Instituto Tecnológico y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Matemáticas y Física

Maestría en Ciencia de Datos

Inteligencia artificial auxiliar en el aprendizaje de aperturas de ajedrez

---

TRABAJO RECEPCIONAL que para obtener el **GRADO** de
Maestro en Ciencia de Datos

Presenta:
Alejandro Noel Hernández Gutiérrez

Director:
Dr. Juan Francisco Muñoz Elguezabal

Tlaquepaque, Jalisco, 18 de abril de 2024

Inteligencia artificial auxiliar en el aprendizaje de aperturas de ajedrez

Alejandro Noel Hernández Gutiérrez

Abstract

Reading chess books can be overwhelming, especially for beginner players, and the existing web resources are often paid tools or designed for passive study in which there are multiple options (there are more than 1,327 variants of openings) and its analysis is difficult and exhaustive. From a positional chess perspective, these opening explorers lack quantitative information about the quality of the position in each move and the probability of victory for each player (counts and percentages are shown based on history, but not positional, speaking clearly of these opening explorers). This is where this work becomes important, since it aims to extract quantitative characteristics of the positional advantage achieved in each movement, as well as classify victory or defeat of the current position and future scenarios or positions thus constituting an artificial intelligence useful for practicing openings and understanding the advantages of each position during the game.

Tabla de Contenidos

Índice de figuras

Página

Índice de tablas

Página

Dedico este texto a mi niño interior, que está orgulloso de quién soy. A mis padres que se partieron la madre eligiendo siempre la honestidad y el trabajo duro. A mis hermanos pilar de mi vida. A los amigos que hice en este proceso y a mi novia que sufrió conmigo días díficiles y me apoyó en todo momento.

1 Introducción

En este capítulo se presenta el contexto del objeto de estudio, la justificación del objeto de estudio, la definición del problema y los objetivos generales y específicos.

1.1 Contexto

El ajedrez es un juego popular y emocionante en el que *personas o máquinas* participan en enfrentamientos de dos jugadores. Una partida de ajedrez cuenta con 16 piezas que inician en una posición sobre un tablero de 8x8 casillas y se mueven a través de un tablero con reglas bien definidas, por tanto, es un juego determinista en el que el único elemento estocástico es el color que con el que cada participante jugará que comúnmente se decide por azar. Por lo tanto, cuando un rey está bajo ataque o en «Jaque» debe escapar.

Según Bobby Fischer [2] Gran Maestro estadounidense en su libro *Bobby Fischer Teaches Chess*, el objetivo del ajedrez es atacar el rey del enemigo en una forma tal que no pueda escapar de su captura. Una vez que la captura se ejecuta, el rey entra en «jaque mate» y el juego se termina.

Por consenso de la comunidad, conceptualmente una partida de ajedrez se divide en tres fases: apertura, medio juego y final. De forma sencilla, **la apertura** es el nombre que recibe una serie de movimientos en un juego y según Di Luca [3] puede constar desde un movimiento hasta alrededor de **25**.

Pero ¿qué se busca con una apertura de ajedrez? Lev Alburt[4] en su libro *Chess openings for white explained* menciona que «con blancas se busca una posición por lo menos igual; aunque es preferible mantener cierta ventaja, sin embargo, demandar una ventaja significativa usualmente no es realista. Con negras, se busca na posición igual, o si en algún punto se vuelve ligeramente peor, se busca que al menos sea una posición que se tenga idea como mantener.»

Muchas aperturas en el ajedrez se conocen como «aperturas estándar». Lo que significa que han sido jugadas y estudiadas por miles de jugadores a través de la historia. Cada apertura producida genera

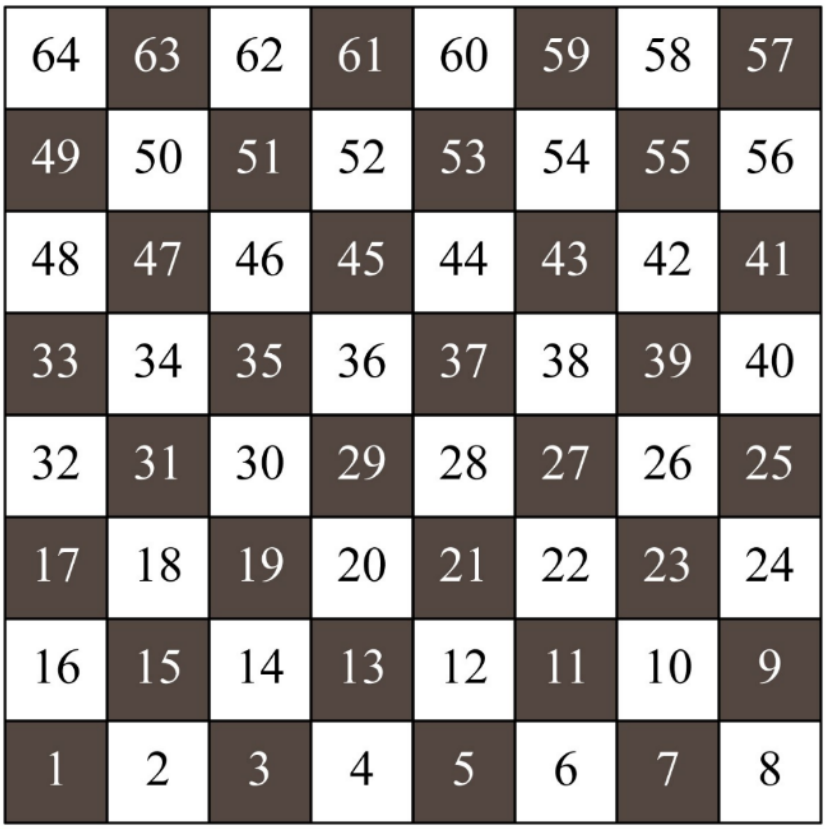

Figura 1.1: Casillas del tablero de ajedrez. Recuperado de chegg.com

cierta ventaja o equilibrio posicional entre las piezas que defienden y las que atacan al oponente.

Las aperturas son tan importantes que muchos jugadores pasan años de sus vidas simplemente estudiando aperturas [3]. Según Di Luca, en el artículo ya citado existen razones por las que las aperturas son muy importantes en el ajedrez. Algunas de ellas son:

La posición inicial del tablero. La apertura está destinada a ayudar a colocar en buena posición las piezas en el tablero y principalmente establecerlas para controlar o amenazar el centro del tablero o la mayor cantidad posible de casillas cruciales y darle forma así a la batalla que los jugadores que participan en la apertura quieren llevar.

Tomar el control del juego. Cuando un principiante se enfrenta a un jugador que conoce aperturas de ajedrez su recurso suele ser reaccionar al oponente que entiende conceptualmente los movimientos que está haciendo, sus objetivos estratégicos y posicionales. Como en la apertura participan de forma secuencial siempre dos jugadores, entre más aperturas y variantes conozcan más posibilidades tiene un jugador de controlar el tipo de juego que quiere tener.

Evitar trucos estudiados que te pueden dejar en seria desventaja.

Figura 1.2: Tablero tradicional de ajedrez. Recuperado de de chessfox

Las aperturas, como ya se estableció, son secuencias de movimientos bien estudiadas a lo largo del tiempo y a veces esconden ventajas estratégicas, si un jugador ignora los planes o ventajas de una apertura puede caer en trampas que arruinen su posición, le cuesten piezas clave o incluso perder la partida.

Prepararse para el medio juego. Después de la apertura, la siguiente etapa del ajedrez se llama medio juego y surge cuando termina de adoptarse una posición estudiada y comienzan movimientos más caóticos con el objetivo siempre de llegar a la etapa final del juego en la que se da el jaque mate. El lector puede imaginar una apertura de ajedrez como un árbol de decisión en el que cada decisión es un tablero de ajedrez y las diferentes respuestas son cada rama del árbol. Entre más se avanza en una partida más ramas y posibilidades existen por lo que una buena apertura es importante para tener opciones solidas (Figura 1.3).

Según The Oxford Companion to Chess [5], que es un catálogo de conceptos de ajedrez, hasta el año de publicación existían alrededor de 1327 aperturas o variantes de aperturas, entonces no es difícil imaginar la dificultad que enfrentan los jugadores que comienzan a jugar de

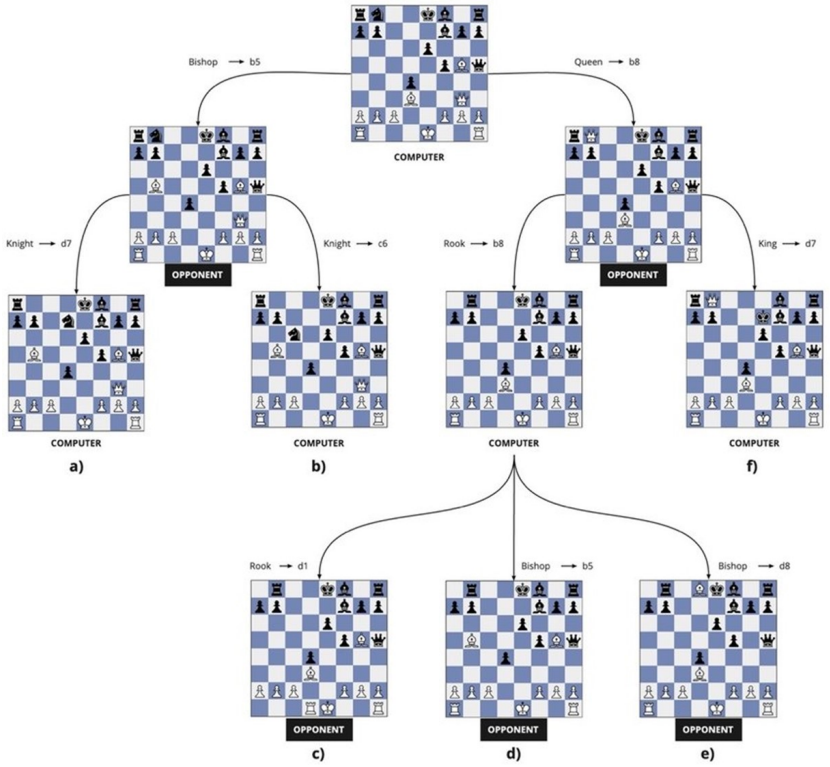

Figura 1.3: Ajedrez como árbol de decisión, ejemplo extraido de Research Gate

forma habitual y cuando llegan a cierto nivel se enfrentan con oponentes que conocen más aperturas y entienden sus ventajas.

Existen innumerables libros y recursos web para que los jugadores aprendan las aperturas. En lo que respecta a los libros, la metodología de enseñanza que siguen la gran mayoría (por nombrar algunos *El ajedrez. Aprender y progresar, de Karpov; Ajedrez lógico jugada a jugada, de Irving Chernev; Chess Openings for Black/White explained, del GM Lev Alburt; Learn Chess Tactics, de John Nunn; The complete idiot's guide to chess, de Patrick Wolff; A first book of morphy, de Frisco del Rosario; Back to basic tactics, de Dan Heisman*) consiste en mostrar una secuencia de ilustraciones comentadas en la que se presentan partidas de forma secuencial y los autores van narrando las ventajas de cada posición alcanzada. Esto suele ser abrumador para jugadores principiantes que apenas conocen las reglas o tienen poca experiencia.

Dentro de los recursos web existen algunas herramientas gratuitas o de paga entre las cuales figuran:

Chess tempo base de datos (gratuito o de paga según características). Es un explorador de aperturas en el que el jugador puede ver el número de veces que se ha juga un movimiento, el porcentaje de veces que en partidas que jugaron ese movimiento

ganaron blancas o negras, el nombre de las aperturas relacionadas entre otras cosas[6](Figura 1.4). **Chess position trainer** (de paga) es

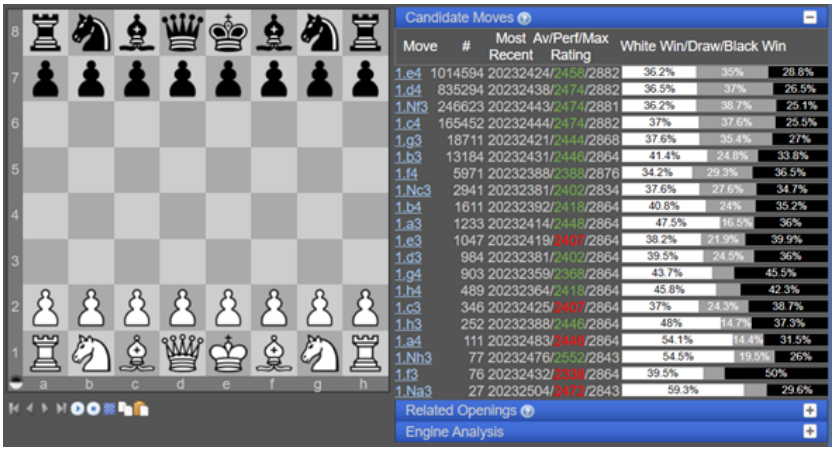

Figura 1.4: Explorador de aperturas Chesstempo

un software de escritorio que permite acceder a una base de datos de aperturas y entrenarlas una vez que fueron escogidas [7]. **Chessbase Reader** 2017 (de paga) es un lector de partidas de ajedrez que muestra bases de datos de partidas y puede leer múltiples tipos de archivo (.cbh, .cbf, .pgn), usualmente usado por grandes maestros para sus análisis [8] (Figura **??**). **Chess Explorer**, de la plataforma online chess.com (de paga) Es un explorador de partidas de grandes maestros similar al ofrecido por chesstempo, pero que es actualizado en todo momento con partidas de grandes maestros en la misma plataforma. Para acceder a este explorador hay que pagar una membresía mensual [9](Figura **??**). **365 Chess Opening explorer**, similar a los exploradores de chesstempo y chess.com. Es posible explorar aperturas iniciando con una posición FEN o desde la posición inicial [10].

1.2 *Justificación*

Establecidas las condiciones descritas en la introducción, es posible entender de forma general la importancia de las aperturas en el ajedrez y las opciones que tienen los jugadores para aprenderlas, pero ¿a qué razón científica o social busca contribuir el presente documento? ¿en qué se centrará este trabajo de obtención de grado?

Como ya fue establecido, la lectura de libros de ajedrez puede resultar abrumadora sobre todo para jugadores principiantes y los recursos web existentes son herramientas en muchas ocasiones de paga o diseñadas para un estudio pasivo en el que se cuenta con múltiples opciones (ya se mencionó que existen más de 1327 variantes de aperturas) y es difícil y exhaustivo elegir las adecuadas y estudiarlas de forma correcta.

Desde una perspectiva de ajedrez posicional, estos exploradores de apertura carecen de información cuantitativa sobre la calidad de la posición en cada movimiento y la probabilidad de victoria para cada jugador en la posición y qué tan buena es a futuro (se muestran conteos y porcentajes basados en históricos).

Según James H. McMillan[1] es probable que los estudiantes se sientan motivados si se satisfacen sus necesidades, si ven valor en lo que están aprendiendo y si creen que pueden tener éxito con un esfuerzo razonable. Haciendo énfasis en el último punto *si creen que pueden tener éxito con un esfuerzo razonable* es posible decir que un aprendiz de ajedrez se ve abrumado con tantas variantes de apertura disponibles en estos libros y herramientas digitales, por lo que su estudio es complicado y con ello la *idea de tener éxito*. Pues bien, este modelo heurístico1.5) será tomado en cuenta para el diseño de la inteligencia artificial abordada en este trabajo de obtención de grado.

Aquí es donde este trabajo toma importancia, ya que pretende ser una inteligencia artificial en la que un usuario pueda practicar aperturas contra la máquina, y visualizar continuamente características cuantitativas de la ventaja posicional alcanzada en cada movimiento, así como un *bias* de victoria o derrota de la posición actual obtenido con un modelo de clasificación.

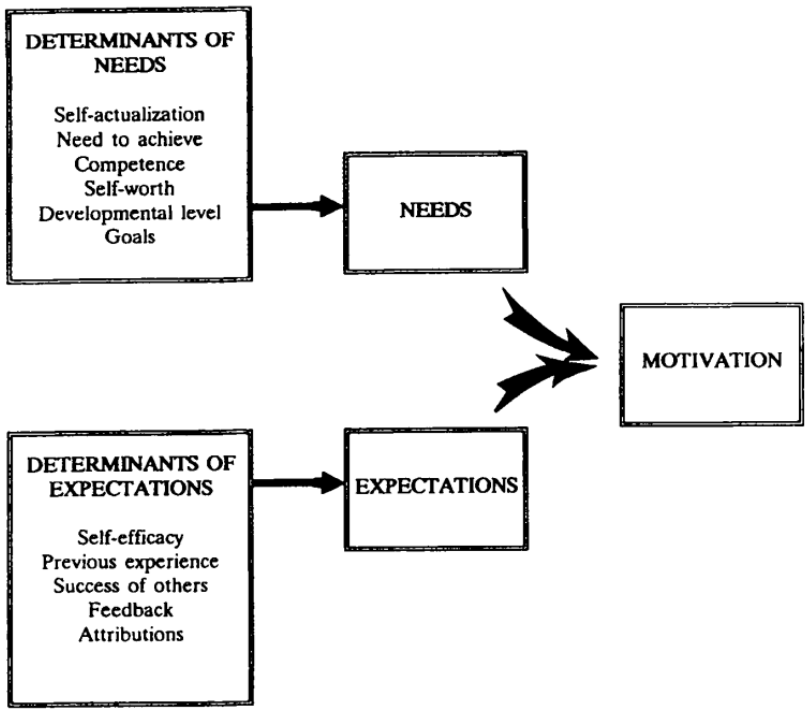

Figura 1.5: A Heuristic Model of College Students' Motivation, recuperado de [1]

1.3 *Problema*

No existe una inteligencia artificial que enseñe aperturas y su variantes con objetivos claros y retroalimentación constante, interactuando con un estudiante real mediante partidas de ajedrez usuario-máquina.

1.4 Objetivos

1.4.1 Objetivo general

Diseñar e implementar un modelo de clasificación que prediga victoria o derrota en partidas de ajedrez a partir de características posicionales a pesar de los retos de no linealidad de los datos de partidas historicas que en muchas ocasiones tienen un desarrollo caótico.

Diseñar e implementar a la par una inteligencia artificial enfocada en la fase de la apertura del ajedrez que sea capaz de responder a movimientos de un humano así como interactuar con él fungiendo la figura de entrenador brindándole datos sobre su posición y sobre las posibles posiciones que puede alcanzar.

1.4.2 Objetivos específicos

Por medio de ingeniería de características, codificar funciones que extraigan los siguientes atributos cuantificables sobre la posición:

- Casillas controladas por peones, caballos, alfiles, torres, reina y rey.
- Puntos de presión con más de una pieza atacando.
- Diagonales controladas por el jugador en turno.
- Columnas verticales y horizontales controladas por el jugador en turno.

Aplicar las funciones creadas a las 20058 observaciones de partidas de ajedrez y convertirlas en nuevas *features* de cada observación.

Entrenar una serie de modelos de aprendizaje supervisado que consideren las características extraídas de las posiciones de partidas históricas y clasifique victoria de negras o blancas entregando un porcentaje para ambas (bias).

Evaluar los distintos modelos y escoger el que destaque según las métricas estudiadas y la complejidad para implementarse en producción.

Implementar una inteligencia artificial que:

- Juegue contra el jugador una apertura y sus variantes al azar
- Utilice la base de datos de partidas históricas para realizar con la maquina movimientos que lleven a cualquier variante (*de la apertura elegida*) de forma aleatoria tomando en cuenta pesos (probabilidades) para que elija jugar en más ocasiones las aperturas más comunes y elija mover en más ocasiones los movimientos más comunes de un humano jugaría en una situación real.

- Muestre al usuario un despliegue de características posicionales en cada movimiento.
- Muestre al usuario la calidad de su posición mediante el modelo elegido.

Constituir así una inteligencia artificial didáctica y gratuita para el entrenamiento de aperturas.

2 Metodología

En este capítulo se presenta en detalle el desarrollo metodológico que incluye el análisis exploratorio, el cual abarca los siguientes aspectos:

- *Tratamiento de valores faltantes*
- *Distribución de variables numéricas*
- *Distribución de variables categóricas*
- *Promedio, mediana y desviación estándar de los niveles de jugadores*
- *Conversión a variable categórica de las columnas white_rating y black_rating*
- *Distribución de nivel de jugadores vs frecuencia de juego*
- *Aperturas más comunes por nivel y frecuencia de juego*
- *Distribución de color y tipo de victoria*

Se abarca también en la sección Ïngeniería de características", todas las características posicionales extraídas de las partidas históricas.

Por último se presenta la descripción de los modelos, métricas y experimentos realizados.

2.1 Descripción de los datos

Los datos fueron obtenidos de el siguiente dataset [11]

La tabla que se muestra a continuación detalla de manera breve cada columna de los datos:

La siguiente tabla detalla de manera técnica el conjunto de datos:

Característica	Valor
game_id	Identificador único para cada partida de ajedrez.
rated	Indica si la partida está clasificada (True/False).
turns	Número total de movimientos (1 a 349).
victory_status	Estado de la victoria (mate, resign, Out of time, draw).
winner	Ganador de la partida (white, black, draw).
time_increment	Incremento de tiempo en segundos por movimiento.
white_id	Identificación única del jugador blanco.
white_rating	Rating del jugador blanco al momento de la partida.
black_id	Identificación única del jugador negro.
black_rating	Rating del jugador negro al momento de la partida.
moves	Lista de movimientos realizados en la partida.
opening_code	Código que identifica la apertura jugada.
opening_moves	Número de movimientos iniciales de la apertura.
opening_fullname	Nombre completo de la apertura.
opening_shortname	Nombre abreviado de la apertura.
opening_response	Respuesta a la apertura por parte del oponente.
opening_variation	Variación específica de la apertura.

Tabla 2.1: Descripción conceptual del conjunto de datos de ajedrez

Nombre	Tipo	Valores faltantes	Valores únicos
game_id	Entero	no	20058
rated	Booleano	no	2
turns	Entero	no	211
victory_status	Objeto	no	4
winner	Objeto	no	3
time_increment	Objeto	no	400
white_id	Objeto	no	9438
white_rating	Entero	no	1516
black_id	Objeto	no	9331
black_rating	Entero	no	1521
moves	Objeto	no	18920
opening_code	Objeto	no	365
opening_moves	Entero	no	23
opening_fullname	Objeto	no	1477
opening_shortname	Objeto	no	128
opening_response	Objeto	18851	3
opening_variation	Objeto	5660	615

Tabla 2.2: Descripción técnica del conjunto de datos de ajedrez

2.2 *Análisis exploratorio*

2.2.1 *Tratamiento de valores faltantes*

Estas son las columnas con datos faltantes.

opening_response: 18851

opening_variation: 5660

Para el caso de opening response, se elimina la columna porque no es necesaria para el análisis y es muy poco representativa (alrededor de 93.98 % de datos faltantes).

Respecto a la columna opening_variation, representa las variantes de las aperturas más tradicionales. Si no existe nombre para una variante,

es porque esa fila representa la apertura en su variante 'tradicional', por tanto, se rellenan los valores faltantes con esta oración:

```
df_1['opening_variation'] =
    chess_data['opening_variation'].fillna('traditional opening')
df_1.info()
```

Después de esto, todas las columnas dejan de presentar valores faltantes:

#	Columna	Non-Null Count	Dtype
0	game_id	20058 non-null	int64
1	rated	20058 non-null	bool
2	turns	20058 non-null	int64
3	victory_status	20058 non-null	object
4	winner	20058 non-null	object
5	time_increment	20058 non-null	object
6	white_id	20058 non-null	object
7	white_rating	20058 non-null	int64
8	black_id	20058 non-null	object
9	black_rating	20058 non-null	int64
10	moves	20058 non-null	object
11	opening_code	20058 non-null	object
12	opening_moves	20058 non-null	int64
13	opening_fullname	20058 non-null	object
14	opening_shortname	20058 non-null	object
15	opening_variation	20058 non-null	object

Tabla 2.3: Descripción de las columnas del conjunto de datos

2.2.2 *Conversión a variable categórica de las columnas white_rating y black_rating*

Al verificar la columnas numéricas white_rating y black_rating se decide hacerlas categóricas ya que al englobarlas en cuartiles es más fácil el estudio.

Clasificaciones:

- Beginner: Abajo de percentil 25 %
- Intermediate: Entre 25 % y 50 %
- Advanced: Entre 50 % y 75 %
- Master: Arriba de 75 %

Por tanto:

- Percentil 25 %: 1394
- Percentil 50 %: 1564
- Percentil 75 %: 1788

Con esta información se crean dos categorías: white_category y black_category

Como datos adicionales el promedio de Ratings de jugadores es de 1592.732. La mediana de Ratings es de 1564. Su desviación estandar es de 291.164 y 50 % de los datos está entre 1394 y 1788 con un Skew de la distribución: 0.280

2.2.3 *Simplificación de la variable categórica time_increment por la nueva columna time_category*

Clasificaciones:

- Bullet:Entre 0 y 2 de tiempo base
- Blitz: Entre 3 y 5 de tiempo base
- Rapid: Entre 6 y 30 de tiempo base
- Classical: Entre 30 y 60 de tiempo base
- Personalizado: Todo lo demás

Con base en esta información se define la nueva columna time_category

2.2.4 *Distribución de variables categóricas*

A modo informativo y para conocer más profundamente los datos. Se muestra la siguiente visualización de las variables categóricas:

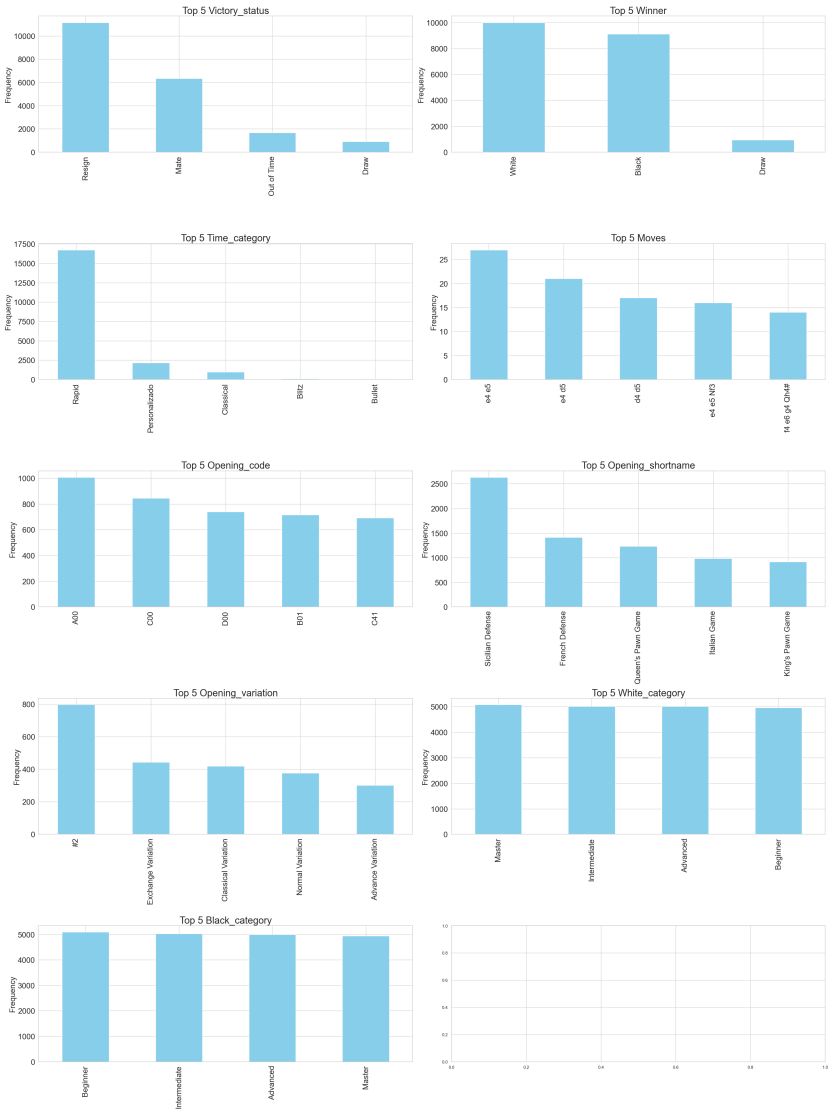

Figura 2.1: Columnas categóricas del conjunto de datos

Los gráficos nos dicen algunas cosas de forma inmediata, como que hay más victorias por resign que por mate, lo que indica que el dataset tendrá muchas partidas que se resolvieron cuando un jugador se dio cuenta que su posición era muy mala o estaba cerca el jaque mate y concedió la victoria al oponente. También es posible observar que la probabilidad de que gane blancas es ligeramente más alta que negras. La mayoría de las partidas entraron en categoría de rápida (entre 6 y 30 minutos de tiempo base). La defensa más común es la siciliana y la variante número 2. También es posible determinar que las categorías para blancas y negras se distribuyeron de forma correcta.

2.2.5 *Distribución de variables numéricas*

A modo informativo y para conocer más profundamente los datos. Se muestra la siguiente visualización de las variables numéricas: Es

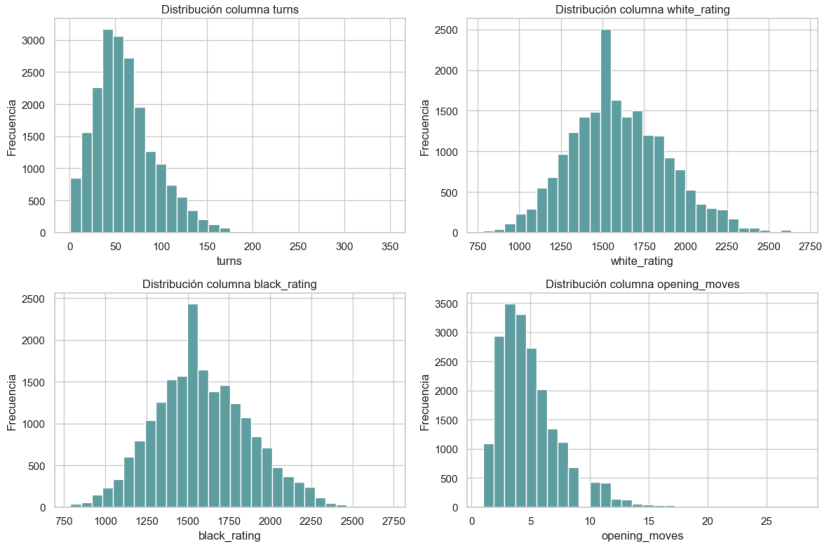

Figura 2.2: Columnas numéricas del conjunto de datos

posible observar una asimetría sobre todo en opening moves y turns. Veamos su Skewness y Kurtosis:

	Asimetría	**Curtosis**
Turnos	0.897284	1.385161
Movimientos de apertura	1.334557	3.089694

Tabla 2.4: Asimetría y Curtosis de columnas Turns y Opening Moves

La presencia de la asimetría positiva en opening_moves indica que la distribución tiene una cola más larga hacia la derecha. Este comportamiento nos dice que hay más partidas con numero de turno superior al promedio, o lo que es lo mismo, que la mayoría de las partidas se resuelve entre 25 y 75 turnos. La curtosis es moderadamente

alta, indicando que la distribución es más puntiaguda de lo normal, pero no es signo de alarma ya que refleja el comportamiento real.

La presencia de asimetría positiva en ambas variables indica un sesgo hacia partidas más largas. La curtosis elevada en opening_moves sugiere que aunque la mayoría de las partidas tienen a un patrón de más movimientos de apertura (4-8), existen casos notables que se desvían de ese patrón, esto quiere decir que hay aperturas atípicas de muchos movimientos que son poco usadas:

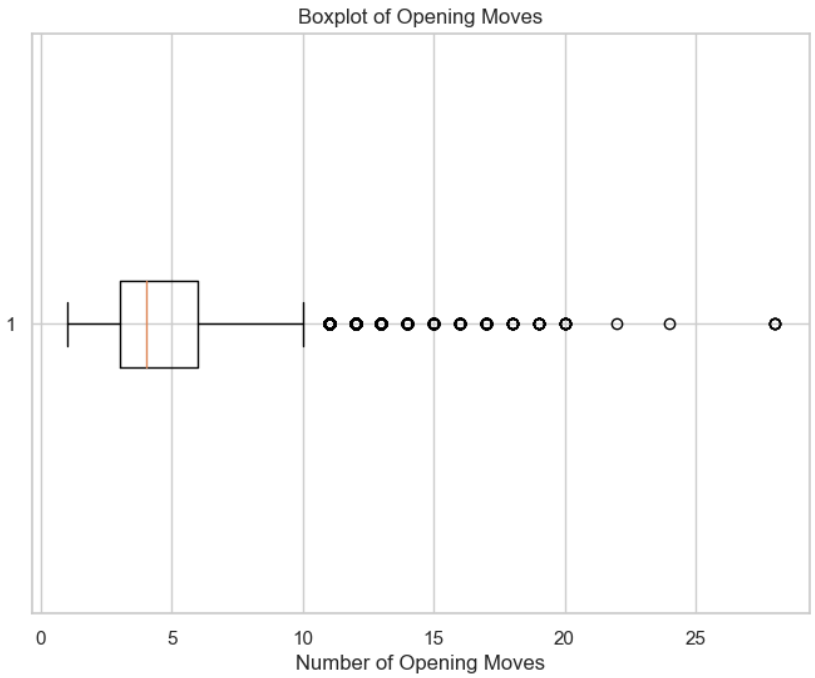

Figura 2.3: Boxplot Movimientos de apertura

2.2.6 *Distribución de nivel de jugadores vs frecuencia de juego*

En el siguiente gráfico es posible observar el elo, nivel o rating de los jugadores en este dataset, siendo 1500 predominante. Las líneas verticales discontinuas representan el percentil 25 y el 75 del conjunto combinado de rating. Estos indican que la mayoría de los jugadores tienen una rating entre estos dos valores: 1394 y 1788. Rango que es más corto que el resto de los niveles. Lo que indica que se necesite un salto de calidad para superar esta puntuación y que hay cierto estancamiento en este rango. Esta afirmación da fuerza al planteamiento de este trabajo, ya que pretende ser una herramienta didáctica para aprender aperturas de manera casi natural y reforzar el aprendizaje con datos importantes de cada posición alcanzada.

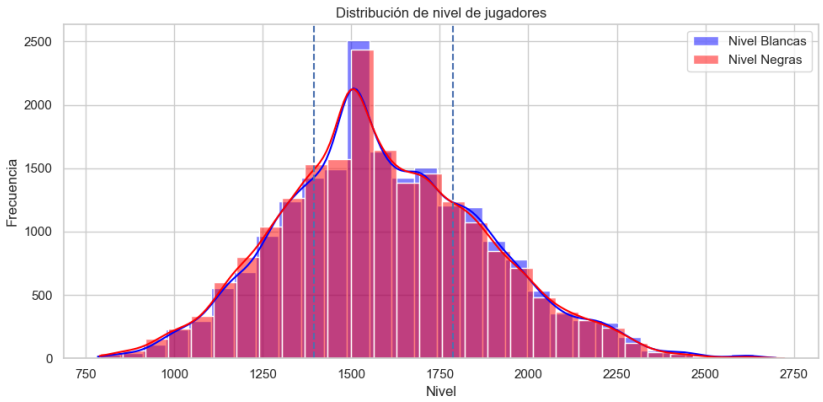

Figura 2.4: Nivel de jugador por frecuencia de partidas

2.2.7 *Distribución de color y tipo de victoria*

En el siguiente gráfico se visualiza con exactitud el balance de la clase objetivo: ganador. Hay un 49.9 % de probabilidad de victoria simplemente por el hecho de comenzar con blancas, situación que se ve más claramente en alto nivel. También se observa que un 55 % de las victorias fueron producidas por el medio de la rendición, situación que ya se mencionó, ayuda al modelo.

Para mitigar un poco el desbalance de las clases y dado que el objetivo es entender que tan ganadora o perdedora es la posición, en la etapa de entrenamiento se elimina el empate de la clasificación.

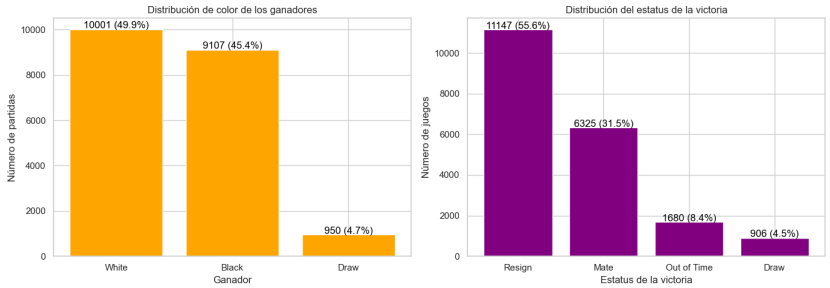

Figura 2.5: Relación a través de las partidas entre la victoria y el color

2.2.8 *Aperturas más comunes por nivel y frecuencia de juego*

Para sustentar la teoría de que los jugadores que llegan a cierto nivel necesitan aprender aperturas para dar un salto de calidad y subir, hay que explorar la frecuencia en la que se juegan las aperturas más comunes en este dataset según el rating de los jugadores en la siguiente ilustración 2.6

Sabiendo que las categorías están perfectamente balanceadas, rápidamente se puede notar que la frecuencia con que las aperturas más populares se juegan va disminuyendo, esto es porque los jugadores avanzados van conociendo y dominando más variedad de aperturas. Este gráfico da fuerza a la teoría de que para dar un salto de calidad en ajedrez hay que aprender aperturas y entender, como ya se dijo, sus ventajas y desventajas posicionales y tácticas.

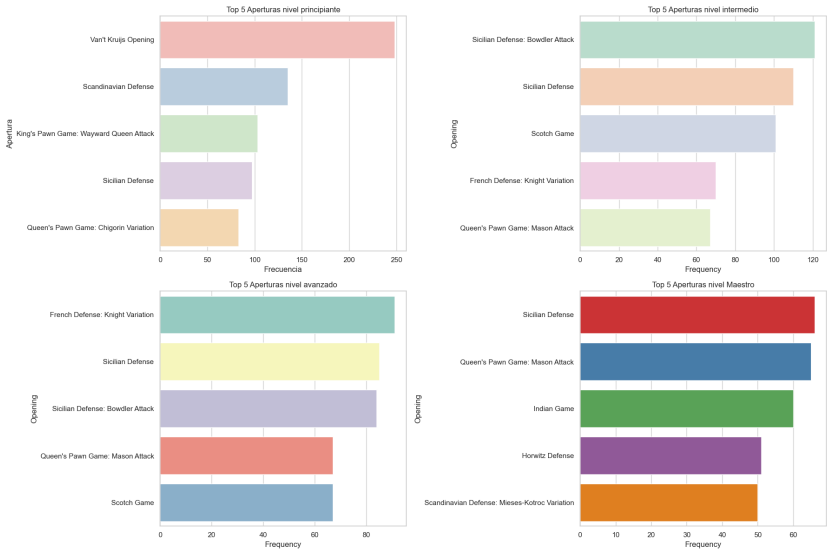

Figura 2.6: Aperturas más comunes por nivel y su frecuencia de juego

Respecto a nivel principiante, el hecho de que Van't Kruijs Opening sea la apertura más popular no indica que sea buena, para entender el problema habrá que visualizar esta apertura: Después de

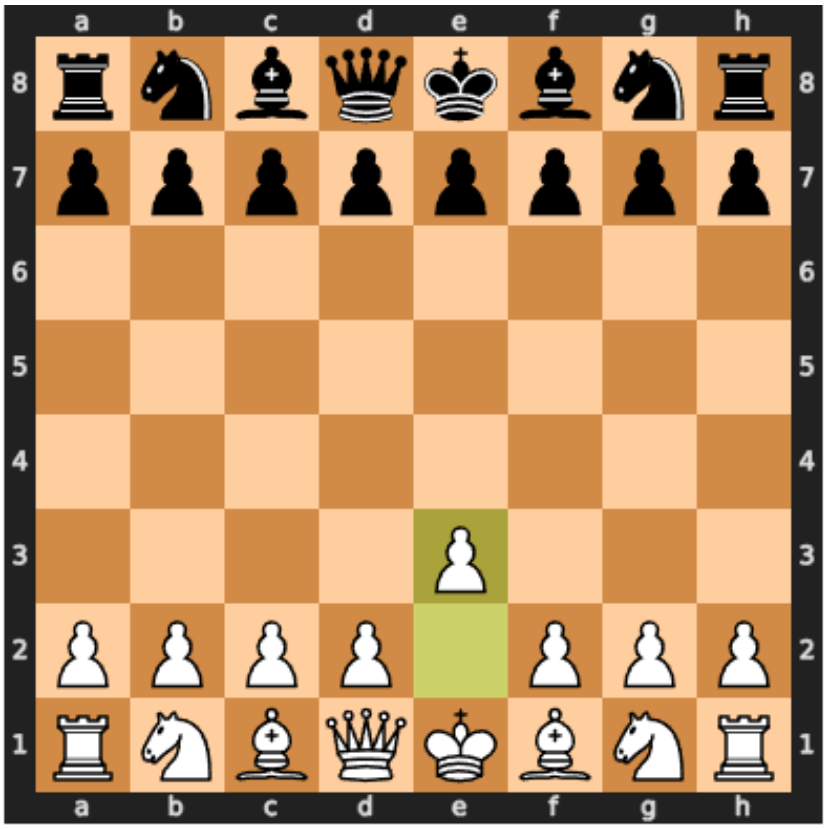

Figura 2.7: Van't Kruijs Opening

revisar el gráfico de la apertura se entiende por qué es la más común entre principiantes: los jugadores que inician en el ajedrez no conocen aperturas y juegan de forma empírica a penas conociendo las reglas. Es posible deducir esto porque habiendo tantas posiciones con nombre, esta posición es la única dentro de la secuencia de movimientos siguientes que recibió nombre en sus partidas, osea que a partir de esta posición el desarrollo del juego es caótico, con poca idea de la posición. Estos jugadores suelen aprender únicamente por memoria muscular, al perder partidas y recordar las posiciones en que se tuvo alguna desventaja. Esto no es malo, es una curva de aprendizaje, pero es allí donde el aprendizaje empírico encuentra el limite y el jugador se estanca en un nivel y se vuelve fundamental conocer las aperturas para subir.

Ahora hay que echar un vistazo a las aperturas más comunes desde el punto de vista de maestros y cómo han terminado esas partidas:

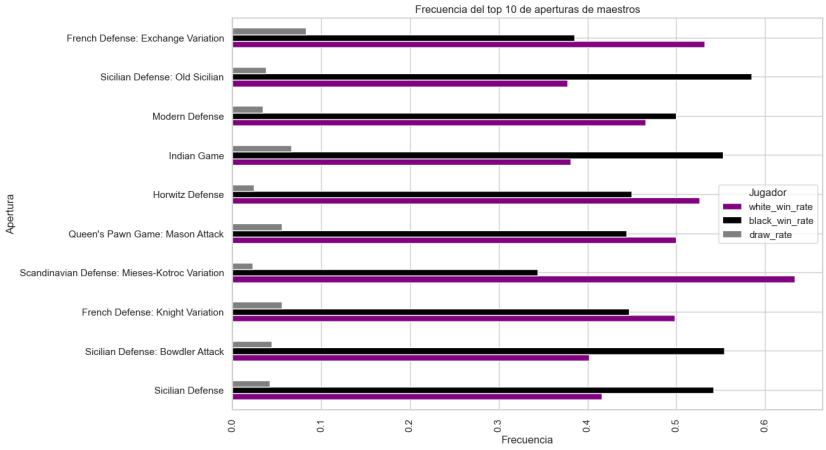

Figura 2.8: Frecuencia de juego aperturas maestros y color ganador

De este gráfico se puede concluir que:

- La Defensa Francesa, Horwitz, Peón de Reina (variante ataque Mason), así como la Defensa Escandinava (variante Mieses-Kotroc) son las defensas que mejores resultados dan para las blancas.
- Algunas variantes de la Defensa Siciliana, así como la Defensa India, son las aperturas que mejores resultados ofrecen para las negras.
- Es posible entonces decir que un buen jugador de ajedrez domina al menos estas aperturas y sus características posicionales y tácticas.

2.3 *Ingeniería de características: Descripción de los nuevas características*

Para construir un modelo que dada una posición específica clasificara victoria o derrota, era necesario extraer una serie de características que suelen considerar los grandes maestros cuando están evaluando su propia posición y la del adversario. Esta serie de características se consiguen a partir de una cadena FEN, que es una representación de la posición de todas y cada una de las piezas del tablero y se abarcará con más detalle en la siguiente sección. Con este objetivo se creo el archivo *extract_features.py* en el que se codificaron 22 funciones para extracción de características que se explicarán desde una perspectiva visual en este las siguiente secciones.

2.3.1 *Moves Fen*

La primera y muy importante característica es la representación de los movimientos secuenciales guardados en notación algebraica (SAN) como una cadena que represente la posición alcanzada y pueda ser utilizada para el estudio de la misma (FEN).

FEN, por sus siglas en inglés "Forsyth-Edwards Notation". Es un método estándar para describir posiciones de ajedrez usando texto. A través de cadenas FEN. Se pueden replicar estas posiciones en infinidad de herramientas computacionales y sitios web como chess.com.

Pero, ¿Cómo se describe una posición usando una cadena FEN? La primera cosa que se tiene que entender es que un tablero de ajedrez se puede descomponer en 8 filas, que en inglés reciben el nombre de ranks". Cada fila se convierte en una cadena de texto que la representa y esas 8 cadenas de texto se compactan juntas separadas por diagonales (/) para formar la cadena FEN[12].

Esto se puede ver claramente en la imagen 2.9:

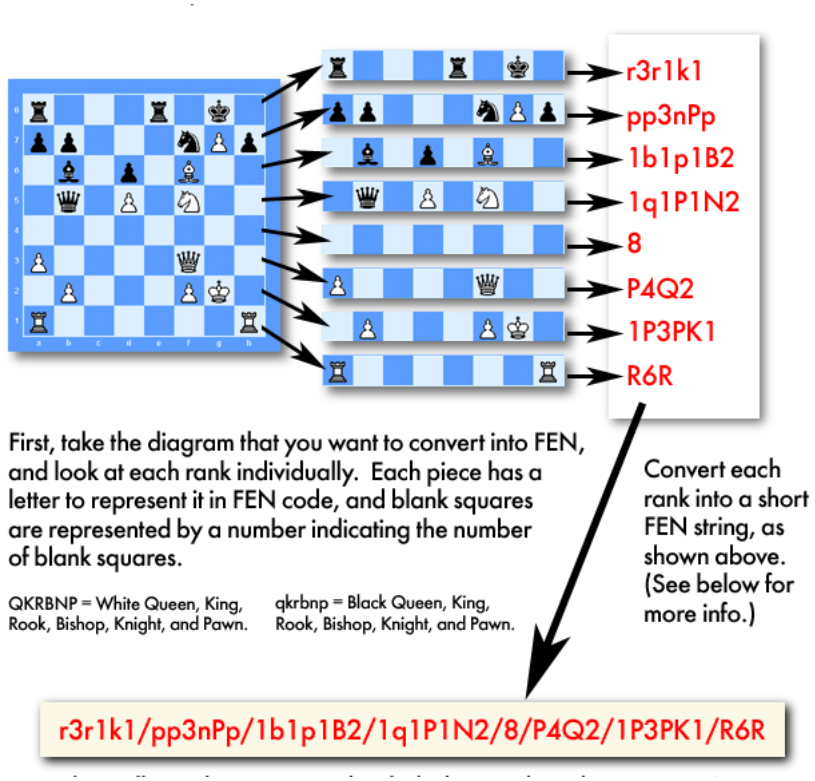

Figura 2.9: Cadena FEN, composición

Las piezas están representadas por sus iniciales en inglés, y se utiliza una letra mayúscula para Blancas y minúscula para negras 2.10

Esta cadena se extrae a partir de la columna 'moves' que representa todos los movimientos hechos en una partida, con la intención de tener

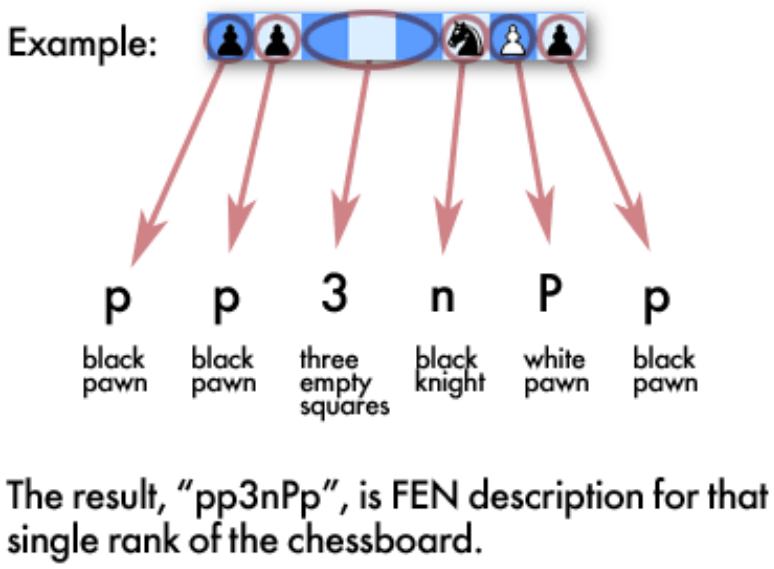

Figura 2.10: Cadena FEN, una sola fila

una representación corta de la posición para poder estudiarla.

2.3.2 *Turn*

La segunda característica que se extrae a partir de la cadena FEN es el turno, en el que sencillamente se guarda un 1 si el turno es de blancas y un 0 si el turno es de negras, esto para darle una perspectiva al modelo de quién evalúa la posición.

2.3.3 *Casillas controladas por peón*

Una de las características más importantes. Para esta y las demás características se generan dos funciones, la que alimenta el modelo que entrega un número entero y la que informa al usuario las casillas. La función que alimenta el modelo cuenta las casillas controladas por peón, esto es, las casillas en las que puede comer una pieza o bien en el siguiente turno o bien si alguna pieza enemiga se moviera a esa casilla. Esta característica es importante porque uno de los objetivos de una buena posición es controlar más casillas centrales para que el rival no se mueva libremente y entre en el territorio enemigo. En la imágen se puede observar un equilibrio en el control de las casillas centrales, visto desde la posición del juegador blanco.

Figura 2.11: Casillas controladas por peón

2.3.4 *Casillas controladas por caballo*

Cuenta el número de casillas que el caballo está atacando al mismo tiempo, observar que si el caballo está ubicado en casillas centrales controla más casillas que un caballo ubicado en la orilla. Dicho esto, en lo que respecta a esta característica, el jugador que controle más casillas tiene sus caballos mejor colocados.

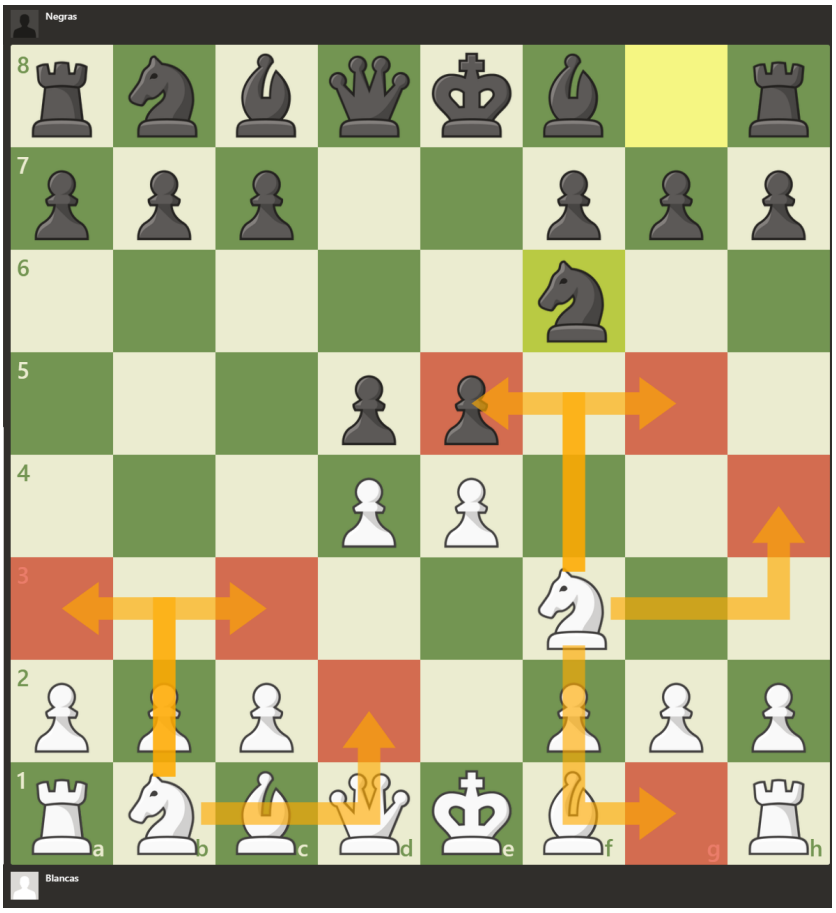

Figura 2.12: Casillas controladas por caballo

2.3.5 *Casillas controladas por alfil*

Cuenta el número de casillas por las que el alfil puede moverse libremente para atacar y por tanto controla. Observar que en esta posición, los peones avanzados hacia el centro permiten que el alcance de los alfiles se extienda. Una característica vital para entender la calidad de la posición.

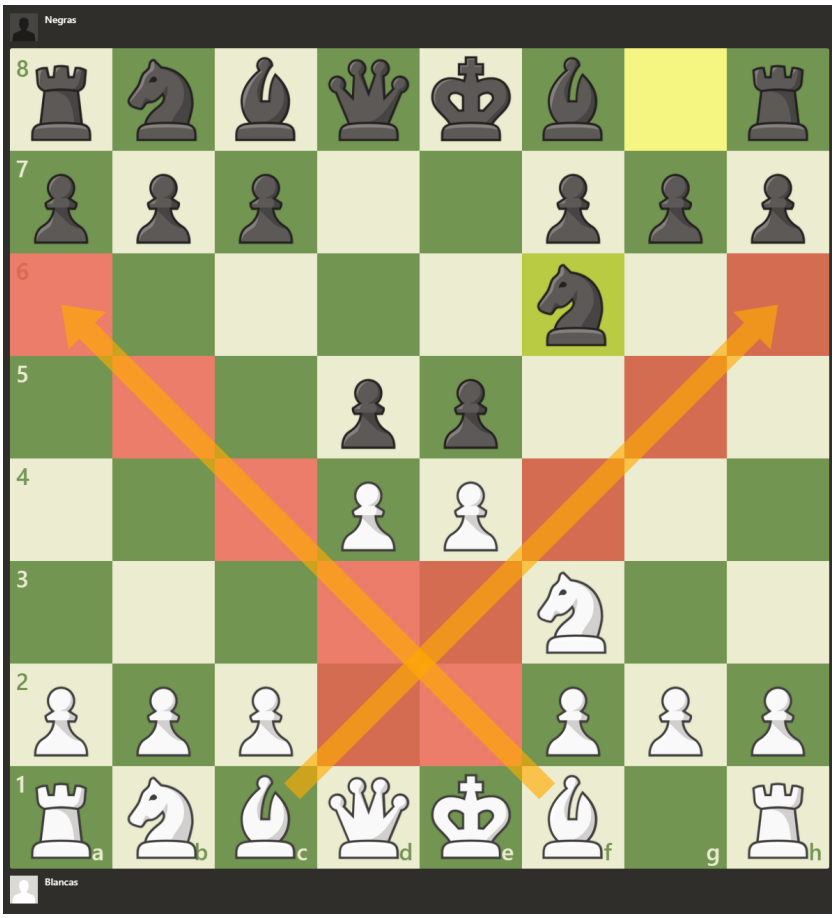

Figura 2.13: Casillas controladas por alfil

2.3.6 *Casillas controladas por torre*

Número de casillas por las que la torre puede moverse libremente. Entre más casillas se controlen, mejor indicativo de una buena posición.

Figura 2.14: Casillas controladas por torre

2.3.7 *Casillas controladas por Reina*

Número de casillas a las que la Reina, que es la pieza más poderosa del tablero, puede acceder libremente y por tanto está atacando. Nótese que en esta posición la Reina blanca accede a una diagonal con mayor número de cuadros que la negra, por tanto ataca más piezas y se puede concluir que está mejor colocada. En lo que respecta a la reina, en el ejemplo de la imagen, la posición blancas es superior.

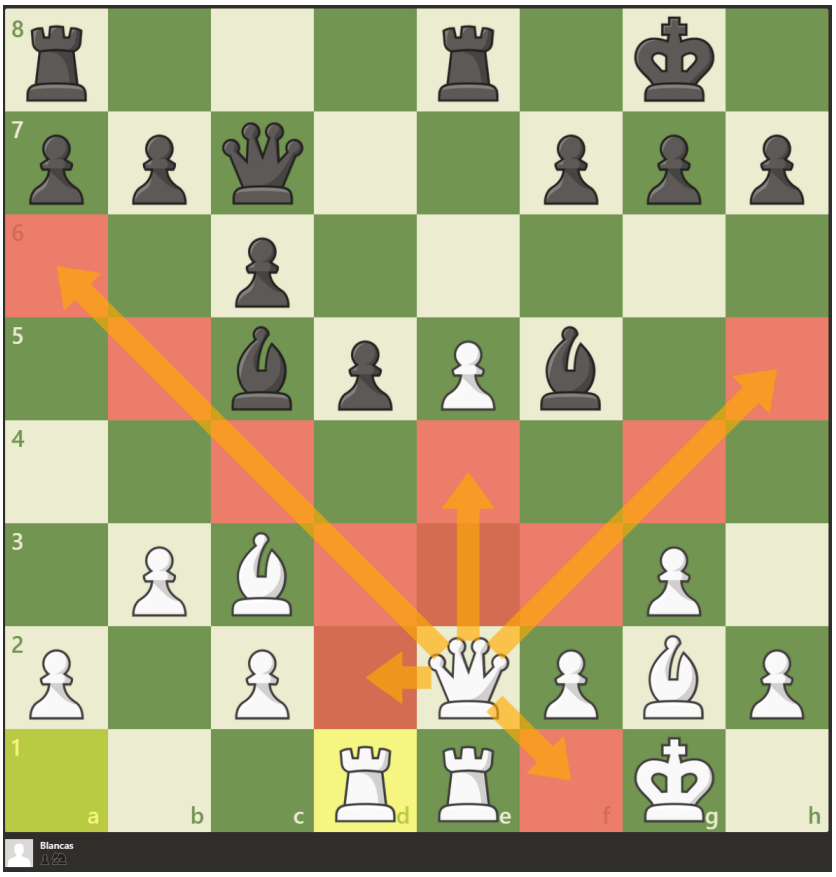

Figura 2.15: Casillas controladas por Reina

2.3.8 *Casillas controladas por Rey*

Casillas por las que puede moverse libremente el rey, en ocasiones funcionan como casillas de escape.

Figura 2.16: Casillas controladas por Rey

2.3.9 *Puntos de presión (>1 atacando)*

Los puntos de presión son casillas enemigas que son atacadas por más de una pieza. Son una de las características más importantes a la hora de evaluar una posición.

En el ejemplo de la imagen se indican en rojo los puntos de presión del jugador blanco. Con las flechas se ejemplifica que el peón negro de d5 está siendo atacado al mismo tiempo por la torre blanca en d1 y el alfil blanco en g2. También se ilustra con las flechas el punto de presión en c4. Los demás puntos de presión no tienen flechas para no complicar el gráfico, pero todos y cada uno son atacados por más de una pieza blanca a la vez. Generalmente los puntos de presión son debilidades del adversario, y, para mejorar la posición, el oponente atacado tiene que agregar defensores a la pieza vulnerable. En este caso, el peón negro en d5 solo tiene un defensor (el peón de c6), pero no sería buena idea que las blancas tomaran el peón porque perderían piezas de mayor valor (ya sea el alfil o la torre).

Figura 2.17: Puntos de presión

2.3.10 *Diagonales controladas*

A diferencia de las demás características, esta cuenta el número de diagonales que son controladas por un jugador y no de casillas. Puede indicar posiciones abiertas o cerradas. Controlar mayor número de diagonales suele indicar una mejor posición porque el oponente no se puede mover libremente.

Figura 2.18: Diagonales controladas por Reina y Alfiles

2.3.11 *Lineas controladas*

Similar a la característica de diagonales controladas, pero enfocada en las lineas horizontales y verticales. Una característica muy importante para ataques combinados en los que torres y dama pueden ser fundamentales para generar combinaciones ganadoras.

Figura 2.19: Lineas controladas por Reina y Torres

2.4 Descripción de las métricas

2.4.1 Accuracy

La métrica de *Accuracy* (exactitud) es ampliamente utilizada para evaluar la efectividad de modelos de clasificación en estadística y aprendizaje automático. Se define como la proporción de predicciones correctas (tanto positivas como negativas) entre el total de predicciones realizadas. Matemáticamente, se expresa como:

$$\text{Accuracy} = \frac{\text{NúmeroPrediccionesCorrectas}}{\text{NúmeroTotalPredicciones}}$$

Aunque es una medida intuitiva y de fácil comprensión, el *Accuracy* puede no ser una métrica fiable en escenarios donde existen clases desequilibradas. En tales casos, se recomienda complementar el análisis con otras métricas como la precisión y el recall, que pueden ofrecer una visión más detallada del rendimiento del modelo [13] [14] [15].

2.4.2 Precision

La métrica de *Precision* es fundamental en la evaluación de modelos de clasificación. Se define como la proporción de identificaciones positivas que fueron correctas, calculada por:

$$\text{Precision} = \frac{\text{Verdaderos Positivos}}{\text{Verdaderos Positivos} + \text{Falsos Positivos}}$$

Esta métrica es crucial en contextos donde los falsos positivos conllevan grandes costos. Por ejemplo, en ajedrez, una alta *Precision* significa que la mayoría de las predicciones de victoría fueron correctas[13] [15].

2.4.3 Recall

La métrica de *Recall*, también conocida como sensibilidad, tasa de verdaderos positivos o cobertura, es esencial para evaluar modelos de clasificación. El *Recall* se define como la proporción de verdaderos positivos identificados por el modelo respecto al total de casos realmente positivos. Se calcula con la fórmula:

$$\text{Recall} = \frac{\text{Verdaderos Positivos}}{\text{Verdaderos Positivos} + \text{Falsos Negativos}}$$

Esta métrica es particularmente valiosa en situaciones donde es crítico detectar todos los casos positivos, como en la detección de enfermedades o en la prevención de fraudes. En este caso la victoria en ajedrez. El *Recall* es prioritario en escenarios donde los falsos negativos representan un riesgo mayor que los falsos positivos. Que no es el caso, pero ayuda tenerlo en cuenta[13] [15].

2.4.4 *Brier Score*

El *Brier Score* es una métrica de evaluación para predicciones probabilísticas, utilizada ampliamente en la predicción de eventos. En el contexto de este documento, el *Brier Score* se utiliza para evaluar la exactitud de las predicciones sobre los resultados de las partidas, midiendo cuán cerca están las probabilidades predichas de los resultados reales y la diferencia entre las probabilidades pronosticadas y los resultados observados. Se calcula mediante la fórmula:

$$\text{Brier Score} = \frac{1}{N}\sum_{i=1}^{N}(p_i - o_i)^2$$

Donde p_i representa la probabilidad predicha de un resultado específico y o_i el resultado real (1 si ocurre, 0 de lo contrario). Esta métrica es particularmente valiosa en ajedrez para modelos predicativos que evalúan la probabilidad de diversos resultados como victorias o derrotas [16] [17].

2.4.5 *Roc Curve*

La Curva ROC (Receiver Operating Characteristic) es una herramienta esencial en el análisis de modelos de clasificación binaria. Representa gráficamente la relación entre la tasa de verdaderos positivos (sensibilidad) y la tasa de falsos positivos (1-especificidad) a varios umbrales de decisión. Es especialmente valiosa para identificar el umbral que balancea de manera óptima la sensibilidad y especificidad. La Curva ROC se acompaña típicamente del AUC (Area Under the Curve), una métrica que cuantifica el área total bajo la curva ROC. Un AUC de 1 indica un modelo perfecto, mientras que un AUC de 0.5 se asocia con un rendimiento aleatorio [18] [19].

2.5 *Descripción de los modelos*

2.5.1 *Máquina soporte vectorial kernel lineal*

Las Máquinas de Soporte Vectorial (SVM) son modelos ampliamente utilizados en aprendizaje automático para tareas de clasificación y regresión. La versión con kernel lineal de la SVM utiliza el producto escalar entre vectores de características para determinar el hiperplano óptimo de separación. Matemáticamente, el kernel lineal se define como:

$$K(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\mathbf{y}$$

Este enfoque es particularmente útil cuando los datos son linealmente separables, permitiendo a la SVM maximizar el margen

entre las clases adyacentes al hiperplano, lo cual es crucial para una buena generalización del modelo [20] [21].

2.5.2 *Máquina soporte vectorial kernel polinomial*

El kernel polinomial es una función de kernel utilizada en las máquinas de soporte vectorial para transformar los datos de entrada en un espacio de características más complejo. Se define por la fórmula:

$$K(\mathbf{x}, \mathbf{y}) = (c + \mathbf{x}^T \mathbf{y})^d$$

donde c es un coeficiente constante y d es el grado del polinomio. Este kernel es capaz de modelar relaciones complejas entre las clases al aumentar la dimensionalidad del espacio de características, lo cual es crucial para clasificaciones que no son linealmente separables en el espacio original[22] [21].

2.5.3 *Máquina soporte vectorial kernel rbf*

El kernel RBF (Radial Basis Function), o kernel gaussiano, es una función de kernel utilizada ampliamente en máquinas de soporte vectorial para clasificaciones donde los datos de entrada no son linealmente separables. Se define por la expresión:

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\gamma \|\mathbf{x} - \mathbf{y}\|^2\right)$$

donde γ es un parámetro que regula la amplitud de la función gaussiana. Este parámetro es crucial porque determina la velocidad con la que la función de similaridad disminuye con el aumento de la distancia entre las muestras. Un γ adecuadamente elegido permite al modelo capturar la complejidad de los datos mientras evita el sobreajuste [23] [21].

2.5.4 *XGboost*

XGBoost (eXtreme Gradient Boosting) es una implementación de árboles de decisión potenciados por gradientes que es notable por su eficiencia y efectividad. La función objetivo en XGBoost es una combinación de una función de pérdida y una función de regularización. La función objetivo de XGBoost se define como:

$$\text{Obj} = \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{2.1}$$

donde:

- y_i son los valores reales,

- $\hat{y}_i$ son las predicciones del modelo,
- f_k son los árboles individuales,
- K es el número de árboles.

La regularización Ω de cada árbol se calcula como:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2 \tag{2.2}$$

donde:

- T es el número de hojas del árbol,
- w son los pesos de las hojas,
- γ y λ son parámetros que controlan la penalización por número de hojas y los pesos de las hojas, respectivamente.

El proceso de optimización en XGBoost utiliza gradientes para minimizar la función objetivo, ajustando iterativamente los árboles para corregir errores residuales basados en el gradiente negativo de la función de pérdida.

La capacidad de XGBoost para manejar efectivamente datos desbalanceados lo hace adecuado para la clasificación de resultados de partidas de ajedrez, donde algunas partidas son más caóticas que otras.

Gracias a su regularización integrada, XGBoost evita el sobreajuste, esencial para desarrollar modelos que generalicen bien a nuevas partidas.

XGBoost mejora iterativamente, permitiendo afinar las predicciones hasta alcanzar un alto grado de precisión, ideal para clasificación de partidas de juegos de estrategia con reglas bien definidas [24] [25].

2.5.5 *LightGBM*

LightGBM es un marco de aprendizaje automático que utiliza algoritmos de aumento de gradiente para construir modelos predictivos eficientes y eficaces. La eficiencia de LightGBM proviene de su algoritmo de construcción de árboles de decisión basado en histogramas y dos técnicas principales: *GOSS* (Gradient-based One-Side Sampling) y *EFB* (Exclusive Feature Bundling).

GOSS mantiene los datos con gradientes grandes, que proporcionan información más valiosa, y muestrea aleatoriamente los datos con gradientes pequeños. Esto mantiene el equilibrio de la distribución de datos mientras reduce la cantidad de cálculos necesarios.

LightGBM mejora la eficiencia de los algoritmos de boosting de gradiente al mantener todas las instancias con gradientes grandes y realizar un muestreo aleatorio en las instancias con gradientes pequeños.

Esto se realiza con la intención de utilizar instancias más informativas para el aprendizaje. El proceso se define como:

$$\text{GOSS} = \begin{cases} \text{Mantiene todas las instancias con gradientes grandes (ej., los datos con los } a \times 100\,\% \text{ mayores gradientes)} \\ \text{Muestrea aleatoriamente una porción } (1-a) \text{ de las instancias con gradientes pequeños} \end{cases} \tag{2.3}$$

donde a es un hiperparámetro del algoritmo GOSS que determina la proporción de datos conservados.

El EFB se basa en la observación de que las características en conjuntos de datos espaciados tienen una baja correlación no nula. EFB agrupa eficientemente las características para reducir la dimensionalidad con una pérdida mínima de información, lo que permite un procesamiento más rápido. El algoritmo se puede describir como:

$$\text{EFB} = \sum_{i=1}^{n} \min_{\mathcal{B}} \left\{ \sum_{f_j, f_k \in \mathcal{B}, j \neq k} \text{conflict}(f_j, f_k) \right\} \tag{2.4}$$

donde $\mathcal{B}$ representa un paquete de características y $\text{conflict}(f_j, f_k)$ es una medida de qué tan a menudo las características f_j y f_k son no nulas simultáneamente.

LightGBM construye un histograma de las características y utiliza estos histogramas para dividir los nodos del árbol, lo que es mucho más eficiente en términos de memoria y velocidad que los métodos basados en listas o matrices [26] [27].

2.6 *Descripción de los experimentos o simulaciones*

Respecto a los modelos, primero se corrieron los benchmarks con la configuración 1 de la tabla 2.5. En esta configuración se hacía un análisis posicional relativo que solo incluía la posición del jugador en turno. En la configuración 2 2.5 el análisis es absoluto y toma en cuenta la posición tanto de blancas como de negras.

El mejor modelo se seleccionó considerando principalmente el accuracy porque interesa maximizar el número de clases predichas de forma correcta y el brier score porque interesa minimizar la diferencia entre las probabilidades pronosticadas y los resultados observados. Después se consideró el tiempo de entrenamiento. Tomando en cuenta la posibilidad de que este modelo pueda implementarse en producción.

Respecto a la inteligencia artificial. En las primeras iteraciones se le mostraba al ser humano en cada turno datos únicamente sobre su posición, lo que dificultaba que evaluara correctamente pros y contras al no visualizar la posición del oponente (la máquina) esto se corrigió en la segunda iteración de ingeniería de características.

Otro problema que se tuvo fue que la máquina no simulaba de forma correcta la aleatoriedad con que un humano juega aperturas porque no existían pesos en esta decisión: te podía jugar con la misma probabilidad

Configuración	Columnas
1	turns, ctrld_pawn, ctrld_knight, ctrld_bishop, ctrld_rook, ctrld_queen, ctrld_king, preassure_points, controlled_diagonals, controlled_lines
2	game_id, rated, turns, winner, time_increment, white_rating, black_rating, moves, opening_code, opening_moves, opening_fullname, opening_shortname, opening_variation, moves_fen, current_turn, w_ctrld_pawn, w_ctrld_knight, w_ctrld_bishop, w_ctrld_rook, w_ctrld_queen, w_ctrld_king, w_preassure_points, w_ctrld_diagonals, w_ctrld_lines, b_ctrld_pawn, b_ctrld_knight, b_ctrld_bishop, b_ctrld_rook, b_ctrld_queen, b_ctrld_king, b_preassure_points, b_ctrld_diagonals, b_ctrld_lines, rated_cod, winner_cod, current_turn_cod, time_increment_cod, opening_code_cod, opening_fullname_cod, opening_shortname_cod, opening_variation_cod, moves_fen_cod

Tabla 2.5: Configuraciones de columnas para análisis de datos de ajedrez

aperturas que en el dataset de juegos históricos habían sido jugadas una sola vez (Australian Defense) a aperturas jugadas más de 2000 veces (Sicilian Defense). Lo mismo sucedía con los movimientos, dentro de una apertura te podía jugar con la misma probabilidad el movimiento más común que el menos común. Esto se resolvió introduciendo el concepto de pesos relativos sacando las probabilidades reales tanto de que un jugador te juegue ciertas aperturas como ciertos movimientos y tomándolas en cuenta para las decisiones de la máquina.

Después de introducir el concepto de pesos relativos, se tuvo el problema de que precisamente porque había pesos definidos había aperturas o movimientos cuya probabilidad de que fueran jugados era tan baja que jamás se jugarían, esto se resolvió haciendo una normalización para ajustar los pesos menores a 1 como 1, por ejemplo la relación 13.12:.0.0049 fue ajustada a 13:1 siendo el primer número el peso para la apertura más jugada (Sicilian Defense) y el segundo la menos jugada (Australian Defense). Algo similar se realizó para la elección de movimientos por parte de la máquina y está documentado en el anexo de código 4.11.

3 Resultados y discusión

La hipótesis que se persigue en este proyecto es que los modelos basados en árboles de decisión son mejores para predecir posiciones ganadoras en partidas de ajedrez porque son más efectivos que las máquinas de soporte vectorial para clasificar resultados de partidas de ajedrez a partir de características posicionales debido a que son capaces de manejar de mejor forma la no linealidad y la alta dimensionalidad de los datos, ya que en el ajedrez las interacciones entre piezas y posiciones no siguen patrones lineales simples . Los modelos basados en árboles pueden manejar grandes cantidades de datos con muchas características sin aumento en la carga computacional, lo que es útil en el ajedrez ya que se pueden extraer diferentes características, las que cubiertas en esta investigación y algunas más como una evaluación de la seguridad del rey o control del centro.

3.1 Resultados del modelo

Estos son los resultados de la primera configuración de columnas, en la que se tomaba en cuenta solo la posición del jugador en turno:

Modelo	Acu (%)	Prec (%)	Reca (%)	Brier
SVM K = linear	51.60 %	26.62 %	51.60 %	0.25
SVM K = poly	51.58 %	49.40 %	51.58 %	0.25
SVM K = rbf	52.85 %	52.64 %	52.85 %	0.25
XGBoost	88.68 %	88.68 %	88.68 %	0.09
LightGBM	88.75 %	88.75 %	88.75 %	0.10

Tabla 3.1: Resultados de modelos de clasificación en el conjunto de Test (Configuración 1)

Por otra parte, estos son los resultados de los modelos evaluando ambas posiciones, negras y blancas a la vez:

Modelo	Acu (%)	Prec (%)	Reca (%)	Brier
SVM K = linear	90.37 %	90.37 %	90.37 %	0.07
SVM K = poly	90.53 %	90.55 %	90.53 %	0.07
SVM K = rbf	82.07 %	84.52 %	82.07 %	0.11
XGBoost	91.45 %	91.45 %	91.45 %	0.06
XGBoost opt. hip.	**91.73 %**	**91.74 %**	**91.73 %**	**0.06**
LightGBM	91.49 %	91.49 %	91.49 %	0.06

Tabla 3.2: Resultados de modelos de clasificación en el conjunto de Test (Configuración 2)

3.2 *Resultados de la inteligencia artificial*

Se logra desarrollar una inteligencia artificial que es capaz de jugar ajedrez alguna de las 128 aperturas y sus 1477 variantes y sirve cómo entrenador porque brinda información de la posición y un porcentaje de victoria después de cada movimiento generado (Humano o máquina).

Esta es una captura de un juego 3.1 en el que la IA escoge jugar con blancas la defensa escandinava y en el transcurso se alcanzan 3 variantes de la misma y la probabilidad de victoria va fluctuando conforme la posición va cambiando.

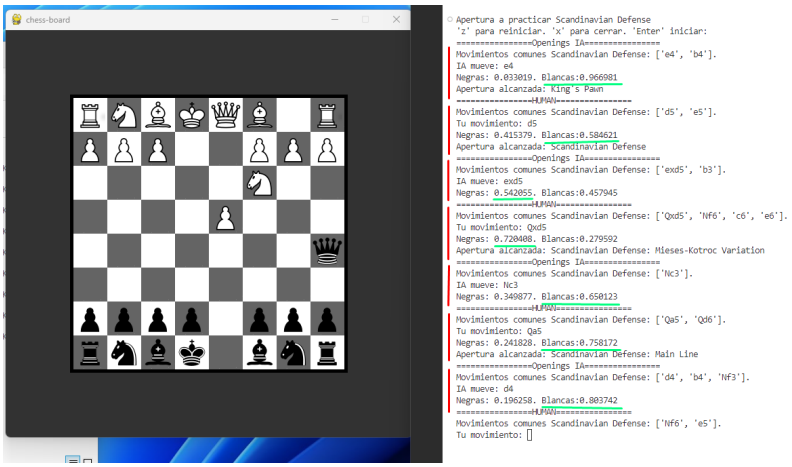

Figura 3.1: Juego de openings IAlex

Y en este ejemplo se muestra una captura 3.2 de pantalla en la que se aprecia como presenta la información de la posición la IA entrenador que recibe el nombre de openings_IAlex:

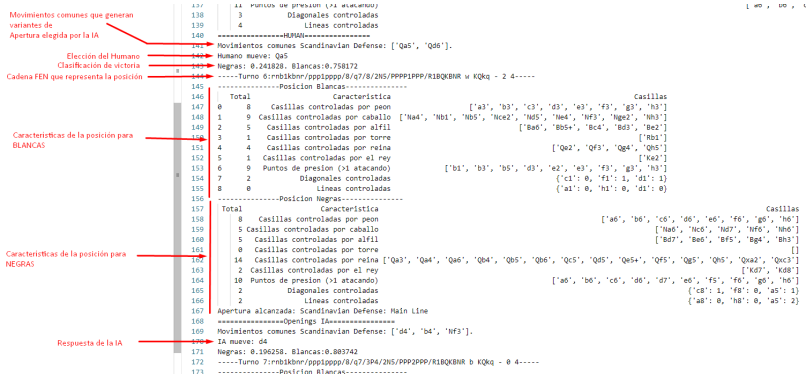

Figura 3.2: IAlex logs: Mostrando características posicionales

4 Conclusiones y trabajo futuro

4.1 Conclusiones

Se obtienen correctamente las características posicionales descritas en los objetivos y se usan de forma satisfactoria para entrenar el modelo de clasificación.

El modelo seleccionado como el mejor es el de la tabla 'Modelo ganador' 4.1, que resulta tener un brief score satisfactiorio indicando que la diferencia entre las probabilidades pronosticadas y los resultados observados es mínima y tiene un accuracy bueno, que indica que el numero de clases predichas de forma correcta es alto.

Modelo	Acu (%)	Prec (%)	Reca (%)	Brier
XGBoost opt. hip.	**91.73 %**	**91.74 %**	**91.73 %**	**0.06**

Tabla 4.1: Modelo ganador

En la matriz de confusión 4.1 se observa que las Blancas predichas como negras o las negras predichas como blancas son pocas, al rededor del 10 %, mientras que el 90 % del tiempo, el modelo acierta.

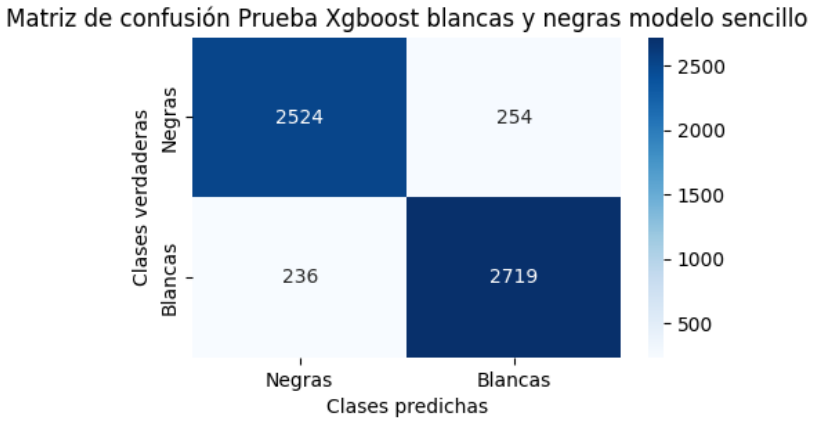

Figura 4.1: Matriz de confusión modelo ganador

En los gráficos de precisión vs recall y curva roc 4.2vemos precisamente el comportamiento ideal.

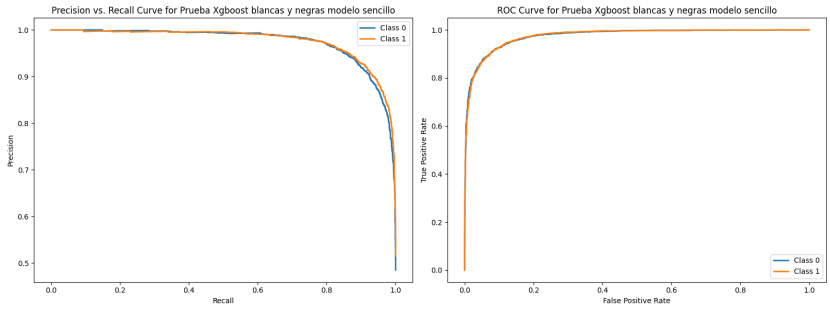

Figura 4.2: Precisión vs Recall y Curva ROC modelo ganador

La inteligencia artificial elije de forma pseudo aleatoria una apertura e interactúa con el humano emulando el comportamiento de un oponente humano siendo capaz de responder a jugadas y mostrando correctamente las características de la posición, lo que la hace un sistema de entrenamiento didáctico.

Se muestra de forma correcta la clasificación de victoria.

4.2 *Trabajo futuro*

Este trabajo de grado puede ser perfectible en los siguientes aspectos:

- Mejorar el aspecto gráfico para que sea más fácil y llamativo.
- Entrenar el modelo con partidas más actuales usando las APIs de las páginas de ajedrez más comunes como chess.com y lichess.com.
- Incrementar el número de características que se muestran al usuario y que se usan para la predicción
  - Calidad de casillas controladas en el centro
  - Piezas defendiendo al rey
  - Piezas atacando al rey enemigo
  - Suma de la calidad de las piezas en el tablero
  - etc.

Apéndice Código

4.3 benchmarks.py

```
# Libreras
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import svm
from model_helpers import eval_perform_multi_class
import pandas as pd
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')
CLASS_NAMES = ["Negras", "Blancas"]

# Carga de datos
data_path = "../CHESS/data/df_3_cod.csv"
df_3 = pd.read_csv(data_path)
df_3.columns

# Seleccin de columnas
X = df_3.copy()
y_name = "winner_cod"
# X es el dataframe eliminando la variable de salida
X = X.drop(columns=['rated','winner',y_name,
    'current_turn','time_increment','opening_code','opening_fullname','opening_shortname','opening_variation','moves_
X = X.drop(columns=['game_id', 'white_rating', 'black_rating',
    'moves', 'current_turn_cod', 'opening_moves', 'rated_cod',
    'current_turn_cod', 'time_increment_cod', 'opening_code_cod',
    'opening_fullname_cod', 'opening_shortname_cod',
    'opening_variation_cod', 'moves_fen_cod'])
# Y es un array unidimensional (ravel) de la variable de salida
Y = df_3[y_name].ravel()
print(X.columns)

# Divisin train/test
X_train, X_test, Y_train, Y_test = train_test_split(X,
    Y,test_size=0.3)

# SVM
```

```
# Crear un clasificador SVM para clasificacin con kernel lineal,
    polinomial y de funcion de base radial
mod_linear = svm.SVC(kernel='linear',C=1, probability=True)
mod_poly = svm.SVC(kernel='poly',degree=2,C=1, probability=True)
mod_rbf = svm.SVC(kernel='rbf',C=1,gamma='auto', probability=True)
# Entrenamiento del kernel lineal
mod_linear.fit(X_train, Y_train)
# Evaluacin: Salida Y & "hat" () que denota predicciones estimadas.
Yhat_linear_test = mod_linear.predict(X_test)
Yhat_linear_train = mod_linear.predict(X_train)
Yhat_linear_test_prob = mod_linear.predict_proba(X_test)
Yhat_linear_train_prob = mod_linear.predict_proba(X_train)
eval_perform_multi_class(Y_test,Yhat_linear_test,Yhat_linear_test_prob,CLASS_NAMES,"Prueba
    SVM kernel lineal caractersticas blancas y negras")
eval_perform_multi_class(Y_train,Yhat_linear_train,Yhat_linear_train_prob,CLASS_NAMES,
    "Entrenamiento SVM kernel lineal caractersticas blancas y
    negras")

# Entrenamiento del kernel polinomial
mod_poly.fit(X_train, Y_train)
# Evaluacin
Yhat_poly_test = mod_poly.predict(X_test)
Yhat_poly_train = mod_poly.predict(X_train)
Yhat_poly_test_prob = mod_poly.predict_proba(X_test)
Yhat_poly_train_prob = mod_poly.predict_proba(X_train)
eval_perform_multi_class(Y_test,Yhat_poly_test,
    Yhat_poly_test_prob,CLASS_NAMES, "Prueba SVM kernel polinomial
    caractersticas blancas y negras")
eval_perform_multi_class(Y_train,Yhat_poly_train,
    Yhat_poly_train_prob,CLASS_NAMES, "Entrenamiento SVM kernel
    polinomial caractersticas blancas y negras")

# Entrenamiento del kernel rbf
mod_rbf.fit(X_train, Y_train)
# Evaluacin
Yhat_rbf_test = mod_rbf.predict(X_test)
Yhat_rbf_train = mod_rbf.predict(X_train)
Yhat_rbf_test_prob = mod_rbf.predict_proba(X_test)
Yhat_rbf_train_prob = mod_rbf.predict_proba(X_train)
eval_perform_multi_class(Y_test,Yhat_rbf_test,
    Yhat_rbf_test_prob,CLASS_NAMES,"Prueba SVM kernel rbf
    caractersticas blancas y negras")
eval_perform_multi_class(Y_train,Yhat_rbf_train,
    Yhat_rbf_train_prob,CLASS_NAMES,"Entrenamiento SVM kernel rbf
    caractersticas blancas y negras")

# Xgboost
import xgboost as xgb
# Crear un clasificador XGBoost
xgboost = xgb.XGBClassifier()
# Entrenar el modelo en los datos de entrenamiento
```

```
xgboost.fit(X_train, Y_train)
# Evaluacin
Yhat_xgboost_test = xgboost.predict(X_test)
Yhat_xgboost_train = xgboost.predict(X_train)
Yhat_xgboost_test_prob = xgboost.predict_proba(X_test)
Yhat_xgboost_train_prob = xgboost.predict_proba(X_train)
eval_perform_multi_class(Y_test,Yhat_xgboost_test,
    Yhat_xgboost_test_prob,CLASS_NAMES,"Prueba Xgboost
    caractersticas blancas y negras")
eval_perform_multi_class(Y_train,Yhat_xgboost_train,
    Yhat_xgboost_train_prob,CLASS_NAMES,"Entrenamiento Xgboost
    caractersticas blancas y negras")

#LightGBM
from lightgbm import LGBMClassifier
lgbm = LGBMClassifier()
lgbm.fit(X_train, Y_train)
# Evaluacin
Yhat_lgbm_test = lgbm.predict(X_test)
Yhat_lgbm_train = lgbm.predict(X_train)
Yhat_lgbm_test_prob = lgbm.predict_proba(X_test)
Yhat_lgbm_train_prob = lgbm.predict_proba(X_train)
eval_perform_multi_class(Y_test,Yhat_lgbm_test,Yhat_lgbm_test_prob,
    CLASS_NAMES, "Prueba LightGBM caractersticas blancas y negras")
eval_perform_multi_class(Y_train,Yhat_lgbm_train,Yhat_lgbm_train_prob,
    CLASS_NAMES,"Entrenamiento LightGBM caractersticas blancas y
    negras")
```

4.4 *data_preparation.py*

```
import pandas as pd
org_data_path = "../CHESS/data/df_3.csv"
df_3 = pd.read_csv(org_data_path)
# Eliminar empate ya que no abona al objetivo
df_3 = df_3[df_3['winner'] != 'Draw']
print(df_3.info())

# Codificacin con Label Encoder
from sklearn.preprocessing import LabelEncoder
# Codificando todas las variables categricas consideradas en
    valores numricos
le1 = LabelEncoder() #rated
le2 = LabelEncoder() #winner
le3 = LabelEncoder() #time_increment
le4 = LabelEncoder() #opening_code
le5 = LabelEncoder() #opening_fullname
le6 = LabelEncoder() #opening_shortname
le7 = LabelEncoder() #opening_variation
le8 = LabelEncoder() #moves_fen
```

```
le9 = LabelEncoder() #current_turn

# Apply label encoding to each categorical column
df_3['rated_cod'] = le1.fit_transform(df_3['rated']) # Ordinal
df_3['winner_cod'] = le2.fit_transform(df_3['winner']) # No
    ordinal, pero solo tres clases
df_3['current_turn_cod'] = le9.fit_transform(df_3['current_turn'])
    # Ordinal
df_3['time_increment_cod'] =
    le3.fit_transform(df_3['time_increment']) # Ordinal
df_3['opening_code_cod'] = le4.fit_transform(df_3['opening_code'])
    # No ordinal, muchas categorias
df_3['opening_fullname_cod'] =
    le5.fit_transform(df_3['opening_fullname']) # No ordinal,
    muchas categorias
df_3['opening_shortname_cod'] =
    le6.fit_transform(df_3['opening_shortname']) # No ordinal,
    muchas categorias
df_3['opening_variation_cod'] =
    le7.fit_transform(df_3['opening_variation']) # No ordinal,
    muchas categorias
df_3['moves_fen_cod'] = le8.fit_transform(df_3['moves_fen']) # No
    ordinal, muchas categorias

# Visualizacin de clases para el target
pares_unicos = df_3[['winner', 'winner_cod']].drop_duplicates()
print(pares_unicos)

# Guardar el CSV
df_3.to_csv("../CHESS/data/df_3_cod.csv", index=False) # Data
    frame con features basadas en ventaja posicional del turno
```

4.5 *eda.py*

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

org_data_path = "../CHESS/data/chess_games.csv"
chess_data = pd.read_csv(org_data_path)

# Tratamiento de valores faltantes
print(chess_data.isnull().sum())

# Calculo de valores faltantes en columna opening response
total_entries = 20058
missing_opening_response = 18851
```

```
percentage_missing = (missing_opening_response / total_entries) *
    100
print(percentage_missing)

# Eliminacin de columna opening response
df_1 = chess_data.drop(['opening_response'], axis=1)

#rellenando la columna Opening_variation los blanks con
    "traditional opening"
df_1['opening_variation'] =
    chess_data['opening_variation'].fillna('traditional opening')
df_1.info()

# Descripcin de las columnas numricas del DF
chess_data.describe()

'''
Ahora que se tiene un dataframe sin valores faltantes. Es posible
    proceder con los siguientes analisis:
- Distribucin de las variables numricas
- Exploracin de las variables categricas
Para dividir las calificaciones de los jugadores en las categoras
    de "Beginner", "Intermediate", "Advanced", y "Master" se
    determinar mediante cuartiles.

Primero, se determinan los valores de estos cuartiles para las
    calificaciones de los jugadores de blanco y negro.

Luego, se crea una funcin que asigna cada calificacin a una de
    estas categoras segn los cuartiles calculados.
'''

import numpy as np
from scipy import stats
combined_ratings = pd.concat([chess_data['white_rating'],
    chess_data['black_rating']])
combined_ratings
print("Promedio de Ratings: %.3f" % np.mean(combined_ratings))
print("Mediana de Ratings: %.3f" % np.median(combined_ratings))
print("Desviacin estandar de elos: %.3f" %
    np.std(combined_ratings))
overall_quartiles = combined_ratings.quantile([0.25, 0.5,
    0.75]).values
print("50%c de los datos entre %.3f y %.3f" % ('%',
    overall_quartiles[0], overall_quartiles[2]))
print("Skew de la distribucin: %.3f" %
    (stats.skew(combined_ratings)))
print(overall_quartiles)

'''
```

```
Conversin a variable categrica de las coluimnas white_rating y
    black_rating

Clasificaciones:
- Begginer: Abajo de percentil 25%
- Intermediate: Entre 25% y 50%
- Advanced: Entre 50% y 75%
- Master: Arriba de 75%

Por tanto:
- Percentil 25%: 1394
- Percentil 50%: 1564
- Percentil 75%: 1788

Con esta informacin se crean dos categoras: white_category y
    black_category
'''
def categorize_rating(rating, overall_quartiles):
   if rating <= overall_quartiles[0]:
      return 'Beginner'
   elif rating <= overall_quartiles[1]:
      return 'Intermediate'
   elif rating <= overall_quartiles[2]:
      return 'Advanced'
   else:
      return 'Master'

chess_data['white_category'] =
    chess_data['white_rating'].apply(lambda x:
    categorize_rating(x, overall_quartiles))
chess_data['black_category'] =
    chess_data['black_rating'].apply(lambda x:
    categorize_rating(x, overall_quartiles))


# Simplificacin de la variable categrica time_increment
# Definimos cadenas vlidas para cada categora
bullet_times = {str(i): 'Bullet' for i in range(0, 2)}
blitz_times = {str(i): 'Blitz' for i in range(3, 5)}
rapid_times = {str(i): 'Rapid' for i in range(6, 30)}
classical_times = {str(i): 'Classical' for i in range(30, 60)}
personalizado_times = {str(i): 'Personalizado' for i in range(60,
    181)}

# Unimos todos los diccionarios en uno solo
all_times = {**bullet_times, **blitz_times, **rapid_times,
    **classical_times, **personalizado_times}

def categorize_time_base(time_increment):
   # Separar el tiempo base y el incremento
   time_base, _ = time_increment.split('+')
```

```
   # Devolver la categora correspondiente basndonos solo en el
       tiempo base
   return all_times.get(time_base, 'Personalizado')

# Aplicar la funcin para crear una nueva columna 'category'
chess_data['time_category'] =
    chess_data['time_increment'].apply(categorize_time_base)

# Verificar los resultados
chess_data[['time_increment', 'time_category']].head()

# Visualizacin de columnas categricas
categorical_columns = ['victory_status', 'winner', 'time_category',
     'moves', 'opening_code', 'opening_shortname',
         'opening_variation', 'white_category',
     'black_category']

# Funcin para trazar las 5 categoras principales de una columna
    dada
def plot_top_categories(data, column, ax, title, font_size=20):
   top_categories = data[column].value_counts().head(5)
   top_categories.plot(kind='bar', ax=ax, color='skyblue')
   ax.set_title(title, fontsize=font_size + 5)
   # ax.set_xlabel(column, fontsize=font_size)
   ax.set_ylabel('Frequency', fontsize=font_size)
   ax.tick_params(axis='both', which='major', labelsize=font_size)
   ax.tick_params(axis='both', which='minor', labelsize=font_size
       - 2)

# Crear la figura y los ejes
fig, axes = plt.subplots(nrows=5, ncols=2, figsize=(30, 40))
fig.tight_layout(pad=5.0)

# Iterar a travs de cada columna y eje, trazando las categoras
    principales
for ax, column in zip(axes.flatten(), categorical_columns):
   if column in chess_data:
      plot_top_categories(chess_data, column, ax, f'Top 5
          {column.capitalize()}')

# Ajustar la figura para mostrar todos los grficos claramente
plt.tight_layout()
plt.show()


# Visualizacin de distribucin de variables numericas
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 8))
fig.tight_layout(pad=5.0)

# Lista de columnas numricas a graficar, excluyendo 'game_id'
```

```
numeric_columns = ['turns', 'white_rating', 'black_rating',
    'opening_moves']

# Graficar histogramas para cada columna numrica
for ax, column in zip(axes.flatten(), numeric_columns):
   chess_data[column].hist(bins=30, ax=ax, color='cadetblue')
   ax.set_title(f'Distribucin columna {column}')
   ax.set_xlabel(column)
   ax.set_ylabel('Frecuencia')

# Ajustar la disposicin para evitar la superposicin de etiquetas
plt.tight_layout()
plt.show()

# Sskewness and kurtosis for the specified columns
skewness = chess_data[['turns',
    'opening_moves']].skew().rename('Skewness')
kurtosis = chess_data[['turns',
    'opening_moves']].kurtosis().rename('Kurtosis')

# Combining the results into a single DataFrame
skew_kurtosis_stats = pd.concat([skewness, kurtosis], axis=1)
skew_kurtosis_stats


# Boxplot for the 'opening_moves' column
plt.figure(figsize=(8, 6))
plt.boxplot(chess_data['opening_moves'], vert=False)
plt.title('Boxplot of Opening Moves')
plt.xlabel('Number of Opening Moves')
plt.show()

# Distribucin de nivel de jugadores vs Frecuencia de juegos
# Distribution of player ratings
plt.figure(figsize=(10, 5))
sns.histplot(chess_data['white_rating'], color='blue', bins=30,
    kde=True, label='Nivel Blancas')
sns.histplot(chess_data['black_rating'], color='red', bins=30,
    kde=True, label='Nivel Negras')
plt.axvline(x=overall_quartiles[0], linestyle='--', markersize=12)
plt.axvline(x=overall_quartiles[2], linestyle='--', markersize=12)
plt.title('Distribucin de nivel de jugadores')
plt.xlabel('Nivel')
plt.ylabel('Frecuencia')
plt.legend()

plt.tight_layout()
plt.show()

# Aperturas ms comunes segn nivel y su frecuencia de juego
# Filter data for each category
```

```
beginner_openings = chess_data[chess_data['white_category'] ==
    'Beginner']['opening_fullname'].value_counts().head(5)
intermediate_openings = chess_data[chess_data['white_category'] ==
    'Intermediate']['opening_fullname'].value_counts().head(5)
advanced_openings = chess_data[chess_data['white_category'] ==
    'Advanced']['opening_fullname'].value_counts().head(5)
master_openings = chess_data[chess_data['white_category'] ==
    'Master']['opening_fullname'].value_counts().head(5)

# Set up the visualization layout
fig, axes = plt.subplots(2, 2, figsize=(18, 12))

sns.barplot(ax=axes[0, 0], x=beginner_openings.values,
    y=beginner_openings.index, palette='Pastel1')
axes[0, 0].set_title('Top 5 Aperturas nivel principiante')
axes[0, 0].set_xlabel('Frecuencia')
axes[0, 0].set_ylabel('Apertura')

sns.barplot(ax=axes[0, 1], x=intermediate_openings.values,
    y=intermediate_openings.index, palette='Pastel2')
axes[0, 1].set_title('Top 5 Aperturas nivel intermedio')
axes[0, 1].set_xlabel('Frequency')
axes[0, 1].set_ylabel('Opening')

sns.barplot(ax=axes[1, 0], x=advanced_openings.values,
    y=advanced_openings.index, palette='Set3')
axes[1, 0].set_title('Top 5 Aperturas nivel avanzado')
axes[1, 0].set_xlabel('Frequency')
axes[1, 0].set_ylabel('Opening')

sns.barplot(ax=axes[1, 1], x=master_openings.values,
    y=master_openings.index, palette='Set1')
axes[1, 1].set_title('Top 5 Aperturas nivel Maestro')
axes[1, 1].set_xlabel('Frequency')
axes[1, 1].set_ylabel('Opening')

plt.tight_layout()
plt.show()


# Distribucin de color y estatus de victoria de los ganadores
sns.set_theme(style="whitegrid")

victory_counts = chess_data['victory_status'].value_counts()
winner_counts = chess_data['winner'].value_counts()
victory_percentages = (victory_counts / victory_counts.sum()) * 100
winner_percentages = (winner_counts / winner_counts.sum()) * 100

# Set up the figure and axes for updated bar charts with numbers
    and percentages
fig, ax = plt.subplots(1, 2, figsize=(14, 5))
```

```
# Bar chart for winners with count and percentage labels
ax[0].bar(winner_counts.index, winner_counts.values,
    color='orange')
ax[0].set_title('Distribucin de color de los ganadores')
ax[0].set_xlabel('Ganador')
ax[0].set_ylabel('Nmero de partidas')
for i, v in enumerate(winner_counts):
   ax[0].text(i, v + 50, f"{v} ({winner_percentages[i]:.1f}%)",
       color='black', ha='center')

# Bar chart for victory statuses with count and percentage labels
ax[1].bar(victory_counts.index, victory_counts.values,
    color='purple')
ax[1].set_title('Distribucin del estatus de la victoria')
ax[1].set_xlabel('Estatus de la victoria')
ax[1].set_ylabel('Nmero de juegos')
for i, v in enumerate(victory_counts):
   ax[1].text(i, v + 50, f"{v} ({victory_percentages[i]:.1f}%)",
       color='black', ha='center')



plt.tight_layout()
plt.show()


# Frecuancia del top 10 de aperturas de maestros
master_openings = chess_data[chess_data['white_category'] ==
    'Master']['opening_fullname'].value_counts().head(10)


# Win rates for these openings
win_rates =
    chess_data[chess_data['opening_fullname'].isin(master_openings.index)].groupby(['opening_fullname',
    'winner']).size().unstack(fill_value=0)
win_rates['total'] = win_rates.sum(axis=1)
win_rates['white_win_rate'] = win_rates['White'] /
    win_rates['total']
win_rates['black_win_rate'] = win_rates['Black'] /
    win_rates['total']
win_rates['draw_rate'] = win_rates['Draw'] / win_rates['total']

# Sorting by total games to see the most popular openings in terms
    of games played
win_rates_sorted = win_rates.sort_values(by='total',
    ascending=False)

# Plotting
fig, ax = plt.subplots(figsize=(12, 8))
```

```
win_rates_sorted[['white_win_rate', 'black_win_rate',
    'draw_rate']].plot(kind='barh', ax=ax, color=['purple',
    'black', 'gray'])


ax.set_title('Frecuencia del top 10 de aperturas de maestros')
ax.set_xlabel('Frecuencia')
ax.set_ylabel('Apertura')
ax.legend(title='Jugador')
plt.xticks(rotation=90)
plt.show()

# Guardar el dataframe
chess_data.to_csv("../CHESS/data/df_1.csv", index=False)
```

4.6 *extract_features.py*

```
"""
Este archivo de python tiene la intencin de proveer una serie de
    metodos para extraer caracteristicas
desde posiciones dadas por cadenas FEN.
"""
from chess import Board
from chess import square_rank, square_file, square
from chess import PIECE_TYPES, SQUARES, square_name
from chess import KNIGHT, BISHOP, ROOK, QUEEN, KING
import random
import pandas as pd

def toggle_turn_on_fen(fen):
   """Cambia el turno en un string FEN.
   Este mtodo tiene la intencin de poder calcular caractersticas
       del oponente estableciendo
   una situacin ficticia en la que tu turno es su turno
   """
   parts = fen.split(' ')
   parts[1] = 'w' if parts[1] == 'b' else 'b'
   return ' '.join(parts)

def get_custom_board(fen, turn = None) -> Board:
   """Devuelve una instancia de Board con el turno especificado
   """
   if turn is not None: # A turn was specified
      parts = fen.split(' ')
      parts[1] = 'w' if turn else 'b'
      fen = ' '.join(parts)

   board = Board(fen)
   return board
```

```
def get_legal_moves_san(fen):
   """
   Devuelve una lista con todos los movimientos legales de cada
       pieza en el tablero
   para el jugador en turno

   Args:
   fen (str): La cadena FEN que representa la posicin actual del
       tablero.

   Returns:
   list: Lista de las casillas de puntos de presin en notacin SAN.
   """
   legal_moves = {}
   # Fill legal moves according one piece type
   for piece_type in PIECE_TYPES:
      legal_moves[piece_type] =
          get_legal_moves_from_piece_type(fen, piece_type)
   return legal_moves

def get_legal_moves_from_piece_type(fen, piece_type, turn = None):
   """
   Devuelve los movimientos legales de una pieza dada en el turno
       correspondiente a la posicin fen

   Args:
   fen (str): The FEN string representing the current board
       position.
   piece_type (int):
      PAWN = 1
      KNIGHT = 2
      BISHOP = 3
      ROOK = 4
      QUEEN = 5
      KING = 6

   Returns:
   list: A list of legal moves for given piece in uci notation.
   """
   board = get_custom_board(fen, turn)
   legal_moves = board.legal_moves
   moves = [board.san(move) for move in legal_moves if
       board.piece_at(move.from_square).piece_type == piece_type]
       # move.uci for uci format
   return sorted(moves)

def count_legal_moves_from_piece_type(fen, piece_type, turn =
    None):
   """
```

```
    Cuenta el nmero de movimientos legales de una pieza dada en el
        turno correspondiente a la posicin fen

    Args:
    fen (str): The FEN string representing the current board
        position.
    piece_type (int):
       PAWN = 1
       KNIGHT = 2
       BISHOP = 3
       ROOK = 4
       QUEEN = 5
       KING = 6

    Returns:
    list: A list of legal moves for given piece in uci notation.
    """
    board = get_custom_board(fen, turn)
    legal_moves = board.legal_moves
    moves = [1 for move in legal_moves if
        board.piece_at(move.from_square).piece_type == piece_type]
        # move.uci for uci format
    return sum(moves)

def get_pawn_capture_squares(fen, turn:bool = True):
    """Obtiene las casillas controladas por peon (puede comer en
        siguiente turno)

    Args:
       fen (str): Cadena Fen
       turn (bool, optional): True: Blancas. False: Negras

    Returns:
       set: Obtiene las casillas controladas por peon (puede comer
           en siguiente turno)
    """
    # Dividir la cadena FEN para obtener la disposicin de las
        piezas en el tablero
    board_setup = fen.split()[0]

    # Convertir la disposicin del tablero a una matriz 8x8 para
        facilitar el acceso a cada casilla
    rows = board_setup.split('/')
    board = [list(row.replace('8', '........')
              .replace('7', '.......')
              .replace('6', '......')
              .replace('5', '.....')
              .replace('4', '....')
              .replace('3', '...')
              .replace('2', '..')
              .replace('1', '.')) for row in rows]
```

```
    captures = {'white': set(), 'black': set()}
    # Movimiento diagonal para captura
    direction_white = -1
    direction_black = 1
    diagonals = [-1, 1] # Diagonales izquierda y derecha
    for row in range(8):
        for col in range(8):
            if board[row][col] == 'P': # Pen Blancas
                for dcol in diagonals:
                    nrow, ncol = row + direction_white, col + dcol
                    if 0 <= nrow < 8 and 0 <= ncol < 8: # Verificar
                        lmites del tablero
                        captures['white'].add(chr(97 + ncol) + str(8 -
                            nrow))
            elif board[row][col] == 'p': # Pen Negras
                for dcol in diagonals:
                    nrow, ncol = row + direction_black, col + dcol
                    if 0 <= nrow < 8 and 0 <= ncol < 8: # Verificar
                        lmites del tablero
                        captures['black'].add(chr(97 + ncol) + str(8 -
                            nrow))
    captures_white = sorted(list(captures['white'])) # Ordenar
        alfabeticamente
    captures_black = sorted(list(captures['black']))

    return captures_white if turn else captures_black

def count_pawn_capture_squares(fen, turn:bool = True):
    """Cuenta las casillas controladas por peon (puede comer en
        siguiente turno)

    Args:
        fen (str): Cadena Fen
        turn (bool, optional): True: Blancas. False: Negras

    Returns:
        int: Cuenta las casillas controladas por peon (puede comer
            en siguiente turno)
    """
    return len(get_pawn_capture_squares(fen, turn))

def get_pressure_points_san(fen, turn = None):
    """
    Devuelve una lista de las casillas que son puntos de presin en
        notacin SAN.
    Un punto de presin se define como una casilla que es atacada
        por ms de una pieza
    del jugador cuyo turno es actualmente.

    Args:
```

```
    fen (str): La cadena FEN que representa la posicin actual del
        tablero.

    Returns:
    list: Lista de las casillas de puntos de presin en notacin SAN.
    """
    board = get_custom_board(fen, turn)
    current_turn = board.turn # TRUE (WHITE) FALSE (BLACK)
    attack_map = {square: 0 for square in SQUARES}

    # Iterar solo sobre las piezas del color que tiene el turno
    for piece_type in PIECE_TYPES:
       for square in board.pieces(piece_type, current_turn):
          attacked_squares = board.attacks(square)
          for attacked_square in attacked_squares:
             attacked_piece = board.piece_at(attacked_square)
             # Incrementar si la casilla atacada est vaca o tiene
                 una pieza del color opuesto
             if not attacked_piece or attacked_piece.color !=
                 current_turn:
                attack_map[attacked_square] += 1

    # Los puntos de presin son cuadrados a los que ms de una pieza
        ataca
    pressure_points = [square_name(sq) for sq, count in
        attack_map.items() if count > 1]

    return sorted(pressure_points)

def count_pressure_points(fen, turn = None):
    """
    Cuenta las casillas que son puntos de presin.
    Un punto de presin se define como una casilla que es atacada
        por ms de una pieza
    del jugador cuyo turno es actualmente.

    Args:
    fen (str): La cadena FEN que representa la posicin actual del
        tablero.

    Returns:
    int: Conteo de las casillas de puntos de presin.
    """
    board = get_custom_board(fen, turn)
    current_turn = board.turn # TRUE (WHITE) FALSE (BLACK)
    attack_map = {square: 0 for square in SQUARES}

    # Iterar solo sobre las piezas del color que tiene el turno
    for piece_type in PIECE_TYPES:
       for square in board.pieces(piece_type, current_turn):
          attacked_squares = board.attacks(square)
```

```
            for attacked_square in attacked_squares:
                attacked_piece = board.piece_at(attacked_square)
                # Incrementar si la casilla atacada est vaca o tiene
                    una pieza del color opuesto
                if not attacked_piece or attacked_piece.color !=
                    current_turn:
                    attack_map[attacked_square] += 1

    # Los puntos de presin son cuadrados a los que ms de una pieza
        ataca
    pressure_points = [1 for _, count in attack_map.items() if
        count > 1]

    return sum(pressure_points)

def get_all_controlled_diagonals(fen, turn = None):
    """
    Obtiene el nmero de diagonales controladas por alfiles o reinas
        en un tablero de ajedrez,
    dado por la notacin FEN.
    Una diagonal est controlada si al menos dos casillas estn
        libres o contienen una
    pieza del color opuesto.
    """
    # Crear un tablero a partir de la notacin FEN
    tablero = get_custom_board(fen, turn)
    current_turn = tablero.turn # TRUE (WHITE) FALSE (BLACK)
    diagonales_controladas = {}
    diagonales_controladas_sum = 0

    # Por cada casilla entre las 64 posibles
    for piece_type in BISHOP, QUEEN:
        for square in tablero.pieces(piece_type, current_turn):
            # Sumar diagonales controladas de alfiles y reinas
            result = get_controlled_diagonals_by_square(square,
                tablero)
            diagonales_controladas[square_name(square)] = result
            diagonales_controladas_sum += result

    return diagonales_controladas

def count_all_controlled_diagonals(fen, turn = None):
    """
    Cuenta el nmero de diagonales controladas por alfiles o reinas
        en un tablero de ajedrez,
    dado por la notacin FEN.
    Una diagonal est controlada si al menos dos casillas estn
        libres o contienen una
    pieza del color opuesto.
    """
    # Crear un tablero a partir de la notacin FEN
```

```
    tablero = get_custom_board(fen, turn)
    current_turn = tablero.turn # TRUE (WHITE) FALSE (BLACK)
    diagonales_controladas_sum = 0

    # Por cada casilla entre las 64 posibles
    for piece_type in BISHOP, QUEEN:
        for square in tablero.pieces(piece_type, current_turn):
            # Sumar diagonales controladas de alfiles y reinas
            result = get_controlled_diagonals_by_square(square,
                tablero)
            diagonales_controladas_sum += result

    return diagonales_controladas_sum

def get_controlled_diagonals_by_square(casilla, board: Board):
    # Direcciones de las diagonales: superior izquierda, superior
        derecha, inferior izquierda, inferior derecha
    direcciones = [(-1, -1), (-1, 1), (1, -1), (1, 1)]
    controlled_diagonals = 0
    # Por cada diagonal en las direcciones
    for dx, dy in direcciones:
        # Verificar las dos primeras casillas en cada direccin
            diagonal
        for i in range(1, 3):
            nueva_fila, nueva_columna = square_rank(casilla) + i *
                dx, square_file(casilla) + i * dy
            # Si alguna de las dos casillas est fuera del tablero, la
                diagonal no esta controlada o es inutil
            if not (0 <= nueva_fila <= 7 and 0 <= nueva_columna <= 7):
                break
            nueva_casilla = square(nueva_columna, nueva_fila)
            pieza_original = board.piece_at(casilla)
            pieza_en_diagonal = board.piece_at(nueva_casilla)

            # Si alguna de las dos casillas tiene una pieza del mismo
                color, la diagonal no est controlada o no es util
            if pieza_en_diagonal and pieza_en_diagonal.color ==
                pieza_original.color:
                break
        else:
            controlled_diagonals += 1
    return controlled_diagonals

def get_all_controlled_lines(fen, turn = None):
    """
    Cuenta el nmero de lneas horizontales y verticales controladas
        por torres y reina en un tablero de ajedrez,
    dada una determinada posicin FEN
    """
    # Crear un tablero a partir de la notacin FEN
    tablero = get_custom_board(fen, turn)
```

```
    current_turn = tablero.turn # TRUE (WHITE) FALSE (BLACK)
    lineas_controladas = {}
    lineas_controladas_sum = 0

    # Por cada torre del jugador actual
    for piece_type in ROOK, QUEEN:
        for square in tablero.pieces(piece_type, current_turn):
            # Sumar lneas horizontales y verticales controladas por
                la pieza
            result = get_controlled_lines_by_square(square, tablero)
            lineas_controladas[square_name(square)] = result
            lineas_controladas_sum += result

    return lineas_controladas

def count_all_controlled_lines(fen, turn = None):
    """
    Cuenta el nmero de lneas horizontales y verticales controladas
        por torres y reina en un tablero de ajedrez,
    dada una determinada posicin FEN
    """
    # Crear un tablero a partir de la notacin FEN
    tablero = get_custom_board(fen, turn)
    current_turn = tablero.turn # TRUE (WHITE) FALSE (BLACK)
    lineas_controladas_sum = 0

    # Por cada torre del jugador actual
    for piece_type in ROOK, QUEEN:
        for square in tablero.pieces(piece_type, current_turn):
            # Sumar lneas horizontales y verticales controladas por
                la pieza
            result = get_controlled_lines_by_square(square, tablero)
            lineas_controladas_sum += result

    return lineas_controladas_sum

def get_controlled_lines_by_square(casilla, board: Board):
    """
    Cuenta el numero de lineas controladas por determinada casilla
    Una linea est controlada si al menos dos casillas contiguas a
        la pieza estn libres o contienen una
    pieza del color opuesto.
    """
    # Direcciones: izquierda, derecha, arriba, abajo
    direcciones = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    controlled_lines = 0
    # Por cada direccin
    for dx, dy in direcciones:
        # Verificar las dos primeras casillas en cada direccin
        for i in range(1, 3):
```

```
            nueva_fila, nueva_columna = square_rank(casilla) + i *
                dy, square_file(casilla) + i * dx
            # Si alguna de las dos casillas est fuera del tablero, la
                lnea no est controlada o es intil
            if not (0 <= nueva_fila <= 7 and 0 <= nueva_columna <= 7):
               break
            nueva_casilla = square(nueva_columna, nueva_fila)
            pieza_original = board.piece_at(casilla)
            pieza_en_linea = board.piece_at(nueva_casilla)

            # Si alguna de las dos casillas tiene una pieza del mismo
                color, la lnea no est controlada o no es til
            if pieza_en_linea and pieza_en_linea.color ==
                pieza_original.color:
               break
         else:
            controlled_lines += 1
      return controlled_lines

def get_fen_from_moves(moves):
   game = Board()
   for move in moves.split():
      game.push_san(move)

   # Get FEN string of position
   fen = game.fen()
   return fen

def get_current_turn(fen):
   # Crear un tablero a partir de la notacin FEN
   tablero = Board(fen)
   current_turn = tablero.turn # TRUE (WHITE) FALSE (BLACK)
   return current_turn

def get_current_opening(df,turn_column_name, fen, opening_move):
   """
   Filtra el DataFrame proporcionado y retorna el opening
       producido del turno.

   Parmetros:
   - df: DataFrame a filtrar.
   - turn_column_name: Name of column, last played
   - opening_move: Entero, nmero de jugada, (2 jugadas por turno).

   Retorna:
   - Nombre de la apertura alcanzada
   """
   if df.empty:
      return None
   turn_column_name = f"{turn_column_name}_fen"
```

```
    # Verificar si la columna generada existe en el DataFrame
        filtrado
    if turn_column_name not in df.columns:
        return None

    df_filter = df[(df[turn_column_name]==fen) &
        (df['opening_moves']==opening_move)]
    if df_filter.empty:
        return None
    return df_filter['opening_fullname'].iloc[0]


def count_all_features(fen:str, moves:int) -> pd.DataFrame:
    '''
    Obtiene todas las features utilizadas para una prediccin
    # fen = board.fen()
    # moves = len(board.move_stack) or board.ply()
    '''
    PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING = range(1, 7)

    df = pd.DataFrame({
        'turns': moves,
        'w_ctrld_pawn':     count_pawn_capture_squares(fen, True),
        'w_ctrld_knight':   count_legal_moves_from_piece_type(fen,
            KNIGHT, True),
        'w_ctrld_bishop':   count_legal_moves_from_piece_type(fen,
            BISHOP, True),
        'w_ctrld_rook':     count_legal_moves_from_piece_type(fen,
            ROOK, True),
        'w_ctrld_queen':    count_legal_moves_from_piece_type(fen,
            QUEEN, True),
        'w_ctrld_king':     count_legal_moves_from_piece_type(fen,
            KING, True),
        'w_preassure_points': count_pressure_points(fen, True),
        'w_ctrld_diagonals': count_all_controlled_diagonals(fen,
            True),
        'w_ctrld_lines': count_all_controlled_lines(fen, True),
        'b_ctrld_pawn':     count_pawn_capture_squares(fen, False),
        'b_ctrld_knight':   count_legal_moves_from_piece_type(fen,
            KNIGHT, False),
        'b_ctrld_bishop':   count_legal_moves_from_piece_type(fen,
            BISHOP, False),
        'b_ctrld_rook':     count_legal_moves_from_piece_type(fen,
            ROOK, False),
        'b_ctrld_queen':    count_legal_moves_from_piece_type(fen,
            QUEEN, False),
        'b_ctrld_king':     count_legal_moves_from_piece_type(fen,
            KING, False),
        'b_preassure_points': count_pressure_points(fen, False),
        'b_ctrld_diagonals': count_all_controlled_diagonals(fen,
            False),
```

```
        'b_ctrld_lines': count_all_controlled_lines(fen, False)
        }, index=[0])
    return df

def get_all_features(fen:str, moves:int) -> pd.DataFrame:
    '''
    Obtiene todas las features utilizadas para una prediccin
    # fen = board.fen()
    # moves = len(board.move_stack) or board.ply()
    '''
    PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING = range(1, 7)

    df = pd.DataFrame({
        'turns': moves,
        'w_ctrld_pawn': str(get_pawn_capture_squares(fen, True)),
        'w_ctrld_knight': str(get_legal_moves_from_piece_type(fen,
            KNIGHT, True)),
        'w_ctrld_bishop': str(get_legal_moves_from_piece_type(fen,
            BISHOP, True)),
        'w_ctrld_rook': str(get_legal_moves_from_piece_type(fen,
            ROOK, True)),
        'w_ctrld_queen': str(get_legal_moves_from_piece_type(fen,
            QUEEN, True)),
        'w_ctrld_king': str(get_legal_moves_from_piece_type(fen,
            KING, True)),
        'w_preassure_points': str(get_pressure_points_san(fen,
            True)),
        'w_ctrld_diagonals': str(get_all_controlled_diagonals(fen,
            True)),
        'w_ctrld_lines': str(get_all_controlled_lines(fen, True)),
        'b_ctrld_pawn': str(get_pawn_capture_squares(fen, False)),
        'b_ctrld_knight': str(get_legal_moves_from_piece_type(fen,
            KNIGHT, False)),
        'b_ctrld_bishop': str(get_legal_moves_from_piece_type(fen,
            BISHOP, False)),
        'b_ctrld_rook': str(get_legal_moves_from_piece_type(fen,
            ROOK, False)),
        'b_ctrld_queen': str(get_legal_moves_from_piece_type(fen,
            QUEEN, False)),
        'b_ctrld_king': str(get_legal_moves_from_piece_type(fen,
            KING, False)),
        'b_preassure_points': str(get_pressure_points_san(fen,
            False)),
        'b_ctrld_diagonals': str(get_all_controlled_diagonals(fen,
            False)),
        'b_ctrld_lines': str(get_all_controlled_lines(fen, False))
        }, index=[0])
    return df

def get_all_features_uf(fen: str, turn: bool) -> pd.DataFrame:
    '''
```

```
    Obtiene todas las features utilizadas para una prediccin
    # turn: TRUE (WHITE) FALSE (BLACK)
    '''
    # Caractersticas a obtener
    ctrld_pawn = get_pawn_capture_squares(fen, turn)
    ctrld_knight = get_legal_moves_from_piece_type(fen, KNIGHT,
        turn)
    ctrld_bishop = get_legal_moves_from_piece_type(fen, BISHOP,
        turn)
    ctrld_rook = get_legal_moves_from_piece_type(fen, ROOK, turn)
    ctrld_queen = get_legal_moves_from_piece_type(fen, QUEEN, turn)
    ctrld_king = get_legal_moves_from_piece_type(fen, KING, turn)
    preassure_points = get_pressure_points_san(fen, turn)
    ctrld_diagonals = get_all_controlled_diagonals(fen, turn)
    ctrld_lines = get_all_controlled_lines(fen, turn)

    features = [
       ('Casillas controladas por peon',ctrld_pawn,
           len(ctrld_pawn)),
       ('Casillas controladas por caballo',ctrld_knight,
           len(ctrld_knight)),
       ('Casillas controladas por alfil', ctrld_bishop,
           len(ctrld_bishop)),
       ('Casillas controladas por torre', ctrld_rook,
           len(ctrld_rook)),
       ('Casillas controladas por reina', ctrld_queen,
           len(ctrld_queen)),
       ('Casillas controladas por el rey', ctrld_king,
           len(ctrld_king)),
       ('Puntos de presion (>1 atacando)', preassure_points,
           len(preassure_points)),
       ('Diagonales controladas', ctrld_diagonals,
           sum(ctrld_diagonals.values())),
       ('Lineas controladas', ctrld_lines,
           sum(ctrld_lines.values())),
    ]

    # Lista para almacenar las filas del DataFrame
    rows = []

    # Iterar sobre cada caracterstica para ambos colores
    for feature_name, feature_values, total in features:
       # total = len(feature_values)

       # Aadir fila al DataFrame
       rows.append({
          'Total': total,
          'Caracteristica': feature_name,
          'Casillas': str(feature_values)
       })
```

```
    return pd.DataFrame(rows)
```

4.7 *feature_engineering.py*

```
import pandas as pd
import extract_features as ef
from chess import Board
from chess import PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING,
    PIECE_NAMES
import numpy as np
data_path = "../CHESS/data/df_1.csv"
df_1 = pd.read_csv(data_path)
df_1.info()

# aplicando la funcin get_fen_from_moves a todas las filas de la
    columna moves
df_1['moves_fen'] = df_1['moves'].apply(ef.get_fen_from_moves)

# Aplicamos la funcin get_current_turn en todas las observaciones
    y generamos la nueva columna
df_1["current_turn"] = df_1['moves_fen'].apply(ef.get_current_turn)

# Aplicar funciones de caractersticas tanto a blancas como a negras
for color in ['w', 'b']:
   turn = True if color == 'w' else False
   df_1[f'{color}_ctrld_pawn'] =  df_1['moves_fen'].apply(lambda x:
       ef.count_pawn_capture_squares(x, turn))
   df_1[f'{color}_ctrld_knight'] = df_1['moves_fen'].apply(lambda
       x: ef.count_legal_moves_from_piece_type(x, KNIGHT, turn))
   df_1[f'{color}_ctrld_bishop'] = df_1['moves_fen'].apply(lambda
       x: ef.count_legal_moves_from_piece_type(x, BISHOP, turn))
   df_1[f'{color}_ctrld_rook'] =  df_1['moves_fen'].apply(lambda x:
       ef.count_legal_moves_from_piece_type(x, ROOK, turn))
   df_1[f'{color}_ctrld_queen'] = df_1['moves_fen'].apply(lambda
       x: ef.count_legal_moves_from_piece_type(x, QUEEN, turn))
   df_1[f'{color}_ctrld_king'] =  df_1['moves_fen'].apply(lambda x:
       ef.count_legal_moves_from_piece_type(x, KING, turn))
   df_1[f'{color}_preassure_points'] =
       df_1['moves_fen'].apply(lambda x:
       ef.count_pressure_points(x, turn))
   df_1[f'{color}_ctrld_diagonals'] =
       df_1['moves_fen'].apply(lambda x:
       ef.count_all_controlled_diagonals(x, turn))
   df_1[f'{color}_ctrld_lines'] = df_1['moves_fen'].apply(lambda
       x: ef.count_all_controlled_lines(x, turn))

# Guardar el CSV con features como df_2.csv
df_1.to_csv("../CHESS/data/df_3.csv", index=False) # Data frame
    con features basadas en ventaja posicional del turno
```

```
# Validacin
muestra_aleatoria = np.random.randint(0, df_1.shape[0])

fen = df_1['moves_fen'][muestra_aleatoria]
print(f'FEN: {fen}')
print(f"w_ctrld_pawn: {df_1['w_ctrld_pawn'][muestra_aleatoria]}")
print(f"w_ctrld_knight:
    {df_1['w_ctrld_knight'][muestra_aleatoria]}")
print(f"w_ctrld_bishop:
    {df_1['w_ctrld_bishop'][muestra_aleatoria]}")
print(f"w_ctrld_rook: {df_1['w_ctrld_rook'][muestra_aleatoria]}")
print(f"w_ctrld_queen: {df_1['w_ctrld_queen'][muestra_aleatoria]}")
print(f"w_ctrld_king: {df_1['w_ctrld_king'][muestra_aleatoria]}")
print(f"w_preassure_points:
    {df_1['w_preassure_points'][muestra_aleatoria]}")
print(f"w_ctrld_diagonals:
    {df_1['w_ctrld_diagonals'][muestra_aleatoria]}")
print(f"w_ctrld_lines: {df_1['w_ctrld_lines'][muestra_aleatoria]}")

# Validar contra la aplicacion de la funcion count all features
print(f'FEN: {fen}')
print(ef.count_all_features(fen, 'n').T)

print(f'FEN: {fen}')
pd.set_option('display.max_colwidth', None)
print(ef.get_all_features(fen, 'n').T)
```

4.8 *generate_openings_dataset.py*

```
import pandas as pd
org_data_path = "../CHESS/data/df_2_just_moves.csv"
df_2 = pd.read_csv(org_data_path)
df_2 = df_2.drop(columns=['moves_fen'])

from extract_features import get_fen_from_moves

def create_fen_columns(df_moves):
   '''
   Se aaden columnas para cada movimiento de apertura (de 1 a 28
       mximo) en formato fen al df_moves proporcionado
   '''
   max_opening_moves = 28//2 # Nmero mximo de turnos de apertura
       (1 turno = 2 movimientos)

   # Crear las columnas para cada movimiento de apertura
   for turn_num in range(1, max_opening_moves + 1):
      # Rellenar con None donde no hay movimientos
```

```
        df_moves[f'0w_{turn_num}'] = None
        df_moves[f'0w_{turn_num}_fen'] = None
        df_moves[f'1b_{turn_num}'] = None
        df_moves[f'1b_{turn_num}_fen'] = None

    for index, row in df_moves.iterrows():
        moves = row['moves'].split()
        opening_moves = int(row['opening_moves'])

        if opening_moves % 2 == 0:
            opening_turns = opening_moves//2
        else:
            opening_turns = (opening_moves//2) + 1

        for turn_num in range(1, opening_turns + 1):
            san_white = moves[:turn_num*2-1]
            san_black = moves[:turn_num*2]
            fen_white = get_fen_from_moves(' '.join(san_white))
            fen_black = get_fen_from_moves(' '.join(san_black))
            df_moves.at[index, f'0w_{turn_num}'] = san_white[-1]
            df_moves.at[index, f'0w_{turn_num}_fen'] = fen_white

            # No rellenar el ultimo movimiento de negras cuando es el
                ultimo turno pero movimientos son impares
            if opening_moves % 2 != 0 and turn_num == opening_turns:
                break

            df_moves.at[index, f'1b_{turn_num}'] = san_black[-1]
            df_moves.at[index, f'1b_{turn_num}_fen'] = fen_black

# Aplicar funcin para crear n columnas como turnos existan para
    generar una apertura
create_fen_columns(df_2)
df_2 = df_2.drop(columns=['moves', 'game_id'])
df_2.columns

# Guardar el CSV con features como df_2.csv
df_2.to_csv("../CHESS/data/df_2_just_moves_fen.csv", index=False)
    # Data frame con features basadas en ventaja posicional del
    turno
```

4.9 *model_helpers.py*

```
# Libreras
from sklearn.metrics import (accuracy_score, precision_score,
    recall_score)
from sklearn.metrics import (brier_score_loss, roc_auc_score,
    confusion_matrix)
from sklearn.metrics import auc as func_auc
```

```
from sklearn.metrics import precision_recall_curve, roc_curve
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import numpy as np
from sklearn.preprocessing import LabelBinarizer


warnings.filterwarnings('ignore')

def one_hot_encode_labels(y):
   lb = LabelBinarizer()
   y_binarized = lb.fit_transform(y)
   # Asegurarse de que y_binarized tiene dos columnas si es un
       problema binario
   if len(lb.classes_) == 2:
      y_binarized = np.hstack([1 - y_binarized, y_binarized])
   return y_binarized

def eval_perform(Y,y_pred = None, y_prob = None, train_or_test:
    str = None, print_results = True):
   """Evala el performance de cada modelo

   Args:
      Y (array): Variable objetivo original
      y_pred (array): Salida Y & "hat" () que denota predicciones
          estimadas.
      y_prob (array): Salida Y & "hat" () que denota
          probabilidades estimadas.
      train_or_test (string): "Entrenamiento" o "Prueba"
      print_results(bool): True if is desired to print the results.
   """
   accu, prec, reca, brier_score, auc, cfm, pr_auc = None, None,
       None, None, None, None, None
   if print_results:
         print(f"\nPerformance del modelo de {train_or_test}")
   if y_pred is not None:
      accu = accuracy_score(Y,y_pred)
      prec = precision_score(Y,y_pred,average='weighted')
      reca = recall_score(Y,y_pred,average='weighted')
      cfm = confusion_matrix(Y, y_pred)
      if print_results:
         print(f' Accu {accu} \n Prec {prec} \n Reca {reca} \n
             Confusin matrix:\n {cfm}')

   if y_prob is not None:
      print(f"\nMtricas de Probabilidad:")
      # Calcular el Brier Score
      brier_score = brier_score_loss(Y, y_prob)
      auc = roc_auc_score(Y, y_prob)
      precision, recall, __ = precision_recall_curve(Y, y_prob)
```

```
        pr_auc = func_auc(recall, precision)
        if print_results:
            print(f' Brier Score: {brier_score} \n AUC {auc:.4f} \n
                PR AUC {pr_auc}')

    return accu, prec, reca, brier_score, auc, cfm, pr_auc


def eval_perform_multi_class(Y, y_pred=None, y_prob=None,
     class_names = None, model_name: str=None, print_results=True):
    """Evala el rendimiento de cada modelo para clasificacin
        multiclase.

    Args:
        Y (array): Variable objetivo original.
        y_pred (array): Salida Y & "hat" () que denota predicciones
            estimadas.
        y_prob (array): Salida Y & "hat" () que denota
            probabilidades estimadas.
        model_name (string): Nombre del modelo: ejemplo
            "Entrenamiento" o "Prueba".
        print_results(bool): True si se desean imprimir los
            resultados.
    """
    accu, prec, reca, brier_score, cfm = None, None, None, None,
        None
    if print_results:
        print(f"\nPerformance del modelo de {model_name}")
    if y_pred is not None:
        accu = accuracy_score(Y, y_pred)
        prec = precision_score(Y, y_pred, average='weighted')
        reca = recall_score(Y, y_pred, average='weighted')
        cfm = confusion_matrix(Y, y_pred)
        if print_results:
            plot_confusion_matrix(cfm, model_name, class_names)
            print(f' Accu {accu} \n Prec {prec} \n Reca {reca}')

    if y_prob is not None and y_prob.shape[1] > 1:
        print("\nMtricas de Probabilidad:")
        brier_score = np.mean([brier_score_loss(Y == i, y_prob[:,
            i]) for i in range(y_prob.shape[1])])
        if print_results:
            print(f" Brier Score: {brier_score}")
            n_classes = len(class_names)
            plot_evaluation_curves(n_classes, Y, y_prob, model_name)


    return accu, prec, reca, brier_score, cfm

def plot_confusion_matrix(confusion_matrix, model_name,
    class_names):
```

```
    # Create a heatmap to visualize the confusion matrix
    plt.figure(figsize=(5, 3))
    sns.heatmap(confusion_matrix, annot=True, fmt="d",
        cmap='Blues', xticklabels=class_names,
        yticklabels=class_names)

    # Adding labels and title
    plt.xlabel('Clases predichas')
    plt.ylabel('Clases verdaderas')
    plt.title(f'Matriz de confusin {model_name}')
    plt.show()

def plot_evaluation_curves(n_classes, y, y_prob, model_name):
    # One-hot encode labels if they are not already
    y = one_hot_encode_labels(y)

    # Prepare figure
    plt.figure(figsize=(16, 6))

    # Plot Precision-Recall Curve
    plt.subplot(1, 2, 1)
    precision = dict()
    recall = dict()
    for i in range(n_classes):
        precision[i], recall[i], _ = precision_recall_curve(y[:, i],
            y_prob[:, i])
        plt.plot(recall[i], precision[i], lw=2, label=f'Class {i}')
    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.legend(loc="best")
    plt.title(f"Precision vs. Recall Curve for {model_name}")

    # Plot ROC Curve
    plt.subplot(1, 2, 2)
    fpr = dict()
    tpr = dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y[:, i], y_prob[:, i])
        plt.plot(fpr[i], tpr[i], lw=2, label=f'Class {i}')
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.legend(loc="best")
    plt.title(f"ROC Curve for {model_name}")

    # Show plots
    plt.tight_layout()
    plt.show()


def show_target_balance(y):
```

```
    """Shows the balance of a class in a given dataset with target
        variable only

    Args:
       y (Dataframe): Pandas dataframe with target variable only
    """
    conteo_clases = y.value_counts()

    # Calcular el porcentaje de cada clase
    total = len(y)

    # Establecer el estilo del grfico
    sns.set_theme(style="whitegrid")

    # Crear el grfico de barras
    plt.figure(figsize=(10, 6))
    barplot = sns.barplot(x=conteo_clases.index,
        y=conteo_clases.values)

    # Aadir etiquetas con el nmero y porcentaje de cada clase
    for p in barplot.patches:
       altura = p.get_height()
       barplot.text(p.get_x() + p.get_width()/2., altura + 0.1,
           f'{int(altura)}\n({altura/total:.2%})', ha="center")

    # Aadir ttulos y etiquetas
    plt.title('Distribucin de Clases')
    plt.xlabel('Clase')
    plt.ylabel('Nmero de Instancias')

    # Mostrar el grfico
    plt.show()
```

4.10 *model_tuning.py*

```
# Libreras
import pandas as pd
from sklearn.model_selection import train_test_split
# Warnings
import warnings
warnings.filterwarnings('ignore')
# Internal tool and helpers
import model_helpers as mh
import xgboost as xgb
from hyperopt import fmin, tpe, hp, STATUS_OK, Trials
from sklearn.metrics import accuracy_score
import numpy as np
import matplotlib.pyplot as plt
CLASS_NAMES = ["Negras", "Blancas"]
```

```
data_path = "../CHESS/data/df_3_cod.csv"
df_3 = pd.read_csv(data_path)

# X e Y
X = df_3.copy()
y_name = "winner_cod"
# X es el dataframe eliminando la variable de salida. Eliminando
    tambin 'moves' que ya est representado
X = X.drop(columns=['rated','winner',y_name,
    'current_turn','time_increment','opening_code','opening_fullname','opening_shortname','opening_variation','moves_
X = X.drop(columns=['game_id', 'white_rating', 'black_rating',
    'moves', 'current_turn_cod', 'opening_moves', 'rated_cod',
    'current_turn_cod', 'time_increment_cod', 'opening_code_cod',
    'opening_fullname_cod', 'opening_shortname_cod',
    'opening_variation_cod', 'moves_fen_cod'])# Y es un array
    unidimensional (ravel) de la variable de salida
Y = df_3[y_name].ravel()
print(X.columns)
# divisin en train y test
X_train, X_test, Y_train, Y_test = train_test_split(X,
    Y,test_size=0.3)

# XGBOOST Simple
# Crear un clasificador XGBoost
xgboost = xgb.XGBClassifier()
# Entrenar el modelo en los datos de entrenamiento
xgboost.fit(X_train, Y_train)
# Evaluacin del modelo
Yhat_xgboost_test = xgboost.predict(X_test)
Yhat_xgboost_train = xgboost.predict(X_train)
Yhat_xgboost_test_prob = xgboost.predict_proba(X_test)
Yhat_xgboost_train_prob = xgboost.predict_proba(X_train)
mh.eval_perform_multi_class(Y_test,Yhat_xgboost_test,
    Yhat_xgboost_test_prob,CLASS_NAMES,"Prueba Xgboost blancas y
    negras modelo sencillo")
mh.eval_perform_multi_class(Y_train,Yhat_xgboost_train,
    Yhat_xgboost_train_prob,CLASS_NAMES,"Entrenamiento Xgboost
    blancas y negras modelo sencillo")

# Optimizacin de hiperparmetros XGboost
# Define the hyperparameter space
space = {
   'max_depth': hp.choice('max_depth', np.arange(1, 14,
       dtype=int)),
   'learning_rate': hp.loguniform('learning_rate', -5, -2),
   'subsample': hp.uniform('subsample', 0.5, 1)
}

# Define the objective function to minimize
def objective(params):
```

```
    xgb_model = xgb.XGBClassifier(**params)
    xgb_model.fit(X_train, Y_train)
    y_pred = xgb_model.predict(X_test)
    score = accuracy_score(Y_test, y_pred)
    return {'loss': -score, 'status': STATUS_OK}

trials = Trials()
# Perform the optimization
best_params = fmin(objective, space, algo=tpe.suggest,
     max_evals=250, trials=trials)
print("Best set of hyperparameters: ", best_params)
# Extraer la funcin de prdida de cada resultado en trials
losses = [x['result']['loss'] for x in trials.trials]

# Graficar la funcin de prdida
plt.figure(figsize=(10, 5))
plt.plot(losses, label='Funcin de Prdida')
plt.xlabel('Iteraciones')
plt.ylabel('Prdida Negativa de Exactitud')
plt.title('Funcin de Prdida durante la Optimizacin de
     Hiperparmetros')
plt.legend()
plt.show()

# Crear un clasificador XGBoost a partir de los mejores parmetros
xgboost_best = xgb.XGBClassifier(**best_params)
# Entrenar el modelo en los datos de entrenamiento
xgboost_best.fit(X_train, Y_train)
# Evaluacin del modelo
Yhat_xgboost_test = xgboost_best.predict(X_test)
Yhat_xgboost_train = xgboost_best.predict(X_train)
Yhat_xgboost_test_prob = xgboost_best.predict_proba(X_test)
Yhat_xgboost_train_prob = xgboost_best.predict_proba(X_train)
mh.eval_perform_multi_class(Y_test,Yhat_xgboost_test,
     Yhat_xgboost_test_prob,CLASS_NAMES,"Prueba Xgboost blancas y
     negras optimizacin de hiperparmetros")
mh.eval_perform_multi_class(Y_train,Yhat_xgboost_train,
     Yhat_xgboost_train_prob,CLASS_NAMES,"Entrenamiento Xgboost
     blancas y negras optimizacin de hiperparmetros")

# Feature importance
import matplotlib.pyplot as plt
# Feature importance
importance = xgboost.feature_importances_

df_importancia = pd.DataFrame({
   'Caracteristica': X.columns,
   'Importancia': importance
})

umbral = 0
```

```
# Filtrar los nombres de las caractersticas importantes
df_importancia = df_importancia[df_importancia['Importancia'] >
    umbral]
df_importancia = df_importancia.sort_values(by='Importancia',
    ascending=True)

features = df_importancia['Caracteristica']
importance = df_importancia['Importancia']


# Crear un grfico de barras horizontal
plt.figure(figsize=(10, 8))
plt.barh(features, importance, color='skyblue')
plt.xlabel('Importance')
plt.ylabel('Features')
plt.title('Feature importance')
plt.show()


# Validacin del modelo
import extract_features as ef
fen = '1r1q1rk1/pb2bppp/4p3/6N1/2pPQ3/4P2P/PP3PP1/R1B2RK1 w - - 1
    17'
moves = 4
new_sample_df = ef.count_all_features(fen, moves)
print(new_sample_df.T)

print("Posicin Blancas: ")
pd.set_option('display.max_columns', None)
print(ef.get_all_features_uf(fen, True))

print("Posicin Negras: ")
pd.set_option('display.max_columns', None)
print(ef.get_all_features_uf(fen, False))


# Decisin del modelo para clasificacin
prob = xgboost.predict_proba(new_sample_df)
print(xgboost.classes_) # 0: negras. 1: blancas
print(f"Negras: {'{:.6f}'.format(prob[0][0])}.
    Blancas:{'{:.6f}'.format(prob[0][1])}")
prob = xgboost_best.predict_proba(new_sample_df)
print(xgboost_best.classes_) # 0: negras. 1: blancas
print(f"Negras: {'{:.6f}'.format(prob[0][0])}.
    Blancas:{'{:.6f}'.format(prob[0][1])}")


'''
Guardar modelo elegido
En un entorno de produccin tiene menos costo computacional
    seleccionar el modelo ms sencillo para entrenar con nuevos
```

```
    datos, y la variacin es despreciable. Por tanto se selecciona
    ese modelo.
'''
import pickle
pickle.dump(xgboost,
    open("./pickles/models/xgoboost_model0416_18:06.pkl", 'wb'))
```

4.11 *openings_ialex.py*

```
import chess
import chess.engine
from chess import Board
from chessboard import display
from extract_features import get_current_opening,
    count_all_features
from extract_features import get_all_features_uf
import pandas as pd
from utils import ChessLogger
import numpy as np
import random

class OpeningsIA:
   def __init__(self, engine, random = False, opening_shortname =
       'Sicilian Defense', b_or_w_input = 'b', logs_path =
       "./logs") -> None:
      self.restart_game = False
      self.engine = engine
      self.logger = ChessLogger(logs_path)
      self.df_hist_moves = self.load_historical_games()
      if random:
         b_or_w_input = self.select_random_color()
         opening_shortname = self.select_random_opening()
      self.play_game = self.play_game_function(b_or_w_input)
      self.opening_shortname = opening_shortname

      self.df_hist_moves_filter =
          self.filter_possible_moves_by_opening()

   def stockfish_move(self, board, displayed_board):
      board_analysis = self.engine.analyse(board,
          limit=chess.engine.Limit(time=1))
      best_move = board_analysis.get("pv")[0]
      san_move = board.san(best_move)
      board.push(best_move)
      fen_robot = board.fen()
      display.check_for_quit()
      display.update(fen_robot, displayed_board)
      return san_move
```

```
    def manual_move(self, move_input:str, board:Board,
        displayed_board):
       try:
          # Salir si el usuario enva x
          if move_input == "x":
             display.terminate()
          if move_input == 'z':
             self.restart_game = True
             return
          # push movement
          board.push_san(move_input)
          # Get fen from board
          fen_human = board.fen()
          display.check_for_quit()
          # Show
          display.update(fen_human, displayed_board)
          return move_input
       except chess.InvalidMoveError as e:
          print(f"Entrada no vlida. Utiliza notacin SAN (por
              ejemplo, 'e3').\n{e}")
       except chess.IllegalMoveError as e:
          print(f"Movimiento Ilegal. Intente de nuevo\n{e}")
       except chess.AmbiguousMoveError as e:
          print(f"Movimiento ambiguo, intente ser ms especfico. Por
              ejemplo especificar la letra origen (Rad1)\n{e}")
       except ValueError as e:
          print(f"Error inesperado.\n{e}")


    def run_human_white_board(self, board, displayed_board,
        unique_opening_moves):
       if board.turn == chess.WHITE:
          return self.run_human_movement(board, displayed_board,
              unique_opening_moves)
       else:
          return self.run_machine_movement(board, displayed_board,
              unique_opening_moves)

    def run_human_black_board(self, board, displayed_board,
        unique_opening_moves):
       if board.turn == chess.BLACK:
          self.flip_board_if_needed(displayed_board)
          return self.run_human_movement(board, displayed_board,
              unique_opening_moves)
       else:
          return self.run_machine_movement(board, displayed_board,
              unique_opening_moves)

    def run_human_movement(self, board, displayed_board,
        unique_opening_moves):
       print("================HUMAN===============")
```

```
        self.logger.write("================HUMAN================")
        # TODO: Ventajas posicionales (A partir del movimiento 3)
        if unique_opening_moves:
            self.logger.write(f"Movimientos comunes
                {self.opening_shortname}: {[move[0] for move in
                unique_opening_moves]}.")
            print(f"Movimientos comunes {self.opening_shortname}:
                {[move[0] for move in unique_opening_moves]}.")
        move_input = input("Tu movimiento: ")
        self.logger.write(f"Humano mueve: {move_input}")
        return self.manual_move(move_input, board, displayed_board)

    def run_machine_movement(self,board, displayed_board,
         unique_opening_moves):
        print(("================Openings IA================"))
        self.logger.write("================Openings
             IA================")
        if unique_opening_moves:
            self.logger.write(f"Movimientos comunes
                {self.opening_shortname}: {[move[0] for move in
                unique_opening_moves]}.")
            print(f"Movimientos comunes {self.opening_shortname}:
                {[move[0] for move in unique_opening_moves]}.")
            move_input =
                self.select_move_by_weighted_choice(unique_opening_moves)
            self.logger.write(f"IA mueve: {move_input}")
            print(f"IA mueve: {move_input}")
            return self.manual_move(move_input, board,
                displayed_board)
        else:
            # Si no existen movimientos para alcanzar alguna posicin
                de apertura
            # Se habilita el motor Stockfish para el resto de la
                partida
            move_input = self.stockfish_move(board, displayed_board)
            self.logger.write(f"IA mueve: {move_input}")
            print(f"IA mueve: {move_input}")
            return move_input

    def flip_board_if_needed(self, displayed_board):
        if not displayed_board.flipped:
            display.flip(displayed_board)

    def load_historical_games(self, moves_data_path =
         "./data/df_2_just_moves_fen.csv"):
        df = pd.read_csv(moves_data_path, encoding='utf-8',
             engine='python')
        return df

    def filter_openings_by_played_move(self, turn_column_name,
         played_move):
```

```
        '''
        Filtra las partidas en el data set historico, para que las
            siguientes sugerencias sean con respecto
        a nuevas posiciones alcanzadas.
        '''
        df = self.df_hist_moves_filter
        if df.empty or df.shape[0]<=1:
            df = pd.DataFrame()
        elif turn_column_name in df.columns:
            played_move_exists =
                df[turn_column_name].isin([played_move]).any()
            if played_move_exists:
                df = df[df[turn_column_name] == played_move]
        else:
            df = pd.DataFrame()
        self.df_hist_moves_filter = df

    def predict_position(self, fen, moves, model):
        features = count_all_features(fen, moves)
        return model.predict_proba(features)

    def show_position_predictions(self, board, loaded_model):
        prob = self.predict_position(board.fen(), board.ply(),
            loaded_model)
        position_predictions = f"Negras:
            {'{:.6f}'.format(prob[0][0])}.
            Blancas:{'{:.6f}'.format(prob[0][1])}"
        self.logger.write(position_predictions)
        print(position_predictions)

    def show_opening_reached(self, board, turn_column_name):
        opening_reached = get_current_opening(self.df_hist_moves,
            turn_column_name, board.fen(), board.ply())
        if opening_reached:
            self.logger.write(f"Apertura alcanzada:
                {opening_reached}")
            print(f"Apertura alcanzada: {opening_reached}")

    def show_position_features(self, fen, turns):
        self.logger.write(f"-----Turno {turns}:{fen}-----")
        df_white = get_all_features_uf(fen, True)
        df_black = get_all_features_uf(fen, False)
        self.logger.write(f"---------------Posicion
            Blancas---------------")
        self.logger.write(df_white.to_string(index=True))
        self.logger.write(f"---------------Posicion
            Negras---------------")
        self.logger.write(df_black.to_string(index=False))

    def play_game_function(self, b_or_w_input):
```

```
    return self.run_human_black_board if b_or_w_input == 'b'
        else self.run_human_white_board

def filter_possible_moves_by_opening(self):
 return
     self.df_hist_moves[self.df_hist_moves['opening_shortname']
     == self.opening_shortname].copy()

def select_random_opening(self):
    # Calcula las proporciones de cada apertura en el conjunto
        de datos
    counts =
        self.df_hist_moves['opening_shortname'].value_counts()
    weights = counts / counts.sum()

    # Redondea las proporciones a dos decimales y convierte a
        porcentajes
    rounded_weights = np.round(weights, 2) * 100

    # Ajuste para asegurar que ninguna apertura tenga una
        probabilidad menor que 1%
    adjusted_weights = np.maximum(rounded_weights, 1).astype(int)

    # Selecciona una apertura al azar basado en los pesos
        ajustados
    opening_selected = random.choices(adjusted_weights.index,
        weights=adjusted_weights.values, k=1)[0]

    # Muestra e imprime la apertura seleccionada
    print(f"Apertura a practicar {opening_selected}")
    self.logger.write(f"Apertura a practicar {opening_selected}")

    return opening_selected

def select_random_color(self):
    return random.choice(['w', 'b'])

def select_move_by_weighted_choice(self, weights):
    """
    Selecciona un movimiento de ajedrez basado en una lista de
        pesos para cada movimiento.

    Parmetros:
    - weights: Lista de tuplas, donde cada tupla contiene un
        movimiento (como 'e4') y su peso asociado.

    Retorna:
    - Movimiento seleccionado de manera ponderada.
    """
    # Desempaquetar la lista de tuplas en movimientos y sus
        respectivos pesos
```

```
        moves, move_weights = zip(*weights) # from
            get_unique_opening_moves

        # Seleccionar un movimiento de manera ponderada basada en
            los pesos
        selected_move = random.choices(moves, weights=move_weights,
            k=1)[0]
        return selected_move

    def get_unique_opening_moves(self, turno, fullmove_number):
        """
        Filtra el DataFrame basado en el 'opening_shortname'
            proporcionado y retorna los valores nicos
        del turno.

        Parmetros:
        - turno: Booleano, True para WHITE, False para BLACK.
        - fullmove_number: Entero, nmero de la jugada de la partida
            de ajedrez.
        - opening_shortname: String, nombre de la apertura a filtrar.

        Retorna:
        - Lista de valores nicos de la columna 'turn_column_name' o
            None si la columna no existe.
        """
        turno_str = "0w" if turno else "1b"
        turn_column_name = f"{turno_str}_{fullmove_number}"

        if self.df_hist_moves_filter.empty:
            return None, turn_column_name

        # Verificar si la columna generada existe en el DataFrame
            filtrado
        if turn_column_name in self.df_hist_moves_filter.columns:
            # Calcular la frecuencia de cada valor nico
            value_counts =
                self.df_hist_moves_filter[turn_column_name].value_counts(normalize=True)
                # Valores entre 0 y 1

            # Normalizacin para aumentar la probabilidad de que la
                mquina te juegue jugadas poco comunes
            # 0-5: 1
            # 5-50: 3
            # 50-100: 20
            determine_weight = lambda proportion: 1 if proportion <=
                0.05 else \
                                    5 if proportion <= 0.50 else 20

            # Crear la lista de tuplas (valor nico, peso normalizado)
                usando la funcin lambda
```

```
            weights = [(value, determine_weight(proportion)) for
                value, proportion in value_counts.items()]

            return weights, turn_column_name
        else:
            return None, turn_column_name
```

4.12 *play_chess.py*

```
import chess
import chess.engine
from chessboard import display
import pickle
from openings_ialex import OpeningsIA
from utils import clean_console

# Descargar y extraer toda la carpeta en ./stockfish
# https://stockfishchess.org/download/windows/
stockfish_path = "./stockfish/stockfish-windows-x86-64.exe"

# Determina si el juego estar generando openings random o los
    pedir al usuario
RANDOM_OPENINGS = True

while True:
    with chess.engine.SimpleEngine.popen_uci(stockfish_path) as
        engine:
        clean_console()
        displayed_board = display.start()
        loaded_model =
            pickle.load(open("./pickles/models/xgoboost_model0416_18:06.pkl",
            'rb'))
        board = chess.Board()
        if RANDOM_OPENINGS:
            openingsIA = OpeningsIA(engine, random=True)
            input("'z' para reiniciar. 'x' para cerrar. 'Enter'
                iniciar:")
        else:
            opening_shortname = input("Apertura a practicar: ")
            b_or_w_input = input("Blancas 'w' o negras 'b': ")
            openingsIA = OpeningsIA(engine, opening_shortname =
                opening_shortname, b_or_w_input = b_or_w_input)


        while not board.is_game_over() and not
            openingsIA.restart_game:
            unique_opening_moves, turn_column_name =
                openingsIA.get_unique_opening_moves(board.turn,
                board.fullmove_number)
```

```
        played_move = openingsIA.play_game(board,
            displayed_board, unique_opening_moves)

        if openingsIA.restart_game:
           break

        openingsIA.show_position_predictions(board, loaded_model)
        openingsIA.show_position_features(board.fen(),
            board.ply())
        openingsIA.filter_openings_by_played_move(turn_column_name,
            played_move)
        openingsIA.show_opening_reached(board, turn_column_name)
```

4.13 *utils.py*

```
import pandas as pd
import tkinter as tk
from tkinter import ttk
import datetime
from tkinter import font
import subprocess
import sys
import os
import platform

class ChessLogger:
   def __init__(self, path,
       filename_format="chess_log_%Y-%m-%d_%H-%M.txt"):
      self.path = path
      self.filename_format = filename_format
      self.filename = self.generate_filename()

   def generate_filename(self):
      """Genera el nombre del archivo basado en la fecha actual y
          el formato especificado."""
      date_str =
          datetime.datetime.now().strftime(self.filename_format)
      return f"{self.path}\{date_str}"

   def write(self, text):
      """Escribe el texto proporcionado en el archivo de log."""
      with open(self.filename, 'a') as log_file:
         log_file.write(text + "\n")

def open_file(ruta_archivo):
   if sys.platform == "win32":
      subprocess.run(["start", ruta_archivo], shell=True)
   elif sys.platform == "linux":
```

```
            subprocess.run(["xdg-open", ruta_archivo])
        else: #IOS
            subprocess.run(["open", ruta_archivo])


# Funcin para mostrar el DataFrame en un cuadro de dilogo emergente
def show_df_in_window(df, title):
    # Crear una ventana emergente
    window = tk.Tk()
    window.title(title)

    frame = ttk.Frame(window)
    frame.pack(fill='both', expand=True)

    tree = ttk.Treeview(frame, columns=df.columns, show='headings')
    vsb = ttk.Scrollbar(frame, orient="vertical",
        command=tree.yview)
    hsb = ttk.Scrollbar(frame, orient="horizontal",
        command=tree.xview)
    tree.configure(yscrollcommand=vsb.set, xscrollcommand=hsb.set)

    tree.pack(side='left', fill='both', expand=True)
    vsb.pack(side='right', fill='y')
    hsb.pack(side='bottom', fill='x')

    # Usar tkinter.font para crear una instancia de fuente
    fonte = font.Font(family="Helvetica", size=10) # Crea una
        fuente para usar en la medicin

    # Configurar las cabeceras
    for col in df.columns:
        tree.heading(col, text=col)
        # Ajustar el ancho de las columnas
        # Calcular el ancho mximo de los datos en cada columna
        max_width = max([fonte.measure(str(val)) for val in df[col]]
            + [fonte.measure(col)])
        tree.column(col, width=max_width)

    # Agregar los datos del DataFrame al Treeview
    for index, row in df.iterrows():
        tree.insert("", "end", values=list(row))

    window.mainloop()

def clean_console():
    sistema_operativo = platform.system()
    if sistema_operativo == 'Windows':
        os.system('cls') # Para Windows
    else:
        os.system('clear') # Para Unix/Linux/MacOS
```

Bibliografía

[1] J. H. McMillan and D. R. Forsyth, "What theories of motivation say about why learners learn," *New Directions for Teaching and Learning*, vol. 45, no. 2, pp. 39–51, 1991.

[2] B. Fischer, S. Margulies, and D. Mosenfelder, *Bobby Fischer Teaches Chess*. Bantam Books, second ed., 1972.

[3] G. D. Luca, "How important are chess openings? (7 reasons to study)." https://chesspulse.com/how-important-are-openings-in-chess, Jan. 2012. Accessed on: 2024-01-29.

[4] L. Alburt, R. Dzindzichashvili, and E. Perelshteyn, *Chess openings for white explained*. Chess information and research center, first ed., 2007.

[5] D. Hooper and K. Whyld, *The oxford companion to chess*. Brithis Library Cataloguing, first ed., 1984.

[6] C. Tempo, "Chess tempo." https://old.chesstempo.com/game-database.html, Dec. 2023.

[7] C. position trainer, "Chess position trainer." http://www.chesspositiontrainer.com/index.php/en/buy, Dec. 2019.

[8] C. Reader, "Chessbase reader 2017." https://en.chessbase.com/pages/download, Dec. 2017.

[9] Chess.com, "Chess.com." https://www.chess.com/home, Dec. 2017.

[10] . chess, "365 chess." https://www.365chess.com/opening.php, Dec. 2023.

[11] D. Sikka, "Online chess match prediction." https://www.kaggle.com/code/dhruvsikka/online-chess-match-prediction/input, Jan. 2022. Accessed on: 2024-04-14.

[12] C. S. LLC, "Learning forsyth-edwards notation." https://www.chessgames.com/fenhelp.html, 2024. Accessed on: 2024-04-15.

[13] G. James, D. Witten, T. Hastie, and R. Tibshirani, "An introduction to statistical learning," 2013.

[14] D. M. W. Powers, "Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation," 2011.

[15] J. Han, J. Pei, and M. Kamber, "Data mining: Concepts and techniques," 2011.

[16] G. W. Brier, "Verification of forecasts expressed in terms of probability," 1950.

[17] "Applying statistical methods to evaluate game outcomes: A chess perspective," 2022. Whitepaper on Statistical Methods in Games.

[18] T. Fawcett, "An introduction to roc analysis," 2006.

[19] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," 1997.

[20] C. Cortes and V. Vapnik, "Support-vector networks," 1995.

[21] B. Schölkopf and A. J. Smola, "Learning with kernels: Support vector machines, regularization, optimization, and beyond," 2002.

[22] J. Shawe-Taylor and N. Cristianini, "Kernel methods for pattern analysis," 2004.

[23] C.-W. Hsu and C.-J. Lin, "A practical guide to support vector classification." Department of Computer Science, National Taiwan University, 2003.

[24] T. Chen, "Xgboost: A scalable tree boosting system," 2016. Available at https://arxiv.org/abs/1603.02754.

[25] "Xgboost documentation." https://xgboost.readthedocs.io, 2021. Comprehensive guide and reference to XGBoost.

[26] "Lightgbm documentation." https://lightgbm.readthedocs.io, 2021. Official documentation for LightGBM.

[27] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree." https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf, 2017. Advances in Neural Information Processing Systems 30.