Victory Vision: Chess winner Forecaster

Introduction:

Chess, a game of timeless elegance and profound complexity, has captivated players for centuries. It represents the epitome of strategy and skill, where the ability to assess one's chances of winning is of paramount importance. In the contemporary world, chess has evolved into an international phenomenon, marked by official competitions conducted by the FIDE (International Chess Federation) to determine grandmasters and, ultimately, the world champion. Given the pivotal role competitive chess plays in the lives of professional players, there is a growing desire to enhance their odds of success and to devise winning strategies.

This project embarks on a remarkable journey to develop a predictive model for chess game outcomes, with the primary aim of creating a tool that empowers players to estimate their chances of winning a game. This estimation will be rooted in an analysis of the current state of the chessboard and will incorporate other relevant factors such as playing style, player ratings, and more.

- **Business Understanding**

  In the world of competitive chess, where one move can be the difference between victory and defeat, the ability to foresee your chances of winning is akin to wielding a strategic advantage. Whether you're a professional player preparing for an official match or an amateur looking to improve your game, having insights into your likelihood of success is invaluable. This project seeks to bridge this informational gap by offering a tool that can inform players about their odds of winning based on a comprehensive analysis of their ongoing game.

  This tool could prove immensely beneficial not only to individual players but also to coaches, trainers, and chess enthusiasts. It has the potential to transform the way chess is played, learned, and taught, enhancing the overall experience of the game.

- **Possible Impacts**

  Enhanced Player Performance: Professional players can use this predictive model to fine-tune their strategies, adapt their play style, and make more informed decisions during a game, ultimately increasing their chances of success.

  Learning and Teaching: The tool can serve as an invaluable learning aid, helping players of all levels to understand the nuances of the game better. It can also be used by coaches and trainers to provide targeted guidance to their students.

Game Enthusiasts: Chess enthusiasts and amateurs can use this tool for fun, practice, and self-improvement, gaining deeper insights into their gameplay.

Tournaments and Competitive Play: Tournament organizers can leverage this technology to offer real-time predictions and analyses, enhancing the excitement and engagement of the audience.

Problem Statement:

In light of the significance of this project, the problem at hand can be defined as follows: "Developing a predictive model to estimate the winning rate of a chess game." This project seeks to transform the vast and intricate world of chess into quantifiable data. The core challenge is to create a model that can provide players with a probabilistic estimate of their chances of winning a given chess game based on various features.

Since the winning rate can be conceptualized as a binary outcome (win or lose), this project can be framed as a binary classification problem. In classification modeling, we require two fundamental elements: a dataset that includes features and a target binary label for every sample.

The dataset at our disposal contains information on over 20,000 chess matches gathered from users on Lichess.org. It encompasses a wide array of features, including game-specific details like starting and ending times, the number of moves, game status, player IDs, ratings, and detailed information about the moves made during each game. Importantly, the 'winner' column in the dataset provides the critical binary target variable required for training our classification model.

Furthermore, the dataset includes various other columns that can be harnessed as feature sets for our model, making this project both feasible and intriguing. By applying machine learning techniques to this dataset, we aim to develop a predictive model that can reliably estimate the probability of winning a chess game and offer a strategic advantage to players, coaches, and chess enthusiasts alike.

Related Works:

In the realm of competitive chess analysis and prediction, our "Victory Vision: Chess Winner Forecaster" project builds upon the achievements of several notable endeavors. While our project aspires to construct a comprehensive predictive model for chess game outcomes, it's imperative to recognize the pioneering contributions of these related efforts.

- **Chess Engines and Software**

  Notable chess engines and software, including Stockfish and Deep Blue, have played pivotal roles in the advancement of computer-assisted chess gameplay. These engines employ intricate algorithms and artificial intelligence to forecast moves and assess game positions, thus setting a significant precedent for predictive capabilities within the realm of chess.

- **Chess Opening Databases**

  Resources such as ChessBase and OpeningTree have furnished extensive datasets relating to historical game openings and their subsequent outcomes. These references prove invaluable for comprehending the significance of initial moves and their influence on game success.

- **Online Chess Platforms**

  Leading online platforms, such as Chess.com and Lichess.org, have introduced a plethora of features for chess enthusiasts, encompassing game analysis, statistical insights, and move recommendations. These platforms harness user-generated data to provide a nuanced understanding of individual performance and strategic approaches.

- **Historical Game Analysis**

  Researchers and dedicated chess enthusiasts have conducted meticulous analysis of historical chess games, deciphering the intricate patterns and strategies employed by renowned players like Bobby Fischer and Garry Kasparov. These studies contribute significantly to an enriched comprehension of chess gameplay.

- **Chess Analytics Tools**

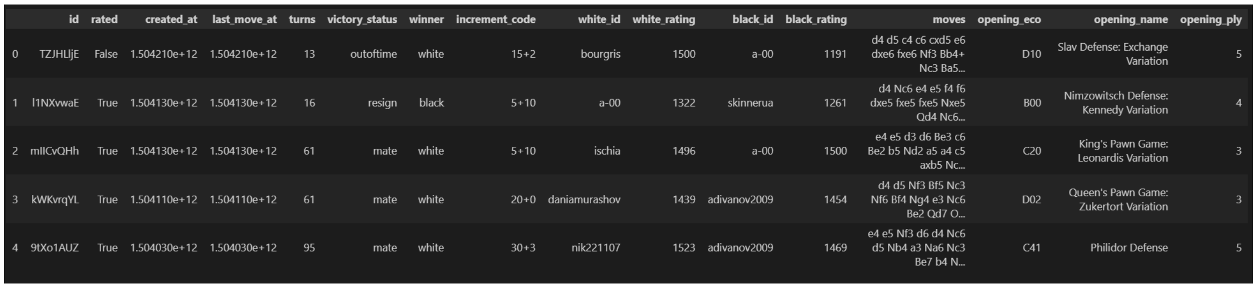
  Tools such as Chess.com's "Game Report" and Lichess.org's "Insights" have provided post-game analytical capabilities, shedding light on critical moments, move precision, and tactical errors. These tools serve as valuable aids for players seeking to extract insights from their games and refine their strategies.

While these antecedent works have unquestionably enriched the domain of chess analysis, our "Victory Vision" project stands out by focusing on real-time predictions. It endeavors to offer a probabilistic estimate of a player's likelihood of winning during an ongoing game. Leveraging a diverse dataset of over 20,000 chess matches, our project is dedicated to the creation of a distinct predictive model, one that factors in a multitude of game features and ultimately augments the strategic acumen of players and enthusiasts alike.

In contrast, previous research in chess endgame prediction has explored the application of machine learning techniques to determine the minimal steps required to secure a win. Methods such as Quinlan's use of Iterative Dichotomiser 3 (ID3) have revealed limitations in generating precise decision trees, resulting in suboptimal accuracy. This study addresses these challenges by employing machine learning models, including decision tree, random forest, and adaboost, to predict optimal depth-of-win for White in the king-rook versus king endgame. Initially, the decision tree model is introduced, but its stability and precision are found lacking. Subsequently, ensemble learning methods, including bagging and random forest, are incorporated to enhance the decision tree's performance. The adaboost method is also utilized to optimize model results, and a neural network model is developed. Collectively, these machine learning methods achieve a promising precision rate of approximately 60%. While these models demonstrate high accuracy and learning capability, they do not explicitly provide insights into chess principles. Future directions may involve model integration to enhance accuracy and expand our understanding of the chess endgame problem.

Datasets:

Chess Game Dataset (Lichess)
Link: https://www.kaggle.com/datasets/datasnaek/chess
This dataset contains information of over 20,000 chess matches collected from a selection of users on the site Lichess.org. The data was collected using Lichess API, which enables collection of any given user's game history. This dataset includes the followings columns: *ID of the game, label on whether the game is rated or not (True/False), game start time, game end time, number of turns, game status (mate, resign, out of time, ect), white and black players IDs, the winner, time increment rate, ratings of white and black players, all moves in standard chess notation, opening move in ECO (Standardized Code for any given opening), opening name, opening ply (Number of moves in the opening phase)*.

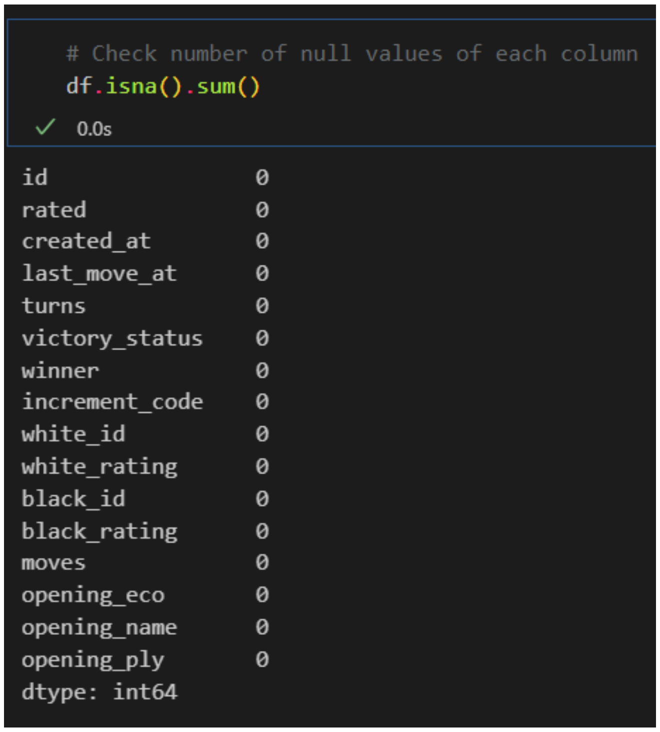


	id	rated	created_at	last_move_at	turns	victory_status	winner	increment_code	white_id	white_rating	black_id	black_rating	moves	opening_eco	opening_name	opening_ply
0	TZJHLljE	False	1.504210e+12	1.504210e+12	13	outoftime	white	15+2	bourgris	1500	a-00	1191	d4 d5 c4 c6 cxd5 e6 dxe6 fxe6 Nf3 Bb4+ Nc3 Ba5...	D10	Slav Defense: Exchange Variation	5
1	l1NXvwaE	True	1.504130e+12	1.504130e+12	16	resign	black	5+10	a-00	1322	skinnerua	1261	d4 Nc6 e4 e5 f4 f6 dxe5 fxe5 fxe5 Nxe5 Qd4 Nc6...	B00	Nimzowitsch Defense: Kennedy Variation	4
2	mIICvQHh	True	1.504130e+12	1.504130e+12	61	mate	white	5+10	ischia	1496	a-00	1500	e4 e5 d3 d6 Be3 c6 Be2 b5 Nd2 a5 a4 c5 axb5 Nc...	C20	King's Pawn Game: Leonardis Variation	3
3	kWKvrqYL	True	1.504110e+12	1.504110e+12	61	mate	white	20+0	daniamurashov	1439	adivanov2009	1454	d4 d5 Nf3 Bf5 Nc3 Nf6 Bf4 Ng4 e3 Nc6 Be2 Qd7 O...	D02	Queen's Pawn Game: Zukertort Variation	3
4	9tXo1AUZ	True	1.504030e+12	1.504030e+12	95	mate	white	30+3	nik221107	1523	adivanov2009	1469	e4 e5 Nf3 d6 d4 Nc6 d5 Nb4 a3 Na6 Nc3 Be7 b4 N...	C41	Philidor Defense	5

Figure 1: First 5 samples of Chess Game dataset

Methodology:

EDA

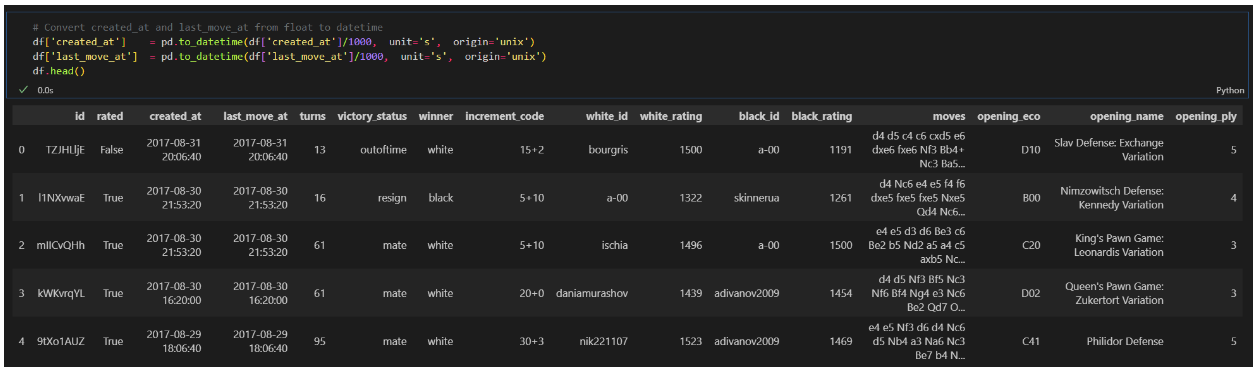


```
# Check number of null values of each column
df.isna().sum()
```

```
id                0
rated             0
created_at        0
last_move_at      0
turns             0
victory_status    0
winner            0
increment_code    0
white_id          0
white_rating      0
black_id          0
black_rating      0
moves             0
opening_eco       0
opening_name      0
opening_ply       0
dtype: int64
```

Figure 2: Finding null value

First, we checked if there is any null value in each of the columns present in the Chess Game dataset.

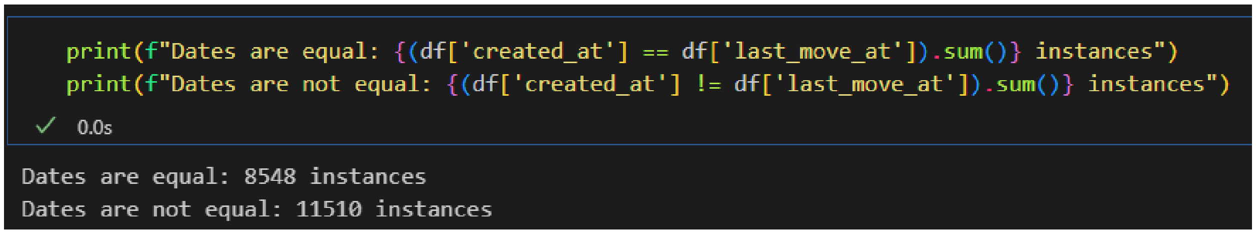


```
# Convert created_at and last_move_at from float to datetime
df['created_at']    = pd.to_datetime(df['created_at']/1000,  unit='s',  origin='unix')
df['last_move_at']  = pd.to_datetime(df['last_move_at']/1000,  unit='s',  origin='unix')
df.head()
```

	id	rated	created_at	last_move_at	turns	victory_status	winner	increment_code	white_id	white_rating	black_id	black_rating	moves	opening_eco	opening_name	opening_ply
0	TZJHLljE	False	2017-08-31 20:06:40	2017-08-31 20:06:40	13	outoftime	white	15+2	bourgris	1500	a-00	1191	d4 d5 c4 c6 cxd5 e6 dxe6 fxe6 Nf3 Bb4+ Nc3 Ba5...	D10	Slav Defense: Exchange Variation	5
1	l1NXvwaE	True	2017-08-30 21:53:20	2017-08-30 21:53:20	16	resign	black	5+10	a-00	1322	skinnerua	1261	d4 Nc6 e4 e5 f4 f6 dxe5 fxe5 fxe5 Nxe5 Qd4 Nc6...	B00	Nimzowitsch Defense: Kennedy Variation	4
2	mIICvQHh	True	2017-08-30 21:53:20	2017-08-30 21:53:20	61	mate	white	5+10	ischia	1496	a-00	1500	e4 e5 d3 d6 Be3 c6 Be2 b5 Nd2 a5 a4 c5 axb5 Nc...	C20	King's Pawn Game: Leonardis Variation	3
3	kWKvrqYL	True	2017-08-30 16:20:00	2017-08-30 16:20:00	61	mate	white	20+0	daniamurashov	1439	adivanov2009	1454	d4 d5 Nf3 Bf5 Nc3 Nf6 Bf4 Ng4 e3 Nc6 Be2 Qd7 O...	D02	Queen's Pawn Game: Zukertort Variation	3
4	9tXo1AUZ	True	2017-08-29 18:06:40	2017-08-29 18:06:40	95	mate	white	30+3	nik221107	1523	adivanov2009	1469	e4 e5 Nf3 d6 d4 Nc6 d5 Nb4 a3 Na6 Nc3 Be7 b4 N...	C41	Philidor Defense	5

Figure 3: Datetime Conversions

As shown in Figure 2, the first problem of this dataset is that the timestamp values of created_at and last_move_at columns are in numerical form thus we converted them to pandas datetime format.



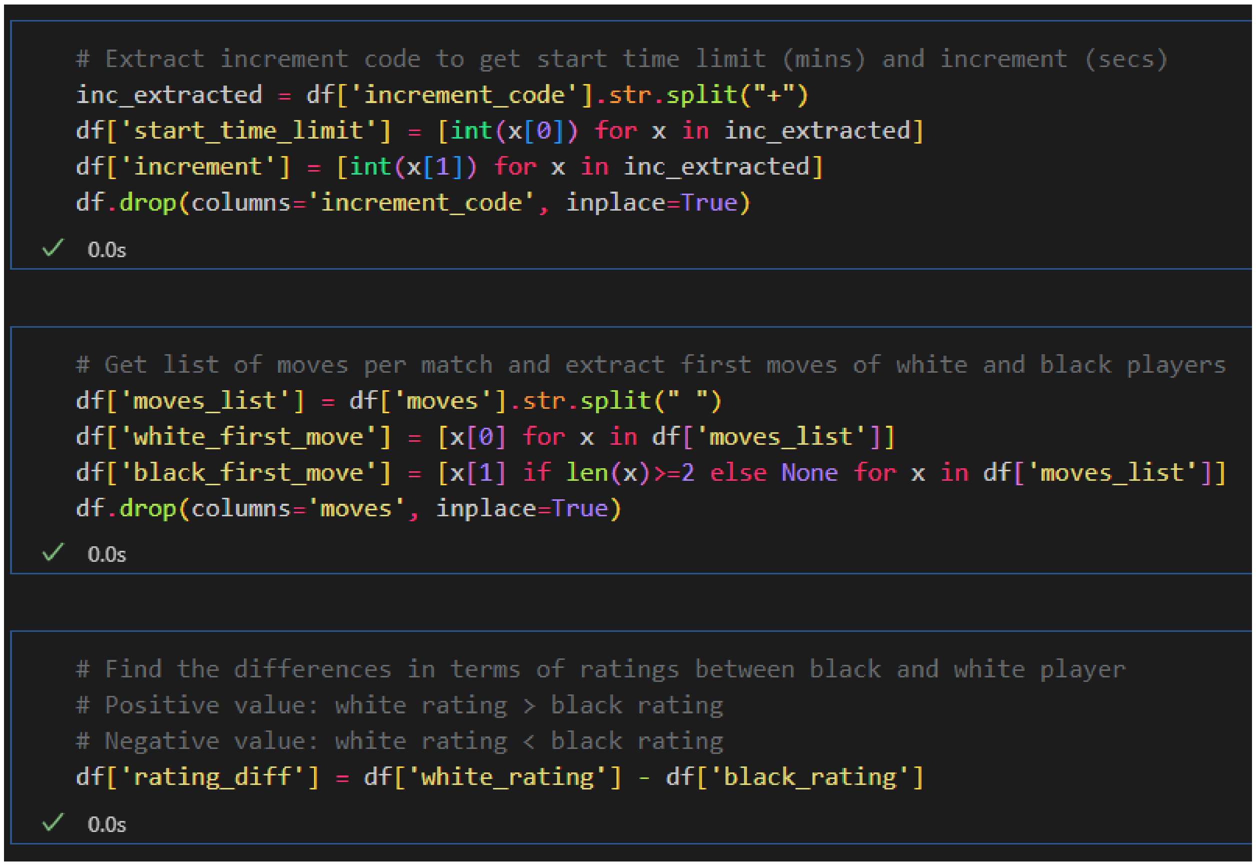
```
print(f"Dates are equal: {(df['created_at'] == df['last_move_at']).sum()} instances")
print(f"Dates are not equal: {(df['created_at'] != df['last_move_at']).sum()} instances")
```

✓ 0.0s

```
Dates are equal: 8548 instances
Dates are not equal: 11510 instances
```

Figure 4: Examine datetimes inconsistency

Next, we obtained the duration time each match took by subtracting the last_move_at with created_at features. However, as seen in Figure 4, in some instances, the two dates have exact same values signifying that there are possibly some errors on the methods to obtain these values.



```
# Extract increment code to get start time limit (mins) and increment (secs)
inc_extracted = df['increment_code'].str.split("+")
df['start_time_limit'] = [int(x[0]) for x in inc_extracted]
df['increment'] = [int(x[1]) for x in inc_extracted]
df.drop(columns='increment_code', inplace=True)
```

✓ 0.0s

```
# Get list of moves per match and extract first moves of white and black players
df['moves_list'] = df['moves'].str.split(" ")
df['white_first_move'] = [x[0] for x in df['moves_list']]
df['black_first_move'] = [x[1] if len(x)>=2 else None for x in df['moves_list']]
df.drop(columns='moves', inplace=True)
```

✓ 0.0s

```
# Find the differences in terms of ratings between black and white player
# Positive value: white rating > black rating
# Negative value: white rating < black rating
df['rating_diff'] = df['white_rating'] - df['black_rating']
```

✓ 0.0s

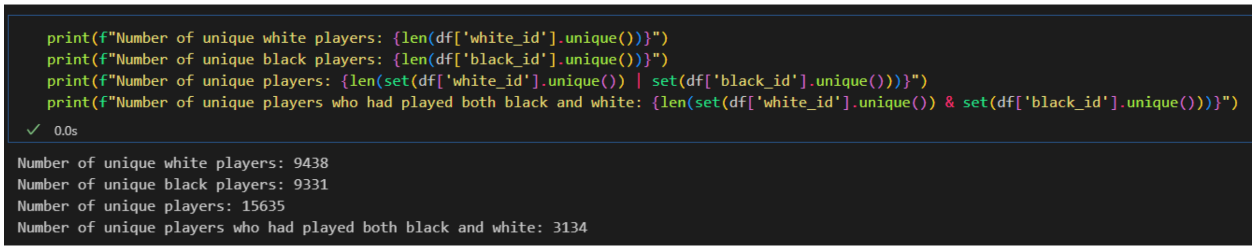
Figure 5: Feature extraction

We can extract both the start time limit and increment time from the increment_code column. In chess, the start time limit refers to the amount of time each player is assigned with. If a player's time expires, they generally lose the game. Increment time describes the amount of time limit increased after a player ended their turn by moving their chess piece.

A list of moves played by each player can also be obtained from the moves column. Chess games are typically played on an 8x8 board. Each column of the board is indexed by an alphabet from a to h while each row is indexed by a number from 1 to 8. A move code is identified by the combination of column and row indices, e.g., ‘e4’ refers to a pawn moving to the position e4 on a chess board.

Since in this project, we are also interested in how the first moves of each player affects their winning chances, we also extracted them from the moves list.

Finally, we also extracted the differences in ratings between black and white players.



```
print(f"Number of unique white players: {len(df['white_id'].unique())}")
print(f"Number of unique black players: {len(df['black_id'].unique())}")
print(f"Number of unique players: {len(set(df['white_id'].unique()) | set(df['black_id'].unique()))}")
print(f"Number of unique players who had played both black and white: {len(set(df['white_id'].unique()) & set(df['black_id'].unique()))}")
```

✓ 0.0s

```
Number of unique white players: 9438
Number of unique black players: 9331
Number of unique players: 15635
Number of unique players who had played both black and white: 3134
```

Figure 6: Number of unique players

Next, we explored the number of unique players on both sides of each match and in addition how many unique players overall who had played on both sides.

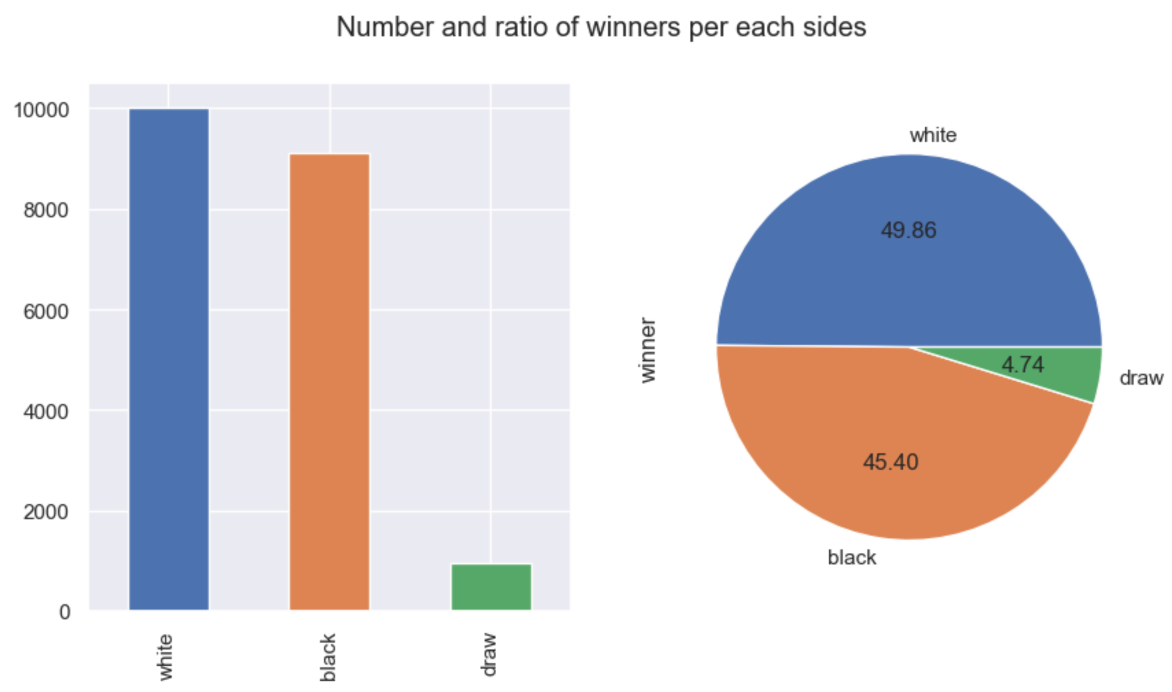


Figure 7: Number and ratio of winners on each side

figure:7 shows a side-by-side comparison of the number and ratio of chess game winners for different sides (e.g., white, black or draw) using a bar chart and a pie chart within the same figure. This visualization can help viewers quickly understand the distribution of winners in the dataset.

Results obtained in Figure 7 shows that, in general, both white players consistently win slightly more than black players. We can assume this is due to white players having the competitive advantage of being the one who move first.

Exploring the number of matches categorized by winner is essential as we will be using the winner column as a target for our classification model. Training a classifier using a dataset with imbalanced classes can easily lead to overfitting which should be avoided.

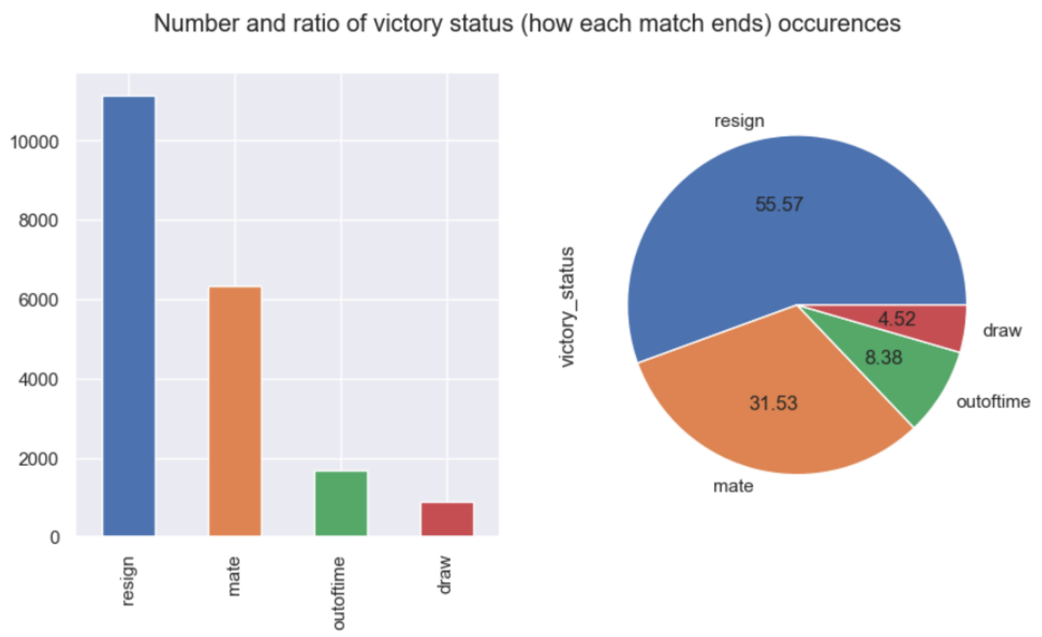


Figure 8: Number and ratio of victory status (how each match ends) occurrences

Figure:8 provides a visual representation that allows for a direct comparison between the counts and percentage distribution of various “victory_status” categories in a dataset. It offers valuable information regarding the manner in which chess matches conclude in relation to their victory status.

Results in figure 8 shows that most players tend to resign before the match ends, resulting in victory for the opponent, and rarely does a match ends in victory due to a player running out of time.

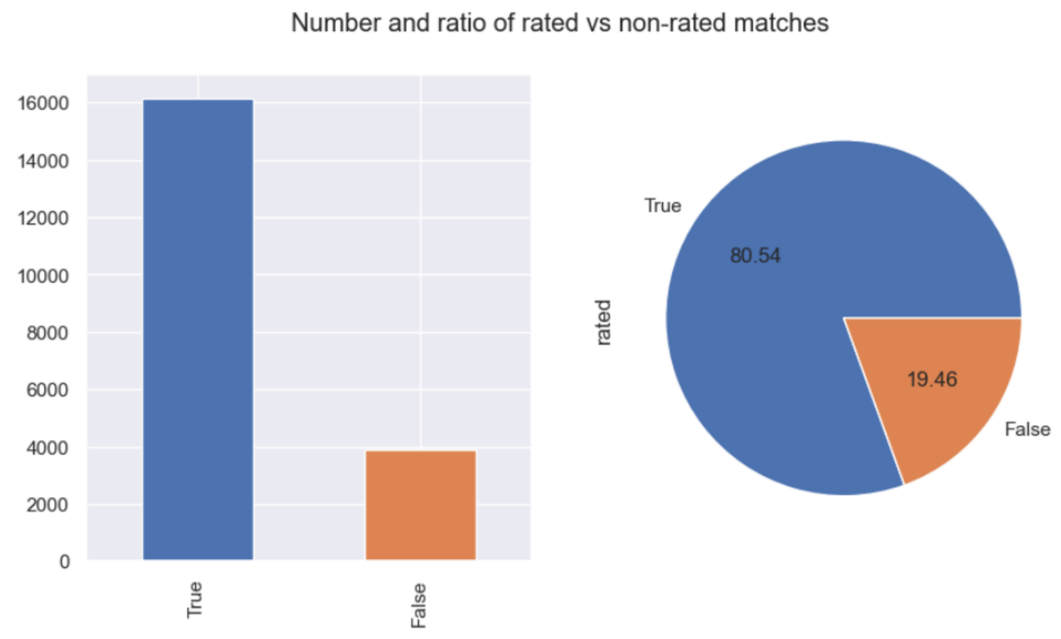


Figure 9: Number and ratio of rated vs non-rated matches

Figure:9 presents a single illustration containing two adjacent subplots. The initial subplot visualizes a bar chart that displays the frequency of rated and non-rated matches, while the subsequent subplot represents this data through a pie chart, offering insights into the distribution of rated and non-rated matches with the dataset.

Results in figure 9 shows that most matches conducted in Lichess websites in this dataset are rated. Rated match refers to a match that, after it is concluded, results in increasing in rating on the victor side and conversely the decreasing in rating on the loser. Non-rated matches are casual matches that do not affect the end results ratings of players.

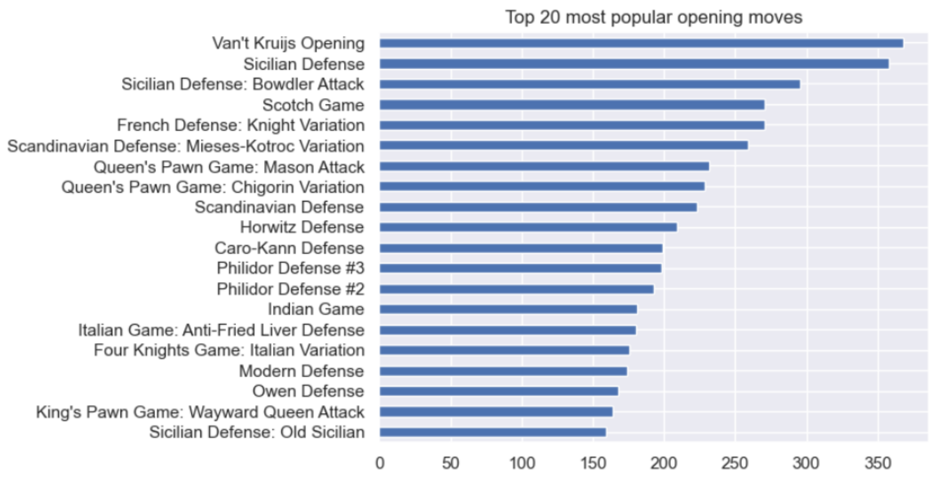


Figure 10: Top 20 most popular opening moves

Figure 10 illustrates a horizontal bar chart depicting the 20 commonly played chess opening moves in the dataset. The chart arranges these moves from the less frequently chosen ones on the left to the more popular ones on the right, enabling a swift assessment of the most commonly utilized chess opening strategies in the dataset.

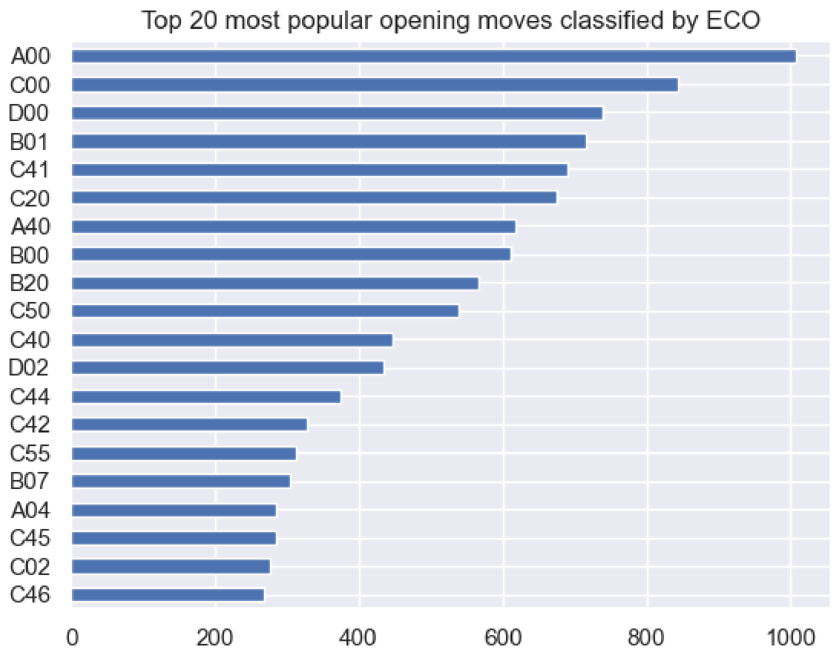


Figure 11: Top 20 most popular opening moves classified by ECO

Figure 11 depicts a horizontal bar graph that displays the 20 most favored chess opening moves. It uses the ECO codes along the vertical axis (y-axis) and represents the frequency of each ECO code along the horizontal axis (x-axis). The bars are organized in descending order, with the least common openings at the top and the most popular ones at the bottom. This visualization offers a concise summary of the preference for various chess opening moves as categorized by their ECO classification.

Results shown in figure 10 and 11 describe the popularity of each opening move played in the Lichess game dataset. The differences between opening moves shown in figure 10 and ECO opening moves shown in figure 11 are the way they are classified. ECO opening moves are classified into 5 main categories from "A" to "E" and each main category is further subdivided into 100 subcategories. ECO codes are usually broad as some opening variations are aggregated together into single code.

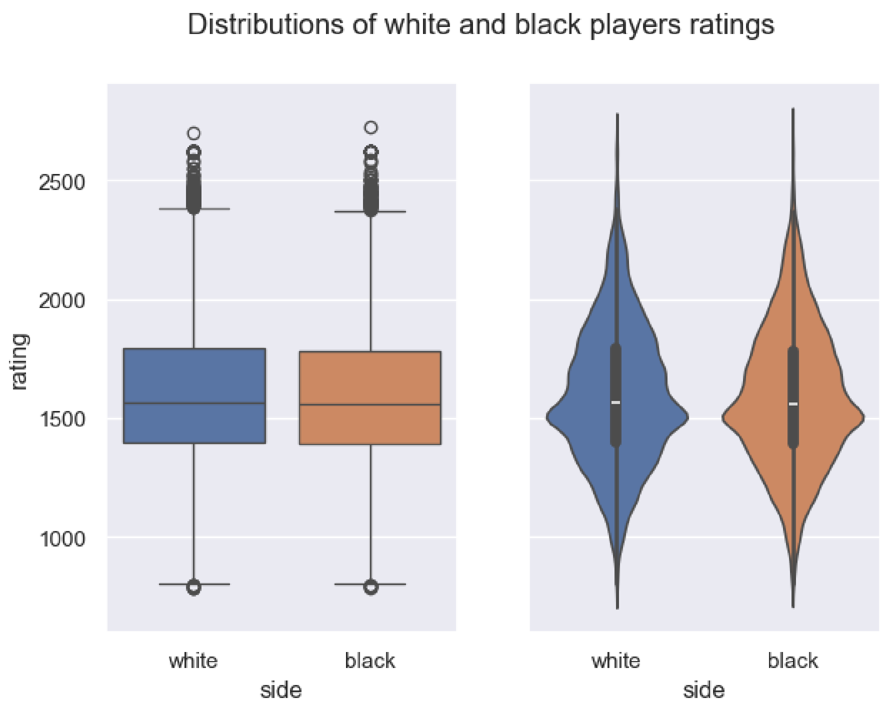


Figure 12: Distributions of white and black players ratings

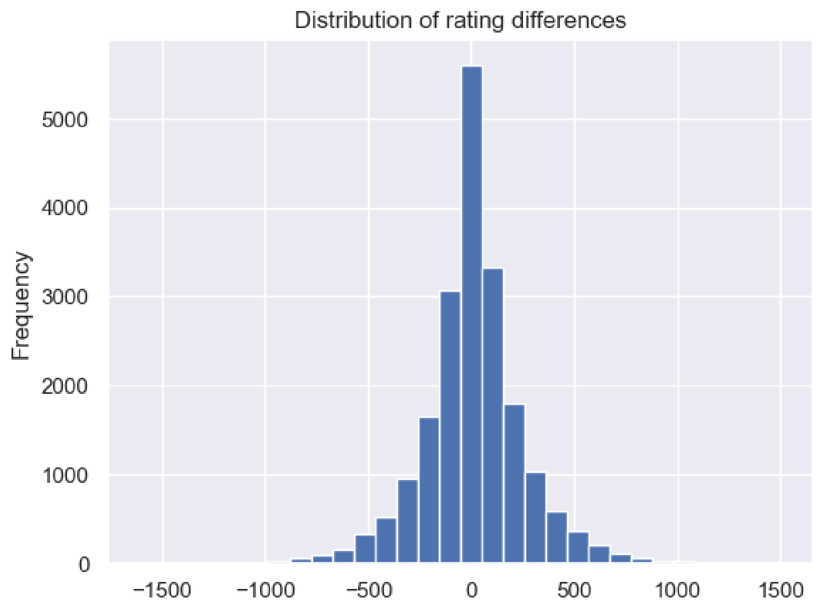


Figure 13: Distributions of rating differences

Figure 12 shows that in our chess matches dataset, both black and white players have virtually the same distribution in terms of ratings. Most players on both sides have ratings of around 1500. Figure 13 also shows that in most matches, both black and white players have around the same ratings. We can assume that this is because the matchmaking system in Lichess prefers putting players of similar ratings together

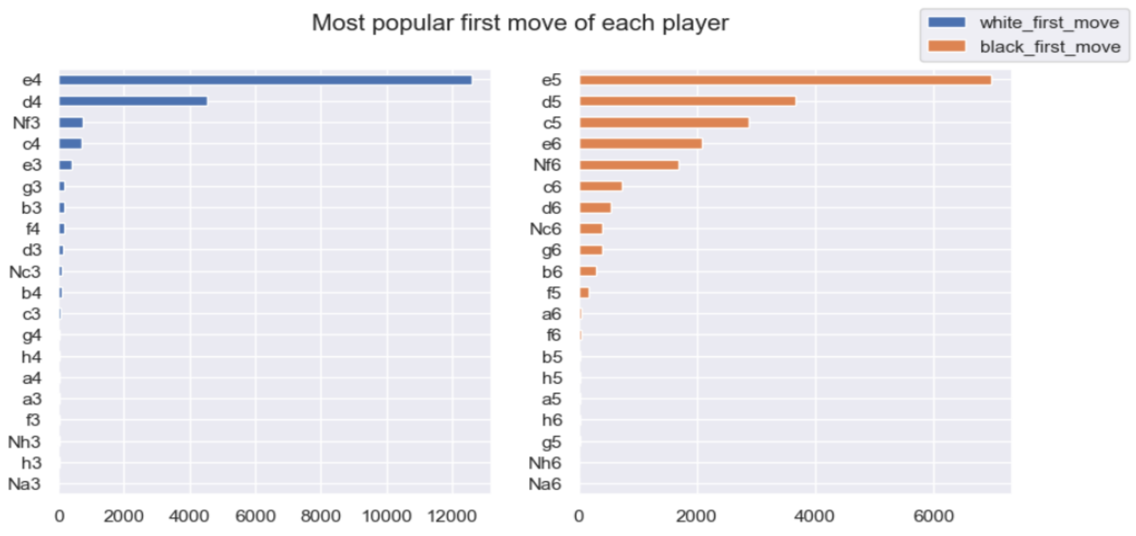
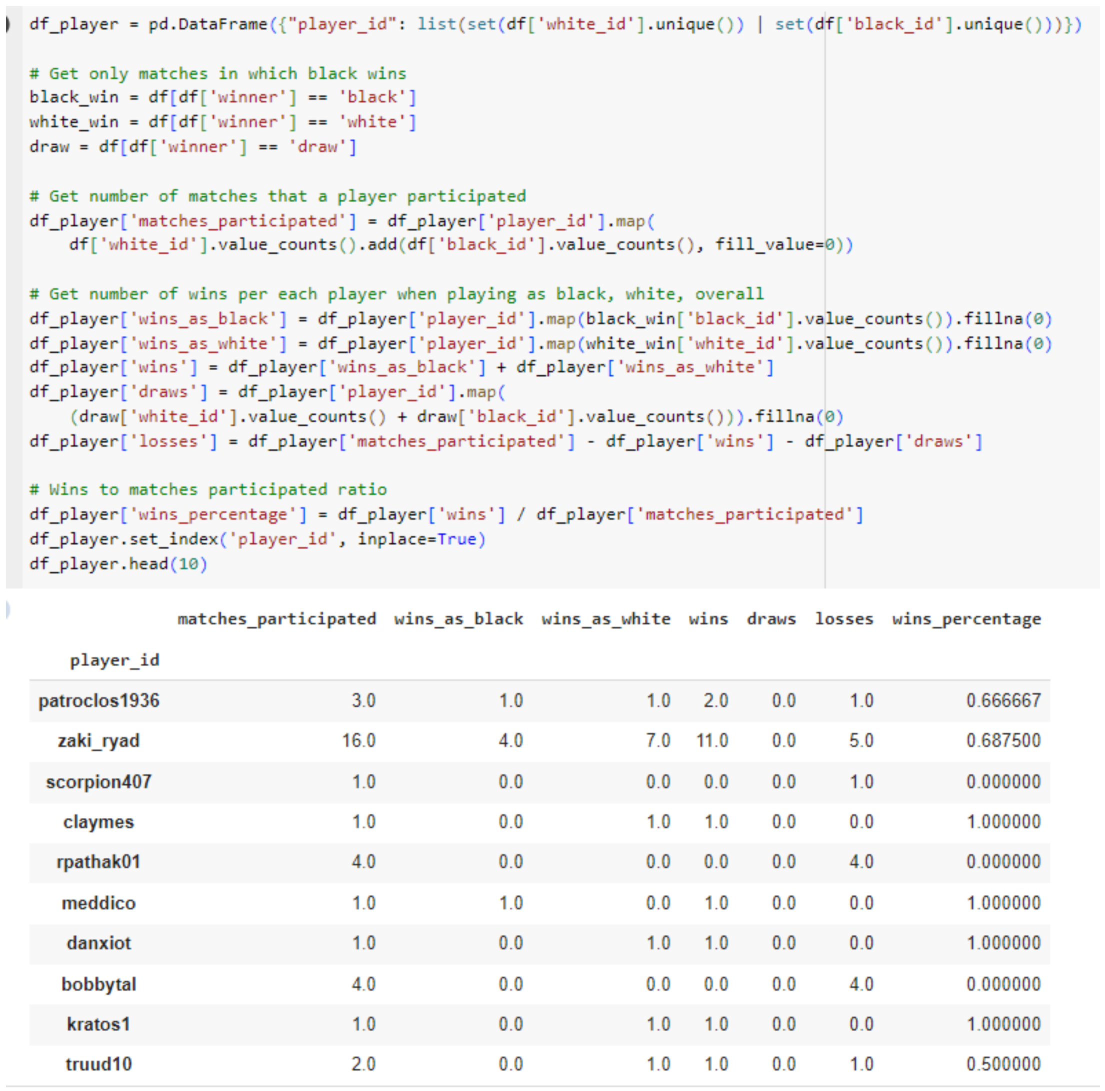


Figure 14: Most popular first move of each player

Figure 14 displays a visual representation consisting of two horizontal bar charts. These charts distinguish between the initial moves made by the white and black players in a chess game. The figure includes a title and a legend to provide context. The bars within the charts reflect the relative popularity of each potential first move.



```python
df_player = pd.DataFrame({"player_id": list(set(df['white_id'].unique()) | set(df['black_id'].unique()))})

# Get only matches in which black wins
black_win = df[df['winner'] == 'black']
white_win = df[df['winner'] == 'white']
draw = df[df['winner'] == 'draw']

# Get number of matches that a player participated
df_player['matches_participated'] = df_player['player_id'].map(
    df['white_id'].value_counts().add(df['black_id'].value_counts(), fill_value=0))

# Get number of wins per each player when playing as black, white, overall
df_player['wins_as_black'] = df_player['player_id'].map(black_win['black_id'].value_counts()).fillna(0)
df_player['wins_as_white'] = df_player['player_id'].map(white_win['white_id'].value_counts()).fillna(0)
df_player['wins'] = df_player['wins_as_black'] + df_player['wins_as_white']
df_player['draws'] = df_player['player_id'].map(
    (draw['white_id'].value_counts() + draw['black_id'].value_counts())).fillna(0)
df_player['losses'] = df_player['matches_participated'] - df_player['wins'] - df_player['draws']

# Wins to matches participated ratio
df_player['wins_percentage'] = df_player['wins'] / df_player['matches_participated']
df_player.set_index('player_id', inplace=True)
df_player.head(10)
```

	matches_participated	wins_as_black	wins_as_white	wins	draws	losses	wins_percentage
player_id							
patroclos1936	3.0	1.0	1.0	2.0	0.0	1.0	0.666667
zaki_ryad	16.0	4.0	7.0	11.0	0.0	5.0	0.687500
scorpion407	1.0	0.0	0.0	0.0	0.0	1.0	0.000000
claymes	1.0	0.0	1.0	1.0	0.0	0.0	1.000000
rpathak01	4.0	0.0	0.0	0.0	0.0	4.0	0.000000
meddico	1.0	1.0	0.0	1.0	0.0	0.0	1.000000
danxiot	1.0	0.0	1.0	1.0	0.0	0.0	1.000000
bobbytal	4.0	0.0	0.0	0.0	0.0	4.0	0.000000
kratos1	1.0	0.0	1.0	1.0	0.0	0.0	1.000000
truud10	2.0	0.0	1.0	1.0	0.0	1.0	0.500000

Figure 15: Player Statistics Dataframe

The output of this code is a DataFrame (df_player) that contains player-specific statistics, including the number of matches participated, wins, draws, losses, and win percentages for each player. You can access these statistics for a specific player by looking up their "player_id" in the DataFrame.

Figure 15 shows statistics of each player who participated in chess matches which include number of matches they have participated overall (both as black and white), number of wins as black, number of overall wins as white, number of wins, draws, losses as well as winning percentage (calculated by number of wins / number of match participated).

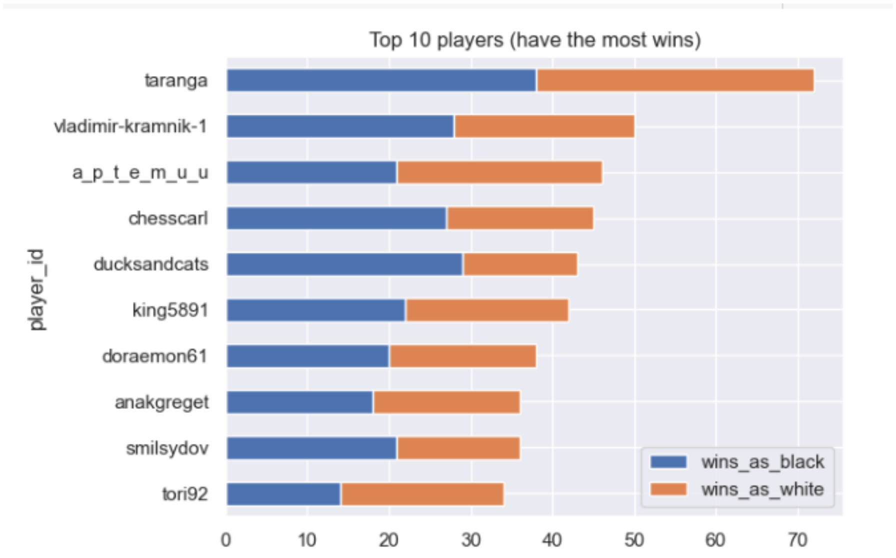


Figure 16: Top 10 players (have the most wins)

Figure 16 illustrates a horizontal bar graph depicting the top 10 chess players who have achieved the highest number of victories. In this chart, each player's bar is segmented into two sections, with one section indicating their wins while playing as black pieces and the other representing their wins as white pieces. The player with the highest total wins is positioned at the top of the chart.

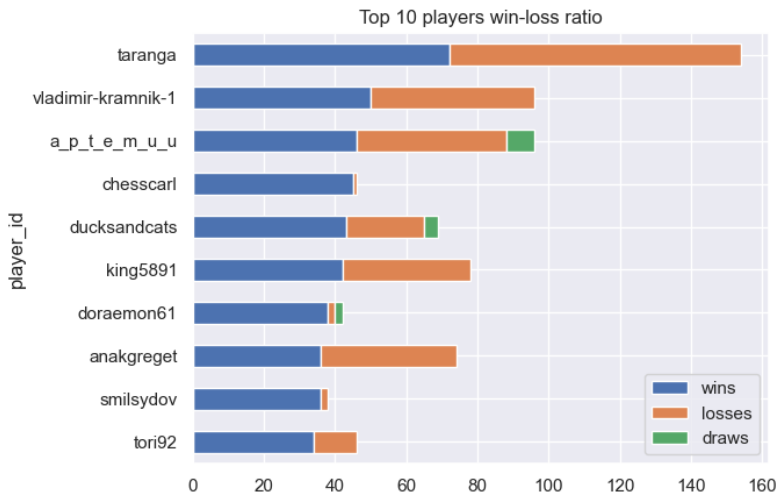
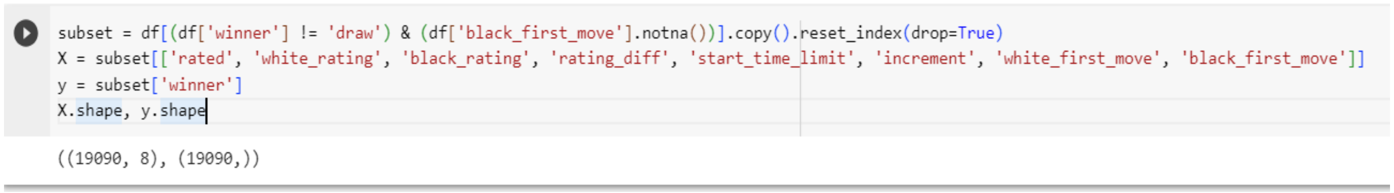


Figure 17: Top 10 players win-loss ratio

This visual representation depicts a horizontal stacked bar chart illustrating the win-loss ratios of the top 10 chess players. In this chart, each player is represented by a horizontal bar, and within each bar, there are distinct segments indicating their wins, losses, and draws. The bars are stacked one above the other, facilitating a straightforward comparison of the top players' performance based on their win-loss ratios.

From figure 17, we can see that although the player 'taranga' has the highest amount of wins overall, they however, also have approximately the same amount of loss. In contrast, the player 'chesscarl' , although ranked 4th in terms of amount winning, had won nearly every match. This result suggests that the number of wins does not correlate with the player's performance.

Feature Selection



```
subset = df[(df['winner'] != 'draw') & (df['black_first_move'].notna())].copy().reset_index(drop=True)
X = subset[['rated', 'white_rating', 'black_rating', 'rating_diff', 'start_time_limit', 'increment', 'white_first_move', 'black_first_move']]
y = subset['winner']
X.shape, y.shape
```

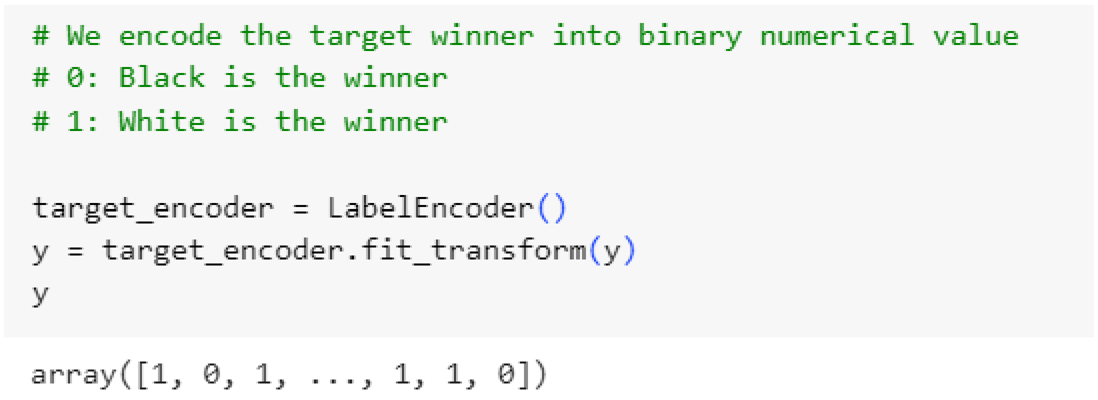
```
((19090, 8), (19090,))
```

Figure 18: DataFrame Subset Creation and Feature Selection for Machine Learning Analysis

For feature selection, we first selected a subset of data that contained matches that ended in either ‘black’ or ‘white’ only, and excluded matches that ended in just one move (black did not move and immediately withdrew). Next, we selected metagame data, which is data that is known before a match is started, as well as white and black players’ first moves as our features set.

As this is a classification task with the probability of white player winning as the final output, we selected the ‘winner’ column as our target.

This piece of code(figure:18) generates a smaller portion of a DataFrame, likely holding information regarding chess games. It proceeds to pick particular columns as attributes (referred to as X) and a target variable (referred to as y) for further analysis. The results obtained from X.shape and y.shape offer the size specifications of the feature matrix X and the target vector y, which are essential for a variety of machine learning and data analysis purposes.

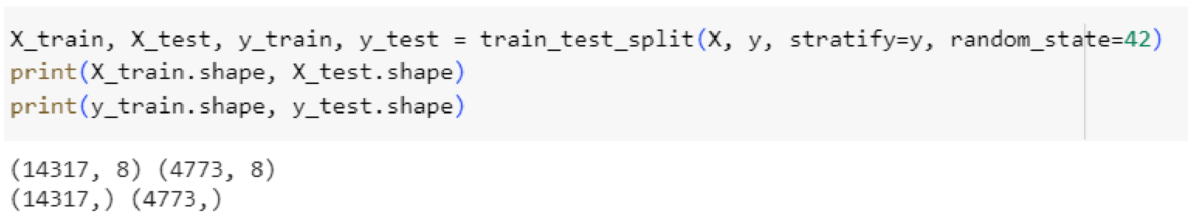


```
# We encode the target winner into binary numerical value
# 0: Black is the winner
# 1: White is the winner

target_encoder = LabelEncoder()
y = target_encoder.fit_transform(y)
y
```

```
array([1, 0, 1, ..., 1, 1, 0])
```

Figure 19: Label encoding for target variable

The given code(figure:19) pertains to the transformation of the target variable in a chess game winner prediction problem. In this scenario, the target variable signifies the victor of a chess match, and the code readies it for utilization in machine learning or predictive modeling. This encoding step is essential because the majority of machine learning algorithms operate with numerical data. By converting categorical variables, such as the chess game-winner, into numerical values, you enable the effective application of these algorithms for prediction purposes.

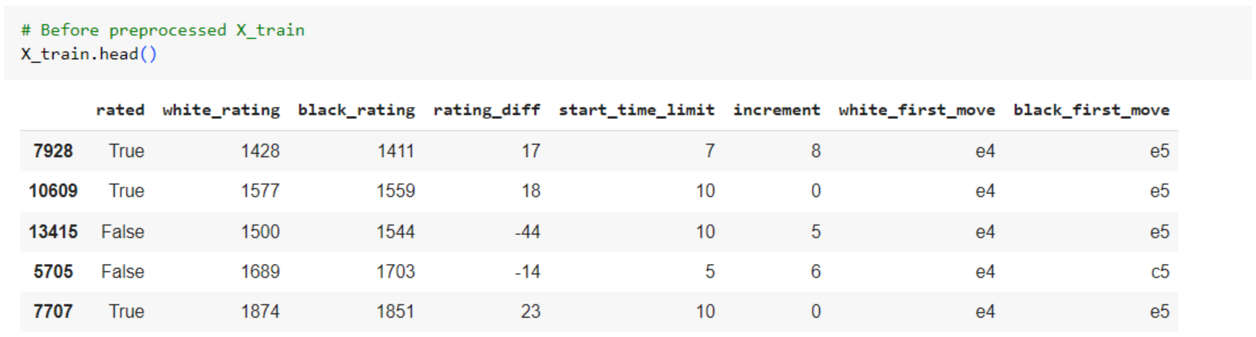


```
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
print(X_train.shape, X_test.shape)
print(y_train.shape, y_test.shape)
```

```
(14317, 8) (4773, 8)
(14317,) (4773,)
```

Figure 20: Train/test splitting

Code shown in figure 20 describes the splitting of the dataset into training and testing sets. By default, Sklearn's train_test_split() split the dataset using 75:25 ratio. We also specified the parameter 'stratify=y' in order to ensure that both training and testing samples have balanced sets of data

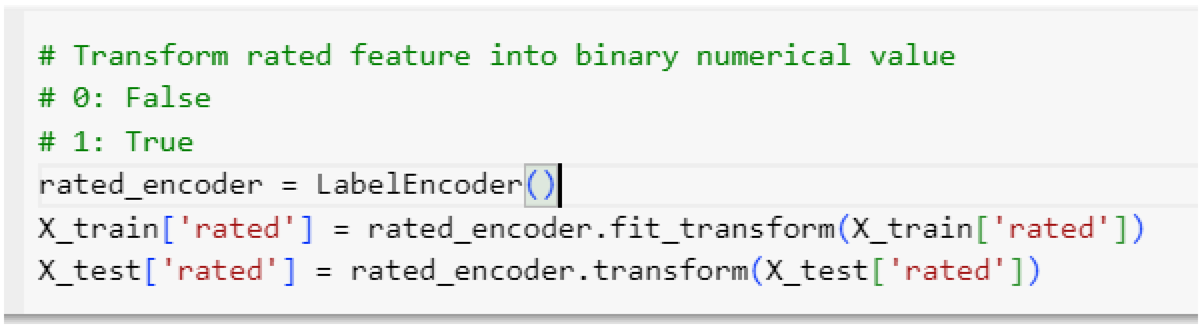


```
# Before preprocessed X_train
X_train.head()
```

	rated	white_rating	black_rating	rating_diff	start_time_limit	increment	white_first_move	black_first_move
7928	True	1428	1411	17	7	8	e4	e5
10609	True	1577	1559	18	10	0	e4	e5
13415	False	1500	1544	-44	10	5	e4	e5
5705	False	1689	1703	-14	5	6	e4	c5
7707	True	1874	1851	23	10	0	e4	e5

Figure 22: Before preprocessed X_train data

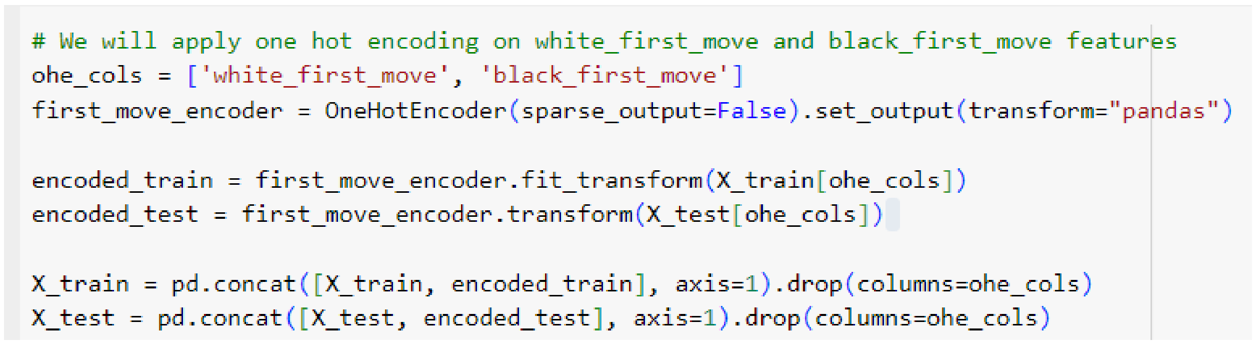
Data Preprocessing



```
# Transform rated feature into binary numerical value
# 0: False
# 1: True
rated_encoder = LabelEncoder()
X_train['rated'] = rated_encoder.fit_transform(X_train['rated'])
X_test['rated'] = rated_encoder.transform(X_test['rated'])
```

Figure 23: Label encoding for 'rated' feature

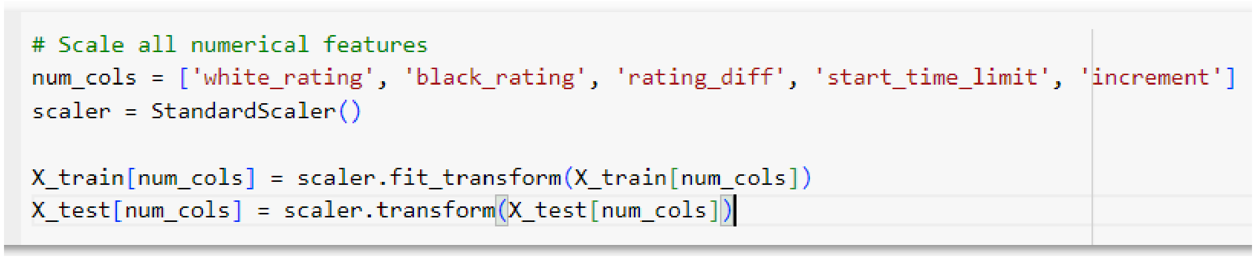
The provided code is about converting the 'rated' feature into binary numeric values (0 and 1) using a LabelEncoder. This is a common step in preparing categorical data for machine learning models that accept only numeric input. When this code is applied, the 'rated' feature in both the training and test datasets will be substituted with numeric values. For instance, 'True' could be mapped to 1, and 'False' to 0. This conversion enables machine learning models to effectively utilize the 'rated' feature, which necessitates numeric input.



```
# We will apply one hot encoding on white_first_move and black_first_move features
ohe_cols = ['white_first_move', 'black_first_move']
first_move_encoder = OneHotEncoder(sparse_output=False).set_output(transform="pandas")

encoded_train = first_move_encoder.fit_transform(X_train[ohe_cols])
encoded_test = first_move_encoder.transform(X_test[ohe_cols])

X_train = pd.concat([X_train, encoded_train], axis=1).drop(columns=ohe_cols)
X_test = pd.concat([X_test, encoded_test], axis=1).drop(columns=ohe_cols)
```

Figure 24: One hot encoding for white and black first moves feature

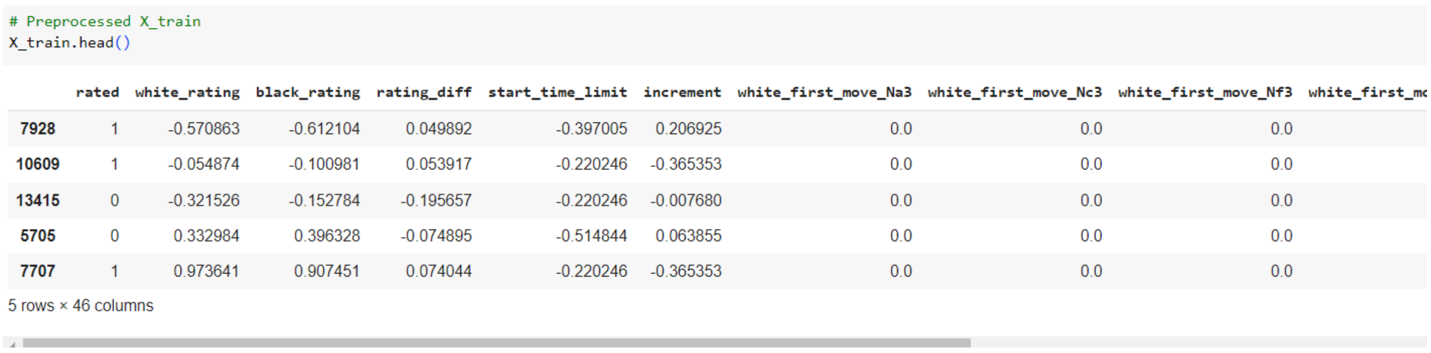
This piece of code(in figure:24) is carrying out data preprocessing through the application of one-hot encoding to the designated categorical columns, namely "white_first_move" and "black_first_move." It then proceeds to modify the training and testing datasets by replacing the original categorical columns with the newly generated one-hot encoded columns. This preparatory procedure is commonly executed to convert categorical data into a numerical format, a requirement for machine learning models.



```
# Scale all numerical features
num_cols = ['white_rating', 'black_rating', 'rating_diff', 'start_time_limit', 'increment']
scaler = StandardScaler()

X_train[num_cols] = scaler.fit_transform(X_train[num_cols])
X_test[num_cols] = scaler.transform(X_test[num_cols])
```

Figure 25: Standardization of all numerical features

Figure 25 illustrates that when we scale the numerical features, we ensure that all features are standardized, which can enhance the effectiveness of machine learning models, particularly those dependent on distance-based computations or gradient descent optimization. This is a typical initial data processing step in machine learning, aimed at enabling models to converge more swiftly and yield improved results.

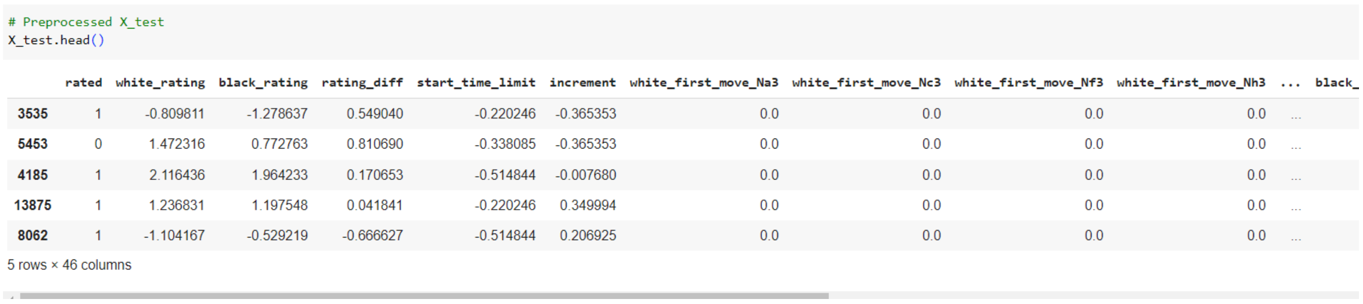


```
# Preprocessed X_train
X_train.head()
```

	rated	white_rating	black_rating	rating_diff	start_time_limit	increment	white_first_move_Na3	white_first_move_Nc3	white_first_move_Nf3	white_first_mo
7928	1	-0.570863	-0.612104	0.049892	-0.397005	0.206925	0.0	0.0	0.0	
10609	1	-0.054874	-0.100981	0.053917	-0.220246	-0.365353	0.0	0.0	0.0	
13415	0	-0.321526	-0.152784	-0.195657	-0.220246	-0.007680	0.0	0.0	0.0	
5705	0	0.332984	0.396328	-0.074895	-0.514844	0.063855	0.0	0.0	0.0	
7707	1	0.973641	0.907451	0.074044	-0.220246	-0.365353	0.0	0.0	0.0	

5 rows × 46 columns

Figure 26: Preprocessed X_train



```
# Preprocessed X_test
X_test.head()
```

	rated	white_rating	black_rating	rating_diff	start_time_limit	increment	white_first_move_Na3	white_first_move_Nc3	white_first_move_Nf3	white_first_move_Nh3	...	black_
3535	1	-0.809811	-1.278637	0.549040	-0.220246	-0.365353	0.0	0.0	0.0	0.0	...	
5453	0	1.472316	0.772763	0.810690	-0.338085	-0.365353	0.0	0.0	0.0	0.0	...	
4185	1	2.116436	1.964233	0.170653	-0.514844	-0.007680	0.0	0.0	0.0	0.0	...	
13875	1	1.236831	1.197548	0.041841	-0.220246	0.349994	0.0	0.0	0.0	0.0	...	
8062	1	-1.104167	-0.529219	-0.666627	-0.514844	0.206925	0.0	0.0	0.0	0.0	...	

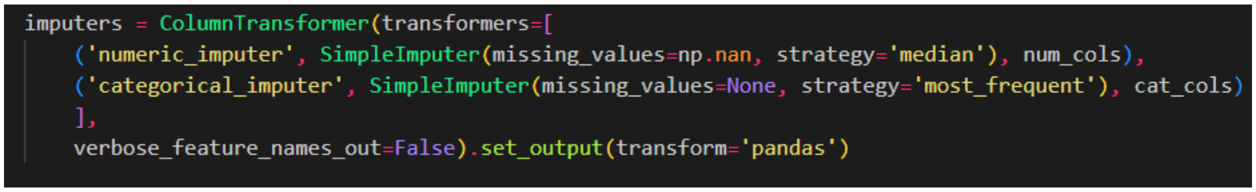
5 rows × 46 columns

Figure 27: Preprocessed X_test

Figure 26 and 27 shows the training and testing input data after preprocessing. In contrast to the data before preprocessing, our input now expanded to 46 columns from the previous 8 columns as a result of the one hot encoding. In addition the rated feature is now a binary numeric format and all numerical features are now standardized so that the mean and standard deviation of each said feature is 0 and 1 respectively.

Data Preprocessing as Pipeline

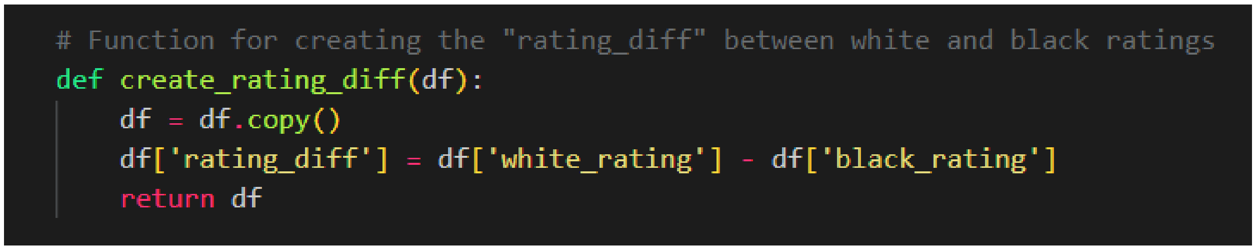
To integrate our machine learning model into an application, we need to modify the preprocessing steps outlined in the above section to align with the inputs of our application users as follows:



```python
imputers = ColumnTransformer(transformers=[
    ('numeric_imputer', SimpleImputer(missing_values=np.nan, strategy='median'), num_cols),
    ('categorical_imputer', SimpleImputer(missing_values=None, strategy='most_frequent'), cat_cols)
    ],
    verbose_feature_names_out=False).set_output(transform='pandas')
```

Figure 28: Column transformer for imputing missing values

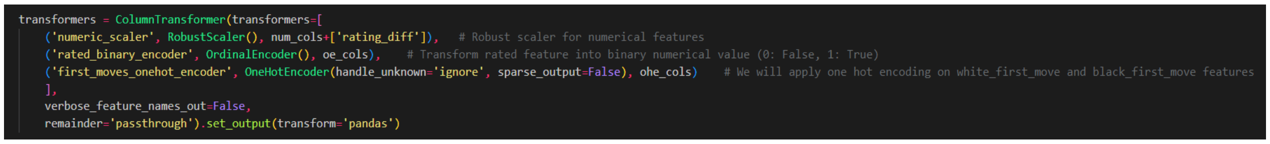
Figure: 28 shows the first step of our preprocessing pipeline which is the column transformer for imputing missing values found in each feature. Although there is no missing value in our training or testing sets, we should consider that once our model has been deployed, users may not have all the necessary information to fill all input values. Therefore, our pipeline should also include the imputers to automatically fill out missing values from users as well.



```python
# Function for creating the "rating_diff" between white and black ratings
def create_rating_diff(df):
    df = df.copy()
    df['rating_diff'] = df['white_rating'] - df['black_rating']
    return df
```

Figure 29: Function for creating the “rating_diff” between white and black ratings

Next, the figure: 29 shows the function needed for feature engineering. Based on the results obtained from the correlation heatmap, we have determined that the "rating_diff" feature should be the most important feature for our predictor. "rating_diff" however is not included in the dataset but must be inferred from "white_rating" and "black_rating" features using our feature engineering transformer.

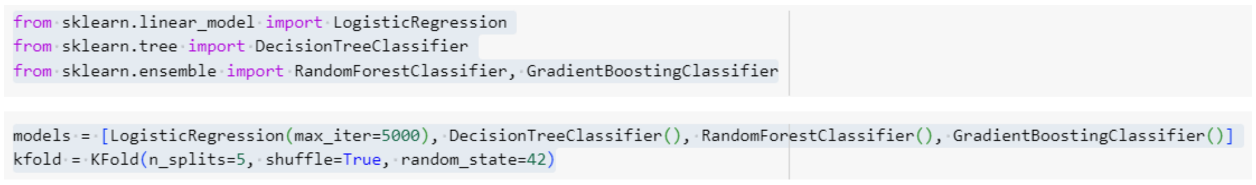


```python
transformers = ColumnTransformer(transformers=[
    ('numeric_scaler', RobustScaler(), num_cols+['rating_diff']),   # Robust scaler for numerical features
    ('rated_binary_encoder', OrdinalEncoder(), oe_cols),    # Transform rated feature into binary numerical value (0: False, 1: True)
    ('first_moves_onehot_encoder', OneHotEncoder(handle_unknown='ignore', sparse_output=False), ohe_cols)    # We will apply one hot encoding on white_first_move and black_first_move features
    ],
    verbose_feature_names_out=False,
    remainder='passthrough').set_output(transform='pandas')
```

Figure 30: Column transformer

The final step of our preprocessing pipeline is the feature transformation column transformer. As illustrated in figure: 30, the column transformer includes the following: the Robust Scaler for scaling all the numerical features (including “rating_diff”) so that our model can converge easier, the Ordinal Encoder for “rated” feature which is a binary variable and lastly, the One-Hot Encoder for “white_first_move” and “black_first_move” features since these two features each contain 20 nominal unique values (from the 20 possible first moves in a chess game) thus, it is more appropriate to separate each value into their own columns using the One-Hot Encoder.

Modeling



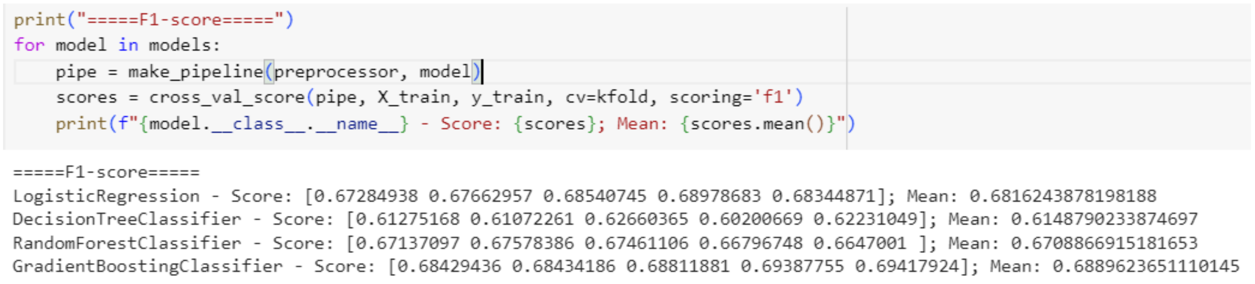
```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
```

```
models = [LogisticRegression(max_iter=5000), DecisionTreeClassifier(), RandomForestClassifier(), GradientBoostingClassifier()]
kfold = KFold(n_splits=5, shuffle=True, random_state=42)
```

Figure 28: Prepares different classification models and sets up a k-fold cross-validation strategy.

In this code(figure: 28), essential classes for various classification models (Logistic Regression, Decision Tree, Random Forest, and Gradient Boosting) are imported from the scikit-learn library. A list called "models" is then initiated, containing instances of these classifiers. Each model is set up with default parameters, and for Logistic Regression, the maximum iteration parameter (max_iter) is configured to 5000. Additionally, the code incorporates the KFold class from scikit-learn for k-fold cross-validation. The KFold object is configured with 5 splits, ensuring the dataset is divided into 5 folds for cross-validation. The shuffle=True parameter indicates data shuffling before splitting, and random_state=42 establishes a seed for reproducibility.

It is important to note that since the output we wish to obtain is the winning rate or the probabilities of the winner for each side, we have to choose the candidate classifiers that predict a target label based on probabilities of each class. Hence the reason we selected these 4 models.

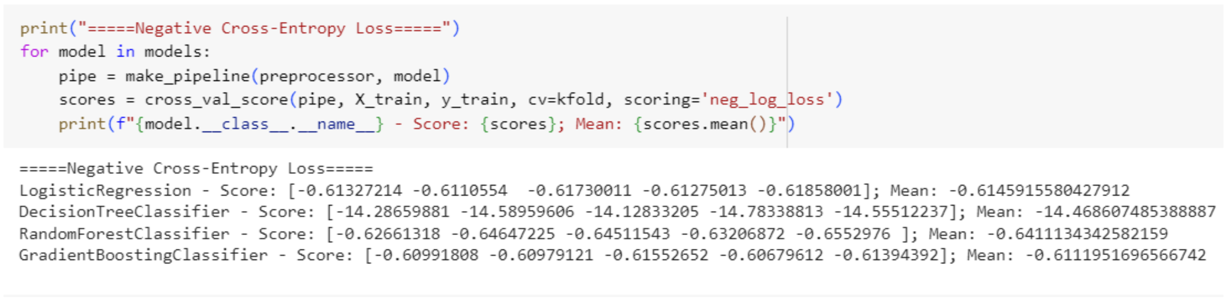


```
print("=====F1-score=====")
for model in models:
    pipe = make_pipeline(preprocessor, model)
    scores = cross_val_score(pipe, X_train, y_train, cv=kfold, scoring='f1')
    print(f"{model.__class__.__name__} - Score: {scores}; Mean: {scores.mean()}")
```

```
=====F1-score=====
LogisticRegression - Score: [0.67284938 0.67662957 0.68540745 0.68978683 0.68344871]; Mean: 0.6816243878198188
DecisionTreeClassifier - Score: [0.61275168 0.61072261 0.62660365 0.60200669 0.62231049]; Mean: 0.6148790233874697
RandomForestClassifier - Score: [0.67137097 0.67578386 0.67461106 0.66796748 0.6647001 ]; Mean: 0.6708866915181653
GradientBoostingClassifier - Score: [0.68429436 0.68434186 0.68811881 0.69387755 0.69417924]; Mean: 0.6889623651110145
```

Figure 31: evaluating and comparing the performance of different classifiers using F1-score

A higher F1 score indicates better performance. In this figure:31, the models are being evaluated using 5-fold cross-validation, and the mean F1 score is provided as a summary metric. The mean F1 scores give you an overall sense of how well each model is performing on average across different folds of the dataset. It looks like the Gradient Boosting Classifier has the highest mean F1 score, suggesting it might be the most effective model among the ones tested for this specific task.



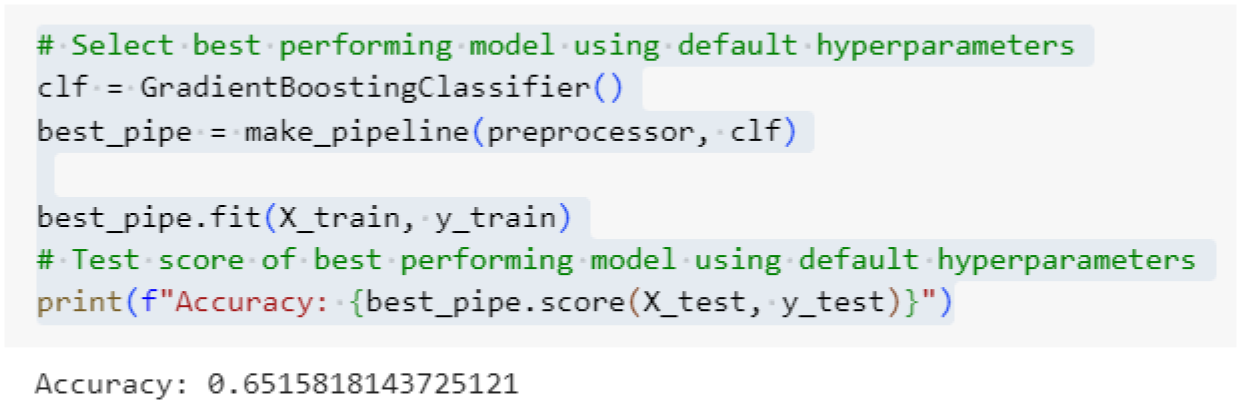
```
print("=====Negative Cross-Entropy Loss=====")
for model in models:
    pipe = make_pipeline(preprocessor, model)
    scores = cross_val_score(pipe, X_train, y_train, cv=kfold, scoring='neg_log_loss')
    print(f"{model.__class__.__name__} - Score: {scores}; Mean: {scores.mean()}")
```

```
=====Negative Cross-Entropy Loss=====
LogisticRegression - Score: [-0.61327214 -0.6110554  -0.61730011 -0.61275013 -0.61858001]; Mean: -0.6145915580427912
DecisionTreeClassifier - Score: [-14.28659881 -14.58959606 -14.12833205 -14.78338813 -14.55512237]; Mean: -14.468607485388887
RandomForestClassifier - Score: [-0.62661318 -0.64647225 -0.64511543 -0.63206872 -0.6552976 ]; Mean: -0.6411134342582159
GradientBoostingClassifier - Score: [-0.60991808 -0.60979121 -0.61552652 -0.60679612 -0.61394392]; Mean: -0.6111951696566742
```

Figure 32: Evaluating performance using cross-validation with Negative Cross-Entropy Loss

In Figure: 32 The lower the negative log loss, the better the model's performance. Therefore, Logistic Regression and Gradient Boosting seem to perform relatively well on this task, while Decision Tree has a much higher negative log loss, suggesting poorer performance.

Results:

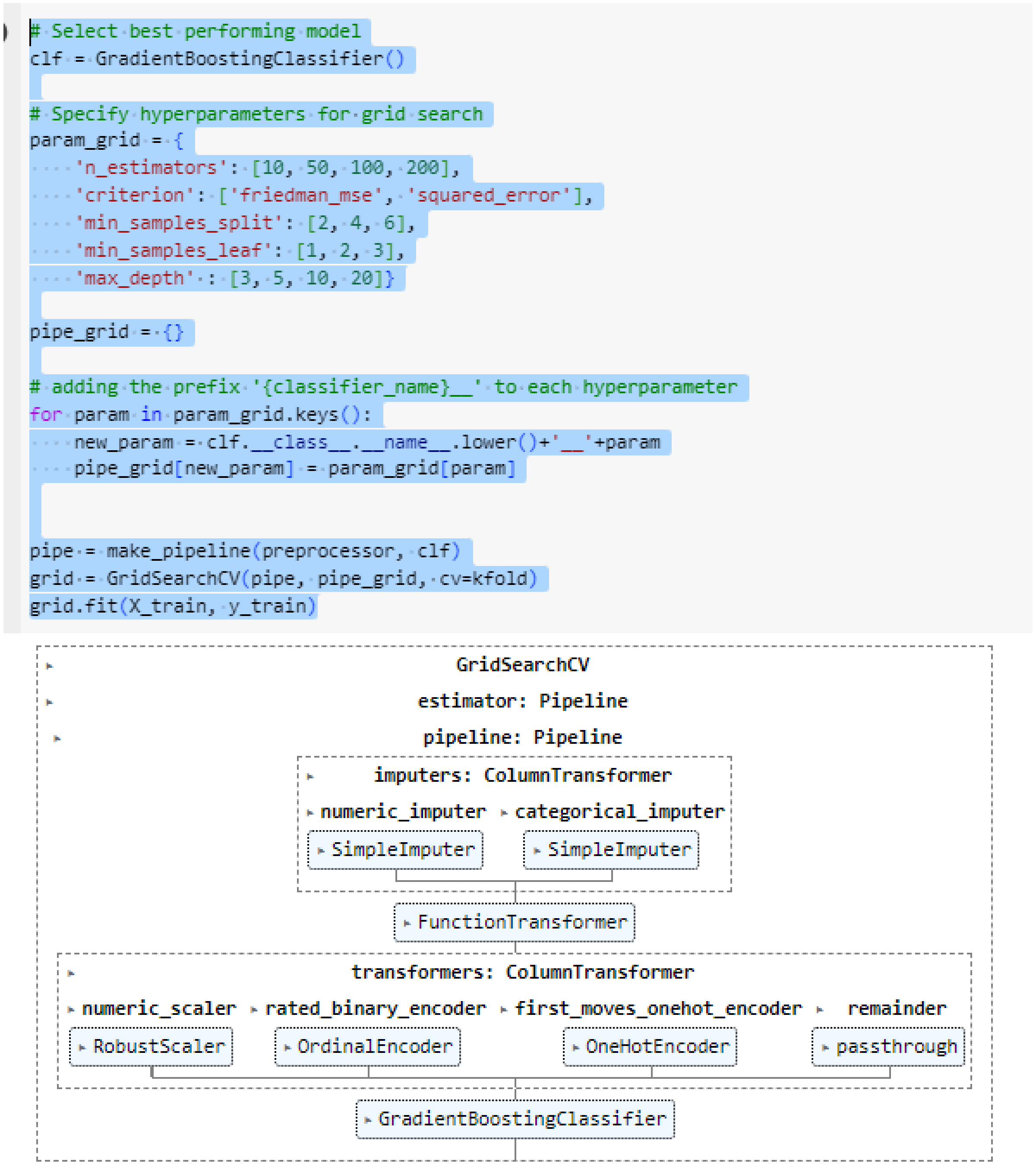


```
# Select best performing model using default hyperparameters
clf = GradientBoostingClassifier()
best_pipe = make_pipeline(preprocessor, clf)

best_pipe.fit(X_train, y_train)
# Test score of best performing model using default hyperparameters
print(f"Accuracy: {best_pipe.score(X_test, y_test)}")
```

```
Accuracy: 0.6515818143725121
```

Figure 33:Test score of best performing model using default hyperparameters

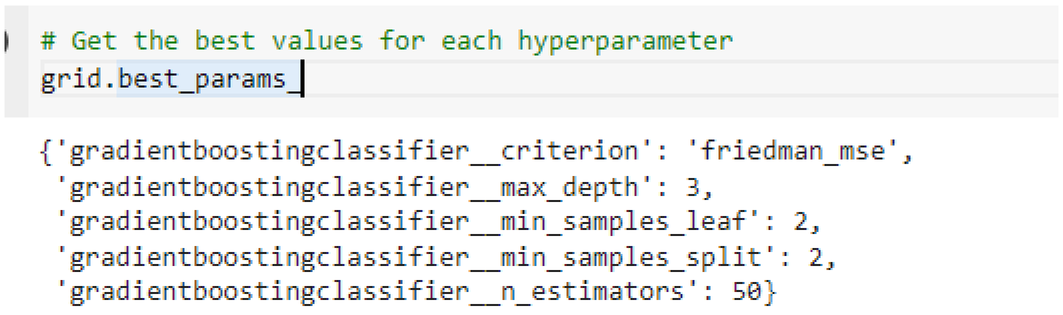
Figure 33 shows that Based on the default hyperparameters and the provided dataset, the Gradient Boosting Classifier achieved an accuracy of approximately 65.16% on the test data. The accuracy is a measure of how well the model predicts the correct outcome compared to the actual outcomes in the test set.



```
# Select best performing model
clf = GradientBoostingClassifier()

# Specify hyperparameters for grid search
param_grid = {
    'n_estimators': [10, 50, 100, 200],
    'criterion': ['friedman_mse', 'squared_error'],
    'min_samples_split': [2, 4, 6],
    'min_samples_leaf': [1, 2, 3],
    'max_depth' : [3, 5, 10, 20]}

pipe_grid = {}

# adding the prefix '{classifier_name}__' to each hyperparameter
for param in param_grid.keys():
    new_param = clf.__class__.__name__.lower()+'__'+param
    pipe_grid[new_param] = param_grid[param]


pipe = make_pipeline(preprocessor, clf)
grid = GridSearchCV(pipe, pipe_grid, cv=kfold)
grid.fit(X_train, y_train)
```

Figure 34: Hyperparameter tuning using grid search.

In Figure 34 the chosen machine learning model is the Gradient Boosting Classifier. A 'GridSearchCV' object is employed to search through a predefined set of parameters, assessing the model's performance through cross-validation. The 'cv' parameter determines the number of folds for cross-validation, and the model is trained using the provided data (X_train and y_train). The 'grid' object stores details about the best hyperparameters identified during the search and their associated performance metrics, accessible through attributes like grid.best_params_ and grid.best_score_.

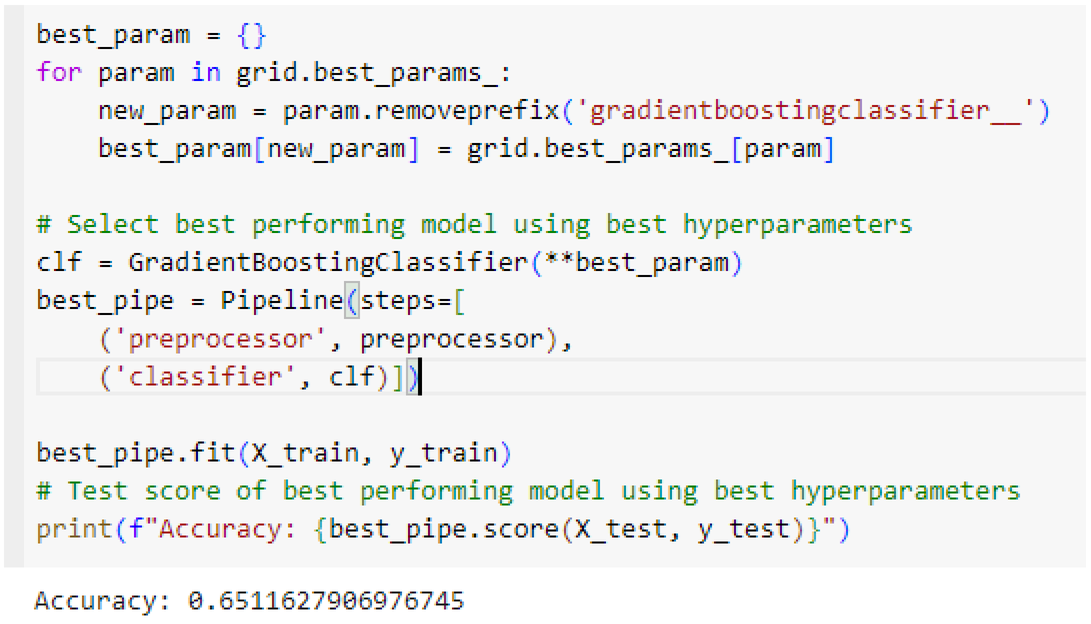


```
# Get the best values for each hyperparameter
grid.best_params_
```

```
{'gradientboostingclassifier__criterion': 'friedman_mse',
 'gradientboostingclassifier__max_depth': 3,
 'gradientboostingclassifier__min_samples_leaf': 2,
 'gradientboostingclassifier__min_samples_split': 2,
 'gradientboostingclassifier__n_estimators': 50}
```

Figure 35: Best hyperparameter values found by a hyperparameter tuning process

Figure 35 provides the hyperparameter values that produced the best performance for the Victory Vision chess winner prediction model, as determined by the grid search. These hyperparameters are used to configure the Gradient Boosting Classifier for the final model.

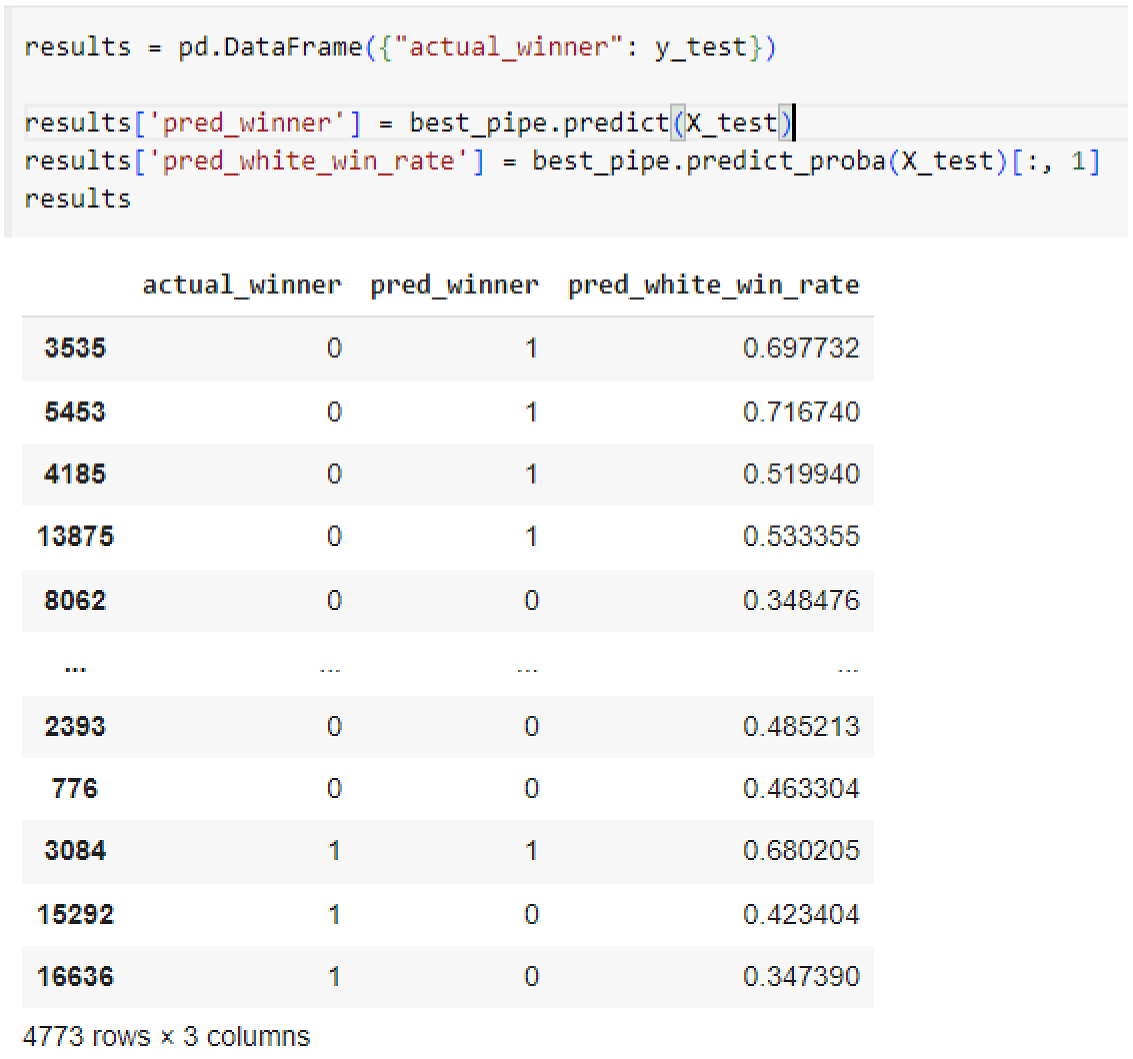


```
best_param = {}
for param in grid.best_params_:
    new_param = param.removeprefix('gradientboostingclassifier__')
    best_param[new_param] = grid.best_params_[param]

# Select best performing model using best hyperparameters
clf = GradientBoostingClassifier(**best_param)
best_pipe = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', clf)])

best_pipe.fit(X_train, y_train)
# Test score of best performing model using best hyperparameters
print(f"Accuracy: {best_pipe.score(X_test, y_test)}")
```

```
Accuracy: 0.6511627906976745
```

Figure 36: Test score of best performing model using best hyperparameters

The output(figure 36) indicates the accuracy of the model on the test set, which is approximately 0.6512 (or 65.12%). This suggests that the model performs reasonably well on the unseen data, and you can use this accuracy as an evaluation metric for the classification task.



```python
results = pd.DataFrame({"actual_winner": y_test})

results['pred_winner'] = best_pipe.predict(X_test)
results['pred_white_win_rate'] = best_pipe.predict_proba(X_test)[:, 1]
results
```

	actual_winner	pred_winner	pred_white_win_rate
3535	0	1	0.697732
5453	0	1	0.716740
4185	0	1	0.519940
13875	0	1	0.533355
8062	0	0	0.348476
...	...	...	...
2393	0	0	0.485213
776	0	0	0.463304
3084	1	1	0.680205
15292	1	0	0.423404
16636	1	0	0.347390

4773 rows × 3 columns

Figure 37: Results with actual_winner, pred_winner & pred_white_win_rate

Here(figure 37), The actual winner and the predicted winner of the chess game by the model (0 for Black, 1 for White). The predicted probability of White winning the game. Prediction Probability: The pred_white_win_rate provides the predicted probability of White winning the game. This can be used to assess the model's confidence in its predictions. The results obtained from the methodology indicate the model's ability to predict the winner of chess games based on the features used in the training. Further analysis and fine-tuning may be necessary to improve the model's performance and ensure its reliability in real-world scenarios.

After implementing the Gradient Boosting Classifier on the dataset containing information from over 20,000 chess matches, the predictive model demonstrated an accuracy level of 65-66%. The model's performance was evaluated using a variety of features, including game-specific details, player ratings, and move information. The achieved accuracy indicates a moderate level of success in predicting the binary outcome of chess games (win or lose).

Conclusion:

The development and application of a predictive model for estimating the winning rate of chess games represent a significant step towards providing valuable insights to players, coaches, and enthusiasts. When compared to random guessing which has an accuracy of 50%, the obtained accuracy of 65-66% suggests that the model has a reasonable ability to assess the probability of winning a chess game based on the features included in the dataset.

This tool holds the potential to enhance player performance by offering strategic guidance and aiding in decision-making during games. Additionally, its application extends to the realms of learning and teaching, enabling players of all levels to deepen their understanding of the game. Chess enthusiasts and amateurs can also benefit from the tool for recreational purposes, practice, and self-improvement.

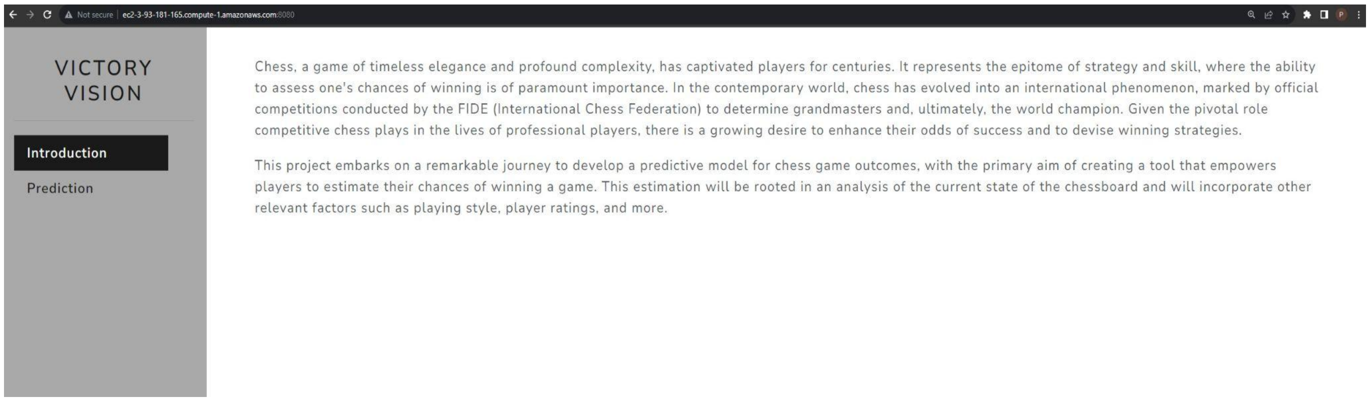
Future Work:

While the current accuracy level is promising, there is room for improvement in the predictive model. Future work should focus on fine-tuning the model parameters, exploring additional features, and incorporating advanced machine learning techniques to enhance its performance. Additionally, gathering more diverse and extensive datasets could contribute to a more robust and generalizable model.

Furthermore, user feedback and engagement should be considered for iterative improvements. Continuous refinement of the model based on real-world usage and feedback from players and coaches will ensure its relevance and effectiveness in the dynamic landscape of competitive chess.

Our Website:

First page of our website is Introduction


Not secure | ec2-3-93-181-165.compute-1.amazonaws.com:8080

VICTORY VISION

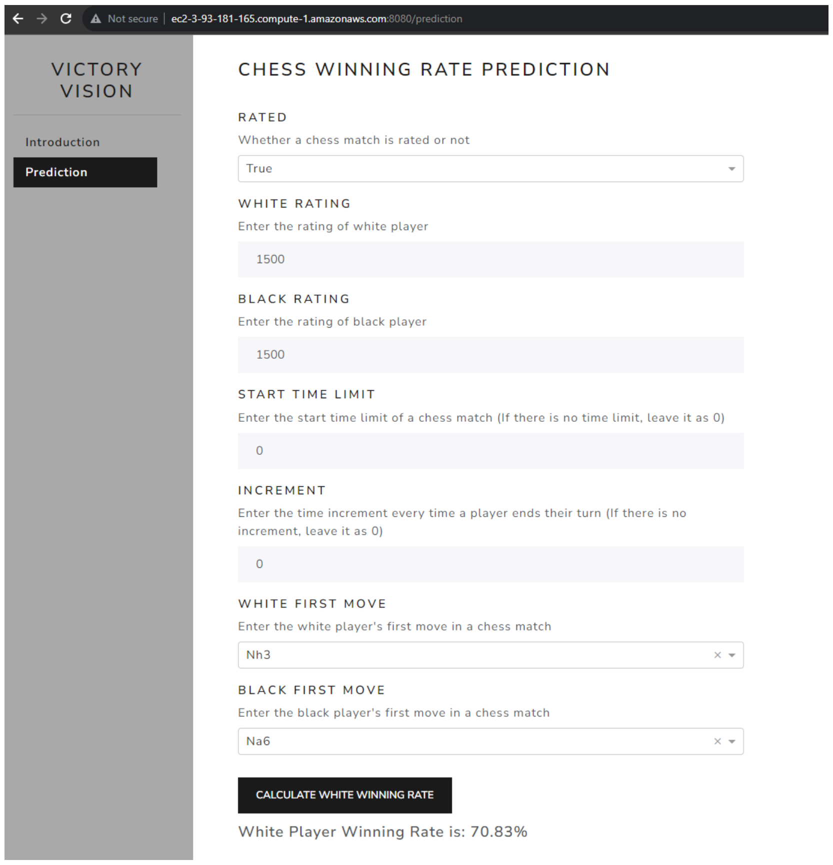
Introduction

Prediction

Chess, a game of timeless elegance and profound complexity, has captivated players for centuries. It represents the epitome of strategy and skill, where the ability to assess one's chances of winning is of paramount importance. In the contemporary world, chess has evolved into an international phenomenon, marked by official competitions conducted by the FIDE (International Chess Federation) to determine grandmasters and, ultimately, the world champion. Given the pivotal role competitive chess plays in the lives of professional players, there is a growing desire to enhance their odds of success and to devise winning strategies.

This project embarks on a remarkable journey to develop a predictive model for chess game outcomes, with the primary aim of creating a tool that empowers players to estimate their chances of winning a game. This estimation will be rooted in an analysis of the current state of the chessboard and will incorporate other relevant factors such as playing style, player ratings, and more.

Second page of our website is Prediction


Not secure | ec2-3-93-181-165.compute-1.amazonaws.com:8080/prediction

VICTORY VISION

Introduction

Prediction

CHESS WINNING RATE PREDICTION

RATED

Whether a chess match is rated or not

True

WHITE RATING

Enter the rating of white player

1500

BLACK RATING

Enter the rating of black player

1500

START TIME LIMIT

Enter the start time limit of a chess match (If there is no time limit, leave it as 0)

0

INCREMENT

Enter the time increment every time a player ends their turn (If there is no increment, leave it as 0)

0

WHITE FIRST MOVE

Enter the white player's first move in a chess match

Nh3

BLACK FIRST MOVE

Enter the black player's first move in a chess match

Na6

CALCULATE WHITE WINNING RATE

White Player Winning Rate is: 70.83%