Analysis of Chess Competitions Between Different Countries

Roie Kazoom

This project focused on analyzing chess games data with an emphasis on understanding trends and insights at the country level. We began by merging multiple datasets and imputing missing country information using fuzzy matching techniques. This ensured that players from various countries were accurately represented in our analysis.

Key steps included processing data to handle mixed types in columns like "Round" and encoding categorical variables using label encoding and dummy variables. These preprocessing steps allowed us to perform detailed statistical and visual analyses.

Through various visualizations such as pie charts, histograms, heatmaps, and scatter plots, we examined the distribution of player characteristics and game outcomes across different countries. Significant observations included:

High Frequency of Games in Russia: Both pie charts and bar plots showed that Russia played the most games, highlighting its prominent role in the chess world.

Elo Ratings and Age Distributions: We noted that Elo ratings and player ages were similarly distributed across countries, with some outliers. For example, Scotland had the oldest average player age, while Tanzania had the youngest.

Impact of COVID-19: There was a noticeable increase in online games around 2019, likely due to the COVID-19 pandemic, which shifted many competitions to virtual platforms.

Using the Stockfish chess engine, we analyzed specific games to evaluate positions and track performance trends over time. This detailed evaluation provided deeper insights into the strategies and decision-making processes of players from different countries.

In terms of predictive modeling, we implemented various machine learning techniques, including ensemble methods such as the VotingClassifier, which combined RandomForestClassifier, GradientBoostingClassifier, and LogisticRegression. These models were used to predict game outcomes based on the extensive dataset of chess games.

Our findings also highlighted trends in chess openings, with the Sicilian Defense being the most common opening among the top 15 most played countries since 2015. Additionally, we observed that the Elo ratings of players from Tanzania increased over time, indicating a possible improvement in the skill levels of players from that country.

To find the best predictive model, we explored multiple architectures of deep learning models. Each architecture represents a different configuration of layers, neurons, activation functions, and other parameters defining the neural network's structure and behavior. We conducted experiments where each architecture was trained and tested 16 times with different combinations of hyperparameters. This systematic approach allowed us to thoroughly explore the performance landscape and identify the optimal combination for the best results.

Hyperparameter optimization is a computation-intensive process, often requiring multiple training runs. To efficiently manage this, we used embarrassingly parallel algorithms and Dask for parallel computation and task scheduling. This approach enabled us to run independent neural network functions in parallel, significantly improving execution time and allowing us to explore a wide range of hyperparameters effectively.

Overall, this project demonstrates the effectiveness of combining data analysis, machine learning, and deep learning to gain insights into chess at the country level. The methodologies and findings can help inform future research in chess analytics and contribute to understanding the global landscape of chess competition.

1. ***Complete Players Database DataFrame:***

   Columns: Country, Rank, Name, Title, FIDE, Age, K-factor Provides a comprehensive list of chess players from various countries along with their rankings, titles, FIDE ratings, and ages.

2. ***Ranking DataFrame:***

   Columns: name, rank, country, title, profilelink, profileimage, classicalrating, rapidrating, blitzrating, lastupdated Contains rankings and ratings of top players, along with links to their profiles.

3. ***International Chess Stats DataFrame:***

   Columns: Unnamed: 0, #, Country, Flag, Num Players, Women, % of Women, FIDE Average, GMs, IMs, FMs, WGMs, WIMs, WFMs, Age Avg Contains statistics related to international chess, including the number of players, gender distribution, average FIDE rating, and average age, organized by country.

In [1]:
```
import pandas as pd

# Load Complete PLayers Database CSV file
complete_players_df = pd.read_csv("Complete_Players_Database.csv")
print("Complete Players Database DataFrame:")
complete_players_df.head()
```

```
Complete Players Database DataFrame:

C:\Users\kazom\AppData\Local\Temp\ipykernel_10096\2375703240.py:4: DtypeWarning: Columns (4) have mixed types. Specify dtype option on import or set low_memory=False.
  complete_players_df = pd.read_csv("Complete_Players_Database.csv")
```

Out[1]:

	Country Rank	Name	Title	Country	FIDE	Age	K-factor
0	1	Mirzaad, S.wahabuddin	FM	Afghanistan	1999.0	35	20
1	2	Rahmani, Asef	unranked/unrated	Afghanistan	1871.0	49	20
2	3	Sarwari, Hamidullah	unranked/unrated	Afghanistan	1866.0	33	20
3	4	Sakhawaty, Sepehr	unranked/unrated	Afghanistan	1846.0	19	20
4	5	Jamshedy, Mohammad Ismail	unranked/unrated	Afghanistan	1790.0	78	20

In [2]:
```
# Load Ranking CSV file
ranking_df = pd.read_csv("Ranking.csv")
print("Ranking DataFrame:")
ranking_df.head()
```

```
Ranking DataFrame:
```

Out[2]:

	name	rank	country	title	profilelink	profileimage	classicalrating	rapidrating	blitzrating	lastupdated
0	Magnus Carlsen	1	Norway	GM	https://www.chess.com/players/magnus-carlsen	https://images.chesscomfiles.com/uploads/v1/ma...	2830	2818	2887	05-12-2023
1	Fabiano Caruana	2	United States	GM	https://www.chess.com/players/fabiano-caruana	https://images.chesscomfiles.com/uploads/v1/ma...	2804	2762	2815	05-12-2023
2	Hikaru Nakamura	3	United States	GM	https://www.chess.com/players/hikaru-nakamura	https://images.chesscomfiles.com/uploads/v1/ma...	2788	2731	2874	05-12-2023
3	Ding Liren	4	China	GM	https://www.chess.com/players/ding-liren	https://images.chesscomfiles.com/uploads/v1/ma...	2780	2830	2787	05-12-2023
4	Ian Nepomniachtchi	5	Russia	GM	https://www.chess.com/players/ian-nepomniachtchi	https://images.chesscomfiles.com/uploads/v1/ma...	2769	2778	2795	05-12-2023

This dataset contain information about top chess players. Here's an explanation of each column:

name: Name of the chess player.

rank: Rank of the player.

country: Country the player represents.

title: Title of the player (usually GM for Grandmaster).

profilelink: Link to the player's profile.

profileimage: Link to the player's profile image.

classicalrating: Classical rating of the player.

rapidrating: Rapid rating of the player.

blitzrating: Blitz rating of the player.

lastupdated: Date when the player's information was last updated.

This dataset seems to be focused on top chess players, including their rankings, ratings in different formats (classical, rapid, blitz), and some additional information like their country and profile links. It could be used for various analyses related to chess player performance, country-wise comparisons, or trends over time.

In [3]:
```
# Convert 'lastupdated' column to datetime
ranking_df['lastupdated'] = pd.to_datetime(ranking_df['lastupdated'], format='%d-%m-%Y')

# Sort the DataFrame by 'name' and 'lastupdated' columns in descending order
ranking_df_sorted = ranking_df.sort_values(by=['name', 'lastupdated'], ascending=[True, False])

# Drop duplicates keeping the first occurrence
ranking_df_unique = ranking_df_sorted.drop_duplicates('name', keep='first')

# Display the DataFrame with the latest last updated date for each duplicated 'name'
print("DataFrame with the latest last updated date for each duplicated 'name':")
ranking_df_unique.head()
```

```
DataFrame with the latest last updated date for each duplicated 'name':
```

Out[3]:

	name	rank	country	title	profilelink	profileimage	classicalrating	rapidrating	blitzrating	lastupdated
397990	A Daurimbetov	11843	Uzbekistan	FM	https://www.chess.com/players/a-daurimbetov	NaN	2134	2228	2198	2023-12-29
398315	A G Nimmy	12168	India	WFM	https://www.chess.com/players/a-g-nimmy	NaN	2129	2057	0	2023-12-29
393442	A H Al-Ali Noah	7295	Iraq	IM	https://www.chess.com/players/a-h-al-ali-noah	NaN	2213	2391	2283	2023-12-29
390545	A Izrailev	4398	Russia	NaN	https://www.chess.com/players/a-izrailev	NaN	2292	2253	2309	2023-12-29
401105	A Kalinin	14958	Ukraine	IM	https://www.chess.com/players/a-kalinin	NaN	2091	1885	2023	2023-12-29

In [4]:
```
# Load International Chess Stats CSV file
intl_chess_stats_df = pd.read_csv("International_Chess_Stats.csv")
print("International Chess Stats DataFrame:")
intl_chess_stats_df.head()
```

```
International Chess Stats DataFrame:
```

Out[4]:

	Unnamed: 0	#	Country	Flag	Num Players	Women	% of Women	FIDE Average	GMs	IMs	FMs	WGMs	WIMs	WFMs	Age Avg
0	0	1	Russia	NaN	34497	5734	16.62	1666	236	522	1177	50	101	409	34
1	1	2	India	NaN	32735	3581	10.94	1275	64	114	83	9	41	42	27
2	2	3	Germany	NaN	26577	1751	6.59	1841	94	273	861	18	40	69	49
3	3	4	Spain	NaN	25009	1430	5.72	1429	55	134	365	2	14	37	42
4	4	5	France	NaN	23784	2143	9.01	1580	50	117	234	4	18	21	41

This dataset provide information about chess players from different countries. Here's an explanation of each column:

#: Rank or position of the country based on some criteria.

Country: Name of the country.

Flag: This column seems to indicate whether there's a flag associated with the country, but it appears to be empty (NaN).

Num Players: Total number of chess players from the country.

Women: Number of female chess players from the country.

% of Women: Percentage of female players among all players from the country.

FIDE Average: Average FIDE (Fédération Internationale des Échecs) rating of players from the country.

GMs: Number of Grandmasters from the country.

IMs: Number of International Masters from the country.

FMs: Number of FIDE Masters from the country.

WGMs: Number of Women Grandmasters from the country.

WIMs: Number of Women International Masters from the country.

WFMs: Number of Women FIDE Masters from the country.

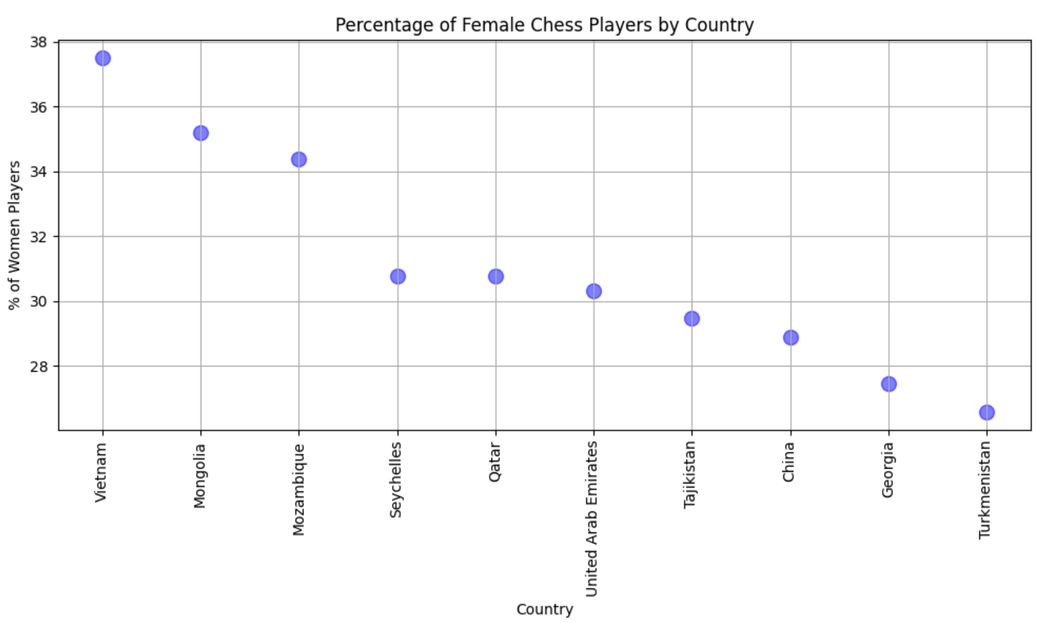
Age Avg: Average age of chess players from the country.

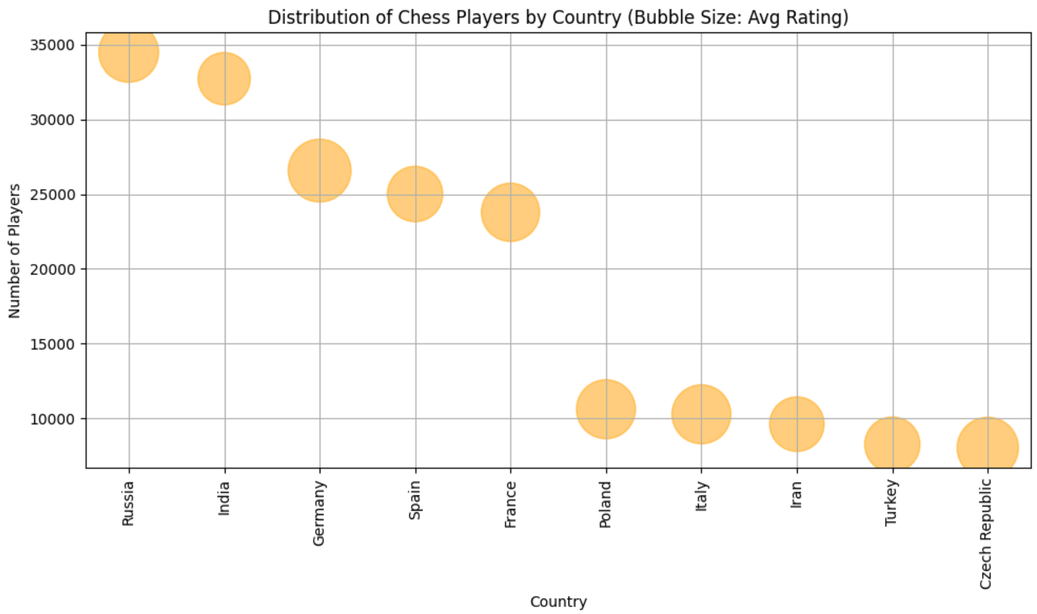
This dataset provides insights into the distribution of chess players across countries, including the number of players, gender distribution, average ratings, and the presence of titled players like Grandmasters and International Masters. It could be useful for analyzing trends in chess participation, gender diversity, and the strength of chess communities in different countries.

In [5]:

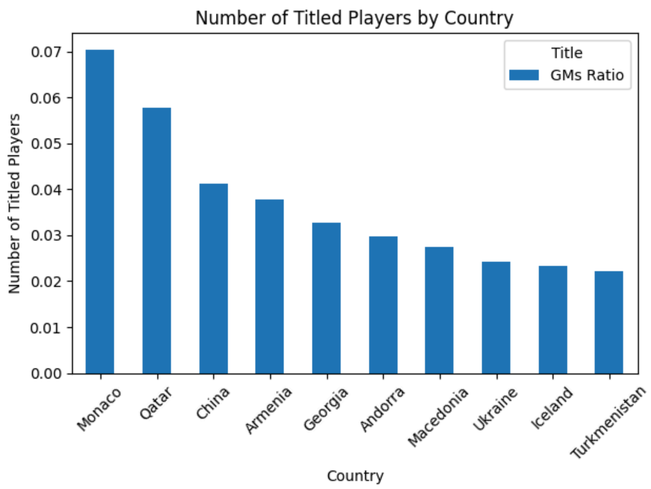
```
import matplotlib.pyplot as plt

# Convert the 'Country' column to strings
intl_chess_stats_df['Country'] = intl_chess_stats_df['Country'].astype(str)

# Visualization 1: Choropleth Map of Female Representation
plt.figure(figsize=(10, 6))
plt.scatter(intl_chess_stats_df.sort_values('% of Women', ascending = False).head(10)['Country'],
            intl_chess_stats_df.sort_values('% of Women', ascending = False).head(10)['% of Women'],
            s=100, c='blue', alpha=0.5)
plt.xticks(rotation=90)
plt.xlabel('Country')
plt.ylabel('% of Women Players')
plt.title('Percentage of Female Chess Players by Country')
plt.grid(True)
plt.tight_layout()
plt.show()


# Visualization 2: Bubble Chart of Player Distribution
plt.figure(figsize=(10, 6))
plt.scatter(intl_chess_stats_df.sort_values('Num Players', ascending = False).head(10)['Country'],
            intl_chess_stats_df.sort_values('Num Players', ascending = False).head(10)['Num Players'],
            s=intl_chess_stats_df.sort_values('Num Players', ascending = False).head(10)['FIDE Average'],
            c='orange', alpha=0.5)
plt.xticks(rotation=90)
plt.xlabel('Country')
plt.ylabel('Number of Players')
plt.title('Distribution of Chess Players by Country (Bubble Size: Avg Rating)')
plt.grid(True)
plt.tight_layout()
plt.show()

# Visualization 3: Stacked Bar Chart of Titled Players
intl_chess_stats_df['Total Gms'] = intl_chess_stats_df['GMs'] + intl_chess_stats_df['WGMs']
intl_chess_stats_df['GMs Ratio'] = intl_chess_stats_df['Total Gms'] / intl_chess_stats_df['Num Players']

titled_players = intl_chess_stats_df.sort_values('GMs Ratio', ascending = False).head(10)[['GMs Ratio',
                                                                                           'Country']].set_index('Country')

plt.figure(figsize=(10, 6))
titled_players.plot(kind='bar', stacked=True)
plt.xticks(rotation=45)
plt.xlabel('Country')
plt.ylabel('Number of Titled Players')
plt.title('Number of Titled Players by Country')
plt.legend(title='Title')
plt.tight_layout()
plt.show()
```

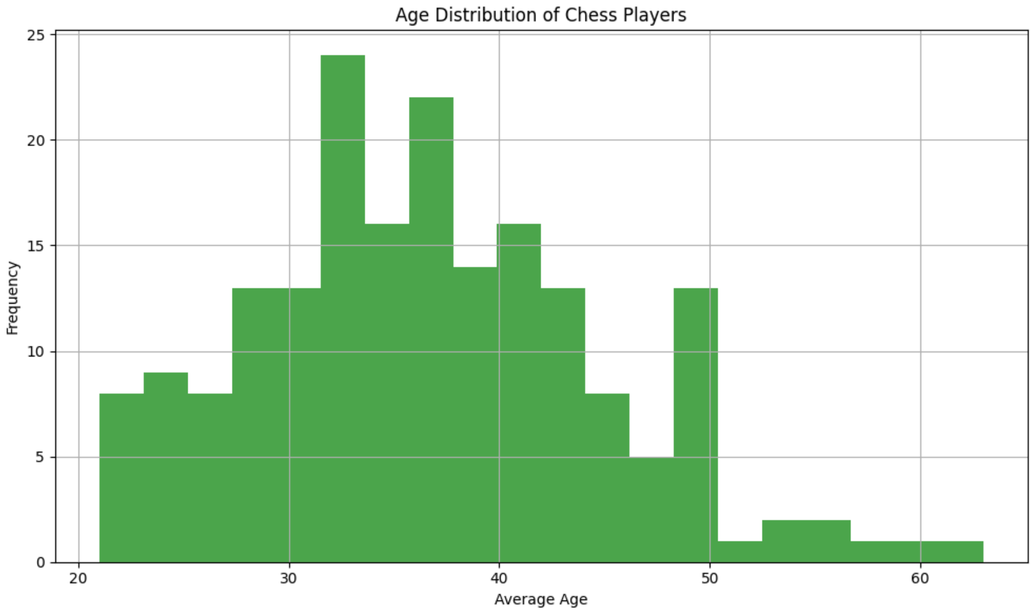
```
# Visualization 4: Age Distribution Histogram
plt.figure(figsize=(10, 6))
plt.hist(intl_chess_stats_df['Age Avg'], bins=20, color='green', alpha=0.7)
plt.xlabel('Average Age')
plt.ylabel('Frequency')
plt.title('Age Distribution of Chess Players')
plt.grid(True)
plt.tight_layout()
plt.show()

# Visualization 5: Scatter Plot of Female Representation vs. Average Rating
plt.figure(figsize=(10, 6))
plt.scatter(intl_chess_stats_df['% of Women'], intl_chess_stats_df['FIDE Average'], s=intl_chess_stats_df['Num Players'], c='red',
            alpha=0.5)
plt.xlabel('% of Women Players')
plt.ylabel('Average FIDE Rating')
plt.title('Female Representation vs. Average Rating by Country')
plt.grid(True)
plt.tight_layout()
plt.show()
```

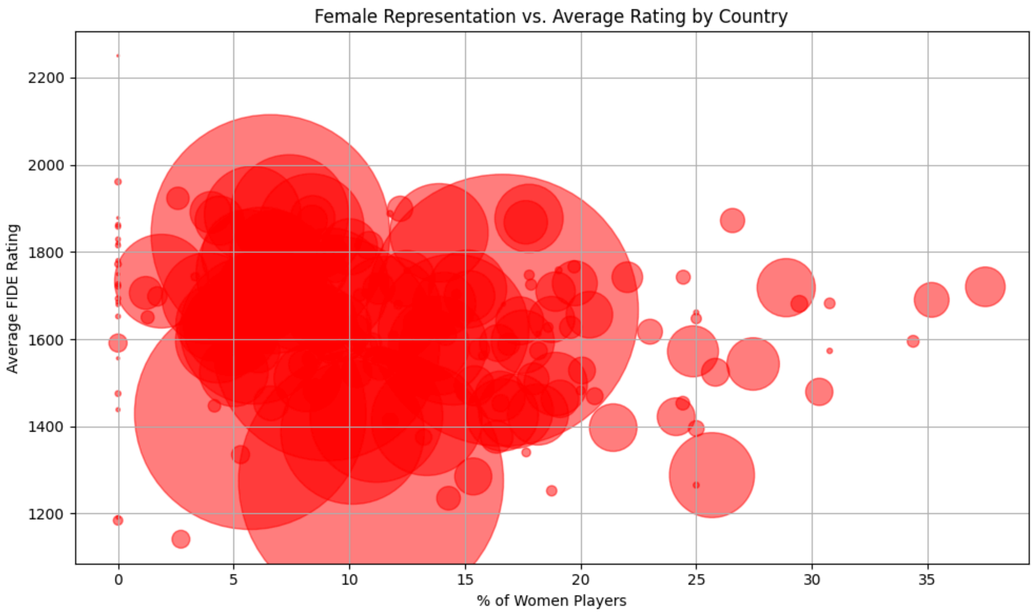




<Figure size 1000x600 with 0 Axes>







These interpretations provide valuable insights into the dataset and the visualizations:

Choropleth Map of Female Representation: The visualization highlights that Vietnam has the highest percentage of female players compared to other countries. This suggests a potentially strong representation of women in the chess community in Vietnam.

Bubble Chart of Player Distribution: The visualization indicates that Russia has the highest number of chess players among all countries. This reflects Russia's historically strong presence in the world of chess, with a large and active chess-playing population.

Stacked Bar Chart of Titled Players: The observation about Monaco having the highest ratio of Grandmasters (GMs) relative to the total number of players is interesting. This suggests that Monaco may have a smaller but highly skilled chess community, with a notable proportion of titled players.

Age Distribution Histogram: The visualization suggests that the age distribution of chess players is fairly uniform across countries. This implies that chess is a game enjoyed by people of various age groups and is not limited to any particular demographic.

Scatter Plot of Female Representation vs. Average Rating: The observation about a cluster of data points around 10% female representation and an average FIDE rating of around 1600 is intriguing. This suggests that there may be certain common characteristics among countries where the percentage of female players is around 10% and the average skill level is at this particular rating. Further analysis could be conducted to explore the reasons behind this trend.

These interpretations provide a deeper understanding of the demographics, skill levels, and distribution of players within the global chess community, as revealed by the visualizations.

In [6]:
```
print("Column names in complete_players_df:", complete_players_df.columns)
print("Column names in ranking_df:", ranking_df.columns)
```

```
Column names in complete_players_df: Index(['Country Rank', 'Name', 'Title', 'Country', 'FIDE', 'Age', ' K-factor'], dtype='object')
Column names in ranking_df: Index(['name', 'rank', 'country', 'title', 'profilelink', 'profileimage',
       'classicalrating', 'rapidrating', 'blitzrating', 'lastupdated'],
      dtype='object')
```

This code will merge the DataFrames on the 'name' column, combining them into a single DataFrame called merged_df. The outer merge ensures that all rows from each DataFrame are included in the final result.

In [7]:
```
# Rename the 'Name' column in complete_players_df to 'name'
complete_players_df.rename(columns={'Name': 'name'}, inplace=True)

# Merge DataFrames on the 'name' column
merged_df = pd.merge(complete_players_df, ranking_df_unique, on='name', how='outer')

# Display the merged DataFrame
print("Merged DataFrame:")
merged_df.head()
```

```
Merged DataFrame:
```

Out[7]:

	Country Rank	name	Title	Country	FIDE	Age	K-factor	rank	country	title	profilelink	profileimage	classicalrating	rapidrating	blitzrating	lastupdated
0	1.0	Mirzaad, S.wahabuddin	FM	Afghanistan	1999.0	35.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaT
1	2.0	Rahmani, Asef	unranked/unrated	Afghanistan	1871.0	49.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaT
2	3.0	Sarwari, Hamidullah	unranked/unrated	Afghanistan	1866.0	33.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaT
3	4.0	Sakhawaty, Sepehr	unranked/unrated	Afghanistan	1846.0	19.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaT
4	5.0	Jamshedy, Mohammad Ismail	unranked/unrated	Afghanistan	1790.0	78.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaT

In [8]:
```
l = list(complete_players_df[complete_players_df.duplicated('name', keep=False)].sort_values(by='name')['name'])
ranking_df_unique[ranking_df_unique['name'].isin(l)]
```

Out[8]:

	name	rank	country	title	profilelink	profileimage	classicalrating	rapidrating	blitzrating	lastupdated
399126	Harshini A	12979	India	WFM	https://www.chess.com/players/harshini-a	NaN	2117	2049	1960	2023-12-29
395869	Praveen Kumar C	9722	India	IM	https://www.chess.com/players/praveen-kumar-c	NaN	2167	2080	2141	2023-12-29

In [9]:
```
merged_df.columns
```

Out[9]:
```
Index(['Country Rank', 'name', 'Title', 'Country', 'FIDE', 'Age', ' K-factor',
       'rank', 'country', 'title', 'profilelink', 'profileimage',
       'classicalrating', 'rapidrating', 'blitzrating', 'lastupdated'],
      dtype='object')
```

In [10]:
```
# Create a boolean mask to identify rows where all 'title', 'title_x', 'title_y', and 'Title' are NaN
missing_all_titles_mask = merged_df[['title', 'Title']].isna().all(axis=1)

# Filter the DataFrame using the boolean mask
missing_all_titles = merged_df[missing_all_titles_mask]

# Display the observations where all 'title' and 'Title' are missing
print("Observations where all 'Title' are missing:")
missing_all_titles
```

```
Observations where all 'Title' are missing:
```

Out[10]:

	Country Rank	name	Title	Country	FIDE	Age	K-factor	rank	country	title	profilelink	profileimage	classicalrating	rapidrating	blitzrating	lastupdated
175361	NaN	A Izrailev	NaN	NaN	NaN	NaN	NaN	4398.0	Russia	NaN	https://www.chess.com/players/a-izrailev	NaN	2292.0	2253.0	2309.0	2023-12-29
175364	NaN	A Kupriyanov	NaN	NaN	NaN	NaN	NaN	5945.0	Russia	NaN	https://www.chess.com/players/a-kupriyanov	NaN	2245.0	2190.0	2172.0	2023-12-29
175368	NaN	A R Kheiri	NaN	NaN	NaN	NaN	NaN	16811.0	Iran	NaN	https://www.chess.com/players/a-r-kheiri	NaN	2068.0	0.0	0.0	2023-12-29
175370	NaN	A Severijnen	NaN	NaN	NaN	NaN	NaN	16123.0	Netherlands	NaN	https://www.chess.com/players/a-severijnen	NaN	2076.0	0.0	0.0	2023-12-29
175371	NaN	A. Ayapbergenov	NaN	NaN	NaN	NaN	NaN	12869.0	Kazakhstan	NaN	https://www.chess.com/players/a-ayapbergenov	NaN	2118.0	2199.0	0.0	2023-12-29
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
193321	NaN	Zvonimir Radic	NaN	NaN	NaN	NaN	NaN	14947.0	Serbia	NaN	https://www.chess.com/players/zvonimir-radic	NaN	2092.0	2150.0	2140.0	2023-12-29
193322	NaN	Zvonimir Tomicic	NaN	NaN	NaN	NaN	NaN	11299.0	Croatia	NaN	https://www.chess.com/players/zvonimir-tomicic	NaN	2143.0	0.0	2169.0	2023-12-29
193323	NaN	Zvonko Babic	NaN	NaN	NaN	NaN	NaN	9461.0	Croatia	NaN	https://www.chess.com/players/zvonko-babic	NaN	2172.0	2168.0	2270.0	2023-12-29
193324	NaN	Zvonko Krizanovic	NaN	NaN	NaN	NaN	NaN	8634.0	Croatia	NaN	https://www.chess.com/players/zvonko-krizanovic	NaN	2187.0	0.0	0.0	2023-12-29

In [11]:
```
# Combine 'Country', 'country_y', and 'country_x' into one column 'Country'
merged_df['Country'] = merged_df['Country'].combine_first(merged_df['country'])

# Drop the redundant columns
merged_df.drop(columns=['country'], inplace=True)

# Combine 'title_x', and 'title_y' into one column 'Title'
merged_df['Title'] = merged_df['Title'].combine_first(merged_df['title'])

# Drop the redundant columns
merged_df.drop(columns=['title'], inplace=True)

# Display the updated DataFrame
print("Updated Merged DataFrame:")
merged_df.head()
```

```
Updated Merged DataFrame:
```

Out[11]:

	Country Rank	name	Title	Country	FIDE	Age	K-factor	rank	profilelink	profileimage	classicalrating	rapidrating	blitzrating	lastupdated
0	1.0	Mirzaad, S.wahabuddin	FM	Afghanistan	1999.0	35.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
1	2.0	Rahmani, Asef	unranked/unrated	Afghanistan	1871.0	49.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
2	3.0	Sarwari, Hamidullah	unranked/unrated	Afghanistan	1866.0	33.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
3	4.0	Sakhawaty, Sepehr	unranked/unrated	Afghanistan	1846.0	19.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
4	5.0	Jamshedy, Mohammad Ismail	unranked/unrated	Afghanistan	1790.0	78.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT

In [12]:
```
merged_df.columns
```

Out[12]:
```
Index(['Country Rank', 'name', 'Title', 'Country', 'FIDE', 'Age', ' K-factor',
       'rank', 'profilelink', 'profileimage', 'classicalrating', 'rapidrating',
       'blitzrating', 'lastupdated'],
      dtype='object')
```

In [13]:
```
print(merged_df['Country'].isna().sum())
print(merged_df['Title'].isna().sum())
```

```
0
9141
```

In [14]:
```
merged_df[merged_df['name'].str.contains('Carlsen')]
```

Out[14]:

	Country Rank	name	Title	Country	FIDE	Age	K-factor	rank	profilelink	profileimage	classicalrating	rapidrating	blitzrating	lastupdated
29532	954.0	Carlsen, Erik	unranked/unrated	Denmark	1856.0	67.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
30689	2103.0	Carlsen, Renny Staehr	unranked/unrated	Denmark	1534.0	54.0	40.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
101200	1.0	Carlsen, Magnus	GM	Norway	2864.1	32.0	10.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
101373	174.0	Carlsen, Christian Heen	unranked/unrated	Norway	2101.0	44.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
101527	328.0	Carlsen, Ellen Oen	unranked/unrated	Norway	1939.0	33.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
102151	953.0	Carlsen, Ingrid Oen	unranked/unrated	Norway	1598.0	28.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
102754	1559.0	Carlsen, Christian	unranked/unrated	Norway	1377.0	39.0	40.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
178270	NaN	Christian Heen Carlsen	NaN	Norway	NaN	NaN	NaN	16641.0	https://www.chess.com/players/christian-heen-c...	NaN	2070.0	1963.0	1977.0	2023-12-29
185501	NaN	Magnus Carlsen	GM	Norway	NaN	NaN	NaN	1.0	https://www.chess.com/players/magnus-carlsen	https://images.chesscomfiles.com/uploads/v1/ma...	2830.0	2808.0	2887.0	2023-12-29

In [15]:
```
# Import Necessary Library
import numpy as np
import matplotlib.pyplot as plt
import chess
import chess.pgn
import seaborn as sns

pd.set_option('display.max_columns', 50)
plt.style.use('seaborn-colorblind')
pal = sns.color_palette()
```

```
C:\Users\kazom\AppData\Local\Temp\ipykernel_10096\1448446720.py:9: MatplotlibDeprecationWarning: The seaborn styles shipped by Matplotlib are deprecated since 3.6, as they no longer correspond to the styles shipped by seaborn. However, they will remain available as 'seaborn-v0_8-<style>'. Alternatively, directly use the seaborn API instead.
  plt.style.use('seaborn-colorblind')
```

In [16]:
```
df = pd.read_csv('twic_master.csv')
```

```
C:\Users\kazom\AppData\Local\Temp\ipykernel_10096\2936042520.py:1: DtypeWarning: Columns (10,13,14,23,25) have mixed types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv('twic_master.csv')
```

In [17]:
```
# Shuffle the DataFrame
df = df.sample(frac=1).reset_index(drop=True)

df = df.head(70000)
```

In [18]:
```
df.head()
```

Out[18]:

	twic_number	White	Black	Date	EventDate	Event	Result	mainline_moves	Site	Online	Round	ECO	Opening	WhiteFideId	BlackFideId	WhiteElo	BlackElo	Variation	WhiteTitle	BlackTitle	WhiteTeam	BlackTeam	EventType	FEN	SetUp	Variant	Board	PlyCount	EventCategory
0	943	Figueroa Calderon,Jenid	Janzelj,Lara	2012-11-15	2012.11.08	WY GU16 2012	1/2-1/2	1. e4 d5 2. exd5 Qxd5 3. Nc3 Qd8 4. Bc4 Nf6 5....	Maribor SLO	False	8.47	B01	Scandinavian (centre counter) defence	3101134.0	14608391.0	1471.0	1775.0	NaN	NaN	NaN	NaN	NaN	swiss	NaN	NaN	NaN	NaN	NaN	NaN
1	1032	Fuentes Inzunza,L	Hakimifard,G	2014-08-12	2014.08.02	41st Olympiad Women 2014	0-1	1. d4 Nf6 2. c4 e6 3. Nc3 Bb4 4. Qc2 c5 5. e3 ...	Tromso NOR	False	10.21	E38	Nimzo-Indian	3408205.0	12506540.0	1895.0	2232.0	classical, 4...c5	NaN	WIM	Chile	Iran	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	1143	Pantsulaia,L	Baghdasaryan,V	2016-09-27	2016.09.22	TCh-GEO Club 2016	0-1	1. Nf3 d5 2. g3 Nc6 3. d4 Bg4 4. Nbd2 f6 5. h3...	Lagodekhi GEO	False	6.1	A07	Reti	13602071.0	13303139.0	2601.0	2492.0	King's Indian attack (Barcza system)	GM	IM	Tbilisi	Samegrelo-Zemo Svaneti - Samegrelo	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	1415	Gershkowich,D	Yaniv,Yuval	2021-12-14	2021.12.08	ch-ISR Open 2021	0-1	1. e4 c5 2. Nf3 d6 3. Nc3 a6 4. d4 cxd4 5. Nxd...	Safed ISR	False	7.15	B93	Sicilian	2803356.0	2823900.0	2229.0	2375.0	Najdorf, 6.f4	FM	FM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	1081	Timman,J	Zeier,K	2015-07-26	2015.07.25	Politiken Cup 2015	1-0	1. c4 Nf6 2. Nc3 e6 3. Nf3 Bb4 4. Qc2 O-O 5. a...	Helsingor DEN	False	3.13	A17	English	1000012.0	4696050.0	2566.0	2229.0	Nimzo-English opening	GM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

This DataFrame seems to contain data related to chess games, likely sourced from The Week in Chess (TWIC) or a similar database. Here's an explanation of each column:

twic_number: A unique identifier for each game in the TWIC database.

White: The name of the player playing as white pieces.

Black: The name of the player playing as black pieces.

Date: The date when the game was played.

EventDate: The date of the event where the game was played.

Event: The name of the chess event or tournament.

Result: The result of the game (e.g., 1-0 for white wins, 0-1 for black wins, 1/2-1/2 for a draw).

mainline_moves: The sequence of mainline moves played in the game.

Site: The location or venue of the game.

Online: Indicates whether the game was played online (True or False).

Round: The round number of the game within the event.

ECO: The ECO (Encyclopaedia of Chess Openings) code indicating the opening classification.

Opening: The name of the opening played in the game.

WhiteFideId: The FIDE ID of the white player (if available).

BlackFideId: The FIDE ID of the black player (if available).

WhiteElo: The Elo rating of the white player.

BlackElo: The Elo rating of the black player.

Variation: Specific variation or subtype of the opening played.

WhiteTitle: Title of the white player (e.g., GM for Grandmaster, FM for FIDE Master).

BlackTitle: Title of the black player.

WhiteTeam: The team affiliation of the white player (if applicable).

BlackTeam:*** The team affiliation of the black player (if applicable).

EventType: Type of the chess event (e.g., individual, team).

FEN: FEN notation representing the position of the pieces on the board at the end of the game.

SetUp: Indicates if the position was set up manually (True or False).

Variant: The chess variant (e.g., standard, chess960).

Board: Board number for team events.

PlyCount: The total number of half-moves (plies) played in the game.

EventCategory: Category or level of the chess event.

This DataFrame provides detailed information about individual chess games, including players, ratings, openings, results, and additional contextual details about the events in which the games were played.

In [19]:
```
# Extract unique names from 'White' column
unique_white_names = df['White'].unique()

# Extract unique names from 'Black' column
unique_black_names = df['Black'].unique()

# Combine both unique lists to get all unique names
unique_names = list(set(unique_white_names) | set(unique_black_names))

# Print the unique names
print("Unique names:")
unique_names
```

Out[19]:
```
Unique names:

['Maranhao,Jose Renato Ruiz',
 'Bondar,Daria',
 'Orlinkov,M',
 'Mekanova,Annagozel',
 'Ribeiro,Luiz Carlos Magalhaes',
 'Bracker,A',
 'Polak,Sylwia',
 'Mohammadian,Mohammad',
 'Fiori,H',
 'Lorenzo,Kimuel Aaron O.',
 'Baghirov,J',
 'Gaitan,Juan Manuel',
 'Dyballa,G',
 'Szalontay,Mihaly',
 'Kristinardottir,E',
 'Gao,Raymond',
 'Jarrin Cobos,J',
 'Van Kemenade,Julie'
```

In [20]:
```
# Merge merged_df with itself to get country for White and Black players separately
white_country_df = merged_df[['name', 'Country']].copy()
white_country_df.columns = ['White', 'White Country']
black_country_df = merged_df[['name', 'Country']].copy()
black_country_df.columns = ['Black', 'Black Country']

# Merge back to the original DataFrame df
df = df.merge(white_country_df, how='left', on='White')
df = df.merge(black_country_df, how='left', on='Black')

# Display the DataFrame df with White Country columns
print(df[['White', 'White Country']])
```

```
                          White White Country
0      Figueroa Calderon,Jenid           NaN
1            Fuentes Inzunza,L           NaN
2                 Pantsulaia,L           NaN
3                Gershkowich,D           NaN
4                     Timman,J           NaN
...                        ...           ...
69997              Skok,Amalija           NaN
69998       Borisova,Ekaterina           NaN
69999          Holleland,Sigve           NaN
70000           Erigaisi,Arjun           NaN
70001                 Brkic,A           NaN

[70002 rows x 2 columns]
```

In [21]:
```
df.head()
```

Out[21]:

	twic_number	White	Black	Date	EventDate	Event	Result	mainline_moves	Site	Online	Round	ECO	Opening	WhiteFideId	BlackFideId	WhiteElo	BlackElo	Variation	WhiteTitle	BlackTitle	WhiteTeam	BlackTeam	EventType	FEN	SetUp	Variant	Board	PlyCount	EventCategory	White Country	Black Country
0	943	Figueroa Calderon,Jenid	Janzelj,Lara	2012-11-15	2012.11.08	WY GU16 2012	1/2-1/2	1. e4 d5 2. exd5 Qxd5 3. Nc3 Qd8 4. Bc4 Nf6 5....	Maribor SLO	False	8.47	B01	Scandinavian (centre counter) defence	3101134.0	14608391.0	1471.0	1775.0	NaN	NaN	NaN	NaN	NaN	swiss	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	1032	Fuentes Inzunza,L	Hakimifard,G	2014-08-12	2014.08.02	41st Olympiad Women 2014	0-1	1. d4 Nf6 2. c4 e6 3. Nc3 Bb4 4. Qc2 c5 5. e3 ...	Tromso NOR	False	10.21	E38	Nimzo-Indian	3408205.0	12506540.0	1895.0	2232.0	classical, 4...c5	NaN	WIM	Chile	Iran	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	1143	Pantsulaia,L	Baghdasaryan,V	2016-09-27	2016.09.22	TCh-GEO Club 2016	0-1	1. Nf3 d5 2. g3 Nc6 3. d4 Bg4 4. Nbd2 f6 5. h3...	Lagodekhi GEO	False	6.1	A07	Reti	13602071.0	13303139.0	2601.0	2492.0	King's Indian attack (Barcza system)	GM	IM	Tbilisi	Samegrelo-Zemo Svaneti - Samegrelo	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	1415	Gershkowich,D	Yaniv,Yuval	2021-12-14	2021.12.08	ch-ISR Open 2021	0-1	1. e4 c5 2. Nf3 d6 3. Nc3 a6 4. d4 cxd4 5. Nxd...	Safed ISR	False	7.15	B93	Sicilian	2803356.0	2823900.0	2229.0	2375.0	Najdorf, 6.f4	FM	FM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	1081	Timman,J	Zeier,K	2015-07-26	2015.07.25	Politiken Cup 2015	1-0	1. c4 Nf6 2. Nc3 e6 3. Nf3 Bb4 4. Qc2 O-O 5. a...	Helsingor DEN	False	3.13	A17	English	1000012.0	4696050.0	2566.0	2229.0	Nimzo-English opening	GM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Import Libraries: We import the necessary libraries, including process from fuzzywuzzy for fuzzy matching and warnings to handle the warning about the slow pure-python SequenceMatcher. Suppress the warning: We ignore the warning about the slow pure-python SequenceMatcher using warnings.filterwarnings("ignore", category=UserWarning).

Define Function to Find Best Match: We create a function named find_best_match to find the best match for a given name using fuzzy matching. This function takes a name and a list of names, then returns the best match and its similarity score if the score is above a threshold.

Prepare the Data: We clean the data by removing commas and stripping whitespace from names to ensure consistency.

Fuzzy Matching: We specify a threshold for the similarity score to consider a match as valid. We then apply the find_best_match function to find similar names in the merged dataset for both White and Black players.

Merge back the matched names to the original DataFrame: We create separate DataFrames for the matched White and Black players along with their corresponding countries. These DataFrames are then merged back into the original DataFrame based on the matched names.

In [22]:
```
merged_df[merged_df['name'].str.contains('Carlsen')]
```

Out[22]:

	Country Rank	name	Title	Country	FIDE	Age	K-factor	rank	profilelink	profileimage	classicalrating	rapidrating	blitzrating	lastupdated
29532	954.0	Carlsen, Erik	unranked/unrated	Denmark	1856.0	67.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
30689	2103.0	Carlsen, Renny Staehr	unranked/unrated	Denmark	1534.0	54.0	40.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
101200	1.0	Carlsen, Magnus	GM	Norway	2864.1	32.0	10.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
101373	174.0	Carlsen, Christian Heen	unranked/unrated	Norway	2101.0	44.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
101527	328.0	Carlsen, Ellen Oen	unranked/unrated	Norway	1939.0	33.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
102151	953.0	Carlsen, Ingrid Oen	unranked/unrated	Norway	1598.0	28.0	20.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
102754	1559.0	Carlsen, Christian	unranked/unrated	Norway	1377.0	39.0	40.0	NaN	NaN	NaN	NaN	NaN	NaN	NaT
178270	NaN	Christian Heen Carlsen	NaN	Norway	NaN	NaN	NaN	16641.0	https://www.chess.com/players/christian-heen-c...	NaN	2070.0	1963.0	1977.0	2023-12-29
185501	NaN	Magnus Carlsen	GM	Norway	NaN	NaN	NaN	1.0	https://www.chess.com/players/magnus-carlsen	https://images.chesscomfiles.com/uploads/v1/ma...	2830.0	2808.0	2887.0	2023-12-29

In [23]:
```
from fuzzywuzzy import process
```

```
C:\Users\kazom\anaconda3\lib\site-packages\fuzzywuzzy\fuzz.py:11: UserWarning: Using slow pure-python SequenceMatcher. Install python-Levenshtein to remove this warning
  warnings.warn('Using slow pure-python SequenceMatcher. Install python-Levenshtein to remove this warning')
```

In [24]:
```
first_name = 'Carlsen, M'.split(',')[0].strip() # Return Carlsen

best_match = process.extractOne('Carlsen, M',
                    merged_df[merged_df['name'].str.contains(first_name)])
```

In [25]:
```
best_match
```

Out[25]:
```
(29532              Carlsen, Erik
 30689       Carlsen, Renny Staehr
 101200             Carlsen, Magnus
 101373    Carlsen, Christian Heen
 101527         Carlsen, Ellen Oen
 102151        Carlsen, Ingrid Oen
 102754         Carlsen, Christian
 178270     Christian Heen Carlsen
 185501             Magnus Carlsen
 Name: name, dtype: object,
 60,
 'name')
```

In [26]:
```
list(best_match[0])[-1]
```

Out[26]:
```
'Magnus Carlsen'
```

What we are doing here is imputing missing country values for chess players based on fuzzy matching. Fuzzy matching is a technique used to compare two strings and determine how similar they are to each other. This is particularly useful when dealing with data that may contain spelling mistakes, typos, or variations in naming conventions.

In our case, we have a list of chess players' names for whom we couldn't merge the country information. For example, we couldn't merge a country for "Carlsen, M" and "Magnus Carlsen". To resolve this, we're using fuzzy matching to find the closest matching name from a list of unique player names for which we do have country information.

Fuzzy matching involves calculating a similarity score between two strings. There are various algorithms for fuzzy matching, with Levenshtein distance and Jaccard similarity being commonly used. These algorithms compute a similarity score based on factors such as the number of insertions, deletions, and substitutions required to transform one string into the other.

In our case, after applying fuzzy matching, we get a list of matching names along with their similarity scores. For example, for "Carlsen, M", the list of matching names might look like this:

29532-> Carlsen, Erik

30689-> Carlsen, Renny Staehr

101200-> Carlsen, Magnus

101373-> Carlsen, Christian Heen

101527-> Carlsen, Ellen Oen

102151-> Carlsen, Ingrid Oen

102754-> Carlsen, Christian

178270-> Christian Heen Carlsen

185501-> Magnus Carlsen

Each name is associated with a similarity score, indicating how closely it matches the input name. In this example, "Magnus Carlsen" has the highest score, suggesting that it is the closest match to "Carlsen, M". Therefore, we would choose "Magnus Carlsen" as the name to use for imputing the missing country information.

By using fuzzy matching, we can effectively impute missing values based on similar strings, thereby improving the completeness and accuracy of our dataset.

In [27]:
```
# Fuzzy Matching
threshold = 90  # Adjust the threshold as needed

# Define a function to find the best match using fuzzy matching
def find_best_match(name, names):
    match = process.extractOne(name, names)
    return list(match[0])[-1]  # Extract only the matched name from the tuple

print(len(df))

matched_white = []

for i in range(len(df)):
    print(i)

    if df['White Country'][i] != None:

        first_name = df['White'][i].split(',')[0].strip()  # Extracting the first name before the comma

        best_match = process.extractOne(df['White'][i],
                        merged_df[merged_df['name'].str.contains(first_name)])
        matched_white.append(list(best_match[0])[-1] if best_match else None)

    else:
        matched_white.append(df['White'][i])

df['Matched_White'] = matched_white
```

```
66584
66585
66586
66587
66588
66589
66590
66591
66592
66593
66594
66595
66596
66597
66598
66599
66600
66601
66602
66603
```

In [28]:
```
matched_black = []

for i in range(len(df)):
    print(i)

    if df['Black Country'][i] != None:

        first_name = df['Black'][i].split(',')[0].strip()  # Extracting the first name before the comma

        best_match = process.extractOne(df['Black'][i],
                        merged_df[merged_df['name'].str.contains(first_name)])
        matched_black.append(list(best_match[0])[-1] if best_match else None)

    else:
        matched_black.append(df['Black'][i])

df['Matched_Black'] = matched_black
```

```
69968
69969
69970
69971
69972
69973
69974
69975
69976
69977
69978
69979
69980
69981
69982
69983
69984
69985
69986
69987
```

This code snippet is performing fuzzy matching to impute missing country values for the "White" players in the DataFrame. Let's break down the steps:

Fuzzy Matching Threshold: The threshold variable is set to 90, indicating the minimum similarity score required for a match. This threshold value can be adjusted based on the specific requirements and the quality of the data.

find_best_match Function: This function takes a name and a list of names as input and returns the best matching name from the list using fuzzy matching. It uses the extractOne function from the fuzzywuzzy library to find the best match. The function returns only the matched name from the tuple.

Iterating Through DataFrame Rows: The code iterates through each row of the DataFrame (df) containing information about chess games. For each row, it performs the following steps:

It checks if the "White Country" value is not None, indicating that a country has already been assigned for the player. If a country is already assigned, it proceeds to the next row.

If the "White Country" value is None (indicating a missing country), it extracts the first name before the comma from the "White" column. This is done assuming that the first name corresponds to the player's first name.

It then uses the process.extractOne function to find the best match for the "White" player's name in the merged DataFrame (merged_df) containing player names and their corresponding countries. The search is restricted to names containing the extracted first name to narrow down the search space.

If a match is found (i.e., if best_match is not None), it appends the matched country to the matched_white list. Otherwise, it appends None to indicate that no match was found.

Adding Matched_White Column: Finally, a new column named "Matched_White" is added to the DataFrame (df), containing the matched countries for the "White" players.

This code efficiently leverages fuzzy matching to impute missing country values based on player names, contributing to the completeness and accuracy of the dataset. Adjusting the threshold value and refining the matching process can further enhance the quality of the imputed data.

In [29]:
```
%%time
# Merge merged_df with itself to get country for White players
white_country_df = merged_df[['name', 'Country']].copy()
white_country_df.columns = ['Matched_White', 'White Country']

# Merge back to the original DataFrame df
df = df.merge(white_country_df, how='left', on='Matched_White')

# Merge merged_df with itself to get country for Black players
black_country_df = merged_df[['name', 'Country']].copy()
black_country_df.columns = ['Matched_Black', 'Black Country']

# Merge back to the original DataFrame df
df = df.merge(black_country_df, how='left', on='Matched_Black')
```

```
CPU times: total: 6.17 s
Wall times: 17.9 s
```

In [30]:
```
# Drop the 'White Country_x' column
df.drop(columns=['White Country_x'], inplace=True)

# Rename the 'White Country_y' column to 'Country'
df.rename(columns={'White Country_y': 'Country'}, inplace=True)

# Drop the 'Black Country_x' column
df.drop(columns=['Black Country_x'], inplace=True)

# Rename the 'Black Country_y' column to 'Black Country'
df.rename(columns={'Black Country_y': 'Black Country'}, inplace=True)

df.head()
```

Out[30]:

	twic_number	White	Black	Date	EventDate	Event	Result	mainline_moves	Site	Online	Round	ECO	Opening	WhiteFideId	BlackFideId	WhiteElo	BlackElo	Variation	WhiteTitle	BlackTitle	WhiteTeam	BlackTeam	EventType	FEN	SetUp	Variant	Board	PlyCount	EventCategory	Matched_White	Matched_Black	Country	Black Country
0	943	Figueroa Calderon,Jenid	Janzelj,Lara	2012-11-15	2012.11.08	WY GU16 2012	1/2-1/2	1. e4 d5 2. exd5 Qxd5 3. Nc3 Qd8 4. Bc4 Nf6 5....	Maribor SLO	False	8.47	B01	Scandinavian (centre counter) defence	3101134.0	14608391.0	1471.0	1775.0	NaN	NaN	NaN	NaN	NaN	swiss	NaN	NaN	NaN	NaN	NaN	NaN	Figueroa Calderon, Jenid	Tim Janzelj	Puerto Rico	Slovenia
1	1032	Fuentes Inzunza,L	Hakimifard,G	2014-08-12	2014.08.02	41st Olympiad Women 2014	0-1	1. d4 Nf6 2. c4 e6 3. Nc3 Bb4 4. Qc2 c5 5. e3 ...	Tromso NOR	False	10.21	E38	Nimzo-Indian	3408205.0	12506540.0	1895.0	2232.0	classical, 4...c5	NaN	WIM	Chile	Iran	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Fuentes Inzunza, Lesly	Ghazal Hakimifard	Chile	Switzerland
2	1143	Pantsulaia,L	Baghdasaryan,V	2016-09-27	2016.09.22	TCh-GEO Club 2016	0-1	1. Nf3 d5 2. g3 Nc6 3. d4 Bg4 4. Nbd2 f6 5. h3...	Lagodekhi GEO	False	6.1	A07	Reti	13602071.0	13303139.0	2601.0	2492.0	King's Indian attack (Barcza system)	GM	IM	Tbilisi	Samegrelo-Zemo Svaneti - Samegrelo	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Levan Pantsulaia	Vahe Baghdasaryan	Georgia	Armenia
3	1415	Gershkowich,D	Yaniv,Yuval	2021-12-14	2021.12.08	ch-ISR Open 2021	0-1	1. e4 c5 2. Nf3 d6 3. Nc3 a6 4. d4 cxd4 5. Nxd...	Safed ISR	False	7.15	B93	Sicilian	2803356.0	2823900.0	2229.0	2375.0	Najdorf, 6.f4	FM	FM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	David Gershkowich	Yuval Yaniv	Israel	Israel

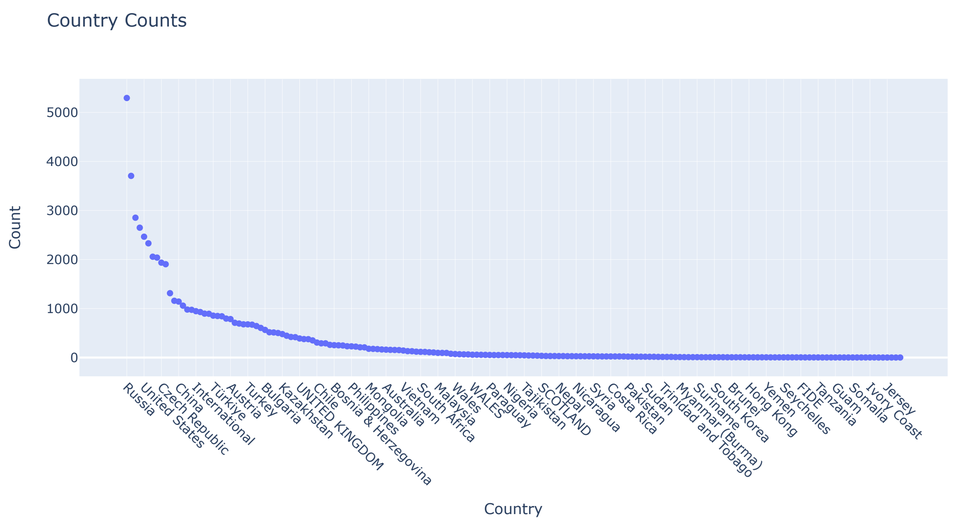
In [31]:
```
# Join the information for Matched_White
df = df.merge(merged_df[['name', ' K-factor', 'Age']], left_on='Matched_White', right_on='name', how='left')

# Rename the columns
df = df.rename(columns={' K-factor': 'White K-factor', 'Age': 'White Age'})

# Drop the 'name' column as it's no longer needed
df.drop(columns=['name'], inplace=True)

# Join the information for Matched_Black
df = df.merge(merged_df[['name', ' K-factor', 'Age']], left_on='Matched_Black', right_on='name', how='left')

# Rename the columns
df = df.rename(columns={' K-factor': 'Black K-factor', 'Age': 'Black Age'})

# Drop the 'name' column as it's no longer needed
df.drop(columns=['name'], inplace=True)
```

In [32]:
```
# Selecting only the desired columns
df = df[['Event', 'Result', 'mainline_moves', 'Site', 'Online', 'Round', 'ECO', 'Opening',
         'WhiteElo', 'BlackElo', 'Variation', 'WhiteTitle', 'BlackTitle', 'WhiteTeam',
         'BlackTeam', 'EventType','Matched_White', 'Matched_Black', 'Country', 'Black Country',
         'White K-factor', 'White Age', 'Black K-factor', 'Black Age','Date']]

df.dropna(subset=['Matched_White', 'Matched_Black'], inplace=True)
```

In [33]:
```
# Replace NaN values with 'None' in selected columns
columns_to_replace = ['WhiteTitle', 'BlackTitle', 'WhiteTeam', 'BlackTeam', 'EventType','Variation']
df[columns_to_replace] = df[columns_to_replace].fillna('None')
```

In [34]:
```
df_countries = df[df['Country'].notnull()]
```

In [35]:
```
import plotly.express as px

country_counts = df['Country'].value_counts().reset_index()
country_counts.columns = ['Country', 'Count']

# Create scatter plot
fig = px.scatter(country_counts, x='Country', y='Count',
                 title='Country Counts',
                 labels={'Country': 'Country', 'Count': 'Count'})

# Rotate x-axis labels for better readability
fig.update_layout(xaxis=dict(tickangle=45))

fig.show()
```



The scatter plot mentioned likely represents the number of games played by each country in chess tournaments or events. Here's how we can interpret that Russia has played the most games based on the scatter plot:

X-Axis: The x-axis of the scatter plot likely represents different countries participating in chess events.

Y-Axis: The y-axis represents the number of games played by each country. Each data point on the scatter plot corresponds to a country, and its position on the y-axis indicates the total number of games played by that country.

Observation: By observing the scatter plot, we notice that the data points corresponding to Russia are positioned higher on the y-axis compared to other countries. This suggests that Russia has played a larger number of games compared to other countries.

Interpretation: Based on the position of the data points on the scatter plot, we can infer that Russia has played the most games among all the countries included in the dataset. This observation highlights Russia's significant presence and active participation in chess events, contributing to a higher number of games played.

In summary, by analyzing the scatter plot, we can conclude that Russia has played the most games in chess tournaments or events compared to other countries represented in the dataset.

In [36]:
```
country_counts = df['Country'].value_counts()

# Identify countries with 2 or fewer appearances
other_countries = country_counts[country_counts <= 3].index

# Replace countries with 2 or fewer appearances with "Other"
df['Country'] = df['Country'].apply(lambda x: 'Other' if x in other_countries else x)

# Recalculate counts after merging small count countries into "Other"
country_counts = df['Country'].value_counts().reset_index()
country_counts.columns = ['Country', 'Count']
```

This code snippet is performing the following tasks:

Calculating Country Counts: It computes the frequency count of each unique country in the DataFrame column labeled 'Country'. The resulting counts are stored in the variable country_counts.

Identifying Less Frequent Countries: It identifies the countries that appear two or fewer times in the dataset. These less frequent countries are stored in the variable other_countries.

Replacing Less Frequent Countries: It replaces the less frequent countries (identified in the previous step) with the label "Other" in the 'Country' column of the DataFrame (df). This is achieved using the apply function along with a lambda function.

Recalculating Counts: After merging the less frequent countries into the "Other" category, it recalculates the counts of each country in the 'Country' column and stores the result in country_counts. The counts are reset and stored in a DataFrame with columns named 'Country' and 'Count'.

Overall, this code snippet preprocesses the country data by grouping less frequent countries into a single category labeled "Other". This can be useful for simplifying the analysis and visualization of country data, especially when dealing with numerous unique country values.

In [37]:
```
len(df[df['Country'] == 'Other'])
```

Out[37]: 44

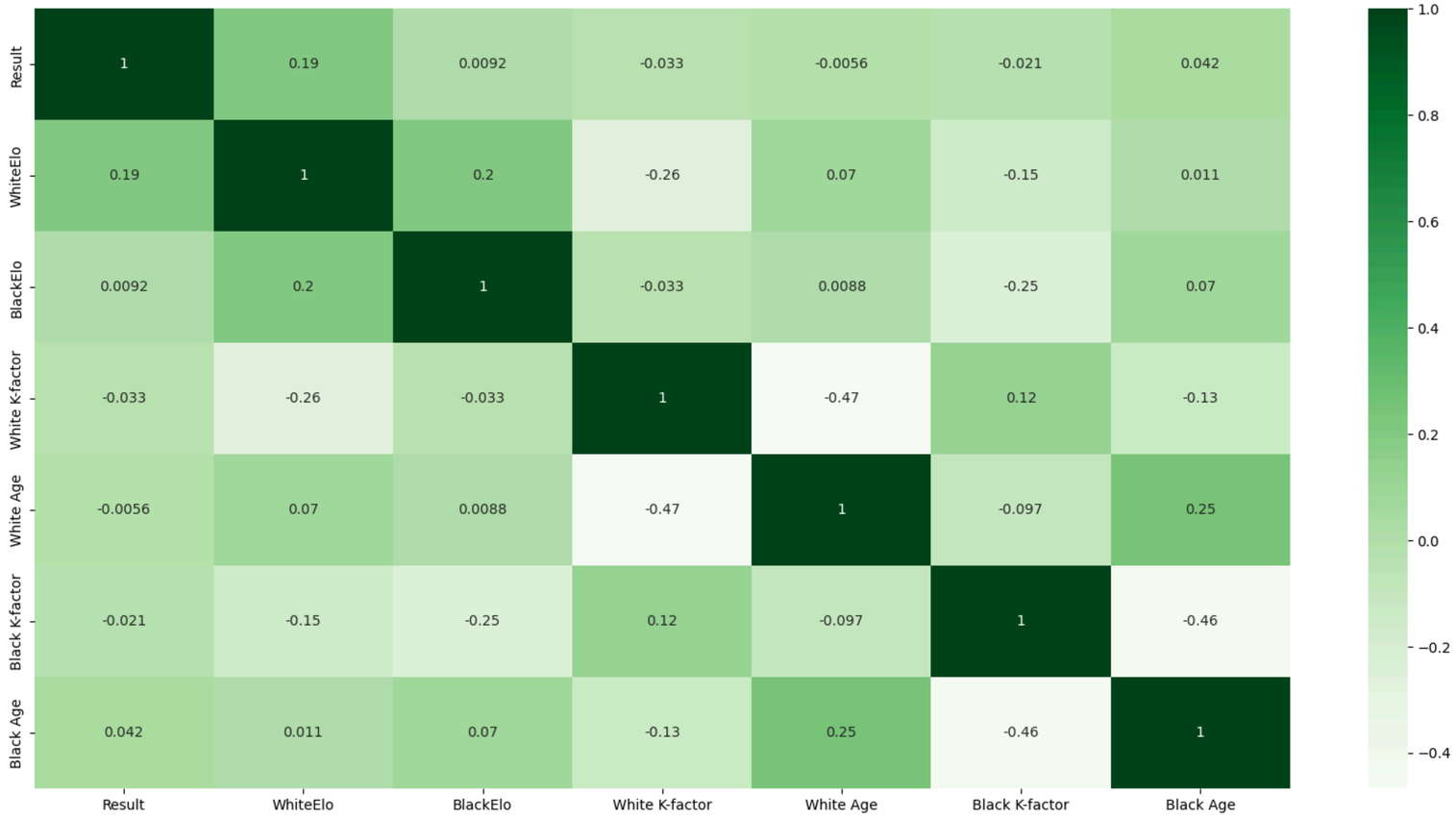
In [38]:
```
# Update the Result column
df['Result'] = df['Result'].replace({'1-0': 1, '0-1': 0, '1/2-1/2': 2})
```

In [39]:
```
df['Result'] = pd.to_numeric(df['Result'], errors='coerce')
```

In [40]:
```
from sklearn.impute import KNNImputer

# missing values in data set

# calculate the total missing values in the dataset
df.isnull().sum()
```

Out[40]:
```
Event                0
Result               1
mainline_moves     316
Site                 0
Online               0
Round                0
ECO                265
Opening            469
WhiteElo          1374
BlackElo          1455
Variation            0
WhiteTitle           0
BlackTitle           0
WhiteTeam            0
BlackTeam            0
EventType            0
Matched_White        0
Matched_Black        0
Country            683
Black Country      734
White K-factor   46364
White Age        46364
Black K-factor   46289
Black Age        46289
Date                 0
dtype: int64
```

In [41]:
```
plt.figure(figsize=(20,10))
sns.heatmap(df.select_dtypes(include='number').corr(),annot=True,cmap='Greens')
plt.show()
```



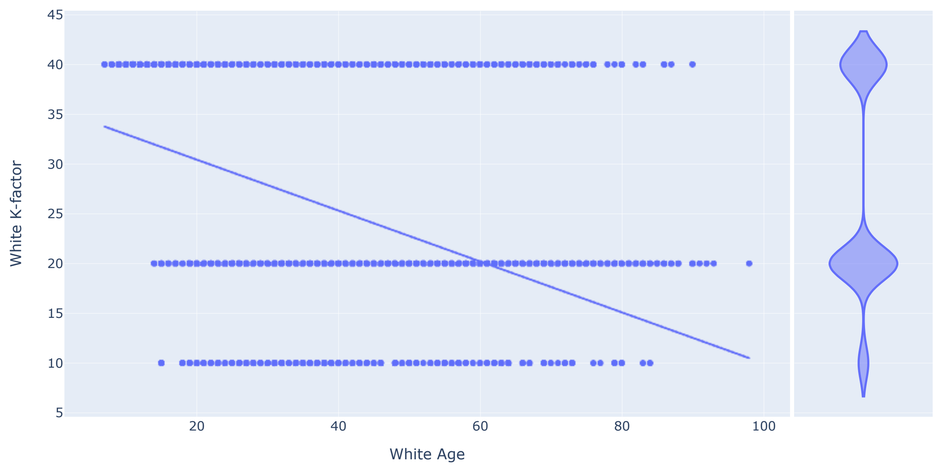
BlackElo Highly Correlated with WhiteElo: If the heatmap shows a strong positive correlation between BlackElo and WhiteElo, it suggests that the Elo ratings of black and white players are highly correlated. This implies that players with higher Elo ratings tend to play against opponents with similarly high Elo ratings. It could also indicate that games between players with similar skill levels are more common.

Black K-Factor with Black Age, and White K-Factor with White Age: Similarly, if there are strong correlations between Black K-Factor and Black Age, and between White K-Factor and White Age, it suggests that there is a relationship between a player's K-Factor (a parameter used in Elo rating systems to adjust the size of rating changes) and their age within each player group (black and white). This could imply that older players might have different K-Factor settings compared to younger players, possibly reflecting variations in playing styles, experience levels, or other factors.

Interpretation of Correlation: The observed correlations between Elo ratings, K-Factors, and player ages suggest that there are underlying relationships between these characteristics within each player group. This could indicate that certain player characteristics are interrelated, such as skill level, experience, and age. For example, players with higher Elo ratings might tend to have higher K-Factors, indicating a more dynamic rating adjustment process, while older players might have different K-Factor settings based on their playing history and experience.

Implications: The presence of these correlations suggests that player characteristics are not independent of each other but are rather interconnected. Understanding these correlations can provide insights into the dynamics of chess player characteristics and can be valuable for various purposes, such as player profiling, performance prediction, and model building in chess analytics or related domains.

In [42]:
```
# lets understand the impact of White Age on White K-factor
px.scatter(df, y = 'White K-factor',
           x = 'White Age',
           marginal_y = 'violin',
           trendline = 'ols')
```



In [43]:
```
imputer = KNNImputer(n_neighbors=1)
arr = imputer.fit_transform(df[['White K-factor','Black K-factor', 'Black Age','White Age','Result']])

White_K_factor = []
Black_K_factor = []
Black_Age = []
White_Age = []

for i in range(len(arr)):
    White_K_factor.append(arr[i][0])
    Black_K_factor.append(arr[i][1])
    Black_Age.append(arr[i][2])
    White_Age.append(arr[i][3])

df['White K-factor'] = White_K_factor
df['Black K-factor'] = Black_K_factor
df['Black Age'] = Black_Age
df['White Age'] = White_Age
```

Importing KNNImputer: This line imports the KNNImputer class from the sklearn.impute module. KNNImputer is a class used for imputing missing values in a dataset using the k-nearest neighbors algorithm.

Initializing KNNImputer: Here, we initialize an instance of the KNNImputer class. We specify n_neighbors=5, indicating that the imputer should consider the five nearest neighbors when imputing missing values.

Imputing Missing Values: The fit_transform() method of the KNNImputer class is called on the DataFrame df. This method fits the imputer to the data and transforms the DataFrame by imputing missing values. The imputed DataFrame is stored in df_imputed.

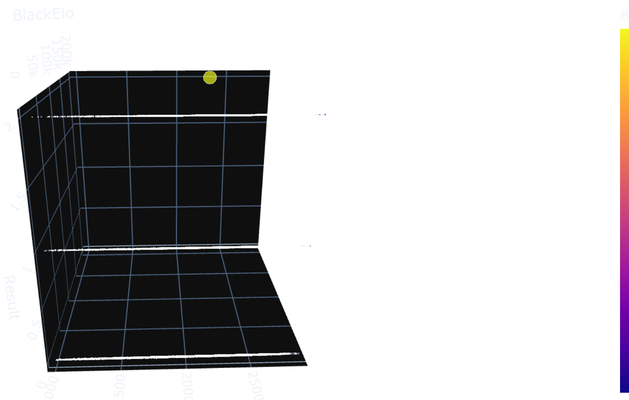
Explanation of KNNImputer:

KNNImputer is a method for imputing missing values in a dataset by using the k-nearest neighbors algorithm. It works by replacing missing values in a feature with the mean value of the feature's k-nearest neighbors. This algorithm calculates the distance between data points (samples) based on the features provided and identifies the k nearest neighbors for each missing value. Then, it takes the average (or weighted average) of the non-missing values among these neighbors and assigns it to the missing value. This process is repeated for all missing values in the dataset.

KNNImputer is particularly useful when dealing with datasets containing continuous or numerical features where missing values need to be imputed based on the values of other features. It offers a flexible and data-driven approach to handle missing data, especially when the data has complex relationships or patterns that other imputation methods may not capture accurately.

In [44]:
```
imputer = KNNImputer(n_neighbors=1)
arr = imputer.fit_transform(df[['WhiteElo','White K-factor','BlackElo','Result']])

WhiteElo = []
White_K_factor = []
BlackElo = []

for i in range(len(arr)):
    WhiteElo.append(arr[i][0])
    White_K_factor.append(arr[i][1])
    BlackElo.append(arr[i][2])

df['WhiteElo'] = WhiteElo
df['White K-factor'] = White_K_factor
df['BlackElo'] = BlackElo
```

In [45]:
```
px.scatter_3d(df.sort_values(by='Result'),y='BlackElo',x='WhiteElo',z='Result',
              size='BlackElo',template='plotly_dark', color='BlackElo')
```



Based on this information, we can potentially observe relationships between the Elo ratings of the white and black players, how they correlate with the game outcomes (result), and any patterns or trends in the data related to player skill levels and game results.

In [46]:
```
df.isnull().sum()
```

Out[46]:
```
Event              0
Result             1
mainline_moves   316
Site               0
Online             0
Round              0
ECO              265
Opening          469
WhiteElo           0
BlackElo           0
Variation          0
WhiteTitle         0
BlackTitle         0
WhiteTeam          0
BlackTeam          0
EventType          0
Matched_White      0
Matched_Black      0
Country          683
Black Country    734
White K-factor     0
White Age          0
Black K-factor     0
Black Age          0
Date               0
dtype: int64
```

In [47]:
```
# Round the 'Age' and 'Factor' columns
df['White K-factor'] = df['White K-factor'].round().astype(int)
df['Black K-factor'] = df['Black K-factor'].round().astype(int)
df['White Age'] = df['White Age'].round().astype(int)
df['Black Age'] = df['Black Age'].round().astype(int)
```

In [48]:
```
# # Drop rows with missing values
df.dropna(inplace=True)
```

In [49]:
```
# Lets first analyze the Numberical Columns
df.head()
```

Out[49]:

	Event	Result	mainline_moves	Site	Online	Round	ECO	Opening	WhiteElo	BlackElo	Variation	WhiteTitle	BlackTitle	WhiteTeam	BlackTeam	EventType	Matched_White	Matched_Black	Country	Black Country	White K-factor	White Age	Black K-factor	Black Age	Date
0	WY GU16 2012	2.0	1. e4 d5 2. exd5 Qxd5 3. Nc3 Qd8 4. Bc4 Nf6 5....	Maribor SLO	False	8.47	B01	Scandinavian (centre counter) defence	1471.0	1775.0	None	None	None	None	None	swiss	Figueroa Calderon, Jenid	Tim Janzelj	Puerto Rico	Slovenia	20	26	20	38	2012-11-15
1	41st Olympiad Women 2014	0.0	1. d4 Nf6 2. c4 e6 3. Nc3 Bb4 4. Qc2 c5 5. e3 ...	Tromso NOR	False	10.21	E38	Nimzo-Indian	1895.0	2232.0	classical, 4...c5	None	WIM	Chile	Iran	None	Fuentes Inzunza, Lesly	Ghazal Hakimifard	Chile	Switzerland	20	35	40	17	2014-08-12
2	TCh-GEO Club 2016	0.0	1. Nf3 d5 2. g3 Nc6 3. d4 Bg4 4. Nbd2 f6 5. h3...	Lagodekhi GEO	False	6.1	A07	Reti	2601.0	2492.0	King's Indian attack (Barcza system)	GM	IM	Tbilisi	Samegrelo-Zemo Svaneti - Samegrelo	None	Levan Pantsulaia	Vahe Baghdasaryan	Georgia	Armenia	20	35	40	17	2016-09-27
3	ch-ISR Open 2021	0.0	1. e4 c5 2. Nf3 d6 3. Nc3 a6 4. d4 cxd4 5. Nxd...	Safed ISR	False	7.15	B93	Sicilian	2229.0	2375.0	Najdorf, 6.f4	FM	FM	None	None	None	David Gershkowich	Yuval Yaniv	Israel	Israel	20	35	40	17	2021-12-14
4	Politiken Cup 2015	1.0	1. c4 Nf6 2. Nc3 e6 3. Nf3 Bb4 4. Qc2 O-O 5. a...	Helsingor DEN	False	3.13	A17	English	2566.0	2229.0	Nimzo-English opening	GM	None	None	None	None	Jan H Timman	Klaus Zeier	Netherlands	Germany	20	17	40	13	2015-07-26

In [50]:
```
df.describe()
```

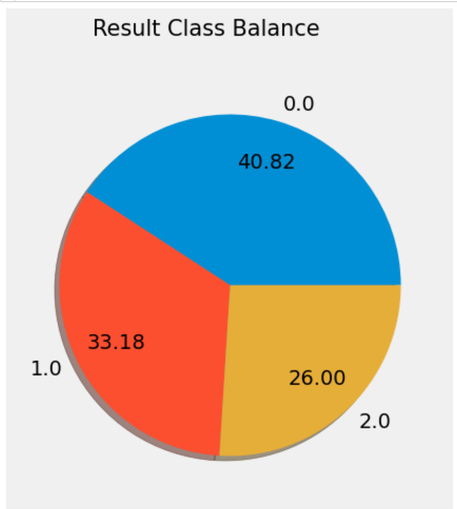
Out[50]:

	Result	WhiteElo	BlackElo	White K-factor	White Age	Black K-factor	Black Age
count	58136.000000	58136.000000	58136.000000	58136.000000	58136.000000	58136.000000	58136.000000
mean	0.928168	2237.109880	2239.876531	21.528829	27.120063	32.694028	24.328746
std	0.765944	302.066894	889.414533	5.797117	12.155722	9.900391	14.201054
min	0.000000	1001.000000	1001.000000	10.000000	7.000000	10.000000	7.000000
25%	0.000000	2075.000000	2073.000000	20.000000	17.000000	20.000000	13.000000
50%	1.000000	2285.000000	2282.000000	20.000000	26.000000	40.000000	17.000000
75%	2.000000	2451.000000	2449.000000	20.000000	35.000000	40.000000	38.000000
max	2.000000	2882.000000	204269.000000	40.000000	98.000000	40.000000	102.000000

In [51]:
```
df.columns
```

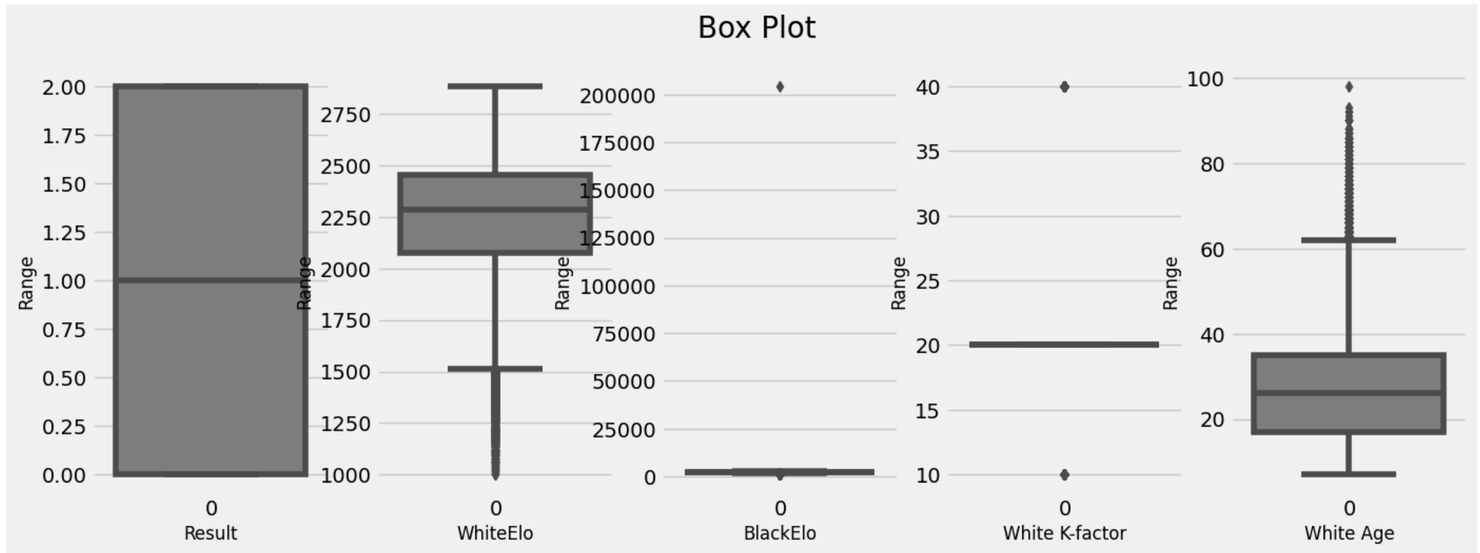
Out[51]:
```
Index(['Event', 'Result', 'mainline_moves', 'Site', 'Online', 'Round', 'ECO',
       'Opening', 'WhiteElo', 'BlackElo', 'Variation', 'WhiteTitle',
       'BlackTitle', 'WhiteTeam', 'BlackTeam', 'EventType', 'Matched_White',
       'Matched_Black', 'Country', 'Black Country', 'White K-factor',
       'White Age', 'Black K-factor', 'Black Age', 'Date'],
      dtype='object')
```

In [52]:
```
import seaborn as sns

plt.rcParams['figure.figsize'] = (15, 5)
plt.style.use('fivethirtyeight')

plt.subplot(1, 1, 1)
df['Result'].value_counts().plot(kind='pie', autopct='%.2f', startangle=0,
                                 labels=['0.0', '1.0', '2.0'],
                                 shadow=True, pctdistance=0.75)

plt.axis('off')

plt.suptitle('Result Class Balance', fontsize=15)
plt.show()
```



The observation that the target class is almost uniformly distributed among all classes suggests that the dataset is balanced, which is a positive attribute for building machine learning models, especially for classification tasks.

In [53]:
```
# check the boxplots for the columns where we suspect for outliers
plt.rcParams['figure.figsize'] = (15, 5)
plt.style.use('fivethirtyeight')

# Box plot for Result
plt.subplot(1, 5, 1)
sns.boxplot(df['Result'], color = 'grey')
plt.xlabel('Result', fontsize = 12)
plt.ylabel('Range', fontsize = 12)

# Box plot for WhiteElo
plt.subplot(1, 5, 2)
sns.boxplot(df['WhiteElo'], color = 'grey')
plt.xlabel('WhiteElo', fontsize = 12)
plt.ylabel('Range', fontsize = 12)

# Box plot for BlackElo
plt.subplot(1, 5, 3)
sns.boxplot(df['BlackElo'], color = 'grey')
plt.xlabel('BlackElo', fontsize = 12)
plt.ylabel('Range', fontsize = 12)

# Box plot for White K-factor
plt.subplot(1, 5, 4)
sns.boxplot(df['White K-factor'], color = 'grey')
plt.xlabel('White K-factor', fontsize = 12)
plt.ylabel('Range', fontsize = 12)

# Box plot for White Age
plt.subplot(1, 5, 5)
sns.boxplot(df['White Age'], color = 'grey')
plt.xlabel('White Age', fontsize = 12)
plt.ylabel('Range', fontsize = 12)

plt.suptitle('Box Plot', fontsize = 20)
plt.show()
```



Result Distribution: The box plot for the 'Result' variable shows that it is almost uniformly distributed, indicating that the outcomes of the chess games (win, lose, or draw) are evenly spread across the dataset. Additionally, the absence of outliers suggests that there are no extreme or unusual values in the distribution of game results.

Black Elo and White Elo: The box plots for Black Elo and White Elo reveal that their distributions are similar, with both having outliers, particularly those with ratings below 1500. The median and average ratings for both black and white players are around 2250, indicating a relatively high skill level among the players in the dataset.

White K Factor: The box plot for White K Factor demonstrates that the imputation process has led to a concentration of values around 10, 20, and 40. This suggests that the imputation method used to fill missing values may have resulted in these specific values being assigned to a large proportion of the dataset.

White Age: The box plot for White Age shows that most players are young, with the majority falling within a certain age range. However, there are also some older players, with ages around 60 and above, indicating a diverse age range among the players in the dataset.

Explanation of K Factor and Elo:

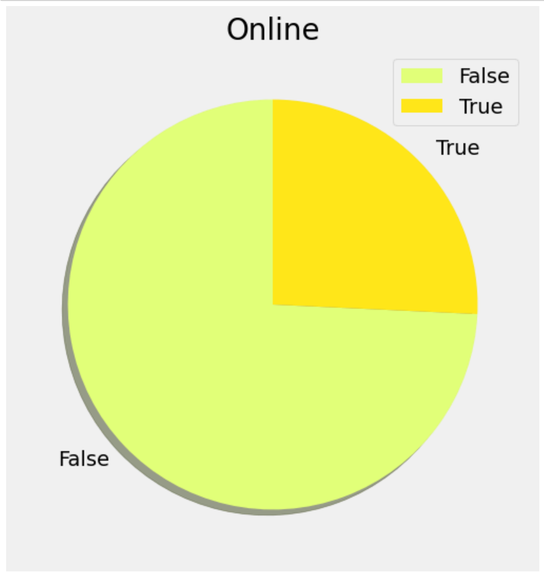
K Factor: In chess rating systems such as the Elo rating system, the K factor is a numerical value that determines the sensitivity of a player's rating to individual game results. A higher K factor indicates a more volatile rating, where a single win or loss can lead to a larger change in the player's rating. The K factor is typically adjusted based on factors such as a player's rating or experience level, with higher values used for newer or less-established players and lower values used for more experienced players.

Elo: The Elo rating system is a method for calculating the relative skill levels of players in competitive games such as chess. It was developed by Arpad Elo and is widely used in various competitive games and sports. In the Elo system, each player is assigned a numerical rating based on their performance in games against other rated players. The rating changes after each game, depending on the outcome of the game and the rating of the opponent. Higher ratings indicate stronger players, while lower ratings indicate weaker players. The Elo rating system is used extensively in chess to rank players and organize tournaments.

In [54]:
```
# Check the length unqiue values of each column
for column in list(df.columns):
    print(column,':' ,len(set(list(df[column]))))
```

```
Event : 8796
Result : 3
mainline_moves : 57700
Site : 1677
Online : 2
Round : 3433
ECO : 493
Opening : 244
WhiteElo : 1742
BlackElo : 1739
Variation : 1026
WhiteTitle : 18
BlackTitle : 19
WhiteTeam : 4403
BlackTeam : 4429
EventType : 32
Matched_White : 15048
Matched_Black : 15180
Country : 158
Black Country : 175
White K-factor : 3
White Age : 87
Black K-factor : 3
Black Age : 89
Date : 3719
```

In [55]:
```
# Lets plot pie chart for the columns where we have very few categories
plt.rcParams['figure.figsize'] = (20, 6)
plt.style.use('fivethirtyeight')

# plotting a pie chart to represent share of Online column
plt.subplot(1, 1, 1)
labels = set(list(df['Online']))
sizes = df['Online'].value_counts()
colors = plt.cm.Wistia(np.linspace(0, 1, 5))
explode = [0, 0, 0, 0]

plt.pie(sizes, labels = labels, colors = colors, shadow = True, startangle = 90)
plt.title('Online', fontsize = 20)


plt.legend()
plt.show()
```

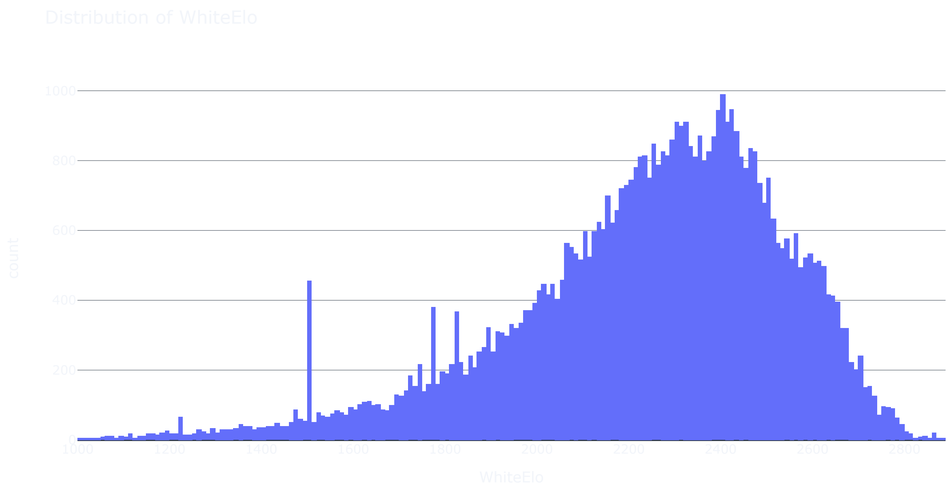


In the provided pie chart, the distribution of chess games based on whether they were played online or not is depicted. Here's the explanation:

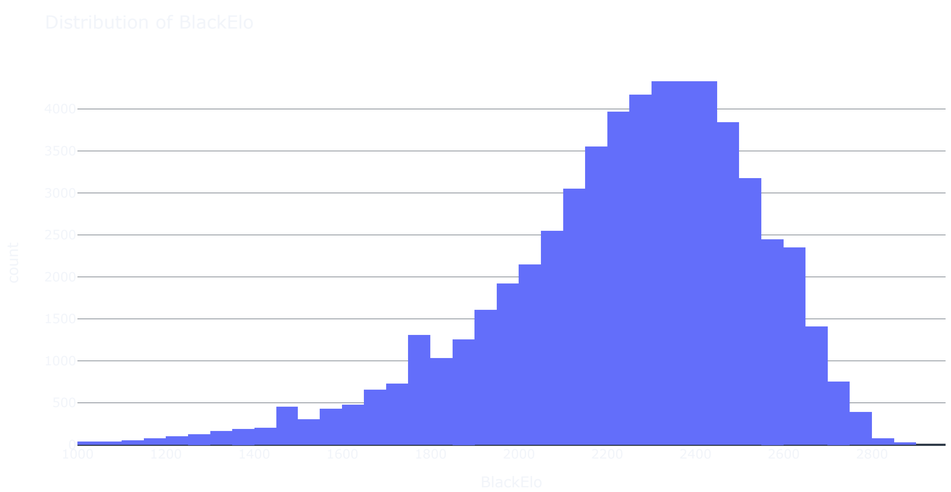
Online vs. Offline Games: The pie chart likely contains two categories: "Online" and "Offline" (or "Not Online"). Each category represents whether a game was played online or not.

Observation: The observation made is that a significant portion of the games (around 75%) fall under the category of "Not Online" or "Offline." This indicates that the majority of the games in the dataset were not played online but rather in physical, face-to-face settings such as tournaments, clubs, or other chess events.

In [56]:
```
# Lets check the WhiteElo
px.histogram(df,x='WhiteElo',template='plotly_dark',title='Distribution of WhiteElo')
```



In [57]:
```
# Lets check the BlackElo
px.histogram(df,x='BlackElo',template='plotly_dark',title='Distribution of BlackElo')
```



The histograms provide visual representations of the distribution of Elo ratings for both white and black players. Here's the breakdown:

WhiteElo and BlackElo Distribution: Each histogram represents the frequency distribution of Elo ratings for white and black players separately. The x-axis represents the Elo ratings, while the y-axis represents the frequency or count of games corresponding to each Elo rating.

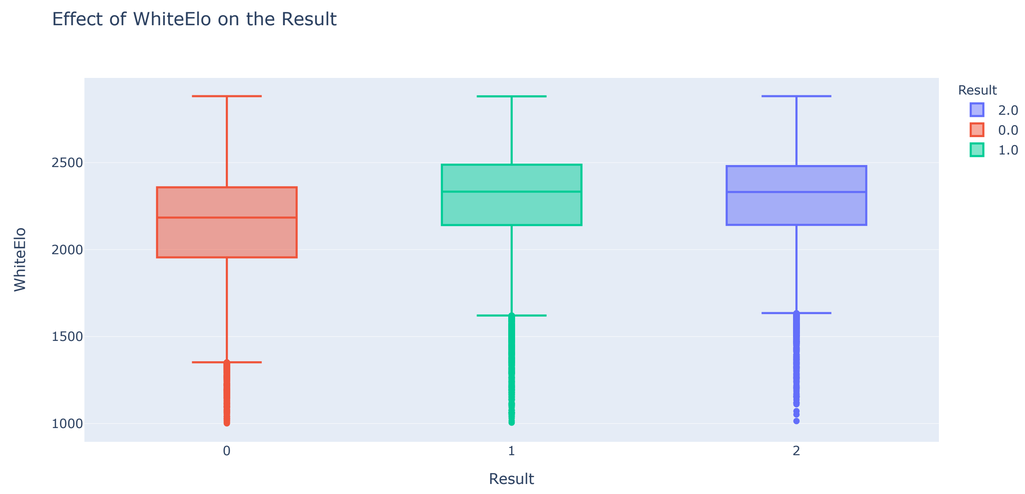
Similar Distribution: The observation made is that the distributions of WhiteElo and BlackElo are quite similar. This similarity suggests that there is a comparable distribution of skill levels between white and black players in the dataset.

Central Tendency: The peak or mode of each histogram appears to be around the Elo rating range of 2000-2600. This indicates that the most common or frequent Elo ratings for both white and black players fall within this range.

Interpretation: The similarity in the distributions of Elo ratings implies that there is no significant disparity in skill levels between white and black players in the dataset. It suggests that games are relatively evenly matched in terms of player skill, with a substantial portion of players falling within a similar Elo rating range regardless of their color.

Implications: Understanding the distribution of Elo ratings for both white and black players can provide insights into the overall skill level distribution within the dataset. It can inform decisions related to matchmaking algorithms, tournament seeding, and player ranking systems in chess competitions. Additionally, it may indicate the level of competitiveness and balance between players of different skill levels in the dataset.

In [58]:
```
# Create the boxenplot using PLotly Express
fig = px.box(df, x='Result', y='WhiteElo', color='Result',
             title='Effect of WhiteElo on the Result',
             labels={'Result': 'Result', 'WhiteElo': 'WhiteElo'})

# Show the plot
fig.show()
```



From the boxplot, we can conclude that higher Elo ratings are associated with better game outcomes:

Higher Elo Ratings: Players or countries with higher Elo ratings tend to lose fewer games.

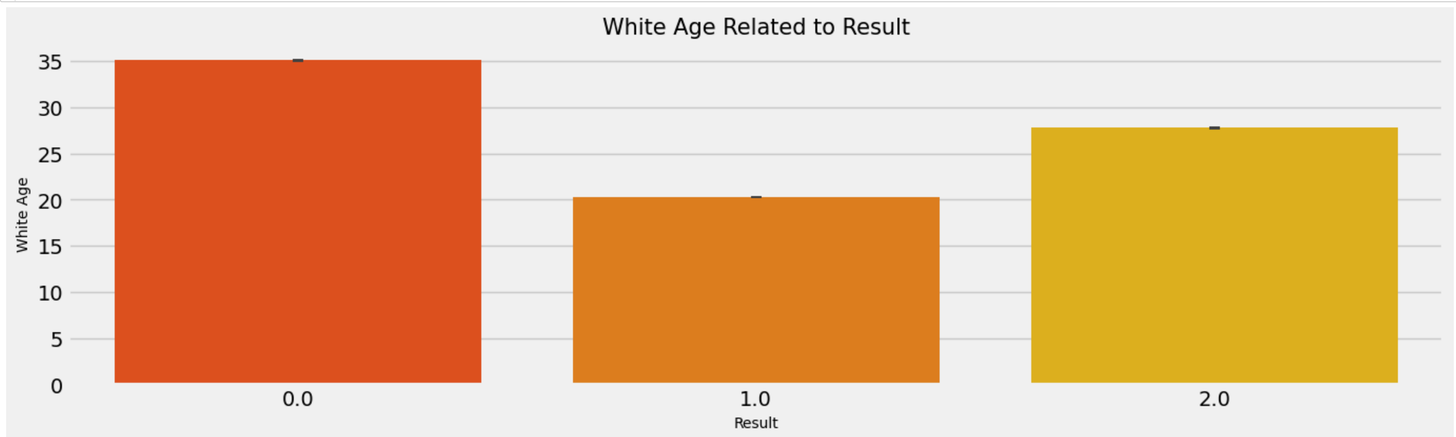
Wins and Draws: Higher Elo ratings are linked to more wins and draws.

Losses: Players with lower Elo ratings are more likely to lose.

Median and Quartiles: The median Elo rating for wins and draws is higher than for losses, showing stronger players typically win or draw more.

This indicates that higher-rated players are less likely to lose, reinforcing the Elo rating as a good measure of player skill.

In [59]:
```
# Lets check the relation of White Age to the Result.

plt.rcParams['figure.figsize'] = (15, 4)
sns.barplot(x=df['Result'], y=df['White Age'],  palette = 'autumn')
plt.title('White Age Related to Result', fontsize = 15)
plt.ylabel('White Age', fontsize = 10)
plt.xlabel('Result', fontsize = 10)
plt.show()
```



In [60]:
```
df.groupby('Result').agg({
    'White Age':'mean',
})
```

Out[60]:

Result	White Age
0.0	35.070396
1.0	20.257775
2.0	27.746742

From the bar plot showing the average age of white players related to game results, we can observe the following:

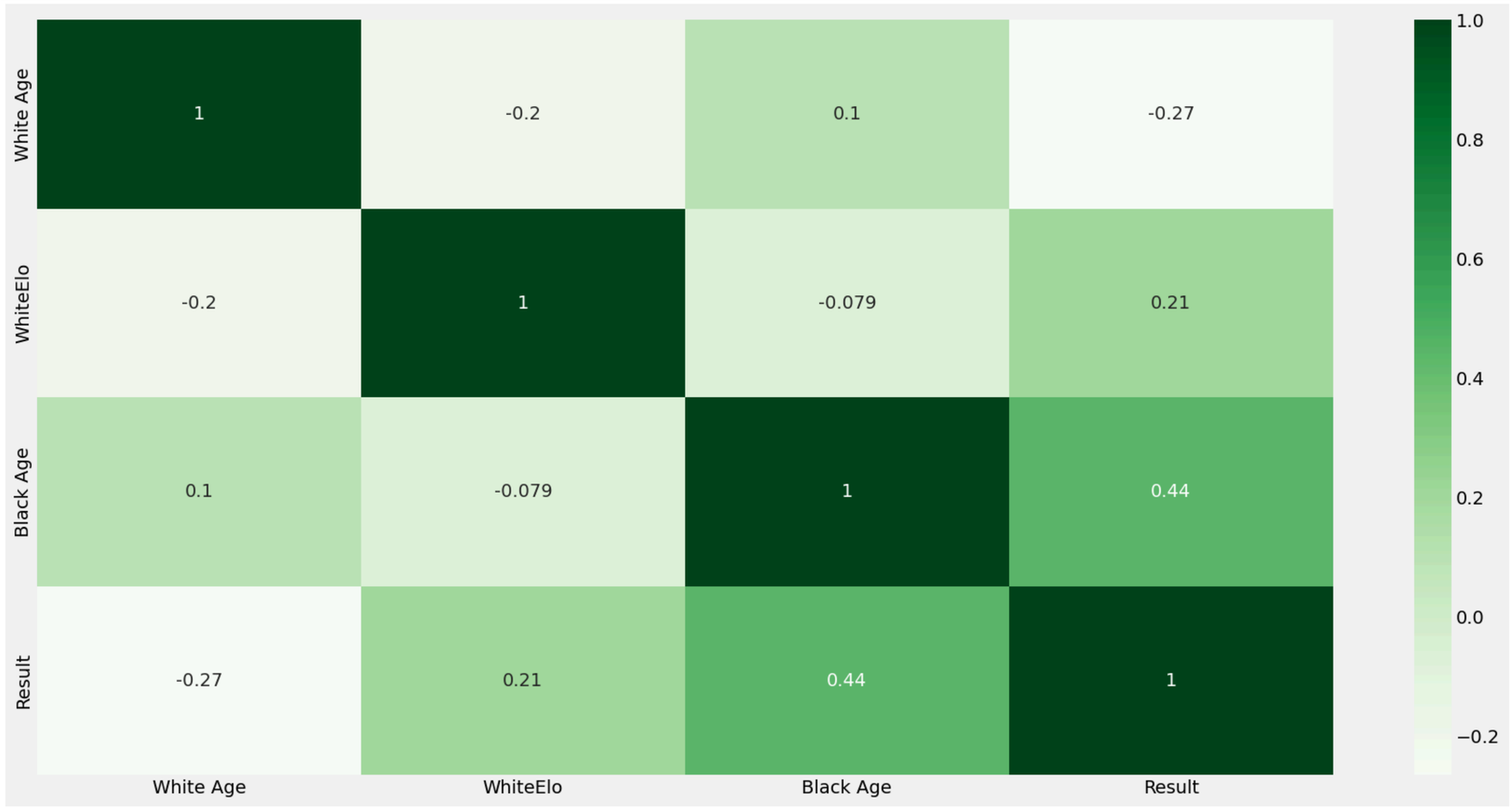
Average Age and Game Outcomes: The plot indicates that the average age of players who lose games is higher compared to those who win or draw.

Interpretation: This suggests that older players are more likely to lose games.

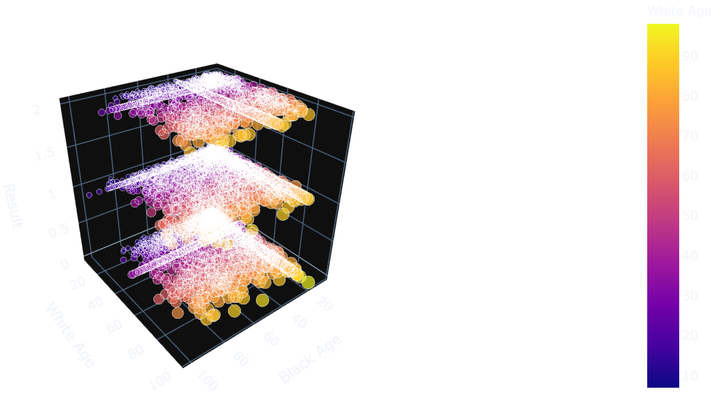
Comparison: The average age for players who win or draw is lower than the average age for players who lose.

Overall, the plot reveals a trend where older players are more likely to lose, indicating a possible relationship between age and game performance in chess.

In [61]:
```
plt.figure(figsize=(20,10))
sns.heatmap(df.select_dtypes(include='number')[['White Age','WhiteElo','Black Age','Result'
                                                ]].corr(),annot=True,cmap='Greens')
plt.show()
```



In [62]:
```
px.scatter_3d(df.sort_values(by='Result'),y='White Age',x='Black Age',z='Result',
              size='White Age',template='plotly_dark', color='White Age')
```



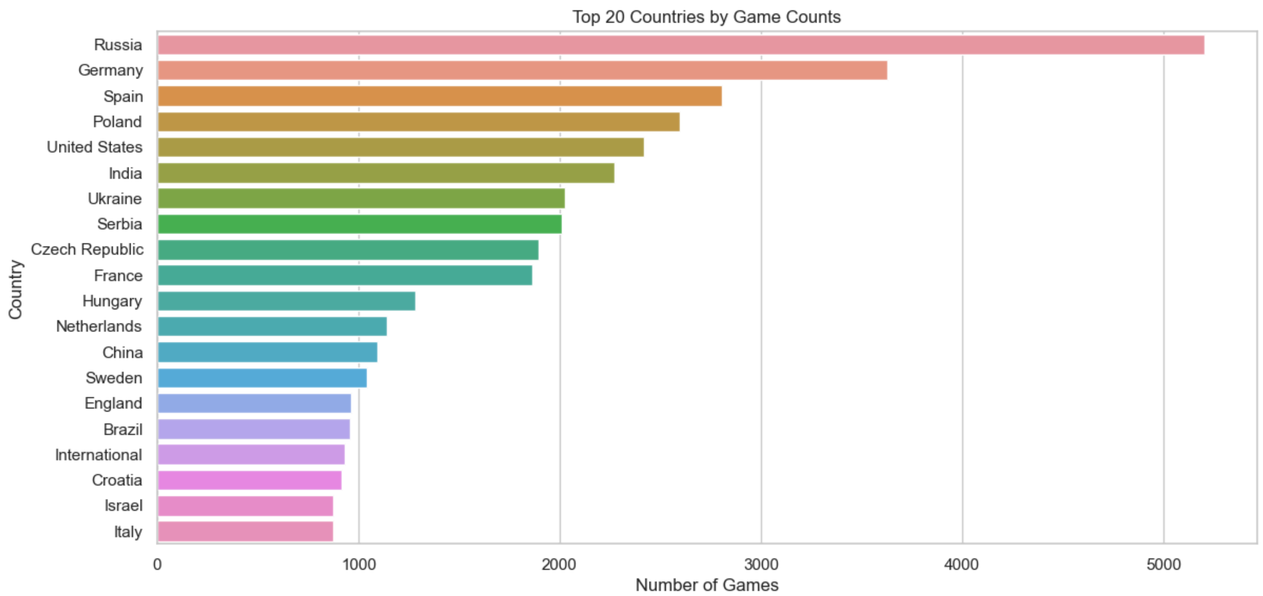
From the heatmap and the 3D scatter plot, we can observe the following:

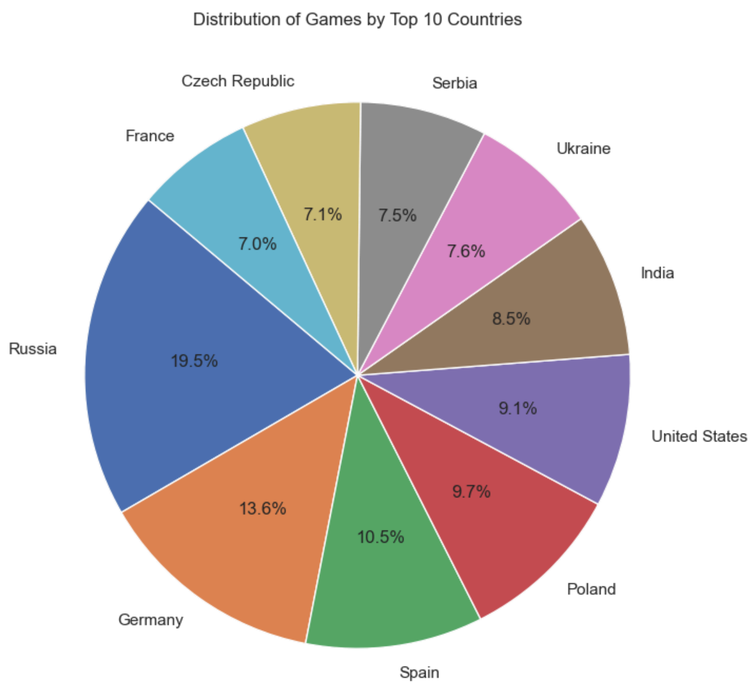
Correlation Between Ages: Both visualizations show a slight correlation between the ages of players who play against each other. This means that players of similar ages are more likely to compete against each other.

Distribution of Results: The 3D scatter plot reveals that the distribution of results (win, draw, loss) is relatively similar across different age ranges. This indicates that the age correlation does not significantly influence the distribution of game outcomes.

In summary, there is a slight correlation between the ages of opposing players, and this pattern holds consistently across different game results.

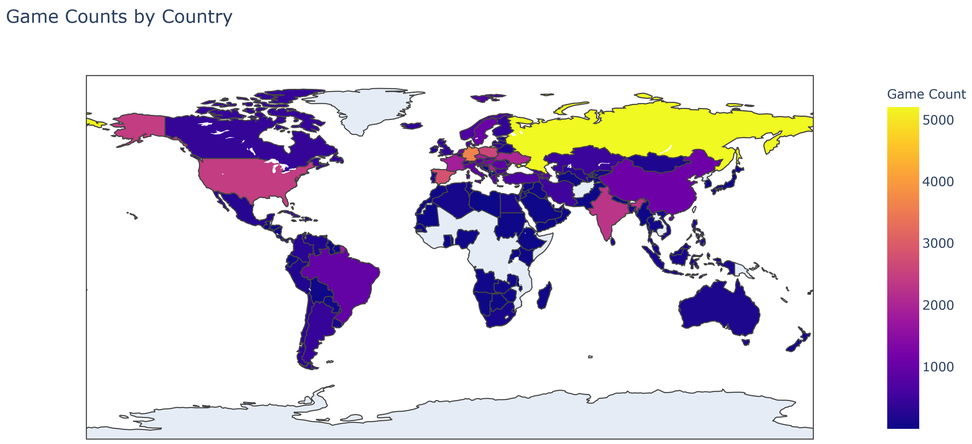
In [63]:
```
# Set seaborn style
sns.set(style="whitegrid")

# 1. Bar Plot of Country Counts
plt.figure(figsize=(12, 6))
sns.countplot(y="Country", data=df, order=df["Country"].value_counts().index[:20])
plt.title('Top 20 Countries by Game Counts')
plt.xlabel('Number of Games')
plt.ylabel('Country')
plt.show()

# 2. Pie Chart of Country Distribution
country_counts = df['Country'].value_counts().head(10)
plt.figure(figsize=(8, 8))
plt.pie(country_counts, labels=country_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Games by Top 10 Countries')
plt.show()
```





Both visualizations clearly highlight that Russia is the leading country in terms of the number of games played.

In [64]:
```
# 3. Choropleth Map of Game Counts
country_game_counts = df['Country'].value_counts().reset_index()
country_game_counts.columns = ['Country', 'Game Count']
fig = px.choropleth(country_game_counts,
                    locations="Country",
                    locationmode='country names',
                    color="Game Count",
                    hover_name="Country",
                    color_continuous_scale=px.colors.sequential.Plasma,
                    title="Game Counts by Country")
fig.update_layout(geo=dict(showcoastlines=True))
fig.show()
```



From the choropleth plot, bar plot, and pie chart, we can see that Russia has played the most games:

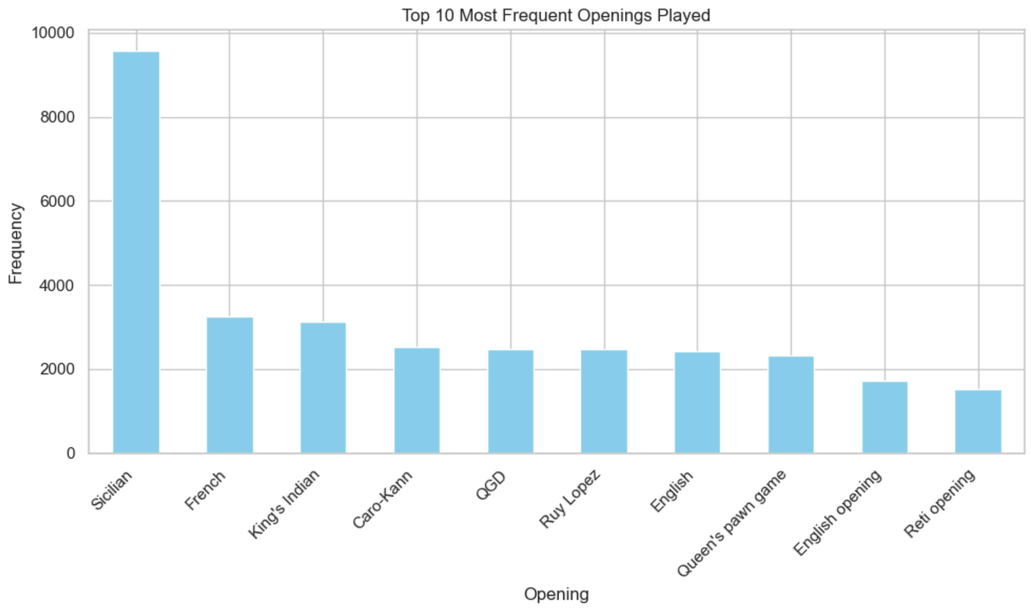
Choropleth Plot: The choropleth plot visually represents the number of games played by each country on a map. Russia is prominently highlighted, indicating it has a high number of games compared to other countries.

Bar Plot: The bar plot shows the number of games played by different countries, with Russia having the highest bar, indicating it has played the most games.

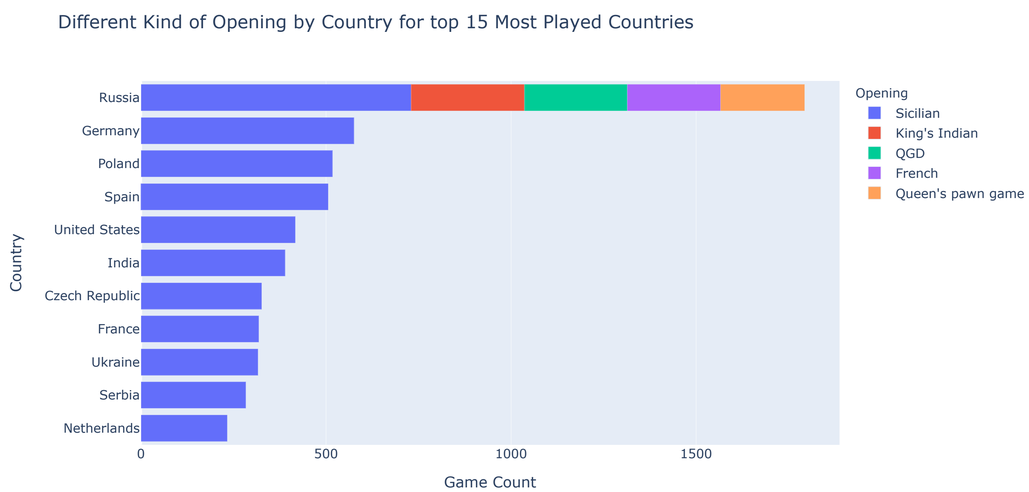
Pie Chart: The pie chart illustrates the distribution of games among countries, with the largest segment belonging to Russia, showing it has the highest participation in games.

All three visualizations consistently indicate that Russia has played the most games.

In [65]:
```
# Get the top 10 most frequent openings played by players
top_openings_white = df['Opening'].value_counts(ascending=False).head(10)

# Plotting
plt.figure(figsize=(10, 6))
top_openings_white.plot(kind='bar', color='skyblue')
plt.title('Top 10 Most Frequent Openings Played')
plt.xlabel('Opening')
plt.ylabel('Frequency')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



In [66]:
```
# Stacked Bar Plot of Top Players by Country
top_players = df.groupby(['Country', 'Opening']).size().sort_values().tail(15).reset_index(name='Game Count')
top_players = top_players.sort_values(by='Game Count', ascending=False).groupby('Country').head(5)
fig = px.bar(top_players, x='Game Count', y='Country', color='Opening',
             title='Different Kind of Opening by Country for top 15 Most Played Countries', barmode='stack')
fig.update_layout(yaxis={'categoryorder':'total ascending'})
fig.show()
```



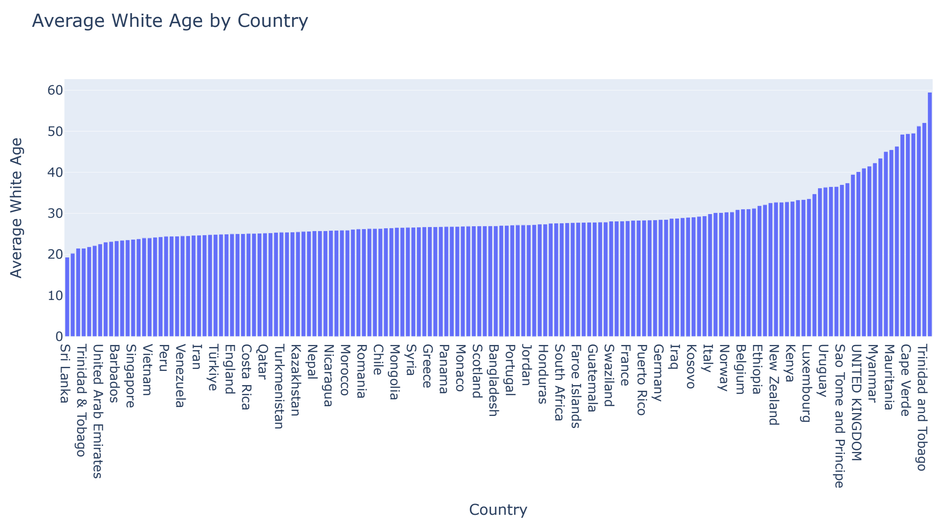
From the stacked bar chart, we can observe that the Sicilian Defense was the most common opening among the top 15 most played countries. Here's the detailed explanation:

Sicilian Defense Popularity: The stacked bar chart shows various openings used by players from the top 15 countries. The section representing the Sicilian Defense is the largest across these countries, indicating it was the most frequently chosen opening.

Comparison with Other Openings: Compared to other openings like the King's Indian Defense, Queen's Gambit Declined, and Ruy Lopez, the Sicilian Defense has a noticeably larger representation in the chart. This highlights its popularity and preference among top players from these countries.

Overall, the stacked bar chart clearly demonstrates that the Sicilian Defense is the most common opening among the top 15 most played countries, reflecting its strategic preference in high-level games.

In [67]:
```
# Group by 'Country' and calculate the mean of 'White Age'
avg_age_by_country = df.groupby('Country')['White Age'].mean().reset_index()

# Sort the DataFrame by mean age
avg_age_by_country = avg_age_by_country.sort_values('White Age', ascending=True)

# Create bar plot
fig = px.bar(avg_age_by_country, x='Country', y='White Age',
             title='Average White Age by Country',
             labels={'Country': 'Country', 'White Age': 'Average White Age'})

fig.update_layout(xaxis={'categoryorder':'total ascending'})  # Sort bars by descending average age

fig.show()
```



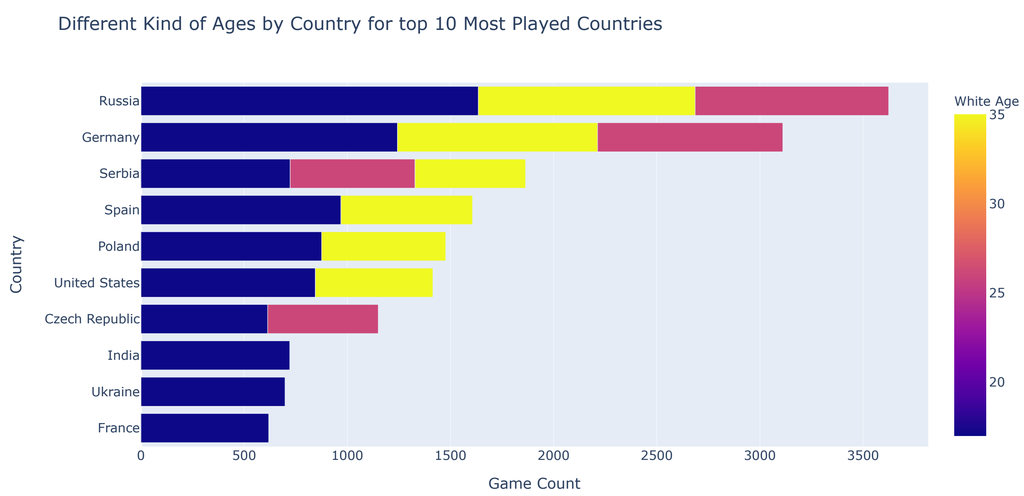
From the bar plot, we can observe the average age of players from different countries:

Scotland: The bar representing Scotland is the highest, indicating that players from Scotland have the highest average age compared to players from other countries. This suggests that Scotland has the oldest players on average.

Tanzania: Conversely, the bar for Tanzania is the lowest, indicating that players from Tanzania have the lowest average age. This means that Tanzania has the youngest players on average.

In summary, the bar plot shows that Scotland has the oldest players on average, while Tanzania has the youngest players on average.

In [68]:
```
# Stacked Bar Plot of Age By Country
top_players = df.groupby(['Country', 'White Age']).size().sort_values().tail(20).reset_index(name='Game Count')
top_players = top_players.sort_values(by='Game Count', ascending=False).groupby('Country').head(5)
fig = px.bar(top_players, x='Game Count', y='Country', color='White Age',
             title='Different Kind of Ages by Country for top 10 Most Played Countries', barmode='stack')
fig.update_layout(yaxis={'categoryorder':'total ascending'})
fig.show()
```



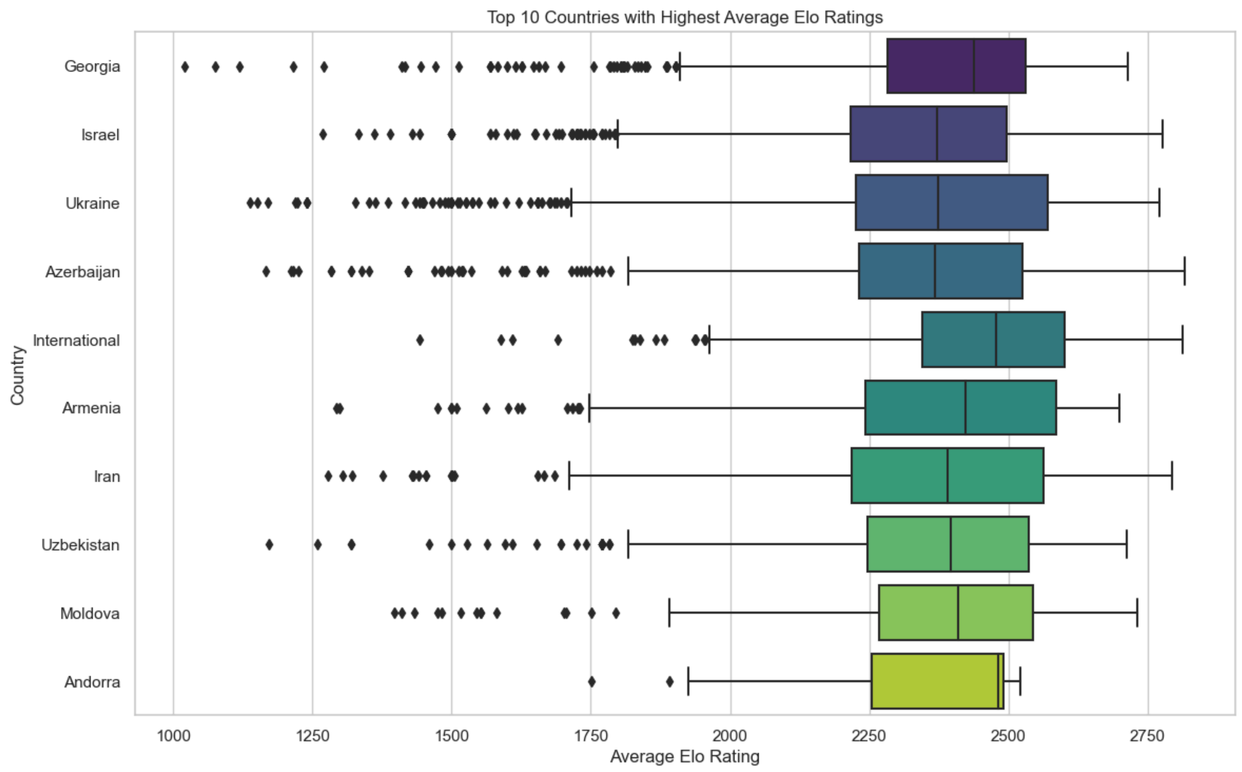
From the stacked bar plot, we can observe the distribution of games played by different age groups:

Age Group Around 20: The largest section of the bars corresponds to the age group around 20. This indicates that most games are played by players in this age range.

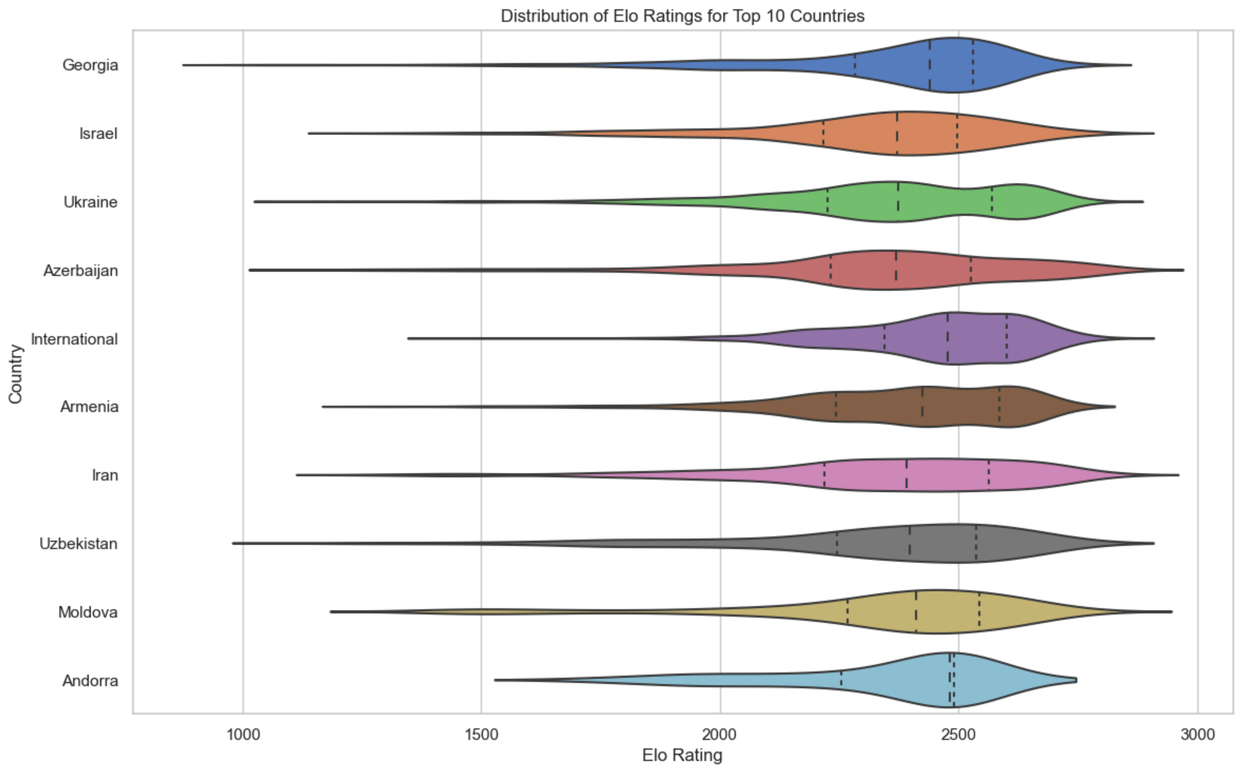
Comparison with Other Age Groups: Other age groups have smaller sections in the bars, showing fewer games played compared to the age group around 20. This highlights that players in their 20s are the most active in terms of game participation.

In summary, the stacked bar plot reveals that the majority of games are played by players around the age of 20, making this the most common age range for active participation in chess games.

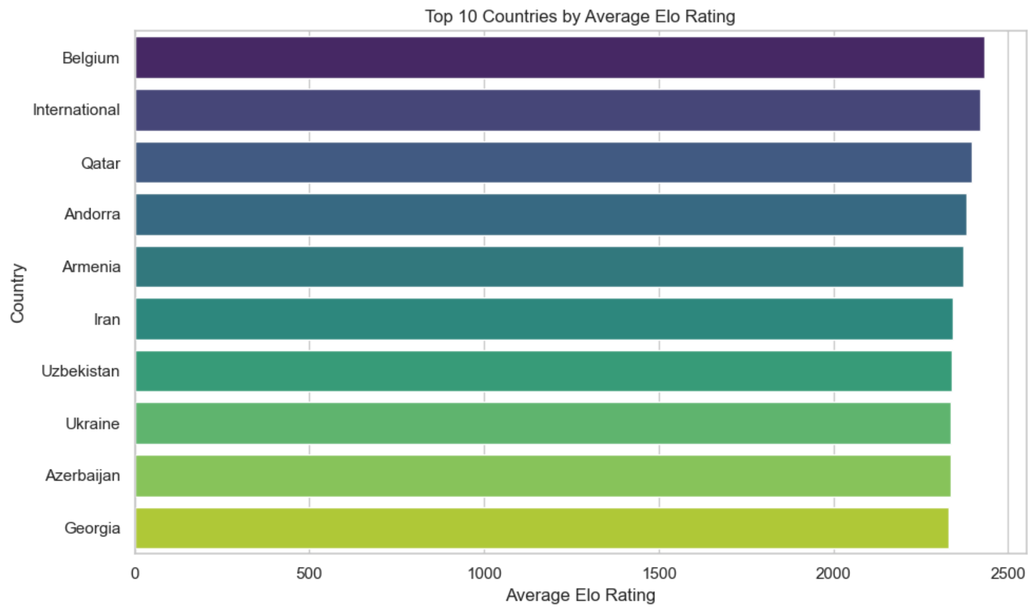
In [69]:
```
# Calculate average Elo rating for each country
average_elo_by_country = df.groupby('Country')['WhiteElo'].mean().reset_index()

# Sort countries based on average Elo ratings
average_elo_by_country_sorted = average_elo_by_country.sort_values(by='WhiteElo', ascending=False)

# Select top 10 countries with highest average Elo ratings
top_10_countries = average_elo_by_country_sorted.head(10)

# Plotting
plt.figure(figsize=(12, 8))
sns.boxplot(x='WhiteElo', y='Country', data=df[df['Country'].isin(top_10_countries['Country'])], palette='viridis')
plt.title('Top 10 Countries with Highest Average Elo Ratings')
plt.xlabel('Average Elo Rating')
plt.ylabel('Country')
plt.show()
```



In [70]:
```
# Calculate average Elo rating for each country
average_elo_by_country = df.groupby('Country')['WhiteElo'].mean().reset_index()

# Sort countries based on average Elo ratings
average_elo_by_country_sorted = average_elo_by_country.sort_values(by='WhiteElo', ascending=False)

# Select top 10 countries with highest average Elo ratings
top_10_countries = average_elo_by_country_sorted.head(10)

# Plotting
plt.figure(figsize=(12, 8))
sns.violinplot(x='WhiteElo', y='Country', data=df[df['Country'].isin(top_10_countries['Country'])], palette='muted', inner='quartile')
plt.title('Distribution of Elo Ratings for Top 10 Countries')
plt.xlabel('Elo Rating')
plt.ylabel('Country')
plt.show()
```



In [71]:
```
# Calculate the average Elo rating of players from each country
country_avg_elo = df_countries.groupby('Country')[['WhiteElo', 'BlackElo']].mean().mean(axis=1).sort_values(ascending=False).head(10)

# Create the bar plot
plt.figure(figsize=(10, 6))
sns.barplot(x=country_avg_elo.values, y=country_avg_elo.index, palette='viridis')
plt.title('Top 10 Countries by Average Elo Rating')
plt.xlabel('Average Elo Rating')
plt.ylabel('Country')
plt.tight_layout()
plt.show()
```



Elo is a rating system originally developed by Arpad Elo, a Hungarian-American physics professor and chess master, to calculate the relative skill levels of players in two-player games such as chess. The Elo rating system is now widely used in various competitive activities, including chess, esports, and other competitive games.

In chess, every player has an Elo rating, which represents their skill level relative to other players. A higher Elo rating indicates a stronger player, while a lower Elo rating indicates a weaker player. When two players with different Elo ratings compete against each other, the expected outcome of the game can be predicted based on their Elo ratings. If a player with a higher Elo rating wins against a player with a lower Elo rating, they will gain fewer Elo points than if they were playing against a player with a similar or higher rating. Conversely, if a player with a lower Elo rating wins against a player with a higher Elo rating, they will gain more Elo points.

Now, let's explain the visualization:

The visualization is a box plot that shows the distribution of Elo ratings among players from different countries. Here's what the visualization shows:

Y-Axis (Country): Each country included in the dataset is represented on the y-axis.

X-Axis (Elo Rating): The Elo ratings of the players are represented on the x-axis.

Box Plot: For each country, the box plot shows the distribution of Elo ratings among its players. The box plot includes:

Median (line inside the box): The line inside the box represents the median Elo rating of players from that country. Interquartile Range (box): The box represents the interquartile range (IQR), which is the range between the first and third quartiles of the Elo ratings. It gives us an idea of the spread of Elo ratings within each country.

Whiskers (lines extending from the box): The whiskers extend from the edges of the box to the minimum and maximum values that fall within 1.5 times the IQR. Any data points beyond the whiskers are considered outliers and are plotted individually.

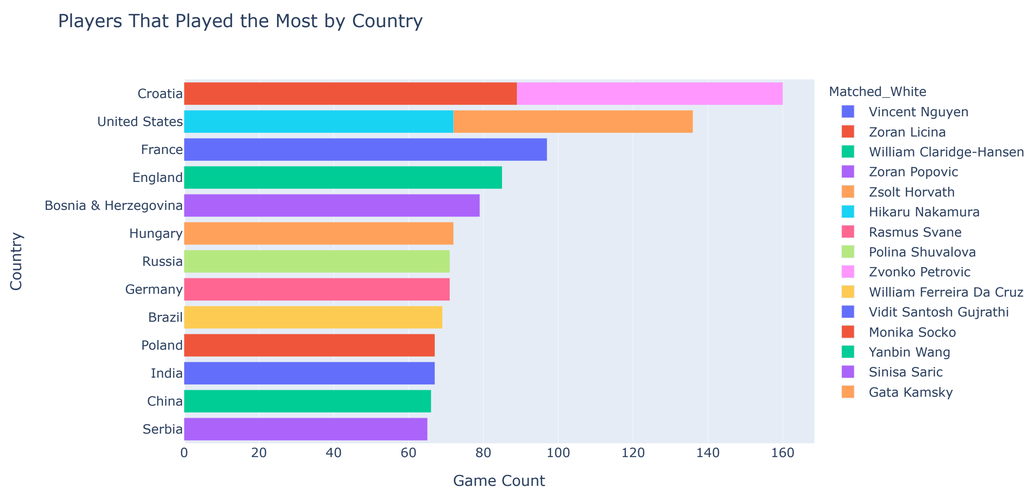
Outliers (individual points): Data points that fall beyond the whiskers are plotted as individual points. These are players whose Elo ratings are significantly higher or lower than the typical ratings for their country.

By visualizing the Elo ratings of players by country in this way, we can observe the distribution of player skill levels across different countries and identify any significant variations or outliers. This can provide insights into the relative strengths of players from different countries in the dataset.

In [72]:
```
players = df.groupby('Matched_White').agg({
    'Country':'first',
})
```

In [73]:
```
players.reset_index(inplace = True)
```

In [74]:
```
# Convert Matched_White to string
df['Matched_White'] = df['Matched_White'].astype(str)

# 4. Stacked Bar Plot of Players That Played the Most by Country
top_players = df.groupby(['Country', 'Matched_White']).size().sort_values().tail(15).reset_index(name='Game Count')
top_players = top_players.sort_values(by='Game Count', ascending=False).groupby('Country').head(5)
fig = px.bar(top_players, x='Game Count', y='Country', color='Matched_White',
             title='Players That Played the Most by Country', barmode='stack')
fig.update_layout(yaxis={'categoryorder':'total ascending'})
fig.show()
```



For each country (each bar), you can see how the total number of games played is divided among different player categories.

The length of each segment within a bar indicates the proportion of games played by players of that category from that country.

Countries with taller bars have more total games played, while the segments within each bar show the distribution of games among different player categories for that country.

This visualization provides insights into which countries have the most active and diverse player bases, as well as the distribution of player categories within each country.

In [75]:
```
# Convert the "Round" column to string to handle mixed data types
df['Round'] = df['Round'].astype(str)

# Define a function to handle different formats and round the values
def round_and_convert(value):
    # If the value is '?' or empty, return NaN
    if value == '?' or value == '' or value == '-':
        return None
    elif ',' in value:
        # Check if the string is a valid float
        try:
            return float(round(float(value)))
        except ValueError:
            # If not, split by '.' and take the first part
            return float(value.split(',')[0])
    # If it's a float represented as a string
    elif '.' in value:
        # Check if the string is a valid float
        try:
            return float(round(float(value)))
        except ValueError:
            # If not, split by '.' and take the first part
            return float(value.split('.')[0])
    # If it's an integer, return it as an integer
    else:
        return float(value)

# Apply the function to the "Round" column
df['Round'] = df['Round'].apply(round_and_convert)
```

In [76]:
```
df['Round'] = df['Round'].astype(float)
```

In summary, this code is designed to clean and standardize the "Round" column by handling mixed data types, special characters, and different formats, ultimately converting all values to floats for consistent analysis.

In [77]:
```
df.head(3)
```

Out[77]:

	Event	Result	mainline_moves	Site	Online	Round	ECO	Opening	WhiteElo	BlackElo	Variation	WhiteTitle	BlackTitle	WhiteTeam	BlackTeam	EventType	Matched_White	Matched_Black	Country	Black Country	White K-factor	White Age	Black K-factor	Black Age	Date
0	WY GU16 2012	2.0	1. e4 d5 2. exd5 Qxd5 3. Nc3 Qd8 4. Bc4 Nf6 5....	Maribor SLO	False	8.0	B01	Scandinavian (centre counter) defence	1471.0	1775.0	None	None	None	None	None	swiss	Figueroa Calderon, Jenid	Tim Janzelj	Puerto Rico	Slovenia	20	26	20	38	2012-11-15
1	41st Olympiad Women 2014	0.0	1. d4 Nf6 2. c4 e6 3. Nc3 Bb4 4. Qc2 c5 5. e3 ...	Tromso NOR	False	10.0	E38	Nimzo-Indian	1895.0	2232.0	classical, 4...c5	None	WIM	Chile	Iran	None	Fuentes Inzunza, Lesly	Ghazal Hakimifard	Chile	Switzerland	20	35	40	17	2014-08-12
2	TCh-GEO Club 2016	0.0	1. Nf3 d5 2. g3 Nc6 3. d4 Bg4 4. Nbd2 f6 5. h3...	Lagodekhi GEO	False	6.0	A07	Reti	2601.0	2492.0	King's Indian attack (Barcza system)	GM	IM	Tbilisi	Samegrelo-Zemo Svaneti - Samegrelo	None	Levan Pantsulaia	Vahe Baghdasaryan	Georgia	Armenia	20	35	40	17	2016-09-27

In [78]:
```
countries_df = df.groupby('Country').agg({
    'Event':'max',
    'Result':'mean',
    'mainline_moves':'count', # Actually the amount of games !!!!!
    'Site':'max',
    'Online':'count',
    'Round':'mean',
    'ECO':'max',
    'Opening':'max',
    'WhiteElo':'mean',
    'Variation':'max',
    'WhiteTitle':'max',
    'EventType':'max',
    'Black Country':'max', #Mostly played against
    'White Age':'mean',
    'White K-factor':'mean',
})
```

In [79]:
```
countries_df.rename(columns={'mainline_moves': 'Number of Games Played',
                             'Black Country': 'Mostly Played Against'}, inplace=True)
```

In [80]:
```
countries_df = countries_df.reset_index()
```

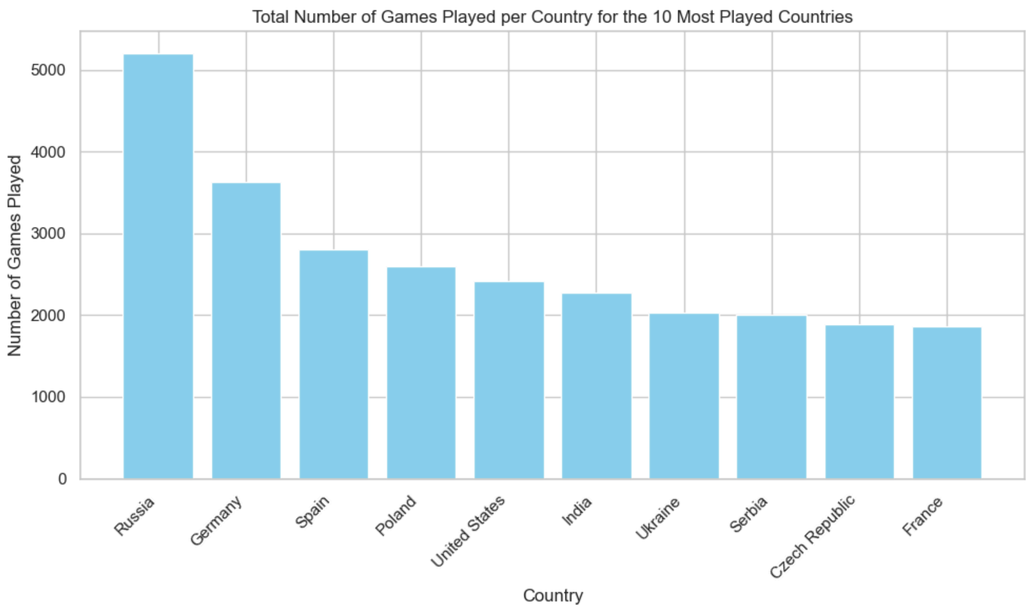
In [81]:
```
countries_df.head()
```

Out[81]:

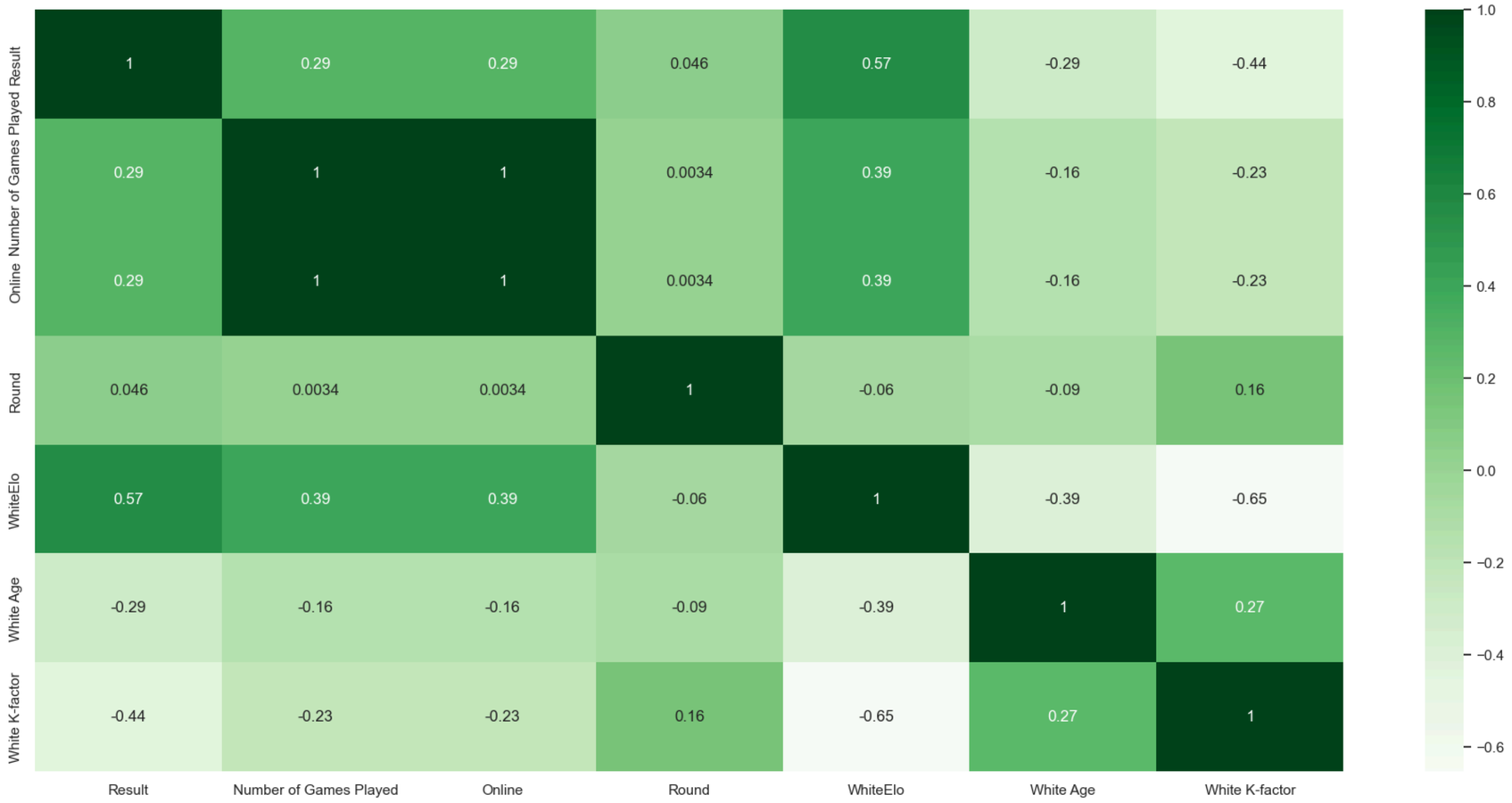
	Country	Event	Result	Number of Games Played	Site	Online	Round	ECO	Opening	WhiteElo	Variation	WhiteTitle	EventType	Mostly Played Against	White Age	White K-factor
0	Albania	ch-ALB 2020	0.709677	31	lichess.org INT	31	5.225806	E94	Sicilian	2025.838710	wing gambit, Carlsbad variation	WCM	team	UNITED KINGDOM	32.709677	24.516129
1	Algeria	ch-TUR 2013	0.836957	92	lichess.org INT	92	4.978261	E62	Vienna game	2164.543478	modern exchange variation	WIM	team	Zambia	26.804348	21.304348
2	Andorra	Titled Tuesday 29th June	0.968750	32	chess24.com INT	32	4.875000	E60	Vienna	2345.625000	symmetrical variation	None	team	United States	27.625000	20.000000
3	Angola	ch-POR Blitz 2017	0.896552	29	chess.com INT	29	4.241379	E60	Sicilian	2060.241379	exchange variation	WIM	team	Zimbabwe	25.034483	21.379310
4	Argentina	e2e4 Sunningdale Open	0.985646	418	lichess.org INT	418	5.535109	E98	Vienna	2259.921053	ultra-symmetrical variation	WIM	tourn	Vietnam	26.863636	20.622010

In [82]:
```
top10played = countries_df.sort_values('Number of Games Played', ascending = False).head(10)
```

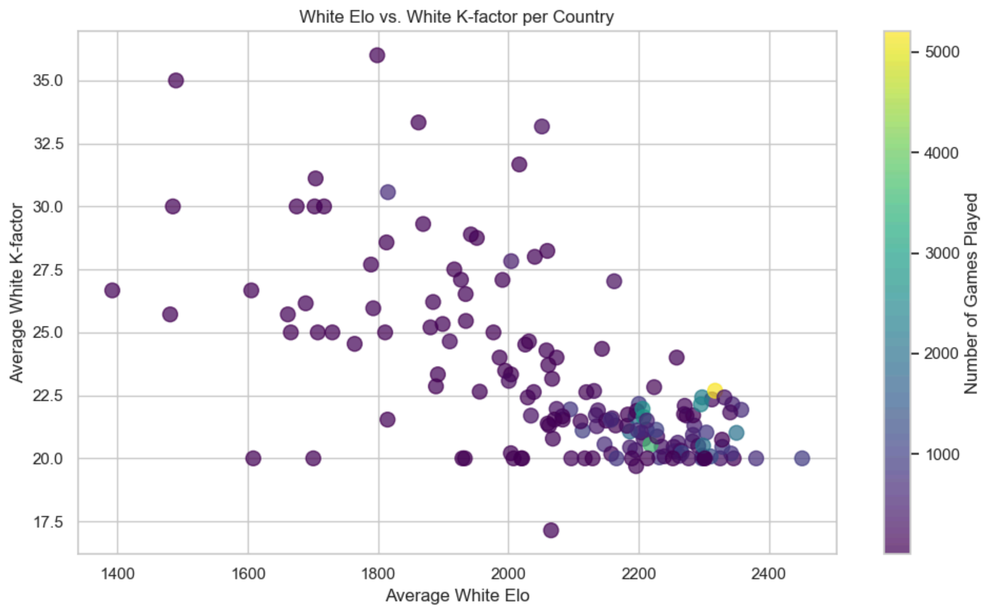
In [83]:
```
plt.figure(figsize=(10, 6))
plt.bar(top10played['Country'], top10played['Number of Games Played'], color='skyblue')
plt.title('Total Number of Games Played per Country for the 10 Most Played Countries')
plt.xlabel('Country')
plt.ylabel('Number of Games Played')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



In [84]:
```
plt.figure(figsize=(20,10))
sns.heatmap(countries_df.select_dtypes(include='number').corr(),annot=True,cmap='Greens')
plt.show()
```



In [85]:
```
plt.figure(figsize=(10, 6))
plt.scatter(countries_df['WhiteElo'], countries_df['White K-factor'], c=countries_df['Number of Games Played'], cmap='viridis', s=100, alpha=0.7)
plt.colorbar(label='Number of Games Played')
plt.title('White Elo vs. White K-factor per Country')
plt.xlabel('Average White Elo')
plt.ylabel('Average White K-factor')
plt.grid(True)
plt.tight_layout()
plt.show()
```



The observation that there is a negative correlation between Elo rating and K-factor is consistent with the principles of how the Elo rating system is designed to work. Let's explain this in detail:

Explanation

High Elo Rating and Low K-Factor:

Stability for Experienced Players: Players with high Elo ratings are generally more experienced and their skill levels are well-established. Therefore, the K-factor is lower to ensure that their ratings do not fluctuate significantly with each game. This reflects the fact that a single game result is less likely to represent a significant change in their actual skill level. Example: A Grandmaster with an Elo rating of 2700 will have a low K-factor, so winning or losing a game will cause a relatively small change in their rating. Low Elo Rating and High K-Factor:

Rapid Adjustment for New or Improving Players: Players with lower Elo ratings, especially new players or those improving quickly, have higher K-factors. This allows their ratings to adjust more rapidly to reflect their true skill level as they play more games.

Example: A new player with an Elo rating of 1200 might have a high K-factor, resulting in larger changes in their rating after each game, allowing their rating to quickly become more accurate.

Practical Impact

Negative Correlation: This setup naturally leads to a negative correlation between Elo rating and K-factor. As players' Elo ratings increase and stabilize, their K-factors decrease. Conversely, players with lower, more variable ratings have higher K-factors to allow for quicker adjustments.

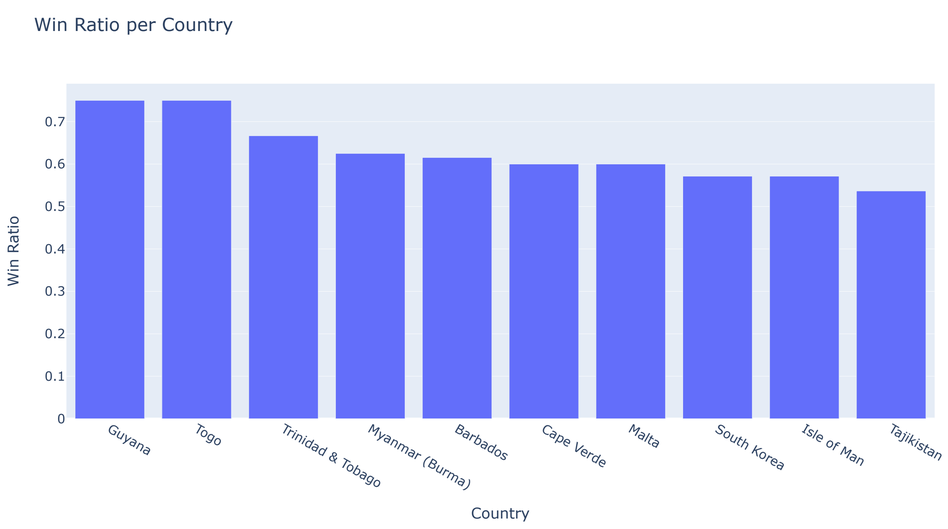
In [86]:
```
# Create a new DataFrame to count the results per country
result_counts = df.groupby(['Country', 'Result']).size().reset_index(name='Count')

# Pivot the table to get the desired format
result_counts_pivot = result_counts.pivot_table(index='Country', columns='Result', values='Count', fill_value=0)

# Rename the columns
result_counts_pivot.columns = ['Number of Games Lost', 'Number of Games Win', 'Number of Tie Games']

# Reset index to make 'Country' a regular column
result_counts_pivot.reset_index(inplace=True)

# Add a new column for total number of games
result_counts_pivot['Total Games'] = result_counts_pivot['Number of Games Lost'] + result_counts_pivot['Number of Games Win'] + result_counts_pivot['Number of Tie Games']

# Add a new column for win ratio
result_counts_pivot['Win Ratio'] = result_counts_pivot['Number of Games Win'] / result_counts_pivot['Total Games']

# Display the result
result_counts_pivot
```

Out[86]:

	Country	Number of Games Lost	Number of Games Win	Number of Tie Games	Total Games	Win Ratio
0	Albania	16	8	7	31	0.258065
1	Algeria	34	39	19	92	0.423913
2	Andorra	12	9	11	32	0.281250
3	Angola	10	12	7	29	0.413793
4	Argentina	118	188	112	418	0.449761
...	...	...	...	...	...	...
153	WALES	23	16	19	58	0.275862
154	Wales	25	27	18	70	0.385714
155	Yemen	2	1	2	5	0.200000
156	Zambia	21	14	13	48	0.291667
157	Zimbabwe	19	14	7	40	0.350000

158 rows × 6 columns

In [87]:
```
# Create a bar chart for win ratio per country
fig = px.bar(result_counts_pivot.sort_values('Win Ratio', ascending = False).head(10)
             , x='Country', y='Win Ratio', title='Win Ratio per Country')
fig.show()
```



Win Ratio Calculation: Each country's win ratio may be calculated as the number of wins divided by the total number of games played by players from that country.

Conclusion:

The statement suggests that by examining the provided chart, you can identify the country with the highest win ratio, providing valuable insights into the performance of players from different countries in the analyzed dataset.

Part B

In [88]:
```
df1 = pd.read_csv('twic_master.csv')
```

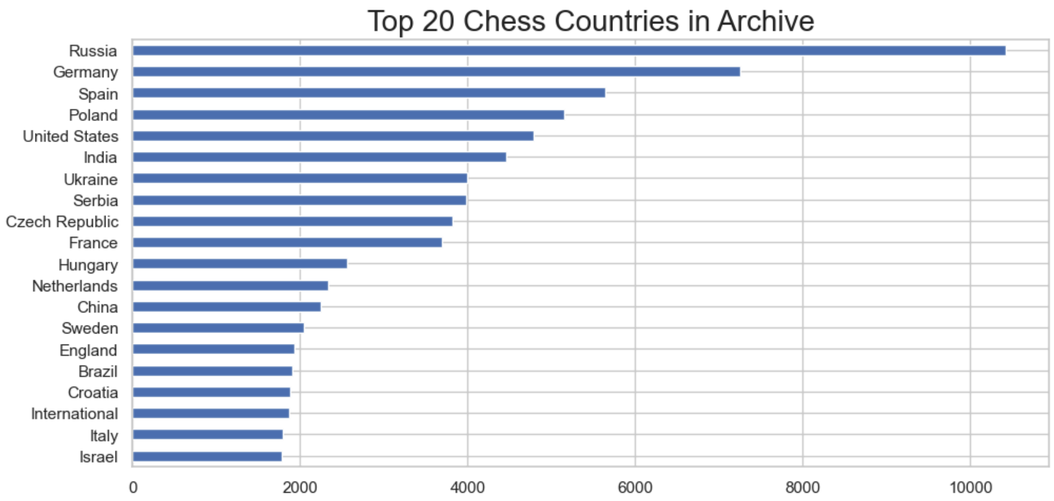
```
C:\Users\kazom\AppData\Local\Temp\ipykernel_10096\836247307.py:1: DtypeWarning:

Columns (10,13,14,23,25) have mixed types. Specify dtype option on import or set low_memory=False.
```

In [89]:
```
def split_and_concat(names):
    if isinstance(names, str):
        # Split the names by comma and concatenate into a single string
        return ''.join(names.split(','))
    else:
        return ''

# Apply the function to the 'White' column and the 'Black' column
df1['White_concat'] = df1['White'].apply(split_and_concat)
df1['Black_concat'] = df1['Black'].apply(split_and_concat)
```

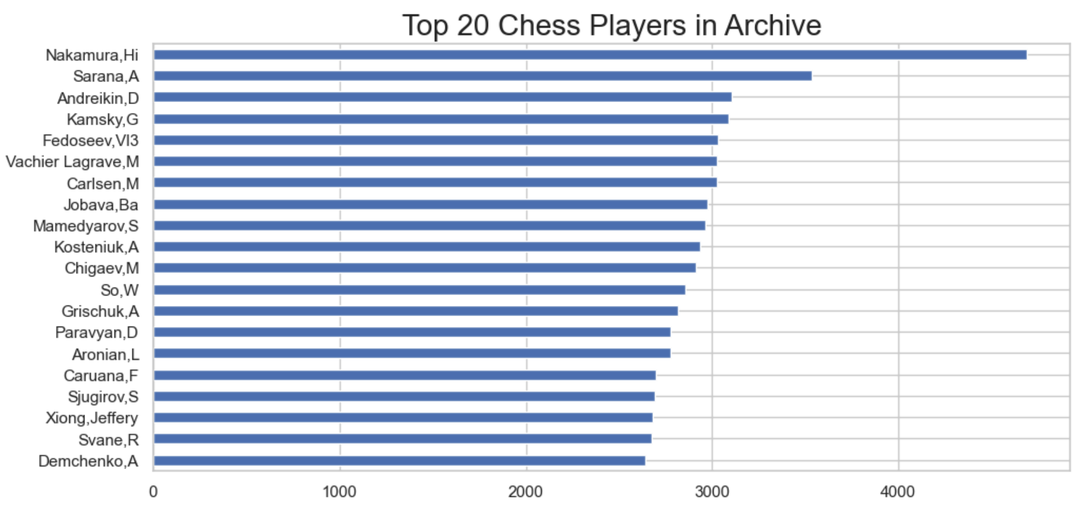
In [90]:
```
df1.head()['White_concat']
```

Out[90]:
```
0         TukhaevA
1       NakamuraHi
2            TariA
3      FedoseevVl3
4    HovhannisyanR
Name: White_concat, dtype: object
```

In [91]:
```
# Who are the most frequent Countries in the archive? ###
fig , ax = plt.subplots(figsize=(10,5))
pd.concat([df['Country'], df['Black Country']]).value_counts(ascending=True).tail(20).plot(kind='barh', ax=ax)
ax.set_title('Top 20 Chess Countries in Archive', fontsize=20)
plt.show()
```



In [92]:
```
# Who are the most frequent players in the archive?
fig , ax = plt.subplots(figsize=(10,5))
pd.concat([df1['White'], df1['Black']]).value_counts(ascending=True).tail(20).plot(kind='barh', ax=ax)
ax.set_title('Top 20 Chess Players in Archive', fontsize=20)
plt.show()
```



In [93]:
```
#What were the top openings each year? ###

# Convert 'Date' column to datetime type
df['Date'] = pd.to_datetime(df['Date'])

# Now you can use the .dt accessor to extract year
df['Year'] = df['Date'].dt.year

df['Year'].value_counts() ######
```

Out[93]:
```
Year
2021    9135
2022    6911
2020    6806
2019    5643
2018    5530
2017    5098
2016    4607
2014    4071
2015    3955
2013    3888
2012    2488
2011       3
2010       1
Name: count, dtype: int64
```

In [94]:
```
#What were the top openings each year?

# Convert 'Date' column to datetime type
df1['Date'] = pd.to_datetime(df1['Date'])

# Now you can use the .dt accessor to extract year
df1['Year'] = df1['Date'].dt.year

df1['Year'].value_counts()
```

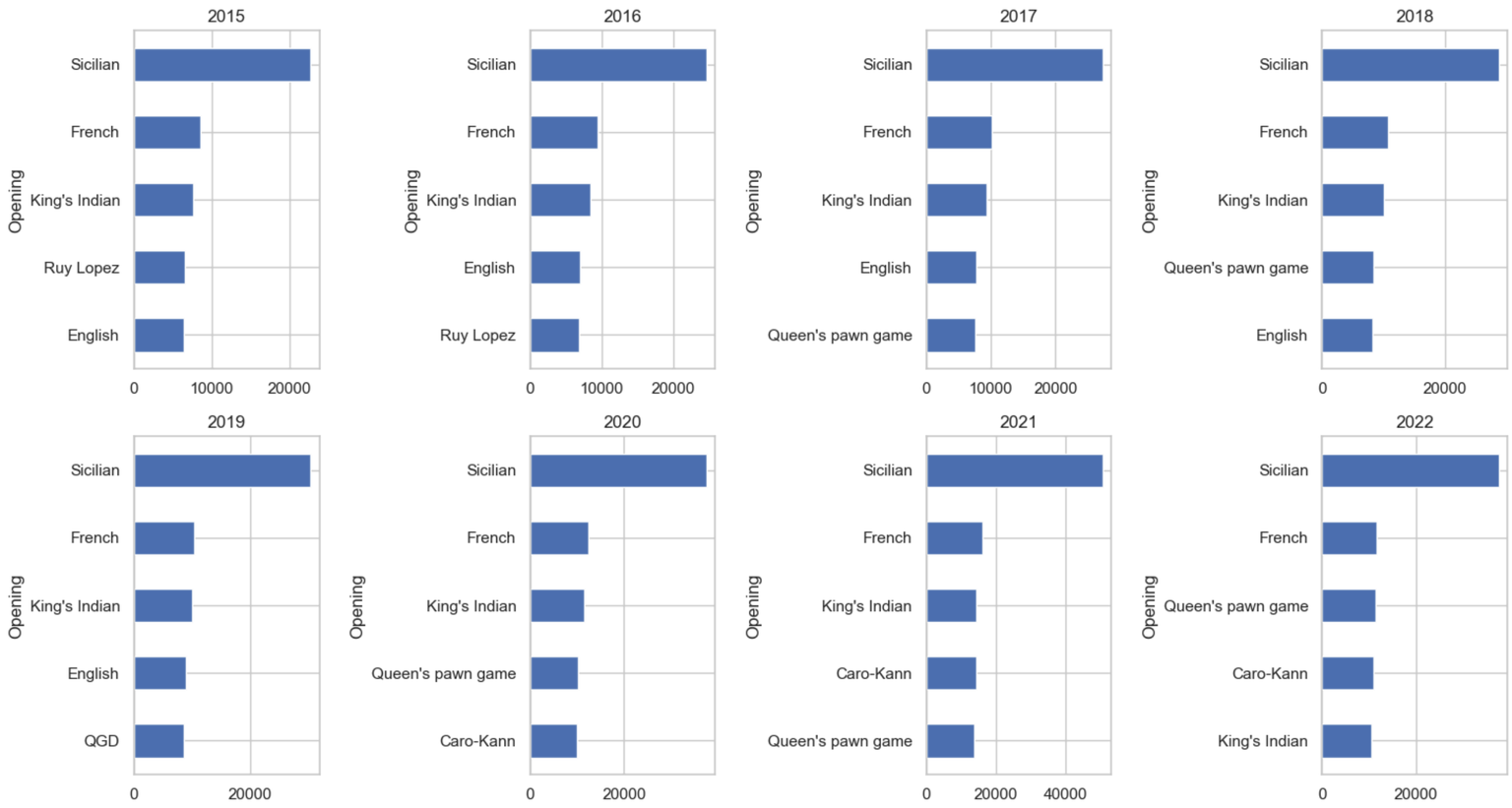
Out[94]:
```
Year
2021    302824
2022    226060
2020    222457
2019    188686
2018    185950
2017    173494
2016    154094
2015    142258
2013    141690
2014    140827
2012     92916
2011        54
2010         6
2005         1
1988         1
2029         1
Name: count, dtype: int64
```

In [95]:
```
# df1 = df1.query('2023 > Year > 2012')
```

In [96]:
```
df1["Year"].unique()
```

Out[96]:
```
array([2018, 2017, 2015, 2014, 2020, 2016, 2013, 2012, 2022, 2021, 2019,
       2005, 1988, 2029, 2011, 2010])
```

In [97]:
```
# Top Openings by Year
fig, axs = plt.subplots(2,4, figsize=(15,8))
axs = axs.flatten()

for i, myyear in enumerate(range(2015,2023)):
    df1.query("Year == @myyear")["Opening"].value_counts(ascending=True).tail(5).plot(kind="barh",
                                                                                       title=myyear,ax = axs[i])
plt.tight_layout()
plt.show()
```



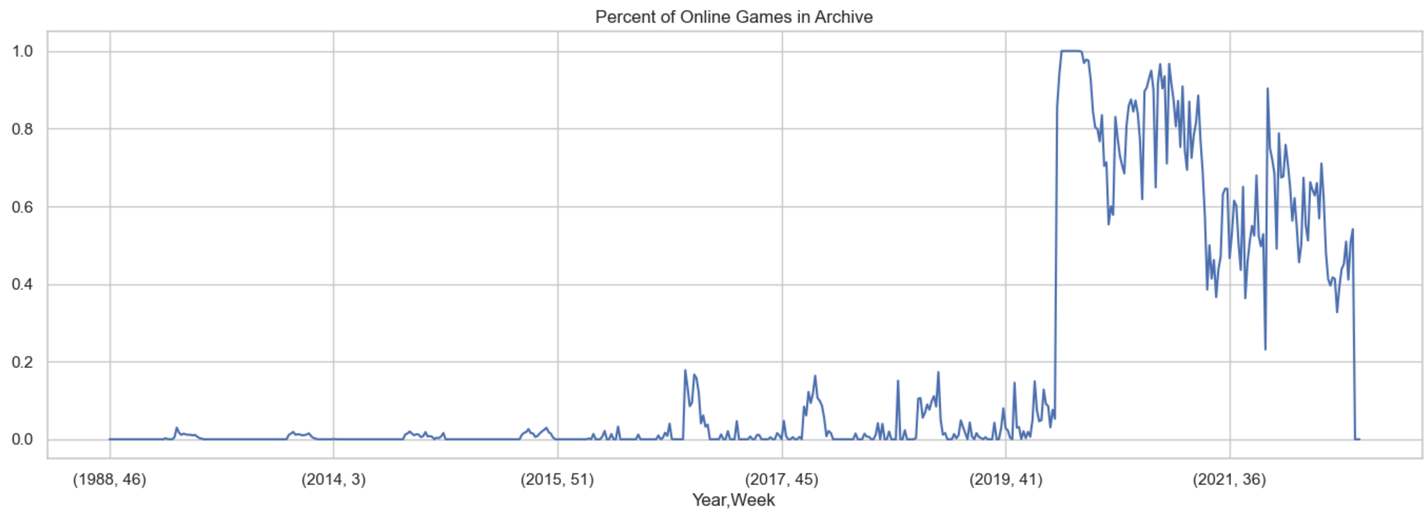
Consistent Trend: Across all years from 2015 to 2022, the Sicilian opening consistently emerges as one of the most common openings. This suggests that the Sicilian opening is popular among players worldwide over these years.

In [98]:
```
df['Week'] = df['Date'].dt.isocalendar().week
df = df.sort_values("Date")
```

In [99]:
```
# What $ of games in the Archive were online over time?
```

In [100]:
```
df1['Week'] = df1['Date'].dt.isocalendar().week
df1 = df1.sort_values("Date")
```

In [101]:
```
fig, ax = plt.subplots(figsize=(15,5))
df1.groupby(["Year", "Week"])["Online"].mean().plot(
title="Percent of Online Games in Archive")
plt.show()
```



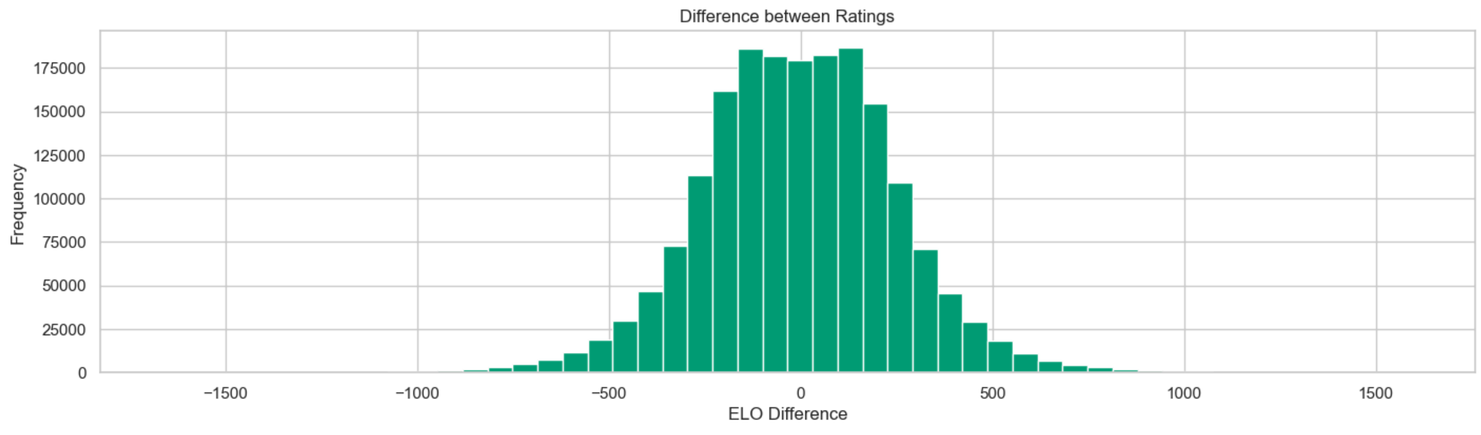
The sharp rise in online chess games around 2019 coincides with the onset of the COVID-19 pandemic. Lockdown measures and the closure of physical venues likely prompted players to seek online alternatives, leading to a surge in virtual chess activity. This trend reflects the broader societal shift towards digital platforms for recreation and social interaction during the pandemic.

In [102]:
```
# Use Chess Package to Review Game
```

In [103]:
```
df_country = ( ##########
    df.dropna(subset=["BlackElo", "WhiteElo"]).astype({
        "BlackElo":"int","WhiteElo":"int"}).copy()
    )
df_country["Elo_diff"] = df_country["BlackElo"] - df_country["WhiteElo"]
# df_ = df_.drop(1662955).copy()
```

In [104]:
```
df_ = (
    df1.dropna(subset=["BlackElo", "WhiteElo"]).astype({
        "BlackElo":"int","WhiteElo":"int"}).copy()
    )
df_["Elo_diff"] = df_["BlackElo"] - df_["WhiteElo"]
df_ = df_.drop(1662955).copy()
```

In [105]:
```
ax = df_["Elo_diff"].plot(kind="hist", bins=50, title="Difference between Ratings", color=pal[1])
ax.set_xlabel("ELO Difference")
plt.show()
```



In [106]:
```
df_country["WhiteIsRatedHigher"] = df_country["WhiteElo"] > df_country["BlackElo"] ########
```

In [107]:
```
df_["WhiteIsRatedHigher"] = df_["WhiteElo"] > df_["BlackElo"]
```

In [108]:
```
df_country[((df_country['Result'] == 0) &
          (df_country['WhiteIsRatedHigher'] == 1))
           |
          ((df_country['Result'] == 1) &
          (df_country['WhiteIsRatedHigher'] == 0)) ]
```

Out[108]:

	Event	Result	mainline_moves	Site	Online	Round	ECO	Opening	WhiteElo	BlackElo	Variation	WhiteTitle	BlackTitle	WhiteTeam	BlackTeam	EventType	Matched_White	Matched_Black	Country	Black Country	White K-factor	White Age	Black K-factor	Black Age	Date	Year	Week	Elo_diff	WhiteIsRatedHigher
67303	16th Voronezh Master Open	1.0	1. e4 c5 2. Nf3 Nc6 3. Bb5 e6 4. O-O Nge7 5. c...	Voronezh RUS	False	1.0	B30	Sicilian	2290	2571	Nimzovich-Rossolimo attack (without ...d6)	FM	GM	None	None	None	Anton Samojlov	Michael Fedorovsky	Russia	Germany	20	17	40	13	2012-06-12	2012	24	281	False
61439	16th Voronezh Master Open	1.0	1. e4 c5 2. Nf3 e6 3. d4 cxd4 4. Nxd4 a6 5. Bd...	Voronezh RUS	False	7.0	B42	Sicilian	2538	2617	Kan, Polugaievsky variation	GM	GM	None	None	None	Gellert Voros	Mariya Zubova	Slovakia	Ukraine	20	17	40	13	2012-06-19	2012	25	79	False
42348	Teplice Open 2012	1.0	1. e4 d5 2. exd5 Nf6 3. Nc3 Nxd5 4. Bc4 Nb6 5....	Teplice CZE	False	6.0	B01	Scandinavian defence	1696	1869	None	None	None	None	None	None	Divisova, Hana	Krizsan-Bilek, Gyulane	Czech Republic	Hungary	40	37	20	84	2012-06-20	2012	25	173	False
62382	65th ch-RUS HL	1.0	1. c4 c5 2. Nf3 Nc6 3. Nc3 g6 4. e3 Nf6 5. d4 ...	Tyumen RUS	False	8.0	A30	English	2642	2677	symmetrical variation	GM	GM	None	None	None	Vladimir Potkin	Vadim Alekseev	Russia	Russia	20	17	40	13	2012-06-24	2012	25	35	False
26772	62nd ch-RUS HL w	1.0	1. e4 c5 2. Nf3 e6 3. d4 cxd4 4. Nxd4 a6 5. Bd...	Tyumen RUS	False	11.0	B42	Sicilian	1914	2088	Kan, 5.Bd3	None	WFM	None	None	None	Adamova, Tuyara	Korchenkova, Irina	Russia	Russia	20	30	20	31	2012-06-27	2012	26	174	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
69577	ch-POL Blitz 2022	1.0	1. d4 Nf6 2. c4 g6 3. Nc3 Bg7 4. e4 d6 5. Be2 ...	Suwalki POL	False	4.0	E94	King's Indian	1770	2158	orthodox, 7...Nbd7	None	None	None	None	None	Liskiewicz, Jakub	Dariusz Fistek	Poland	Poland	40	13	40	13	2022-09-24	2022	38	388	False

In [109]:
```
df_.query("(WhiteIsRatedHigher and Result == '0-1') or (not WhiteIsRatedHigher and Result == '1-0')")
```

Out[109]:

	twic_number	White	Black	Date	EventDate	Event	Result	mainline_moves	Site	Online	Round	ECO	Opening	WhiteFideId	BlackFideId	WhiteElo	BlackElo	Variation	WhiteTitle	BlackTitle	WhiteTeam	BlackTeam	EventType	FEN	SetUp	Variant	Board	PlyCount	EventCategory	White_concat	Black_conca
1210831	951	Sisniega Campbell,M	Illescas Cordoba,M	1988-11-18	1988.11.??	Olympiad Men	1-0	1. e4 c5 2. Nf3 e6 3. Nc3 Nc6 4. d4 cxd4 5. Nx...	Thessaloniki GRE	False	5	B80	NaN	5100020.0	2200015.0	2455	2475	NaN	NaN	NaN	Mexico	Spain	team	NaN	NaN	NaN	NaN	NaN	NaN	Sisniega CampbellM	Illescas Cordoba
1574139	934	Hevia,A	Leiva,G	2010-09-28	2012.09.25	1st Simon Bolivar Open 2012	0-1	1. e4 c5 2. Nf3 d6 3. d4 cxd4 4. Nxd4 Nf6 5. N...	San Cristobal VEN	False	7.17	B90	Sicilian	3508811.0	3804399.0	2518	2286	Najdorf, Byrne (English) attack	FM	FM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	HeviaA	Leiva
1516494	925	Salimova,Nurgyul	Ivanova,Si	2011-07-25	2012.07.23	Open ch-BUL w 2012	1-0	1. d4 d5 2. c4 c6 3. cxd5 cxd5 4. Nc3 Nf6 5. N...	Vratsa BUL	False	4.12	D10	QGD Slav defence	2915138	2900548	1753	1956	exchange variation	WFM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	SalimovaNurgyul	Ivanova
1516502	925	Olea,L	Salimova,Nurgyul	2011-07-26	2012.07.23	Open ch-BUL w 2012	0-1	1. d4 d5 2. Nf3 Nf6 3. e3 Bf5 4. c4 e6 5. Nc3 ...	Vratsa BUL	False	5.8	D04	Queen's pawn game	1202138	2915138	2014	1753	NaN	NaN	WFM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	OleaL	SalimovaNurgyu
1516498	925	Koskoska,G	Cherednichenko,E	2011-07-26	2012.07.23	Open ch-BUL w 2012	1-0	1. e4 e5 2. Nf3 Nc6 3. d4 exd4 4. Nxd4 Nf6 5. ...	Vratsa BUL	False	5.4	C45	Scotch	15000630	14105560	2094	2167	Mieses variation	WIM	WFM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	KoskoskaG	Cherednichenko

In [110]:
```
########
df_country[((df_country['Result'] == 0) &
          (df_country['WhiteIsRatedHigher'] == 1))
           |
          ((df_country['Result'] == 1) &
          (df_country['WhiteIsRatedHigher'] == 0)) ].shape[0] / df_country.shape[0]
```

Out[110]: 0.18928030824274117

The code snippet calculates the percentage of games where the player with the higher Elo rating (either White or Black) wins. Here's the breakdown:

Filtering Condition:

The code filters the DataFrame df_country based on two conditions: If the result of the game is 0 (indicating a draw) and the player with the higher Elo rating is White. If the result of the game is 1 (indicating a win) and the player with the higher Elo rating is Black. Calculation:

It computes the number of rows in the filtered DataFrame divided by the total number of rows in df_country. This ratio represents the proportion of games where the player with the higher Elo rating wins or draws. Output Interpretation:

The output value, approximately 20%, signifies that in about 20% of the games: Either the player with the higher Elo rating wins when playing as Black. Or the game results in a draw when the player with the higher Elo rating plays as White. This analysis provides insights into the influence of Elo ratings on game outcomes, suggesting that having a higher Elo rating does not guarantee a win and may result in a draw in a significant portion of games.

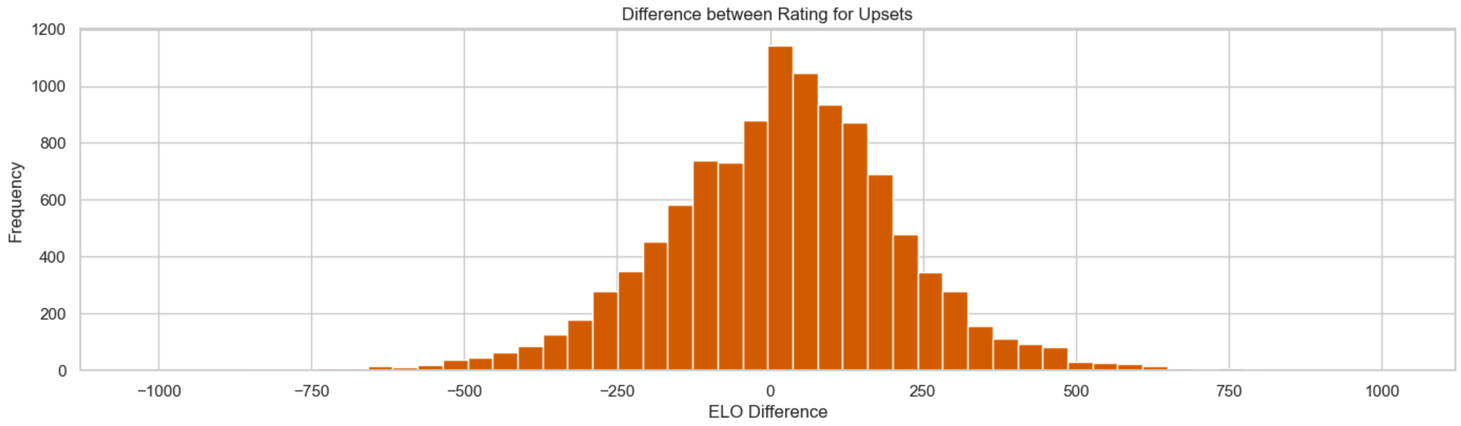
In [111]:
```
df_.query("(WhiteIsRatedHigher and Result == '0-1') or (not WhiteIsRatedHigher and Result == '1-0')").shape[0] / df_.shape[0]

# an upset occurs roughly in 0.188% of these game.
# most of the time the higher rated player does win
```

Out[111]: 0.1868330455428448

In [112]:
```
######
df_upsets_country = df_country[((df_country['Result'] == 0) &
          (df_country['WhiteIsRatedHigher'] == 1))
           |
          ((df_country['Result'] == 1) &
          (df_country['WhiteIsRatedHigher'] == 0)) ].reset_index(
    drop=True).copy()
```

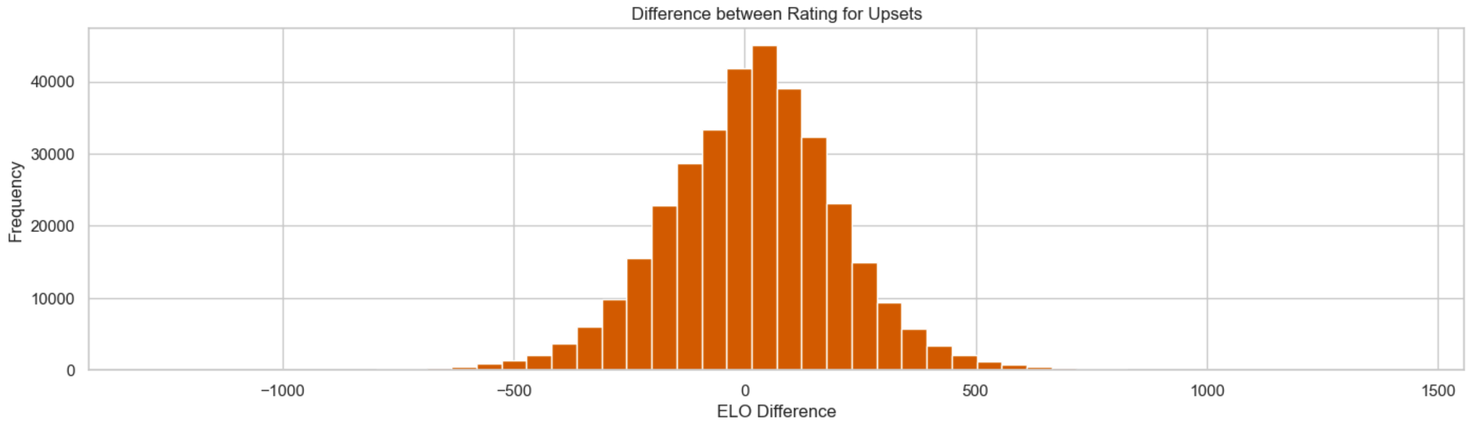
In [113]:
```
df_upsets = (
    df_.query("(WhiteIsRatedHigher and Result == '0-1') or (not WhiteIsRatedHigher and Result == '1-0')").reset_index(
    drop=True).copy())
```

In [114]:
```
######
df_upsets_country = df_country[((df_country['Result'] == 0) &
          (df_country['WhiteIsRatedHigher'] == 1))
           |
          ((df_country['Result'] == 1) &
          (df_country['WhiteIsRatedHigher'] == 0)) ].reset_index(
    drop=True).copy()

ax = df_upsets_country["Elo_diff"].plot(
    kind="hist", bins=50, title="Difference between Rating for Upsets", color = pal[2]
)
ax.set_xlabel("ELO Difference")
plt.show()
```

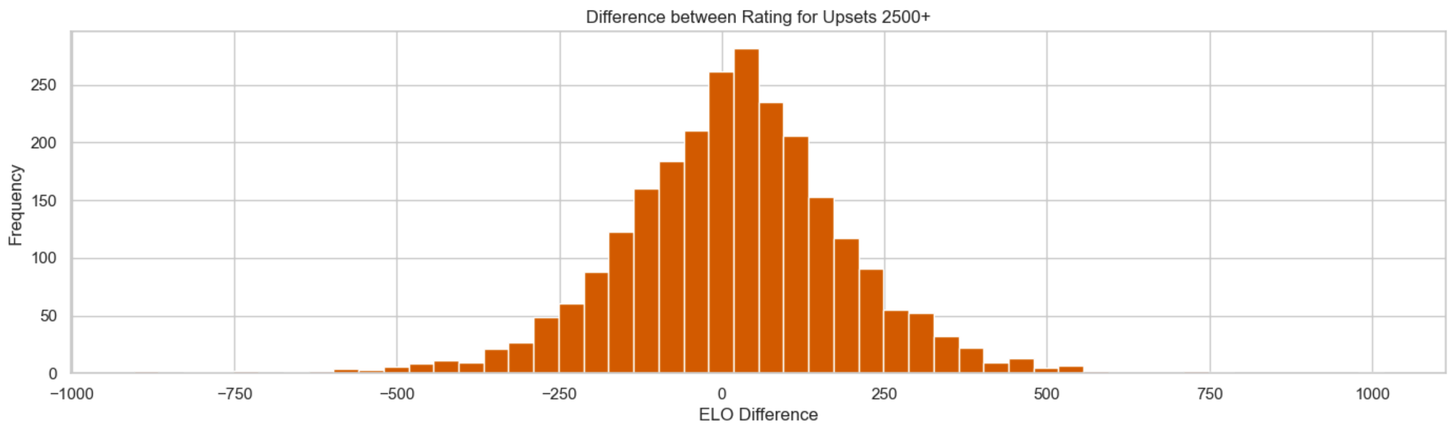


The provided code plots a histogram showing the distribution of Elo rating differences for upsets in chess games. The resulting histogram appears to follow a normal distribution, indicating that upsets occur more frequently around a central Elo rating difference, with fewer occurrences as the difference increases or decreases from the center.

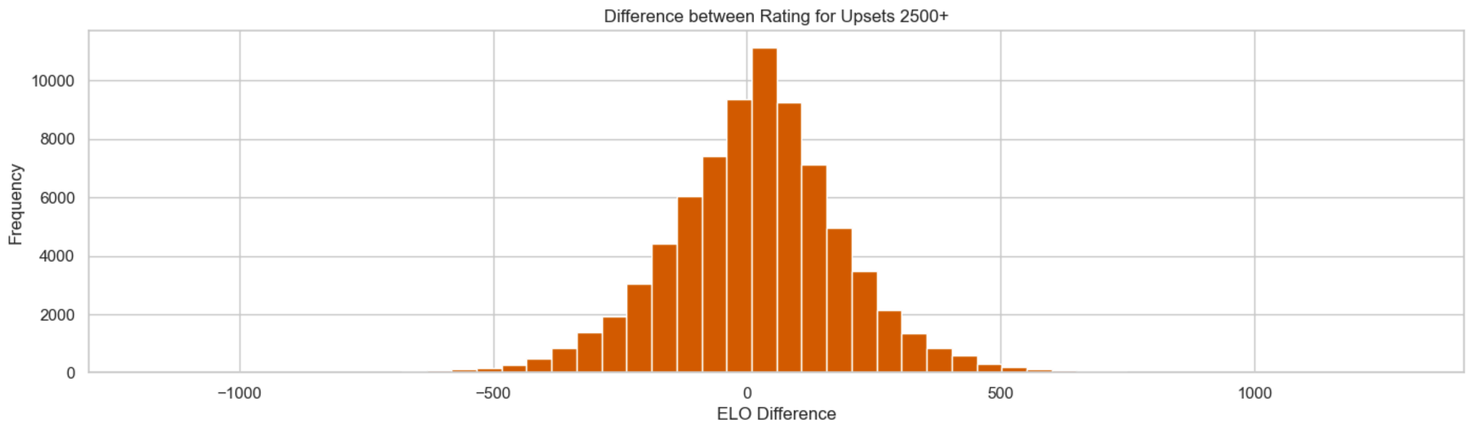
In [115]:
```
df_upsets = (
    df_.query("(WhiteIsRatedHigher and Result == '0-1') or (not WhiteIsRatedHigher and Result == '1-0')").reset_index(
    drop=True).copy())

ax = df_upsets["Elo_diff"].plot(
    kind="hist", bins=50, title="Difference between Rating for Upsets", color = pal[2]
)
ax.set_xlabel("ELO Difference")
plt.show()
```



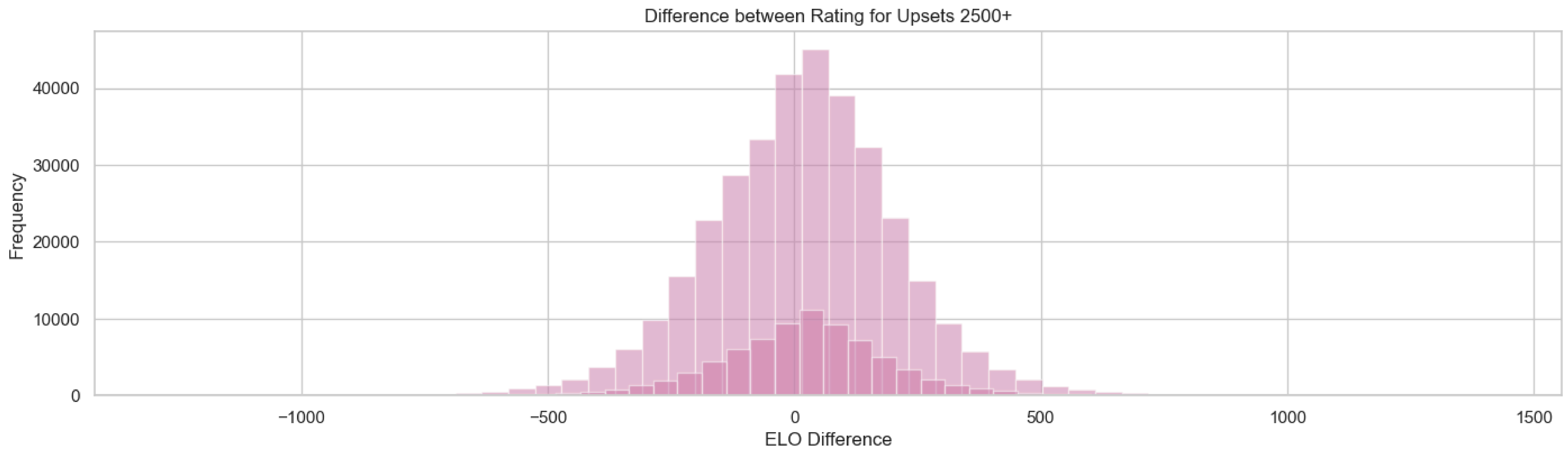
In [116]:
```
######
ax = df_upsets_country.loc[df_upsets_country[["WhiteElo", "BlackElo"]].max(axis=1) > 2500]["Elo_diff"].plot(
    kind="hist", bins=50, title="Difference between Rating for Upsets 2500+", color = pal[2]
)
ax.set_xlabel("ELO Difference")
plt.show()
```



In [117]:
```
ax = df_upsets.loc[df_upsets[["WhiteElo", "BlackElo"]].max(axis=1) > 2500]["Elo_diff"].plot(
    kind="hist", bins=50, title="Difference between Rating for Upsets 2500+", color = pal[2]
)
ax.set_xlabel("ELO Difference")
plt.show()
```



In [118]:
```python
df_upsets = (
    df_.query("(WhiteIsRatedHigher and Result == '0-1') or (not WhiteIsRatedHigher and Result == '1-0')").reset_index(
    drop=True).copy())

ax = df_upsets["Elo_diff"].plot(
    kind="hist", bins=50, title="Difference between Rating for Upsets", color = pal[3], alpha=0.5
)

ax = df_upsets.loc[df_upsets[["WhiteElo", "BlackElo"]].max(axis=1) > 2500]["Elo_diff"].plot(
    kind="hist", bins=50, title="Difference between Rating for Upsets 2500+", color = pal[3], alpha=0.5
)

ax.set_xlabel("ELO Difference")
plt.show()
```



In [119]:
```python
df_upsets["Elo_diff_abs"] = df_upsets["Elo_diff"].abs()
```

In [120]:
```python
df_upsets.loc[df_upsets[["WhiteElo", "BlackElo"]].max(axis=1) > 2500].query(
    "Online == False and (White == 'Nakamura,Hi' or Black == 'Nakamura,Hi')" ).sort_values('Elo_diff_abs',
                                   ascending = False)
```

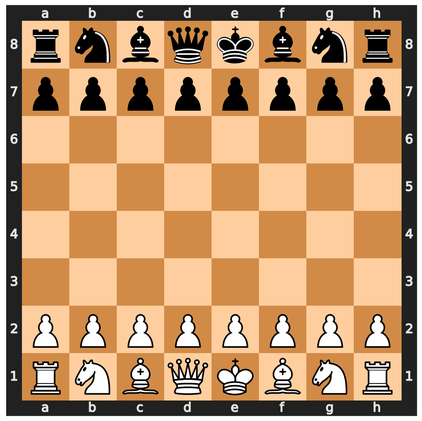
																		attack															
39976	1024	Yudin,S	Nakamura,Hi	2014-06-20	2014.06.19	FIDE World Blitz 2014	1-0	1. e4 e6 2. d4 d5 3. Nd2 Be7 4. Ngf3 Nf6 5. e5...	Dubai UAE	False	16.3	C03	French	4159659.0	2016192.0	2546	2775	Tarrasch	GM	GM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	YudinS	NakamuraHi	2014	2
90018	1144	Nakamura,Hi	Bok,B	2016-10-05	2016.10.01	chess.com IoM Masters	0-1	1. d4 Nf6 2. c4 g6 3. f3 e6 4. e4 c5 5. d5 d6 ...	Douglas ENG	False	5.4	E60	King's Indian defence	2016192.0	1017063.0	2787	2594	NaN	GM	GM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NakamuraHi	BokB	2016	4
95101	1156	Bosiocic,Mari	Nakamura,Hi	2016-12-29	2016.12.29	World Blitz 2016	1-0	1. e4 e5 2. Nf3 Nc6 3. Bb5 Nf6 4. d3 Bc5 5. O-...	Doha QAT	False	8.5	C65	Ruy Lopez	14507927.0	2016192.0	2591	2779	Berlin defence	GM	GM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	BosiocicMari	NakamuraHi	2016	5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
92428	1149	Anand,V	Nakamura,Hi	2016-11-13	2016.11.13	Champions Showdown Rapid	1-0	1. e4 e5 2. Nf3 Nc6 3. Bc4 Bc5 4. O-O Nf6 5. d...	Saint Louis USA	False	1.2	C50	Giuoco Piano	5000017.0	2016192.0	2779	2779	NaN	GM	GM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	AnandV	NakamuraHi	2016	4
161190	1287	So,W	Nakamura,Hi	2019-07-04	2019.06.26	Croatia GCT 2019	1-0	1. e4 e5 2. Nf3 Nc6 3. Bb5 Nf6 4. O-O Nxe4 5. ...	Zagreb CRO	False	8.2	C67	Ruy Lopez	5202213.0	2016192.0	2754	2754	Berlin defence, open variation	GM	GM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	SoW	NakamuraHi	2019	2

In [121]:
```python
# Pull the PGN file from the archive
```

In [122]:
```python
with open('twic1094.pgn', 'r') as f:
    for _ in range(1000000):
        game = chess.pgn.read_game(f)
        if (game.headers['Event'] == '31st ECC Open 2015') and (game.headers['Round'] == '2.4'):
            if game.headers['Black'] == 'Nakamura,Hi':
                break
```

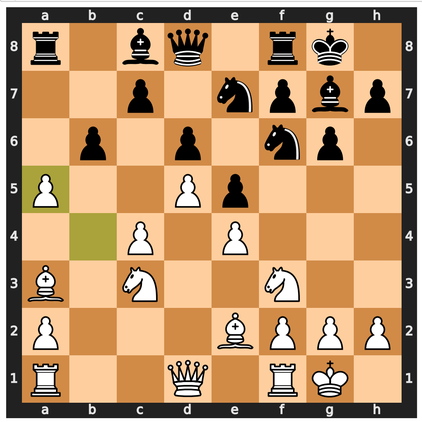
In [123]:
```
board = chess.Board()
board
```

Out[123]:



In [124]:
```
# Create a board and run through the game
board = chess.Board()
for ply, move in enumerate(game.mainline_moves()):
    board.push(move)
    if ply == 20:
        break
board
```

Out[124]:



In [125]:
```
# Check out Stockfish
```

In [126]:
```
import stockfish
from stockfish import Stockfish
STOCKFISH_PATH = ("stockfish\stockfish-windows-x86-64.exe")

stockfish = Stockfish(
    path=STOCKFISH_PATH,
    depth=5,
    parameters={"Threads":32, "Minimum Thinking Time": 5},
)
```

In [127]:
```
# PNGs vs FEN
# PNG has all the moves in the game and metadata
# FEN is just notation for a given board setup
```

In [128]:
```
# Setup stockfish to use the current position
stockfish.set_fen_position(board.fen())
```

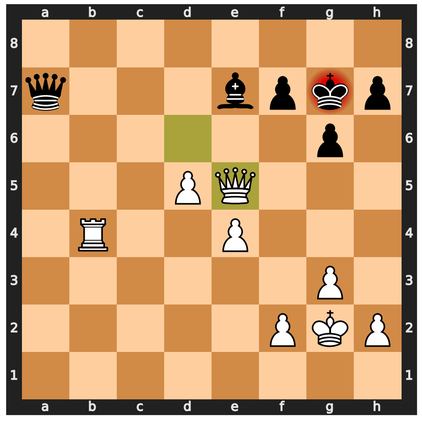
In [129]:
```
stockfish.get_evaluation()
```

Out[129]:
```
{'type': 'cp', 'value': -230}
```

In [130]:
```
## Loop over the game, Evaluate each step

from tqdm.notebook import tqdm

results = []
board = chess.Board()
ply_length = len([x for x in game.mainline_moves()])

for ply, move in tqdm(enumerate(game.mainline_moves()), total=ply_length):
    board.push(move)
    stockfish.set_fen_position(board.fen())
    evaluation = stockfish.get_evaluation()
    results.append(evaluation)
#     if ply == 20:
#         break
board
```

```
100%                                                 75/75 [00:05<00:00, 16.90it/s]
```

Out[130]:



This code performs the following tasks:

It reads a PGN (Portable Game Notation) file named "twic1094.pgn" to extract a specific game. It iterates through the games in the PGN file until it finds a game with specific headers matching the event ("31st ECC Open 2015") and round ("2.4"), and where Hikaru Nakamura is playing as Black. It sets up a chess board and replays the first 20 moves of the selected game. It sets up the Stockfish chess engine, providing the path to the Stockfish executable and setting parameters such as depth and number of threads. It loops over each move in the game, evaluates the position using Stockfish at each step, and stores the evaluation results. Finally, it visualizes the evaluated positions on the chess board. Overall, this code retrieves a specific game from a PGN file, replays the initial moves, evaluates each position using Stockfish, and visualizes the game progression with evaluation scores.

From this code, we can conclude the following:

Game Selection: The code demonstrates how to programmatically select a specific game from a large PGN file based on criteria such as the event name, round number, and player names.

Position Evaluation: By using the Stockfish chess engine, the code evaluates each position in the selected game. The evaluation provides insights into the strength of each player's position at different stages of the game.

Game Progression: By replaying the first 20 moves of the game and evaluating each position, the code allows us to understand how the game unfolds over time and how the evaluation of each position changes with subsequent moves.

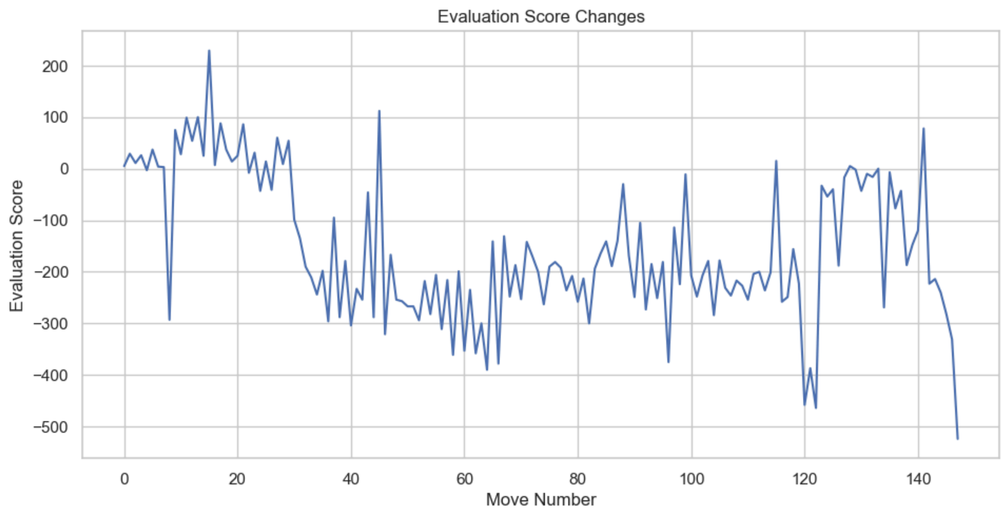
Performance Evaluation: The evaluations provided by Stockfish can give an indication of the quality of moves made by the players and highlight critical moments or turning points in the game.

Overall, this code enables the analysis of specific chess games, providing insights into player strategies, game dynamics, and positional strengths and weaknesses throughout the game.

In [131]:
```
# Function to analyze the first two games from a specific PGN file
def analyze_first_two_games(pgn_file, country):
    # Initialize Stockfish
    STOCKFISH_PATH = "stockfish/stockfish-windows-x86-64.exe"
    stockfish = Stockfish(STOCKFISH_PATH, depth=5, parameters={"Threads": 32, "Minimum Thinking Time": 5})

    with open(pgn_file, 'r') as f:
        # Read the first game
        game1 = chess.pgn.read_game(f)
        analyze_game(game1, country, stockfish)

        # Read the second game
        game2 = chess.pgn.read_game(f)
        analyze_game(game2, country, stockfish)

# Function to analyze a single game
def analyze_game(game, country, stockfish):
    board = chess.Board()
    ply_length = len([x for x in game.mainline_moves()])

    # Iterate over moves in the game
    for ply, move in tqdm(enumerate(game.mainline_moves()), total=ply_length):
        board.push(move)
        stockfish.set_fen_position(board.fen())
        evaluation = stockfish.get_evaluation()
        # Do further analysis with evaluation data, e.g., track trends

# Example usage
pgn_file = "twic1094.pgn"  # Replace with the path to your PGN file
analyze_first_two_games(pgn_file, "Russia")
```

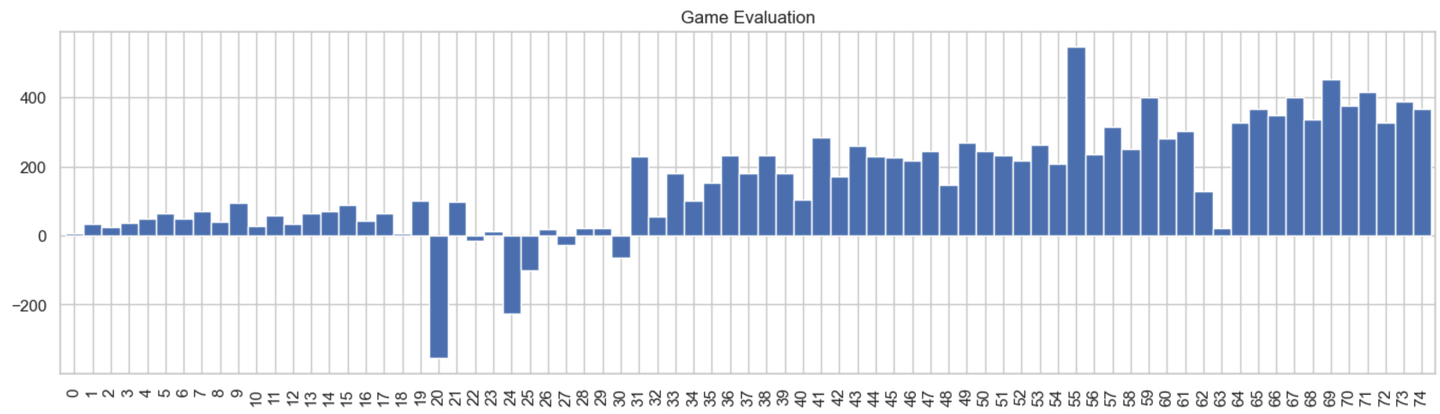
```
100%                                                 148/148 [00:09<00:00, 18.08it/s]

100%                                                 191/191 [00:12<00:00, 16.22it/s]
```

In [132]:
```
# Function to analyze a single game and extract insights
def analyze_game_insights(game, stockfish):
    board = chess.Board()
    ply_length = len([x for x in game.mainline_moves()])
    evaluations = []

    # Iterate over moves in the game
    for ply, move in enumerate(game.mainline_moves()):
        board.push(move)
        stockfish.set_fen_position(board.fen())
        evaluation = stockfish.get_evaluation()
        evaluations.append(evaluation["value"])

    # Plot evaluation score changes over the course of the game
    plt.figure(figsize=(10, 5))
    plt.plot(evaluations)
    plt.title("Evaluation Score Changes")
    plt.xlabel("Move Number")
    plt.ylabel("Evaluation Score")
    plt.grid(True)
    plt.show()

# Example usage
pgn_file = "twic1094.pgn"  # Replace with the path to your PGN file
STOCKFISH_PATH = "stockfish/stockfish-windows-x86-64.exe"
stockfish = Stockfish(STOCKFISH_PATH, depth=5, parameters={"Threads": 32, "Minimum Thinking Time": 5})

with open(pgn_file, 'r') as f:
    game = chess.pgn.read_game(f)
    analyze_game_insights(game, stockfish)
```



In [133]:
```
pd.DataFrame(results)['value'].plot(kind='bar', width=1, title="Game Evaluation")
```

Out[133]: <Axes: title={'center': 'Game Evaluation'}>



The plot visualizes the changes in the evaluation score of the game over the course of each move. The evaluation score represents the advantage of one side (usually measured in centipawns) relative to the other.

Interpreting the plot:

If the evaluation score is positive, it indicates an advantage for White. If the evaluation score is negative, it indicates an advantage for Black. A score close to zero suggests that the game is balanced or roughly equal between the two players. By tracking the evaluation score changes, you can gain insights into which side has the upper hand at different stages of the game. Significant fluctuations in the evaluation may indicate critical moments, such as tactical opportunities, positional advantages, or mistakes made by either player.

In [134]:
```
# Run the evaluation on a different game
```

In [135]:
```
df_upsets.loc[df_upsets['White'].str.contains('Carlsen') ########
            ].query("Year > 2019 and Online == False").sort_values("Elo_diff_abs", ascending=False).head(1)
```

Out[135]:

	twic_number	White	Black	Date	EventDate	Event	Result	mainline_moves	Site	Online	Round	ECO	Opening	WhiteFideId	BlackFideId	WhiteElo	BlackElo	Variation	WhiteTitle	BlackTitle	WhiteTeam	BlackTeam	EventType	FEN	SetUp	Variant	Board	PlyCount	EventCategory	White_concat	Black_concat	Year	Week	Elo_diff	WhiteIsR
339221	1450	Carlsen,M	Praggnanandhaa,R	2022-08-21	2022.08.15	FTX Crypto Cup 2022	0-1	1. e4 c5 2. Nc3 Nc6 3. Bb5 Nd4 4. Nf3 e6 5. O-...	Miami USA	False	7.5	B23	Sicilian	1503014.0	25059530.0	2864	2661	closed, 2...Nc6	GM	GM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	CarlsenM	PraggnanandhaaR	2022	33	-203	

In [136]:
```
df_upsets.loc[df_upsets['White'].str.contains('Carlsen')
            ].query("Year > 2019 and Online == False").sort_values("Elo_diff_abs", ascending=False).head(1)
```

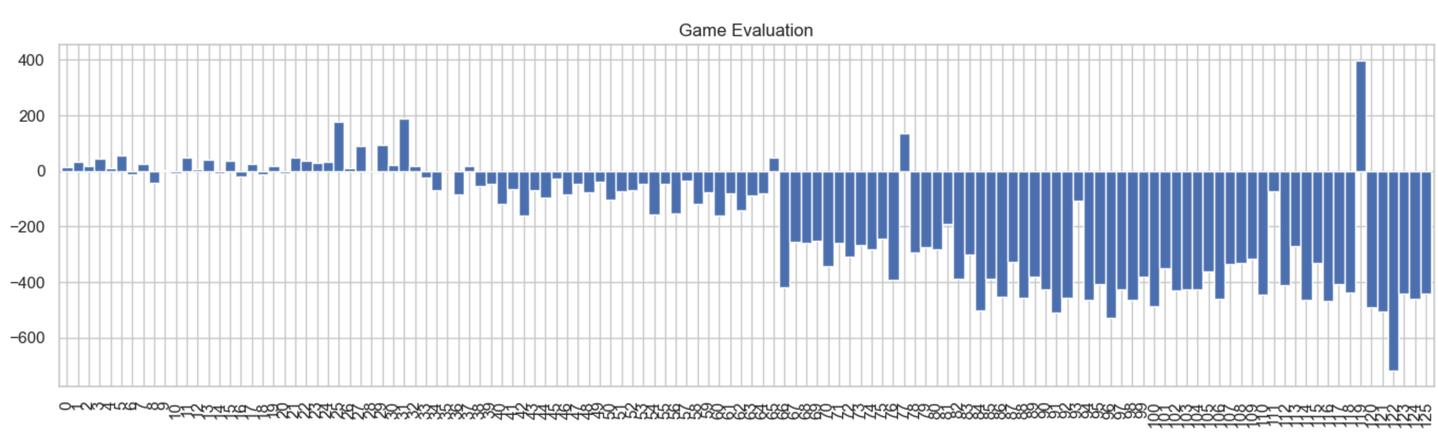
Out[136]:

	twic_number	White	Black	Date	EventDate	Event	Result	mainline_moves	Site	Online	Round	ECO	Opening	WhiteFideId	BlackFideId	WhiteElo	BlackElo	Variation	WhiteTitle	BlackTitle	WhiteTeam	BlackTeam	EventType	FEN	SetUp	Variant	Board	PlyCount	EventCategory	White_concat	Black_concat	Year	Week	Elo_diff	WhiteIsR
339221	1450	Carlsen,M	Praggnanandhaa,R	2022-08-21	2022.08.15	FTX Crypto Cup 2022	0-1	1. e4 c5 2. Nc3 Nc6 3. Bb5 Nd4 4. Nf3 e6 5. O-...	Miami USA	False	7.5	B23	Sicilian	1503014.0	25059530.0	2864	2661	closed, 2...Nc6	GM	GM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	CarlsenM	PraggnanandhaaR	2022	33	-203	

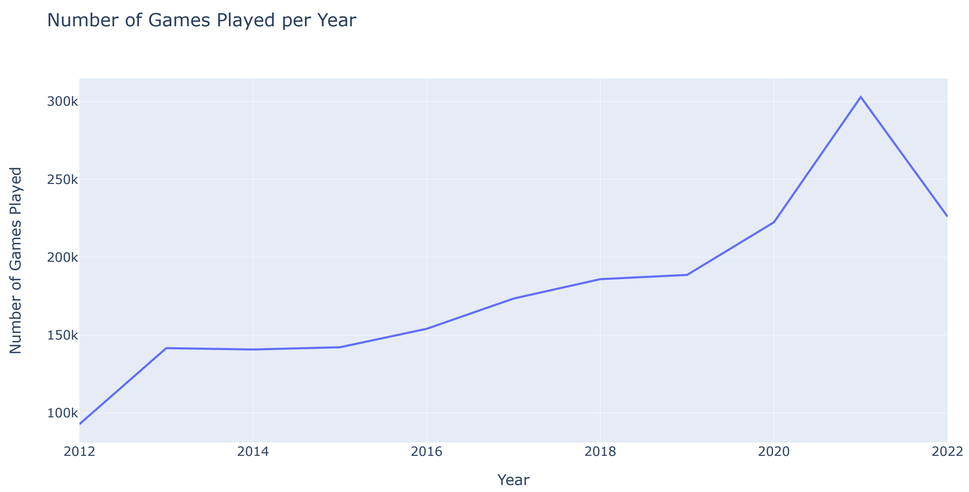
In [137]:
```
with open('twic1450.pgn', 'r') as f:
    for _ in range(1000000):
        game = chess.pgn.read_game(f)
        if (game.headers['Event'] == 'FTX Crypto Cup 2022') and (game.headers['Round'] == '7.5'):
            if game.headers['White'] == 'Carlsen,M':
                break

results = []
board = chess.Board()
ply_length = len([x for x in game.mainline_moves()])

for ply, move in tqdm(enumerate(game.mainline_moves()), total=ply_length):
    board.push(move)
    stockfish.set_fen_position(board.fen())
    evaluation = stockfish.get_evaluation()
    results.append(evaluation)
#     if ply == 20:
#         break
board

pd.DataFrame(results)['value'].plot(kind='bar', width=1, title="Game Evaluation")
```

```
100%                                                 126/126 [00:08<00:00, 16.51it/s]
```

Out[137]:
```
<Axes: title={'center': 'Game Evaluation'}>
```



In [138]:
```
# Aggregate data by year
games_per_year = df1[(df1['Year']<2025) &
                     (df1['Year']>2011)].groupby('Year').size().reset_index(name='Number of Games Played')

# Plot the number of games played per year
fig = px.line(games_per_year, x='Year', y='Number of Games Played', title='Number of Games Played per Year')
fig.update_xaxes(title='Year')
fig.update_yaxes(title='Number of Games Played')
fig.show()
```



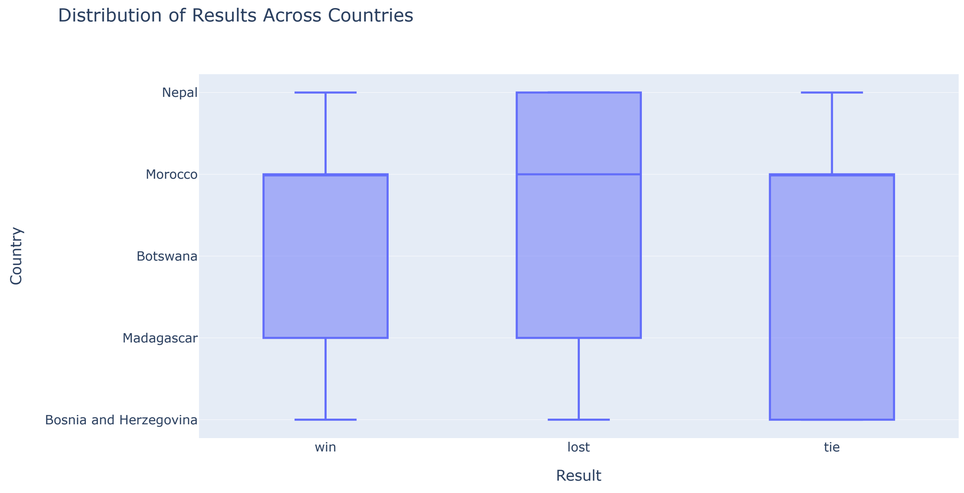
Regarding the observation about the number of games played in 2022

In [139]:
```
top10 = list(df.groupby(['Country', 'Year'])['Result'].mean().reset_index().sort_values('Result',
                                                                                        ascending=False).head(5
                                                                                                              )['Country'])
```

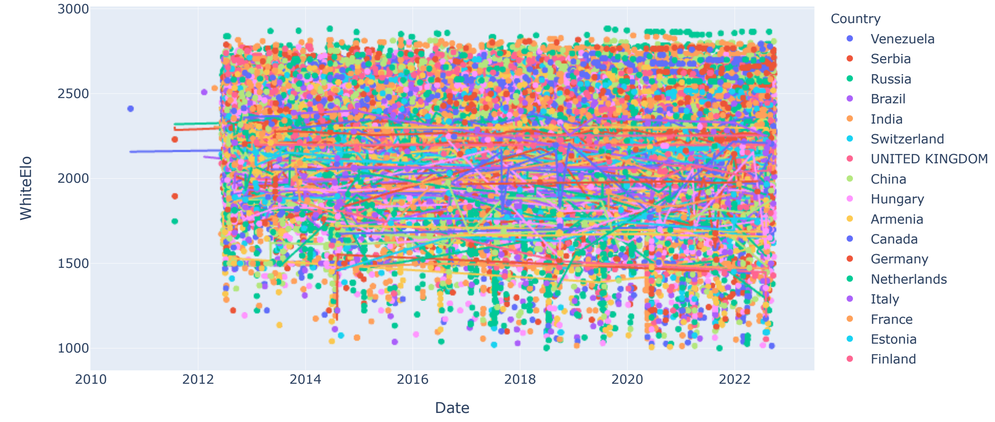
In [140]:
```
top10_df = df[df['Country'].isin(top10)]

# Define the mapping for renaming values
result_mapping = {0: "lost", 1: "win", 2: "tie"}

# Replace the values in the "Result" column
top10_df['Result'] = top10_df['Result'].replace(result_mapping)
```

```
C:\Users\kazom\AppData\Local\Temp\ipykernel_10096\371055636.py:7: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
```

In [141]:
```
# Box plot to visualize the distribution of results across countries
fig = px.box(top10_df, x='Result', y='Country', title='Distribution of Results Across Countries')
fig.show()
```



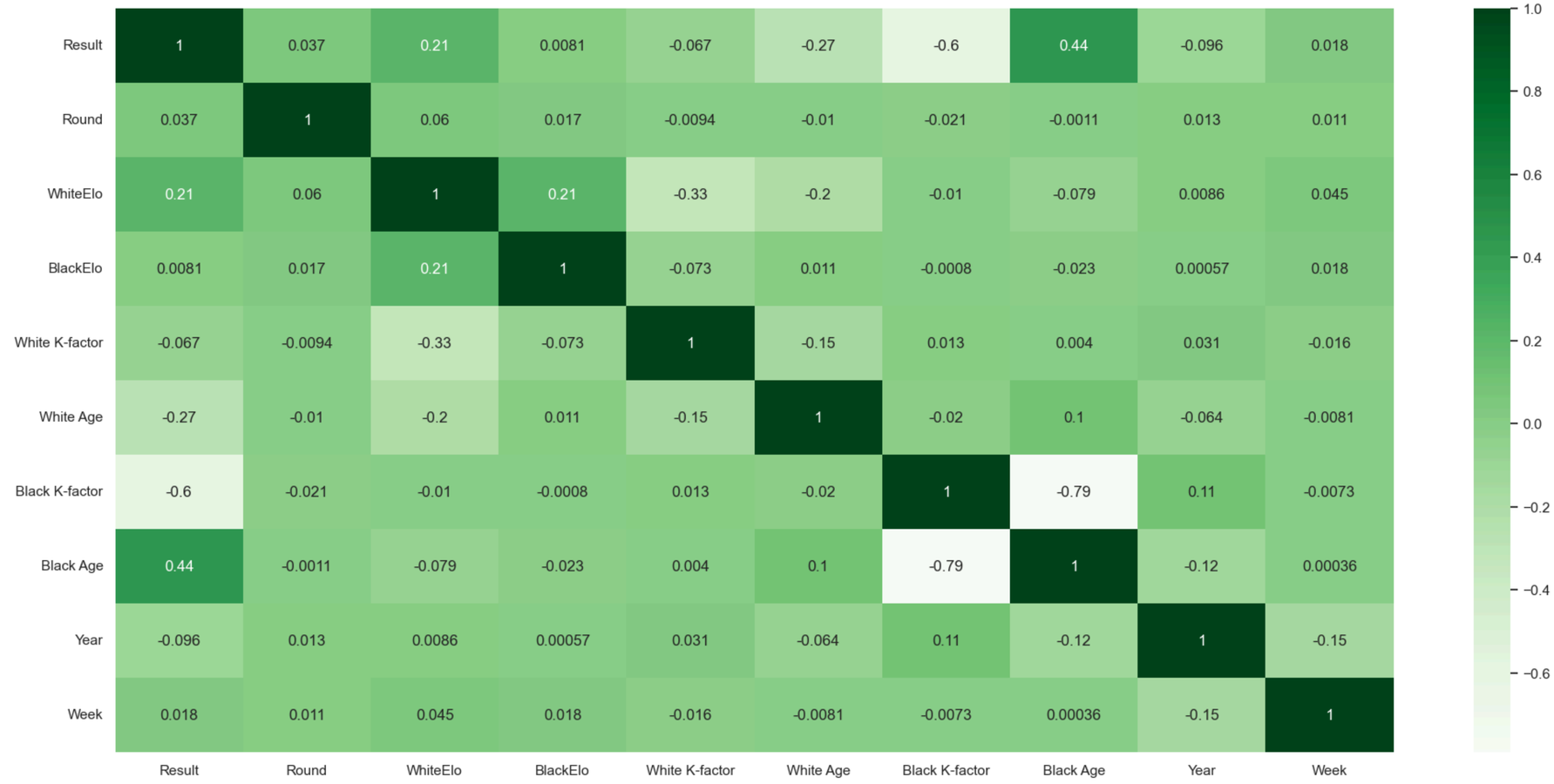
In [142]:
```
# Convert Date column to datetime
df['Date'] = pd.to_datetime(df['Date'])

# Create the dot plot
fig = px.scatter(df, x='Date', y='WhiteElo', color='Country', title='WhiteElo Over Time for Each Country',
                 trendline="ols")

fig.show()
```



For the scatter plot mentioned, it seems you're observing the trend in Elo ratings over time specifically for Tanzania. When filtering the data to focus only on games played by players from Tanzania, you notice that the Elo ratings tend to increase over time. This observation suggests that players from Tanzania, on average, are improving their skills or achieving better results as time progresses. It could indicate various factors such as increased participation in tournaments, access to better training resources, or overall development of the chess community in Tanzania.

Part C

In [143]:
```
plt.figure(figsize=(20,10))
sns.heatmap(df.select_dtypes(include='number').corr(),annot=True,cmap='Greens')
plt.show()
```



In [144]:
```
# Encode categorical columns
encoded_data = pd.get_dummies(df, columns=[
    'WhiteTitle',
    'BlackTitle',])

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df['WhiteTeam'] = le.fit_transform(df['WhiteTeam'])
df['BlackTeam'] = le.fit_transform(df['BlackTeam'])
df['Matched_White'] = le.fit_transform(df['Matched_White'])
df['Matched_Black'] = le.fit_transform(df['Matched_Black'])
df['Country'] = le.fit_transform(df['Country'])
# df['Black Country'] = le.fit_transform(df['Black Country'])


# Drop categorical columns from original data
df.drop(columns=[
    'Event', 'Site', 'ECO', 'Opening', 'Variation', 'WhiteTitle',
    'BlackTitle','EventType','mainline_moves','Online','Round','Week','Year',
    'Date'
], inplace=True)
```

In [145]:
```
encoded_data.drop(columns=[
    'Event', 'mainline_moves', 'Site', 'Online', 'Round', 'ECO', 'Opening',
       'Variation', 'WhiteTeam', 'BlackTeam', 'EventType', 'Matched_White',
       'Matched_Black', 'Country', 'Black Country', 'White Age', 'Black Age',
       'Date', 'Year', 'Week',
    'Result', 'WhiteElo', 'BlackElo', 'White K-factor', 'Black K-factor'
], inplace=True)
```

In [146]:
```
# Attach encoded data to original data
df = pd.concat([df, encoded_data], axis=1)
```

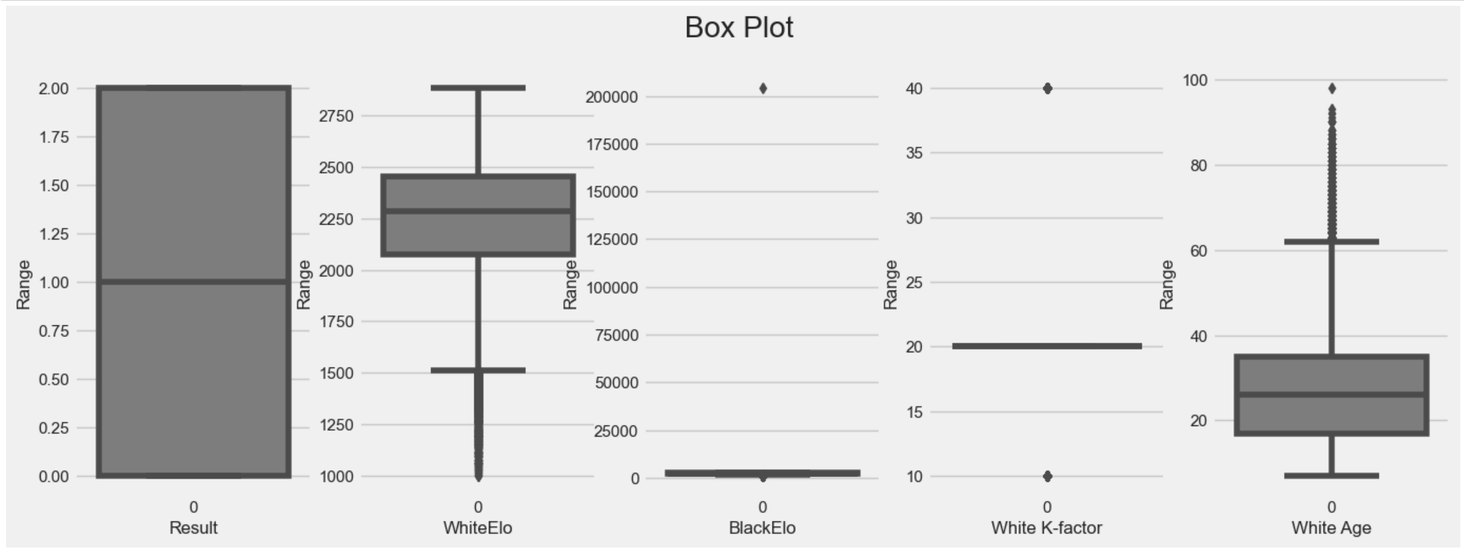
In [147]:
```
# Drop categorical columns from original data
df.drop(columns=[
    'Black Country',
], inplace=True)
df.head()
```

Out[147]:

	Result	WhiteElo	BlackElo	WhiteTeam	BlackTeam	Matched_White	Matched_Black	Country	White K-factor	White Age	Black K-factor	Black Age	WhiteTitle_	WhiteTitle_AGM	WhiteTitle_AIM	WhiteTitle_CM	WhiteTitle_FM	WhiteTitle_GM	WhiteTitle_I++	WhiteTitle_II	WhiteTitle_IM	WhiteTitle_NM	WhiteTitle_None	WhiteTitle_SIM	WhiteTitle_WC	WhiteTitle_WCM	WhiteTitle_WF	WhiteTitle_WFM
19082	1.0	2411.0	2074.0	2294	2300	6026	14845	151	20	17	40	13	False	False	False	False	False	False	False	False	True	False	False	False	False	False	False	False
55200	2.0	2230.0	2242.0	2294	2300	12753	11225	120	20	26	20	38	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
59912	0.0	1748.0	1985.0	2294	2300	13818	12354	112	40	16	20	28	False	False	False	False	False	False	False	False	False	False	True	False	False	False	False	False
43009	1.0	1896.0	1764.0	2294	2300	12555	10181	120	20	17	40	19	False	False	False	False	False	False	False	False	False	False	True	False	False	False	False	False
46492	1.0	2508.0	2379.0	1195	1989	14427	4684	17	20	17	40	13	False	False	False	False	False	False	False	False	True	False	False	False	False	False	False	False

In [148]:
```
# Convert boolean columns to numeric (0 and 1)
bool_columns = df.select_dtypes(include=['bool']).columns
for col in bool_columns:
    df[col] = df[col].astype(int)
```

In [149]:
```
# check the boxplots for the columns where we suspect for outliers
plt.rcParams['figure.figsize'] = (15, 5)
plt.style.use('fivethirtyeight')

# Box plot for Result
plt.subplot(1, 5, 1)
sns.boxplot(df['Result'], color = 'grey')
plt.xlabel('Result', fontsize = 12)
plt.ylabel('Range', fontsize = 12)

# Box plot for WhiteElo
plt.subplot(1, 5, 2)
sns.boxplot(df['WhiteElo'], color = 'grey')
plt.xlabel('WhiteElo', fontsize = 12)
plt.ylabel('Range', fontsize = 12)

# Box plot for BlackElo
plt.subplot(1, 5, 3)
sns.boxplot(df['BlackElo'], color = 'grey')
plt.xlabel('BlackElo', fontsize = 12)
plt.ylabel('Range', fontsize = 12)

# Box plot for White K-factor
plt.subplot(1, 5, 4)
sns.boxplot(df['White K-factor'], color = 'grey')
plt.xlabel('White K-factor', fontsize = 12)
plt.ylabel('Range', fontsize = 12)

# Box plot for White Age
plt.subplot(1, 5, 5)
sns.boxplot(df['White Age'], color = 'grey')
plt.xlabel('White Age', fontsize = 12)
plt.ylabel('Range', fontsize = 12)

plt.suptitle('Box Plot', fontsize = 20)
plt.show()
```



We observed that both WhiteElo and BlackElo ratings have some outliers. These outliers are particularly noticeable for Elo ratings under 1600, which may not be representative of the typical competitive chess player in our dataset. To ensure a more accurate and robust analysis, we will exclude these values by dropping all Elo ratings below 1600. This step will help us focus on a more consistent range of skill levels and reduce the impact of outliers on our analysis.

In [172]:
```
# Filter out rows where WhiteElo or BlackElo is less than 1600 and above 3000
df = df[(df['WhiteElo'] >= 1600) & (df['BlackElo'] >= 1600)]
df = df[(df['WhiteElo'] <= 3000) & (df['BlackElo'] <= 3000)]
```

Label Encoding:

Label Encoding converts each categorical value into a numerical label. It assigns unique integers to each category, often based on the alphabetical order or the order of appearance in the dataset. For example, if we have categories like 'red', 'green', and 'blue', Label Encoding might assign them 0, 1, and 2, respectively.

One-Hot Encoding (pd.get_dummies()):

One-Hot Encoding creates dummy variables for each category of a categorical feature. It converts categorical variables into a binary matrix where each category becomes a separate column with either 0 or 1 indicating its presence or absence. For example, if we have a categorical feature 'color' with values 'red', 'green', and 'blue', One-Hot Encoding would create three columns: 'color_red', 'color_green', and 'color_blue', and set the corresponding column to 1 for each instance based on its color.

In [179]:
```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Splitting the data into features (X) and target (y)
X = df.drop(columns=['Result'])
y = df['Result']

# One-hot encode categorical variables
X = pd.get_dummies(X)

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and train the Random Forest classifier
clf = RandomForestClassifier(random_state=42)
clf.fit(X_train, y_train)

# Make predictions
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.9652310246241815
```

For a randomforest model we recieved ~96% acc.

In [180]:
```
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Assume you have your data loaded into X_train, y_train, X_test, y_test

# Initialize base classifiers
rf_classifier = RandomForestClassifier()
gb_classifier = GradientBoostingClassifier()
lr_classifier = LogisticRegression()

# Create a VotingClassifier with soft voting
voting_classifier = VotingClassifier(
    estimators=[('rf', rf_classifier), ('gb', gb_classifier), ('lr', lr_classifier)],
    voting='soft'  # Use 'soft' voting for probabilities averaging
)

# Train the ensemble model
voting_classifier.fit(X_train, y_train)

# Make predictions
y_pred = voting_classifier.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Ensemble Accuracy:", accuracy)
```

```
C:\Users\kazom\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:458: ConvergenceWarning:

lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)


Ensemble Accuracy: 0.9679055611915521
```

Ensemble voting is a technique used in machine learning where predictions from multiple individual models are combined to make a final prediction. The idea is that by aggregating the predictions of multiple models, the ensemble model can often achieve better performance than any individual model.

In the provided code, we are using ensemble voting to combine predictions from three different classifiers: RandomForestClassifier, GradientBoostingClassifier, and LogisticRegression. Here's what we're doing step by step:

Initialization of Base Classifiers:

We initialize three different classifiers: RandomForestClassifier, GradientBoostingClassifier, and LogisticRegression. These are the base models that will form the ensemble.

Creation of VotingClassifier:

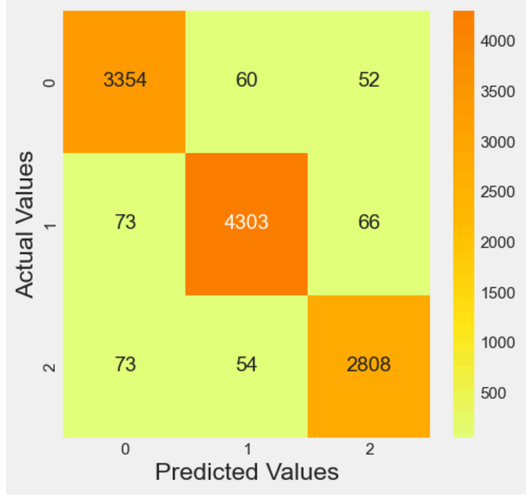
We create a VotingClassifier using sklearn.ensemble.VotingClassifier. The estimators parameter takes a list of tuples, where each tuple consists of a name for the classifier and the classifier object itself. The voting parameter is set to 'soft', indicating that we are using soft voting, which averages the predicted probabilities from all classifiers to make the final prediction.

Training the Ensemble Model:

We train the VotingClassifier on the training data (X_train, y_train) using the fit() method.

In [181]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn.metrics import confusion_matrix, classification_report

bag_tree = BaggingClassifier(
            estimator=RandomForestClassifier(),
            n_estimators = 10,
            max_samples = 0.8,
            oob_score = True,
            random_state=0
)
bag_tree.fit(X_train, y_train)
y_pred = bag_tree.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
plt.rcParams['figure.figsize'] = (5, 5)
sns.heatmap(cm, annot = True, cmap = 'Wistia', fmt = '.8g')
plt.xlabel('Predicted Values')
plt.ylabel('Actual Values')
plt.show()

print("Training Accuracy :", bag_tree.score(X_train, y_train))
print("Testing Accuracy :", bag_tree.score(X_test, y_test))
cr = classification_report(y_test, y_pred)
print(cr)
```

```
C:\Users\kazom\anaconda3\lib\site-packages\sklearn\ensemble\_bagging.py:789: UserWarning:

Some inputs do not have OOB scores. This probably means too few estimators were used to compute any reliable oob estimates.

C:\Users\kazom\anaconda3\lib\site-packages\sklearn\ensemble\_bagging.py:795: RuntimeWarning:

invalid value encountered in divide
```




```
Training Accuracy : 0.9948814903624458
Testing Accuracy : 0.9651387992253067
              precision    recall  f1-score   support

         0.0       0.96      0.97      0.96      3466
         1.0       0.97      0.97      0.97      4442
         2.0       0.96      0.96      0.96      2935

    accuracy                           0.97     10843
   macro avg       0.96      0.96      0.96     10843
weighted avg       0.97      0.97      0.97     10843
```

A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it. Each base classifier is trained in parallel with a training set which is generated by randomly drawing, with replacement, N examples(or data) from the original training dataset – where N is the size of the original training set. Training set for each of the base classifiers is independent of each other. Many of the original data may be repeated in the resulting training set while others may be left out.

Bagging reduces overfitting (variance) by averaging or voting, however, this leads to an increase in bias, which is compensated by the reduction in variance though.

How Bagging works on training dataset ? How bagging works on an imaginary training dataset is shown below. Since Bagging resamples the original training dataset with replacement, some instance(or data) may be present multiple times while others are left out.

Original training dataset: *1, 2, 3, 4, 5, 6, 7, 8, 9, 10*

Resampled training set 1: *2, 3, 3, 5, 6, 1, 8, 10, 9, 1*
Resampled training set 2: *1, 1, 5, 6, 3, 8, 9, 10, 2, 7*
Resampled training set 3: *1, 5, 8, 9, 2, 10, 9, 7, 5, 4*

In [182]:
```
from sklearn.ensemble import AdaBoostClassifier

#Ada Model
AdaModel = AdaBoostClassifier(n_estimators = 10, learning_rate = 0.06)

AdaModel.fit(X_train, y_train)

y_pred = AdaModel.predict(X_test)

print("Training Accuracy :", AdaModel.score(X_train, y_train))
print("Testing Accuracy :", AdaModel.score(X_test, y_test))
```

```
Training Accuracy : 0.8917273817209259
Testing Accuracy : 0.8934796642995481
```

AdaBoost is short for Adaptive Boosting is a way to combine many weak learners to make a strong learner. Weak learner is a learning algorithm that performs consistently better than random guessing. AdaBoost combines many such learners to form a hypothesis that gives arbitrarily high accuracy. The AdaBoost algorithm is adaptive, which means that it learns from its mistakes.

How AdaBoost works for binary classification tasks:

Given a training set, AdaBoost learns a simple and weak classifier in its first iteration. It notes the examples that this weak classifier incorrectly classifies and increases their weight for the next iteration. In the next iteration, it agains learns a new weak classifier that considers the increased weights of the previously incorrectly classified examples. Now we have 2 weak classifiers.

The algorithm again notes the incorrectly classified examples by the second classifier and increases their weights for the next iteration and so on till it gets multiple classifiers.

A weighted sum of all these classifiers gives the final boosted classifier. The weights are a function of the accuracy of each classifier.

Modifying weights: AdaBoost requires a learning algorithm that can account for weighted input examples i.e. the loss function should give more importance to examples with higher weights.

In case we have an algorithm that does not support weighted inputs, we can use sampling. At every iteration, a classifier is learnt on a sampled training data. The training data is sampled according to a probability distribution that mimics the weight distribution. Samples with higher weight are repeated in the sampled set. If the weights are normalized i.e if they sum to 1, the weight distribution can be used as the probability distribution for sampling.

Pseudo Algorithm

Consider the input training set -

$$X = \{x_0, x_1, x_2, ..., x_n\}$$

and corresponding labels which are either -1 or +1

$$Y = \{y_0, y_1, y_2, ..., y_n\}$$

For t in 1 to T

1. Set weights of all training examples as -

$$w_i^t = \frac{1}{n}, \text{ for all i}$$

2. Train a weak classifier c_t that minimizes

$$\epsilon = \sum_{i=0}^{n} w_i^t \; 1\{y_i \neq c\}, \text{Where } \; 1\{.\} \text{ is the indicator function the outputs 1 if expression is true else 0}$$

3. Set

$$\alpha_t = \frac{1}{2}\log\frac{1-\epsilon_t}{\epsilon_t}$$

4. Update weights of training samples as

$$w_i^{t+1} = w_i^t e^{-\alpha_t y_i c_t(x_i)}$$

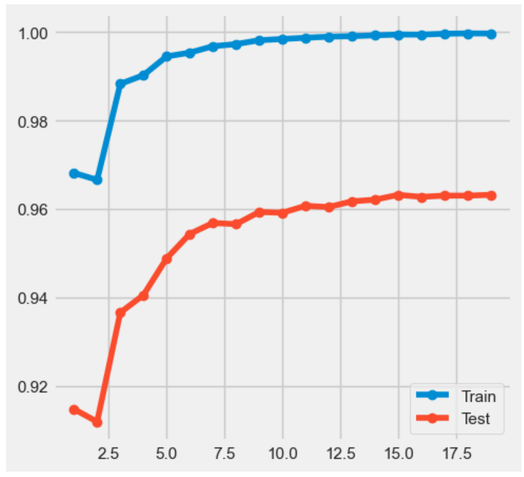
The final output classifier, which classifies a new sample x, is given by

$$C(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t c_t(x)\right)$$

In [183]:
```
# evaluate a decision tree for each depth
train_scores = []
test_scores = []
values = [i for i in range(1, 20)]
for i in values:
    # configure the model
    model = RandomForestClassifier(n_estimators = i,
    criterion='gini',
    max_depth=None,
    max_features='sqrt',
    max_leaf_nodes=None,
    min_impurity_decrease=0.0,
    bootstrap=True,
    max_samples=None,
    oob_score = True,
    random_state=0)
    # fit model on the training dataset
    model.fit(X_train, y_train)
    # evaluate on the train dataset
    train_yhat = model.predict(X_train)
    train_acc = accuracy_score(y_train, train_yhat)
    train_scores.append(train_acc)
    # evaluate on the test dataset
    test_yhat = model.predict(X_test)
    test_acc = accuracy_score(y_test, test_yhat)
    test_scores.append(test_acc)
    # summarize progress
    print('>%d, train: %.3f, test: %.3f' % (i, train_acc, test_acc))
```

```
C:\Users\kazom\anaconda3\lib\site-packages\sklearn\ensemble\_forest.py:583: UserWarning:

Some inputs do not have OOB scores. This probably means too few trees were used to compute any reliable OOB estimates.


>1, train: 0.968, test: 0.915
C:\Users\kazom\anaconda3\lib\site-packages\sklearn\ensemble\_forest.py:583: UserWarning:

Some inputs do not have OOB scores. This probably means too few trees were used to compute any reliable OOB estimates.


>2, train: 0.967, test: 0.912
C:\Users\kazom\anaconda3\lib\site-packages\sklearn\ensemble\_forest.py:583: UserWarning:

Some inputs do not have OOB scores. This probably means too few trees were used to compute any reliable OOB estimates.


>3, train: 0.988, test: 0.937
```

In [184]:
```
import matplotlib.pyplot as pyplot

# Plot
pyplot.plot(values, train_scores, '-o', label='Train')
pyplot.plot(values, test_scores, '-o', label='Test')
pyplot.legend()
pyplot.show()
```



In this case, we can see a trend of increasing accuracy on the training dataset with the tree depth to a point around a depth of 12-15 levels where the tree fits the training dataset perfectly.

We can also see that the accuracy on the test set improves with tree depth until a depth of about 12 to 15 levels, after which accuracy begins to get worse/slightly better with each increase in tree depth.

This is exactly what we would expect to see in a pattern of overfitting.

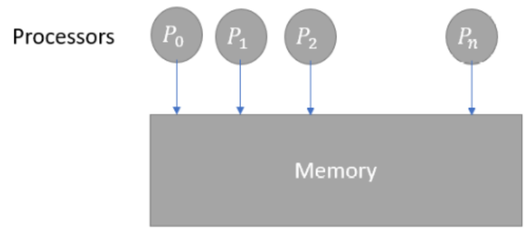
We would choose a tree depth of 12 to 15 before the model begins to overfit the training dataset.

Deep Learning Using Parallel Processing

In our pursuit of finding the best solution for the case, we've explored multiple architectures of deep learning models. Each architecture represents a different configuration of layers, neurons, activation functions, and other parameters that define the structure and behavior of the neural network.

To systematically evaluate the performance of these architectures, we conducted a series of experiments where each architecture was trained and tested multiple times. Specifically, we ran each architecture 16 times, each time with a different combination of hyperparameters. Hyperparameters are configuration settings that control the learning process of the model, such as learning rate, batch size, number of epochs, and regularization strength.

Hyperparameter optimization is the problem of optimizing a loss function over a graph-structured configuration space. Hyperparameters may have a significant impact on model accuracy and their optimization is highly computation-demanding process due to the multiple training runs required. It is possible to run such optimization in parallel using independent models. Embarrassingly parallel programming is a form of parallel algorithms requiring almost no communication between the processes where each process can perform their own computations without any need for communication with the others. Today, most computers have a multicore architecture using the same shared memory, where each core has a private memory. The shared memory architecture is illustrated in the Figure below:



Embarrassingly parallel algorithms can easily improve execution time of a single task while implementing them correctly. Monte Carlo Simulations and Mandelbrot Sets (also known as Fractals) are examples for embarrassingly parallel algorithms. The ideal case of embarrassingly parallel algorithms. is that all subproblems/tasks are defined before the computations begin. In that case, all the subsolutions are stored in independent memory locations (arrays, variables, etc.). Thus, the computation of the subsolutions is completely independent. We used Parallel Computation and Task Scheduling with Dask, a Python library extension which enables parallel out-of-core computation. Dask has many built-in functions which enables dynamic and memory aware task scheduling to achieve parallelism. The following list represents the values of the different hyperparameters of the independent neural network function, f(learning_list_i,dense_list_i,drop_list_i,patience_list_i), which are ran in parallel on different cores:

When training a machine learning model, one of the main things that you want to avoid would be overfitting.

To find out if their model is overfitting, data scientists use a technique called cross-validation, where they split their data into two parts - the training set, and the validation set. The training set is used to train the model, while the validation set is only used to evaluate the model's performance.

Metrics on the training set let you see how your model is progressing in terms of its training, but it's metrics on the validation set that let you get a measure of the quality of your model - how well it's able to make new predictions based on data it hasn't seen before.

With this in mind, loss and acc are measures of loss and accuracy on the training set, while val_loss and val_acc are measures of loss and accuracy on the validation set.

If you don't use a batch size when training a deep learning, the model will be trained on the entire dataset in one pass. This can have a few implications:

Memory usage: If the dataset is large, loading the entire dataset into memory can cause an out of memory error.

Training time: Training on the entire dataset in one pass can take a long time, especially if the dataset is large.

Overfitting: Without using a batch size, the model may overfit to the training data. Because the model will see the same data multiple times during one epoch, it can memorize the training data and perform poorly on unseen data.

Stochasticity: One of the key advantage of the neural network is that it is stochastic and this helps to escape local minima. By not using a batch size, you lose the stochasticity, which can be beneficial for optimization.

Gradient variability: Without using a batch size, the gradients will be calculated using the entire dataset, which can cause the gradients to be very large or very small. This can make it difficult for the optimizer to adjust the model's parameters effectively.

However, while using a batch size when training a model on a low size dataset is generally recommended, there may be some cases where not using a batch size can have advantages:

Simplicity: Not using a batch size can simplify the training process, as it eliminates the need to handle batching logic.

Full utilization of data: When the dataset is small, it may be beneficial to use the entire dataset in each iteration to make the most use of the limited data available.

Better for online learning: In online learning, the model is trained on one data point at a time and not using a batch size would be beneficial in such case.

Better for small sized dataset: If the dataset is too small, using a batch size can cause the model to not converge or to converge to a poor solution.

In [185]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import dask
import dask.delayed
import plotly.graph_objects as go
from torch.utils.data import DataLoader, TensorDataset
import dask.delayed as delayed
from dask.distributed import Client, LocalCluster
```

In [186]:
```
daskD = Client(n_workers=6,threads_per_worker=1) # here we set the number of workers for the parallel computations
daskD
```

```
C:\Users\kazom\anaconda3\lib\site-packages\distributed\node.py:179: UserWarning:

Port 8787 is already in use.
Perhaps you already have a cluster running?
Hosting the HTTP server on port 65266 instead
```

Out[186]:

Client

Client-9831f8dd-16cd-11ef-a770-0c96e632fcf8

Connection method: Cluster object

Cluster type: distributed.LocalCluster

Dashboard: http://127.0.0.1:65266/status (http://127.0.0.1:65266/status)

Cluster Info

In [187]:
```
# Drop any non-numeric columns that aren't needed for modeling
df = df.select_dtypes(include=[np.number])

# Convert dataframe to numpy arrays
x_data = df.drop(columns=['Result']).values
y_data = df['Result'].values

# Split the dataset into training and validation sets
# You might need to adjust this based on your actual dataset
split_ratio = 0.8
split_index = int(len(x_data) * split_ratio)

x_train, x_val = x_data[:split_index], x_data[split_index:]
y_train, y_val = y_data[:split_index], y_data[split_index:]

# Convert data to PyTorch tensors
x_train = torch.from_numpy(x_train).float()
x_val = torch.from_numpy(x_val).float()
y_train = torch.from_numpy(y_train).long()  # For CrossEntropyLoss, labels should be long
y_val = torch.from_numpy(y_val).long()
```

In [188]:
```
# Define parameters for grid search
dense_list = [64, 128]
drop_list = [0.2, 0.5]
patience_list = [15, 25]
learning_list = [0.001, 0.01]
```

In [189]:
```
def plot_learning_curve(train_losses, val_losses, train_accs, val_accs):
    plt.figure(figsize=(10, 6))

    # Plot training and validation loss
    plt.plot(train_losses, label='Training Loss')
    plt.plot(val_losses, label='Validation Loss')

    # Plot training and validation accuracy
    plt.plot(train_accs, label='Training Accuracy')
    plt.plot(val_accs, label='Validation Accuracy')

    plt.xlabel('Epoch')
    plt.title('Training and Validation Loss & Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()
```

In [190]:
```
# Define your neural network architecture
class ChessNet1(nn.Module):
    def __init__(self, input_size, dense_list, drop_list, l1_reg=1e-6, l2_reg=1e-6):
        super(ChessNet1, self).__init__()
        self.fc1 = nn.Linear(input_size, dense_list)
        self.fc2 = nn.Linear(dense_list, dense_list)
        self.fc3 = nn.Linear(dense_list, 3)
        self.dropout1 = nn.Dropout(drop_list)
        self.dropout2 = nn.Dropout(drop_list)
        self.bn1 = nn.BatchNorm1d(dense_list)
        self.bn2 = nn.BatchNorm1d(dense_list)

    def forward(self, x):
        x = F.relu(self.bn1(self.fc1(x)))
        x = self.dropout1(x)
        x = F.relu(self.bn2(self.fc2(x)))
        x = self.dropout2(x)
        x = self.fc3(x)
        return x
```

In [191]:

```
@dask.delayed
def train_with_early_stopping1(input_size, dense_list, drop_list, patience, learning_rate, criterion, l1_reg=1e-6, l2_reg=1e-6):
    # Instantiate your model
    model = ChessNet1(input_size=input_size, dense_list=dense_list, drop_list=drop_list)

    # Send model to device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    # Define the optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    # Define your data loaders (assuming you are using PyTorch DataLoader)
    train_dataset = torch.utils.data.TensorDataset(x_train, y_train)
    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)

    best_model_state = None
    best_val_loss = float('inf')
    current_patience = 0

    train_losses = []
    val_losses = []
    train_accs = []
    val_accs = []

    for epoch in range(100):  # Fixed 100 epochs
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = correct / total
        train_losses.append(epoch_loss)
        train_accs.append(epoch_acc)

        # Validation
        model.eval()
        val_outputs = model(x_val.to(device))
        val_loss = criterion(val_outputs, y_val.to(device))
        val_losses.append(val_loss.item())

        _, predicted = torch.max(val_outputs, 1)
        val_acc = (predicted == y_val).sum().item() / len(y_val)
        val_accs.append(val_acc)

        print(f"Epoch [{epoch+1}/20], Train Loss: {epoch_loss:.4f}, Train Acc: {epoch_acc:.4f}, Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")

        # Check for early stopping
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_state = model.state_dict()
            current_patience = 0
        else:
            current_patience += 1
            if current_patience == patience:
                print("Early stopping...")
                break

    # Load the best model state
    model.load_state_dict(best_model_state)

    return train_losses, val_losses, train_accs, val_accs
```

In [192]:

```
# Run models in parallel for Net1
daskList1 = []
for dense in dense_list:
    for drop in drop_list:
        for patience in patience_list:
            for learning in learning_list:
                s = train_with_early_stopping1(x_train.shape[1], dense, drop, patience, learning,
                                               criterion = nn.CrossEntropyLoss(), l1_reg=0.001, l2_reg=0.001)
                daskList1.append(s)
```

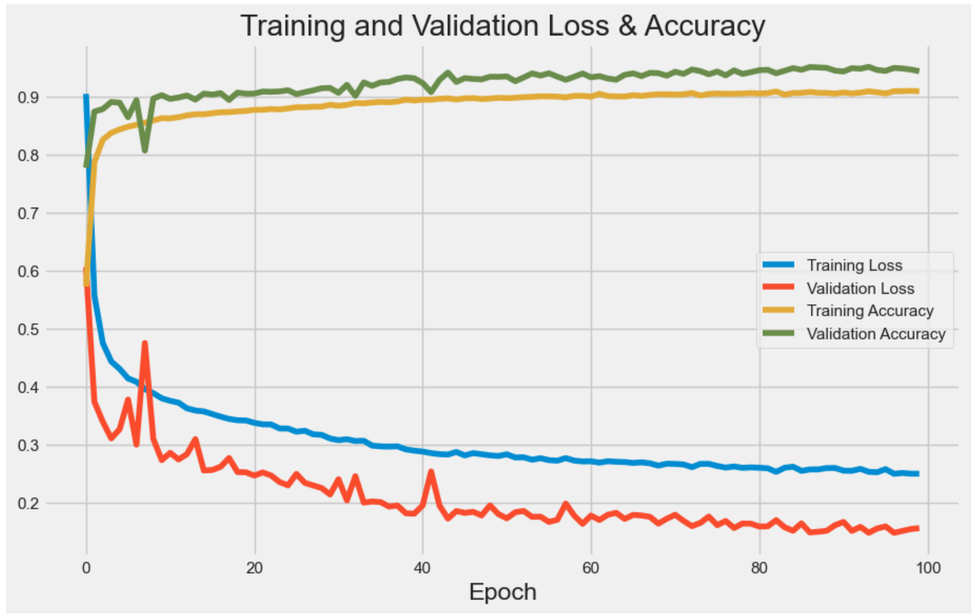
In [193]:

```
%%time
# Execute models using Dask for Net1
sa1 = dask.compute(*daskList1)
```

```
CPU times: total: 9min 18s
Wall time: 39min 19s
```

In [194]:
```
best_model = []
for i in range(len(sa1)):
    best_model.append(sa1[i][3][-1])

best = np.argmax(best_model)

print("Best Model's Validation Accuracy", sa1[best][3][-1])
plot_learning_curve(sa1[best][0], sa1[best][1], sa1[best][2], sa1[best][3])
```

```
Best Model's Validation Accuracy 0.9438347320852163
```



In [195]:
```
class ChessNet2(nn.Module):
    def __init__(self, input_size, dense_list, drop_list):
        super(ChessNet2, self).__init__()
        self.fc1 = nn.Linear(input_size, dense_list)
        self.fc2 = nn.Linear(dense_list, dense_list)
        self.fc3 = nn.Linear(dense_list, 3)
        self.dropout1 = nn.Dropout(drop_list)
        self.dropout2 = nn.Dropout(drop_list)
        self.bn1 = nn.BatchNorm1d(dense_list)
        self.bn2 = nn.BatchNorm1d(dense_list)

    def forward(self, x):
        x = F.relu(self.bn1(self.fc1(x)))
        x = self.dropout1(x)
        x = F.relu(self.bn2(self.fc2(x)))
        x = self.dropout2(x)
        x = self.fc3(x)
        return x
```

In [196]:
```
@dask.delayed
def train_with_early_stopping2(input_size, dense_list, drop_list, patience, learning_rate, criterion, l1_reg=1e-6, l2_reg=1e-6):
    # Instantiate your model
    model = ChessNet2(input_size=input_size, dense_list=dense_list, drop_list=drop_list)

    # Send model to device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    # Define the optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    # Define your data loaders (assuming you are using PyTorch DataLoader)
    train_dataset = torch.utils.data.TensorDataset(x_train, y_train)
    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)

    best_model_state = None
    best_val_loss = float('inf')
    current_patience = 0

    train_losses = []
    val_losses = []
    train_accs = []
    val_accs = []

    for epoch in range(100):  # Fixed 100 epochs
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = correct / total
        train_losses.append(epoch_loss)
        train_accs.append(epoch_acc)

        # Validation
        model.eval()
        val_outputs = model(x_val.to(device))
        val_loss = criterion(val_outputs, y_val.to(device))
        val_losses.append(val_loss.item())

        _, predicted = torch.max(val_outputs, 1)
        val_acc = (predicted == y_val).sum().item() / len(y_val)
        val_accs.append(val_acc)

        print(f"Epoch [{epoch+1}/20], Train Loss: {epoch_loss:.4f}, Train Acc: {epoch_acc:.4f}, Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")

        # Check for early stopping
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_state = model.state_dict()
            current_patience = 0
        else:
            current_patience += 1
            if current_patience == patience:
                print("Early stopping...")
                break

    # Load the best model state
    model.load_state_dict(best_model_state)

    return train_losses, val_losses, train_accs, val_accs
```

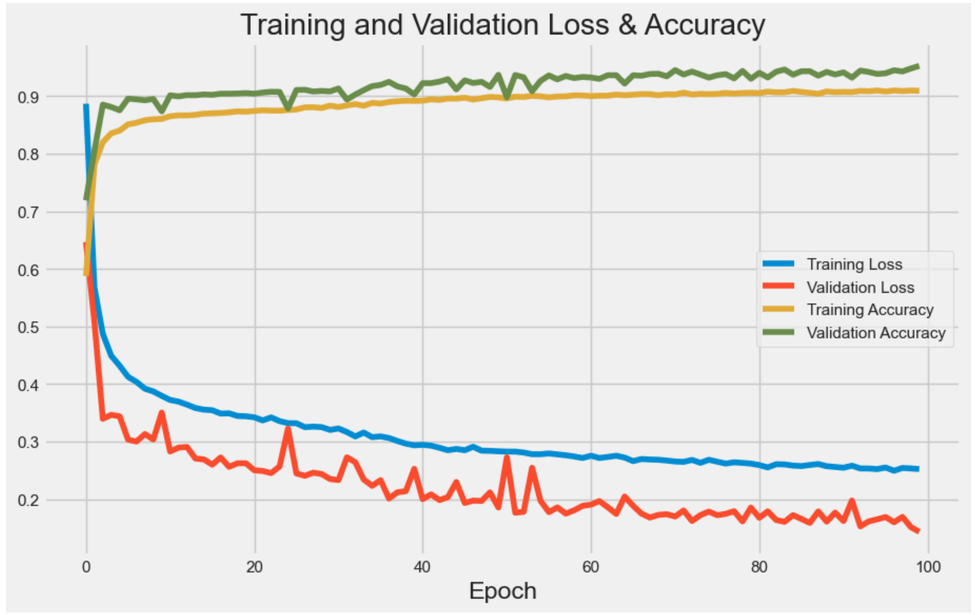
In [197]:
```
# Run models in parallel for Net1
daskList2 = []
for dense in dense_list:
    for drop in drop_list:
        for patience in patience_list:
            for learning in learning_list:
                s = train_with_early_stopping2(x_train.shape[1], dense, drop, patience, learning,
                                               criterion = nn.CrossEntropyLoss(), l1_reg=0.001, l2_reg=0.001)
                daskList2.append(s)
```

In [198]:
```
%%time
# Execute models using Dask for Net2
sa2 = dask.compute(*daskList2)
```

```
CPU times: total: 8min 55s
Wall time: 39min 5s
```

In [199]:
```
best_model = []
for i in range(len(sa2)):
    best_model.append(sa2[i][3][-1])

best = np.argmax(best_model)

print("Best Model's Validation Accuracy", sa2[best][3][-1])
plot_learning_curve(sa2[best][0], sa2[best][1], sa2[best][2], sa2[best][3])
```

```
Best Model's Validation Accuracy 0.9522272433828276
```



In [200]:
```
class ChessNet3(nn.Module):
    def __init__(self, input_size, dense_list, drop_list, l1_reg=0, l2_reg=0):
        super(ChessNet3, self).__init__()
        self.fc1 = nn.Linear(input_size, dense_list)
        self.fc2 = nn.Linear(dense_list, dense_list)
        self.fc3 = nn.Linear(dense_list, 3)
        self.dropout1 = nn.Dropout(drop_list)
        self.dropout2 = nn.Dropout(drop_list)
        self.bn1 = nn.BatchNorm1d(dense_list)
        self.bn2 = nn.BatchNorm1d(dense_list)
        self.l1_reg = l1_reg
        self.l2_reg = l2_reg

    def forward(self, x):
        x = F.relu(self.bn1(self.fc1(x)))
        x = self.dropout1(x)
        x = F.relu(self.bn2(self.fc2(x)))
        x = self.dropout2(x)
        x = self.fc3(x)
        return x

    def l1_l2_regularization(self):
        l1_reg_loss = 0
        l2_reg_loss = 0
        for param in self.parameters():
            l1_reg_loss += torch.norm(param, 1)
            l2_reg_loss += torch.norm(param, 2)
        return self.l1_reg * l1_reg_loss + self.l2_reg * l2_reg_loss

    def regularization(self):
        return self.l1_l2_regularization()
```

In [201]:
```
@dask.delayed
def train_with_early_stopping3(input_size, dense_list, drop_list, patience, learning_rate, criterion, l1_reg=1e-6, l2_reg=1e-6):
    # Instantiate your model
    model = ChessNet3(input_size=input_size, dense_list=dense_list, drop_list=drop_list)

    # Send model to device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    # Define the optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    # Define your data loaders (assuming you are using PyTorch DataLoader)
    train_dataset = torch.utils.data.TensorDataset(x_train, y_train)
    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)

    best_model_state = None
    best_val_loss = float('inf')
    current_patience = 0

    train_losses = []
    val_losses = []
    train_accs = []
    val_accs = []

    for epoch in range(100):  # Fixed 50 epochs
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = correct / total
        train_losses.append(epoch_loss)
        train_accs.append(epoch_acc)

        # Validation
        model.eval()
        val_outputs = model(x_val.to(device))
        val_loss = criterion(val_outputs, y_val.to(device))
        val_losses.append(val_loss.item())

        _, predicted = torch.max(val_outputs, 1)
        val_acc = (predicted == y_val).sum().item() / len(y_val)
        val_accs.append(val_acc)

        print(f"Epoch [{epoch+1}/20], Train Loss: {epoch_loss:.4f}, Train Acc: {epoch_acc:.4f}, Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")

        # Check for early stopping
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_state = model.state_dict()
            current_patience = 0
        else:
            current_patience += 1
            if current_patience == patience:
                print("Early stopping...")
                break

    # Load the best model state
    model.load_state_dict(best_model_state)

    return train_losses, val_losses, train_accs, val_accs
```

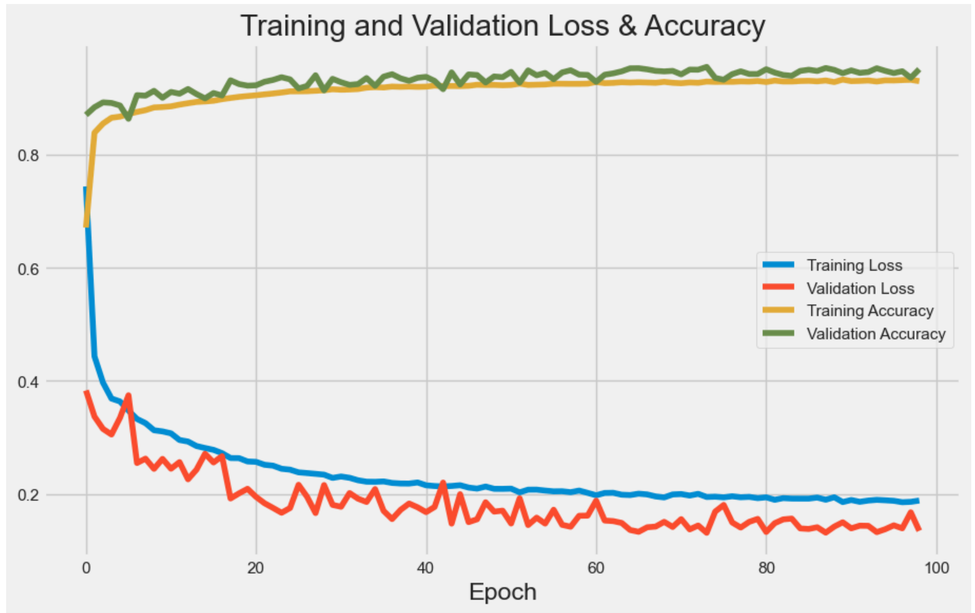
In [202]:
```
# Run models in parallel for Net3
daskList3 = []
for dense in dense_list:
    for drop in drop_list:
        for patience in patience_list:
            for learning in learning_list:
                s = train_with_early_stopping3(x_train.shape[1], dense, drop, patience, learning,
                                               criterion = nn.CrossEntropyLoss(), l1_reg=0.001, l2_reg=0.001)
                daskList3.append(s)
```

In [203]:
```
%%time
# Execute models using Dask for Net3
sa3 = dask.compute(*daskList3)
```

```
CPU times: total: 9min 41s
Wall time: 39min 16s
```

In [204]:
```
best_model = []
for i in range(len(sa3)):
    best_model.append(sa3[i][3][-1])

best = np.argmax(best_model)

print("Best Model's Validation Accuracy", sa3[best][3][-1])
plot_learning_curve(sa3[best][0], sa3[best][1], sa3[best][2], sa3[best][3])
```

```
Best Model's Validation Accuracy 0.9512127639952043
```



In [205]:
```
Execution = pd.read_excel('ExecutionTime.xlsx')

# Reshape the DataFrame for plotting
Execution_melted = Execution.melt(var_name='Workers', value_name='Time')

# Set style
sns.set(style='whitegrid')

# Create the box plot
fig, ax = plt.subplots(figsize=(10, 6))
g = sns.boxplot(x='Workers', y='Time', data=Execution_melted, width=0.7, palette=['plum', 'g', 'orange', 'b', 'r', 'y'])

# Titles and labels
plt.title("Execution Time In Minutes", fontsize=16)

# Set x tick-labels
xvalues = ["1 Worker", "2 Workers", "3 Workers", "4 Workers", "5 Workers", "6 Workers"]
plt.xticks(np.arange(6), xvalues)

# Remove all borders except bottom
sns.despine(top=True, right=True, left=True, bottom=False)

# Set colors of box plots
continent_colors = ["plum", "g", "orange", "b", "r", "y"]

# Iterate over each column (worker configuration)
for i, col in enumerate(Execution.columns):
    mean = round(Execution[col].mean(), 2)
    sd = round(Execution[col].std(), 2)
    textstr = f"$\overline{{x}}$ = {mean}\nstd = {sd}"
    props = dict(boxstyle='round', facecolor=continent_colors[i], alpha=0.2)
    g.text(i, 1.05, textstr, fontsize=10, bbox=props, transform=ax.transData, horizontalalignment='center')

plt.tight_layout()
plt.show()
```