Predicción del resultado de partidas de ajedrez a partir de características extraídas del grafo de historial de partidas

Asier López Zorrilla
Mikel de Velasco Vázquez

Abril de 2017

1. Introducción

Este trabajo se basa en una competición planteada en la plataforma de *data science* Kaggle. El objetivo de esta competición es la predicción del resultado de partidas de ajedrez basándonos en resultados de partidas de ajedrez previamente jugadas [1], las cuales se encuentran en la base de datos Chess network dataset [2]. Es decir, para cada partida, conocemos el identificador único (ID) del jugador que fue blancas, el ID del que fue negras, el resultado de la partida (blancas ganan, tablas o negras ganan) y el mes en el que la partida fue jugada. Como la base de datos fue obtenida de campeonatos oficiales de ajedrez, algunos ID se repiten a menudo (porque muchos jugadores juegan en varios campeonatos a lo largo del tiempo), mientras que otros aparecen en una o muy pocas ocasiones. En total tenemos un historial de 65053 partidas, repartidas a lo largo de cien meses consecutivos, entre enero de 1998 y abril de 2006.

En lo que respecta a la competición de Kaggle, el mejor sistema de predicción estaba basado en ELO. Normalmente, el ELO se utiliza como una medida matemática de la calidad de un jugador en un juego o deporte textcolorredref: elo. Es ampliamente utilizada en juegos online, como el propio ajedrez, pero también en distintos juegos de habilidad o multijugador. El ELO no es más que un escalar, cuanto mayor sea, mayor será la calidad estimada del jugador. El funcionamiento de este tipo de sistemas es relativamente sencillo, si bien requiere de un gran ajuste sus parámetros y funciones para que rinda correctamente. Cuando un jugador se incorpora al juego en el que va a ser valorado, comienza con un ELO por defecto. Después comienza a jugar con jugadores que tienen un ELO similar al suyo. Cuando gana, su ELO aumenta, cuando pierde disminuye, y si empata el cambio en el ELO no será muy grande. Hay que decir, que en las primeras partidas el cambio en el ELO siempre será mayor, para facilitar que el jugador se estabilice lo más rápido posible. Después, el cambio en el ELO será mayor al derrotar a rivales con ELO mayor al del jugador, e igualmente al perder contra contrincantes de ELO menor.

Nuestro enfoque a este problema no está basado en el ELO, sino como un problema de predicción de pesos del grafo. Tal y como la hemos descrito, podemos interpretar la base de datos como un multigrafo orientado, dinámico pesado. Como el historial de partidas está repartido en cien meses, tendremos cien multigrafos distintos que forman el grafo dinámico. En cada uno de estos los nodos serán los jugadores que disputaron algún encuentro en ese mes. Por cada partida habrá una arista orientada, el origen será el jugador que fue blancas, y el destino el que fue negras. El peso de dichas aristas representa el resultado de la partida, si es -1 ganaron negras, 0 indica empate, y 1 victoria de las blancas. Hablamos de multigrafos porque en varios casos ocurre que dos jugadores se enfrentaron más de una vez en el mismo instante de tiempo.

[1] Chess ratings - Elo versus the Rest of the World, Kaggle, https://www.kaggle.com/c/chess#description
[2] Network analysis of Chess - KONECT, http://konect.uni-koblenz.de/networks/chess

1.1. Problema de clasificación

Una vez descrita la base de datos con la que vamos a trabajar y su propósito original, ahora vamos a definir y concretar nuestro problema y metodología particular. Al igual que en la competición de Kaggle, nuestro objetivo será la predicción del resultado de las mencionadas partidas de ajedrez. Como hemos dicho, esto es equivalente a un problema de predicción de los pesos del multigrafo. Un esbozo de nuestra metodología es el siguiente: para predecir el resultado de la partida entre el nodo i y el nodo j, construiremos uno (en realidad varios) clasificadores que realicen esta predicción a partir de distintas características de los dos nodos en instantes anteriores.

Siendo este nuestro problema, se abren varios frentes. En primer lugar hemos de crear, de alguna forma uno o varios corpus de entrenamiento a partir de la base de datos de la competición de Kaggle. Las variables predictoras serán características de dos nodos, y la clase uno de los tres posibles resultados de un encuentro de ajedrez. Por tanto debemos plantearnos, primero cómo transformar la base de datos original, y segundo qué características de los nodos utilizar. Discutiremos estas dos cuestiones en la Sección 2.

Cuando tengamos la matriz de características preparada tendremos analizar como conseguir predecir de la forma más precisa posible el resultado de las partidas de ajedrez. Probaremos con una gran cantidad de clasificadores: redes bayesianas, *K Nearest Neighbours* (KNNs), *Support Vector Machines* (SVMs), redes neuronales (DNNs), y árboles de decisión. Discutiremos todos estos clasificadores y la configuración de cada uno de ellos que hemos utilizado en la Sección 3. Finalmente, compararemos el rendimiento de todos ellos en la Sección 4, y finalizaremos con una discusión de los resultados y distintas conclusiones en la Sección 5.

2. Generación de la matriz de características

2.1. Consideraciones para generar los ejemplos de entrenamiento

La primera cuestión que analizaremos en esta sección será cómo organizar los múltiples grafos con los que contamos para conseguir un número grande de ejemplos de entrenamiento, y a la vez que estos tengan bastante información temporal. En este punto se nos plantean bastantes posibilidades. Ya que disponemos de la información temporal necesaria para ordenar los grafos, parece razonable utilizarla de alguna forma para favorecer nuestra predicción. Por ejemplo, para predecir una partida en el instante t entre los nodos i y j, podemos no solo utilizar características correspondientes a esos nodos en el instante $t-1$, sino también la correspondiente a los instantes $t-2$, $t-3$, etc. Aún así, tener en cuenta múltiples instantes hacia atrás tiene también un inconveniente principal. Como exigiremos tener características sobre los nodos i y j en todos los instantes utilizados para la predicción, aumentar el número de instantes disminuye necesariamente el número de ejemplos que podemos conseguir para entrenar los clasificadores, ya que será cada vez más probable que el nodo i o el j no estén en alguno de los grafos.

Una técnica que hemos utilizado para evitar este último problema ha sido combinar grafos para conseguir mayor cantidad de nodos en cada uno de los grafos resultantes. Por ejemplo, podemos pasar de tener grafos correspondientes a periodos mensuales a que el periodo sea bimestral, trimestral, cuatrimestral, etc. A continuación, en la Figura 1, se muestran la cantidad de ejemplos de entrenamiento que se pueden conseguir, en función de cuantos instantes hacia atrás miremos para predecir, y de cuantos meses agrupemos para conseguir más nodos por grafo.

Teniendo en cuenta este resultado, nos hemos decantado primero por utilizar grafos correspondientes a periodos anuales (es decir, hemos unido los grafos de doce en doce), y segundo por tener en cuenta información sobre los tres años previos para realizar predicciones. Así, se consiguen alrededor de 13000 ejemplos para entrenar y comparar los clasificadores.

2.2. Características utilizadas

Hasta ahora hemos comentado que vamos a realizar predicciones sobre las partidas en el instante t entre los nodos i y j a partir de características de ambos nodos en los grafos anteriores, pero no hemos especificado cuales

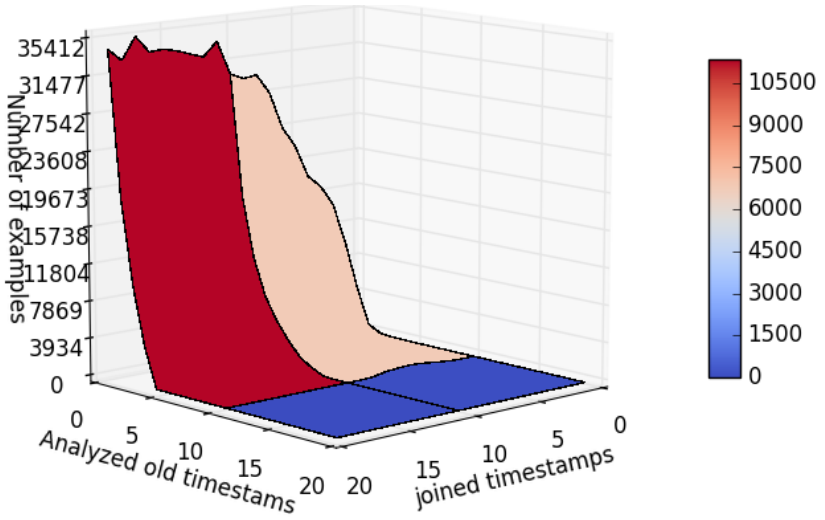

Figura 1: Número de ejemplos de entrenamiento posibles combinando los grafos en distintos periodos, y teniendo en cuenta distinto número de periodos hacia atrás.

serán éstas. En esta sección profundizaremos en las distintas características con las que hemos experimentado, y también sobre como hemos tratado con el hecho de que los grafos sean dirigidos, pesados y que además tengan en algunas aristas paralelas.

Lo primero que tenemos que decir es que todas las características utilizadas son medidas correspondientes a nodos, por tanto, no utilizamos ninguna medida sobre aristas o el grafo en su totalidad. Además hay que recalcar que no todas las características que hemos utilizado funcionan directamente para multigrafos orientados y pesados. En función de cómo de restrictivas sean las características, hemos empleados distintas técnicas para adaptar nuestros grafos a ellas. Más específicamente, si una característica funcionara con grafos dirigidos y pesados, pero no si el grafo contiene aristas paralelas, dejaremos tan solo una de las paralelas. Después, si una característica no funcionase ni para aristas paralelas ni para grafos pesados, separaremos el grafo en tres grafos dirigidos distintos, uno para las partidas en las que las blancas ganaron, otro para las partidas empatadas, y otro para la que las blancas perdieron. Además, en varios casos en las que las medidas si son aplicables a multigrafos pesados dirigidos, también dividiremos los grafos en los tres grafos recientemente mencionados, ya que consideramos que esta división puede aportar más información.

A continuación se describen brevemente todas las características utilizadas y a qué grafos han sido aplicadas para conseguir el corpus de entrenamiento final.

- Centralidad *closeness*. La closeness es una medida de centralidad de un nodo, que se calcula como el inverso de la suma de los caminos más cortos entre el nodo y el resto de nodos. [3] Esta medida no sirve para grafos pesados, así que se la aplicamos al grafo completo sin tener en cuenta los pesos, y a los tres subgrafos (blancas ganan, empatan o pierden).

- Centralidad *betweenness*. La betweenness es otra medida de centralidad que se define como la fracción de todos los caminos más cortos entre dos nodos que pasan por el nodo [4]. Igual que con la closeness, se ha calculado la betweenness del grafo completo sin pesos, y de los tres subgrafos.

[3] Freeman, L.C., 1979. Centrality in networks: I. Conceptual clarification. *Social Networks 1*.
[4] A Faster Algorithm for Betweenness Centrality. Ulrik Brandes, *Journal of Mathematical Sociology* 25(2):163-177, 2001

- *In degree*. El in degree es la cuenta de cuantos arcos entran a un nodo dado. Esta medida es fácilmente extensible para grafos con pesos, basta con sumar el peso de cada arco entrante. Por tanto hemos aplicado esta medida (en sus dos versiones) al grafo completo, y también su versión no pesada a los tres subgrafos.
- *Out degree*. La única diferencia entre el in degree y el out degree es que en el último se contabilizan los arcos (o los pesos) salientes. Esta medida se ha aplicado a los mismos grafos que el in degree.
- *Degree*. El degree no es más que la suma entre el in degree y el out degree, luego se ha computado para los mismos casos.
- *Average neighbour degree*. Con la idea de que un nodo será importante si sus vecinos son importantes, podemos calcular no solo el degree de cada nodo, sino también la media de degree de sus vecinos. Se aplicará esta medida en los mismos casos que el resto de degrees.
- *Eigenvector centrality*. La eigenvector centrality se puede entender como una extensión del degree, pero en este caso no todos los arcos cuentan lo mismo. Intuitivamente, la puntuación de cada arco depende de la importancia del nodo del que proviene, de forma que un nodo se considerará importante si está conectado a nodos importantes. El algoritmo se detalla más específicamente en [5]. Se ha aplicado esta medida al grafo completo (quitando las aristas paralelas) teniendo y sin tener en cuenta los pesos. No se ha aplicado a los subgrafos debido a errores numéricos.
- *Katz centrality*. La Katz centrality es similar a la eigenvector centrality. La influencia relativa de cada nodo se calcula a partir de sus vecinos de primer orden (los nodos conectados directamente) y del resto de nodos del grafo conectados al nodo en cuestión mediante los vecinos de primer orden [6]. Se ha aplicado al grafo completo (quitando las aristas paralelas) teniendo y sin tener en cuenta los pesos, y también a los tres subgrafos.
- *Load centrality*. La load centrality es una versión de la ya mencionada betweenness, la cual hemos aplicado al grafo completo no pesado, y a los tres subgrafos [7].
- *Harmonic centrality*. La centralidad armónica de un nodo dado se calcula como la suma de los inversos de los caminos más cortos entre el nodo y el resto de nodos [8]. Difiere de la closeness porque primero se realizan los inversos y después la suma. Se ha aplicado al grafo completo (sin tener en cuenta los pesos) y a los tres subgrafos.
- *Número de triángulos*. Esta medida de un nodo está más relacionada con el clustering que con la centralidad. Se trata de calcular la fracción de triángulos del grafo (olvidándonos del sentido de los arcos) que incluyen al nodo.
- *Número de cliques*. Para realizar predicciones también hemos contado el número de cliques [9] en los que un nodo está involucrado. Los cliques se han computado sobre el grafo sin aristas paralelas y sin tener en cuenta ni los pesos ni el sentido de los arcos.
- *Core number*. Un k-core es el subgrafo máximo que solo contiene nodos de grado de k o mayores cuando se retiran del grafo los nodos que tenían grado k o menor. El core number de un nodo indica el máximo valor de k de los k-cores que contienen al nodo. [10]. Se ha aplicado esta medida al grafo completo (sin pesos, aristas paralelas o sentidos en los arcos) y a los tres subgrafos.
- *Coeficiente de clustering*. Para grafos no pesados, el coeficiente de clustering de un nodo es la fracción de los posibles triángulos que pasan por ese nodo que existen. [11]. Se ha aplicado esta medida al grafo total (sin pesos) y a los tres subgrafos.

[5] The Structure of American Economy, 1919-1929. *Harvard University Press*, 1941

[6] M. Newman, Networks: An Introduction. *Oxford University Press*, USA, 2010, p. 720.

[7] Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, M. E. J. Newman, *Phys. Rev.* E 64, 016132 (2001).

[8] Boldi, Paolo, and Sebastiano Vigna. Axioms for centrality. *Internet Mathematics* 10.3-4 (2014): 222-262.

[9] Clique (Graph Theory), colaboradores de Wikipedia, https://en.wikipedia.org/wiki/Clique_(graph_theory)

[10] An O(m) Algorithm for Cores Decomposition of Networks Vladimir Batagelj and Matjaz Zaversnik, 2003

[11] Generalizations of the clustering coefficient to weighted complex networks by J. Saramäki, M. Kivelä, J.-P. Onnela, K. Kaski, and J. Kertész, *Physical Review* E, 75 027105 (2007).

- *Coeficiente de clustering por cuadrados*. La última medida que calcularemos para después llevar a cabo nuestras tareas de predicción será una versión del mencionado coeficiente de clustering. La única diferencia consiste en sustituir los triángulos por cuadrados.

Con todas las mencionadas características hemos conseguido un dataset de 342 variables (las cuales han sido normalizadas linealmente entre 0 y 1) y, como hemos mencionado en la Sección 2.1, 13352 casos. Como 342 variables pueden llegar a resultar excesivas para algunos clasificadores, hemos creado un segundo corpus donde las variables predictoras son las 20 componentes más principales de un *Principal Component Analysis* (PCA). Más adelante, cuando experimentemos con los distintos clasificadores, probaremos con cada uno de ellos cómo conseguimos mejores resultados, si con las características, o con las 20 componentes más principales.

3. Clasificadores

Una vez analizado cuál es entorno dónde tenemos que construir los clasificadores, vamos a proceder a describir cada uno de los que hemos utilizado.

3.1. Descripción de los clasificadores

Naive Bayes

Los Naive Bayes son una familia de clasificadores probabilísticos que se basan en aplicar el teorema de Bayes, para lo cual se asume la independencia entre las variables predictoras. Con esta asunción de independencia, se predice la clase más probable, es decir, la que maximice el producto de cada variable dado la clase por la prioridad a priori de la clase, tal y como se muestra en la Ecuación 1.

$$\hat{y} = \underset{k \in \{1,\dots,K\}}{\operatorname{argmax}} \; p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k) \tag{1}$$

donde $\hat{y}$ es la clase predicha de las K posibles clases, y x_i representa la variable predictora i de las n variables. Como en nuestro caso las variables son continuas, no hemos podido utilizar directamente la versión estándar de Naive Bayes. En su defecto, hemos probado el Naive Bayes gaussiano, que es una generalización para tratar con datos continuos. En este caso se asume que la probabilidad de cada variable dada cada clase sigue una distribución normal (Ecuación 2). Así se modelizan las probabilidades con los típicos dos parámetros de las gaussianas, la media y desviación típica.

$$p(x = v \mid c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} \, e^{-\frac{(v-\mu_c)^2}{2\sigma_c^2}} \tag{2}$$

Máquinas de vector de soporte

Las máquinas de vector de soporte (SVM) son un conjunto de modelos de aprendizaje supervisado no probabilísticos. Se basan en encontrar la frontera (un hiperplano o conjunto de hiperplanos) que más distancie los puntos de cada clase entre sí. Experimentaremos con tres SVM distintas. En primer lugar dividiremos el espacio con funciones lineales; en segundo lugar con polinomios de tercer grado; y en tercer lugar añadiremos a la SVM de tercer grado un parámetro para controlar el número de vectores de soporte.

K vecinos más cercanos

El principio detrás de los métodos KNN (del inglés, K Nearest Neighbours) es encontrar un número dado de muestras de entrenamiento más cercanas al caso que queremos predecir. En la versión estándar del clasificador, la cual vamos a utilizar, el número de muestras se puede especificar como una constante. Por tanto de ese número, el cual denotaremos como k, dependerá en gran parte el rendimiento del algoritmo. En los experimentos buscaremos qué

k es la que mejor funciona en nuestro caso. La otra variable de este algoritmo es la distancia utilizada. Nosotros experimentaremos con la distancia de Manhattan, la euclídea, y la de Chebyshev [12].

Árboles de decisión

Los árboles de decisión son un método no paramétrico de aprendizaje supervisado usados para clasificación y regresión. Son los clasificadores automáticos más parecidos a los sistemas basados en reglas. De hecho, el algoritmo de aprendizaje de estos consiste en inferir reglas (en forma jerárquica de árbol, obviamente) de forma automática para después poder clasificar nuevos casos.

Selvas aleatorias

Las selvas aleatorias se basan en entrenar un gran número de árboles de decisión, y después promediar la salida de todos ellos para realizar la predicción final. La aleatoriedad proviene del hecho de que cada árbol es entrenado con un subconjunto distinto de las variables de entrenamiento. En la práctica, esta generalización de los árboles de decisión suele funcionar mejor, ya que es más robusto al ruido y tiene un mayor capacidad de generalización.

Hemos experimentado con dos criterios distintos para entrenar las selvas aleatorias. En primer lugar hemos utilizado el criterio de ganancia de información basada en la entropía para dividir las hojas. En segundo lugar, el criterio ha estado basado en la impureza de Gini. La impureza de Gini es una medida de cómo de a menudo un elemento aleatoriamente elegido del conjunto de ejemplos de entrenamiento se clasificaría incorrectamente si hubiera sido clasificado aleatoriamente de acuerdo con la distribución de las clases en una hoja del árbol.

Redes neuronales

Las redes neuronales (profundas) (DNN, por sus siglas en inglés) son un paradigma de programación que sirven para procesar información. Están inspiradas en el funcionamiento de las neuronas y las conexiones del cerebro. Con una cierta estructura, las redes neuronales son funciones globales, es decir, son capaces de computar cualquier función $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ [13], lo cual hace de ellas una excelente técnica de clasificación. Aún así, el *boom* de las redes neuronales no se ha dado hasta esta última década. Debido al gran número de parámetros que hay que ajustar para utilizar redes neuronales para resolver problemas complejos como el reconocimiento de patrones, los tiempos de computación para trabajar con redes neuronales no eran prácticos en la mayoría de casos. La mejora en las computadoras en estos últimos años han permitido el uso (con resultados muy buenos) de las redes neuronales para tareas como el reconocimiento de voz, procesamiento del lenguaje natural, procesamiento de imagen, etc.

Se han desarrollado redes neuronales con diversas estructuras, cada una de ellas generalmente aplicable a un tipo de problema en concreto. Aunque podríamos haber utilizado redes recurrentes, las cuales procesan eficazmente secuencias temporales, como nuestras secuencias son muy cortas (3 instantes temporales), utilizaremos redes neuronales *feedforward*. Son las redes más estándar por así decirlo, no son demasiado complejas, lo cuál hace que parezcan apropiadas para nuestro problema, ya que tampoco tenemos un corpus gigantesco, que es dónde funcionan mejor las redes neuronales más complejas.

La estructura de las redes utilizadas es la siguiente (es la configuración que mejores resultados ha reportado). La entrada consiste en las 342 variables normalizadas. Tras la capa de entrada (la cual constará de tantas neuronas como variables de entrada) se añaden dos capas ocultas, de 256 neuronas cada una. La función de activación elegida ha sido la clásica función sigmoide. La capa de salida es una capa *softmax*, para facilitar la interpretación probabilística de la red. La capa softmax consta de 3 neuronas de salida, una por cada clase. Para combatir el desbalanceo entre clases se normaliza cada salida de acuerdo a la probabilidad a priori de cada clase.

[12] Chebyshev distance, colaboradores de Wikipedia, https://en.wikipedia.org/wiki/Chebyshev_distance
[13] Michael A. Nielsen, Neural Networks and Deep Learning, *Determination Press*, 2015

Máquinas de aprendizaje extremo

Las máquinas de aprendizaje extremo o ELM (del inglés, *Extreme Learning Machine*) son un caso particular de un red neuronal feedforward con una sola capa oculta. Las ELM suelen utilizarse en entornos donde la red ha de ser entrenada o reentrenada de forma muy rápida. Para realizar este entrenamiento rápido, es necesario simplificar la red. Para ello, en las ELM solo se afinan los pesos que unen las salidas de la capa oculta con la capa de salida, mientras que la primera capa de pesos se mantiene constante (y aleatoria). Incluso con esta simplificación, las ELM suelen conservar la capacidad de generalización de las redes neuronales.

Teniendo en cuenta la simplificación que se realiza para crear una ELM a partir de una red neuronal, se puede sustituir el algoritmo de aprendizaje de descenso de gradiente + propagación hacia atrás, por una simple resolución de un sistema lineal. Aún así, como en este proyecto no estamos centrados en el tiempo de ejecución de los algoritmos, entrenaremos la ELM mediante los algoritmos de descenso de gradiente y de propagación hacia atrás, ya que resulta más sencillo una vez esta preparado el código para las redes neuronales estándar.

La estructura de la ELM será la misma a la red neuronal, pero solo se entrenará la última capa. Esto es equivalente a tener una ELM con 256 neuronas en la capa oculta, aunque nosotros realmente tenemos 2 capas ocultas, pero al ser sus pesos fijos (no se entrenan) no influye el número de capas ocultas.

3.2. Prediciendo tres clases

Todos clasificadores funcionan o son directamente extensibles a problemas multiclase como este. Por tanto, en primer lugar utilizaremos esta vía directa para compararlos y analizar su comportamiento. Aún así, también experimentaremos con dos vías alternativas para realizar las predicciones.

La primera alternativa consiste en llevar a cabo una estrategia que tenga en cuenta las probabilidades a priori de cada clase (0.53 para tablas, 0.29 para victoria de blancas, 0.18 para victoria de negras). Es bien sabido que la inmensa mayoría de clasificadores funcionan mejor cuando las clases están balanceadas, luego decidimos comenzar agrupando las clases de victorias blancas y negras, ya que más o menos la mitad de acabaron en tablas, y por tanto en la otra mitad alguien resultó vencedor. Una vez unificadas estas dos clases, primero intentaremos predecir si una partida acabó en tablas o no, y en caso negativo, quién resultó vencedor. Para ello, entrenaremos, obviamente los correspondientes dos clasificadores por separado.

La segunda alternativa que hemos seguido para desarrollar clasificadores de tres clases se basa en construir tres clasificadores distintos. Cada clasificador se especializará en clasificar entre cada una de las parejas posibles de clases. En nuestro caso, esto significa que crearemos un clasificador para decidir si una partida ha acabado en tablas o victoria de las blancas, otro para si ha acabado en tablas o victoria de las negras, y el último para decidir si las blancas o las negras han ganado. Obviamente, entrenaremos cada conjunto de clasificadores con el subconjunto correspondiente de ejemplos de entrenamiento. Después, para realizar la predicción distinguiremos entre dos posibilidades. Si dos clasificadores predicen la misma clase, esa clase será la elegida. Si por el contrario todos los clasificadores difieren, entonces asumimos no tener información suficiente como para realizar una predicción en concreto, y elegimos, ya que algo hay que elegir la clase mayoritaria. Como explicaremos en la Sección 4.1 (la siguiente sección), el rendimiento medido del clasificador no dependerá de esta última elección. Podríamos elegir aleatoriamente la clase, cualquiera de las otras dos, o cada una de la clase con una distribución de probabilidad cualquiera, y nada cambiaría.

4. Experimentación

4.1. Medida del rendimiento de los clasificadores

Una vez descritos los clasificadores que van a ser comparados, tenemos que fijar cuál va a ser nuestro criterio para valorar su rendimiento. Lo primero que tenemos que tener en cuenta es que estamos tratando con un problema

de clasificación en tres clases, que además no están balanceadas: como acabamos de decir, el 53 % de las partidas acabaron en tablas, en el 29 % ganaron blancas, mientras que en el restante 18 % las negras resultaron vencedoras. La primera opción que barajamos fue utilizar la clásica medida de la precisión. Aún así, rápidamente desechamos esta alternativa, debido a que un clasificador que prediga todo el rato la clase mayoritaria (empates) obtendría un 0.53 de precisión. Este valor es muy alto para un clasificador tan simple, por lo cual finalmente nos decantamos por otra medida, el *weighted empirical error*. Este se define tal y como se muestra en la Ecuación 3.

$$\text{weighted empirical error} = \frac{\sum_{i=1}^{C} \frac{1}{p_i} \text{aciertos}(i)}{C \cdot N} \tag{3}$$

donde i es un índice que indica cada una de las clases, C el número de clases (en nuestro caso 3), p_i la probabilidad a priori de la clase i, aciertos(i) el número de muestras correctamente clasificadas de la clase i en el conjunto de test, y N la cantidad de muestras en el conjunto de test.

El weigthted empirical error tiene características que consideramos adecuadas para la comparación de los clasificadores en este proyecto. En primer lugar, siempre que las probabilidades a priori de las clases sean estimadas sobre el conjunto de muestras de test (lo cual siempre se puede hacer), esta medida estará acotada entre 0 y 1. Será 1 cuando todas las predicciones han sido correctas, y 0 si no se acertado en ningún caso. La segunda propiedad beneficiosa es la independencia a la distribución de probabilidad con el que se haya construido un predictor aleatorio. Es decir, el weighted empirical error siempre será el mismo (y valdrá $\frac{1}{C}$) si se predicen todas las partidas como tablas o victorias blancas o negras, o si se predicen cada una de las clases de forma equiprobable, o si se predice cada una de las clases con su probabilidad a priori, etc. Finalmente, se premian los aciertos sobre las clases minoritarias y los aciertos sobre las clases más probables tienen un impacto menor, siempre de acuerdo a las probabilidades a priori de cada clase.

4.2. Comparación entre los clasificadores

A continuación presentamos los resultados obtenidos con los clasificadores descritos en la Sección 3.1 y con las tres metodologías para la predicción multiclase introducidas en la Sección 3.2. Las características han sido extraídas de los grafos con la ayuda de la librería para Python NetworkX [14], y los clasificadores han sido implementados mediante las librerías scikit-learn [15] y Keras [16], también en Python.

Para evaluar justamente todos los experimentos hemos realizado un validación cruzada de 10 iteraciones. En la Figura 2 se muestran los resultados obtenidos en la validación cruzada para todos los clasificadores prediciendo directamente cada una de las 3 posibles clases. La primera reseña es general, ningún clasificador predice de forma del todo satisfactoria el resultado de las partidas de ajedrez en cuestión. La razón principal es clara, creemos que existe muy poca correlación entre las clases y las variables, sean las 342 variables originales o las 20 componentes más principales. En relación con estas dos posibles opciones hemos de decir que algunos clasificadores (árboles de decisión, selvas aleatorias, redes neuronales y ELM) funcionaron mejor con las 342 variables originales y otros (SVM y kNN) con las 20 componentes principales, en la Figura 2 se muestran los mejores casos.

El mejor clasificador ha resultado ser el Naive Bayes, seguido del ELM y de las redes neuronales, con las salidas normalizadas. Después están el kNN (el mejor caso se ha dado con k = 7 y utilizando la distancia euclídea), las selvas aleatorias, el árbol de decisión, los SVM, y finalmente, la red neuronal y la ELM sin aplicar la normalización de las probabilidades a priori a la salida. Respecto a los dos criterios utilizados en las selvas aleatorias, tanto la ganancia de información basada en la entropía como la impureza de Gini han rendido de forma similar, siendo el segundo algo más preciso. El mejor caso del kNN se ha dado con k = 7 y utilizando la distancia euclídea. En el terreno de las SVM, la lineal ha funcionado mejor que la que utiliza el polinomio de tercer grado, que ni al aplicar a esta última el

[14] NetworkX: Python software for complex networks, https://networkx.github.io/
[15] scikit-learn: Machine Leaning in Python, http://scikit-learn.org/stable/
[16] Keras: Deep Learning library for Theano and TensorFlow, https://keras.io/

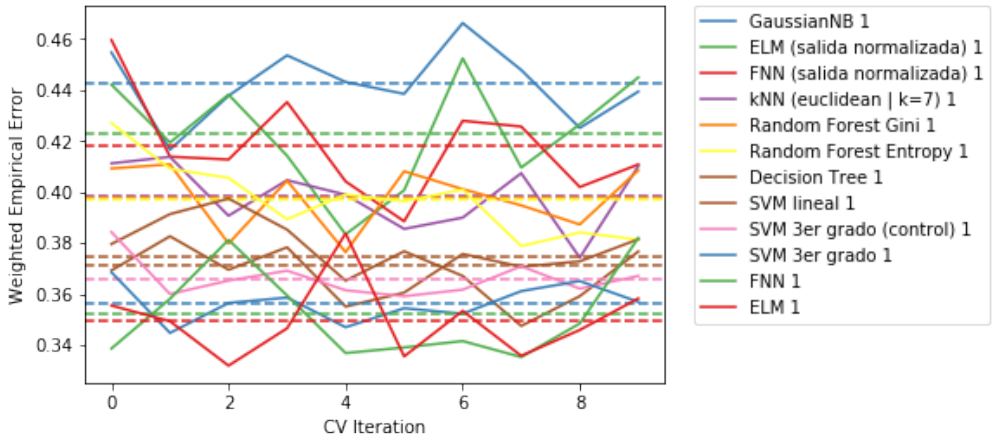

Figura 2: Weighted empirical error para todos los clasificadores e iteraciones de la validación cruzada, prediciendo cada uno de ellos cada clase directamente. También se muestran los valores medios sobre las 10 iteraciones.

mecanismo de control del número de vectores de soporte, ha conseguido mejorar el rendimineto de la SVM lineal. Ha quedado probado que, para esta aplicación, normalizar la salida de tanto la red neuronal como la de la ELM de acuerdo a las probabilidades a priori de cada clase ha sido altamente beneficioso.

En la Figura 3 se pueden observar los resultados para la segunda estrategia de predicción de tres clases. Lo más llamativo en este caso es la menor varianza entre el rendimiento de los clasificadores, los mejores no son tan buenos esta vez, y los peores han mejorado. Aún así, el Naive Bayes gaussiano sigue siendo el mejor.

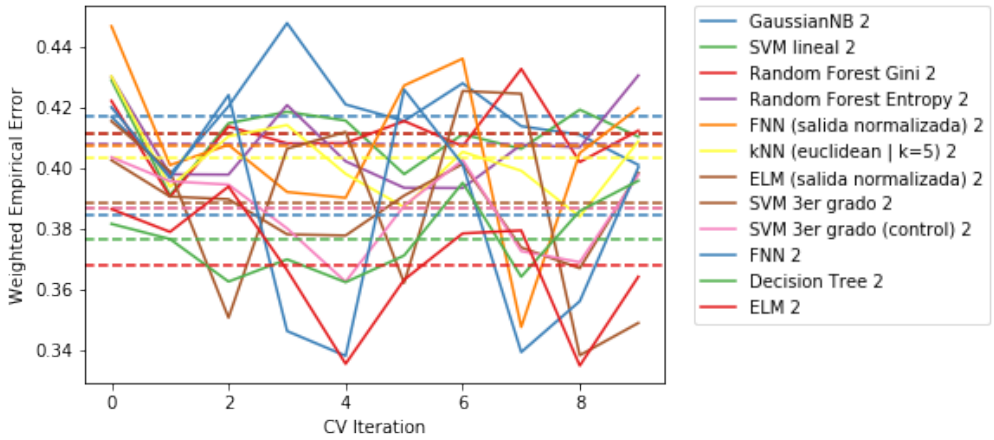

Figura 3: Weighted empirical error para todos los clasificadores e iteraciones de la validación cruzada, prediciendo primero si la partida ha acabado en tablas o no, y en caso negativo quién ha resultado vencedor.

Finalmente, la Figura 4 muestra el weighted empirical error para los distintos clasificadores, habiendo seguido la tercera técnica de predicción de tres clases (predecir la case más probable por cada pareja de clases posible). Es curioso que el mejor caso (que sigue siendo el Naive Bayes) en este caso es exactamente igual a la primera estrategia de clasificación, debido a la cómo se realizan las predicciones de forma probabilística en estos clasificadores. La clase más probable siempre será más probable, independientemente de con qué clase se compare. También es interesante observar que con esta forma de predicción mejora considerablemente el rendimiento de las redes

neuronales y ELM, mientras que el resto de clasificadores se mantienen sin grandes cambios.

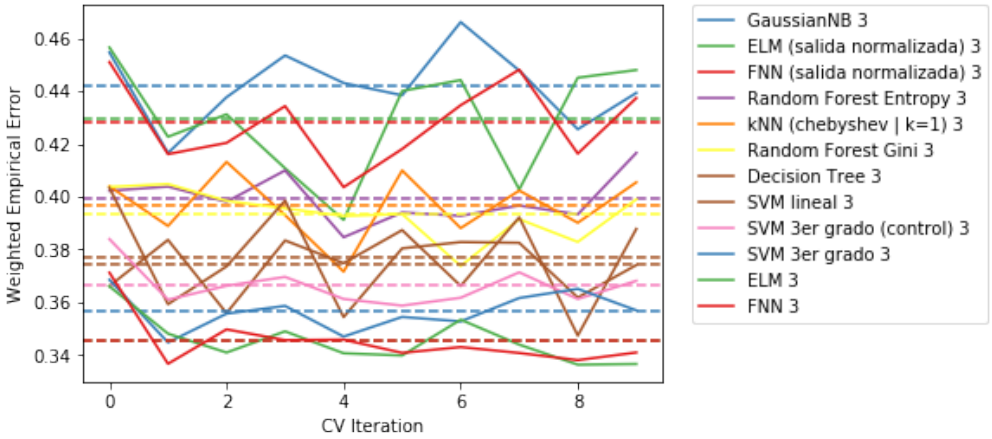

Figura 4: Weighted empirical error para todos los clasificadores e iteraciones de la validación cruzada, prediciendo la clase más probable por cada pareja de clases posible.

5. Conclusiones

En este proyecto se ha presentado una alternativa a los sistemas de ránking basados en ELO mediante el uso de características extraídas en grafos. Se ha trabajado con una gran cantidad de características, y también procesado la evolución temporal de estas para predecir el resultado de partidas de ajedrez, teniendo el cuenta un historial de partidas de varios años atrás. Se han realizado predicciones a partir de las mencionadas características mediante los clasificadores más clásicos como actuales. Como una partida de ajedrez puede acabar en victoria de las blancas, victoria de las negras o tablas, nos hemos enfrentado a un problema multiclase. Para tratar con él, hemos implementado tres estrategias: una predicción directa de alguna de las tres clases; predecir primero si la partida ha finalizado en tablas, y en caso contrario predecir quién ha sido el vencedor; y finalmente agrupar las clases por parejas, predecir en cada una de las parejas cual es la clase más probable, y de esa predicción realizar la predicción final.

Teniendo en cuenta el weighted empirical error, hemos comparado lo más justamente posible los distintos clasificadores y técnicas de clasificación dándoles distintas importancias a cada clase, se premian más las clases minoritarias, siempre de acuerdo a la probabilidad a priori de cada clase. En general, el clasificador Naive Bayes (en su versión generalizada para datos continuos) ha sido el mejor, independientemente de la metodología de clasificación. Los segundos clasificadores han sido las ELM y redes neuronales, con resultados similares. En cuanto a las técnicas de clasificación para elegir entre tres clases, la más adecuada ha sido la tercera, es decir, predecir la clase más probable entre las tres posibles parejas de clases, y elegir la clase elegida más veces como más probable. Aunque el Naive Bayes funciona exactamente igual utilizándolo con esta metodología o prediciendo una de las tres clases directamente, el resto ha visto su rendimiento mejorado con esta técnica alternativa para la predicción multiclase.

En cualquier caso, la alternativa planteada en este proyecto a los sistemas de ELO no ha resultado, en general, altamente satisfactoria. El weighted empirical error no ha superado extensamente el nivel de 0.33, que es el valor obtenido si se predicen de cualquier forma aleatoria los resultados. Este déficit de rendimiento puede deberse a que la metodología seguida puede que no sea capaz de sacar partido a todas las dependencias temporales de las que dependemos en nuestra base de datos (hemos de unificar varios grafos, y no cogemos todo el contexto disponible). También es posible que las características de los nodos no contengan información suficiente como para realizar

predicciones fiables. Aún así, siendo estrictos, hay que decir predecir resultados de partidas es una tarea muy difícil, y que tampoco disponemos de ningún sistema basado en ELO para afirmar que nuestra metodología es peor. Así, queda pendiente la comparación de nuestro sistema con uno basado en ELO, y también un análisis más exhaustivo de qué características de los grafos son las más relevantes y eficaces para realizar este tipo de predicciones.