Lichess Player Error Analysis

Investigating the Lichess API and data preprocessing for creating rich datasets and predictive models for chess game analysis.

Andrew Zhang

Abstract

In this project, I explore different ways in which chess performance can be quantitatively analyzed using game data created with Lichess' API. The first main objective was to create a comprehensive dataset containing key gameplay metrics like player ELO ratings, game outcomes, and move evaluation statistics like inaccuracies, mistakes, and blunders. I then used this dataset to investigate relationships between game results and player error evaluations through statistical testing and data science techniques. Data cleaning and feature extraction were used to facilitate these analyses. Models like logistic regression were employed to evaluate predictive capabilities of the scraped metrics. My findings highlight the predictive power of ELO ratings, move evaluations, and the questionable capabilities of transformers and neural networks for chess games.

Introduction

The game of chess has been a foundation of intellectual competition throughout history, with players' skill levels measured through standardized ELO ratings. Although ELO ratings capture long-term performance, a player's rating varies across chess websites and international systems. On the other hand, in-game decision-making reflects a player's real-time proficiency. Since online chess platforms like Lichess offer public access to their players' game records, some including chess-bot assisted move evaluations, we are provided a basis for detailed study of in-game decision-making.

For my AP Statistics final project in high school, I applied statistical testing to a Kaggle dataset created by Ahmed Alghafri seven years ago using Lichess' API. As an augment to that project, this project builds a similar dataset directly through the Lichess API. The dataset includes game features like player ratings, game outcomes, and centipawn evaluation scores, along with individual move assessments like inaccuracies, mistakes, and blunders. The project involves (1) collecting Lichess game data, (2) preprocessing the game data and extracting relevant features, (3) applying statistical testing, (4) building predictive models using machine learning techniques, and (5) evaluating each model's performance. The actual chronological order was a lot more back and forth, but overall, this report details this entire process.

Lichess Data Collection - Methodology

To access the data Lichess makes available, I first had to learn how to connect to the Lichess API with Python. Through reading the documentation, I was able to generate my own API key and manually request data from Lichess. The API allows you to get game data from a specific user, but you also need to input the username into the URL that you request from for that to happen, meaning that a list of usernames is also required. Lichess also has its own URL that you can query from with the input of a team name that returns a list of all members in a team. So, if you query a bunch of popular teams, you could get a lot of player usernames to work with.

Thus, the general algorithm arises: Loop through a list of teams (manually inputted), query the full team roster for each team, loop through all usernames gathered and scrape data from each of their games (I decided to do the 10 latest games for each player).

Lichess Data Collection - Application

The API proved to be quite finicky. Despite the Lichess API documentation telling users to use their API keys when querying data, some queries would return Error 403 if I used my key, and some would return the same error if I did not use the key. Aftering figuring out which calls required the key and which did not want the key, I then had to figure out how to fix a battery of networking issues that I had no idea how to deal with. The small segment of network handling in my code is written in blood—blood in this case being nights spent running the code and waking up to a network error. The main two networking errors I ran into looked like this:

1. ConnectionError: Max retries exceeded with url…
2. TimeoutError: [WinError 10060] A connection attempt failed…

The first error indicates that I was hitting the Lichess API's "rate limit," a limit set for the amount of requests you can send during a certain time frame. A pretty obvious answer to this is to slow down requests (by adding a delay). According to the documentation, another fix is to use an API token, but as usual, this did not work.

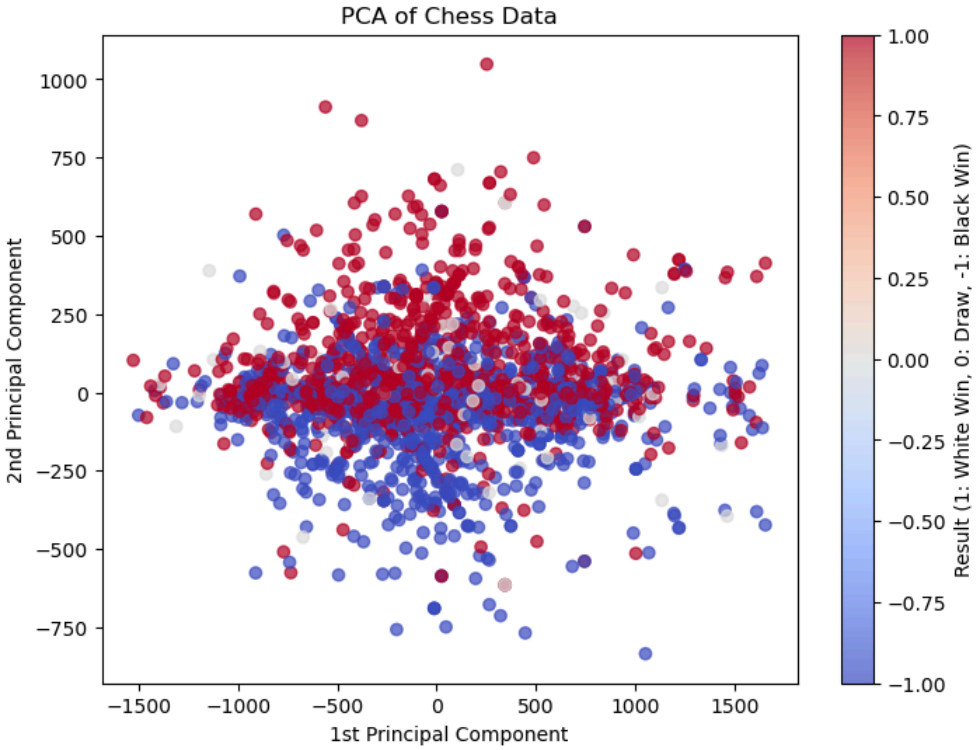
The second error is a direct result of the first error as it is a timeout error, meaning that the code tried too long to connect to the Lichess server without any response and as a result decided to terminate itself. I.e. Lichess got upset that I was requesting too much data and stopped responding, then the code saw that Lichess stopped responding, decided there was no point in continuing to try, and stopped.

I added some time.sleep() calls around the game collection code and some branching print statements to check for when either of these errors occurred. This final draft allowed me

to finally save a csv file of all of the data. However, the queries were the most basic possible, and as a result the dataset had no information about computer-analyzed centipawn values that show up in certain games (the values that I wrongly assumed at the time were what represented the different player error types). Instead, it just had the most basic information, including player ELO ratings, moves, termination type, and opening code.

PCA and KMeans Clustering

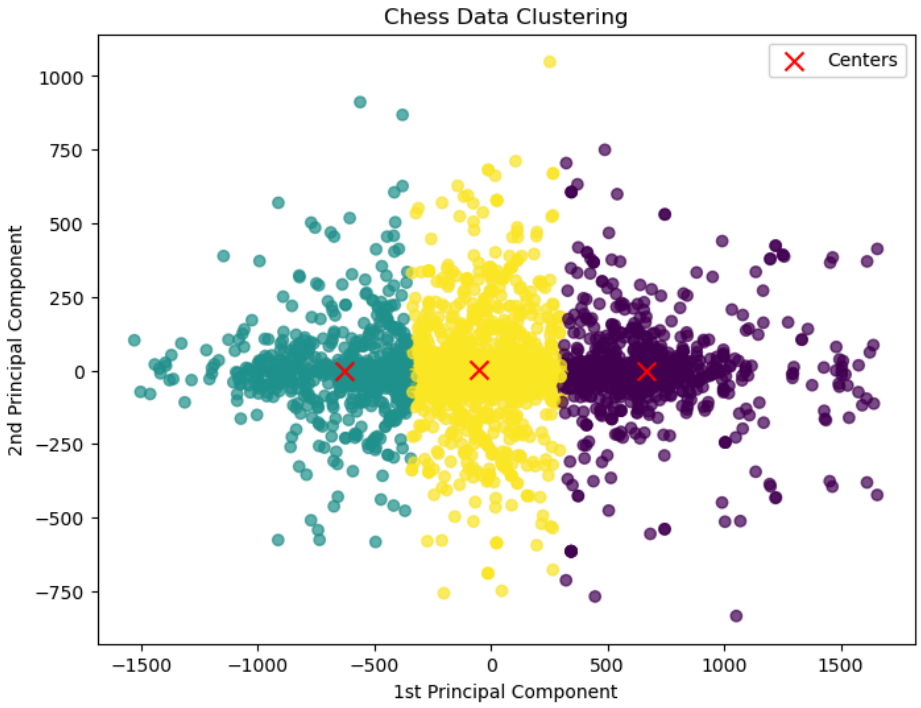
This segment was discussed in my progress report, so I will be more brief here. Before going back to tackling Lichess' API documentation, I applied some machine learning techniques to the data I already had. First, I applied PCA processing to explore the relationship between ELO ratings and game results.



Many of the games bunch up in the center, while one color takes over on the top and bottom. This is not by design, and thus it suggests a pattern in the data. Based on skill level influence, I guessed that the middle had games where the white and black ratings were relatively close, leading to a more balanced outcome distribution, while the top and bottom had games where the black/white player's ELO rating was higher by some significant margin. Checking the data confirmed this:

```
Elo for data point with largest 2nd principal component:
white_elo    2610
black_elo    1122
Name: 3298, dtype: object

Elo for data point with smallest 2nd principal component:
white_elo    1846
black_elo    3019
Name: 1862, dtype: object
```

I then applied KMeans clustering to the dataset:



In contrast to the PCA graph, the clusters here are divided vertically. I examined the elo rating averages within each cluster, which showed that each cluster was grouped by absolute ELO rating levels (centering around the 2100s, 1200s, and 1600s). So, each cluster essentially represents a general skill level range rather than a disparity.

Since game outcomes appear spread horizontally across the PCA graph, it seems that they aren't significantly influenced by general ELO rating levels alone, even if they might be affected by disparity. Instead, game outcomes appear consistently balanced across skill levels, suggesting that skill level doesn't strongly sway results for specific sides (later statistical testing would disprove this). The clustering doesn't contradict the idea that the existence of skill disparity influences game outcomes, as it doesn't assess ELO differences.

<u>Lichess Data Collection Revisited</u>

I then decided to go back to my data collection code to attempt to figure out how I could get player error counts. Because the Lichess API doesn't allow you to specifically query for inaccuracies, mistakes, and blunders, I had to filter out computer-analyzed games and then manually parse the moves column for such errors. Each element within the moves column includes the entire chess game in PGN (portable game notation) format, which was annoying for the data collection process as it meant that I could not request the data in JSON format and instead had to request it as a block of pure string.

To get the data I needed, I found that I had to have 3 parameters set to true when querying the games from each user: 'analysed,' 'evals,' and 'literate.' This was tricky to figure out as 'analysed' requires the query to only return computer analyzed games, 'evals' turns on centipawn evaluations, and 'literate' includes annotations. If any of these were not mentioned in my query parameters, the query would simply crash. Furthermore, I learned that with 'literate,' player errors would be directly printed into the moves, and as a result I would not need to use centipawn calculations to infer them.

Since I imported all of the data of each game as a pure block of string, I then had to regular-expression everything into columns in a dataframe. Thankfully, each section of data had the same tag format, so it was more tedious than difficult to code. For the final dataset, I decided to query the latest 10 games of 500 members from each major Lichess team I chose: Lichess Swiss, Coders, Bengal Tiger, IM Eric Rosen Fan Club, Zhigalko Sergei Fan Club, and Arab World Team. The result was theoretically 30,000 rows of data, corresponding to 30,000 games, but due to the enormous amounts of query requests I was sending, Lichess began to block my connection to its site in random 10 minute intervals. The result was a dataset in the 17,000s instead, but the data collection was accomplished; I had gotten the player error annotations.

Player Error Detection and Logging

The resulting moves column elements were blocks of string full of mashed up data including move notations, centipawn analyses, mistake annotations, and missed move possibilities. The numerous inconsistent dashes, brackets, and parenthesis made it nearly impossible to create a regex expression to extract the mistake annotations. The only thing to latch onto was the fact that white pieces had a single period next to their move number, while black pieces had three periods next to their move number, so we could at least assign a mistake to its player.

The resulting regex pattern that finally worked was:

```
r'(\d+\.)\s+(\S+)(?:\s+\{[^}]*?(Inaccuracy|Mistake|Blunder)[^}]*?\})?|\d+\.\.\.\s+(\S+)(
?:\s+\{[^}]*?(Inaccuracy|Mistake|Blunder)[^}]*?\})?'
```

The first half of the pattern deals with white moves, while the second half deals with black moves. They are required to be in one regex pattern so that they are chained together. For each half, the move number is captured, followed by the specific move notation. It then optionally captures an evaluation comment, which is where any centipawn evaluations, branching move possibilities, and player error annotations would be. By chaining all of these facets together, it guarantees that we capture the right data while also ensuring it is tied to the correct player and does not over count.

From the tuples created from the regex, I then extracted the moves of both sides and the type of mistake either side made, and checked whether either of the players made a mistake. If they did, I incremented the relevant counter. This finally added both sides' inaccuracy, mistake, and blunder counts to my dataset, matching it to the Kaggle dataset.

Statistical Analysis

The first thing I did with the completed dataset was perform some basic statistical testing on it to answer a few questions that have been historically popular in chess. For some context, the white winrate of the dataset was 49.92%, black winrate was 46.48%, and draw rate was 3.59%.

The first question I sought to answer was whether white wins significantly more than black, as this is widely considered to be true due to the white pieces' first-move advantage.[1] I performed a one-tailed Z-test for proportions, with my null hypothesis being that white and black win at the same rate, and my alternative hypothesis being that white wins more often than black. With a significance level of 0.05, I rejected the null hypothesis, finding that white wins significantly more than black.

Then, I computed the average ELO rating for each game by adding white and black's ELOs, and dividing it by 2. Using this, I split the games into lower-ELO and higher-ELO games, with lower being < 2000 and higher being >= 2000. I then did one-tailed Z-tests for means with the inaccuracies, mistakes, and blunders columns for both my dataset and the Kaggle dataset. I did these tests for an alternative hypothesis of lower-ELO committing more errors and of higher-ELO committing more errors. For my dataset, I found that there was significant evidence that lower-ELO games have more mistakes and blunders, but that higher-ELO games have more inaccuracies. This contrasted with the Kaggle dataset, as I found that lower-ELO had more of all types of errors. Either finding is explainable. Logically speaking, we expect lower-rated players in chess to make more mistakes, no matter the type. On the other hand, once players start entering the upper levels of gameplay, they often begin to engage in end-game battles. These dynamic battles are fast, desperate struggles where moves are often made in fractions of a second due to the lack of time that each player has remaining on their clocks, leaving much room for inaccuracies due to the lack of thinking time they have. Perhaps my dataset had more end-game battles than the Kaggle one.

Logistic Regression - Prediction with Player Errors

I then performed logistic regression using standardized average ELO ratings, inaccuracies, mistakes, and blunders for both sides to explore the relationship between game outcomes

[1] "First-move advantage in chess," *Wikipedia,* *https://en.wikipedia.org/wiki/First-move_advantage_in_chess#cite_ref-8*.

and key features. The target variable was whether the player with the white pieces won the game. After fitting the model, the classification report indicated a 79% accuracy, and the log-likelihood ratio test returned a p-value of 0.000, confirming that the overall model was statistically significant. All error-related coefficients were statistically significant, while the standardized ELO was not. The absolute values of the coefficients were approximately 0.3 for inaccuracies, 0.5 for mistakes, and 1.0 for blunders, reflecting the increasing severity of these errors. These findings suggest meaningful relationships between player errors and game outcomes. However, the model's predictive power is still constrained by its accuracy, highlighting the inherent complexity of chess games beyond what the player errors captured.

Deep Learning - Prediction with Player Errors

I then tried to do the same predictions but with a 3 layer neural network and 4 layer neural network using TensorFlow. Before training, I normalized the features, and split the training and test sets by 80-20.

I experimented extensively with hyperparameters, adding and removing layers, incorporating feature engineering, and applying early stopping, among other adjustments found through research. Despite these efforts, the highest accuracy achieved was 78.9% using a basic three-layer neural network, similar to what we've done in class but with larger hyperparameters.

The model had three ReLU activation layers, three dropout layers for regularization, and batch normalization to stabilize the training process. As usual, the output layer used a sigmoid activation function for binary classification. The model was then compiled with the Adam optimizer, and binary cross entropy loss function.

The simpler configuration yielding better results suggests that the underlying relationships in the dataset might not be complex enough to have a significant benefit from more complex architectures. Also, like with logistic regression, this performance highlights the complexity of chess outcomes, which depend on many tactical factors not captured by these features.

Deep Learning - PGN Notations

Something I also thought would be particularly interesting was whether neural networks would be able to "read" entire PGN notations and predict their outcomes. ChatGPT, a very famous transformer, has been notoriously bad at chess. Would a home-made one be just as bad?

Since each PGN game record explicitly states who wins the game at the end of its notation, I first had to use regex again to filter out those results. I then vectorized the records using TF-IDF and fed them into a basic three-layer neural network, and was surprised to find that it got the highest accuracy out of everything before with 0.82! This result suggests that chess outcomes might be predictable from PGN notations using neural networks, even with relatively simple architectures. However, perhaps the model could be improved further with a "recurrent" neural network, as they are apparently better at capturing sequential dependencies.

Then, mirroring ChatGPT, I applied the data to a transformer using Tokenizer. The transformer model included an embedding layer for token encodings, a multi-head attention layer, and a feedforward neural network. Despite its more advanced architecture, it did relatively…bad, with an accuracy of 0.67 despite taking the longest to train by a very large margin, significantly lower than the simpler neural networks tested previously. This result reveals several potential challenges. Though ChatGPT 3.5 was horrendous at chess, ChatGPT 4 is allegedly pretty good, so analyzing PGN data might require a more sophisticated Transformer model. It is also possible that the limited dataset size constrained the model's ability to generalize the complexity of chess. Overall, it seems that applying Transformer models to chess game analysis can be a pretty difficult task without further fine-tuning and more resources.

Conclusion

This project was a tumultuous and exciting dive into the world of data scraping, chess analysis, and machine learning. Scraping thousands of PGN records with Lichess' API and then extracting and structuring the annotation data often felt impossible, but I certainly learned a lot from it. Though counting player errors seemed simple at first, I really learned to appreciate the data cleaning side of data science by the end. While my tests and models weren't going to dethrone Magnus Carlsen, they did confirm some expected patterns and reveal some surprising nuances in how game outcomes unfold.

Finals really ate up my time after presentations, but if I were to continue this project, I would explore applying one-hot encoding to the opening move codes as an additional feature. This could enhance my models' predictive accuracy by capturing strategic implications of different chess openings.

Works Cited

Alghafri, Ahmed. “Lichess: Python Chess Games Statistics.” Kaggle, January 4, 2022. https://www.kaggle.com/datasets/ahmedalghafri/lichess-chess-games-statistics.

“First-Move Advantage in Chess.” Wikipedia, November 30, 2024. https://en.wikipedia.org/wiki/First-move_advantage_in_chess#cite_ref-8.

Super Duper Lichess Scraper 5000

December 17, 2024

[1]:
```
import requests
import pandas as pd
import time
import json
import re
```

[8]:
```
API_TOKEN = "I removed this for obvious reasons"

headers = {
    'Authorization': f'Bearer {API_TOKEN}'
}

response = requests.get('https://lichess.org/api/account', headers=headers)

# if response.status_code == 200:
#     print(response.json())
# else:
#     print(f"Error fetching account data: {response.status_code}")
```

0.1 Getting team members from a specific team

[57]:
```
def get_team_members(team_id, full=False, max_members=None):
    url = f'https://lichess.org/api/team/{team_id}/users'
    params = {'full': str(full).lower()}
    response = requests.get(url, stream=True)  # Use stream=True to process␣
 ↪data incrementally

    if response.status_code == 200:
        members = []
        for i, line in enumerate(response.iter_lines(decode_unicode=True)):
            if max_members and i >= max_members:
                break  # Stop processing if reach limit that I set
            members.append(json.loads(line))
        return members
    else:
        print(f"Error fetching team members: {response.status_code}")
        return []
```

0.2 Getting the games of a specific user

[51]:
```
def get_user_analyzed_games(username, max_games=10, retries=3):
    url = f'https://lichess.org/api/games/user/{username}'
    params = {
        'max': max_games,          # Limit the number of games
        'analysed': 'true',        # Only fetch analyzed games
        'evals': 'true',           # Include centipawn evaluations
        'literate': 'true',        # Include annotations like "mistake" and␣
↪"inaccuracy"
        'sort': 'dateDesc',        # Most recent games first
    }

    for i in range(retries):
        try:
            response = requests.get(url, headers=headers, params=params,␣
↪timeout=10)
            if response.status_code == 200:
                return response.text  # get the PGN response as a plain string
            elif response.status_code == 429:
                print(f"Rate limit hit. Retrying in {2 ** i} seconds...")
                time.sleep(2 ** i)  # Backoff exponentially to give it time to␣
↪rest
            else:
                print(f"Error fetching games for {username}: {response.
↪status_code}")
                return ""
        except requests.exceptions.Timeout:
            print(f"Timeout occurred. Retrying in {2 ** i} seconds...")
            time.sleep(2 ** i)
        except requests.exceptions.ConnectionError as e:
            print(f"Connection error for user {username}: {e}. Retrying in␣
↪{60*i} seconds...")
            time.sleep(60 * i)
    return ""
```

0.3 Extracting the player error counts

[52]:
```
def count_mistakes(moves):
    white_inaccuracies = white_mistakes = white_blunders = 0
    black_inaccuracies = black_mistakes = black_blunders = 0

    # Regex pattern to capture moves annotations
    move_pattern = r'(\d+\.)\s+(\S+)(?:\s+\{[^}]*?
↪(Inaccuracy|Mistake|Blunder)[^}]*?\})?|\d+\.\.\.\s+(\S+)(?:\s+\{[^}]*?
↪(Inaccuracy|Mistake|Blunder)[^}]*?\})?'
```

```
    matches = re.findall(move_pattern, moves)

    for match in matches:
        move_number, white_move, white_error, black_move, black_error = match

        if white_move and white_error:
            if white_error == "Inaccuracy":
                white_inaccuracies += 1
            elif white_error == "Mistake":
                white_mistakes += 1
            elif white_error == "Blunder":
                white_blunders += 1

        if black_move and black_error:
            if black_error == "Inaccuracy":
                black_inaccuracies += 1
            elif black_error == "Mistake":
                black_mistakes += 1
            elif black_error == "Blunder":
                black_blunders += 1

    return white_inaccuracies, white_mistakes, white_blunders,␣
↪black_inaccuracies, black_mistakes, black_blunders
```

0.4 Seperate all game data into their respective columns

[23]:
```
def process_pgn_data_with_mistakes(pgn_data):
    games = []
    games_data = pgn_data.strip().split("\n\n")  # Separate games based on␣
↪blank lines

    temp_game = None

    for game in games_data:
        if re.search(r'\[Event "', game):  # Can find main rows with the␣
↪'[Event' tag
            if temp_game:
                games.append(temp_game)  # Make sure we save the last game␣
↪before starting a new one
            temp_game = {}
            temp_game['event'] = re.search(r'\[Event "(.*?)"\]', game).group(1)
            temp_game['site'] = re.search(r'\[Site "(.*?)"\]', game).group(1)
            temp_game['date'] = re.search(r'\[Date "(.*?)"\]', game).group(1)
            temp_game['white'] = re.search(r'\[White "(.*?)"\]', game).group(1)
            temp_game['black'] = re.search(r'\[Black "(.*?)"\]', game).group(1)
            temp_game['result'] = re.search(r'\[Result "(.*?)"\]', game).
↪group(1)
```

```
            temp_game['white_elo'] = re.search(r'\[WhiteElo "(.*?)"\]', game).
↪group(1) or "Unknown"
            temp_game['black_elo'] = re.search(r'\[BlackElo "(.*?)"\]', game).
↪group(1) or "Unknown"
            temp_game['time_control'] = re.search(r'\[TimeControl "(.*?)"\]',␣
↪game).group(1)
            temp_game['eco'] = re.search(r'\[ECO "(.*?)"\]', game).group(1) or␣
↪"Unknown"
            temp_game['termination'] = re.search(r'\[Termination "(.*?)"\]',␣
↪game).group(1) or "Unknown"
        elif temp_game:
            # For unknown rows with moves
            moves = game.strip()
            if not moves.startswith('['):  # Make sure its not just another␣
↪header tag
                temp_game['moves'] = moves  # Attach  moves to the current game

                # Count mistakes from the moves with the function
                (
                    temp_game['white_inaccuracies'],
                    temp_game['white_mistakes'],
                    temp_game['white_blunders'],
                    temp_game['black_inaccuracies'],
                    temp_game['black_mistakes'],
                    temp_game['black_blunders']
                ) = count_mistakes(moves)

                games.append(temp_game)
                temp_game = None

    if temp_game:
        games.append(temp_game)

    return games
```

0.5 Saving

[24]:
```
def save_to_csv(games, filename='lichess_user_games.csv'):
    """Save the collected game data to a CSV file."""
    df = pd.DataFrame(games)
    df.to_csv(filename, index=False)
    print(f"Saved {len(games)} games to {filename}")
```

1 Master

Output has been cleared for obvious reasons

```
[ ]: team_ids = ['lichess-swiss' ,'coders', 'bengal-tiger',␣
      ↪'im-eric-rosen-fan-club', 'zhigalko_sergei-fan-club', 'arab-world-team']  #␣
      ↪Replace with your list of team IDs

     all_games = []

     for team_id in team_ids:
         print(f"Fetching members for team: {team_id}")
         members = get_team_members(team_id, max_members = 500)  # Fetch members for␣
      ↪the current team

         if members:
             for member in members:
                 username = member['id']  # 'id' = username
                 print(f"Fetching games for user: {username} from team: {team_id}")
                 pgn_data = get_user_analyzed_games(username, max_games=10)  # Limit␣
      ↪to 10 games per user

                 if pgn_data:
                     games = process_pgn_data_with_mistakes(pgn_data)
                     all_games.extend(games)
         else:
             print(f"No members found or error retrieving members for team:␣
      ↪{team_id}")

         print(f"Finished team: {team_id}. Total games collected: {len(all_games)}␣
      ↪rows")

     save_to_csv(all_games, 'games1.csv')
```

analyses

December 17, 2024

1 Looking at averages with the data

[49]:
```
import pandas as pd
df = pd.read_csv('games1.csv')

#  Win rate calculations
win_rate_white = len(df[df['result'] == '1-0']) / len(df) * 100
win_rate_black = len(df[df['result'] == '0-1']) / len(df) * 100
draw_rate = len(df[df['result'] == '1/2-1/2']) / len(df) * 100

print(f"White Win Rate: {win_rate_white:.2f}%")
print(f"Black Win Rate: {win_rate_black:.2f}%")
print(f"Draw Rate: {draw_rate:.2f}%")
```

```
White Win Rate: 49.92%
Black Win Rate: 46.48%
Draw Rate: 3.59%
```

[50]:
```
# Grouping ELO by ranges
df['white_elo'] = pd.to_numeric(df['white_elo'], errors='coerce')
df['black_elo'] = pd.to_numeric(df['black_elo'], errors='coerce')
df['elo_range'] = pd.cut(df[['white_elo', 'black_elo']].mean(axis=1), bins=[0,␣
 ↪1200, 1400, 1600, 1800, 2000, 2200, 2400], labels=['<1200', '1200-1400',␣
 ↪'1400-1600', '1600-1800', '1800-2000', '2000-2200', '>2200'])

# Find average inaccuracies by ELO range
mistakes_by_elo = df.groupby('elo_range')[['white_inaccuracies',␣
 ↪'black_inaccuracies']].mean()
print(mistakes_by_elo)
```

```
           white_inaccuracies  black_inaccuracies
elo_range
<1200                2.470008            2.508682
1200-1400            2.615385            2.600190
1400-1600            2.616856            2.629109
1600-1800            2.838897            2.904499
1800-2000            2.906936            2.929936
2000-2200            2.785079            2.844368
```

```
>2200                2.921911            2.874126
```

[51]:
```
df['white_elo'] = pd.to_numeric(df['white_elo'], errors='coerce')
df['black_elo'] = pd.to_numeric(df['black_elo'], errors='coerce')
df['elo_range'] = pd.cut(df[['white_elo', 'black_elo']].mean(axis=1), bins=[0,␣
 ↪1200, 1400, 1600, 1800, 2000, 2200, 2400], labels=['<1200', '1200-1400',␣
 ↪'1400-1600', '1600-1800', '1800-2000', '2000-2200', '>2200'])

# Mistakes instead
mistakes_by_elo = df.groupby('elo_range')[['white_mistakes', 'black_mistakes']].
 ↪mean()
print(mistakes_by_elo)
```

```
           white_mistakes  black_mistakes
elo_range
<1200            1.168508        1.174033
1200-1400        1.123457        1.157645
1400-1600        1.122236        1.130006
1600-1800        1.187228        1.190131
1800-2000        1.197098        1.212668
2000-2200        1.138834        1.131423
>2200            1.155012        1.149184
```

[52]:
```
df['white_elo'] = pd.to_numeric(df['white_elo'], errors='coerce')
df['black_elo'] = pd.to_numeric(df['black_elo'], errors='coerce')
df['elo_range'] = pd.cut(df[['white_elo', 'black_elo']].mean(axis=1), bins=[0,␣
 ↪1200, 1400, 1600, 1800, 2000, 2200, 2400], labels=['<1200', '1200-1400',␣
 ↪'1400-1600', '1600-1800', '1800-2000', '2000-2200', '>2200'])

# Blunders
mistakes_by_elo = df.groupby('elo_range')[['white_blunders', 'black_blunders']].
 ↪mean()
print(mistakes_by_elo)
```

```
           white_blunders  black_blunders
elo_range
<1200            2.313733        2.329124
1200-1400        2.121083        2.119658
1400-1600        2.065152        2.055589
1600-1800        1.926560        1.882729
1800-2000        1.843241        1.824133
2000-2200        1.658103        1.675395
>2200            1.741259        1.688811
```

[53]:
```
import scipy.stats as stats

total_games = len(df)
white_wins = len(df[df['result'] == '1-0'])
```

```
black_wins = len(df[df['result'] == '0-1'])
white_proportion = white_wins / total_games
black_proportion = black_wins / total_games

# Find pooled proportion
pooled_proportion = (white_wins + black_wins) / total_games

# Get sample sizes
n_white = total_games
n_black = total_games

# Calculate z-score
z_stat = (white_proportion - black_proportion) / (
    (pooled_proportion * (1 - pooled_proportion) * (1 / n_white + 1 / n_black))␣
 ↪** 0.5
)


p_value = 1 - stats.norm.cdf(z_stat)

print(f"Z-statistic: {z_stat:.4f}")
print(f"P-value: {p_value:.4f}")

if p_value < 0.05:
    print("Reject the null hypothesis: White wins significantly more than Black.
 ↪")
else:
    print("Fail to reject the null hypothesis: No significant difference in win␣
 ↪rates.")
```

```
Z-statistic: 17.2564
P-value: 0.0000
Reject the null hypothesis: White wins significantly more than Black.
```

[60]:
```
import numpy as np
import scipy.stats as stats

# Divide by average ELO
df['avg_elo'] = (df['white_elo'] + df['black_elo']) / 2
lower_elo = df[df['avg_elo'] < 2000]
higher_elo = df[df['avg_elo'] >= 2000]

# NOTE: This is 2 tailed
def z_test(group1, group2, column1, column2):
    n1, n2 = len(group1), len(group2)
    mean1, mean2 = (group1[column1] + group1[column2]).mean(), (group2[column1]␣
 ↪+ group2[column2]).mean()
```

```
    var1 = ((group1[column1] + group1[column2]).var(ddof=1))
    var2 = ((group2[column1] + group2[column2]).var(ddof=1))

    z_stat = (mean1 - mean2) / np.sqrt(var1 / n1 + var2 / n2)
    p_value = 2 * (1 - stats.norm.cdf(abs(z_stat)))

    return z_stat, p_value

tests = [('white_inaccuracies', 'black_inaccuracies'),
         ('white_mistakes', 'black_mistakes'),
         ('white_blunders', 'black_blunders')]

for test in tests:
    z, p = z_test(lower_elo, higher_elo, test[0], test[1])
    print(f"Z-test for combined {test[0].replace('white_', '')}:")
    print(f"  Z-statistic: {z:.4f}, P-value: {p:.4f}")
    if p < 0.05:
        print("  Significant difference between lower and higher Elo groups.")
    else:
        print("  No significant difference.")
```

```
Z-test for combined inaccuracies:
  Z-statistic: -3.1726, P-value: 0.0015
  Significant difference between lower and higher Elo groups.
Z-test for combined mistakes:
  Z-statistic: 1.9793, P-value: 0.0478
  Significant difference between lower and higher Elo groups.
Z-test for combined blunders:
  Z-statistic: 11.3134, P-value: 0.0000
  Significant difference between lower and higher Elo groups.
```

1.0.1 Lower has more than Upper

[61]:
```
import numpy as np
import scipy.stats as stats

# THIS is 1 tailed (lower favored)
def one_tailed_z_test(group1, group2, column1, column2):
    n1, n2 = len(group1), len(group2)
    mean1, mean2 = (group1[column1] + group1[column2]).mean(), (group2[column1]␣
 ↪+ group2[column2]).mean()
    var1 = ((group1[column1] + group1[column2]).var(ddof=1))
    var2 = ((group2[column1] + group2[column2]).var(ddof=1))

    z_stat = (mean1 - mean2) / np.sqrt(var1 / n1 + var2 / n2)
    p_value = 1 - stats.norm.cdf(z_stat)
```

```
    return z_stat, p_value

for test in tests:
    z, p = one_tailed_z_test(lower_elo, higher_elo, test[0], test[1])
    print(f"One-Tailed Z-test for combined {test[0].replace('white_', '')}:")
    print(f"  Z-statistic: {z:.4f}, P-value: {p:.4f}")
    if p < 0.05:
        print("  Significant evidence that lower Elo games have more of this.")
    else:
        print("  No significant evidence of a difference.")
```

```
One-Tailed Z-test for combined inaccuracies:
  Z-statistic: -3.1726, P-value: 0.9992
  No significant evidence of a difference.
One-Tailed Z-test for combined mistakes:
  Z-statistic: 1.9793, P-value: 0.0239
  Significant evidence that lower Elo games have more of this.
One-Tailed Z-test for combined blunders:
  Z-statistic: 11.3134, P-value: 0.0000
  Significant evidence that lower Elo games have more of this.
```

1.0.2 Upper has higher than lower

[62]:
```
# Opposite
def one_tailed_z_test_flipped(group1, group2, column1, column2):
    n1, n2 = len(group1), len(group2)
    mean1, mean2 = (group1[column1] + group1[column2]).mean(), (group2[column1]␣
 ↪+ group2[column2]).mean()
    var1 = ((group1[column1] + group1[column2]).var(ddof=1))
    var2 = ((group2[column1] + group2[column2]).var(ddof=1))

    z_stat = (mean1 - mean2) / np.sqrt(var1 / n1 + var2 / n2)
    p_value = stats.norm.cdf(z_stat)

    return z_stat, p_value

for test in tests:
    z, p = one_tailed_z_test_flipped(lower_elo, higher_elo, test[0], test[1])
    print(f"One-Tailed Z-test for combined {test[0].replace('white_', '')}␣
 ↪(higher > lower):")
    print(f"  Z-statistic: {z:.4f}, P-value: {p:.4f}")
    if p < 0.05:
        print("  Significant evidence that higher Elo games have more of this.")
    else:
        print("  No significant evidence of a difference.")
```

```
One-Tailed Z-test for combined inaccuracies (higher > lower):
  Z-statistic: -3.1726, P-value: 0.0008
```

```
  Significant evidence that higher Elo games have more of this.
One-Tailed Z-test for combined mistakes (higher > lower):
  Z-statistic: 1.9793, P-value: 0.9761
  No significant evidence of a difference.
One-Tailed Z-test for combined blunders (higher > lower):
  Z-statistic: 11.3134, P-value: 1.0000
  No significant evidence of a difference.
```

Possibly due to endgame battles etc...

2 Regression

[63]:
```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler

df = pd.read_csv('games1.csv')

# Make sure ELO is numeric
df['white_elo'] = pd.to_numeric(df['white_elo'], errors='coerce')
df['black_elo'] = pd.to_numeric(df['black_elo'], errors='coerce')

# Average ELO calcs & define the target
df['avg_elo'] = (df['white_elo'] + df['black_elo']) / 2
df['white_win'] = (df['result'] == '1-0').astype(int)

# Drop NaNs
df.dropna(subset=['avg_elo', 'white_mistakes', 'black_mistakes', 'white_win'],␣
 ↪inplace=True)


# Standardizing the average ELO
scaler = StandardScaler()
df['avg_elo_std'] = scaler.fit_transform(df[['avg_elo']])

X = df[['avg_elo_std', 'white_mistakes', 'black_mistakes',␣
 ↪'white_inaccuracies', 'black_inaccuracies', 'white_blunders',␣
 ↪'black_blunders']]

y = df['white_win']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

```
log_reg = LogisticRegression(max_iter=1000)  # Ensure convergence for large␣
 ↪datasets
log_reg.fit(X_train, y_train)

y_pred = log_reg.predict(X_test)
print(classification_report(y_test, y_pred))

coef_df = pd.DataFrame({'Predictor': ['Intercept'] + X.columns.tolist(),
                        'Coefficient': [log_reg.intercept_[0]] + list(log_reg.
 ↪coef_[0])})
print(coef_df)
```

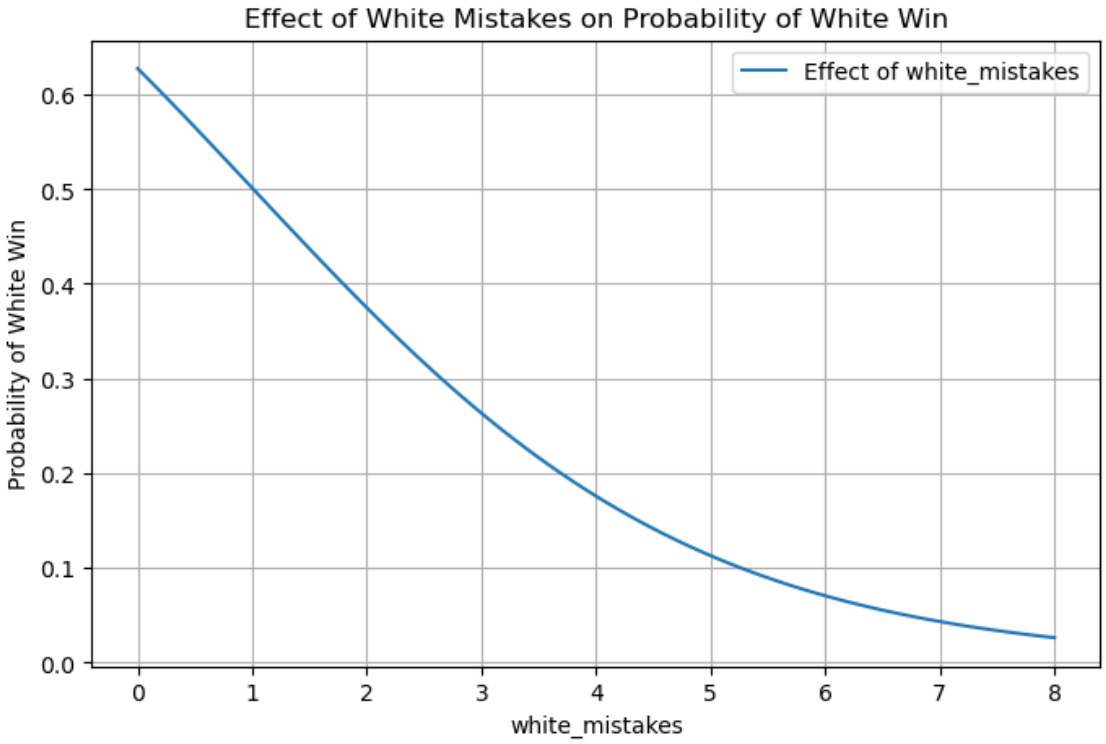
```
              precision    recall  f1-score   support

           0       0.80      0.78      0.79      1749
           1       0.77      0.79      0.78      1674

    accuracy                           0.79      3423
   macro avg       0.79      0.79      0.79      3423
weighted avg       0.79      0.79      0.79      3423

            Predictor  Coefficient
0           Intercept     0.042640
1         avg_elo_std    -0.010313
2      white_mistakes    -0.516763
3      black_mistakes     0.497439
4  white_inaccuracies    -0.309743
5  black_inaccuracies     0.300359
6      white_blunders    -1.021781
7      black_blunders     1.016079
```

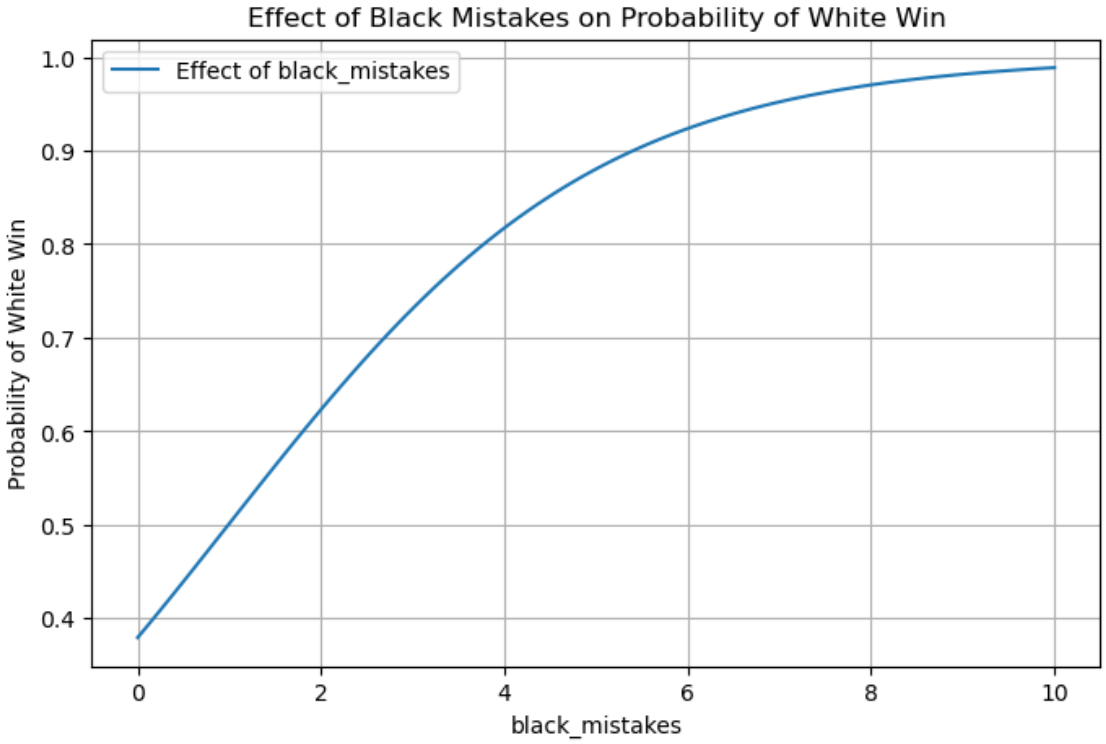
Funny thing is that it saw the avg_elo as a positive toward white as it went higher (though it is negligible so whatever)

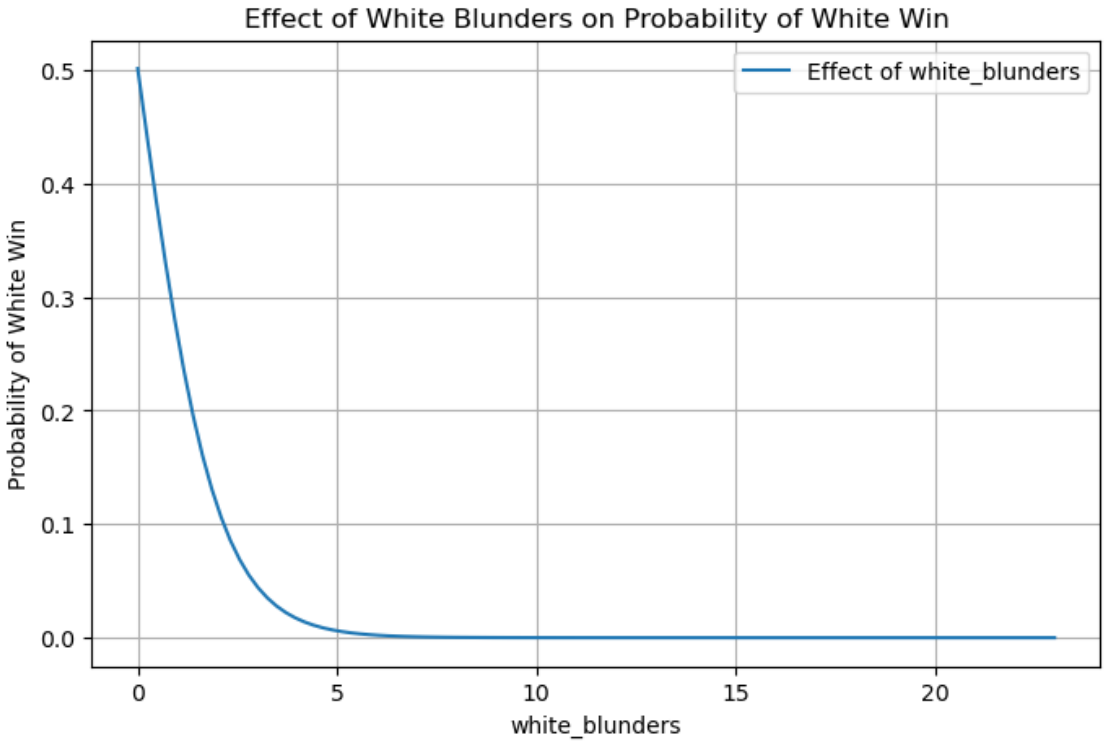
2.0.1 Visualization

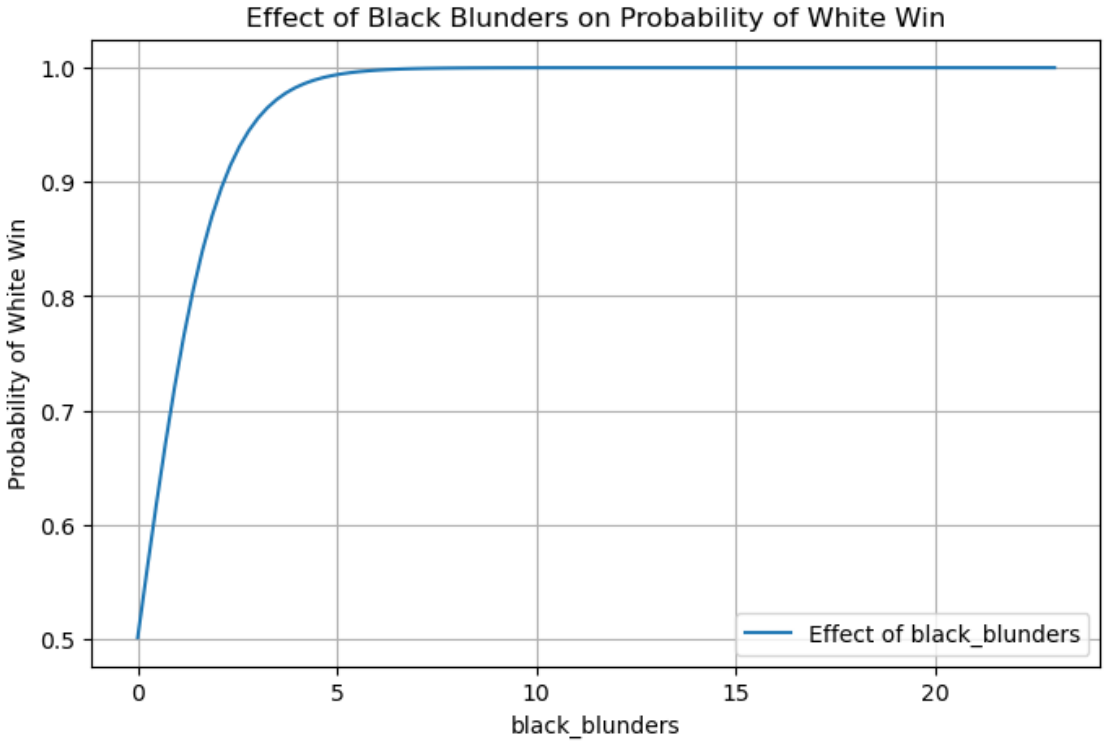
[41]:
```
# Try plotting the effect of the predictors
def plot_predictor_effect(predictor, X, log_reg, fixed_values, title):
    x_values = np.linspace(X[predictor].min(), X[predictor].max(), 100)
    X_plot = pd.DataFrame({predictor: x_values})

    # Fix the values for the other predictors
    for col, value in fixed_values.items():
        if col != predictor:
            X_plot[col] = value

    X_plot = X_plot[X.columns]
```

```
    # Predict probability of white win
    probabilities = log_reg.predict_proba(X_plot)[:, 1]

    # Plot stuff
    plt.figure(figsize=(8, 5))
    plt.plot(x_values, probabilities, label=f"Effect of {predictor}")
    plt.title(title)
    plt.xlabel(predictor)
    plt.ylabel("Probability of White Win")
    plt.legend()
    plt.grid()
    plt.show()

# Assigning the fixed values for other predictors
fixed_values = {
    "avg_elo": df['avg_elo'].mean(),
    "white_mistakes": 1,
    "black_mistakes": 1,
    "white_inaccuracies": 1,
    "black_inaccuracies": 1,
    "white_blunders": 0,
    "black_blunders": 0
}

# Plot for key predictors
# The mistakes and inaccuracies one look about the same
plot_predictor_effect("white_mistakes", X, log_reg, fixed_values, "Effect of␣
 ↪White Mistakes on Probability of White Win")
plot_predictor_effect("black_mistakes", X, log_reg, fixed_values, "Effect of␣
 ↪Black Mistakes on Probability of White Win")
plot_predictor_effect("white_blunders", X, log_reg, fixed_values, "Effect of␣
 ↪White Blunders on Probability of White Win")
plot_predictor_effect("black_blunders", X, log_reg, fixed_values, "Effect of␣
 ↪Black Blunders on Probability of White Win")
```


Effect of White Mistakes on Probability of White Win
Effect of white_mistakes
Probability of White Win
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
1
2
3
4
5
6
7
8
white_mistakes


Effect of Black Mistakes on Probability of White Win
Effect of black_mistakes
Probability of White Win
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0
2
4
6
8
10
black_mistakes


Effect of White Blunders on Probability of White Win
Effect of white_blunders
Probability of White Win
0.0
0.1
0.2
0.3
0.4
0.5
0
5
10
15
20
white_blunders


Effect of Black Blunders on Probability of White Win
Effect of black_blunders
Probability of White Win
0.5
0.6
0.7
0.8
0.9
1.0
0
5
10
15
20
black_blunders

[64]:
```
import statsmodels.api as sm
from scipy.stats import chi2

# Intercept-only model (can be just a constant)
X_null = sm.add_constant(X[['avg_elo_std']])
null_model = sm.Logit(y, X_null).fit()

# Full model (with all predictors)
X_full = sm.add_constant(X)
full_model = sm.Logit(y, X_full).fit()

# Likelihood ratio test
lr_stat = 2 * (full_model.llf - null_model.llf)
df_diff = X_full.shape[1] - X_null.shape[1]
p_value = chi2.sf(lr_stat, df_diff)

print(f"Likelihood Ratio Test Statistic: {lr_stat}")
print(f"Degrees of Freedom: {df_diff}")
print(f"P-Value: {p_value}")

if p_value < 0.05:
    print("The model is statistically significant.")
else:
    print("The model is not statistically significant.")
```

```
Optimization terminated successfully.
         Current function value: 0.693113
         Iterations 3
Optimization terminated successfully.
         Current function value: 0.508201
         Iterations 6
Likelihood Ratio Test Statistic: 6329.555460686144
Degrees of Freedom: 6
P-Value: 0.0
The model is statistically significant.
```

[65]:
```
import statsmodels.api as sm
from scipy.stats import chi2

X_full = sm.add_constant(X)
logit_model = sm.Logit(y, X_full).fit()

print(logit_model.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.508201
         Iterations 6
                           Logit Regression Results
```

```
==============================================================================
Dep. Variable:              white_win   No. Observations:                17115
Model:                          Logit   Df Residuals:                    17107
Method:                           MLE   Df Model:                            7
Date:                Wed, 04 Dec 2024   Pseudo R-squ.:                  0.2668
Time:                        18:36:42   Log-Likelihood:                -8697.9
converged:                       True   LL-Null:                       -11863.
Covariance Type:            nonrobust   LLR p-value:                     0.000
==============================================================================
======
                         coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
------
const                  0.0538      0.036      1.491      0.136      -0.017
0.125
avg_elo_std           -0.0152      0.019     -0.805      0.421      -0.052
0.022
white_mistakes        -0.5220      0.018    -28.852      0.000      -0.557
-0.487
black_mistakes         0.5132      0.018     28.853      0.000       0.478
0.548
white_inaccuracies    -0.3156      0.011    -28.815      0.000      -0.337
-0.294
black_inaccuracies     0.2946      0.011     26.782      0.000       0.273
0.316
white_blunders        -1.0317      0.018    -56.303      0.000      -1.068
-0.996
black_blunders         1.0259      0.019     55.021      0.000       0.989
1.062
==============================================================================
======
```

LLR p-value is from a chi squared test on the log liklihood ratio statistic

3 Deep Learning

[43]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn import metrics
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.preprocessing import StandardScaler
```

```
df = pd.read_csv('games1.csv')

# handle bad values
df['white_elo'] = pd.to_numeric(df['white_elo'], errors='coerce')
df['black_elo'] = pd.to_numeric(df['black_elo'], errors='coerce')
df.dropna(subset=['white_elo', 'black_elo'], inplace=True)

X = df[['white_elo', 'black_elo', 'white_mistakes', 'black_mistakes',␣
 ↪'white_inaccuracies', 'black_inaccuracies', 'white_blunders',␣
 ↪'black_blunders']].values
y = (df['result'] == '1-0').astype(int).values

# Normalizing
scaler = StandardScaler()
X = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# same stuff like from class
def create_model():
    model = Sequential()
    model.add(Dense(128, input_dim=X.shape[1], activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(64, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.2))
    model.add(Dense(32, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(16, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(
        loss='binary_crossentropy',
        optimizer='adam',
        metrics=['accuracy']
    )
    return model
```

[44]:
```
model = create_model()
# early_stop = EarlyStopping(monitor='val_loss', patience=5,␣
 ↪restore_best_weights=True)

model.fit(
    X_train, y_train, validation_data=(X_test, y_test),
    epochs=20, batch_size=32, verbose=2
```

```
)

scores = model.evaluate(X_test, y_test, verbose=0)
print(f"Accuracy: {scores[1] * 100:.2f}%")

model.summary()
```

```
Epoch 1/20

C:\Users\ripst\anaconda3\lib\site-packages\keras\src\layers\core\dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

428/428 - 2s - 4ms/step - accuracy: 0.6833 - loss: 0.6033 - val_accuracy: 0.7721
- val_loss: 0.4940
Epoch 2/20
428/428 - 0s - 1ms/step - accuracy: 0.7503 - loss: 0.5203 - val_accuracy: 0.7771
- val_loss: 0.4864
Epoch 3/20
428/428 - 0s - 1ms/step - accuracy: 0.7574 - loss: 0.5120 - val_accuracy: 0.7821
- val_loss: 0.4783
Epoch 4/20
428/428 - 0s - 1ms/step - accuracy: 0.7607 - loss: 0.5074 - val_accuracy: 0.7856
- val_loss: 0.4766
Epoch 5/20
428/428 - 0s - 1ms/step - accuracy: 0.7618 - loss: 0.5013 - val_accuracy: 0.7841
- val_loss: 0.4872
Epoch 6/20
428/428 - 0s - 1ms/step - accuracy: 0.7673 - loss: 0.4994 - val_accuracy: 0.7864
- val_loss: 0.4754
Epoch 7/20
428/428 - 0s - 1ms/step - accuracy: 0.7699 - loss: 0.4992 - val_accuracy: 0.7850
- val_loss: 0.4760
Epoch 8/20
428/428 - 0s - 1ms/step - accuracy: 0.7672 - loss: 0.4971 - val_accuracy: 0.7815
- val_loss: 0.4761
Epoch 9/20
428/428 - 0s - 1ms/step - accuracy: 0.7700 - loss: 0.4946 - val_accuracy: 0.7815
- val_loss: 0.4826
Epoch 10/20
428/428 - 0s - 1ms/step - accuracy: 0.7674 - loss: 0.4966 - val_accuracy: 0.7856
- val_loss: 0.4757
Epoch 11/20
428/428 - 0s - 1ms/step - accuracy: 0.7718 - loss: 0.4930 - val_accuracy: 0.7850
- val_loss: 0.4788
Epoch 12/20
428/428 - 0s - 1ms/step - accuracy: 0.7702 - loss: 0.4930 - val_accuracy: 0.7870
```

```
- val_loss: 0.4771
Epoch 13/20
428/428 - 0s - 1ms/step - accuracy: 0.7665 - loss: 0.4931 - val_accuracy: 0.7882
- val_loss: 0.4735
Epoch 14/20
428/428 - 0s - 1ms/step - accuracy: 0.7732 - loss: 0.4917 - val_accuracy: 0.7824
- val_loss: 0.4743
Epoch 15/20
428/428 - 0s - 1ms/step - accuracy: 0.7695 - loss: 0.4906 - val_accuracy: 0.7838
- val_loss: 0.4764
Epoch 16/20
428/428 - 0s - 1ms/step - accuracy: 0.7751 - loss: 0.4919 - val_accuracy: 0.7809
- val_loss: 0.4800
Epoch 17/20
428/428 - 0s - 1ms/step - accuracy: 0.7730 - loss: 0.4891 - val_accuracy: 0.7876
- val_loss: 0.4745
Epoch 18/20
428/428 - 0s - 1ms/step - accuracy: 0.7710 - loss: 0.4858 - val_accuracy: 0.7847
- val_loss: 0.4755
Epoch 19/20
428/428 - 0s - 1ms/step - accuracy: 0.7710 - loss: 0.4884 - val_accuracy: 0.7835
- val_loss: 0.4775
Epoch 20/20
428/428 - 0s - 1ms/step - accuracy: 0.7729 - loss: 0.4873 - val_accuracy: 0.7894
- val_loss: 0.4767
Accuracy: 78.94%
```

Model: "sequential_12"

Layer (type)	Output Shape	Param #
dense_56 (Dense)	(None, 128)	1,152
dropout_32 (Dropout)	(None, 128)	0
dense_57 (Dense)	(None, 64)	8,256
batch_normalization_8 (BatchNormalization)	(None, 64)	256

```
 dropout_33 (Dropout)                (None, 64)                  ␣
↪  0

 dense_58 (Dense)                    (None, 32)                  ␣
↪2,080

 dropout_34 (Dropout)                (None, 32)                  ␣
↪  0

 dense_59 (Dense)                    (None, 16)                  ␣
↪528

 dense_60 (Dense)                    (None, 1)                   ␣
↪ 17


 Total params: 36,613 (143.02 KB)

 Trainable params: 12,161 (47.50 KB)

 Non-trainable params: 128 (512.00 B)

 Optimizer params: 24,324 (95.02 KB)
```

[45]:
```
pred = (model.predict(X_test) > 0.5).astype(int).flatten()
print(
    f"Classification report for classifier:\n"
    f"{metrics.classification_report(y_test, pred, digits=4)}\n"
    f"{metrics.confusion_matrix(y_test, pred)}\n"
)
```

```
107/107             0s 949us/step
Classification report for classifier:
              precision    recall  f1-score   support

           0     0.8093    0.7690    0.7886      1749
           1     0.7706    0.8106    0.7901      1674

    accuracy                         0.7894      3423
   macro avg     0.7899    0.7898    0.7894      3423
weighted avg     0.7903    0.7894    0.7893      3423

[[1345  404]
 [ 317 1357]]
```

Tried other methods like feature engineering, early stopping, etc… That didn't work, this is the best so far.

[ ]:

Neural

December 17, 2024

1 Text Classification

1.0.1 Simple Neural Network

[2]:
```
import pandas as pd
import re

file_path = 'games1.csv'
df = pd.read_csv(file_path)

# Cleaning the moves column (can't let it cheat)
def clean_moves_column(moves):

    patterns_to_remove = [
        r'White wins', r'Black wins', r'1-0', r'1/2-1/2', r'0-1'
    ]

    combined_pattern = '|'.join(patterns_to_remove)

    cleaned_moves = re.sub(combined_pattern, '', moves, flags=re.IGNORECASE)
    return cleaned_moves

df['moves'] = df['moves'].apply(clean_moves_column)

output_path = 'cleaned_lichess_user_games.csv'
df.to_csv(output_path, index=False)

print(f"Cleaned data saved to {output_path}")
```

```
Cleaned data saved to cleaned_lichess_user_games.csv
```

[5]:
```
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Dropout
from tensorflow.keras.utils import to_categorical
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import classification_report


file_path = 'cleaned_lichess_user_games.csv'
df = pd.read_csv(file_path)

# Preprocess target variable
df['result_encoded'] = df['result'].map({'1-0': 1, '0-1': 0, '1/2-1/2': 2})
df = df.dropna(subset=['moves', 'result_encoded'])  # Drop rows withh problems␣
↪(missing data)

vectorizer = TfidfVectorizer(
    max_df=0.5,
    min_df=10,
    stop_words="english"
)
vector_data = vectorizer.fit_transform(df['moves'])
print(f"Vectorized moves data shape: {vector_data.shape}")

X_train, X_test, y_train, y_test = train_test_split(
    vector_data, df['result_encoded'], test_size=0.2, random_state=42
)

y_train = to_categorical(y_train, num_classes=3)
y_test = to_categorical(y_test, num_classes=3)

# Same stuff as from class
def neural_model(input_dim, num_classes):
    model = Sequential()
    model.add(Input(shape=(input_dim,)))
    model.add(Dense(512, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(128, activation='relu'))

    model.add(Dense(num_classes, activation='softmax'))

    model.compile(
        loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy']
    )
    return model

# Training
model = neural_model(vector_data.shape[1], 3)
```

```
model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=10,
    batch_size=32,
    verbose=2
)

# Evaluation
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

print("Classification Report:")
print(classification_report(y_test_classes, y_pred_classes,␣
 ↪target_names=['Black Win', 'White Win', 'Draw']))
```

```
Vectorized moves data shape: (17445, 1979)
Epoch 1/10
437/437 - 5s - 10ms/step - accuracy: 0.7172 - loss: 0.6092 - val_accuracy:
0.7931 - val_loss: 0.4707
Epoch 2/10
437/437 - 3s - 8ms/step - accuracy: 0.8278 - loss: 0.3748 - val_accuracy: 0.8054
- val_loss: 0.4258
Epoch 3/10
437/437 - 3s - 8ms/step - accuracy: 0.8846 - loss: 0.2568 - val_accuracy: 0.8077
- val_loss: 0.4371
Epoch 4/10
437/437 - 3s - 8ms/step - accuracy: 0.9291 - loss: 0.1612 - val_accuracy: 0.8085
- val_loss: 0.4764
Epoch 5/10
437/437 - 3s - 8ms/step - accuracy: 0.9646 - loss: 0.0935 - val_accuracy: 0.8137
- val_loss: 0.6564
Epoch 6/10
437/437 - 3s - 8ms/step - accuracy: 0.9771 - loss: 0.0630 - val_accuracy: 0.8011
- val_loss: 0.7268
Epoch 7/10
437/437 - 3s - 8ms/step - accuracy: 0.9845 - loss: 0.0463 - val_accuracy: 0.8151
- val_loss: 0.8581
Epoch 8/10
437/437 - 3s - 8ms/step - accuracy: 0.9879 - loss: 0.0339 - val_accuracy: 0.8126
- val_loss: 0.7471
Epoch 9/10
437/437 - 3s - 8ms/step - accuracy: 0.9893 - loss: 0.0320 - val_accuracy: 0.8097
- val_loss: 0.8666
Epoch 10/10
437/437 - 3s - 8ms/step - accuracy: 0.9890 - loss: 0.0293 - val_accuracy: 0.8166
- val_loss: 0.9442
```

```
110/110             0s 2ms/step
Classification Report:
              precision    recall  f1-score   support

   Black Win       0.81      0.80      0.81      1591
   White Win       0.82      0.83      0.83      1770
        Draw       0.77      0.83      0.80       128

    accuracy                           0.82      3489
   macro avg       0.80      0.82      0.81      3489
weighted avg       0.82      0.82      0.82      3489
```

1.1 Transformer

[6]:
```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import layers, Model, Sequential
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
```

[7]:
```
file_path = 'cleaned_lichess_user_games.csv'
df = pd.read_csv(file_path)

df['result_encoded'] = df['result'].map({'1-0': 1, '0-1': 0, '1/2-1/2': 2})
df = df.dropna(subset=['moves', 'result_encoded'])
```

[8]:
```
# Tokenizing and padding
tokenizer = Tokenizer()
tokenizer.fit_on_texts(df['moves'])
vocab_size = len(tokenizer.word_index) + 1
print("Vocabulary Size:", vocab_size)

maxlen = 500
X = tokenizer.texts_to_sequences(df['moves'])
X = pad_sequences(X, maxlen=maxlen, padding='post', truncating='post')

y = to_categorical(df['result_encoded'], num_classes=3)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

```
Vocabulary Size: 4758
```

[9]:
```
# Pretty much all from in-class material
class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, heads, neurons):
        super(TransformerEncoder, self).__init__()
        self.att = layers.MultiHeadAttention(num_heads=heads, key_dim=embed_dim)
        self.ffn = Sequential([
            layers.Dense(neurons, activation="relu"),
            layers.Dense(embed_dim),
        ])
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = layers.Dropout(0.5)
        self.dropout2 = layers.Dropout(0.5)

    def call(self, inputs, training):
        attn_output = self.att(inputs, inputs)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)

class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, maxlen, vocab_size, embed_dim):
        super(TokenAndPositionEmbedding, self).__init__()
        self.token_emb = layers.Embedding(input_dim=vocab_size,␣
 ↪output_dim=embed_dim)
        self.pos_emb = layers.Embedding(input_dim=maxlen, output_dim=embed_dim)

    def call(self, x):
        positions = tf.range(start=0, limit=tf.shape(x)[-1], delta=1)
        positions = self.pos_emb(positions)
        x = self.token_emb(x)
        return x + positions
```

[10]:
```
embed_dim = 128
heads = 4
neurons = 64

# Build!
inputs = layers.Input(shape=(maxlen,))
embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size, embed_dim)
x = embedding_layer(inputs)
transformer_block = TransformerEncoder(embed_dim, heads, neurons)
x = transformer_block(x, training=True)
x = layers.GlobalAveragePooling1D()(x)
x = layers.Dropout(0.3)(x)
```

```
outputs = layers.Dense(3, activation="softmax")(x)
```

```
WARNING:tensorflow:From C:\Users\ripst\anaconda3\lib\site-
packages\keras\src\backend\tensorflow\core.py:204: The name tf.placeholder is
deprecated. Please use tf.compat.v1.placeholder instead.
```

[11]:
```
model = Model(inputs=inputs, outputs=outputs)
model.compile(optimizer=tf.keras.optimizers.Adam(0.0003),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.summary()
```

```
Model: "functional_10"
```

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 500)	0
token_and_position_embedding (TokenAndPositionEmbedding)	(None, 500, 128)	673,024
transformer_encoder (TransformerEncoder)	(None, 500, 128)	280,896
global_average_pooling1d (GlobalAveragePooling1D)	(None, 128)	0
dropout_5 (Dropout)	(None, 128)	0
dense_9 (Dense)	(None, 3)	387

```
Total params: 954,307 (3.64 MB)
```

Trainable params: 954,307 (3.64 MB)

Non-trainable params: 0 (0.00 B)

[12]:
```
# Training!
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=10, batch_size=32, verbose=2)

# Evaluation
y_pred = np.argmax(model.predict(X_test), axis=-1)
y_true = np.argmax(y_test, axis=-1)
```

```
Epoch 1/10
437/437 - 230s - 527ms/step - accuracy: 0.5107 - loss: 0.8254 - val_accuracy:
0.5417 - val_loss: 0.8059
Epoch 2/10
437/437 - 224s - 513ms/step - accuracy: 0.5741 - loss: 0.7848 - val_accuracy:
0.5870 - val_loss: 0.7883
Epoch 3/10
437/437 - 231s - 528ms/step - accuracy: 0.6510 - loss: 0.6867 - val_accuracy:
0.6664 - val_loss: 0.6599
Epoch 4/10
437/437 - 226s - 516ms/step - accuracy: 0.6960 - loss: 0.6060 - val_accuracy:
0.6856 - val_loss: 0.6176
Epoch 5/10
437/437 - 222s - 509ms/step - accuracy: 0.7131 - loss: 0.5733 - val_accuracy:
0.6930 - val_loss: 0.6239
Epoch 6/10
437/437 - 225s - 516ms/step - accuracy: 0.7244 - loss: 0.5535 - val_accuracy:
0.6647 - val_loss: 0.6998
Epoch 7/10
437/437 - 222s - 509ms/step - accuracy: 0.7413 - loss: 0.5233 - val_accuracy:
0.6876 - val_loss: 0.6496
Epoch 8/10
437/437 - 223s - 511ms/step - accuracy: 0.7568 - loss: 0.5031 - val_accuracy:
0.6784 - val_loss: 0.7086
Epoch 9/10
437/437 - 222s - 508ms/step - accuracy: 0.7668 - loss: 0.4807 - val_accuracy:
0.6713 - val_loss: 0.7252
Epoch 10/10
437/437 - 222s - 509ms/step - accuracy: 0.7830 - loss: 0.4550 - val_accuracy:
0.6667 - val_loss: 0.7800
110/110             19s 175ms/step
```

[13]:
```
from sklearn.metrics import classification_report, confusion_matrix
print("Classification Report:")
print(classification_report(y_true, y_pred, target_names=['Black Win', 'White␣
↪Win', 'Draw']))
print("Confusion Matrix:")
print(confusion_matrix(y_true, y_pred))
```

```
Classification Report:
              precision    recall  f1-score   support

   Black Win       0.71      0.57      0.63      1591
   White Win       0.66      0.78      0.71      1770
        Draw       0.36      0.36      0.36       128

    accuracy                           0.67      3489
   macro avg       0.58      0.57      0.57      3489
weighted avg       0.67      0.67      0.66      3489

Confusion Matrix:
[[ 902  656   33]
 [ 342 1378   50]
 [  27   55   46]]
```

2 PCA and KMeans

In the end I wasn't quite sure how to analyze this so I just left it out of my report. Here it is though if you were curious about this.

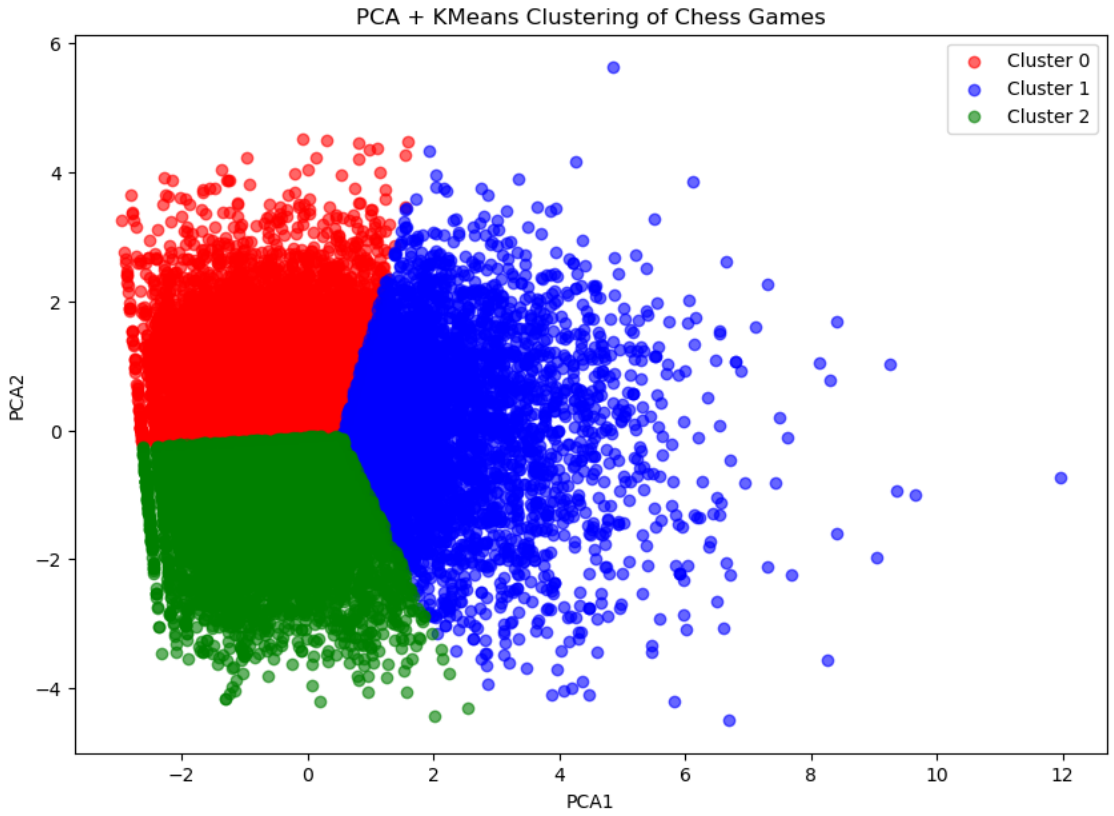
[15]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
```

[16]:
```
file_path = 'cleaned_lichess_user_games.csv'
df = pd.read_csv(file_path)

features = ['white_inaccuracies', 'white_mistakes', 'white_blunders',
            'black_inaccuracies', 'black_mistakes', 'black_blunders',
            'white_elo', 'black_elo']

# Replace the question marks with NaN
df[features] = df[features].replace('?', np.nan)
df[features] = df[features].apply(pd.to_numeric, errors='coerce')

# Remove rows with missing values
```

```
df = df.dropna(subset=features)
```

[17]:
```
scaler = StandardScaler()
scaled_features = scaler.fit_transform(df[features])

# Applying PCA
pca = PCA(n_components=2)
pca_result = pca.fit_transform(scaled_features)

pca_df = pd.DataFrame(pca_result, columns=['PCA1', 'PCA2'])
pca_df['result'] = df['result']
```

[18]:
```
# Applying KMeans
kmeans = KMeans(n_clusters=3, random_state=42)
pca_df['cluster'] = kmeans.fit_predict(pca_result)

plt.figure(figsize=(10, 7))
colors = ['red', 'blue', 'green']
for cluster in range(3):
    cluster_data = pca_df[pca_df['cluster'] == cluster]
    plt.scatter(cluster_data['PCA1'], cluster_data['PCA2'],
                label=f'Cluster {cluster}', alpha=0.6, color=colors[cluster])
plt.title('PCA + KMeans Clustering of Chess Games')
plt.xlabel('PCA1')
plt.ylabel('PCA2')
plt.legend()
plt.show()

print("Explained Variance Ratio:", pca.explained_variance_ratio_)
print(pca_df.groupby('cluster')['result'].value_counts())
```




```
Explained Variance Ratio: [0.33325647 0.23172469]
cluster  result
0        1-0        3394
         0-1        3226
         1/2-1/2     239
1        1-0        2118
         0-1        1914
         1/2-1/2     151
2        1-0        2853
         0-1        2693
         1/2-1/2     198
Name: result, dtype: int64
```

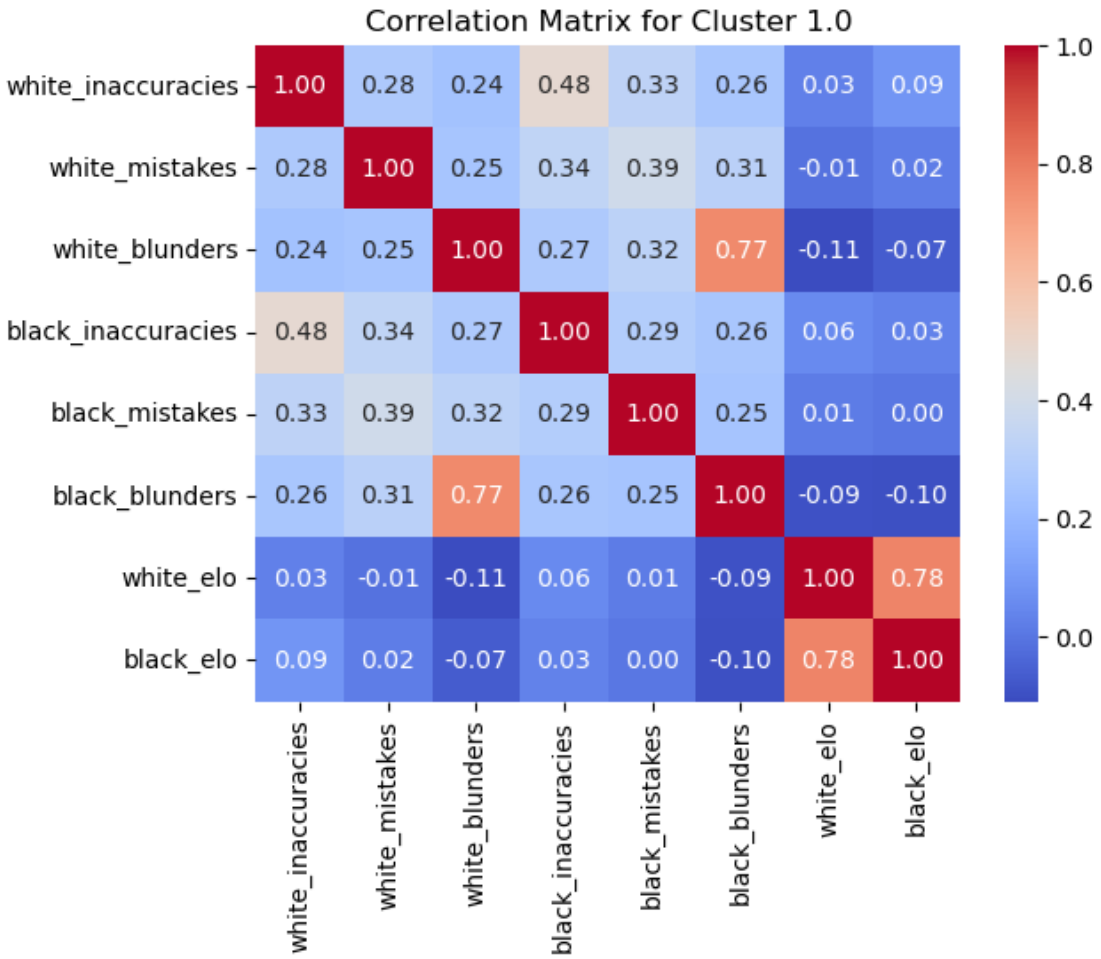
[20]:
```
# Just some stats
cluster_stats = df.groupby('cluster')[features].describe()
print(cluster_stats)
```

```
        white_inaccuracies                                                \
                     count      mean       std  min  25%  50%  75%   max
cluster
0.0                 6859.0  2.754337  2.159608  0.0  1.0  2.0  4.0  16.0
1.0                 4183.0  2.740617  2.102785  0.0  1.0  2.0  4.0  14.0
```

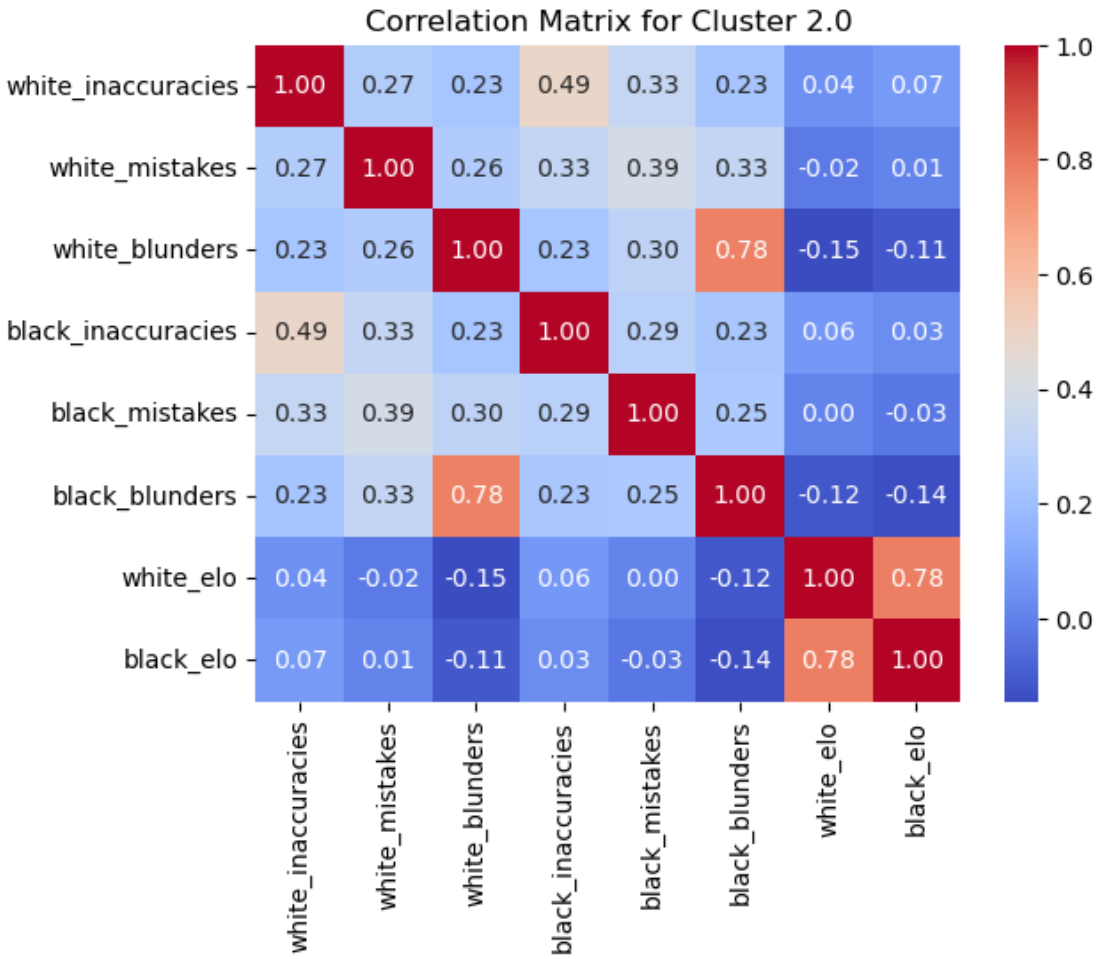
```
2.0                5744.0  2.685237  2.090454  0.0  1.0  2.0  4.0  13.0

        white_mistakes                 … white_elo          black_elo  \
                 count      mean       …       75%     max     count
cluster                                …
0.0             6859.0  1.167371       …    1957.0  2994.0    6859.0
1.0             4183.0  1.151805       …    1934.0  3068.0    4183.0
2.0             5744.0  1.149896       …    1888.0  3181.0    5744.0


                mean         std    min      25%     50%     75%     max
cluster
0.0      1658.371045  411.157543  400.0  1384.00  1671.0  1956.0  2988.0
1.0      1631.052833  422.112186  400.0  1350.00  1643.0  1934.5  3235.0
2.0      1601.213962  419.881355  422.0  1324.75  1605.0  1894.0  3122.0

[3 rows x 64 columns]
```

[22]:
```python
import seaborn as sns

# Take a look at the correlation matrix
for cluster in df['cluster'].unique():
    print(f"Correlation matrix for Cluster {cluster}:")
    cluster_data = df[df['cluster'] == cluster][features]
    corr_matrix = cluster_data.corr()
    sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap="coolwarm")
    plt.title(f'Correlation Matrix for Cluster {cluster}')
    plt.show()
```

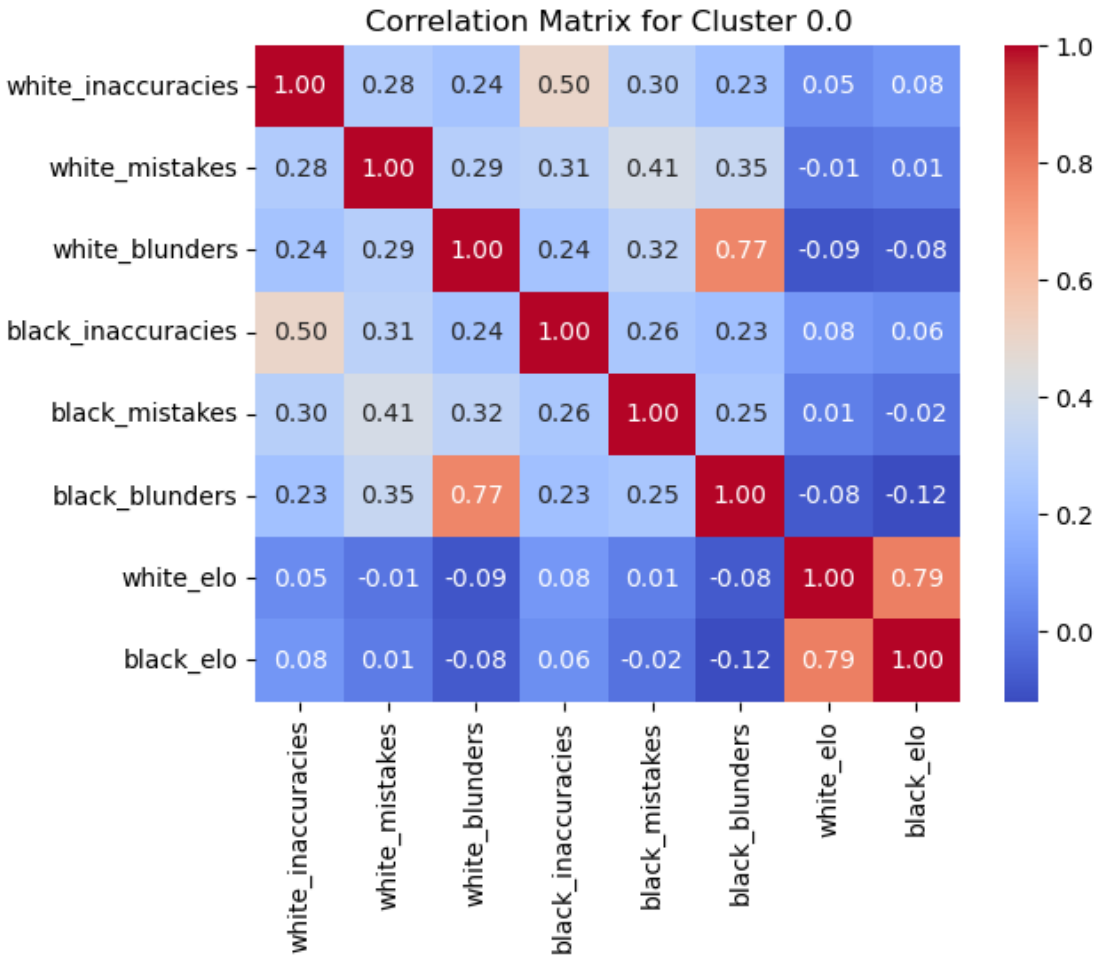
```
Correlation matrix for Cluster 1.0:
```



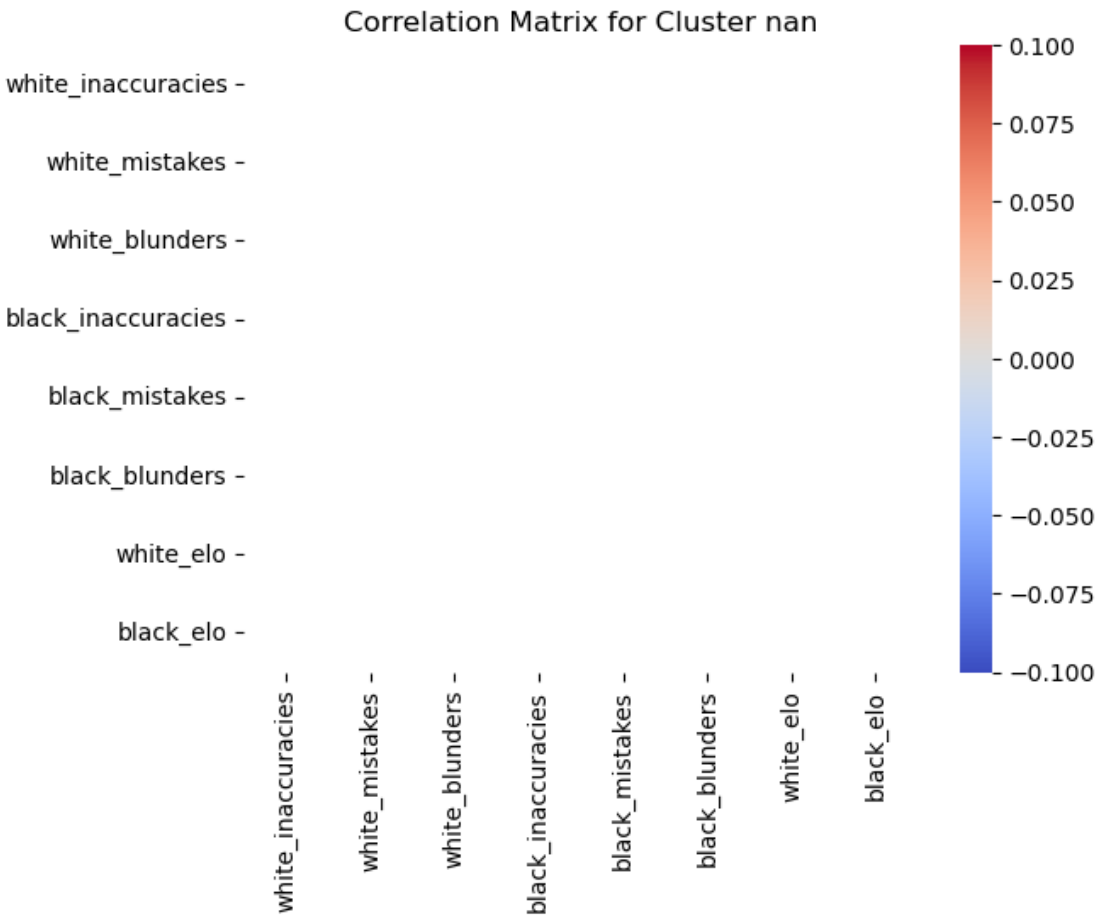
Correlation matrix for Cluster 2.0:



Correlation matrix for Cluster 0.0:




```
Correlation matrix for Cluster nan:

C:\Users\ripst\anaconda3\lib\site-packages\seaborn\matrix.py:198:
RuntimeWarning: All-NaN slice encountered
  vmin = np.nanmin(calc_data)
C:\Users\ripst\anaconda3\lib\site-packages\seaborn\matrix.py:203:
RuntimeWarning: All-NaN slice encountered
  vmax = np.nanmax(calc_data)
```


Correlation Matrix for Cluster nan
white_inaccuracies
white_mistakes
white_blunders
black_inaccuracies
black_mistakes
black_blunders
white_elo
black_elo
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100

[ ]: