivos de los mismos, puesto que supondría una violación de la privacidad. Y la segunda, es que los creadores de *cheats* reaccionan

de forma rápida ante las actualizaciones del VAC (realizando ingeniería inversa), y son capaces de sobreponerse a ellas.

Existen diferentes tipos de *cheats*, pero principalmente se podrían clasificar en dos:

- Ayudas o **modificaciones visuales** en los modelos gráficos del juego. En esta categoría, se podrían incluir el *wallhack* (*cheat* que permite visualizar a los enemigos u otros modelos a través de las paredes) y el *radarhack* (*cheat* que permite visualizar a los enemigos en el mini-mapa de la parte superior izquierda de la pantalla) entre otros. Estos *cheats* son difíciles de detectar por el método estadístico nombrado anteriormente, así como difícil de limitar. El mejor método para detectar ese tipo de *cheat* es el primero, búsqueda de firmas.
- Ayudas o **asistencias de disparo**. En esta otra categoría, estarían, entre otros, el *triggerbot* (*cheat* que dispara automáticamente cuando un enemigo se sitúa sobre el *crosshair*) y el *aimbot* o *aim-assistant* (*cheat* que simula el movimiento del ratón de tal forma que ayude al jugador a situar su *crosshair* sobre algún jugador enemigo –normalmente el más cercano– y/o le ayude a disparar). Estos *cheats*, son más fácil de detectar de forma estadística que los anteriores, no obstante, no está implementado en el videojuego y tampoco es una forma de detección infalible. Estos *cheats* son los más utilizados por los jugadores profesionales, dado que en las competiciones presenciales no podrían utilizar los *cheats* visuales, porque serían detectados por los árbitros de la competición al ser vistos en sus pantallas.

Aunque las modificaciones visuales son muy recurridas en la modalidad competitiva del videojuego, accesible por todos los jugadores, en este caso se enfocará el problema hacia los *cheats* de asistencia de disparo, y más concretamente al ***aimbot***, dado que es lo que más está presente en las competiciones, y es utilizado tanto en competiciones como en la modalidad competitiva. Además, en el diseño de la solución, se deberá tener en cuenta que debe ser un sistema con la mínima intrusión posible en la privacidad del jugador.

1. 3. Objetivo

Una vez identificado el problema, el objetivo consiste en implementar un sistema que sea capaz de identificar de la forma más eficaz posible el comportamiento de un *aimbot* frente a la habilidad de un jugador sin él.

Dicho sistema, se tratará de una IA (Inteligencia Artificial), que, mediante **aprendizaje supervisado** (técnica en la cual se deduce una función a partir de unos datos de entrenamiento, que serán el conocimiento previo del sistema con clases pre-asignadas) clasificará las nuevas instancias, de las que se desconoce la clase.

El método de aprendizaje supervisado del **modelo final** será elegido tras realizar una serie de pruebas y ver qué modelo es el que clasifica un mayor porcentaje de instancias correctamente, y tiene menos falsos positivos.

Las **instancias** o **patrones** del sistema de clasificación serán las muertes realizadas por un jugador (cada vez que un jugador mata a un enemigo), que se deberán poder extraer

de alguna forma de la partida en cuestión, ya sea durante o después de la misma. En principio, cada muerte quedará representada en una instancia, pero no se descarta el anexo de varias muertes en una misma instancia durante las pruebas.

Las **clases**, en este caso, serán *cheater* o *skilled* (con habilidad), y se asignarán a cada una de las instancias. Se decide asignar clases a muertes y no a jugadores, dado que, si al menos una de las muertes pertenecientes a un jugador se consideran de la clase *cheater*, el jugador será considerado como tal.

El formato de los patrones, es decir, la elección de los atributos, deberá permitir al modelo generado ser capaz de distinguir entre ambas clases, es decir, deberá aportar la información que se considere necesaria para distinguir movimientos de ratón de carácter humano (clase *skilled*), de movimientos de ratón "artificiales", generados por un programa externo.

Para obtener las instancias de clase *cheater* con las que se entrenará o con las que se probará el modelo, se deberá disponer (o crear), un software que, mediante la **lectura y escritura del proceso** principal del juego (*csgo.exe*), sea capaz de simular un movimiento de ratón que ayude o asista al jugador a la hora de apuntar a un enemigo.

En este caso, se priorizará tanto en las instancias *cheater* como en las instancias *skilled*, que las muertes que recojan sean de ***headshot*** (disparo en la cabeza). Además, todas las pruebas se realizarán con el **mismo arma**.

Como se puede afirmar con certeza que existirá algún falso positivo (un jugador *skilled* clasificado como *cheater*), pues no existen modelos de clasificación perfectos, cuando el modelo detecte algún patrón clasificado como *cheater*, y por tanto el jugador sea clasificado también como tal, se supone que la partida del jugador en cuestión, sería enviada a una revisión por parte de otros jugadores, es decir, a *Overwatch*, como se ha explicado anteriormente.

Además, en este objetivo, aparte de lo nombrado anteriormente, también se espera que el modelo construido sea capaz de responder de una forma más rápida y eficaz a los métodos implementados actualmente, y contar con la ventaja de poder detectar a los *cheaters*, sin un conocimiento previo necesario del *cheat* en cuestión, algo que sí que requiere por ejemplo la detección por firma de software.

1. 4. Marco regulador

La finalidad del presente trabajo, es implementar una IA que detecte a usuarios que utilizan *cheats* en el videojuego CS:GO. Para ello, será necesario recopilar datos en los que se estén utilizando estos *cheats*. Obviamente, Valve prohíbe el uso de *cheats* en la sección 4 (trampas, acciones ilegales y conducta en línea) del Acuerdo de Suscriptor a Steam [8].

Se pretende respetar dicha prohibición, realizando las pruebas bajo una cuenta que está vetada de los servidores oficiales de Valve, con el *anticheat* VAC desactivado, y en un entorno de recolección en el que no se perjudique a ningún jugador de CS:GO.

Dicha recolección se realizará únicamente con los fines de desarrollar el presente trabajo, el cual está enfocado en, precisamente, detectar dichos *cheats*, por lo que se considera éticamente correcto.

Además, el sistema utiliza datos recogidos por otros usuarios, siendo posible acceder de forma pública a la cuenta de los mismos, donde es posible identificarlos si el usuario dispone de información personal.

Por ello, es necesario que se cumpla la Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal [9]. Además, también deberá cumplir con el Reglamento General de Protección de Datos (RGPD) [10], debido a la reciente incorporación de esta nueva normativa.

Para asegurar el cumplimiento de estas regulaciones, se llevan a cabo las siguientes acciones:

- Se informa a cada uno de los usuarios sobre la recogida de datos, solicitando consentimiento explícito a los mismos sobre el almacenamiento y uso de las repeticiones de las partidas (archivos *demo*), así como para el uso de sus nombres personales y su nombre de jugador en el presente documento.
- Los datos recolectados que contengan dicha información, serán eliminados y/o cancelados cuando dejen de ser necesarios en el proyecto, es decir, cuando se hayan transformado las repeticiones en archivos *csv* y dichos archivos, que también contienen información sobre los usuarios, en archivos *arff*.
- Los usuarios que han participado en la recolección de datos, pueden ejercer sus derechos de acceso, cancelación, olvido, oposición, portabilidad y rectificación en todo momento.
- Se designa un responsable de los ficheros, siendo este el autor del trabajo, encargado del tratamiento y seguridad de los datos.

No obstante, en todos los sistemas informáticos existen una serie de vulnerabilidades que pueden hacer que el sistema deje de funcionar, por lo que se deberá elaborar un plan de riesgos. Dicho plan, se elaborará en el *Apartado 6.3. Plan de riesgos*.

1. 5. Entorno operacional

A continuación, se exponen las herramientas utilizadas, tanto software como hardware, para la realización del trabajo de fin de grado.

La herramienta hardware principal ha sido un ordenador personal con las siguientes especificaciones:

- Procesador Intel® Core™, i7-8700K CPU @ 5.1GHz.
- Memoria RAM de 32GB.
- Chip gráfico NVIDIA GeForce RTX 2080Ti.
- SO Windows 10 Pro de 64 bits.

Las herramientas software utilizadas han sido:

- Paquete ofimático de Microsoft Office Standard 2016:
  - Microsoft Word 2016.
  - Microsoft Excel 2016.
  - Microsoft PowerPoint 2016.
- Entorno de desarrollo integrado (IDE) Visual Studio Professional 2017 [11].
- El videojuego en cuestión, CS:GO (Counter-Strike: Global Offensive).
- Software para el aprendizaje automático, Weka [12].
- Software para el aprendizaje automático, RapidMiner [13].
- Editor de texto y de código fuente libre, Notepad++ [14].
- Extractor de información de archivos de repetición de partidas (*.demo*) de CS:GO, csgo-demoinfogo [15].

1. 6. Organización del documento

El presente documento corresponde a la memoria del trabajo de fin de grado, que recoge todo el trabajo realizado sobre el mismo. La estructura del documento es la que se expone a continuación.

En el Capítulo 1 se realiza una introducción, que incluye la motivación del mismo, la descripción del problema planteado, el objetivo que se persigue, el marco regulador y el entorno operacional sobre el cual se realiza el trabajo. Por último, se detalla la organización del documento y se incluye una lista de todos los acrónimos utilizados a lo largo del trabajo, así como otra lista de las definiciones de los términos más relevantes para la comprensión del documento.

En el Capítulo 2 se realiza un análisis y comparación de las investigaciones que comparten un objetivo similar al planteado en el presente trabajo de fin de grado. Se explicará la relación de dichas investigaciones con el mismo. Por último, se analizará la importancia de la IA en los videojuegos.

En el Capítulo 3 se describen la etapa de análisis y diseño del sistema, donde se establecen las funcionalidades del mismo, y cómo van a ser implementadas. En primer lugar, se describirán los requisitos del sistema, tanto funcionales como no funcionales. A continuación, se especificarán los casos de usos del sistema, tanto en forma tabular, como en forma gráfica. Y, por último, se describirá la arquitectura del sistema.

En el Capítulo 4 se describe de forma detallada el proceso seguido en la implementación de cada uno de los subsistemas que componen el sistema de detección de *cheats*.

En el Capítulo 5 se describen las pruebas unitarias del sistema, el proceso de recogida de datos, la parametrización de los mismos, los algoritmos propuestos para la solución, las distintas pruebas colectivas e individuales a realizar, que medirán la evaluación del sistema, y, por último, los resultados obtenidos y un análisis de los mismos.

En el Capítulo 6 se describen los aspectos relativos a la gestión del proyecto. Se detallan la planificación del trabajo, el presupuesto para el desarrollo de este proyecto en una empresa estimando los costes necesarios y el impacto socioeconómico.

En el Capítulo 7 se detallan las conclusiones técnicas y personales obtenidas a lo largo de la realización del trabajo de fin de grado, así como la valoración sobre los resultados obtenidos y los posibles trabajos futuros.

Por último, se indican la bibliografía y las referencias utilizadas y consultadas para la investigación y desarrollo del presente trabajo de fin de grado.

1. 7. Acrónimos

CS:GO: Counter-Strike: Global Offensive.

ESEA: E-Sports Entertainment Association League.

F2P: *Free To Play*, en castellano, jugar gratis.

FOV: F*ield of View*, en castellano, campo de visión.

HWID: *Hardware ID*, identificador único del hardware.

IA: Inteligencia Artificial.

IDE: *Integrated Development Environment*, en castellano, entorno de desarrollo integrado.

IP: *Internet Protocol*, Protocolo de Internet, identificador único de conexión.

MLP: *MultiLayer Perceptron*, en castellano, Perceptrón Multicapa.

NPC: *Non-Player Character*, es decir, jugadores no controlados por personas.

PID: Process ID, es decir, el número de identificador único perteneciente al proceso.

VAC: Valve *Anti-Cheat*.

1. 8. Definiciones

Aimbot o **aim-assistant**: *cheat* que simula el movimiento del ratón de tal forma que ayude al jugador a situar su *crosshair* sobre algún jugador enemigo –normalmente el más cercano– y/o le ayude a disparar.

Anticheat: programa encargado de detectar trampas en los videojuegos.

Aprendizaje supervisado: técnica en la cual se deduce una función a partir de unos datos de entrenamiento, que serán el conocimiento previo con clases pre-asignadas.

Ban: restricción parcial, total, temporal o permanente impuesta a un jugador por el incumplimiento de alguna de las normas/reglas del videojuego o de la competición en la que se encuentre.

Bot: jugador controlado por Inteligencia Artificial.

Cheat: trampa utilizada en los videojuegos.

Cheater: jugador que utiliza *cheats* en los videojuegos.

Crosshair: puntero del jugador, cuya función es apuntar a los enemigos a los que se requiere disparar.

Deep learning: redes neuronales profundas. Es una rama de la inteligencia artificial que se ocupa de emular el modelo de aprendizaje que utilizan los seres humanos para obtener ciertos tipos de conocimiento.

Free To Play: videojuego al cual se puede acceder de forma gratuita.

Headshot: disparo en la cabeza.

Ingeniería inversa: método por el cual se extrae la información de un determinado archivo y se averigua su comportamiento y funcionalidad.

Offset: número de direcciones añadidas a una dirección base necesarias para acceder a otra dirección, es decir, la diferencia entre la dirección final y la dirección base.

Online: en línea, conectado.

Overwatch: sistema de revisión de jugadas sospechosas en el que dichas revisiones son llevadas a cabo por otros jugadores, que deciden si el jugador en cuestión es *cheater* o no.

Radarhack: *cheat* que permite visualizar a los enemigos en el mini-mapa de la parte superior derecha de la pantalla.

Rango competitivo: nivel del jugador en la modalidad competitiva del CS:GO.

Servidor blade: servidor utilizado para los centros de procesado de datos, diseñado para aprovechar el espacio y reducir consumo.

Skilled; conseguido con habilidad, sin terceros medios o programas.

Tick: unidad de medida de tiempo en la que se actualiza el estado de la partida en el servidor donde tiene lugar la misma.

Triggerbot: *cheat* que dispara automáticamente cuando un enemigo se sitúa sobre el *crosshair*.

VACNet: sistema de deep learning implementado por Valve en el CS:GO, que es utilizado para resolver el problema de los *cheaters* en CS:GO.

Validación cruzada: método de validación de modelos en el que se separa el conjunto inicial de datos en particiones y se realizan tantas pruebas como particiones haya, entrenando con todas las particiones menos una, y realizando el test con la sobrante.

Wallhack: *cheat* que permite visualizar a los enemigos u otros modelos a través de las paredes.

2. ESTADO DEL ARTE

En este capítulo se realiza un análisis y comparación de las investigaciones que comparten un objetivo similar al planteado en el presente trabajo de fin de grado. Se explicará la relación de dichas investigaciones con el mismo. Por último, se analizará la importancia de la IA en los videojuegos.

Cabe destacar que solo se encuentran dos trabajos similares sobre este videojuego, siendo el primero un trabajo con motivos educacionales y de investigación, y, el segundo, un proyecto llevado a cabo por la propia empresa desarrolladora del videojuego (Valve), que actualmente está en funcionamiento.

No obstante, existen muchos casos más de aplicación de aprendizaje supervisado y de Inteligencia Artificial en videojuegos, y, entre ellos se encuentra un caso muy similar a estos dos anteriores y al presente trabajo de fin de grado, siendo su entorno de desarrollo otro videojuego FPS, el *Unreal Tournament 3*.

2. 1. Detección de *cheats* en CS:GO mediante redes de neuronas

El siguiente trabajo de investigación [6] tiene lugar en *New Mexico Tech,* formalmente conocido como *New Mexico Institute of Mining and Technology*. Se trata de un instituto de educación e investigación con especialización en ciencia, ingeniería y tecnología.

Como se puede observar en el título del trabajo en cuestión, y, en el propio documento, el objetivo que define este trabajo de investigación es el mismo que el perseguido en el presente trabajo de fin de grado, detectar a los *cheaters* que utilizan programas externos para la simulación de movimiento de ratón, con la finalidad de ayudar o asistir en el proceso de apuntar y disparar a un enemigo.

No obstante, en su caso, únicamente consideran redes de neuronas artificiales para la creación de distintos modelos de clasificación por aprendizaje supervisado, a diferencia del presente trabajo, cuyo objetivo, entre otros, es comparar la eficiencia de una variedad más amplia de algoritmos, como puedan ser los árboles de decisión o el *deep learning*, por ejemplo.

El planteamiento realizado para presentar el problema es parecido, pues presentan la existencia de *cheaters* en competiciones profesionales, pero no explican la importancia de erradicar también los mismos en la modalidad competitiva del juego, que es donde más abunda este tipo de práctica.

Las instancias que deciden emplear, son las muertes de cada uno de los jugadores, al igual que el presente trabajo, y los datos obtenidos con los que deciden entrenar y probar el modelo, son un total de 2400 muertes realizadas en un mapa de práctica de puntería utilizando el mismo arma en todas ellas (AK-47), en las cuales distinguen 4 categorías, conteniendo cada una de ellas 600 instancias.

- Jugador con nivel de habilidad medio, sin uso de *cheats*.
- Jugador con nivel de habilidad alto, sin uso de *cheats*.

- Jugador con nivel de habilidad medio, con uso de *cheats* de forma obvia.
- Jugador con nivel de habilidad medio, con uso de *cheats*, intentando disimular el uso de los mismos (es decir, con una configuración que se asemeje más al movimiento humano).

Los resultados que obtienen utilizando un 75% de la muestra como entrada para el entrenamiento de la red, y el otro 25% restante para probar la misma, varían entre un 90 y un 98% de acierto, aproximadamente, lo que quiere decir que el objetivo ha sido conseguido. No obstante, al no tratarse de un 100% de acierto en la clasificación de las instancias, no se pueden tomar decisiones sobre el baneo de jugadores basándose únicamente en los resultados obtenidos por la red de neuronas; y en el trabajo analizado, no se identifica, y tampoco se especifica una solución a dicho problema.

Tabla 2. Resultados obtenidos por MLP [6]

N	C	C, V, A	C, V, A, 3
5	73.2% / 32.6%	86.2% / 13.8%	93.9% / 6.1%
10	71.5% / 28.5%	88.5% / 11.5%	97.1% / 2.9%
20	81.2% / 18.8%	93.8% / 6.2%	98.0% / 2.0%

Table 2: Results for MLP. Format is % Correctly Classified / % Incorrectly Classified. N = Number of previous cursor positions, C = Raw cursor positions, V = velocity of cursor movement, A = Acceleration of cursor movement, 3 = Three vectors appended together.

2. 2. VACNet

Se trata de una red de neuronas artificiales implementada por Valve [4] que utiliza *deep learning* cuya finalidad es la misma que la planteada en el presente trabajo.

Al tratarse de un proyecto privado de la propia empresa desarrolladora del videojuego, se tienen pocos datos del mismo, existiendo como material de estudio únicamente una presentación realizada por John McDonald, programador de Valve, realizada en la conferencia de desarrolladores de videojuegos en San Francisco.

En dicha conferencia, se presenta una breve introducción sobre el videojuego, y se presenta también la motivación que les impulsa a la creación de VACNet, que es la misma que se ha descrito en el presente trabajo, la existencia de *cheaters*, tanto en un ambiente profesional, como en las partidas competitivas que transcurren cada día en CS:GO.

Al igual que en el presente trabajo, la red de neuronas se enfoca en la detección de *cheaters* que utilizan el segundo tipo de *cheats* explicado en el presente trabajo, los de asistencias de disparo.

En la misma presentación, también se explica la introducción de una propiedad adherida a cada jugador, llamada *Trust score* (**factor de confianza**), que se plantea como una

primera solución al problema de que los jugadores que no utilizan *cheats* tengan que jugar con los que sí que los utilizan.

Dicho factor de confianza, ayudará a emparejar a los jugadores cuyos factores de confianza sean parecidos, es decir, la finalidad será que los *cheaters* jueguen entre ellos (o los que se cree que son propensos a utilizar *cheats* por determinadas razones esbozadas en la presentación), y los jugadores que no los usan, tengan menor posibilidad de cruzarse con alguno. Con este método, la gráfica explicada en la Figura 2 del presente trabajo, dejaría de tener sentido, pues, los jugadores con un factor de confianza bajo, obtendrían un mayor número de *cheaters* por partida, y los jugadores con un factor de confianza alto, obtendrían un menor número.

No obstante, no se logra solucionar el problema de los *cheaters* con este método, concluyendo que este método solo sirve para mitigar el daño que hacen los *cheaters* en el juego.

También se presentan las limitaciones en el videojuego que se tienen programadas para hacer la programación de ciertos *cheats* imposible, como se ha explicado en el cuarto tipo de *cheats* explicado en el presente documento, pero se llega a la misma conclusión que la obtenida en el caso del factor de confianza, no son soluciones definitivas, y se necesita algo más eficaz.

Por último, antes de llegar al tema principal de la conferencia, VACNet, se presenta VAC (el *anticheat* por búsqueda de firma de software que tiene CS:GO implementado). Primero, se especifica que su tasa de acierto a la hora de banear a los *cheaters* es de "100%" (a excepción de algunas veces, que ocurren falsos positivos, se banean a jugadores legítimos, y posteriormente se rectifican dichos baneos). Pero, sin embargo, no sirve de nada tener un 100% de acierto, si los casos en los que logra detectar a un *cheater* son pocos, que es el caso en el que se encuentra VAC.

El caso es que VAC, implementa una serie de mejoras en alguna de las actualizaciones del videojuego, pero, los desarrolladores de *cheaters*, no tardan mucho en encontrar la solución a dichas mejoras, y poder hacer su *cheat* seguro otra vez. De hecho, se han dado casos en los que en menos de 3 horas después del despliegue de una de las mejoras, ya había desarrolladores de *cheats* que las habían conseguido sobreponer.

Por lo tanto, como los jugadores no quieren que VAC implemente métodos más intrusivos, aunque fueran más eficaces a la hora de detectar *cheats*, se necesita un *anticheat* que no dependa de la detección de un software en concreto, sino del **comportamiento de dicho software**.

Entonces, se presenta VACNet, la red de neuronas artificiales que analizará patrones de comportamiento de asistencia de disparo en los jugadores. Como base de entrenamiento, se utilizan los casos en los que los jugadores han dictaminado si un jugador es *cheater* o no, y a través de ellos se genera un modelo que es capaz de monitorizar las partidas competitivas, y de determinar si alguno de los jugadores de una partida en cuestión es *cheater*.

En lugar de utilizar como patrones de entrenamiento cada muerte de un jugador, como se plantea en el presente trabajo, se utiliza cada disparo de un jugador, en cuyo patrón también se especifica el tipo de arma.

Como se ha explicado, no existe la posibilidad de tener una red de neuronas perfecta, por lo tanto, el caso deberá ser enviado a *Overwatch*, para que sea revisado por otros jugadores. Finalmente, las revisiones de los jugadores son procesadas por un clasificador Naïve Bayes, y se dictamina un resultado. En la presentación se hace especial hincapié en que dicho clasificador, absuelve fácilmente a los jugadores, y los hace difícil de declarar culpables (*cheaters*), lo cual es una ventaja. Únicamente en el 0,2% de los casos, se declara culpable y se banea permanentemente un jugador que no lo es.

Entonces existen dos métodos por los que un caso (partida de un jugador sospechoso de ser *cheater*) puede llegar a *Overwatch*. La primera, es si el jugador ha sido reportado por otros jugadores, y, la segunda, si VACNet ha determinado que dicho jugador es *cheater*.

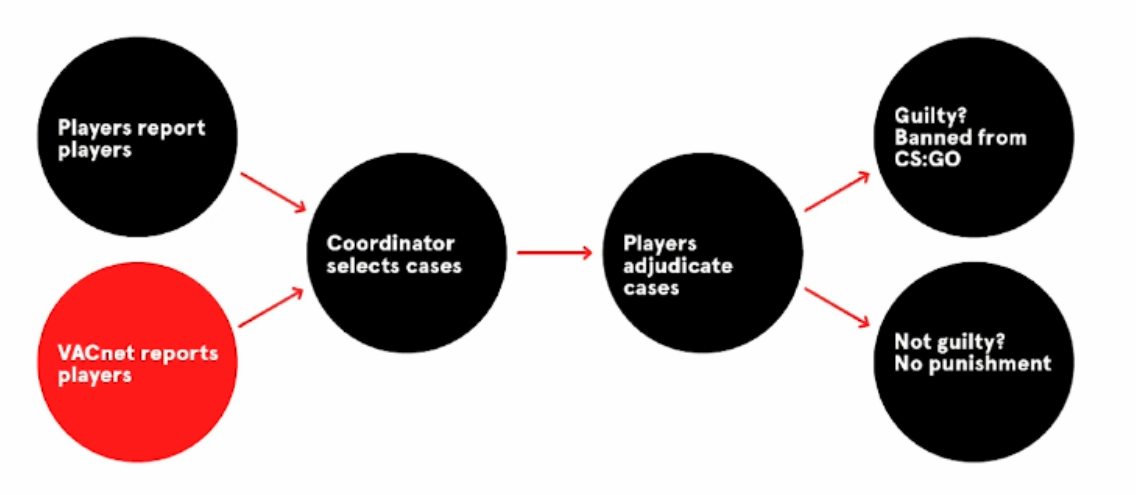


Figura 3. Inclusión de VACNet en la selección de casos enviados a Overwatch [4]

Debido a la subjetividad de los jugadores, el primer método obtiene una tasa de baneo muy baja, concretamente de entre el 15 y el 30%. Es decir, la mayoría de las veces en las que el caso de un jugador llega a *Overwatch* por medio de reportes de los jugadores, el jugador reportado, resulta ser un jugador legítimo, o, al menos no se tiene la evidencia necesaria para banearlo.

Por otra parte, cuando el caso llega a *Overwatch* por medio de VACNet, entre el 80 y el 95% de las veces, el jugador de dicho caso resulta ser *cheater*, y, por tanto, es baneado del videojuego permanentemente (aunque, días antes de la conferencia, se re-entrena el modelo y se consigue un porcentaje aún mayor).

Dicho rango de valores, será al que se aspire en el presente trabajo para poder afirmar que se ha realizado correctamente.

El modelo utilizado es Keras [16], que es una API de alto nivel de redes de neuronas, que permite combinar distintas capas de redes de neuronas, como las convolucionales o las de normalización, entre otras.

La cantidad de información computacional requerida para hacer funcionar dicho modelo durante las partidas de cada día, es decir, monitorizarlas, es de 4 minutos por partida, y existen unas 600.000 partidas al día, como se ha nombrado anteriormente; por lo tanto, se necesitaría 2,4 millones de minutos al día para procesar todas ellas, pero el día dispone únicamente de 1440, así que se necesitarían aproximadamente 1.666,6 procesadores para procesar toda esa cantidad de información.

Por ello, deciden implementar VACNet en un sistema con 4 chasis, contando cada chasis con 16 servidores blade (servidor utilizado para los centros de procesado de datos, diseñado para aprovechar el espacio y reducir consumo), y cada uno de ellos con 54 núcleos y 128GB de RAM. En total, se dispone de 3.456 núcleos (unidades de procesamiento), con lo que se cubren las 1.666,6 necesarias explicadas anteriormente.

Figura 4. Representación física de VACNet (mitad de los chasis) [4]

El entrenamiento del sistema, se realiza en chips gráficos NVIDIA GTX 1080Ti, que se encuentran fuera de los chasis, y que, en el momento de la presentación, eran el último modelo disponible de esta marca y por tanto el mejor.

Gracias a este proyecto, podemos ver una aplicación real de la solución perseguida en el objetivo del presente trabajo de fin de grado.

2. 3. Detección de *cheats* en Unreal Tournament III mediante IA

El siguiente trabajo de investigación [17] pertenece a la *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, que es el evento anual más importante para los investigadores que aplican técnicas de computación y de técnicas de inteligencia artificial a los videojuegos.

Aunque el videojuego en cuestión es otro FPS distinto al presentado hasta ahora, la descripción del problema relatada en la introducción, es exactamente la misma a las observadas hasta ahora, la existencia de *cheaters* en los juegos, que comprometen la experiencia de juego de los jugadores que no lo son. Además, en el propio trabajo se citan otros 4 trabajos más en los que la temática es la misma, la detección de *cheaters* en los videojuegos mediante técnicas de IA.

El videojuego en cuestión se trata de Unreal Tournament III [18], un videojuego un poco anticuado, aunque todavía popular y muy bien valorado por las revisiones de los jugadores.

Figura 5. Captura de pantalla de Unreal Tournamet 3 [17]

El objetivo, es el mismo que el descrito en el presente trabajo, desarrollar modelos capaces de clasificar *cheaters* de jugadores *honestos* (término empleado en el trabajo). Para ello, previamente, al igual que en el presente trabajo, crean un *cheat* configurable con el cual extraerán las instancias con clasificación de *cheater*.

También, otra de las similitudes con el presente trabajo, es la explicación y clasificación de los diferentes *cheats* presentes en dicho videojuego, que son muy similares a los primeros.

La obtención de datos es una de las principales diferencias planteadas. Deciden implementar un servidor que actúa de "recolector" de información de la propia partida que tiene lugar dentro del mismo. Entre dicha información, se encuentra el posicionamiento de los jugadores dentro del mapa y los ángulos donde apunta cada uno de ellos, entre otros.

El pre-procesado de datos y la generación de modelos se realiza en la herramienta RapidMiner, donde emplean 5 técnicas distintas de clasificación: árboles de decisión, Naive Bayes, Random Forest, redes de neuronas y máquinas de soporte de vectores.

Aunque los resultados obtenidos en este trabajo son bastante buenos (ver Tabla 3), la recolección de instancias de entrenamiento es muy pobre (concretamente, 250 instancias de entrenamiento y 39 de test), con lo que el espacio casuístico es muy reducido, y es más fácil obtener buenos resultados.

Tabla 3. Resultados de clasificación en Unreal Tournament 3 [17]

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%**

A favor de este trabajo, está el factor de que la recolección de datos tiene lugar durante la partida, y no después, como se plantea en el presente trabajo de fin de grado. Esto favorece que la clasificación de nuevas instancias se puede llevar a cabo durante la partida, y obtener un resultado antes de la finalización de la misma.

Sin embargo, esto último es irrelevante, dado que lo único que importa en este tipo de proyectos, es obtener el máximo porcentaje de clasificación realizada correctamente.

Por último, se destaca la importancia de minimizar la cantidad de falsos positivos (clasificar un jugador *honesto* como si fuera uno *cheater*), algo que también tiene importancia en el presente trabajo.

2. 4. Inteligencia artificial en videojuegos

En los anteriores apartados, se han expuesto casos en los que se utiliza la IA en los videojuegos con la única intención de detectar *cheaters*, pero, esa sería únicamente uno de los muchos usos que se aplican. En los siguientes apartados, se va a describir la amplia historia de la IA en los videojuegos, de forma resumida, y se expondrán algunas de las muchas aplicaciones de la misma, con tal de destacar su importancia.

2. 4. 1. Historia de la IA en los videojuegos

La IA tiene una larga historia en los videojuegos [24], donde su importancia ha ido creciendo a una gran velocidad.

Uno de los primeros estudios del uso de la IA en los videojuegos, fue publicado en 1959 por Arthur Samuel en su libro "*Some studies in machine learning using the game of checkers*", gracias al cual programó un juego de damas en el que se implementaba por primera vez un sistema que era capaz de utilizar aprendizaje automático.

Este, es uno de los puntos de partida de la IA en los videojuegos, permitiendo así la aparición de juegos con la modalidad de un solo jugador como fueron el caso de *Qwak*, *Pursuit*, *Hunt the Wumpus* o *Star Trek*, donde se comenzaba a almacenar los movimientos que realizaba el jugador y a aprender sobre ellos.

Los avances tecnológicos, como los microprocesadores, permitieron que este campo mejorara añadiendo una aleatoriedad a los enemigos a los que se enfrentaba el jugador, mejorando así su inteligencia, es decir, un proceso estocástico.

A lo largo de los años 70, se consiguieron numerosos avances en el campo de la IA en los videojuegos, en famosos videojuegos como *Space Invaders* (1978) o *Galaxian* (1979), que añadieron funciones hash que conseguían una complejidad adicional en el videojuego, y que aprendían de los actos del jugador.

En los años 80, juegos famosos como el *Pac-Man* (1980) y *Karate Champ* (1984), introdujeron el factor "personalidad" dentro de los enemigos. Esto se puede observar aún mejor en videojuegos como *Madden Football*, videojuego de fútbol donde se intentaban replicar algunos de los aspectos del comportamiento de los jugadores más importantes de la liga.

No obstante, donde realmente se afianzó el uso de la IA en los videojuegos fue en los años 90, con las recientemente aparecidas nuevas y famosas consolas, DreamCast, Game Boy, NEO-GEO, PS-ONE, Sega Saturns, SNES o Super Nintendo. En estas consolas aparecieron videojuegos como *Black and White* o *Creatures*, en los que se empezaba a utilizar las máquinas de estados, decisones en tiempo real, toma de caminos, aprendizaje, etc.

A finales de los años 90, las IA seguían mejorando debido al gran desarrollo e innovación en el mundo de los videojuegos. Por ejemplo, *GoldenEye 007* sería el primer videojuego de disparos en primera persona que introduciría una IA que reaccionaba en tiempo real a las acciones y movimientos de los jugadores, o, *Half-life* (en el que se basa el videojuego del presente trabajo de fin de grado), que introduciría el concepto de cooperación entre enemigos para cubrirse, flanquear al adversario, buscar al jugador, etc.

Figura 6. Half-life (1999) [24]

Ya en el siglo XXI, el videojuego Halo (2001) introdujo en los enemigos el reconocimiento de amenazas, como granadas, precipicios o vehículos, los cuales también podían manejar.

Otro de los avances ocurrió en 2004, donde se utilizaban métodos que leían el comportamiento del jugador para utilizar tácticas militares personalizadas basadas en

dicho comportamiento en su contra, además de otros factores que dotaban de un realismo no visto anteriormente a la experiencia de juego.

Uno de los primeros videojuegos en introducir la planificación en tiempo real, que le permitía adaptarse de forma dinámica a los sucesos del entorno, fue FEAR (2005), que destacó por la inteligencia de los enemigos.

A partir de aquí, y hasta la actualidad, los videojuegos han ido destacando aún más por sus IAs, que cada vez mejoraban la experiencia del jugador, haciendo más realista el comportamiento de los enemigos.

2. 4. 2. Control de NPCs

Uno de los usos más comunes de la IA en los videojuegos es el control de NPCs (*Non-Player Character*, es decir, jugadores no controlados por personas).

Esto permite al jugador poder enfrentarse a un reto, y, por tanto, obtener un entretenimiento (finalidad principal de los videojuegos) sin la necesidad de que haya más jugadores en la partida, es decir, en la modalidad "un jugador". Por ejemplo, uno de los videojuegos más famosos en esta modalidad es *PacMan*, en el que, controlando a un personaje con forma de círculo amarillo, el jugador deberá, en unas determinadas circunstancias, huir de los enemigos (fantasmas) y en otras intentar comérselos. Estos fantasmas, al igual, a veces deberán perseguir al personaje principal, y, otras veces, deberán huir de él. Esto se llevará a cabo controlando a los mismos mediante diferentes métodos de IA.

Además, los NPCs también se encuentran en videojuegos online y/o multijugador, para aumentar la experiencia del jugador y ampliar la jugabilidad del videojuego en cuestión.

Es muy importante que el jugador obtenga un balance equilibrado entre reto (dificultad lo suficientemente alta a la hora de que obtenga sus objetivos) y entretenimiento (que dicha dificultad no haga que el jugador se frustre, al ser demasiado elevada), es decir, que la dificultad que los NPCs (y otros factores) ejerzan sobre el jugador sea la adecuada para ofrecer una resistencia del mismo calibre que la habilidad del jugador.

Y otro de los factores importantes que debería tener una IA a la hora de controlar un NPC, es que, su comportamiento sea, en medida de lo posible, indistinguible del de una persona, haciendo así que la experiencia del jugador mejore.

Por ejemplo, en trabajos como [25] se realiza el test de Turing, en el cual se pone a prueba la capacidad de que los jugadores detecten a la IA como tal y no como un humano. En los resultados, se observa que tan solo se obtiene un 50% de acierto.

En trabajos como [24] o [26] se explican más en detalle los procesos de creación y uso de los comportamientos inteligentes de los NPCs en los videojuegos, así como su historia de forma más precisa. En ambos, se destacan algunos de los algoritmos más utilizados para el desarrollo de IA en los videojuegos, como los árboles de decisión o el aprendizaje por refuerzo, así como los videojuegos que más destacan gracias al realismo y la experiencia que ofrece su IA. Entre algunos de estos videojuegos, se encuentran el famoso

Far Cry 3, donde desde los animales de la selva como los enemigos que pueden aparecer desde cualquier lugar, hacen sentir al jugador como si estuviera en una selva real; o el *Grand Theft Auto V*, donde se podría decir que la inteligencia artificial rebosa vida.

Figura 7. Captura de pantalla de GTA V [29]

2. 4. 3. Búsqueda de caminos

Otro de los usos más comunes de la IA en los videojuegos es la búsqueda de caminos.

Como su nombre indica, esta técnica permite determinar el camino a seguir para ir de un punto a otro en un entorno dado. Decidido un punto de origen y un punto de destino, un algoritmo de búsqueda devolverá una lista de acciones o movimientos (representados en vectores) a realizar para llegar a la meta. El algoritmo más empleado en la actualidad dentro de este campo es el A*.

Este algoritmo, con una heurística admisible, ofrece soluciones óptimas, es decir, el camino más corto posible. A veces puede ser conveniente encontrar dicho camino, pero, otras veces, para dar más realismo al movimiento y a la jugabilidad, puede ser conveniente utilizar heurísticas no admisibles con tal de encontrar caminos que no sean siempre óptimos.

Un ejemplo de búsqueda de caminos en la industria de los videojuegos [27] es el *Age of Empires*, que es un videojuego de estrategia en tiempo real clásico que utiliza cuadrículas para representar las localizaciones de un mapa, y, en el cual, el jugador decide los movimientos seleccionando una casilla de destino. El algoritmo de búsqueda que se emplea en este videojuego es el A*, aunque, en este caso, por una pobre implementación, existen algunos errores de los cuales se quejan los jugadores, como, por ejemplo, cuando un grupo de unidades atraviesa un bosque, y algunas de las unidades se quedan atascadas en los árboles.

Figura 8. Captura de pantalla de Age of Empires II [27]

Otro ejemplo de búsqueda de caminos parecido, sería el caso del *Civilization V*, otro videojuego de estrategia. En este, se utilizan casillas hexagonales para representar el mapa. En este otro videojuego, de nuevo, ocurren algunos fallos debido a una pobre implementación en la búsqueda de caminos. Así que, cabe destacar la importancia en este tipo de videojuegos de una buena implementación de la IA, ya que el videojuego depende totalmente de esta, haciendo uso de la misma en todo momento.

Figura 9. Captura de pantalla de Civilization V [27]

Sin embargo, en estos videojuegos la dificultad de la búsqueda de caminos reside en que se necesita mover muchas unidades simultáneamente, y los algoritmos funcionan mejor con menos unidades. Por ejemplo, en los videojuegos de la saga sobre la que trata el presente trabajo de fin de grado, *Counter-Strike*, solo se involucran unas pocas unidades al mismo tiempo.

Esto se debe a que, cuanto mayor es el número de unidades, mayor será el tiempo de computación necesario para encontrar caminos óptimos para cada unidad, y, debido a la naturaleza del videojuego (estrategia en **tiempo real**), las decisiones deben ser rápidas, y, por tanto, a veces no óptimas.

Por último, otro ejemplo de que involucra masivamente la búsqueda de caminos en tiempo real, es *World of Warcraft*, uno de los videojuegos más famosos de la última década. Por ello, se puede ver la importancia de este uso de la IA en los videojuegos.

2. 4. 4. Generación procedural de contenido

Otro de los usos de la IA en los videojuegos, es la generación procedural de contenido (PCG, siglas en inglés) en los mismos [28]. Esto permite la creación de nuevos tipos de videojuegos dinámicos, en los que la historia, estrategia, o mapa de los mismos varían acorde al contenido generado, apaciguando así la repetitividad y el aburrimiento del jugador.

Esta técnica de generación de contenido permite aumentar la imaginación por encima del pensamiento humano, dejando que los algoritmos creen niveles, narrativas o reglas. Además, el uso de esta técnica también puede servir de inspiración inicial a los diseñadores de videojuegos, que pueden utilizar de base los contenidos creados para crear niveles a partir de dichos contenidos.

El primer videojuego en utilizar PCG fue *Akalabeth: World of Doom*, que utilizaba una semilla para generar el mapa del videojuego. Otro videojuego muy famoso, *Rogue* (1980), es el antecesor de videojuegos muy famosos que utilizan PCG, por ejemplo, el primer videojuego de la saga *Diablo* (1996), desarrollado por Blizzard Entertainment, que fue uno de los primeros de la era moderna de los videojuegos adaptado de *Rogue*.

Figura 10. Captura de pantalla de Rogue [28]

En *Diablo*, PCG se encargó de generar mazmorras aleatorias y distribución de objetos de recolección aleatoria a lo largo de las mismas. Estas generaciones aleatorias siguen un algoritmo con elementos aleatorios, como en el resto de videojuegos existentes hasta ese momento. No obstante, fue pionero en crear dichas mazmorras utilizando gráficos de 2 dimensiones isométricos. Además, dicho sistema también fue el primero en crear objetos de distinta rareza basados en el código de color generado. Numerosos videojuegos utilizan dicho sistema actualmente, como, por ejemplo, *Borderlands*.

Actualmente, esta técnica es utilizada para instanciar los objetos de muchos videojuegos, como los árboles, enemigos, NPCs, ítems, tesoros y más. Un claro y muy conocido ejemplo de videojuego donde se aplica, es *Minecraft* (2011).

En editores de videojuegos o de modificaciones para ellos (mods), los usuarios pueden cambiar los parámetros de PCG para generar contenido personalizado e intercambiar dichos parámetros con otros usuarios.

En concreto, entre algunos de estos editores que son aplicables a varios videojuegos, podemos encontrar *CityEngine* (2016) para la generación de entornos urbanizados o *SpeedTree* (2016) para la generación de distintos árboles o bosques. Este último se utiliza en videojuegos AAA (tope de gama, con muy altos gráficos), como *The Witcher 2*.

No obstante, no se utilizan únicamente en generación de gráficos (a pesar de ser su uso más común), sino que también sirven para generar comportamientos aleatorios dentro de los videojuegos o para generar narrativa.

2. 5. Conclusión

La principal conclusión que se puede extraer de observar todos estos trabajos, es la gran importancia de mantener los videojuegos libres de *cheaters*, pues se hace hincapié en cada introducción y conclusión de los trabajos.

Con esto, los jugadores que no utilizan *cheats* podrán disfrutar tranquilamente del videojuego en cuestión, sin tener que preocuparse de otros jugadores que toman ventaja sobre ellos debido a causas externas. A su vez, esto influirá positivamente en la opinión de los jugadores, y, por ello, más gente se animará a jugar.

En otras palabras, los *cheats* son el mayor problema de los videojuegos en la mayoría de los casos (y más aún en videojuegos donde prima la competitividad), y son los causantes por tanto de la decadencia de los mismos, y de que la gente se canse de la situación y se frustre y/o abandone el videojuego.

La mejor solución para este problema, es el uso de un *anticheat* intrusivo, pero, en la mayoría de los videojuegos, no es implementado dado que a la gente no les gusta la intrusión en su privacidad. Por ello, se escoge el mejor método no intrusivo, la Inteligencia Artificial, que es capaz de detectar a *cheaters* aprendiendo sobre un conocimiento anterior y sobre decisiones ya tomadas.

No obstante, este es un área aún en desarrollo (y prueba de ello es la reciente incorporación de VACNet a CS:GO), y aún hay pocos trabajos sobre este campo.

Por último, se puede destacar la gran variedad de usos de la Inteligencia Artificial en los videojuegos. La IA está presente en casi todos los videojuegos actuales, y, por ello, existen cientos de casos importantes que se podrían exponer, pero se han elegido unos pocos significativos para destacar la gran importancia de esta en los mismos.

Uno de los motivos que hacen esto posible, es que los videojuegos son un entorno de pruebas perfecto para probar los algoritmos de IA. Esto se debe a que cuentan con un conjunto de reglas bien definidas, con metas claras y la posibilidad de evaluar los resultados de forma objetiva y sencilla.

3. ANÁLISIS Y DISEÑO DEL SISTEMA

En este capítulo se describe la etapa de análisis y diseño del sistema, donde se establecen las funcionalidades del mismo, y cómo van a ser implementadas. En primer lugar, se describirán los requisitos del sistema, tanto funcionales como no funcionales. A continuación, se especificarán los casos de usos del sistema, tanto en forma tabular, como en forma gráfica. Y, por último, se describirá la arquitectura del sistema.

3. 1. Requisitos

El sistema deberá cumplir con una serie de requisitos que definan la funcionalidad y comportamiento del mismo. Estos requisitos se dividirán en dos categorías:

- **Funcionales**: son aquellos que establecen detalles técnicos que describen la funcionalidad del sistema. Los requisitos funcionales se describen desde la Tabla X hasta la Tabla Y.
- **No funcionales**: son aquellos que describen las características de la aplicación, es decir, especifican cómo se cumplen los requisitos funciones. Los requisitos no funcionales se describen desde la Tabla X hasta la Tabla Y.

El formato tabular en ambos casos, será el siguiente:

Tabla 4. Plantilla para la especificación de requisitos

Identificador: RT-XX			
Nombre			
Versión		**Dependencias**	
Descripción			
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

A continuación, se explica el significado de cada una de las filas de la plantilla para la especificación de requisitos:

- **Identificador**. Es el código identificativo, unívoco, de cada uno de los requisitos. La nomenclatura utilizada será RT-XX, donde:
  - R: Indica que es un requisito.
  - T: Indica el tipo de requisito de los cuales se distinguen requisito funcional (F) y requisito no funcional (NF).
  - XX: Es el número de requisito dentro de la clasificación.
- **Nombre**. Breve descripción del requisito.
- **Versión**. Representa la versión actual de la definición del requisito.

- **Dependencias**. Enumeración de los requisitos con los que el requisito tiene dependencias, indicados por su identificador.
- **Descripción**. Descripción breve y detallada del requisito donde se detalla su finalidad.
- **Prioridad**. Nivel de importancia en el conjunto de requisitos y que le dan una prioridad asociada en el desarrollo y en la planificación.
- **Estabilidad**. Define si un requisito será fijo durante todo el desarrollo o puede ser susceptible a cambios debidos a la implementación o el diseño.
- **Verificabilidad**. Define la capacidad de comprobar si un requisito se ha incorporado al diseño de manera correcta.

3. 1. 1. Requisitos funcionales

Cada uno de los requisitos funcionales, describe una funcionalidad del sistema. Para describir cada requisito, se utilizará una tabla. Los requisitos funcionales son los descritos a continuación.

Tabla 5. Requisito funcional RF-01

Identificador: RF-01			
Nombre	Obtención de archivo *csv* desde un archivo *demo*.		
Versión	1.0	**Dependencias**	-
Descripción	El sistema *demoinfo* deberá recopilar toda la información necesaria de un archivo que almacene la repetición de una partida en formato visual (archivo *demo*), y guardarla en un archivo de texto *csv*.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 6. Requisito funcional RF-02

Identificador: RF-02			
Nombre	Selección de archivo de apertura *demo*.		
Versión	1.0	**Dependencias**	RF-01
Descripción	La interfaz visual del sistema *demoinfo* deberá disponer de un botón que abra una ventana de selección de archivos con una extensión *.demo*.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 7. Requisito funcional RF-03

Identificador: RF-03			
Nombre	Nombre de archivo de salida *csv*.		
Versión	1.0	**Dependencias**	RF-01
Descripción	La interfaz visual del sistema *demoinfo* deberá disponer de una entrada que almacene el nombre de salida del archivo *csv* al que se ha convertido el archivo *demo*.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 8. Requisito funcional RF-04

Identificador: RF-04			
Nombre	Obtención de archivo *.arff* desde un archivo *csv*.		
Versión	1.0	**Dependencias**	-
Descripción	El sistema *demoParser* deberá recopilar toda la información necesaria para la conversión del archivo *csv* en un archivo *arff* comprensible por el programa Weka. En el proceso, deberá añadir las cabeceras pertinentes.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 9. Requisito funcional RF-05

Identificador: RF-05			
Nombre	Selección de archivo de apertura *csv*.		
Versión	1.0	**Dependencias**	RF-04
Descripción	La interfaz visual del sistema *demoParser* deberá disponer de un botón que abra una ventana de selección de archivos con una extensión *.csv*.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 10. Requisito funcional RF-06

Identificador: RF-06			
Nombre	Selección de clase de patrón.		
Versión	1.0	**Dependencias**	RF-04
Descripción	La interfaz visual del sistema *demoParser* deberá disponer de un botón que abra una ventana desplegable que permita seleccionar entre las dos clases con las que se etiquetará a cada patrón, *skilled* o *cheater*.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 11. Requisito funcional RF-07

Identificador: RF-07			
Nombre	Elección de tipo de ángulo		
Versión	1.0	**Dependencias**	RF-04
Descripción	La interfaz visual del sistema *demoParser* deberá disponer de un botón que abra una ventana desplegable que permita seleccionar entre los distintos tipos de ángulo de los que se dispone. Estos son las modificaciones de ángulos previos a una muerte, y los ángulos en bruto antes de la misma.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 12. Requisito funcional RF-08

Identificador: RF-08			
Nombre	Elección de formato de patrón.		
Versión	1.0	**Dependencias**	RF-04
Descripción	La interfaz visual del sistema *demoParser* deberá disponer de un botón que abra una ventana desplegable que permita seleccionar entre los distintos tipos de formato con los que se podrá almacenar un patrón.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 13. Requisito funcional RF-09

Identificador: RF-09			
Nombre	Tipos de patrón.		
Versión	1.0	**Dependencias**	RF-04, RF-07
Descripción	El sistema *demoParser* deberá disponer de los siguientes formatos para los patrones: ángulos anteriores a una muerte, ángulos anteriores a la muerte y velocidad de movimiento de ratón, velocidades y desviación total de movimiento de ratón antes de la muerte, y velocidades y desviaciones parciales de movimiento antes de la muerte.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 14. Requisito funcional RF-10

Identificador: RF-10			
Nombre	Elección de número de muertes en un patrón.		
Versión	1.0	**Dependencias**	RF-04
Descripción	La interfaz visual del sistema *demoParser* deberá disponer de una entrada de texto que permita elegir el número de muertes que almacenará un patrón.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 15. Requisito funcional RF-11

Identificador: RF-11			
Nombre	Elección de número de *ticks* anteriores a la muerte.		
Versión	1.0	**Dependencias**	RF-04
Descripción	La interfaz visual del sistema *demoParser* deberá disponer de una entrada de texto que permita elegir el número de *ticks* anteriores a la muerte de los cuales se recopilará la información que se almacenará en cada patrón.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 16. Requisito funcional RF-12

Identificador: RF-12			
Nombre	Nombre de archivo de salida *arff*.		
Versión	1.0	**Dependencias**	RF-04 – RF-10
Descripción	La interfaz visual del sistema *demoinfo* deberá disponer de una entrada que almacene el nombre de salida del archivo *arff* al que se ha convertido el archivo *csv*, y de un botón que genere un nombre automáticamente en base a los parámetros anteriores.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 17. Requisito funcional RF-13

Identificador: RF-13			
Nombre	Manejador de archivos *arff*.		
Versión	1.0	**Dependencias**	-
Descripción	El sistema *manageArffs* deberá ser capaz de imprimir las cabeceras en un archivo (con sobrescritura) obteniéndolas de un archivo *arff*, y de juntar el contenido de dos archivos *arff* con la misma cabecera (mismo formato de patrón).		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 18. Requisito funcional RF-14

Identificador: RF-14			
Nombre	Selección de archivo de entrada *arff*.		
Versión	1.0	**Dependencias**	RF-12
Descripción	La interfaz visual del sistema *manageArffs* deberá disponer de un botón que abra una ventana de selección de archivos con una extensión *.arff* que permita elegir el archivo de entrada.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 19. Requisito funcional RF-15

Identificador: RF-15			
Nombre	Elección de tipo de operación.		
Versión	1.0	**Dependencias**	RF-12
Descripción	La interfaz visual del sistema *manageArffs* deberá disponer de un botón que abra una ventana desplegable que permita seleccionar el tipo de operación, que podrá ser impresión de cabeceras, o añadir las líneas contenidas en un archivo *arff* a otro.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 20. Requisito funcional RF-16

Identificador: RF-16			
Nombre	Selección de archivo de salida *.arff* .		
Versión	1.0	**Dependencias**	RF-12
Descripción	La interfaz visual del sistema *manageArffs* deberá disponer de un botón que abra una ventana de selección de archivos con una extensión *.arff* que permita elegir el archivo de salida.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 21. Requisito funcional RF-17

Identificador: RF-17			
Nombre	Botón de ejecución.		
Versión	1.0	**Dependencias**	RF-01, RF-04, RF-12
Descripción	Las interfaces visuales de los sistema *demoinfo*, *demoParser* y *manageArffs*, deberán tener un botón de ejecución, que realizará la operación escogida si los parámetros han sido correctamente rellenados.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 22. Requisito funcional RF-18

Identificador: RF-18			
Nombre	Programa de asistencia de movimiento de ratón.		
Versión	1.0	**Dependencias**	-
Descripción	El sistema *WhatAreThose* deberá ser capaz de, mediante operaciones de lectura y escritura en memoria sobre el proceso *csgo.exe*, asistir al jugador en el movimiento de ratón hacia los enemigos (*aimbot*), siendo capaz de escoger la tecla con la que se activara la asistencia, la parte del cuerpo del enemigo a la que se apuntará con el *crosshair*, la velocidad del movimiento de ratón, un factor numérico que regule la apertura de ángulo respecto a la distancia, y el FOV (*field of view*, en castellano, campo de visión) al que se detectarán los enemigos.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 23. Requisito funcional RF-19

Identificador: RF-19			
Nombre	Carga de opciones.		
Versión	1.0	**Dependencias**	RF-18
Descripción	El sistema *WhatAreThose* deberá ser capaz de cargar desde un archivo de opciones *options.txt*, todas las opciones configurables dentro del sistema, y ser capaz de recargarlas en partida después de ser editadas, mediante la tecla F8.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 24. Requisito funcional RF-20

Identificador: RF-20			
Nombre	Generación de modelos.		
Versión	1.0	**Dependencias**	-
Descripción	Se diseñarán con Weka y RapidMiner modelos con los datos de entrenamiento que serán capaces de clasificar nuevas instancias, que no han sido utilizadas para el entrenamiento de los modelos.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

3. 1. 2. Requisitos no funcionales

Cada uno de los requisitos no funcionales, describe una característica del sistema. Para describir cada característica, se utilizará una tabla. Los requisitos no funcionales son los descritos a continuación.

Tabla 25. Requisito no funcional RNF-01

Identificador: RNF-01			
Nombre	Lenguajes de desarrollo de las aplicaciones.		
Versión	1.0	**Dependencias**	-
Descripción	Las aplicaciones serán desarrolladas en C++, en Visual Studio Professional 2017. En el caso de *WhatAreThose*, por ser un lenguaje rápido en ejecución.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 26. Requisito no funcional RNF-02

Identificador: RNF-02			
Nombre	Lenguajes de desarrollo de la interfaz de la aplicación.		
Versión	1.0	**Dependencias**	-
Descripción	La interfaz de la aplicación será un *Windows Form* desarrollado en el lenguaje C# en Visual Studio Professional 2017.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 27. Requisito no funcional RNF-03

Identificador: RNF-03			
Nombre	Aplicaciones finales .exe.		
Versión	1.0	**Dependencias**	-
Descripción	Todas las aplicaciones deben ser distribuidas finalmente como aplicaciones con extensión *.exe*.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 28. Requisito no funcional RNF-04

Identificador: RNF-04			
Nombre	Archivos demo		
Versión	1.0	**Dependencias**	-
Descripción	Se deben obtener repeticiones de partidas con extensión *.demo* que reúnan una cantidad suficiente de muertes para la clasificación correcta de un jugador. Para ello, en la consola de CS:GO se deberá escribir el comando "*record nombre_demo*".		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

Tabla 29. Requisito no funcional RNF-05

Identificador: RNF-05			
Nombre	Tiempo de respuesta de la aplicación de lectura/escritura.		
Versión	1.0	**Dependencias**	-
Descripción	La aplicación de lectura/escritura en memoria sobre el proceso CS:GO, deberá tener un tiempo de respuesta inmediato, permitiendo al usuario de la misma tener una interacción fluida.		
Prioridad	Baja	Media	Alta
Estabilidad	Baja	Media	Alta
Verificabilidad	Baja	Media	Alta

3. 2. Casos de uso

Un caso de uso consiste en una descripción de los pasos que sigue un actor, es decir, uno de los usuarios del sistema, para la realización de una tarea concreta en el sistema.

Los casos de uso se pueden representar de forma tabular, y de forma gráfica, mediante diagramas.

3. 2. 1. Descripción tabular de los casos de uso

El formato tabular será el siguiente:

Tabla 30. Plantilla para la especificación de casos de uso

Identificador: CU-XX	
Nombre	
Actores	
Objetivo	
Precondiciones	
Postcondiciones	
Verificabilidad	

Cada caso de uso debe ser definido con una serie de atributos:

- **Identificador**: Es el número identificativo único de cada caso de uso y que seguirá el formato CU-XX, donde CU indica que es un caso de uso, y XX representa el número del caso de uso dentro de la clasificación.
- **Nombre.** Es el nombre del caso de uso.
- **Actores.** Son los actores que interactúan con el sistema en el caso de uso.
- **Objetivo.** Es la acción que se quiere realizar por parte del actor o actores en el sistema.
- **Precondiciones.** Son las condiciones que se tienen que cumplir para que se pueda iniciar el caso de uso.
- **Postcondiciones.** Reflejan el estado en que se queda el sistema una vez ejecutado el caso de uso (caso base).
- **Condiciones de fallo.** Son los caminos posibles en un caso de uso que producen excepciones en el sistema.

Los casos de uso de los diferentes sistemas son los mostrados a continuación.

Tabla 31. Caso de uso CU-01

Identificador: CU-01	
Nombre	Conversión de *.demo* a *.csv*
Actores	Usuario
Objetivo	Obtención de un archivo de texto con extensión *.csv* desde un archivo de repetición visual de la partida con extensión *.demo* mediante el uso de la interfaz visual creada para el sistema *demoinfo*.
Precondiciones	1) Selección del archivo mediante la ventana de navegación que despliega el botón "*Search file*" de la interfaz visual de *demoinfo*. 2) Introducción del nombre del archivo *csv* de salida en la entrada de texto "*.csv File name*" de la interfaz visual de *demoinfo*. 3) Pulsar botón "*Convert*".
Postcondiciones	1) Fichero *csv* correctamente obtenido, con el mismo nombre de salida que el introducido en la entrada de la interfaz visual.
Condiciones de fallo	Alguno de los parámetros no se introduce, por lo que el archivo no es convertido.

Tabla 32. Caso de uso CU-02

Identificador: CU-02	
Nombre	Conversión de *.csv* a *.arff*
Actores	Usuario
Objetivo	Obtención de un archivo con extensión *.arff* comprensible y procesable por Weka, desde un archivo *.csv* obtenido en realizando el CU-01 mediante el uso de la interfaz visual creada para el sistema *demoParser*.
Precondiciones	1) Selección del archivo mediante la ventana de navegación que despliega el botón "*Search file*" de la interfaz visual de *demoParser*. 2) Selección del tipo de jugador del que proceden los datos *.csv* (clase de los patrones, *skilled* o *cheater*) mediante la ventana desplegable "*Player type*" de la interfaz visual de *demoParser*. 3) Selección del tipo de patrón a utilizar en el proceso de conversión mediante la ventana desplegable "*Parse type*" de la interfaz visual de *demoParser*. 4) Elección del número de muertes a utilizar en cada patrón en el proceso de conversión mediante la entrada de texto "*Kills appended*" de la interfaz visual de *demoParser*. 5) Elección del número de *ticks* anteriores a cada muerte de los cuales recopilar la información necesaria para el tipo de patrón elegido en el proceso de conversión, mediante la entrada de texto "*Tick slice*" de la interfaz visual de *demoParser*. 6) Introducción del nombre del archivo *arff* de salida en la entrada de texto o pulsando el botón "*auto*" (que proporcionará un nombre en base a los parámetros introducidos) de la interfaz visual de *demoinfo*. 7) Pulsar botón "*Convert*".
Postcondiciones	1) Fichero *arff* correctamente obtenido, con el mismo nombre de salida que el introducido en la entrada de la interfaz visual, en los que los patrones siguen el formato indicado en los parámetros introducidos en la misma..
Condiciones de fallo	Alguno de los parámetros no se introduce, por lo que el archivo no es convertido.

Tabla 33. Caso de uso CU-03

Identificador: CU-03	
Nombre	Impresión de cabeceras a un archivo *.arff*
Actores	Usuario
Objetivo	Impresión de las cabeceras pertenecientes a un archivo de entrada a un archivo de salida, mediante el uso de la interfaz visual creada para el sistema *arffManager*.
Precondiciones	1) Selección del archivo de entrada mediante la ventana de navegación que despliega el botón "*Search file*" de la interfaz visual de *arffManager*. 2) Selección del tipo de operación "*Print headers to file*" mediante la ventana desplegable "*Parse type*" de la interfaz visual de *arffManager*. 3) Introducción del nombre del archivo *arff* de salida en la entrada de texto "*.arff File name output*" o selección de archivo de salida mediante la ventana de navegación que despliega el botón segundo botón "*Search file*" de la interfaz visual de *arffManager*. 4) Pulsar botón "*Convert*".
Postcondiciones	1) Obtención de un archivo *arff* en el que el único texto que hay en el archivo son las cabeceras del archivo de entrada elegido en la interfaz visual de *arffManager*.
Condiciones de fallo	Alguno de los parámetros no se introduce, por lo que el archivo no es obtenido.

Tabla 34. Caso de uso CU-04

Identificador: CU-04	
Nombre	Unión de dos archivo *.arff*
Actores	Usuario
Objetivo	Unión de patrones de dos archivos compatibles (con mismas cabeceras de Weka) en uno de ellos, mediante el uso de la interfaz visual creada para el sistema *arffManager*.
Precondiciones	1) Selección del archivo de entrada mediante la ventana de navegación que despliega el botón "*Search file*" de la interfaz visual de *arffManager*. 2) Selección del tipo de operación "*Add arff data to file*" mediante la ventana desplegable "*Parse type*" de la interfaz visual de *arffManager*. 3) Selección del archivo *arff* de salida en el que se juntarán los patrones de ambos *arff* mediante la ventana de navegación que despliega el botón "*Search file*" de "*.arff File name output*" de la interfaz visual de *arffManager*. 4) Pulsar botón "*Convert*".
Postcondiciones	1) Obtención de un archivo *arff* en el que se respeta el texto del archivo *.arff* de salida, y se añaden los patrones contenidos en el *.arff* de entrada, ambos elegidos en la interfaz visual de *arffManager*.
Condiciones de fallo	Alguno de los parámetros no se introduce, por lo que el archivo de salida no sufre cambio alguno.

Tabla 35. Caso de uso CU-05

Identificador: CU-05	
Nombre	Generación de modelos y test en Weka
Actores	Usuario
Objetivo	Obtención de un modelo de clasificación mediante aprendizaje supervisado, y el porcentaje de clasificación realizada correctamente mediante un método de test, utilizando el programa Weka.
Precondiciones	1) Introducción de los datos de entrenamiento a utilizar en la generación del modelo, mediante el botón "*Open file…*" de la herramienta "*Experimenter*" de la interfaz visual de Weka. 2) Selección de un clasificador mediante la ventana desplegable obtenida al pulsar el botón "*Choose*" de la pestaña "*Classify*". 3) Edición de los parámetros del clasificador. 4) Elección del modo de realización del test (obtención de porcentajes) y relleno de los parámetros del mismo (en caso de archivo externo, selección del mismo). 5) Pulsar el botón "*Start*".
Postcondiciones	1) Obtención del modelo de clasificación. 2) Obtención de los porcentajes de clasificación. 3) Obtención de otros datos, entre ellos la matriz de confusión.
Condiciones de fallo	Alguno de los parámetros no se introduce o es incorrecto, por lo que se obtendrá un error y no se obtendrá nada de lo descrito en las postcondiciones.

Tabla 36. Caso de uso CU-06

Identificador: CU-06	
Nombre	Generación de modelos y test en RapidMiner
Actores	Usuario
Objetivo	Obtención de un modelo de clasificación mediante aprendizaje supervisado, y el porcentaje de clasificación realizada correctamente mediante un método de test, utilizando el programa RapidMiner.
Precondiciones	1) Selección de los datos a utilizar en el entrenamiento, mediante el botón "*Import Data*", especificando fichero binario, mediante la interfaz visual de RapidMiner. 2) Generación del diagrama de flujo en el que se convierte el fichero binario a lectura *arff* y se elige el clasificador y el método de test con los operadores pertinentes. 3) Edición de los parámetros del clasificador. 4) Pulsar el botón "*Play*".
Postcondiciones	1) Obtención del modelo de clasificación. 2) Obtención de los porcentajes de clasificación. 3) Obtención de otros datos, entre ellos la matriz de confusión.
Condiciones de fallo	Alguno de los parámetros no se introduce o es incorrecto, o el diagrama de flujo no está correctamente conectado, por lo que se obtendrá un error y no se obtendrá nada de lo descrito en las postcondiciones.

3. 2. 2. Descripción gráfica de los casos de uso

A continuación, en la Figura 11, se muestra la descripción gráfica de los casos de uso, en la que se puede observar en un bajo nivel de detalle, las relaciones entre los casos de uso y el único actor, el usuario.

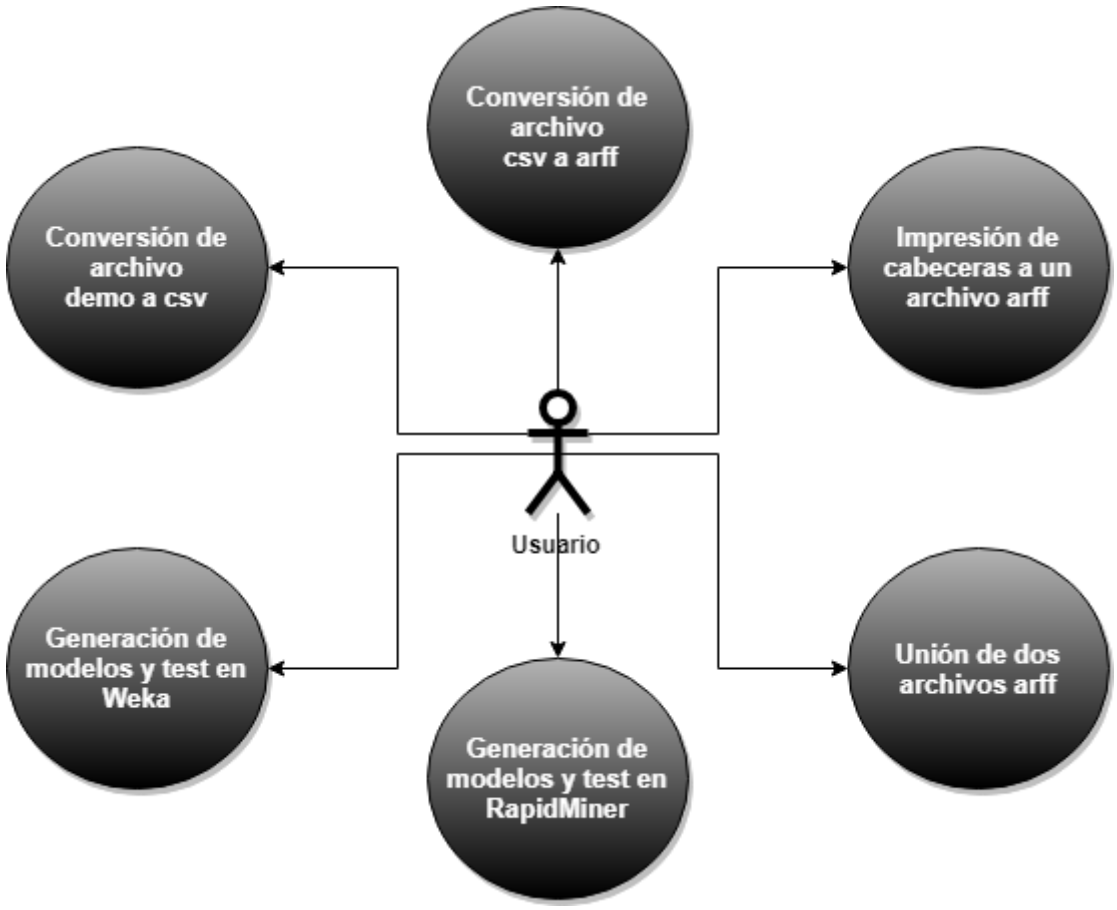


Figura 11. Diagrama de casos de uso del sistema

3. 3. Arquitectura del sistema

En la Figura 12, se muestra el diagrama de componentes que representa la arquitectura del sistema del presente trabajo de fin de grado. Esta arquitectura, encaja con las especificaciones reunidas en la definición de requisitos del sistema.

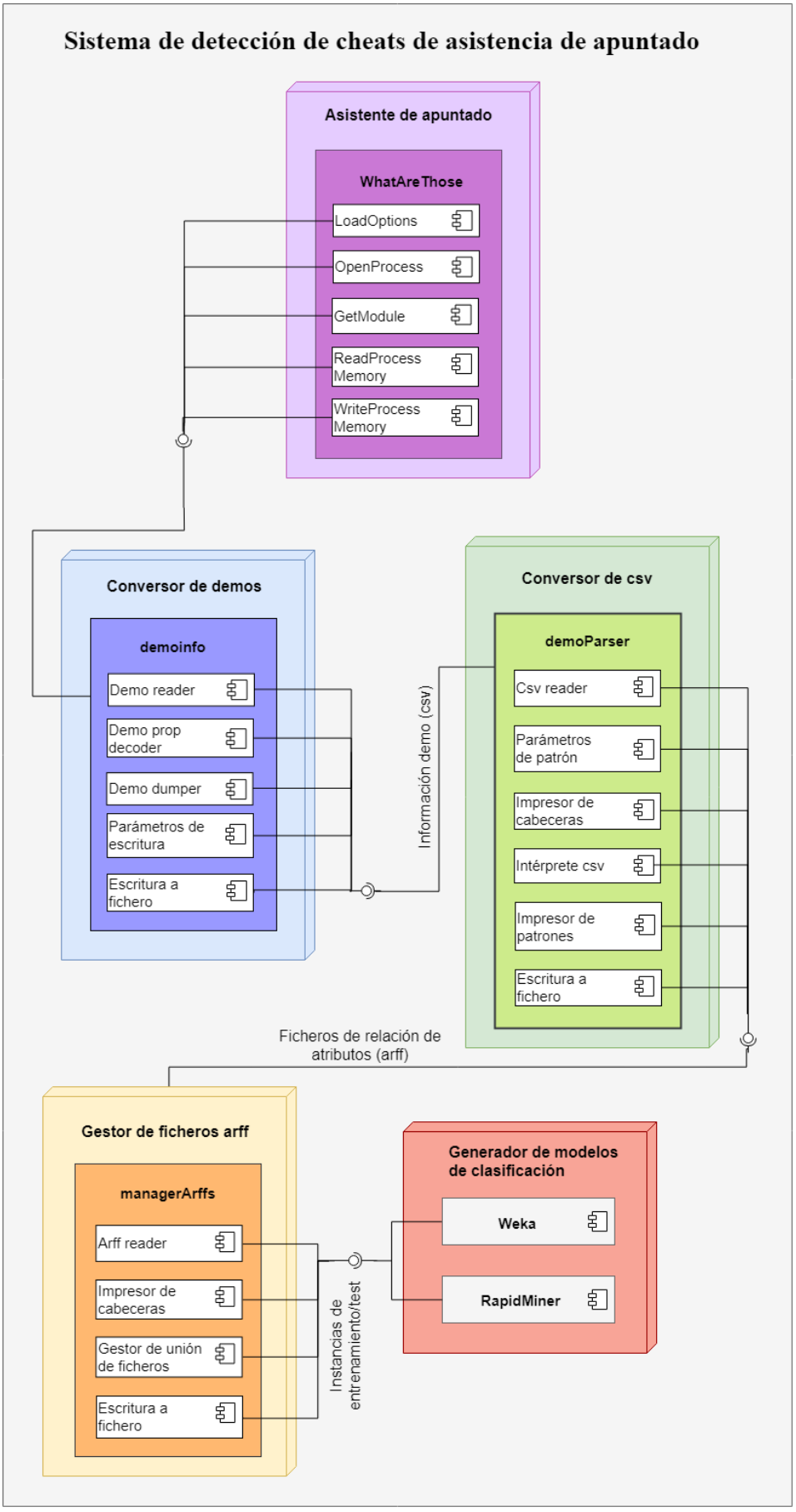


Figura 12. Arquitectura del sistema

Como se puede observar, el sistema principal se divide en 5 subsistemas. Dichos subsistemas son descritos a continuación.

- **Subsistema Asistente de puntería.** Este subsistema se corresponde con el software que se encarga de, mediante operaciones de lectura y escritura de memoria en el proceso *csgo.exe*, simular los movimientos de ratón que ayudarán al usuario a apuntar a los enemigos y matarlos en función de unos parámetros que permiten configurar la velocidad de movimiento y otros parámetros del *cheat* (puesto que este subsistema, es el *cheat* a detectar). Se decide que el software que maneja este subsistema se llame *WhatAreThose*. Este subsistema, cuenta con 5 principales componentes:
    - ***LoadOptions.*** Este componente se encarga de leer y cargar las opciones del *cheat* (subsistema) desde el archivo de texto *options.txt*.
    - ***OpenProcess.*** Este componente se encarga de encontrar un proceso con el nombre *csgo.exe*, almacenarlo en una estructura capaz de almacenar procesos, y de "abrir" dicho proceso mediante su PID (número identificador único perteneciente al proceso), almacenándolo en una estructura de tipo *HANDLE*. Dicha estructura será necesaria para la lectura y escritura en la memoria del proceso.
    - ***GetModule.*** Este componente se encarga de encontrar la posición de memoria base almacenada de forma dinámica en el computador (es decir, dicha posición de memoria cambia en cada ejecución del subsistema) del módulo a partir del cual se hallan todas las estructuras que será necesario leer. Dicho módulo, será *client_panorama.dll*. Es decir, se encargará de obtener la primera dirección de memoria donde se aloje dicho módulo. No obstante, este proceso se explicará en detalle en los apartados posteriores del presente trabajo.
    - ***ReadProcessMemory.*** Este componente se encargará de realizar las operaciones de lectura de las direcciones de memoria cuyos valores se requiere modificar dentro de la memoria del proceso, y de almacenar los valores recogidos en las estructuras y variables necesarias.
    - ***WriteProcessMemory.*** Este componente se encargará de realizar las operaciones de escritura de los nuevos valores, previamente calculados, en las direcciones de memoria cuyos valores se requiere modificar.
- **Subsistema Conversor de demos.** Este subsistema se corresponde con el software encargado de convertir los archivos visuales de repetición de partida (archivos con extensión *demo*) en archivos de texto con extensión *csv*. Dichos archivos de texto, contendrán toda la información necesitada por el siguiente subsistema. Además, este subsistema tiene interacción con el subsistema anterior, dado que recibe las demos en las que ha actuado el Asistente de puntería. El software que maneja este subsistema se llama *demoinfogo*, y se trata de un programa de código abierto proporcionado por la propia empresa desarrolladora del juego, Valve (aunque con un par de modificaciones realizadas que se explicarán más adelante). Este subsistema, cuenta con 5 principales componentes:

- ***Demo reader.*** Este componente se encargará de abrir y leer el archivo con extensión *demo*.
- ***DemoPropDecoder.*** Este componente se encargará de leer y decodificar la información a obtener, y de almacenarla en estructuras para su posterior escritura. Esto es posible, debido a que las demos contienen mensajes que utilizan *Google's Protocol Buffers* (*protobuf*), que es un lenguaje de serialización de mensajes y objetos que generan código.
- ***Parámetros de escritura.*** Este componente se encargará de seleccionar la tipología de los mensajes a escribir en fichero, es decir, seleccionará qué mensajes se escriben.
- ***DemoFileDumper y Escritura a fichero.*** Estos componentes se encargarán de leer las estructuras almacenadas por el componente *DemoPropDecoder*, e imprimir la información de las mismas a un fichero en base a los parámetros de escritura recogidos por el componente anterior.

- **Subsistema Conversor de csv.** Este subsistema se corresponde con el software encargado de convertir los archivos de texto con extensión *csv* a archivos de texto de relación de atributos (archivos con extensión *arff*). Dichos archivos de texto, contendrán toda la información necesitada por el siguiente subsistema, es decir, las cabeceras y los patrones recogidos, previamente obtenidos mediante la manipulación de texto en el presente subsistema. Este subsistema tiene interacción con el subsistema anterior, dado que recibe los archivos *csv* que han sido obtenidos dicho subsistema. Se decide que el software que maneja este subsistema se llame *demoParser*. Este subsistema, cuenta con 6 principales componentes:
  - ***CSV Reader.*** Este componente se encargará de abrir y leer el archivo con extensión *csv*.
  - ***Parámetros de patrón.*** Este componente se encargará de determinar el formato con el que serán escritos los patrones. Además, dichos parámetros influirán en la impresión de las cabeceras del archivo *arff*.
  - ***Impresor de cabeceras.*** Este componente se encargará de imprimir las cabeceras necesarias para la comprensión del archivo *arff* por Weka.
  - ***Intérprete csv.*** Este componente se encargará de transformar y seleccionar la información contenida en el archivo de texto obtenido en el subsistema anterior, en base a los parámetros que debe contener cada patrón.
  - ***Impresor de patrones.*** Este componente se encargará de imprimir los patrones con el formato indicado en los parámetros de patrón.
  - ***Escritura a fichero.*** Este componente se encargará de realizar las escrituras a fichero.

- **Subsistema Gestor de ficheros arff.** Este subsistema se corresponde con el software encargado de manejar la unión e impresión de los ficheros *arff*. Este subsistema tiene interacción con el subsistema anterior dado que recibe los archivos *arff* con los que opera de dicho subsistema. Se decide que el software que maneja este subsistema se llame *managerArffs*. Este subsistema, cuenta con 6 principales componentes:
    - ***Arff reader.*** Este componente se encargará de abrir y leer el archivo con extensión *arff*.
    - ***Impresor de cabeceras.*** Este componente se encargará de imprimir las cabeceras del archivo *arff* de entrada a un archivo de salida, ambos especificados por parámetro, con sobrescritura.
    - ***Gestor de unión de ficheros.*** Este componente se encargará de añadir los datos (patrones) del archivo *arff* de entrada a los del archivo *arff* de salida, ambos especificados por parámetro. No realiza sobrescritura.
    - ***Escritura a fichero.*** Este componente se encargará de escribir los cambios en el fichero de salida.
- **Subsistema Generador de modelos de clasificación.** Este subsistema se corresponde con el software encargado de la generación de los distintos modelos de clasificación, a partir de ficheros *arff*. Este subsistema tiene interacción con el subsistema anterior dado que recibe como entrada los archivos *arff* proporcionados por el subsistema anterior. El subsistema tiene dos componentes, los dos programas que generarán los modelos de clasificación, Weka y RapidMiner.

4. IMPLEMENTACIÓN DEL SISTEMA

En este capítulo se describe de forma detallada el proceso seguido en la implementación de cada uno de los subsistemas que componen el sistema de detección de *cheats*.

4. 1. Asistente de puntería

El asistente de puntería será el *cheat* a detectar por el sistema, y en el caso del presente trabajo será un software desarrollado en el lenguaje de programación C++, debido a que se tratará de un programa con bucles infinitos, que realizará operaciones de lectura y de escritura, y que deberá funcionar lo más fluido posible. Se utilizará el IDE Visual Studio Professional 2017 para su desarrollo.

Antes de comenzar a desarrollar un *cheat*, primero hay que comprender su funcionamiento. Existen tres tipos de *cheat* en relación a la forma en la que funcionan: internos, externos e internos/externos.

En el caso del presente trabajo se plantea desarrollar uno de **tipo externo**, aunque cualquiera de los tres tipos funcionaría de la misma manera, y, por tanto, se explicará únicamente el funcionamiento de uno de este tipo.

Los *cheats* de tipo externo, como indica su nombre, son ejecutables externos al programa, que simplemente acceden mediante un ***handle*** (manejador) al proceso de en el cual se requiere modificar la memoria, sin llegar a "adherirse" a él.

Primero, habrá que localizar cada una de las **direcciones de memoria** de las cuales se requiere saber su valor para realizar las operaciones oportunas. Para ello, se sabe por cómo está implementado el juego, que todas las direcciones de memoria siguen el siguiente formato:

Dirección de memoria inicial del módulo "client_panorama.dll" + offset de la estructura o dato a la cual se quiere acceder.

Se sabe también que la dirección del módulo varía cada vez que se abre el ejecutable del juego, dado que la imagen del proceso (*csgo.exe*) es almacenada de forma dinámica en el sistema, así como lo son los diferentes módulos pertenecientes al proceso. Es decir, seguirían una estructura como la presentada en la Figura 13. En dicha figura, se puede ver un ejemplo de acceso a la dirección de memoria que contiene la vida del jugador.

Por lo tanto, habría que conocer la dirección inicial del proceso *csgo.exe*, el ***offset*** (número de direcciones añadidas a una dirección base necesarias para acceder a otra dirección, es decir, la diferencia entre la dirección final y la dirección base) necesario para acceder al módulo (*client_panorama.dll*) que contiene todas las estructuras a las que se accederá tanto para lectura como para escritura, y cada uno de los *offset* de las estructuras a las que vamos a acceder (que se sumarán a la dirección base de *client_panorama.dll*).

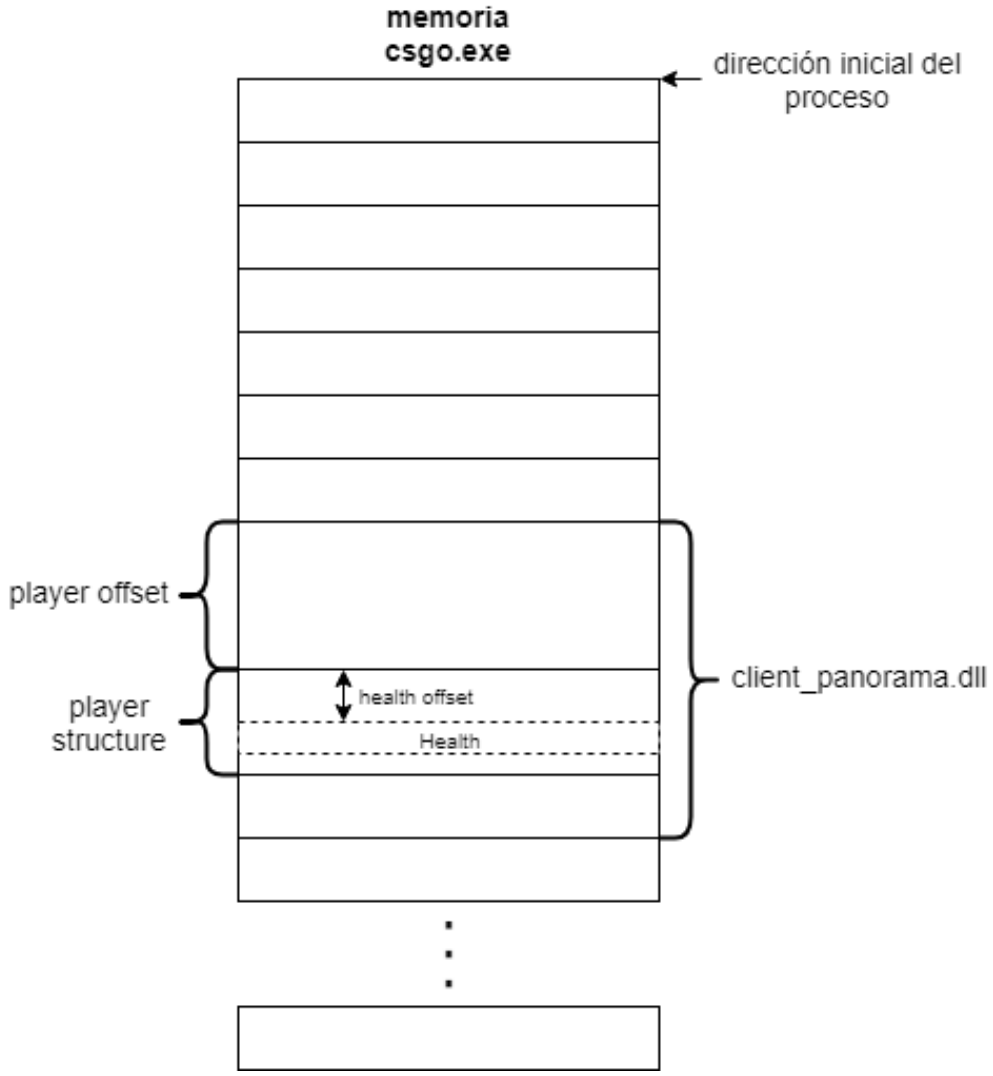


Figura 13. Representación de la memoria de csgo.exe

En el ejemplo de esta figura se busca acceder a la dirección de memoria que contiene la vida del jugador. Primero, se tendrían que obtener la dirección inicial del proceso, el *offset* de *client_panorama.dll* (que cambia **cada vez que se aloja el proceso** en memoria) y los *offset* de la estructura del jugador y de la vida del jugador (dichos *offset*, varían normalmente con **cada actualización** del videojuego, pero si no hay actualización alguna, no varían en la ejecución del videojuego).

Una vez conocidos todos los valores, y, teniendo en cuenta que el jugador es una estructura que contiene diversos valores como la vida o la armadura del mismo, habría que leer la siguiente dirección de memoria, de la cual se extraería la vida restante del jugador en cuestión:

dirección inicial csgo.exe + client_panorama.dll offset + player offset + health

Se necesita, entonces, antes de programar el asistente en sí mismo, una forma de obtener todos esos valores.

Para obtener la **dirección del proceso**, crearemos un *handle* al mismo, buscándolo mediante un método que recorre todos los procesos de 32 bits del sistema (puesto que *csgo.exe* es un proceso de 32 bits).

Para obtener la **dirección de los módulos necesarios**, que, en este caso, además de necesitar la del módulo *client_panorama.dll*, también se necesitará la de *engine.dll*, se sigue la misma técnica, pero recorriendo los módulos.

Una vez obtenidos estos valores, se necesitan los *offset* de todas y cada una de las estructuras almacenadas dentro de los módulos. Estos valores, además, pueden variar con las actualizaciones del juego. Para ello, habría que realizar una tarea de **ingeniería inversa** (decompilar) dentro de cada uno de dichos módulos. No obstante, existen otros métodos como la obtención de firmas de cada uno de los *offset*, que facilitan esta tarea. Para el caso del presente trabajo, se decide utilizar una herramienta de uso libre y código abierto que emplea este método, ***hazedumper*** [19]. Esta herramienta, proporcionará todos los *offset* de estos módulos, como se puede ver en la Figura 14, en la que se puede observar una muestra de dichos *offset*.

```
constexpr ::std::ptrdiff_t m_ArmorValue = 0xB324;
constexpr ::std::ptrdiff_t m_Collision = 0x31C;
constexpr ::std::ptrdiff_t m_CollisionGroup = 0x474;
constexpr ::std::ptrdiff_t m_Local = 0x2FBC;
constexpr ::std::ptrdiff_t m_MoveType = 0x25C;
constexpr ::std::ptrdiff_t m_OriginalOwnerXuidHigh = 0x31B4;
constexpr ::std::ptrdiff_t m_OriginalOwnerXuidLow = 0x31B0;
constexpr ::std::ptrdiff_t m_aimPunchAngle = 0x302C;
constexpr ::std::ptrdiff_t m_aimPunchAngleVel = 0x3038;
```

Figura 14. Muestra de offset de memoria de CS:GO

Implementados los métodos por los cuales se obtendrán las direcciones necesarias, se necesita una función con la cual leer los valores de dichas direcciones, e igual para escribir dentro de ellas los nuevos valores modificados. Para ello se utilizarán las siguientes funciones de Windows:

- **ReadProcessMemory**. Esta función recibirá por parámetro el *handle* del proceso del cuál leer, la dirección de memoria a leer, la dirección de memoria de la variable donde almacenar el valor leído, el tamaño en bytes de la variable a leer, y el puntero a una variable que recibirá los bytes leídos (si este valor es NULL, el parámetro se ignorará) [20].
- **WriteProcessMemory**. Esta función recibirá por parámetro el *handle* del proceso al cuál escribir, la dirección de memoria donde escribir, la variable a escribir, el tamaño en bytes de la variable a escribir, y el puntero a una variable que recibirá los bytes escritos (si este valor es NULL, el parámetro se ignorará) [21].

En este punto, ya se dispone de todas las herramientas necesarias para la implementación del *cheat*. A continuación, se explicará el funcionamiento del asistente de puntería, sin entrar al máximo detalle, y obviando un par de operaciones.

Primero, se comprobará si el *cheat* está activado o desactivado, puesto que se podrá activar o desactivar pulsando una tecla. Si está activado, se entrará en el siguiente bucle infinito (puesto que el *cheat* debe estar funcionando en todo momento):

1. Se recorrerá un bucle de 32 iteraciones. Cada iteración del bucle accederá a lo que puede ser un jugador o entidad (dado que el máximo de jugadores en una partida puede ser de 32 en algunos modos de juego). Para ello se almacenará en cada iteración del bucle la dirección base correspondiente a la entidad de la iteración, teniendo en cuenta el tamaño de la estructura de una entidad en direcciones de memoria, y saltando dentro de la lista de jugadores el número de posiciones correspondientes.
2. Se almacenará el equipo al que pertenece la entidad, su vida, y la posición del mismo como vector de tres dimensiones (concretamente, la posición de la parte del cuerpo a la que se vaya a disparar, que es leída del archivo de opciones).
3. Se almacenará la dirección base del jugador controlado por el usuario, el equipo al que pertenece y la posición del mismo como vector de tres dimensiones (al que hay que sumar la altura de la posición de los ojos del jugador, dado que, debido al diseño del juego, la trayectoria de las balas es calculada desde los ojos del mismo).
4. Si la entidad de la iteración del bucle de entidades está muerta en ese mismo instante, o si pertenece al mismo equipo del jugador, se avanzará de iteración en el bucle de entidades, descartando así dicha entidad.
5. Mediante un cálculo matemático, teniendo en cuenta la posición del jugador y la posición del enemigo, se calculará a que ángulo (en vector de tres dimensiones) deberá apuntar el jugador para situar el *crosshair* en la parte del cuerpo elegida de la entidad de la correspondiente iteración del bucle.
6. A continuación, se almacenará el ángulo de visión actual del jugador, y se calculará la diferencia entre el ángulo calculado en el paso 5, y este último ángulo, almacenando el resultado en una estructura que contiene también la entidad de la iteración, teniendo en cuenta que la disposición de ángulos en CS:GO es de -180º a 180º
7. Se comprobará si la tecla que activa el asistente de puntería está presionada, y si la entidad más cercana al jugador (en movimiento de ratón, es decir, en diferencia de ángulos) se encuentra entre los límites de *FOV* definidos por el usuario en las opciones del *cheat*. Si es así, se dividirá el ángulo calculado en el paso 6 en X partes (siendo X otro parámetro del *cheat*, que tendrá influencia directa en la velocidad de movimiento del ratón), y se irá añadiendo una de dichas partes en otro bucle, que tendrá tantas iteraciones como partes en las que se divida el ángulo. En cada iteración del bucle, se escribirá en memoria el nuevo ángulo calculado, que será la suma del ángulo actual del jugador y el valor de una de las partes en las que se divide el ángulo, para así avanzar paulatinamente hacia la posición en la que se encuentra la parte del cuerpo elegida del enemigo al que disparar.
8. Además, se calculará el retroceso del arma, y se tendrá en cuenta el mismo en el cálculo de ángulos, dado que, a partir del primer disparo del arma, esta adquiere un retroceso.

A continuación, en la Figura 15, se muestra un ejemplo de funcionamiento del *cheat*.

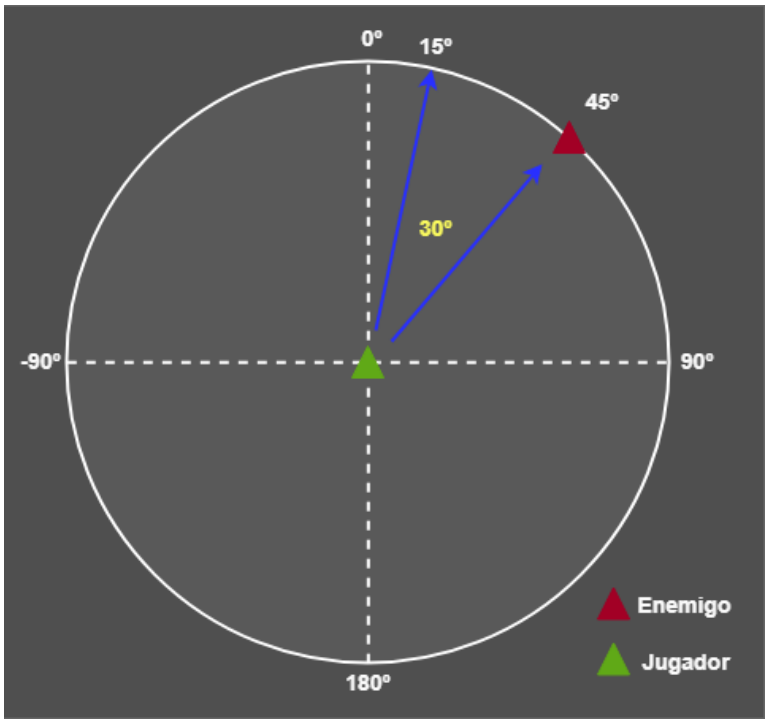


Figura 15. Ejemplo de funcionamiento de cheat

En el ejemplo, el jugador se encuentra apuntando al ángulo 15º. El ángulo al cual tiene que apuntar para posicionar el *crosshair* sobre el enemigo, es el ángulo 45º (que es el ángulo calculado en el paso 5. La diferencia en ángulo entre este último y el primero es de 30º, que es el ángulo calculado en el paso 6 y que se dividirá en X partes y se irán sumando las partes al ángulo del jugador, que inicialmente es de 15º.

Además, en el cálculo de ángulos, también se tendrá en cuenta la distancia del enemigo, puesto que el *FOV* debería adaptarse a esta última [22].

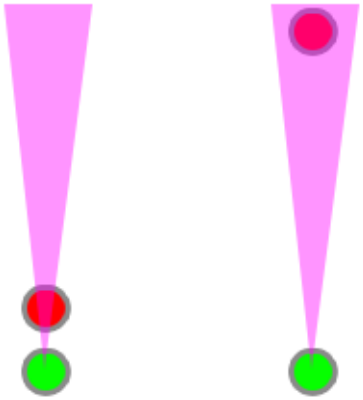


Figura 16. FOV sin tener en cuenta la distancia del enemigo [22]

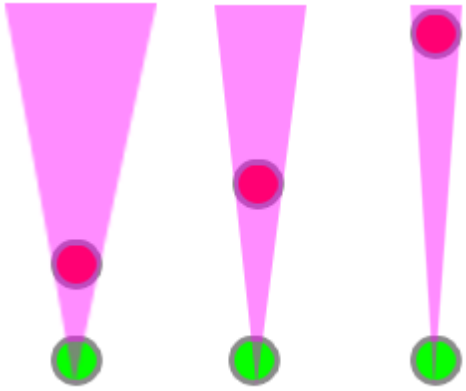


Figura 17. FOV teniendo en cuenta la distancia del enemigo [22]

En la Figura 16, se favorece a los enemigos que estén lejos, mientras que en la Figura 17 el ángulo se adapta en función de la distancia. Por ejemplo, el enemigo cercano al jugador no sería detectado en la Figura 16, mientras que en la Figura 17 sí.

La carga de opciones del fichero *options.txt* se realizará en la apertura del *cheat*, y también cada vez que se oprima la tecla F8, que es a la que se asocia para la actualización de opciones. Además, oprimiendo la tecla F6, se cerrará el proceso del *cheat*.

Por último, cabe destacar que este subsistema puede tener problemas con algunos antivirus, así que habría que añadirlo a las exclusiones del mismo. Esto se debe a que se trata de un proceso que accede a la memoria de otro, y puede ser detectado como una amenaza.

4. 2. Conversor de *demos* a *csv*

El conversor utilizado en el presente trabajo, es el proporcionado por Valve, de código abierto y libre uso. Está programado en el lenguaje C++.

Se comenta una línea de impresión, y se añade una comprobación en las impresiones que reducirá considerablemente el tamaño del fichero de salida producido por el conversor, ya que se observaba que el tamaño de los ficheros era considerablemente alto. Dicha comprobación filtrará las propiedades de la partida, favoreciendo la impresión de únicamente los ángulos de visión del jugador. El resto de código se mantiene intacto.

La interfaz visual es implementada en lenguaje C#, utilizando *Windows Forms*, las únicas funciones implementadas son las asociadas a los botones y entradas, que se encargan del paso de argumentos en las funciones principales del ejecutable generado en C++.

Dicha interfaz visual no incluirá en este caso todos los parámetros del programa, y solo recibirá el archivo *demo* de entrada y el nombre del archivo de salida, ya que se obvia que todas las ejecuciones se realizarán con los mismos parámetros (aquellos que desechen la información que no servirá posteriormente en el presente trabajo).

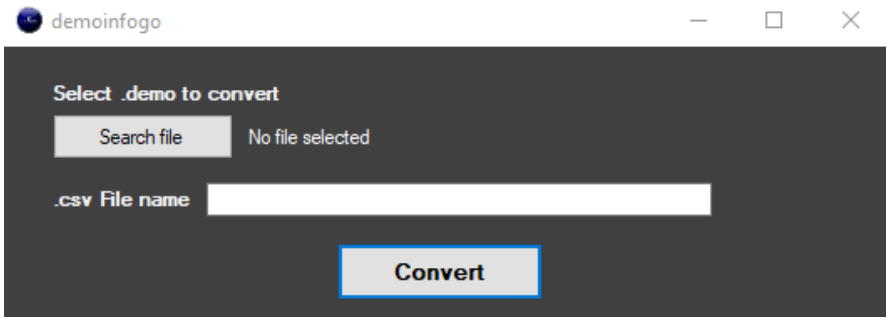


Figura 18. Interfaz visual conversor de demos

4. 3. Conversor de *csv* a *arff*

Este conversor será un software desarrollado en el lenguaje de programación C++, que se encargará de reunir la información relevante para construir los patrones, y de imprimirlos junto con las cabeceras necesarias para formar un archivo *arff*. Se utilizará el IDE Visual Studio Professional 2017 para su desarrollo.

Para implementar este conversor, antes se ha de comprender la estructura del fichero *csv* extraído del conversor anterior. En dicho fichero, hay un número de líneas iniciales (dependiente del tamaño total del fichero) que proporcionan información sobre la partida que es irrelevante al objetivo de este trabajo. Después, se encuentran una serie de mensajes que contienen el *tick* (momento de la partida) en el que se produce dicho mensaje, y si hay alguna actualización en los ángulos hacia los que está mirando el jugador.

Por lo tanto, habrá tantos mensajes como *ticks* hayan transcurrido en la partida, y cada mensaje contendrá el *tick* correspondiente al mismo, y podrá contener, o no, los ángulos de visión del jugador (si no hay habido actualización de los mismos, no los contendrá).

Dichos mensajes, tienen como cabecera la siguiente línea:

"---- CNETMsg_Tick (X bytes) ----------"

Donde X, es un número entre 11 y 13, que será el tamaño del mensaje.

Además, se registrarán los disparos del jugador con la cabecera "*weapon_fire*", y la muerte de las entidades con la cabecera "*player_death*". La primera no será de interés, pero la segunda indicará el momento en el que se debe formar un patrón.

A continuación, se muestra un ejemplo de un *tick* en el que se modifican los ángulos de visión del jugador en la Figura 19, y también se muestra un ejemplo de la muerte de una entidad en la Figura 20.

```
---- CNETMsg_Tick (12 bytes) -----------------
tick: 20981
host_computationtime: 1617
host_computationtime_std_deviation: 1196
host_framestarttime_std_deviation: 94
Entity Delta update: id:1, class:35, serial:762
Field: 20, m_angEyeAngles[0] = 359.956055
Field: 21, m_angEyeAngles[1] = 289.627075
```

Figura 19. Ejemplo de mensaje en fichero csv en un tick

```
player_death
{
 userid: Ethan (id:26)
 attacker: (id:2)
  position: 1.159507, -166.850601, 0.031250
  facing: pitch:0.109863, yaw:308.776245
  team: T
 assister: 0
 weapon: ak47
 weapon_itemid: 0
 weapon_fauxitemid: 17293822569102704647
 weapon_originalowner_xuid: 76561198134794737
 headshot: 1
 dominated: 0
 revenge: 0
 penetrated: 0
 noreplay: 1
}
```

Figura 20. Ejemplo de muerte de un jugador en fichero csv

Como se puede observar en la Figura 19, los ángulos son mostrados como los parámetros "*m_angEyeAngles[X]*", donde X indica la coordenada del ángulo. Se puede observar que en este fichero los ángulos se muestran de 0 a 360º, a pesar de que en el juego los ángulos se utilicen en el rango -180 a 180º, aunque esto no supone ningún impedimento.

Una vez conocido el formato del fichero *csv* con el que tratará este conversor, la implementación será más sencilla, como se explica a continuación:

1. Se crearán estructuras que almacenen las últimas X modificaciones de ángulos, o ángulos en sí mismos (dependiendo de la opción), junto con sus respectivos números de *ticks*, donde X es un número especificado por parámetro. Los últimos ángulos a añadir a las estructuras, reemplazarán a los primeros.
2. Se abrirá el fichero *csv* y se imprimirán las cabeceras en relación a los argumentos introducidos (tipo de jugador, tipo de patrón, etc) en el fichero *arff* de salida indicado.
3. Se avanzará línea por línea en el fichero *csv* hasta el primer mensaje del fichero. Para ello, se comparará la línea con las posibles cabeceras de los mensajes, con el formato explicado anteriormente.
4. Se almacenará el *tick* actual de la partida (independientemente de si se modifican los ángulos) en una variable global. Si ocurre una modificación de ángulos, se almacenarán dichos ángulos y *ticks* en las estructuras explicadas anteriormente.
5. Si se encuentra la cadena que indica la muerte de una entidad, *player_death*, se imprimirá el patrón en una línea, con la clase correspondiente (*cheater* o *skilled*) si se cumple que el número de muertes en el patrón es el mismo número que el especificado por parámetro (normalmente, cada patrón tendrá una muerte, pero podrá tener más). Si el patrón no tiene el suficiente número de muertes, esperará a recibir más, hasta poder imprimir la clase al final del patrón. Además, los patrones tendrán la información especificada por parámetro, que podrá ser:

a. Ángulos anteriores a una muerte.

b. Ángulos anteriores a una muerte y velocidad de movimiento de ratón.

c. Velocidades y desviación total de movimiento de ratón antes de la muerte.

d. Velocidades y desviaciones parciales de movimiento antes de la muerte.

6. Por último, si el último patrón en el fichero no tiene clase, se eliminará dicho patrón. Esto se debería a que el número de muertes por patrón no fuera divisor del número de muertes totales en el fichero *csv*.

La interfaz visual es implementada en lenguaje C#, utilizando *Windows Forms*, las únicas funciones implementadas son las asociadas a los botones y entradas, que se encargan del paso de argumentos en las funciones principales del ejecutable generado en C++.

Figura 21. Interfaz visual de conversor de csv

4. 4. Manejador de *arff*

Este conversor será un software desarrollado en el lenguaje de programación C++, que se encargará de facilitar la unión de los diferentes ficheros *arff* producidos por el conversor anterior. Se utilizará el IDE Visual Studio Professional 2017 para su desarrollo.

Si la opción introducida en el programa es la de imprimir las cabeceras, se abrirán el fichero de entrada y el de salida, y se copiarán las cabeceras del primero en el segundo, sobre-escribiendo el contenido de este último, si es que tenía alguno.

Por otro lado, si la opción introducida en el programa es la de unir ficheros, se abrirán el fichero de entrada y el de salida, y se copiarán los patrones del primero en el segundo, a partir del final del este último fichero.

La interfaz visual es implementada en lenguaje C#, utilizando *Windows Forms*, las únicas funciones implementadas son las asociadas a los botones y entradas, que se encargan del paso de argumentos en las funciones principales del ejecutable generado en C++.

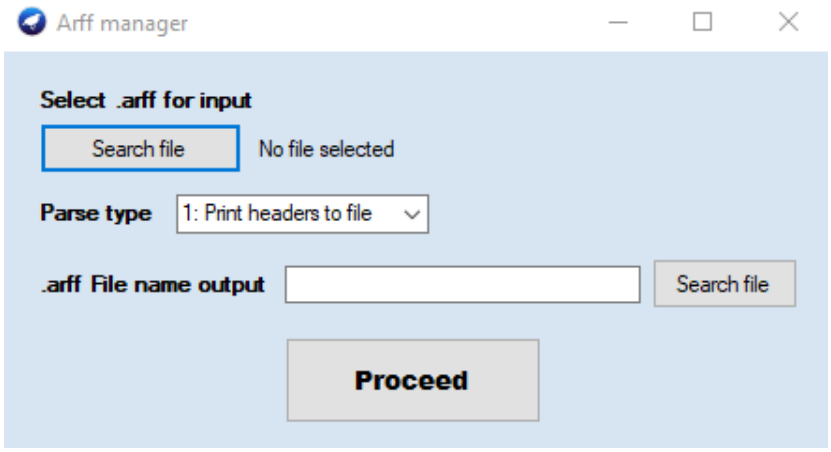


Figura 22. Interfaz visual manejador arff

5. RESULTADOS Y EVALUACIÓN

En este capítulo se describen las pruebas unitarias del sistema, el proceso de recogida de datos, la parametrización de los mismos, los algoritmos propuestos para la solución, las distintas pruebas colectivas e individuales a realizar, que medirán la evaluación del sistema, y, por último, los resultados obtenidos y un análisis de los mismos.

5. 1. Pruebas unitarias

En este apartado se detallarán las pruebas unitarias del sistema, que tienen como objetivo comprobar que los requisitos descritos en el *Apartado 3.1. Requisitos* se cumplen y que el sistema funciona correctamente.

El formato tabular que se utilizará para describir las pruebas, se detalla a continuación, en la Tabla 37.

Tabla 37. Plantilla para las pruebas unitarias

Identificador: P-XX	
Descripción	
Objetivo	
Resultado esperado	
Resultado obtenido	
RF/s cubierto/s	

A continuación, se explica el significado de cada una de las filas de la plantilla para la especificación de requisitos:

- **Identificador**. Es el código identificativo, unívoco, de cada uno de los requisitos. La nomenclatura utilizada será P-XX, donde:
    - P: Indica que es una prueba.
    - XX: Es el número de prueba dentro de la clasificación.
- **Descripción**. Descripción breve y detallada de la prueba.
- **Resultado esperado**. Resultado que se espera obtener con la ejecución de la prueba.
- **Resultado obtenido**. Resultado que se obtiene tras la ejecución de la prueba, indicando el resultado de la misma (si ha coincidido con el esperado o no).
- **RF/s cubiertos**. Requisitos funcionales cuyo correcto funcionamiento se comprueba con la prueba.

Las pruebas unitarias son las descritas a continuación:

Tabla 38. Prueba unitaria P-01

Identificador: P-01	
Descripción	En la interfaz visual del sistema conversor de demos se introduce un archivo *demo*, un nombre del archivo *csv* de salida y se pulsa el botón de convertir.
Objetivo	Comprobar la correcta obtención de un archivo *csv* que reúne los datos necesarios de una *demo*.
Resultado esperado	Se genera un fichero *csv* con el nombre introducido en la interfaz visual del programa que contiene toda la información relevante sobre la partida contenida en el archivo *demo*.
Resultado obtenido	Se comprueba el fichero obtenido, se observa que el resultado es el esperado y se determina, por tanto, como un resultado satisfactorio.
RF/s cubierto/s	RF-01, RF-02, RF-03

Tabla 39. Prueba unitaria P-02

Identificador: P-02	
Descripción	En la interfaz visual del sistema generador de ficheros *arff* se introduce un archivo *csv*, un nombre del archivo *arff* de salida, un tipo de patrón, un tipo de recogida de ángulo, una clase para el patrón, un número de muertes por patrón, el número de *ticks* anteriores a una muerte y se pulsa el botón de convertir.
Objetivo	Comprobar la correcta obtención de un archivo *arff* que reúne los datos necesarios de un fichero *csv* para la comprensión del archivo por Weka.
Resultado esperado	Se genera un fichero *arff* que cumple con todos los parámetros introducidos en la interfaz visual, en el que se ha recopilado toda la información necesaria y especificada del fichero *csv*.
Resultado obtenido	Se comprueba el fichero obtenido, se observa que el resultado es el esperado y se determina, por tanto, como un resultado satisfactorio.
RF/s cubierto/s	RF-04, RF-05, RF-06, RF-07, RF-08, RF-09, RF-10, RF-11, RF-12

Tabla 40. Prueba unitaria P-03

Identificador: P-03	
Descripción	En la interfaz visual del sistema manejador de ficheros *arff* se introduce un archivo *arff* de entrada, un archivo *arff* de salida, un tipo de operación y se pulsa el botón de convertir.
Objetivo	Comprobar la correcta obtención de un archivo *arff* que recoge los datos especificados en el tipo de operación a realizar.
Resultado esperado	Se genera un fichero *arff* con el mismo nombre de salida al especificado, y que cumple con la petición introducida, ya sea de impresión de cabeceras o de combinación de datos.
Resultado obtenido	Se comprueba el fichero obtenido, se observa que el resultado es el esperado y se determina, por tanto, como un resultado satisfactorio.
RF/s cubierto/s	RF-13, RF-14, RF-15, RF-16, RF-17

Tabla 41. Prueba unitaria P-04

Identificador: P-04	
Descripción	Se ejecuta el programa que implementa el asistente de puntería, *WhatAreThose*, mientras el proceso *csgo.exe* está activo, con unas opciones por defecto en el archivo *options.txt* y se pulsa el botón de activación del asistente apuntando (posicionando el *crosshair*) cerca de un enemigo.
Objetivo	Comprobar que el asistente de puntería simula el movimiento de ratón correctamente.
Resultado esperado	El *crosshair* se desplaza hacia la cabeza del jugador enemigo cercano, posicionándose finalmente en la misma.
Resultado obtenido	Se observa que el resultado es el esperado y se determina, por tanto, como un resultado satisfactorio.
RF/s cubierto/s	RF-18, RF-19

Tabla 42. Prueba unitaria P-05

Identificador: P-05	
Descripción	Se introduce un fichero *arff* generado previamente en las pruebas anteriores en Weka, se selecciona un algoritmo cualquiera y se genera un modelo.
Objetivo	Comprobar la correcta comprensión de un archivo *arff* generado previamente, y la correcta generación de un modelo.
Resultado esperado	El fichero es reconocido por Weka y se genera un modelo.
Resultado obtenido	Se observa que el resultado es el esperado y se determina, por tanto, como un resultado satisfactorio.
RF/s cubierto/s	RF-20

A continuación, se muestra la matriz de trazabilidad para el sistema en la Tabla 43. En esta matriz, se muestra como las pruebas cubren todos los requisitos funcionales del sistema, asegurando un correcto funcionamiento de los mismos.

Tabla 43. Matriz de trazabilidad del sistema

RF\P	P-01	P-02	P-03	P-04	P-05
RF-01	X				
RF-02	X				
RF-03	X				
RF-04		X			X
RF-05		X			
RF-06		X			
RF-07		X			
RF-08		X			
RF-09		X			
RF-10		X			
RF-11		X			
RF-12		X			
RF-13			X		
RF-14			X		
RF-15			X		
RF-16			X		
RF-17			X		
RF-18				X	
RF-19				X	
RF-20					X

5. 2. Recogida de datos

Una vez terminada la implementación, se dispone de todas las herramientas necesarias para la recolección y manipulación de datos.

Los datos, como se ha explicado anteriormente, serán recogidos grabando archivos *demo* de la partida (escribiendo en la consola del juego "*record nombre_demo*") matando, en este caso, *bots* (jugadores controlados por IA).

Dichas *demos*, serán convertidas a archivos *csv* que contengan, entre muchos datos, la información necesaria para crear los patrones que entrenarán los modelos que se desarrollen (convirtiendo los archivos *csv* a archivos comprensibles por Weka, es decir, archivos *arff*).

Para ello, las instancias (muertes) se recogerán en un mapa de práctica de puntería [23]. Además, para tener unos datos más limpios, es decir, en los que únicamente se trate de obtener unos patrones de movimiento de ángulos (producidos por el movimiento del ratón, o por el *cheat*), el jugador permanecerá en la misma posición sin moverse. Así, obtendremos una situación que favorezca la distinción entre clases. Esta será la posición tridimensional (*1.159507, -166.850601, 0.031250*), a la cual se puede llegar mediante el comando *setpos* en la consola del videojuego. En la Figura 23 se muestra la posición inicial del jugador y la situación de la partida, antes de empezar a matar *bots* y la grabación de la *demo*.

Figura 23. Posicionamiento antes de la grabación de la demo

Además, se puede observar en la esquina superior izquierda de la captura de pantalla de la figura, la posición del jugador mostrada por un comando introducido en la consola, que verifica que el jugador está correctamente situado.

Todas las muertes se realizarán con el arma sostenida por el jugador en la Figura 23, el AK-47, que es un arma con una potencia de disparo lo suficientemente alta para matar a los enemigos de un disparo en la cabeza. No obstante, dado que la recolección de datos se realiza por ángulos, el arma es indiferente en el resultado de la clasificación, así que se podría utilizar cualquier otra. Además, se tratará de, en la medida de lo posible, realizar las muertes utilizando únicamente un disparo del arma, tratando de, así, asemejar el movimiento al del *cheat* un poco más, añadiendo así una "dificultad extra" a la clasificación de instancias.

La recolección de muertes utilizando el asistente de puntería, es decir, el *cheat*, será realizado por el mismo jugador (el autor del presente trabajo de fin de grado), dado que es indiferente el usuario del mismo, siendo el comportamiento igual en todos los casos. No obstante, en la mitad de las muertes se utilizará el asistente con una configuración muy rápida e inhumana (*rage*), y en la otra mitad, se utilizará una configuración que haga parecer el movimiento más humano (*legit*). Con esto, se buscará que el clasificador sea capaz de detectar ambos tipos de *cheater*.

A continuación, se muestran las opciones del *cheat* utilizadas en cada caso:

Cheater_legit	Cheater_rage
Aimbot_key = 2 *(click_derecho)*	Aimbot_key = 2 *(click_derecho)*
Aimbot_bone = 8 *(cabeza)*	Aimbot_bone = 8 *(cabeza)*
Aimbot_smooth = 400	Aimbot_smooth = 200
Aimbot_factorDist = 3.5	Aimbot_factorDist = 3.5
Aimbot_xFOV = 7	Aimbot_xFOV = 70
Aimbot_yFOV = 7	Aimbot_yFOV = 20

Como se puede observar, la suavidad (*smooth*), que es el inverso a la rapidez, es la mitad en el caso obvio, *rage*, dado que no importa que no parezca humano, así como los grados a los que detectará enemigos son mucho mayores.

Para la recolección de muertes con clase *skilled* (sin el uso del asistente de puntería), con tal de obtener una mayor variabilidad de datos, participarán varias personas con distintos niveles de habilidad en el videojuego, y con distintas sensibilidades de movimiento de ratón y distintas resoluciones de pantalla.

Tabla 44. Datos recolectados y jugadores participantes

Jugador	Nivel de habilidad	Muertes	Clase
Daniel “paiN” Fernández	Global elite (muy alto)	1500	*cheater* (obvio)
Daniel “paiN” Fernández	Global elite (muy alto)	1500	*cheater* (disimulado)
Daniel “paiN” Fernández	Global elite (muy alto)	500	*skilled*
Ramón “LawLiNho” Moñino	Global elite (muy alto)	500	*skilled*
Eloy “bySNH” Lopez	Global elite (muy alto)	500	*skilled*
Abel “SaR3N” Montalbán	Global elite (muy alto)	500	*skilled*
Xavi “nanfyah” Hernández	Legendary Eagle (alto)	500	*skilled*
Diego “XPLICIT” Preciado	Master Guardian Elite (medio)	500	*skilled*

Como se puede ver en la Tabla 44, la mayoría de jugadores que participan en la recolección de datos, disponen de un nivel alto o muy alto, con lo que debería ser más difícil para los modelos distinguir entre clases, dado que el movimiento de ratón de los jugadores más experimentados es más preciso y rápido.

En total, se dispondrá aproximadamente (puesto que, en algunos casos, la recolección fue un poco mayor) de 3000 instancias con clase *skilled*, y 3000 de clase *cheater*.

5. 3. Algoritmos propuestos

Se propone una variedad muy amplia de algoritmos, con tal de comprobar el comportamiento de cada uno de ellos respecto al problema planteado en el presente trabajo.

Los algoritmos propuestos son los siguientes: *BayesNet*, *Naive Bayes*, *SimpleLogistic*, *SMO*, *Ibk*, *PART*, *J48*, *RandomTree*, *RandomForest* y *MLP* (Perceptrón Multicapa).

Todos los algoritmos forman parte de Weka, y la parametrización de los mismos será la utilizada por defecto en dicho programa, excepto en el Perceptrón Multicapa, donde se utilizará una tasa de aprendizaje de 0.15, con 250, 500 y 1000 ciclos. Por último, se realizarán algunas pruebas con *Deep Learning* mediante la herramienta RapidMiner.

5. 4. Parametrización de los datos

Cada archivo *csv* proveniente de una *demo* de un jugador deberá ser convertido a un número de archivos *arff*. Este número vendrá dado por las combinaciones posibles de los parámetros de entrada de *demoParser*, que son los siguientes:

- **Tipo de ángulo.** Se proponen dos formas:
  - Recoger las X modificaciones de ángulos anteriores a una muerte. Con esta primera forma, no habría dos ángulos iguales de forma continua.
  - Recoger los X ángulos anteriores a una muerte. Con esta segunda forma, se podrían encontrar (y, de hecho, es muy probable), dos ángulos iguales de forma continua.
- **Tipo de patrón.** Se proponen cuatro tipos de patrón:
  - Ángulos en bruto.
  - Ángulos en bruto y velocidad transcurrida en el movimiento.
  - Velocidades parciales y velocidad y desviación total.
  - Velocidades y desviaciones parciales, y velocidad y desviación total.
- **Número de muertes por patrón.** Se propone estudiar de 1 a 4 muertes por patrón, siendo 1 el caso base. No se consideran más de 4, ya que, suponiendo una partida en el que un jugador no pase de 4 muertes, dicho jugador no podría ser evaluado por el modelo generado.
- **Número de ángulos anteriores a considerar.** Se propone estudiar el conjunto cerrado {*5, 10, 15, 20, 25, 30*}. Puede que, en algún caso, si se observa una mejora conforme al aumento de este parámetro, se necesite probar con un mayor número.

5. 5. Pruebas de evaluación

Las pruebas de evaluación se llevarán a cabo realizando todas las combinaciones posibles entre algoritmos y datos. En este caso, se realizarán todas en Weka, utilizando la herramienta *Experimenter*, que permite lanzar varias pruebas de forma simultánea, facilitando las mismas.

Cada prueba se realizará entrenando el modelo con un 75% de las instancias, y testeando el mismo con el 25% restante. Las instancias se separarán de forma aleatoria por Weka.

Debido a que se proponen 12 algoritmos y 192 conjuntos de datos (número de posibles combinaciones entre los parámetros de los datos), se realizarán un mínimo de **2304 pruebas**, de las cuales se realizará un análisis, para observar el comportamiento de los algoritmos con los conjuntos de datos, y viceversa.

En dicho análisis, se observará la influencia de los parámetros de los conjuntos de datos en los algoritmos, y, si se observa una posible mejora en alguno de los casos, se explorará dicha opción si es viable.

En los mejores modelos, se realizará validación cruzada (método de validación de modelos en el que se separa el conjunto inicial de datos en particiones y se realizan tantas pruebas como particiones haya, entrenando con todas las particiones menos una, y

realizando el test con la sobrante) para verificar los porcentajes obtenidos hasta ahora con el método de separación de conjunto de entrenamiento y test.

Finalmente, se realizará un test con 1000 instancias (500 de cada clase) con el mejor o los mejores modelos. Dichas instancias serán totalmente ajenas a los datos recogidos anteriormente, y contendrá 300 instancias de *cheat* con opciones aún más inhumanas que las recolectadas hasta ahora, y otras 200 instancias con opciones más humanas que las recolectadas hasta ahora, como se muestra a continuación:

Cheater legit	Cheater rage
Aimbot_key = 2 *(click_derecho)*	Aimbot_key = 2 *(click_derecho)*
Aimbot_bone = 8 *(cabeza)*	Aimbot_bone = 8 *(cabeza)*
Aimbot_smooth = 500	Aimbot_smooth = 150
Aimbot_factorDist = 3.5	Aimbot_factorDist = 3.5
Aimbot_xFOV = 6	Aimbot_xFOV = 80
Aimbot_yFOV = 5	Aimbot_yFOV = 25

Como se puede ver, se ha aumentado un poco la suavidad del movimiento (*smooth*) en el caso *legit*, y se ha reducido aún más el campo de visión, consiguiendo un aspecto más humano. Además, en el caso del *cheater rage* (obvio), se ha disminuido un poco la suavidad y se han aumentado el campo de visión, consiguiendo así un aspecto de *cheater* muy obvio.

Dicho test, aportará una valoración final sobre el modelo, ya que estará clasificando datos con los que no había trabajado antes.

5. 6. Resultados de pruebas de evaluación

En el presente apartado, se realizará un análisis de los resultados de las pruebas que miden la evaluación del sistema para extraer los mejores conjuntos de datos, los mejores algoritmos, las mejores combinaciones entre ambos, y el mejor modelo final, es decir, aquel que presente el mayor porcentaje de clasificación correcta de las instancias.

5. 6. 1. Conjuntos de datos

A continuación, se muestra un análisis de la media de los porcentajes de acierto en la clasificación de los modelos para cada distinto tipo de patrón, dentro de los dos tipos de ángulos que se han establecido.

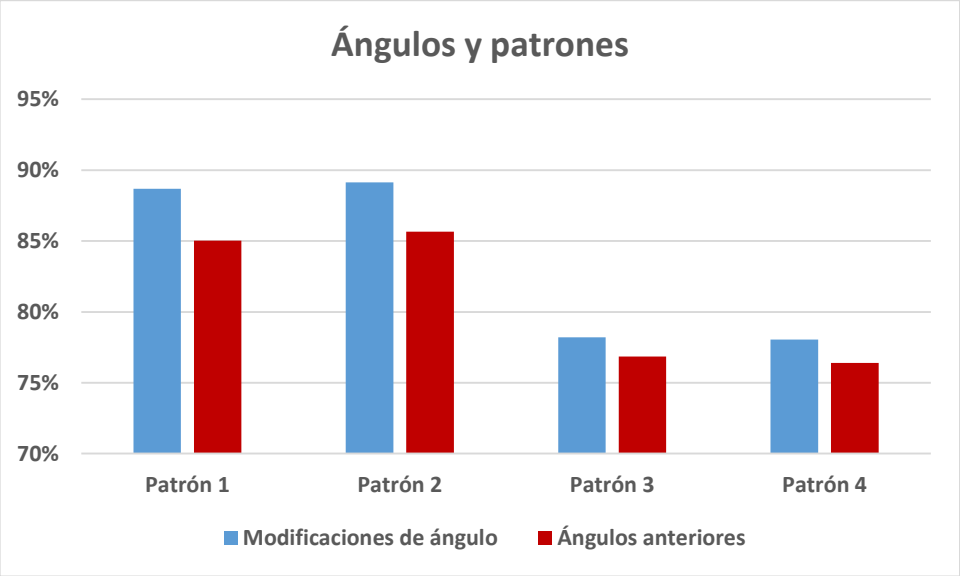


Figura 24. Gráfica comparativa de tipos de ángulos y patrones

Como se puede observar en la Figura 24, la media de los porcentajes de clasificación correcta en todos los patrones posibles, es mayor en el caso de recoger las X modificaciones de ángulos anteriores a una muerte, que en el caso de recoger los ángulos en bruto. Esto no quiere decir que no haya un conjunto de datos dentro de esta segunda forma de recoger los ángulos que sea candidato a ser el mejor modelo, pero sí que quiere decir, que recoger las modificaciones de ángulos (donde no se puede tener 2 veces el mismo ángulo de forma continuada) aporta más información, por norma general, que recoger los ángulos en sí mismos.

Además, se puede observar que los mejores tipos de patrón, son el primero y el segundo, es decir, aquellos donde se ven involucrados los ángulos en bruto. En aquellos donde estos no aparecen, los resultados son significativamente peores (en general, puesto que, es una media). Esto coincide con los parámetros utilizados por VACNet.

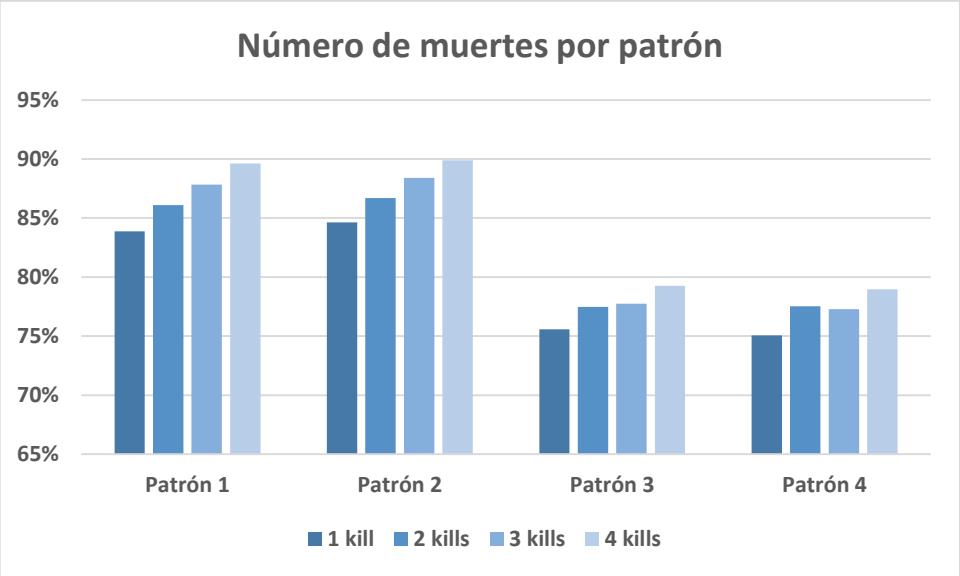


Figura 25. Número de muertes por patrón

Como se puede ver en la Figura 25, a medida que aumenta el número de muertes recogidas en un patrón, los modelos mejoran. No obstante, también se debe a que el número de patrones decrece conforme a este número aumenta.

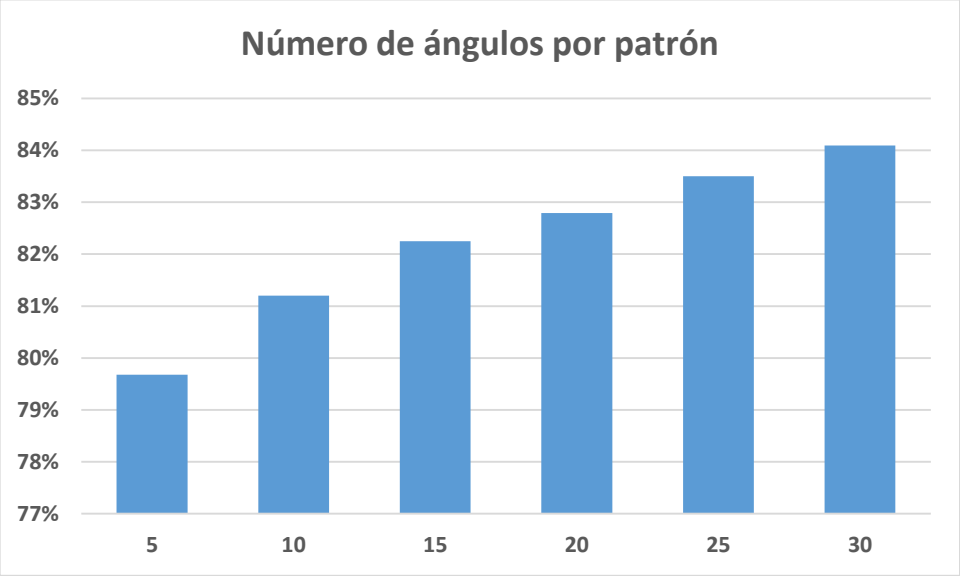


Figura 26. Número de ángulos por patrón

Por norma general, a medida que aumenta el número de información (ángulos) por patrón, aumenta el porcentaje de clasificación correcta de los modelos. No obstante, como se verá en los siguientes apartados, no funciona así en todos los algoritmos. Algunos de los algoritmos, tienen un mejor comportamiento con un menor número de datos, lo que significará, como es obvio, que los ángulos más recientes a una muerte son más significativos que los más lejanos en tiempo.

5. 6. 2. Algoritmos

En la Figura 27, se muestra una comparación realizada entre los diferentes algoritmos utilizados, representando en dos columnas por algoritmo, el comportamiento de cada uno de ellos con las distintas formas de recoger los ángulos, dado son la principal división de datos realizada en el trabajo.

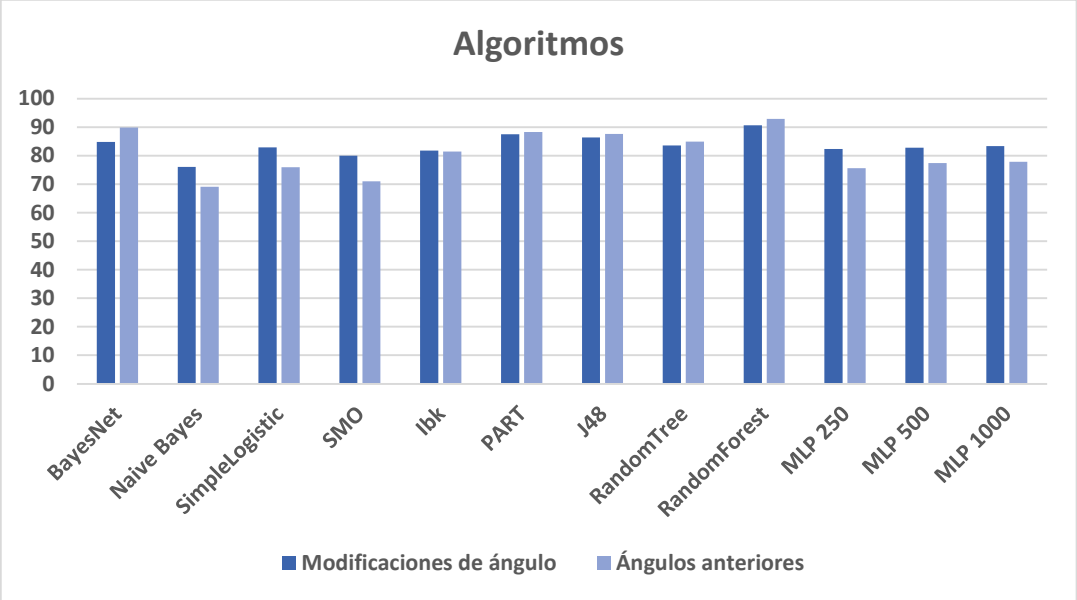


Figura 27. Algoritmos frente a tipo de ángulo

Observando el gráfico mostrado en la Figura 27, se pueden obtener dos conclusiones importantes.

La primera de estas conclusiones es que, a pesar de haber concluido en el apartado anterior que las modificaciones de ángulo son el mejor tipo de ángulo recolectado, en algunos de estos algoritmos no es así. De hecho, en la mitad de los 10 algoritmos utilizados (dado que *MLP* cuenta como uno, a pesar de utilizar distintos parámetros), ocurre que los ángulos en bruto obtienen mejor resultado que las modificaciones de ángulos. Con esto podemos observar que, se pueden realizar análisis de comportamientos colectivos, pero que, realmente, no engloban todos los comportamientos posibles.

La segunda conclusión que se puede extraer, es que hay algoritmos que son claramente superiores a otros. Por ejemplo, el algoritmo *RandomForest* es el único capaz de obtener una media de porcentajes de clasificación por encima del 90%, además, con ambos tipos de recolección de ángulos. Cerca del 90% se encuentran *BayesNet*, utilizando los ángulos anteriores en bruto o *PART* y *J48* con ambos conjuntos de datos.

No obstante, se observa que, en cualquier caso, los resultados en general son bastante prometedores, ya que, en muchos de los algoritmos se obtiene una media tan alta, significará que habrá resultados muy buenos que compensen otros peores.

5. 6. 3. Mejores modelos

Tras haber realizado un análisis colectivo, tanto de los conjuntos de datos, como de los algoritmos, se procederá a analizar los mejores modelos. Para ello, se mostrarán los mejores resultados para cada algoritmo, y el conjunto de datos que produce dicho resultado. Los conjuntos de datos se expresarán con el formato *x1_x2_x3_x4*, donde:

- *x1* será la forma de recoger el ángulo, siendo el valor 1 las modificaciones de ángulos, y el valor 2 los ángulos en bruto.
- *x2* será el tipo de patrón utilizado, siendo un valor entre 1 y 4, que expresa en orden los tipos expuestos en el apartado de parametrización de datos.
- *x3* será el número de muertes por patrón.
- *x4* será el número de ángulos antes de una muerte.

Tabla 45. Mejores modelos de cada algoritmo

Algoritmo	*Conjunto de datos*	*% split*
BayesNet	**2_4_4_30**	**97,63**
Naive Bayes	1_2_4_5	90,55
SimpleLogistic	1_2_4_30	92,91
SMO	1_2_4_30	92,39
Ibk	1_1_1_10	89,29
PART	1_2_2_20	95,03
J48	1_1_1_5	94,51
RandomTree	1_1_1_5	92,55
RandomForest	**1_2_4_20**	**97,38**
RandomForest	**2_3_4_25**	**98,17**
MLP 250	1_2_3_30	93,33
MLP 500	1_2_3_30	92,75
MLP 1000	1_2_4_25	92,39
Deep Learning	**1_2_4_30**	**96,34**

Como se puede observar en la Tabla 45, donde se muestran los mejores modelos de cada algoritmo, predominan los conjuntos de datos donde se ha recolectado las modificaciones de ángulos (tipo 1 de recolección). No obstante, 2 de los 4 mejores modelos son precisamente aquellos que no cumplen con esta tendencia, es decir, recolectan los ángulos en bruto (tipo 2 de recolección).

Además, como se había observado en el análisis de conjuntos de datos, los tipos de patrón que más aparecen son los tipos 1 y 2, es decir, aquellos que utilizan los ángulos y no las velocidades de movimiento. Sin embargo, de nuevo, 2 de los 4 mejores modelos rompen con esta regla, y utilizan estos últimos tipos de patrón.

Únicamente en 3 de los 11 algoritmos se utiliza una muerte por patrón, siendo estos algoritmos *Ibk*, *J48*, y *RandomTree*. Además, en estos, también coincide que el número de ángulos recogidos es de 5 o 10, es decir, el mínimo. Con esto, se deduce que dichos

algoritmos funcionan mejor con pocos datos (siempre y cuando dichos datos aporten la información necesaria, es decir, sea representativa).

En el resto, predominan 3 y 4 muertes por patrón, y en todos ellos excepto en *Naive Bayes* (donde se utilizan 5 ángulos) se utilizan entre 20 y 30 ángulos, es decir, la mayor cantidad de datos posible. Con esto, se deduce que dichos algoritmos funcionan mejor con una mayor cantidad de datos (siempre y cuando dichos datos sean representativos).

Es importante destacar que todos los algoritmos tienen al menos un modelo (y en la mayoría de ellos, varios) donde el porcentaje de clasificación correcta supera el 90%, excepto *Ibk*, donde se tiene un mejor resultado de 89,29%. Esto indica que se ha realizado una correcta recolección de datos y elección de parámetros.

Los mejores algoritmos son claramente *BayesNet* (modelo probabilístico) y *RandomForest* (árboles de decisión), ya que con ambos se supera incluso el 97,5% de clasificación correcta. Después les siguen *PART* (reglas de decisión) y *J48* (árbol de decisión), con, aproximadamente, un 95%, y *MLP* con una tasa de aprendizaje de 0,15 y 250 ciclos, con un 93,33%.

En el algoritmo *RandomForest*, se muestran un resultado para el conjunto de datos donde se recogen modificaciones de ángulos, y otro resultado para el conjunto de ángulos en bruto, puesto que es el único algoritmo donde ambos tipos de recogida de ángulos obtienen resultados muy similares.

Por último, se muestra la mejor de las pruebas realizadas con *Deep Learning*. Se realizaron únicamente pruebas con los 10 mejores conjuntos de datos para *MLP*, debido a la similitud con este algoritmo, y al alto tiempo computacional empleado en cada una de las pruebas, dado que, además, en cada una de ellas se agregaba un componente de optimización de parámetros, que realizaba aproximadamente 500 combinaciones por prueba. El porcentaje es de un 96,34%, siendo este más elevado que en *MLP*. No obstante, sigue siendo menor que los otros dos mejores algoritmos. Seguramente, con un modelo más complejo, como el modelo *Keras*, se podría ajustar aún más este porcentaje.

A continuación, se muestra una tabla con los porcentajes de clasificación correcta para los 4 mejores modelos obtenidos, donde en lugar de utilizar el método de separación de datos, se utilizará validación cruzada con 10 particiones.

Tabla 46. Validación cruzada mejores modelos

Algoritmo	*Conjunto de datos*	*% split*	*% validación cruzada*
BayesNet	2_4_4_30	97,63	**95,54**
RandomForest	1_2_4_20	97,38	**95,15**
RandomForest	2_3_4_25	98,17	**96,33**
DeepLearning	1_2_4_30	96,34	**98,10**

Una vez mostrados los resultados para los mejores modelos, se mostrarán los mejores resultados para los conjuntos de datos de una muerte por patrón, dado que son los más asequibles a la hora de implementar, tanto en sencillez, como en versatilidad (puesto que

puede haber jugadores que realicen únicamente una muerte en la partida), y se realizará una comparación con los resultados anteriores.

En la Tabla 47, se muestran los mejores resultados de los algoritmos *RandomForest* y *BayesNet* (dado que son los mejores) con una muerte por patrón.

Tabla 47. Mejores resultados con una muerte por patrón

Algoritmo	*Conjunto de datos*	*% split*	*% validación cruzada*
BayesNet	2_3_1_30	94,97	**95,75**
RandomForest	2_3_1_30	95,69	**95,74**
RandomForest	1_1_1_5	96,21	**95,75**

En primer lugar, se observan unos resultados muy parecidos en ambos casos, ambos sobre el 95%. Además, se observa que, en el caso de *BayesNet*, el patrón utilizado es el tipo 3, que es el más parecido al tipo 4, el utilizado en el mejor modelo expuesto hasta ahora con este algoritmo, con el mismo número de ángulos anteriores.

También se observa que el conjunto de datos *2_3_1_30* aparece en ambos algoritmos, por lo que se deduce que es uno de los mejores. No obstante, con *RandomForest* se ve superado por el conjunto de datos *1_1_1_5* por muy poco.

Como este último conjunto de datos no es mejorable, puesto que, si se retiran más datos de un patrón, carecería de información suficiente para su clasificación, se procede a intentar mejorar los otros dos.

En *RandomForest*, se observa una ligera mejora conforme al aumento del número de ángulos, es decir, del conjunto *2_3_1_5* al conjunto *2_3_1_30*. Por ello, se probará a aumentar en saltos de 10 unidades el número de ángulos por patrón, hasta que se produzca una bajada en el porcentaje de clasificación realizada correctamente, como se muestra en la Tabla 48.

Tabla 48. Mejora del modelo RandomForest con una muerte por patrón

Algoritmo	*Conjunto de datos*	*% split*	*% validación cruzada*
RandomForest	2_3_1_30	95,69	**95,74**
RandomForest	2_3_1_40	96,40	**95,88**
RandomForest	2_3_1_50	96,07	**95,56**

Como modelo final del algoritmo *RandomForest* con una muerte por patrón, se obtiene el conjunto de datos *2_3_1_40*, es decir, con 40 ángulos por patrón. Este conjunto obtiene sobre un 96% de acierto en los 2 tipos de validación.

Por otra parte, en *BayesNet* se detecta que el algoritmo funciona mejor con más datos por patrón, puesto que se observa esto mismo, tanto en el mejor modelo con una muerte por patrón, como con el mejor modelo con cuatro muertes por patrón. Por ello, se prueba a aumentar el número de ángulos recogidos en un patrón de forma secuencial, como se muestra en la Tabla 49.

Tabla 49. Mejora del modelo BayesNet con una muerte por patrón

Algoritmo	*Conjunto de datos*	*% split*	*% validación cruzada*
BayesNet	2_3_1_30	94,97	**95,75**
BayesNet	2_3_1_64	96,01	**97,14**
BayesNet	2_3_1_128	97,52	**98,60**
BayesNet	2_3_1_256	98,69	**99,40**
BayesNet	2_3_1_512	X	**X**

Como modelo final del algoritmo *BayesNet* con una muerte por patrón, se obtiene el conjunto de datos *2_3_1_256*, es decir, con 256 ángulos por patrón. Como, en este segundo tipo de recolección de ángulos, se utiliza un ángulo por *tick*, y las partidas se desarrollan en servidores de 64 y 128 *ticks* por segundo, se recolectarán los datos de los 2 o 4 segundos anteriores. Este conjunto obtiene sobre un 99% de acierto en los 2 tipos de validación.

El conjunto de datos con 512 ángulos por patrón, ofrecía un conflicto, dada la enorme cantidad de datos por patrón, por lo que no es viable.

Como los resultados para una muerte por patrón son incluso mejores que los expuestos con más de una muerte, se decide descartar estos últimos.

Una vez escogidos los mejores modelos, se realiza un test sobre ellos en el que, se utilizarán todos los patrones para entrenar (los más de 6000), y se pasará como conjunto de test el explicado en el apartado de pruebas. Los resultados de la clasificación se muestran en la Tabla 50.

Tabla 50. Test de los mejores modelos con datos externos

Algoritmo	*Conjunto de datos*	*% fichero externo*
BayesNet	2_3_1_256	**99,80**
RandomForest	1_1_1_5	**97,60**
RandomForest	2_3_1_40	**90,90**

Se observa que el modelo *RandomForest*, realiza **sobre-aprendizaje** dado que su porcentaje de acierto disminuye considerablemente al tratar con datos nuevos. Concretamente, la mayor parte del error es cometido con las 200 muertes recogidas con nuevas opciones en el *cheat*, que hacen que este parezca aún un movimiento más humano. Es decir, este modelo se adapta mal a los datos con los que no ha sido entrenado, a pesar de que aprende correctamente.

Por otra parte, el mismo algoritmo, con el conjunto de datos *1_1_1_5*, obtiene un 97,60% de acierto. Esto se considera un resultado considerablemente bueno, dada la escasa cantidad de datos necesaria que utiliza el modelo.

Por último, el algoritmo *BayesNet* consigue clasificar correctamente 99,80% de las instancias, lo que se considera un resultado excelente, ya que apenas comete errores (2 de

las 1000 instancias). Este modelo, sería un candidato perfecto para ser implementado a mayor escala.

A continuación, se muestran las matrices de confusión de estos últimos 2 modelos, y *RandomForest* y conjunto de datos *1_1_1_5, y BayesNet* y conjunto de datos *2_3_1_256*, en la Tabla 51 y Tabla 52 respectivamente.

Tabla 51. Matriz de confusión del modelo RandomForest 1_1_1_5

Clasificado como -->	***Skilled***	***Cheater***
Skilled	500	0
Cheater	24	476

Tabla 52. Matriz de confusión del modelo BayesNet 2_3_1_256

Clasificado como -->	***Skilled***	***Cheater***
Skilled	500	0
Cheater	2	498

Como se puede observar en la Tabla 51, clasifica correctamente todas las instancias de la clase *skilled*, sin cometer ningún fallo, lo cual es, a pesar de cometer 24 fallos, un comportamiento muy bueno, dado que es el número que conviene reducir para no inculpar injustamente a jugadores inocentes que no utilizan *cheats*. De los 24 fallos producidos al clasificar *cheaters* como *skilled*, solo 2 fueron producidos con las opciones inhumanas, mientras que las otras 22, fueron producidas por **las opciones más humanas** del *cheat*. Esto se debe a que el modelo no ha sido entrenado con dichas opciones, pero, podría detectar estos 24 fallos si se entrenara el modelo con dichas opciones.

En la Tabla 52, se puede observar como, al igual que el modelo anterior, este clasifica correctamente todas las instancias de la clase *skilled*, sin cometer ningún fallo. Además, los 2 fallos son producidos únicamente por **las opciones más humanas** del *cheat*, es decir, clasifica correctamente todas las inhumanas, y casi todas las instancias *cheater* que intentan parecer humanas. De nuevo, con un entrenamiento en el que se utilizaran más opciones, se podría lograr que clasificara correctamente esas 2 instancias que no logra clasificar.

Como **conclusión** de este apartado, se puede decir que se ha logrado obtener los resultados deseados en un principio, logrado distinguir correctamente, con unos porcentajes muy altos, a usuarios que utilizan *cheats* de usuarios que no los utilizan.

No obstante, al no tratarse de modelos perfectos, puesto que no existen (pues la cantidad de *cheats* es inmensa, así como su configuración), la única medida que se podrá tomar con la detección de *cheaters* mediante los modelos encontrados, será el envío de las repeticiones de los partidos del jugador sospechoso a revisión, es decir, a *Overwatch*.

También se ha observado como cada algoritmo maneja mejor unos tipos de datos en concreto, así como unas cantidades distintas de información.

6. GESTIÓN DEL PROYECTO

En este capítulo se describen los aspectos relativos a la gestión del proyecto. Se detallan la planificación del trabajo, el presupuesto para el desarrollo de este proyecto en una empresa estimando los costes necesarios y el impacto socioeconómico.

6. 1. Planificación

El trabajo de fin de grado tiene dos grandes bloques de desarrollo.

El primero de ellos, se realizó una vez decidida la temática del trabajo, y se empezó el 2 de julio de 2018. En esta parte del trabajo, se desarrollaron las herramientas software necesarias para la recopilación y manipulación de datos del trabajo. El desarrollo de este primer bloque, tuvo lugar durante las vacaciones de verano, y finalizó a mediados de agosto.

El segundo de los bloques, se empezó a realizar una vez quedó formalizada la propuesta del trabajo al tutor, y se realizó una primera reunión con el mismo. En este segundo bloque se realiza la recogida de datos que se emplean en las pruebas, y el desarrollo de la memoria.

Ambos bloques quedan representados en la Tabla 53, donde se expone la duración en horas de cada uno de los bloques, y sus fechas de comienzo y finalización.

Tabla 53. Planificación general del trabajo de fin de grado

ITERACIÓN	Duración (horas)	Fecha de comienzo	Fecha de finalización
WhatAreThose	**61**	**02/07/2018**	**24/07/2018**
DemoInfo	**12**	**25/07/2018**	**28/07/2018**
DemoParser	**26**	**29/07/2018**	**13/08/2018**
ManagerArffs	**5**	**13/08/2018**	**14/08/2018**
Memoria	**145**	**04/12/2018**	**15/01/2019**

Hay que destacar que dentro de cada uno de los desarrollos de las herramientas software, hay un proceso de investigación y de estudio de código previo, así como una versión inicial de los requisitos y unas pequeñas pruebas con cada una de ellas.

En la Tabla 54, se detallan todas las tareas generales y sub-tareas del segundo bloque, es decir, la memoria.

Tabla 54. Planificación del desarrollo de la memoria

ETAPA	Duración (horas)	Fecha de comienzo	Fecha de finalización
Introducción	**19**	**04/12/2018**	**08/12/2018**
Estado del arte	**36**	**09/12/2018**	**16/12/2018**
Análisis y diseño del sistema	**18**	**17/12/2018**	**20/12/2018**
Requisitos del sistema	**5**	**17/12/2018**	**18/12/2018**
Casos de uso	**5**	**18/12/2018**	**19/12/2018**
Arquitectura del sistema	**8**	**19/12/2018**	**20/12/2018**
Implementación del sistema	**22**	**21/12/2018**	**26/12/2018**
Resultados y evaluación	**41**	**27/12/2018**	**07/01/2019**
Gestión del proyecto	**12**	**08/01/2019**	**12/01/2019**
Conclusiones	**4**	**13/01/2019**	**15/01/2019**

A continuación, en la Figura 28, se muestra el cronograma del proceso de elaboración del trabajo de fin de grado.

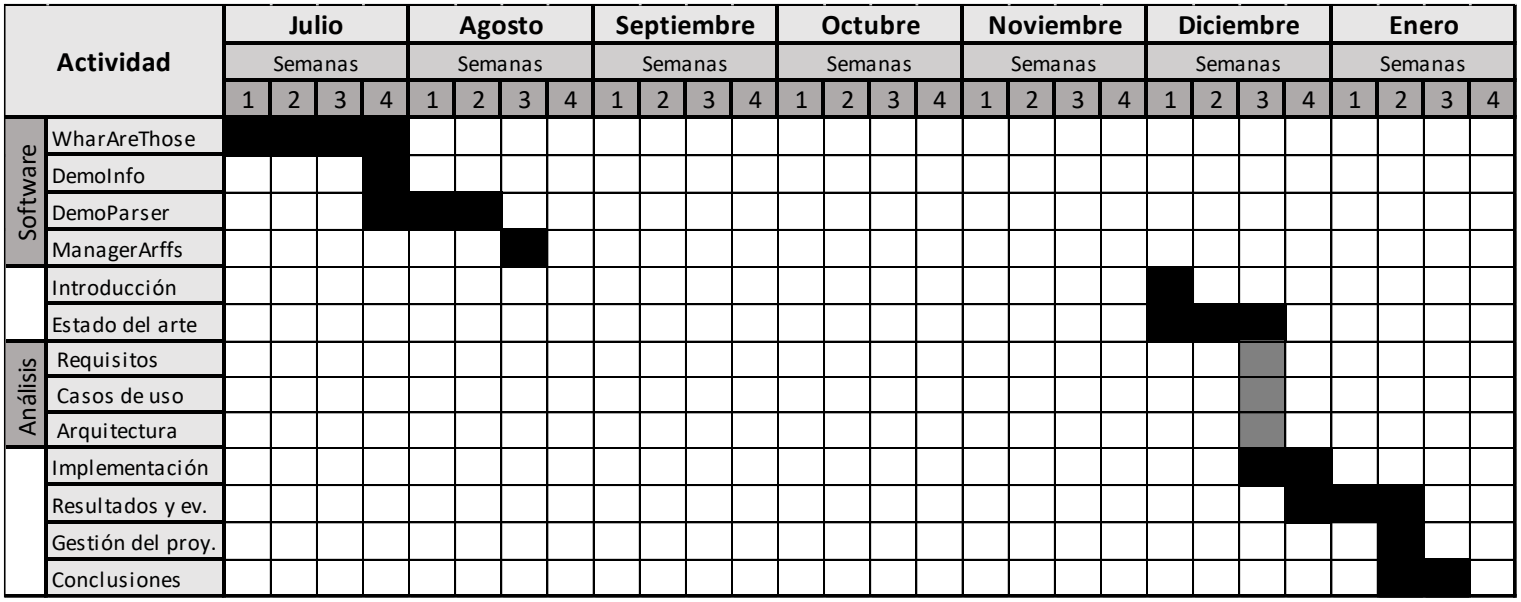


Figura 28. Cronograma de la planificación

6. 2. Presupuesto

A continuación, se detalla el presupuesto que sería necesario presentar para el desarrollo del proyecto que representa el presente trabajo de fin de grado. Para ello, se calcularán los costes de personal, los costes materiales y los costes indirectos. Finalmente se realizará el cálculo del coste total.

6. 2. 1. Coste de personal

Los roles que participan en el proyecto son los siguientes:

- **Jefe de proyecto**. Persona encargada de dirigir y coordinar el proyecto.
- **Ingeniero principal**. Persona responsable del análisis, diseño e investigación del proyecto.
- **Ingeniero desarrollador**. Persona encargada de desarrollar y probar el sistema.

Una vez definidos los roles, será necesario establecer los salarios y calcular el coste de cada uno de ellos al proyecto, teniendo en cuenta el número de horas de desempeño en el mismo. Los salarios se establecen a partir de la tabla salarial publicada en la resolución de la Dirección General de Empleo del 22 de febrero de 2018 por el BOE, en el XVII Convenio colectivo estatal de empresas de consultoría y estudios de mercado y de opinión pública. El coste de personal se detalla en la Tabla 55.

Tabla 55. Coste del personal del proyecto

Rol	Coste por hora (€)	Horas de trabajo	Total (€)
Jefe de proyecto	29,74	17	505,58
Ingeniero principal	27,52	52	1.431,04
Ingeniero desarrollador	20,87	188	3.923,56
Total	-	236	5.860,18

6. 2. 2. Coste material

El coste material incluye el dinero invertido en software y en hardware. Se detallará en la Tabla 56 el software y hardware utilizado en el proyecto, así como su precio (no incluyendo los que no suponen un desembolso económico), el periodo de amortización, el tiempo de uso, y el coste de cada uno de ellos en relación a las tres propiedades anteriores. Este coste, será calculado mediante la siguiente ecuación:

$$coste = \frac{precio}{periodo\ amortización} * tiempo\ uso \quad (1)$$

Tabla 56. Coste material del proyecto

Producto	Precio (€)	Periodo amortización (meses)	Tiempo uso (meses)	Coste en proyecto (€)
PC Desarrollo	3.379,06	48	3	211,19
Licencia Visual Studio Professional 2017	641,00	48	3	40,06
Licencia Microsoft Office 2016	299,00	48	3	18,69
Licencia RapidMiner	5.000	12	3	1250
Total	1519,94 €			

6. 2. 3. Coste total

Con los costes calculados en los apartados anteriores, únicamente queda calcular los costes indirectos para poder calcular el coste total del proyecto. Para calcular dichos costes, se añaden los siguientes costes:

- **Costes indirectos**. Se estima un 10% de costes indirectos, en los que se engloban gastos como la electricidad o el acceso a internet, entre otros.
- **Margen de riesgo**. Se estima un 15% de margen de riesgo, en el que se cubren posibles imprevistos como averías de equipo o adquisición de nuevo software no contemplado.
- **Beneficio del proyecto**. Se estima un 30% de beneficio, es decir, lo que ganaría la empresa que realizara este proyecto.
- **Impuesto**. Se sumará un 21% del coste calculado hasta el momento, que representará el IVA.

En la Tabla 57, se muestra el coste total del proyecto, calculando los porcentajes anteriores.

Tabla 57. Coste total del proyecto

Concepto	Cantidad (€)
Costes directos	7.380,12
Costes indirectos (10%)	738,01
Margen de riesgo (15%)	1.107,02
Beneficio del proyecto (30%)	2.214,04
Base imponible	**11.439,19**
IVA (21%)	2.402,23
Coste total del proyecto	**13.841,42**

6. 3. Plan de riesgos

Como se ha explicado en el *Apartado 1.4. Marco Regulador*, existen una serie de vulnerabilidades, por lo que se ha de elaborar un plan de riesgos en el que se identifiquen dichas vulnerabilidades, y se presenten las medidas oportunas para prevenir los problemas que puedan surgir.

La evaluación de riesgos incluye las siguientes actividades y acciones, tal y como se describe en la Norma ISO/IEC 27001:2005 de la familia de normas ISO 27000 para la seguridad informática:

- Identificación de activos.
- Identificación de los requisitos legales y de negocio relevantes para la identificación de activos.
- Valoración de activos identificados, teniendo en cuenta los requisitos legales y de negocio identificados en el punto anterior, y el impacto de una pérdida de confidencialidad (C), disponibilidad (D) e integridad (I).
- Identificación de amenazas y vulnerabilidades para los activos identificados.
- Evaluación y cálculo del riesgo (alto, medio, bajo).

Las acciones que se pueden tomar ante estos riesgos, son mitigarlos (mediante controles de seguridad), eliminarlos (eliminando el activo), traspasarlos (hacer que terceros se ocupen de los mismos), o asumirlos (el nivel de los mismos es considerado como aceptable).

En este caso, como se utiliza un único equipo para el desarrollo del proyecto, el único riesgo existente sería de un nivel alto, en el que intervendrían el requisito legal de la RGPD y el requisito de negocio de la disponibilidad. Las principales vulnerabilidades de este riesgo serían la avería del equipo, corrupción del sistema de ficheros y filtraciones de datos personales. El impacto del riesgo sería C D I (según los identificadores establecidos anteriormente), y la acción a tomar sería mitigar el riesgo.

6. 4. Impacto socioeconómico

El objetivo principal del proyecto que se desarrolla en el presente trabajo de fin de grado, es su incorporación al videojuego en cuestión, es decir, CS:GO, o, a otro videojuego de disparos en primera persona, adaptando previamente el proyecto a las necesidades y las propiedades del mismo (puesto que no habría que modificar muchos parámetros).

El impacto socioeconómico que se espera una vez aplicado el proyecto al videojuego en cuestión, se espera una reducción considerable del número de jugadores que utilizan *cheats* en el mismo, aplicando un baneo permanente y un posible bloqueo por HWID (*Hardware ID*, identificador único del hardware) o por IP (*Internet Protocol*, Protocolo de Internet, identificador único de conexión) a dichos jugadores. Con estos dos últimos tipos de bloqueo, se evitaría que los *cheaters* baneados vuelvan a formar parte del juego, y, por tanto, se fortalecería aún más la protección del juego.

Con esta reducción, los jugadores que no utilizan ningún *cheat*, se sentirían más seguros y disfrutarían más del tiempo empleado jugando al mismo, mejorando la experiencia a largo plazo de los jugadores. Como consecuencia de esta mejora de experiencia, se espera que la base de los jugadores activos al mes aumente, tanto por parte de los jugadores que ya disponían del juego, como por parte de nuevos jugadores que se incorporen al juego debido a las buenas calificaciones obtenidas en las revisiones de Steam por los usuarios y a la opinión directa de los jugadores que se conozcan entre sí. Este aumento del número de jugadores, produciría un beneficio económico directo al videojuego, dado que aumentaría el número de jugadores que pagan por tener cosméticos y armas decoradas dentro del videojuego. Además, las empresas externas que organizan competiciones y torneos de este videojuego se sentirían más seguras al saber de la reducción de *cheaters*, y organizarían un número aún mayor de competiciones y torneos, con premios incluso mayores de los que se dispone actualmente (dado que también aumentaría el número de equipos participantes).

Con este aumento de equipos, se promovería la creación de clubes de deportes electrónicos, en los cuales se paga a los jugadores por competir con bajo su nombre.

Por último, la empresa que implemente el proyecto, vería un aumento de su reputación, dado que el problema de los *cheaters* es uno de los mayores problemas actualmente en los videojuegos en los que existe algún tipo de competición, y combatirlo mostraría la dureza y contundencia de la empresa contra este gran problema.

Los únicos afectados de forma negativa, serían los vendedores de *cheats*, que verían un decremento en el número de ventas, y, por tanto, en el beneficio, al existir menos compradores interesados en los mismos debido al temor a ser baneados.

7. CONCLUSIONES

En este último capítulo se detallan las conclusiones técnicas y personales obtenidas a lo largo de la realización del trabajo de fin de grado, así como la valoración sobre los resultados obtenidos y los posibles trabajos futuros.

7. 1. Conclusiones

Lo primero a concluir, y lo más importante, es que los objetivos planteados en el presente trabajo, han sido cumplidos tras la realización del mismo.

Se ha logrado construir un sistema de detección de *cheaters* (que utilicen un asistente de puntería) con una tasa de acierto por encima incluso del 99%.

Además, dicho sistema se basa única y exclusivamente en las repeticiones de las partidas de los jugadores (que son directamente obtenibles del servidor de la partida competitiva después de la misma), utilizando la información extraída de dichas repeticiones. Por ello, dicho sistema no es nada intrusivo, que era otro de los objetivos, priorizando así la privacidad de los datos de los jugadores.

Otro de los objetivos era, en medida de lo posible, obtener un bajo número de falsos positivos, es decir, que no reconozca jugadores *skilled* como *cheaters*, inculpando así a jugadores inocentes. En todos los mejores modelos se ha cumplido que la inmensa mayoría de los fallos obtenidos en la clasificación de las instancias ha sido a la hora de clasificar *cheaters* (siendo aun así un número bajo de fallos), con lo que, también se ha conseguido dicho objetivo. No obstante, como se ha mencionado a lo largo de los apartados anteriores, no existen sistemas de clasificación perfectos (puesto que no se pueden prever todas las casuísticas posibles), y, por tanto, aunque se detecte un *cheater*, se deberá enviar la repetición de la partida a un "jurado", que, en este caso, serán los jugadores que tengan acceso a la plataforma *Overwatch* de Valve (plataforma donde se revisan los casos).

El sistema tiene un tiempo de ejecución muy bajo para la gran cantidad de datos que ha manejado, respondiendo de forma rápida y eficaz (en la mayoría de los modelos, y en el caso de los mejores) a las peticiones generadas, con lo que sería posible su implementación en un sistema más complejo, con muchas más instancias, realizando las modificaciones oportunas para ello, y dotando al sistema con las especificaciones hardware y software necesarias para un procesamiento masivo de datos.

Una de las conclusiones técnicas extraídas, es que, aparte de que lo lógico y más óptimo sería no utilizar más de una muerte por patrón, dado que puede haber partidas en las que los jugadores no realicen más de una muerte (aunque sea un caso muy aislado), también coincide que los mejores modelos han sido aquellos que han utilizado dicho número de muertes. No obstante, se ha observado como cada uno de los diferentes algoritmos ha interactuado de distinta forma con los conjuntos de datos, es decir, cada uno de ellos se ha comportado de distinta forma al variar el número de ángulos anteriores a una muerte, el tipo de patrón, la forma de recoger el ángulo y el número de muertes por patrón. Por

ejemplo, los árboles de decisión, por norma general, han funcionado mejor con pocos datos (siempre y cuando dichos datos sean significativos y suficientes para describir una situación) si se trata de ángulos y no de velocidades de movimiento de ratón.

Por último, como conclusión general, se considera que el alumno ha sido capaz de aplicar los conocimientos adquiridos a lo largo del grado de ingeniería informática a un problema de índole científico, y perfectamente aplicable a un sistema real. Además de la aplicación de dichos conocimientos, se ha debido realizar una investigación previa y una adquisición de nuevos conocimientos (como la lectura y escritura en memoria de un proceso), demostrando así una capacidad autodidacta, que se considera muy relevante en el ámbito informático.

Se considera que la elección de la temática del trabajo es apropiada, porque, además de ser interesante y aplicable a un sistema real, está basada completamente en la mención en la que se encuentra el alumno, la inteligencia artificial.

7. 2. Trabajos futuros

En el trabajo de detección de *cheaters* que se ha llevado a cabo, se han realizado una serie de simplificaciones, que, aunque se ha demostrado en apartados previos que no influyen en el desarrollo y comportamiento del sistema, facilitan de alguna manera la obtención de unos mejores resultados debido a que el entorno de pruebas es más cerrado.

La mayor diferencia que han supuesto estas simplificaciones, es que se ha necesitado un menor número de instancias para representar de mejor forma el conjunto total de casuísticas posibles en los datos. Es decir, de no haber realizado dichas simplificaciones, se podrían obtener resultados equivalentes o similares a los obtenidos, reuniendo un mayor número de instancias, que cubran el espectro de datos.

Por ello, en trabajos futuros, se podrían eliminar dichas simplificaciones y observar el comportamiento del sistema en un entorno totalmente real, recopilando las partidas de los jugadores, evaluándolas manualmente, e incluyéndolas en el sistema. Cuando dicho sistema se considere lo suficientemente inteligente, será el mismo el que evalúe las partidas y las incluya a su base de conocimiento, para su posterior uso en la construcción del modelo.

Primero, se debería eliminar la restricción de utilizar únicamente un arma, siendo posible utilizar todas y cada una de las que dispone el videojuego, incluyendo así, no solo rifles automáticos, sino francotiradores, pistolas, fusiles de corto alcance y escopetas.

También se debería eliminar la restricción de apuntar a la cabeza, puesto que los jugadores, aparte de priorizar el apuntado a la cabeza de los enemigos (puesto que el daño en dicha parte del cuerpo es mayor), hay situaciones en las que prefieren no hacerlo así, debido a la falta de habilidad, o a que la situación lo requiere (se conoce que el jugador enemigo está bajo de vida, y es más fácil apuntar a una zona del cuerpo con mayores dimensiones).

Otra restricción a eliminar sería la recopilación de instancias en un único mapa, puesto que existen más de 10 mapas en los que un jugador puede competir. No obstante, como la representación de una instancia se basa en los ángulos a los que apunta un jugador en el momento de la muerte de un jugador, resultaría indiferente a la hora de recopilar instancias.

Una vez eliminadas estas restricciones, el único proceso restante sería la recolección de un inmenso número de instancias con las que se represente el mayor número de casuísticas posibles. Es decir, se deberían utilizar todas las posibles armas en todos los posibles mapas, reuniendo un amplio número de muertes con cada arma en distintas partes del cuerpo, de tal forma que se representen las muertes en todos los ángulos posibles (con diferentes alturas, puesto que, en las pruebas del trabajo, todos los enemigos se encontraban a la misma altura).

Por último, se podrían crear *aimbots* con un comportamiento más humano, es decir, que no calculen una trayectoria hacia el enemigo en línea recta, sino con desviaciones propias de la puntería de una persona.

REFERENCIAS

[1] Steam Charts, "Steam Charts", *Steamcharts.com*. [En línea]. Disponible: https://steamcharts.com/app/730. [Acceso: 04- Dec- 2018]

[2] HLTV, "forsaken caught cheating at eXTREMESLAND; OpTic India disqualified", *HLTV.org*, 2018. [En línea]. Disponible: https://www.hltv.org/news/25118/forsaken-caught-cheating-at-extremesland-optic-india-disqualified. [Acceso: 05- Dec- 2018]

[3] HLTV, "KQLY handed VAC ban", *HLTV.org*, 2014. [En línea]. Disponible: http://www.hltv.org/news/13636-kqly-handed-vac-ban. [Acceso: 05- Dec- 2018].

[4] NVIDIA, *Using Deep Learning to Combat Cheating*. (2018). [Video en línea]. Disponible en: http://on-demand.gputechconf.com/gtc/2018/video/S8732/. Acceso: 05- Dec- 2018

[5] Valve, "Counter-Strike: Global Offensive", *Blog.counter-strike.net*, 2018. [En línea]. Disponible: http://blog.counter-strike.net/index.php/2018/12/21530/. [Acceso: 05- Dec- 2018]

[6] K. Maberry, S. Paustian and S. Bakir, *Using an Artificial Neural Network to detect aim assistance in Counter-Strike: Global Offensive*. 2017

[7] Scream, "Scream Twitter message", *Twitter.com*, 2015. [En línea]. Disponible: https://twitter.com/g2scream/status/607502183869771776. [Acceso: 06- Dec- 2018]

[8] Steam, "Acuerdo de Suscriptor a Steam". [En línea]. Disponible: https://store.steampowered.com/subscriber_agreement/?l=spanish. [Acceso: 06- Dec- 2018]

[9] BOE, Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal, 2000

[10] BOE, Reglamento (UE) 2016/679 del Parlamento Europeo y del Consejo de 27 de abril de 2016 relativo a la protección de las personas físicas en lo que respecta al tratamiento de datos personales y a la libre circulación de estos datos, y por el que se deroga la Directiva 95/46/CE (Reglamento general de protección de datos), 2016

[11] Microsoft, "IDE de Visual Studio, editor de código, VSTS y centro de aplicaciones - Visual Studio", *Visual Studio*. [En línea]. Disponible: https://visualstudio.microsoft.com/es/. [Acceso: 07- Dec- 2018]

[12] Waikato, "Weka 3 - Data Mining with Open Source Machine Learning Software in Java", *Cs.waikato.ac.nz*. [En línea]. Disponible: https://www.cs.waikato.ac.nz/ml/weka/. [Acceso: 07- Dec- 2018]

[13] RapidMiner, "Lightning Fast Data Science Platform for Teams | RapidMiner©", *RapidMiner*. [En línea]. Disponible: https://rapidminer.com/. [Acceso: 07- Dec- 2018]

[14] Notepad++, "Notepad++ Home", *Notepad-plus-plus.org*. [En línea]. Disponible: https://notepad-plus-plus.org/. [Acceso: 07- Dec- 2018]

[15] Valve, "demoinfo", *GitHub*, 2016. [En línea]. Disponible: https://github.com/ValveSoftware/csgo-demoinfo. [Acceso: 07- Dec- 2018].

[16] Keras, "Keras Documentation", *Keras.io*. [En línea]. Disponible: https://keras.io/. [Acceso: 10- Dec- 2018]

[17] Galli, Luca & Loiacono, Daniele & Cardamone, Luigi & Lanzi, Pier Luca. (2011). A cheating detection framework for Unreal Tournament III: A machine learning approach. 2011 IEEE Conference on Computational Intelligence and Games, CIG 2011. 266 - 272. 10.1109/CIG.2011.6032016

[18] Steam, "Unreal Tournament 3", *Store.steampowered.com*. [En línea]. Disponible: https://store.steampowered.com/app/13210/Unreal_Tournament_3_Black/. [Acceso: 11- Dec- 2018]
[19] frk1, "hazedumper", *GitHub*. [En línea]. Disponible: https://github.com/frk1/hazedumper. [Acceso: 23- Dec- 2018].
[20] Microsoft, "ReadProcessMemory function (Windows)", *Msdn.microsoft.com*. [En línea]. Disponible: https://msdn.microsoft.com/es-es/library/windows/desktop/ms680553(v=vs.85).aspx. [Acceso: 18- Dec- 2018]
[21] Microsoft, "WriteProcessMemory function (Windows)", *Msdn.microsoft.com*. [En línea]. Disponible: https://msdn.microsoft.com/es-es/library/windows/desktop/ms681674(v=vs.85).aspx. [Acceso: 18- Dec- 2018]
[22] SEGnosis, "Real Aimbot Crosshair Distance", *Unknowncheats.me*. [En línea]. Disponible: https://www.unknowncheats.me/forum/c-and-c/114805-real-aimbot-crosshair-distance.html. [Acceso: 20- Dec- 2018]
[23] uLLeticaL, "Aim Botz - Training", *Steamcommunity.com*, 2014. [En línea]. Disponible: https://steamcommunity.com/sharedfiles/filedetails/?id=243702660. [Acceso: 22- Dec- 2018]
[24] J. Sotillo and C. Martínez, "Inteligencia Artificial en los Videojuegos". [En línea]. Disponible: https://www.academia.edu/22310357/Inteligencia_Artificial_en_los_Videojuegos. [Acceso: 17- Jan- 2019]
[25] Tulk Jesso, Stephanie & Cumings, Ryon & Zafar, Taha & Wiese, Eva. (2018). Better know who you are starving with: Judging humanness in a multiplayer videogame. 1-6.
[26] Á. Parra and C. Linares, "Búsqueda de caminos en mapas de videojuegos", E-archivo.uc3m.es, 2015. [En línea]. Disponible: https://e-archivo.uc3m.es/bitstream/handle/10016/23954/TFG_Alvaro_Parra_De_Miguel_2015.pdf. [Acceso: 18- Jan- 2019]
[27] Cui, Xiao & Shi, Hao. (2010). A*-based Pathfinding in Modern Computer Games [En línea]. Disponible: https://www.researchgate.net/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games [Acceso: 18- Jan- 2019]
[28] Amato, Alba. (2017). Procedural Content Generation in the Game Industry. [En línea]. Disponible: https://www.researchgate.net/publication/315862589_Procedural_Content_Generation_in_the_Game_Industry [Acceso: 18- Jan- 2019]
[29] YourGamesZone. Captura de pantalla GTA V, “GTA 5 utilizará funcionalidades de dualshock 4”. [En línea]. Disponible: https://www.yourgameszone.com/gta-5-utilizara-funcionalidades-dualshock-4/
[30] "Premier Tournaments - Liquipedia Counter-Strike Wiki", Liquipedia.net, 2018. [En línea]. Available: https://liquipedia.net/counterstrike/Premier_Tournaments. [Acceso: 22- Dec- 2018]