[IIC2433] Entrega 2 Proyecto Minería de datos

Maximiliano Friedl y Felipe Gómez

Jueves 6 de diciembre de 2018

1 Introducción: *Heroes of the Storm*

Se desea hacer un proyecto de optimización en torno al videojuego Heroes of the Storm. El videojuego se juega en partidas 5 vs 5, en donde cada jugador escoge a un héroe. Para el modo de juego a analizar, la elección de los héroes siempre se realiza con un formato en específico (ver sección "El *Draft*" en el Anexo). La idea a realizar es que dado un estado actual del *draft* con héroes prohibidos y elegidos hasta el momento, se desea indicar cuál es el equipo que más probablemente obtendrá la victoria, así como las probabilidades de ganar de cada equipo. De esta manera, se obtendrá una herramienta potente a la hora de tomar la decisión de qué héroe seleccionar con el objetivo de que sea más probable ganar la partida.

El problema de la elección de la composición óptima de un equipo tiene aplicaciones en innumerables áreas de la vida cotidiana, como la elección de profesionales para una empresa o la convocatoria de jugadores de selecciones deportivas, por lo que resolver existosamente el problema planteado puede ser el punto de partida hacia otras áreas de investigación.

Se elige HOTS para hacer el proyecto por su gran y entusiasta comunidad y su "simplicidad" en las mecánicas del juego. En otros videojuegos hay una gran cantidad de elecciones previas, como glifos u objetos dentro del juego. En HOTS, en cambio, la decisión más relevante son los héroes a seleccionar. Además, los dos integrantes del grupo tienen experiencia en el juego, por lo que hay un valor adicional al tener la capacidad de interpretar de mejor forma los resultados (ver que se cumplan restricciones del juego que podrían no haber sido consideradas en el modelo o analizar si tiene sentido la predicción que se está haciendo en cada caso).

2 Investigación del estado del arte y definición del problema

2.1 Resultados de la investigación

Inicialmente, el tema de interés fue la recomendación de un héroe durante el *draft*. Al realizar la investigación sobre el estado del arte actual del problema, se concluyó que no hay investigaciones de relevancia sobre este campo. Tampoco hay trabajos del mismo género y ni siquiera hay algo similar para otros videojuegos de otros géneros. Las investigaciones actuales se centran en cómo deben ser los jugadores para maximizar la eficiencia de un equipo y no en la elección de los personajes dentro del videojuego. Debido a esto, se eligió un nuevo problema a resolver.

2.2 Problema a solucionar, supuestos y relación con el dilema original

Lo que se decició hacer como proyecto es: *dado un estado del draft, estimar la probabilidad de victoria de un equipo del otro*. Para ello, se tomarán en consideración los siguientes supuestos:

- En una partida no existen empates, es decir, siempre un equipo gana y el otro pierde

- Se hará un clasificador binario sobre el resultado de la partida para el equipo 1, es decir, a cada resultado $y \in 0, 1$ se le asignará 1 si el equipo 1 fue el ganador y 0 en caso contrario
- Se asume que existe una forma válida de encontrar la probabilidad con la que un dato pertenece a la categoría y que esto se puede interpretar como la probabilidad de victoria de un equipo. Es otras palabras, si el resultado de la clasificación para un estado X del *draft* es 1 con una probabilidad de 0.6, esto significa que el equipo 1 tiene un 60% de probabilidades de victoria dado el escenario X
- El videojuego ha lanzado héroes nuevos constantemente desde su lanzamiento hace varios años. Por tanto, los héroes más nuevos no existían al momento en el que se jugaron partidas más antiguas. Por ejemplo, el último héroe en salir, Orphea, está disponible desde noviembre, por lo que estrictamente hablando no sería válido que se recomendara para una partida jugada en el 2016. A pesar de esto, se ha decidido ignorar esta restricción, pues se asume que el modelo será ocupado para partidas "en el presente" (con las últimas actualizaciones de héroes, mapas y balance del juego), por lo tanto, no tiene sentido hacer recomendaciones para un estado del videojuego en el que nadie volverá a jugar jamás
- La predicción de la victoria de un equipo por sobre el otro asumirá que ambos equipos se encuentran en igualdad de condiciones: los jugadores de ambos equipos son igualmente hábiles, se coordinan con la misma eficiencia y tienen el mismo grado de estabilidad en la conexión. La idea es asumir en la medida de lo razonable que el resultado de la partida puede ser explicado mediante la elección de héroes en el *draft*
- La información de cada *replay* tiene una descripción de eventos en donde sale, entre otras informaciones, cada uno de los héroes escogidos y prohibidos. Se asume que esta información viene en orden, es decir, el primer héroe que sale en el archivo fue la primera elección en el *draft*. Aunque no hay confirmación de los desarrolladores sobre este punto, todo parece indicar que esto efectivamente se cumple (de acuerdo a lo desarrollado en códigos en el repositorio oficial de *Blizzard* en GitHub.

El nuevo problema recibirá como input un escenario del *draft* y entregará 1 o 0 si el equipo 1 gana o pierde de acuerdo a ese escenario. Si fuera posible hacer un modelo que entregue la probabilidad con la que se está clasificando al dato en una determinada categoría, eso podría interpretarse como la probabilidad de victoria. Ahora, imagínese la situación en donde se han hecho todos los *bans* y todos los *pick* a excepción del último héroe del equipo 2. En este caso solo falta un héroe por decidir. Una forma ingenua, pero válida de encontrar al mejor héroe es simplemente probar todos las elecciones válidas de héroe y elegir la que maximice la probabilidad de victoria. De esta manera, estaríamos resolviendo el problema inicial, en apariencia inabordable, a partir de uno que sí sabemos cómo solucionar.

3 Preprocesamiento y base de datos

3.1 Construcción de la base de datos

Como se mencionó en la entrega anterior, para la construcción de la base de datos se utilizó HotsAPI. Particularmente, utilizamos el *endpoint*:

`hotsapi.net/api/v1/replays/paged`

A continuación, se hablará en detalle de cada paso realizado en un *script* de *Python 2* para poder llevar a cabo la construcción de la base de datos. Este *script* se puede encontrar en nuestro repositorio público (ver anexo), bajo la carpeta `Preprocessing/main.py`.

3.1.1 HotsAPI

Con el uso de esta API se pudo obtener cierta información básica sobre las *replays* almacenadas. Esta información, sin embargo, no es suficiente para lo que queríamos lograr (faltaba información clave del *draft* para poder hacer las predicciones). Afortunadamente, en cada respuesta de la API, se incluye un *link* para poder descargar el archivo `.StormReplay` original, desde el cual es posible extraer la información completa respecto a la partida relacionada a esa *replay*. Esto hizo que el problema aumentara de tamaño, pues cada repetición es un archivo bastante grande (alrededor de 2MB), por lo que hacer un análisis para cientos de miles de repeticiones se volvió una tarea considerablemente más difícil.

La forma de descargar las repeticiones se explica en la siguiente subsección.

3.1.2 *Amazon AWS S3 bucket*

El servicio de almacenaje que utiliza la API para guardar las replays, es *S3* provisto por *Amazon Web Services*, por lo que es necesario tener una cuenta en AWS. Además, la configuración del *bucket* es de *Requester pays for traffic*.

Para unas pocas repeticiones, se puede utilizar el servicio gratuito de AWS. De todas formas, una limitante para utilizar este servicio es que la inscripción requiere de una tarjeta de crédito.

Recomendación: si no se desea colocar los datos de una tarjeta de crédito real, existen bancos (por ejemplo, el Banco de Chile) que crean una tarjeta de crédito virtual con un saldo predefinido (por ejemplo, 1 dólar). Esta es una buena práctica para asegurarse de poder registrarse en la aplicación sin temor a que se cargue automáticamente a la tarjeta de crédito por un uso intensivo de los servicios de AWS.

3.1.3 *Heroprotocol* de *Blizzard*

Una vez que ya tenemos el archivo `.StormReplay`, es necesario extraer la información de este. Estos archivos están codificados completamente a nivel de *bytes*, por lo que es imposible descubrir el significado de estos. Afortunadamente, *Blizzard Entertainment*, la compañía creadora de *Heroes of the Storm*, tiene un proyecto público en *GitHub* llamado *Heroprotocol*, el cual es, justamente, un *parser* para este tipo de archivos, escrito en *Python 2*.

Así pues, se utilizó *Heroprotocol* extraer toda la información relevante de cada *replay*. Dentro de esta información se encuentra: héroes escogidos, héroes prohibidos, nivel en que tenía cada jugador el héroe que escogió en el momento del *draft*, tipo de partida, mapa de juego y resultado final de la partida.

Finalmente, el *script* realizado elimina el archivo `.StormReplay` para no acumular un exceso de archivos de alto peso (entre uno y dos megabytes por *replay*, un estimado de 110 GB de espacio para 60.000 repeticiones).

3.1.4 Almacenamiento

Para el almacenamiento de toda la información se utilizó *PostgresSQL*. Cada *replay* es ingresada a una tabla llamada `replay`, con toda la información mencionada anteriormente. Decidimos usar una base de datos como *postgres* en vez de simples archivos `.csv` ya que la API cuenta con más de 12 millones de *replays*, por lo que eventualmente el archivo podía volverse demasiado grande y generarnos problemas.

Dentro de nuestro repositorio público es posible encontrar el archivo `database.SQL`, con el cual se puede crear una réplica de la tabla que estamos utilizando para guardar la información. Además, hay un archivo llamado `database_to_csv.py`, el cual es un *script* que toma la totalidad de nuestra base de datos, y lo transforma a un archivo `.csv`. Además, con el uso de este *script*, se incluyó en el repositorio

el archivo `data.csv`, en el cual se encuentra la información de las más de 53.000 *replays* que logramos procesar hasta este momento.

3.2 Preprocesamiento

Con todo el proceso explicado anteriormente, fuimos capaces de poder construir nuestra propia base de datos, además de tener un mecanismo para poder ampliar constantemente esta base de datos (la API recolecta nuevas *replays* todos los días, aportadas por muchos jugadores en todo el mundo), pero, el formato en que se encuentra esta información no es apto para poder entrenar los distintos modelos que pretendemos crear.

Por esto, es que es necesario encontrar una codificación para la información que tenemos disponible, que sea entregable a nuestros modelos. Lo más tradicional, es utilizar *encodings* en forma de vectores, es por esto, que intentaremos vectorizar toda nuestra información.

Primero, buscaremos codificar los héroes escogidos y prohibidos. Para esto, construiremos cuatro vectores de largo n, donde n representa la cantidad de héroes disponibles en el juego (número que varía con el tiempo, ya que constantemente se están liberando héroes nuevos). Luego, debemos asociar cada heroe del juego a un índice particular de estos vectores.

Luego, construiremos el primer vector con unos en las posiciones de los héroes escogidos por el equipo uno. El segundo vector, con héroes prohibidos por el equipo uno. El tercer vector, con héroes escogidos por el equipo dos y, finalmente, el cuarto por héroes prohibidos por el equipo dos.

Finalmente, realizaremos el mismo proceso de asociar los héroes a un índice particular para los mapas del juego y agregaremos un último vector de largo m, donde m es la cantidad de mapas disponibles, con un uno en la posición del mapa en que se jugó la partida específica.

Vale la pena mencionar que estamos dejando de lado alguna de la información que recopilamos, como el nivel de los héroes escogidos. Esto es porque, ya que logramos juntar solo 53.000 *replays* en total (se explica en detalle el por qué en la sección dificultades). El agregar demasiada información al *encoding* iba a originar malos resultados al momento en el que el modelo tuviera que aprender, ya que la cantidad de datos disponibles no era suficiente para poder determinar victorias solo a partir una pequeña fracción del total de posibilidades.

Finalmente, para ver el código de este proceso implementado (en un `Jupyter notebook`), dirigirse al archivo `Models/Vectorization.ipynb` en nuestro repositorio público (*link* en anexos).

4 Modelos

Tal como se mencionó anteriormente, la idea es construir modelos clasificadores, en donde existen dos resultados posibles: 1 si el modelo cree que el equipo será ganador y 0 si el modelo cree que el equipo será perdedor. Con este resultado, se extrae la "confianza", o la posibilidad con que el modelo cree que este será el resultado, y este valor será interpretado como la posibilidad de ganar de un equipo. Para esto, hemos considerado la utilización de dos técnicas de *machine learning* distintas, descritas a continuación.

4.1 *support vector machines (SVM)*

Nuestro primer modelo a considerar es un SVM, para lo cual utilizamos la implementación de `scikit learn`. Para esto, utilizamos un *SVM* de tipo *SVC*, manteniendo los hiperparámetros estándar implementados por la librería, ya que, con la cantidad de datos disponible, le estaba tomando más de tres horas entrenar el modelo.

Si bien no realizamos ajuste de hiperparámetros para mejorar el rendimiento, aún así fuimos capacez de conseguir un *accuracy* de 0.5246, lo cual es bastante bueno. Ahora, vale la pena mencionar también que este *accuracy* fue calculado utilizando una parte recortada del *set* de *training*, lo cual no es una buena práctica, pero, nuevamente, debido a el tiempo que demoraba en entrenar el modelo, no fuimos capaces de volver a entrenar el modelo para hacer pruebas más robustas.

A pesar de todos los detalles mencionados anteriormente, se realizaron más pruebas del rendimiento de nuestro modelo, y los resultados fueron bastante buenos. Primero que nada, se evaluó como variaba la predicción del modelo al construir dos equipos relativamente equivalentes, dado los estándares del juego, al cambiar un heroe por otro lo más similar posible. En este caso, la predicción del modelo no cambió, lo que indica que este fue capaz de reconocer que el cambio de un heroe a otro fue mínimo, ya que ambos cumplen el mismo rol.

Luego, se probó como cambia la predicción al mantener los dos equipos constantes, y variar el mapa de juego. En este caso, se vió como el modelo cambiaba la predicción de un equipo hacia el otro dependiendo del mapa, pero jamás alejándose mucho de los valores. Esto quiere decir que el modelo es capaz de reconocer que el mapa tiene un efecto considerable en qué equipo es mejor que el otro, y como el mapa afecta la efectividad de algunos héroes y composiciones.

Finalmente, se hicieron pruebas para ver como se comportaba el modelo dependiendo del *winrate* de los héroes escogidos por el equipo. Se construyó un equipo con poco sentido, pero con héroes que tienen un *winrate* muy alto, y otro equipo *estándar*, pero con héroes con *winrate* bajo. El resultado fue que el modelo efectivamente atribuyó una mayor probabilidad de ganar a los héroes con *winrate* alto, lo que quiere decir que el modelo está aprendiendo qué héroes tienen mayor probabilidad de ganar que otros.

Para ver en más detalle el código y los resultados, revisar el archivo `Models/SVM.ipynb`, donde se puede encontrar la totalidad de este, además de algunos gráficos que facilitan la vista de los resultados.

4.2 Red neuronal

Las redes neuronales son un método de *machine learning* que se basa en la construcción de una función matemática, usualmente altamente complejas, utilizando unidades conocidas como neuronas. Cada neurona tiene valores de entrada, una función de activación, y valores de salida. Básicamente lo que esta realiza es aprender del *set* de datos para encontrar los mejores pesos para cada valor de entrada para cada una de las neuronas de la red.

Las redes se organizarán por capas, en donde la primera capa es la denominada capa de entrada y es la que recibe los valores de la base de datos. Para nuestro modelo en particular, se utilizó la matriz descrita en la sección de Preprocesamiento. La capa de salida es una clasificación que indica 1 o 0 de acuerdo a si se ganó o se perdió la partida en cuestión. La idea es que la red neuronal utilice el método *Backpropagation* para corregir los valores de los pesos asociados a cada neurona y así poder ajustarse para tomar el valor que permita que la red entregue como resultado el output correcto en la capa de salida.

Se utilizó la librería `sklearn` para hacer un `split` de la base de datos en *Training set* y *Testing set* de manera estratificada.

Para obtener las probabilidades con que nuestra red neuronal cree que ganará cada equipo, podemos utilizar una capa de salida con función de activación softmax. Esta función de activación nos ayudará a que nuestro clasificador se convierta en un predictor de qué tan probable es que gane un equipo o el otro. Es decir, exactamente lo que queríamos lograr.

5 Dificultades

En la siguiente sección se listarán las distintas dificultades con las que nos encontramos durante el desarrollo del progreso actual en que se encuentra el proyecto.

5.1 *Requester pays*

Para la construcción del *parser* el poder descargar las *replays* fue una gran dificultad, ya que significaba que debíamos crearnos y configurar una cuenta en *Amazon Web Services*, donde además estaríamos utilizando un servicio que es de pago.

De acuerdo al panel de control de AWS hay un límite de 20.000 *GET requests* gratuito mensual por cuenta, por lo que considerando que tenemos dos cuentas (una de cada uno), podíamos ejecutar un total de 40.000 *requests*, y por lo tanto obtener 40.000 *replays*.

Si bien, en un intento de aumentar esta cantidad, intentamos utilizar el paquete de *GitHub student*, una vez llegamos al límite gratuito, nos dimos cuenta que el beneficio de *Amazon Web Services* no cubría transacciones hechas a *buckets* de *S3*, por lo que nos resultó imposible el poder pasar la barrera de, aproximadamente, 53.000 *replays*. Vale la pena mencionar, que nos dimos cuenta que el beneficio no aplicaba de mala forma, cuando nos dimos cuenta que nos habían realizado cobros.

5.2 *Heroprotocol* y *replays*

Muchas de las *replays* analizadas no traían toda la información respecto a los héroes prohibidos. Si bien estos debiesen ser seis, algunas veces *heroprotocol* entregaba información de menos.

Una posibilidad para este problema es que se trate de *replays* antiguas, cuando en el *draft* efectivamente existían solo cuatro héroes a prohibir, pero esta posibilidad se descartó ya que nuestro *parser* se construyó para utilizar solo *replays* recientes, ya que nos entregarán información más veridica del estado actual del juego, además de que el número de héroes prohibidos entregado por *heroprotocol* no era siempre cuatro cuando no era seis, si no que podía ser cualquier número entre uno y seis.

Es por esto, que se decidió el poder aceptar columnas con valor `null` para los héroes prohibidos, y así no perder información de una gran cantidad de *replays*. Si a futuro se considera que la cantidad de *replays* que se tiene es lo suficientemente grande como para poder filtrar aquellas con datos incompletos, entonces se puede filtrar la base de datos antes de entrenar los modelos, pero por el momento se consideró que esta era la mejor opción.

6 Trabajo futuro

6.1 Mejoras al *parser*

6.1.1 Funcionamiento actual del *parser*

Actualmente, nuestro parser de replays tiene cuatro etapas importantes, las cuales son actividades bloqueantes, y lentas. Estas cuatro son:

- Descargar el archivo `.StormReplay`
- *Parsear* el archivo, y obtener la información relevante.
- Eliminar el archivo del disco dico (para no acumular una gran cantidad de archivos, ya que cada `.StormReplay` pesa entre 1MB y 2MB).
- Insertar los datos relevantes obtenidos a la base de datos.

Por el momento, el parser está obteniendo aproximadamente 1.000 *replays* por hora, lo cual es bastante lento, considerando que la API completa contiene más de 12 millones de *replays*. Por este motivo es que a continuación se propondrán dos métodos distintos de como este se podría mejorar para aumentar su eficiencia.

6.1.2 Primera propuesta: *threads* y *queues*

Claramente, el problema de nuestro *parser* actual es que tiene demasiadas tareas lentas, las cuales no se están ejecutando de forma continua cada una. La primera depende de la velocidad de descarga, la segunda depende de la velocidad de la CPU, mientras que la tercera y cuarta dependen de la velocidad del disco duro, por lo que facilmente estas podrían separarse, para poder ejecutarlas todas al mismo tiempo.

Nuestra primera propuesta es utilizar un *thread* para cada tarea. Para lograr una comunicación entre cada *thread*, se utilizan *queues*, donde, cada *thread* al terminar de trabajar en una *replay*, inserta esta, junto con la información necesaria para los pasos siguientes a una cola, y el *thread* del siguiente paso puede comenzar a trabajar en esta, mientras los otros threads pueden seguir ejecutando sus tareas, con sus propios queues, sin bloquearse entre sí, generando un *pipeline* de trabajo constante.

En este método, el único factor bloqueante es que la velocidad de descarga de los archivos *replay* sea más lento que los demás pasos del *parser*.

6.1.3 Segunda propuesta: múltiples *cores* de la CPU

Esta segunda propuesta, no es independiente de la primera, si no que una extensión. Se mencionó anteriormente que la velocidad de descarga puede ser un factor bloqueante. Ahora, ¿qué sucede si nuestro cuello de botella resultara ser la velocidad de la CPU?

En este caso, podríamos aprovecharnos de los múltiples *cores* que tienen las CPUs modernas, e implementar el segundo paso del *parser* en más de un *core*, disminuyendo, e incluso eliminando, el cuello de botella formado.

Utilizando estos métodos, se podría incrementar considerablemente la eficiencia de nuestro *parser*, y así poder obtener una masa mucho mayor de datos en comparación a lo que somos capaces de obtener en este momento.

6.2 Ajuste de hiperparámetros del *SVM*

Como se mencionó anteriormente, debido al gran tiempo que toma entrenar el modelo de *SVM* con la cantidad de datos que tenemos, no fue posible realizar un ajuste de hiperparámetros. Es por esto, que se propone como trabajo futuro el realizar este ajuste, donde la idea sería probar con distintos valores de *C*, *kernel* y *gamma*, como mínimo, intentando obtener el mayor *accuracy* posible.

Además de esto, el ideal sería poder probar cada modelo utilizando *cross-validation*, en vez de probar el modelo con los mismos datos utilizados para entrenarlo, o al menos, realizar un *split* del *dataset*, para que los datos con los que se entrena, y aquellos con que se prueba el modelo sean distintos.

7 Conclusiones y aprendizajes de la entrega

A continuación, se entrega un comentario general de las principales conclusiones y lecciones aprendidas luego de realizar esta entrega:

- Para modelar un problema complejo, una buena forma de partir es enfocarse solo en las restricciones más importantes e ir iterativamente agregando más y más restricciones hasta acercarse a la situación

del problema en la realidad. Para este caso, conviene ignorar en un comienzo aspectos como el mapa, el nivel de cada héroe para cada jugador, la liga o el *pool* de héroes de cada participante. Si se intentar modelar la totalidad del problema de una vez, muy posiblemente no se tenga claridad de los errores que pasen en el camino. Otra ventaja de agregar restricciones de forma iterativa es que el cambio en las predicciones del modelo otorga una intuición de cómo está afectando cada una de las restricciones que se van agregando a la decisión final del clasificador

8 Anexo

8.1 Introducción: Qué es Heroes of the Storm (HOTS)

Heroes of the Storm es un videojuego multijugador masivo en línea de Blizzard Entertainment, que se juega en formato 5 vs 5 jugadores, en donde cada uno controla a un héroe con el objetivo de destruir el Nexo enemigo (el edificio principal). El videojuego entra en a categoría de *Mobile Online Battle Arena* (MOBA), del mismo estilo que otros títulos famosos como *League of Legends*, *Dota II* o *Smite*.

Actualmente el videojuego cuenta con una comunidad de casi 6.5 millones de jugadores y torneos con más de 11 millones de dólares en premios.

8.2 El *Draft*

Dentro del modo de juego competitivo, también conocido como *ranked game*, antes de comenzar la partida, existe un proceso conocido como *draft* [1], en el cual los jugadores de ambos equipos proceden a escoger los héroes con que desarrollarán la partida.

El *draft* es un proceso de 12 pasos, detallados a continuación:

1. Equipo uno prohíbe un héroe (este héroe no puede ser escogido por ningún equipo durante esta partida).
2. Equipo dos prohíbe un héroe.
3. Equipo uno prohíbe un héroe.
4. Equipo dos prohíbe un héroe.
5. Equipo uno escoge un héroe (Un jugador del equipo utilizará este héroe durante la partida).
6. Equipo dos escoge dos héroes.
7. Equipo uno escoge dos héroes.
8. Equipo dos prohíbe un héroe.
9. Equipo uno prohíbe un héroe.
10. Equipo dos escoge dos héroes.
11. Equipo uno escoge dos héroes.
12. Equipo dos escoge un héroe.

[1] Para ver un ejemplo de *draft*, se sugiere ver hasta el minuto 4:11 del siguiente link: https://youtu.be/GuXwWtAonYI

Además de los héroes prohibidos, cada héroe no puede ser escogido más de una vez. Por lo tanto, un héroe puede estar presente solo una vez en toda la partida (no es posible que esté una vez por equipo, tampoco). Esto nos lleva a que, para cada paso de este proceso, además de tener una gran cantidad de héroes a escoger, tengamos ciertas limitantes a los héroes que podemos elegir.

Un factor importante a considerar es que un jugador no necesariamente sabe ocupar a los 82 héroes disponibles en el juego, además de que es necesario que el jugador haya comprado previamente el héroe para poder utilizarlo. Por lo tanto, cada *draft* está limitado por restricciones propias de cada jugador.

Finalmente, dentro del juego existen ciertos tipos de héroes que son necesarios para cada equipo. Dentro de la totalidad de héroes, podemos clasificarlos en cinco roles básicos:

- Tanque: personaje con mucha resistencia que inicia los combates en primera línea
- Daño: utilizado para hacer la mayor cantidad de daño en poco tiempo, útil para derribar héroes enemigos
- *Bruiser*: un híbrido entre Tanque y Daño, muy útil en el 1vs1 y en composiciones específicas
- *Healer*: el personaje que puede sanar rápidamente a los héroes aliados y así mantenerlos con vida
- Especialista: por sus habilidades únicas tienen un rol que no encaja en el resto de las categorías

En general, se considera que como mínimo un equipo debe contener un tanque, un *healer* y un daño. Si ahora consideramos todos los factores mencionados anteriormente y el hecho de que para cada paso del *draft* se tienen solo 25 segundos para escoger o prohibir un héroe, el proceso puede volverse muy complejo y con demasiadas variables para considerar todas las opciones posibles y escoger la ideal, de ahí la utilidad de un sistema que permita tomar rápidamente la mejor decisión.

9 Enlaces de interés

En esta sección se incluye referencias a otros sitios que pueden ayudar a la comprensión del problema y de lo hecho en esta entrega.

9.1 Contexto del problema

- Sitio oficial de *Heroes of the Storm*
- *Heroes of the Storm match Grubby*: video con una partida jugada por un ex jugador profesional. Hasta el minuto 4:11 ocurre el draft de la partida, se recomienda ver si se tiene problemas para entender el proceso de *draft*
- HotsAPI: API utilizada para descargar las repeticiones de partidas
- *Heroprotocol Blizzard*: Este link dirige al repositorio en GitHub de HeroProtocol. El proyecto permite obtener información de las repeticiones. El código está implementado en Python 2.

9.2 Referencias de la investigación realizada

- *Investigating the Impact of Game Features and Content on Champion Usage in League of Legends*. Fecha de consulta: 6 de diciembre de 2018.
- *Analysis of the Heroes of the Storm*. Fecha de consulta: 6 de diciembre de 2018.
- *The Well-Played MOBA: How DotA 2 and League of Legends use Dramatic Dynamics*. Fecha de consulta: 6 de diciembre de 2018.

9.3 Modelos

- **SVM:**

  - *Support vector machine* . *Wikipedia - The free encyclopedy* [en línea]. Fecha de consulta: 6 de diciembre de 2018.

  - Scikit-learn SVM [en línea].

- **Neural Network:**

  - *Artificial neuronal networks* . *Wikipedia - The free encyclopedy* [en línea]. Fecha de consulta: 6 de diciembre de 2018.

  - *Losses - Keras Documentation* . *Keras.io* [en línea]. Fecha de consulta: 6 de diciembre de 2018.

  - *Optimizers - Keras Documentation* . *Keras.io* [en línea]. Fecha de consulta: 6 de diciembre de 2018.

  - *Metrics - Keras Documentation* . *Keras.io* [en línea]. Fecha de consulta: 6 de diciembre de 2018.

  - *Developing your first neuronal network with Keras* . *MachineLearningMastery.com* [en línea]. Fecha de consulta: 6 de diciembre de 2018.

9.4 Otros

- Repositorio público con nuestro código. Aquí se incluye todo lo realizado hasta el momento. Incluye el parser de Blizzard, el código en Python 2 para el Preprocesamiento de la base de datos, así como el inicio de los primeros modelos a realizar

- *AWS service limits*. Contiene información sobre los límites de la capa gratuita de AWS. Sirve para tener un respaldo formal del límite de lo que podíamos descargar en el proyecto antes de tener que empezar a pagar por el servicio.