Estimating social rank among signees of 12th century legal documents using MCMC

1 Introduction

The dataset describes 134 documents (games) signed by random permutations of 24 individuals (players/members) during the 12th century. These games consist of legal documents signed in approximate order of social rank. The data on each game consist of the order of signatures for the players involved and their relative age (seniority level) at the time of signing. We explore this data and create a Bayesian Markov chain Monte Carlo (MCMC) Metropolis-Hastings algorithm to obtain the distributions of the underlying social ranks (skills) of the 24 players. Additionally, the relationship between seniority and game-ranking is explored, with seniority used as a covariate in the model.

2 Data Exploration

2.1 Numbers of games, players, and opponents

In the dataset, each of the 24 members has a unique player ID number from one to twenty-four. A majority (63%) of the games consisted of only two players, with the average game consisting of 2.7 players. A bar graph showing the frequency of different game-sizes is shown in Figure 1. Some members participated in far more games than other members, resulting in a highly skewed distribution. However, the number of (unique) opponents each player encountered over the course of the 134 games was more uniform, indicating that those who played in fewer games encountered a more diverse set of opponents within their games, and vice versa. Plots of the number of games played and the number of opponents encountered by each member are given in Figure 2.

Crucially for analysis, players who participated in more games and encountered more opponents have more data, in a certain sense. This may mean that the posterior skill-distributions of these players will be more precise. As we will discuss player 7 in more detail, it is worth remarking that there is an above-average amount of data on player 7, who played in 47 games against 12 other members.

Though not shown here, the distribution of the number of games, or interactions, members had with the opponents they faced is also fairly uniform. Across the 24 players, the number of interactions per opponent has a mean of 2.93 and a median of 2.45 with an interquartile range (IQR) of 1.73. That is, players typically got about two or three opportunities throughout the games to face off against the opponents they encountered. Player 7 had the greatest number of interactions per opponent, with 7.4 interactions per opponent faced.

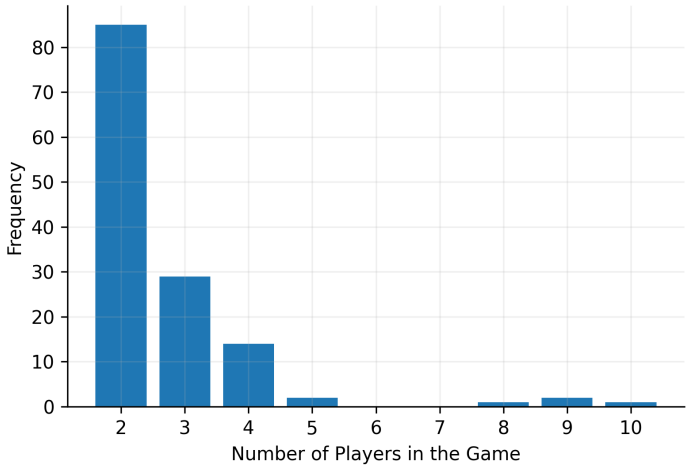

Figure 1: Bar graph of the frequency of different game-sizes.

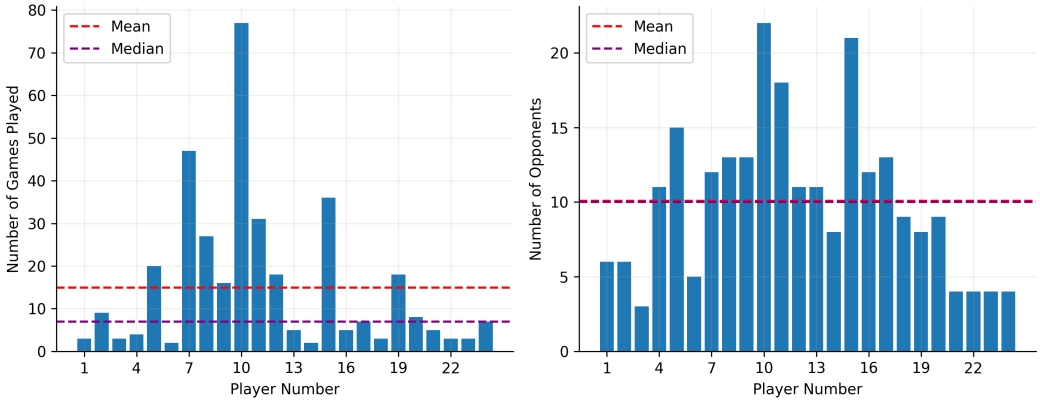

Figure 2: Bar graph of the player ID number against the number of games played and number of unique opponents faced.

2.2 Seniority

The seniority of a player gives his or her age as a ranking relative to the other players in the field at the time the game was played, with the eldest ranked first and the youngest last. The seniority of a player changes over time as the field of players changes (players join and drop out).

The seniority of players remained fairly constant across the games they played, plots of seniority over the number of games played are shown in Figure 3 for the 18 members who played in less than twenty games. The remaining six members who played in more games are not shown in these plots to allow for more clear visualization and avoid overcrowding by these members. The plots for these six members are not characteristically different. As can be seen in the plots, seniority changes by a relatively small amount between games. Across all the members, the change in seniority between a player's first and last game has an average of 1.4, a median of 0.5, and an IQR of 2.25.

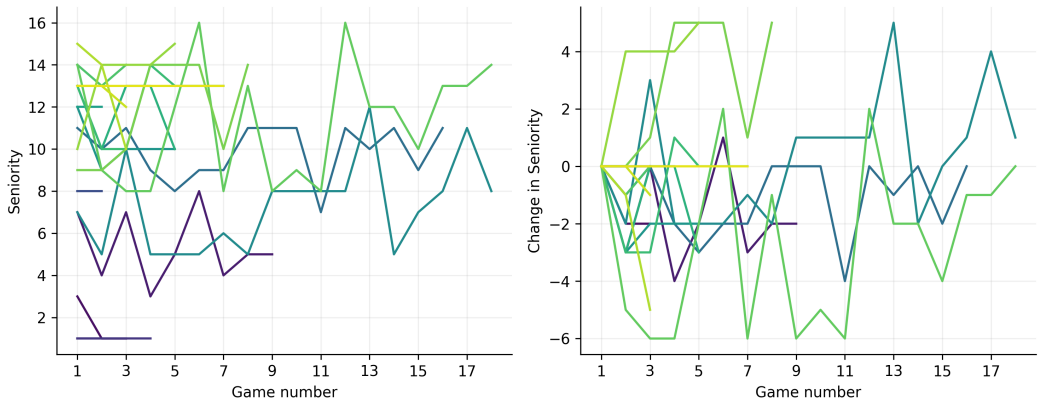

Figure 3: Movement in seniority over the number of games played by each member, in the order that they were played.

Oddities

Players 2 and 8 encountered each other in seven games and in each of these games, they had the same seniority. Perhaps these two members shared the same birthday.

We will define a player's mean seniority ranking to be the mean seniority ranking across all of the games he or she participated in. The player ID numbers provided by the dataset have a close correspondence to the mean seniority rankings of each player. The correlation coefficient between player number and mean seniority is 0.89 and highly statistically significant.

Relation between seniority and social rank within each game

We may wish to illuminate the relationship between the seniority and signature ranking (outcome of the game). One way to accomplish this is to calculate Spearman's Rho between the seniority and outcome ranking for each game. These are plotted in Figure 4. As most of the games consisted of only two players, Spearman's Rho is often only -1 or 1. Because players 2 and 8 shared the same seniority in their encounters, Spearman's Rho could not be calculated for the two games in which only players 2 and 8 faced off against each other. Across all the games Spearman's Rho has an average of 0.023 and a median of 0.17. In general, this approach shows virtually no association between seniority and signature ranking.

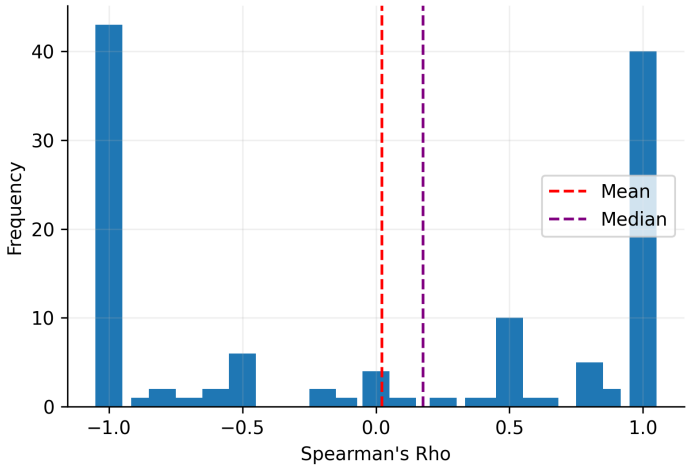

Figure 4: Bar graph of the frequency of different Spearman's Rho calculations between seniority and signature ranking for 132 of the 134 games.

2.3 Matrices to compare the players

Average win rate and net wins

One way to get a sense of the so-called higher or lower skill players is through matrices where the entry in row i and column j gives either the win rate or net wins of player i against player j. For example, row 7 and column 9 would give the percentage of games played between players 7 and 9 in which player 7 out-ranked player 9 (meaning player 7 won the interaction). Alternatively, row 7 and column 9 could give the number of those games in which player 7 won minus the number of those games in which player 9 won, yielding the net-win rate in favour of player 7.

These matrices are shown in Figures 5 and 6, however, the rows are sorted by the row average and row sum, respectively, to get a clear descending order of better to worse players. The row averages of Figure 5 correspond to average win rate the the row sum of Figure 6, correspond to net wins. Player 21 has the highest average win rate, with a record of 100% against the four players he or she encountered. Player 7 has the greatest net wins (33), i.e, the number of player 7's wins minus the number of player 7's loses is 33.

Association with seniority

The net wins of a player and his or her mean seniority had a negative correlation coefficient of -0.22, though this was not significant, with a p-value of 0.31. Additionally, the correlation coefficient between average win rate and mean seniority was 0.027 with a p-value of 0.90. A negative correlation coefficient would mean that seniority is associated with winning. This suggests that perhaps in games with more than two players (leading to large magnitudes for net wins), seniority is associated with winning. Indeed, returning to the Spearman's Rho from earlier, when games involving only two players are excluded (which can only result in -1 or 1), the average Spearman's Rho is 0.12 with a median of 0.38, a stronger correspondence than observed earlier. We conclude that in some cases, particularly in games with many players, seniority may be a light predictor of greater signature ranking (winning).

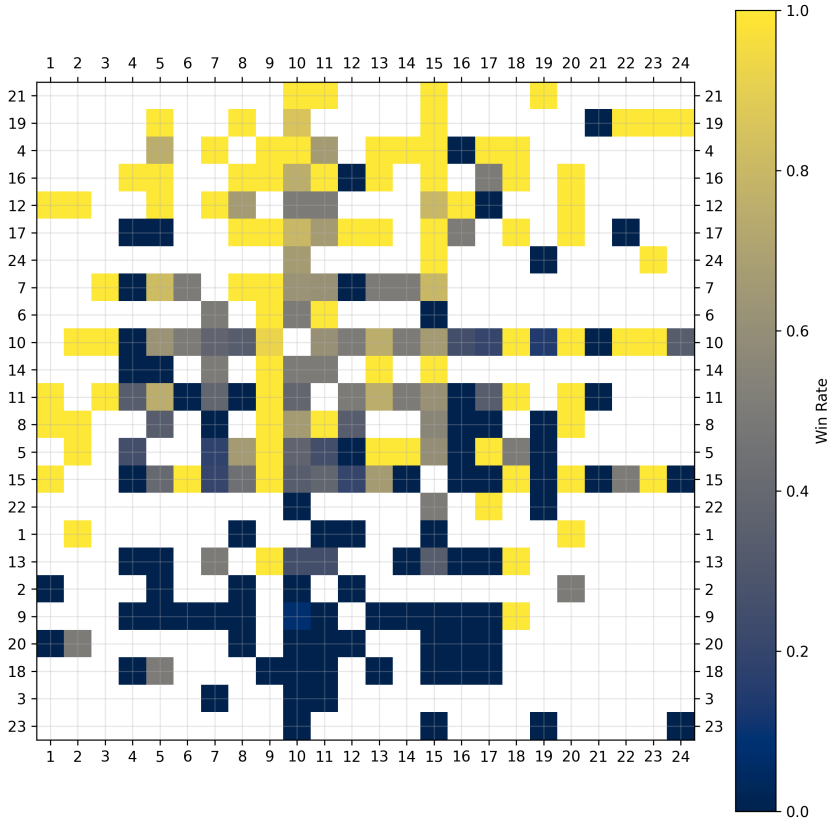

Figure 5: Matrix in which each entry gives the win rate in favour of the row-number for games played between row-number and column-number. The rows of the matrix are sorted by the row averages (average win rates).

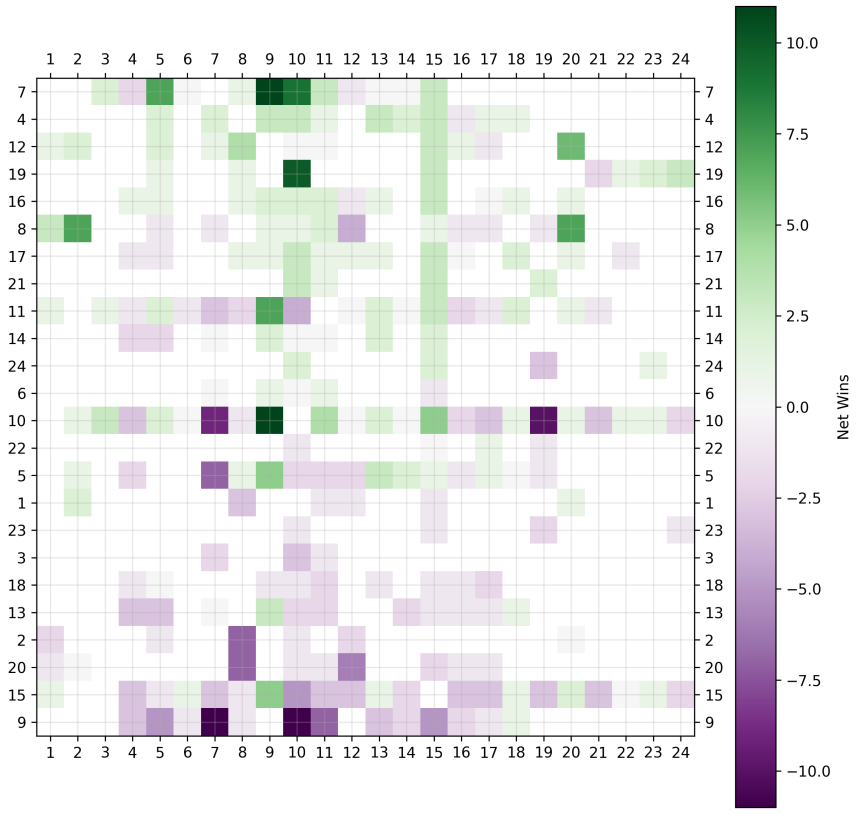

Figure 6: Matrix in which each entry gives the net wins in favour of the row-number for games played between row-number and column-number. The rows of the matrix are sorted by the row sums (net wins).

3 Modelling

3.1 Model selection

We would like to obtain the underlying skill-levels of the players that determine (albeit with some randomness) their signature ranking in a game. One way to do this is by using a Bayesian approach, in which a prior for the players' skills and the likelihood of the 134 game outcomes can be combined to obtain a posterior for the skills. However, we likely will not have a closed-form solution to the Bayes theorem equation combining the prior and the likelihood. In that case, one approach is to use MCMC, specifically Metropolis-Hastings, to sample from the posterior and thereby obtain the posterior distribution empirically.

Constructing a likelihood

A natural observation model (likelihood) is the Plackett-Luce model. In this model, each player $i = 1, ..., n$ has skill $\lambda_i \in \mathbb{R}$. For some game, the total number of players in that game is m, with the player ID numbers of that game being $i_1, ..., i_m$, where $1 \leq m \leq n$. If the set of all permutations of these m players is $\mathcal{P}_{i_1,...,i_m}$, then the outcome of the game $O \in \mathcal{P}_{i_1,...,i_m}$ is some random permutation $o = (o_1, ..., o_m)$, where $o \in \mathcal{P}_{i_1,...,i_m}$. With $\lambda = (\lambda_1, ..., \lambda_n)$, the probability for the random rank-outcome O to have the ranking o is:

$$Pr\{O = o|\lambda\} = \prod_{i=1}^{m} \frac{exp(\lambda_{o_i})}{\sum_{j=i}^{m} exp(\lambda_{o_j})}$$

Other covariates, such as seniority, can be included, where β is the vector of seniority parameters, $\beta = (\beta_1, ..., \beta_n)$. Then, the probability of the outcome-permutation is given by:

$$Pr\{O = o|\lambda, \beta\} = \prod_{i=1}^{m} \frac{exp(\lambda_{o_i} + \beta_{e_i})}{\sum_{j=i}^{m} exp(\lambda_{o_j} + \beta_{e_j})}$$

The index e_i is the seniority rank of player i at the time of the game.

The outcomes of the games can be assumed to be conditionally independent given the skill and seniority. Therefore, the joint probability of the outcome of all 134 games is given by the product of the probabilities of each individual game-outcome.

Constructing a prior and determining the seniority parameters

It should be noted that in the likelihood equation, when each element of λ and β is zero, the probability of any given permutation being the outcome of the game is $\frac{1}{m!}$. In other words, the likelihood distribution is uniform as there are $m!$ possible permutations and each is equally likely. We can therefore consider the skills and seniority parameters equaling zero as a state of perfect, unbiased ignorance.

In choosing the prior for the skill levels λ, we would like it to be centred at zero. One natural choice is a normal distribution with mean 0 and variance σ^2, where we set $\sigma^2 = 0.75$. This corresponds to our prior belief that the skills are closely concentrated around zero.

$$\lambda \sim \mathcal{N}(0, \sigma^2 I)$$

The skill levels of the players are assumed to be independent, which is why the covariance matrix is diagonal.

Furthermore, we would like the seniority parameters β to be on a similar scale to λ. In doing so, we can combine the seniority rankings in all of the games and obtain the overall mean and standard deviation (SD). Then, we centre and scale all seniority rankings to this mean and SD. We observed earlier that the seniority ranking has only a marginal correspondence to the signature ranking, therefore we set the variance of the β to in fact be one quarter the variance of the prior for λ (one half the SD). In other words, the variance of β is $0.75 \times 0.25 = 0.1875$. Then, the sum of the variances of the prior for λ and the seniority parameters β is 0.9375, close to one.

Constructing a proposal distribution

For the MCMC algorithm, we can create a proposal distribution that is also normal. Because some of the players played in very few games and against very few opponents, we may wish to

further regularize the skills to prevent overfitting. In other words, we can ensure that the skills are constantly fighting a tendency to return to zero from both the proposal distribution and the prior. This can be accomplished with a weight $w \in (0, 1]$, which tends to regress the skills towards zero. Note that if $w = 1$, then there is no such regularization and the proposal probabilities are always equal. From the current state λ, a proposal state λ' is generated as follows, with the random variable ϵ normally distributed:

$$\lambda' = wI\lambda + \epsilon$$

$$\epsilon \sim \mathcal{N}(0, d^2 I)$$

Note that the diagonal covariance matrix of ϵ means that the elements of λ are proposed independently of one another (given w).

What is therefore needed is to tune the parameters w and d to ensure that proposals are accepted at a reasonable rate (around ten to thirty percent) and the space of possible λ is explored well with relatively short burn-ins. Trial and error led to the parameters $w = 0.95$ and $d^2 = 0.05$ (corresponding to a SD of 0.22).

3.2 Model simulation

The MCMC simulation was run with 500,000 iterations over 46 minutes. The overall acceptance rate was 16%. The burn-in was reasonably rapid, occurring in less than 1,000 iterations for player 7 as shown in Figure 7. The tail 80% of the chain was used as the empirical distributions of λ, allowing ample time for burn-in.

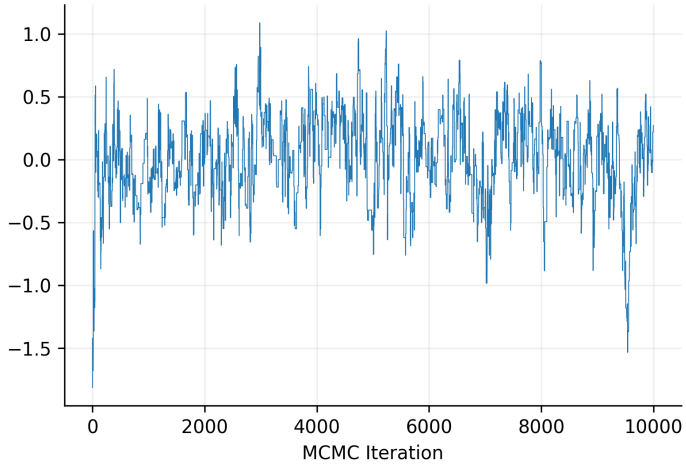

Figure 7: The first 10,000 MCMC states for the skill level of player 7.

The simulated distributions of each of the player's skills are shown in Figure 8. Note that as expected, the players who played in fewer games against fewer opponents had lower precision estimates (larger variances) as there was less data on these players. For instance, player 10 played in the most games (77) and faced off against the most opponents (22). Unsurprisingly, player 10 has a very narrowly distributed posterior skill distribution. In contrast, player 23, who played in only 3 games and against only 4 opponents, has a very wide skill estimate. Further comparisons

can be made using Figure 9. Another notable feature of the distributions is that they are each highly symmetric and even appear to be approximately normal.

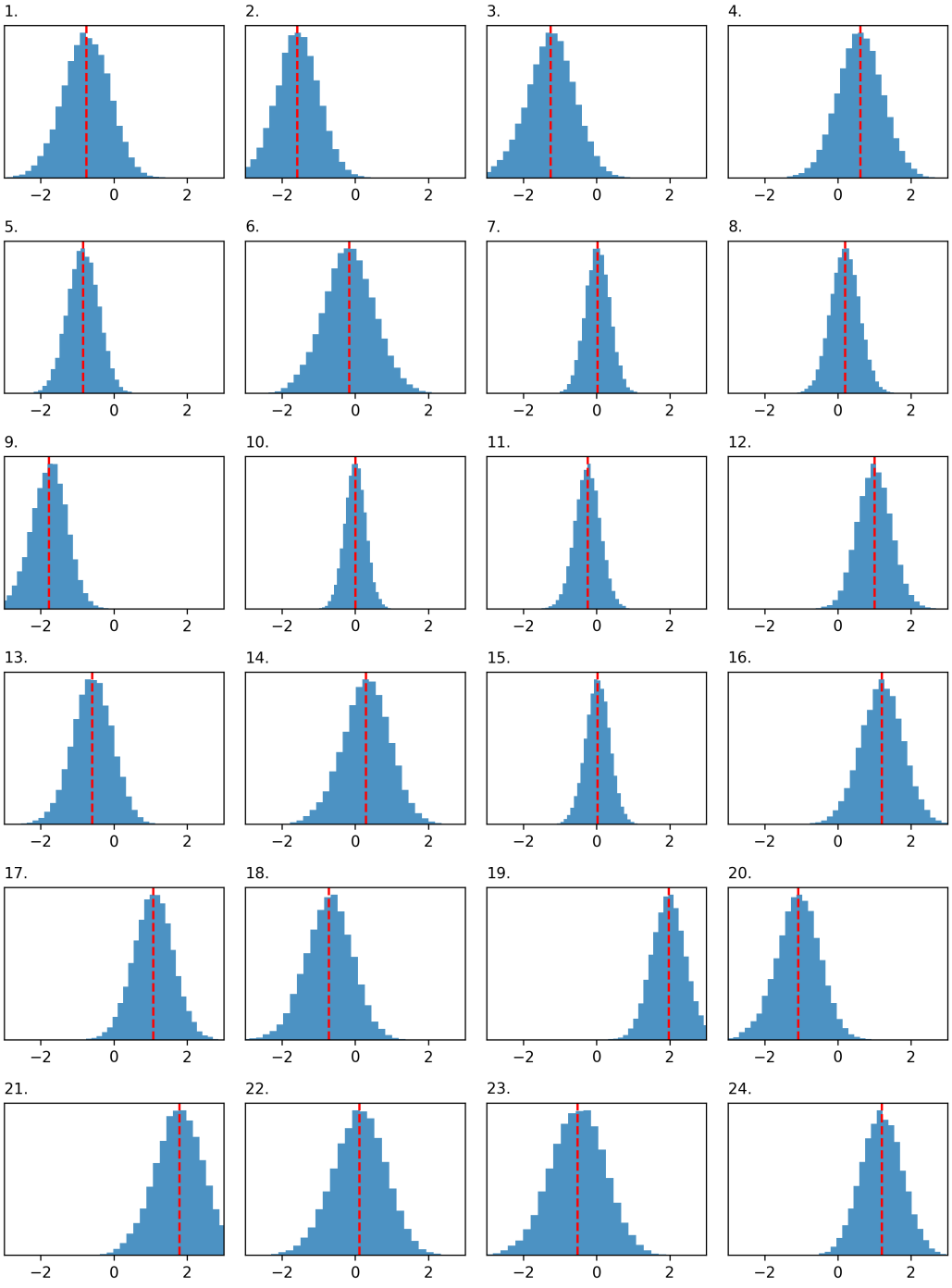

Figure 8: Distributions of the skill levels of all 24 players, with the sample means shown in dashed red.

The sample means of the posterior distributions for the skills are estimates of skill and can be used to compare the players, as shown in Figure 9. It should be noted that estimated skill has a positive correlation coefficient of 0.34 (p-value = 0.11) with mean seniority, indicating that greater estimated skill is associated with lower seniority. Because seniority largely appeared to have only a slight relationship with the game outcomes, this observation indicates that some of the estimated skill may be the result of effectively cancelling out or accounting for the largely random noise of the seniority covariate. In future work, it may therefore be desired to construct a model without the seniority estimates and compare use the estimated skills as another useful metric.

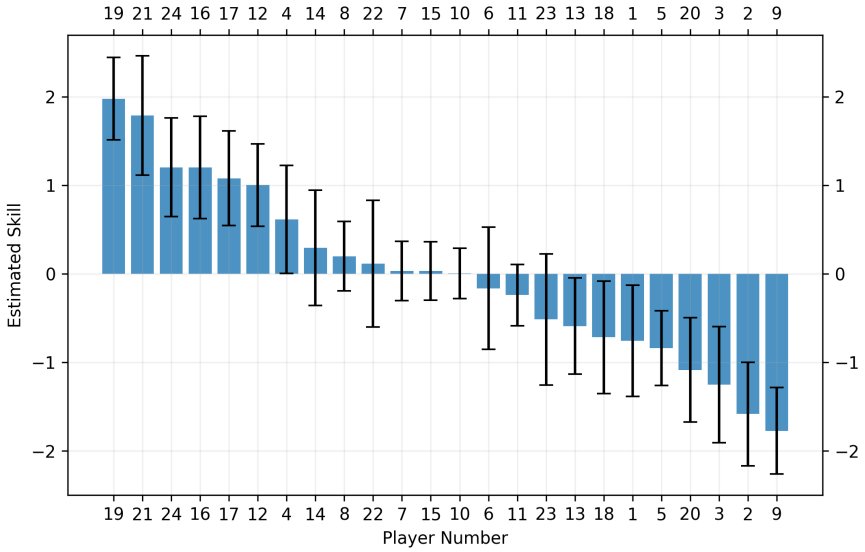

Figure 9: The sample means of the distribution of skill levels of the 24 players, with sample standard deviations provided by the error bars.

The skill estimates have strong positive and highly significant correlations with both the total net wins and average win rate discussed earlier (correlation coefficients of 0.70 and 0.88, respectively). Scatter plots of the skill estimates against these quantities are shown in Figure 10.

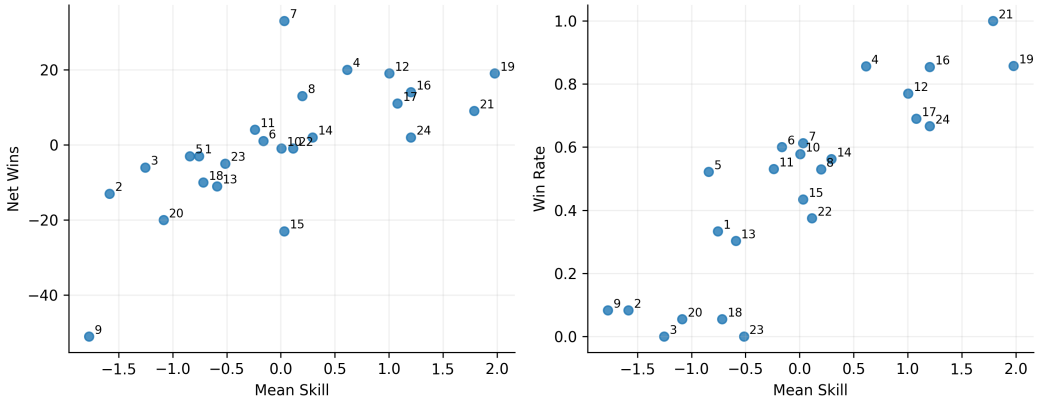

Figure 10: Scatter plots of the sample means of the skills of the 24 players against their total net wins and average win rate.

4 Predicting player 7's rank in game 68

If we suppose that game 68 was replayed, we may wish to obtain the probability that player 7 ranks first. Because there are nine players in game 68, there are 8! of 40,320 permutations in which player 7 comes in first. We can use the estimated skills for the likelihood model and sum over the probabilities of each of these permutations. The result is a probability of 0.18. This is greater than random chance, which would be 1/9 = 0.11.

We can consider the estimated skills of the players in game 68, which are shown in Figure 11. Player 7 has an above average skill-level among these players, which is substantially higher than the skills of players 5 and 9. Player 7 also has a very low mean ranking (2.7). It is therefore unsurprising that player 7 arrives in greater-than-random proportion of games. On the other hand, players 4 and 14 have notably higher skill-levels than player 7 and therefore might be expected to dominate the first-place entry.

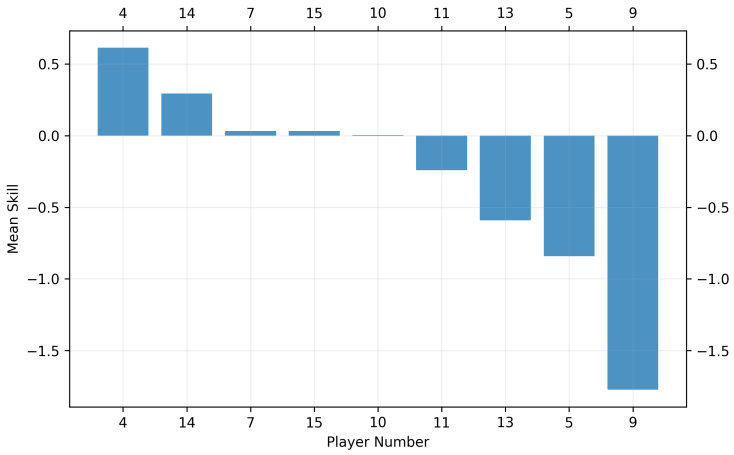

Figure 11: The sample means of the distribution of skill levels of the players who participated in game 68.

One potential issue with this approach is that the estimates were obtained (the model was trained) using game 68. Therefore, the data from game 68 is used both for training and prediction. In future work, it might be desired to exclude game 68 from the MCMC simulation and treat it as a hold-out set.

5 Conclusions

A Metropolis-Hastings algorithm was used to estimate social rank from data on the 134 signatures on legal documents between 24 individuals in the 12th century. A Plackett-Luce model was used for the observation model, providing the conditional probability of a given order of signatures on a legal document given the individuals' social ranks and ages. Independent normal distributions were used for the prior and proposal distributions. Estimates of the individuals' social ranks were obtained from their sample means in the MCMC distributions. These estimates were used to predict the probability that individual 7 would have the first signature in document 68 if that signing event were to occur again.

Python Code

```
import pandas as pd
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
import seaborn as sns
import itertools

### Data
games = [[[10, 7], [7, 3]],
 [[10, 7], [7, 3]],
 [[7, 10], [3, 7]],
 [[7, 10, 9], [3, 7, 11]],
 [[11, 13], [6, 12]],
 [[19, 10 ], [14, 4]],
 [[10, 7, 5], [7, 3, 5]],
 [ [7, 9, 10], [3, 10, 7]],
 [[6, 10, 7, 11, 9], [8, 7, 3, 6, 11]],
 [[10, 11], [7, 6]],
 [[21, 15], [10, 6]],
 [[7, 10], [3, 7]],
 [[10, 11], [6, 5]],
 [[12, 15], [7, 8]],
 [[7, 5], [3, 5]],
 [[15, 8], [12, 6]],
 [[15, 8], [9, 5]],
 [ [8, 15], [7, 13]],
 [[19, 23], [9, 13]],
 [[10, 11], [7, 6]],
 [[8, 15], [5, 9]],
 [[8, 5, 2], [7, 4, 7]],
 [ [10, 8, 2], [3, 4, 4]],
 [[7, 10], [3, 7]],
 [[10, 11, 15, 9], [6, 5, 13, 9]],
 [[15, 10], [9, 4]],
 [[24, 10], [13, 2]],
 [[10, 24], [2, 13]],
 [ [15, 11], [13, 5]],
 [[15, 23], [7, 13]],
 [[8, 1, 2], [7, 3, 7]],
 [[8, 12], [3, 5]],
 [[18, 5], [14, 4]],
 [ [19, 24, 10, 15], [8, 13, 2, 6]],
 [[10, 19, 24, 23], [2, 8, 13, 12]],
 [[12, 15, 5, 8], [10, 13, 4, 7]],
 [[15, 5], [13, 4]],
 [[7, 10], [3, 7]],
```

```
[[7, 11, 10, 3], [3, 6, 7, 1]],
[[10, 12], [2, 5]],
[[19, 10], [12, 4]],
[[24, 15], [13, 6]],
[[16, 11, 5, 4, 17, 15, 10, 13, 9, 18], [13, 4, 3, 1, 13, 12, 5, 9, 8, 13]],
[[10, 15, 22], [4, 9, 15]],
[[12, 8, 20], [5, 3, 9]],
[[12, 8, 20], [5, 3, 9]],
[[12, 8, 15, 20], [6, 4, 7, 10]],
[[12, 2], [5, 3]],
[[11, 10], [6, 7]],
[[11, 15, 10], [3, 9, 4]],
[[19, 10], [16, 5]],
[[7, 10], [3, 7]],
[[4, 5, 14, 13, 10, 11, 7, 15, 9], [1, 4, 12, 10, 6, 5, 2, 13, 9]],
[[10, 13], [6, 10]],
[[19, 24, 10], [8, 13, 2]],
[[7, 10], [3, 7]],
[[17, 12, 16, 8, 10, 15, 11, 20], [10, 8, 10, 5, 4, 9, 3, 14]],
[[19, 10 ], [13, 4]],
[[15, 11], [13, 5]],
[ [10, 7, 3], [7, 3, 1]],
[[12, 20], [8, 14]],
[[8, 12, 20], [5, 8, 14]],
[[12, 8], [8, 5]],
[[10, 15], [4, 9]],
[[10, 15], [4, 9]],
[[22, 17, 15], [14, 10, 9]],
[[8, 12], [8, 12]],
[[4, 7, 11, 5, 10, 14, 15, 13, 9], [1, 2, 5, 4, 6, 12, 13, 10, 9]],
[[24, 10], [13, 2]],
[[5, 8], [4, 7]],
[[17, 11, 18], [14, 5, 14]],
[[12, 8], [5, 3]],
[[19, 22], [8, 10]],
[[12, 2], [7, 5]],
[ [11, 15], [4, 12]],
[[17, 16, 10], [13, 13, 5]],
[[10, 16, 17], [5, 13, 13]],
[[7, 9], [3, 11]],
[[11, 10, 7], [5, 6, 2]],
[[7, 15, 10, 6], [3, 15, 7, 8]],
[[19, 10], [9, 3]],
[[15, 8, 11, 12, 1], [9, 5, 3, 8, 1]],
[[21, 11], [14, 3]],
[[4, 5], [1, 3]],
[[10, 7], [7, 3]],
[[19, 10], [8, 2]],
[ [12, 7, 5], [11, 3, 5]],
```

```
[[10, 7, 5], [7, 3, 5]],
[[15, 7, 11, 10], [14, 3, 6, 7]],
[[10, 11], [6, 5]],
[[10, 5, 9], [7, 5, 11]],
[[7, 10, 11], [2, 6, 5]],
[[10, 3], [7, 1]],
[[7, 8, 10, 9], [3, 8, 7, 11]],
[[8, 2], [8, 8 ]],
[[8, 20, 2], [4, 10, 4]],
[[8, 2], [5, 5]],
[[8, 15], [5, 10]],
[ [19, 10], [16, 5]],
[[12, 8], [8, 5]],
[[11, 7], [6, 3]],
[[11, 7], [5, 2]],
[[8, 1, 2, 20], [5, 1, 5, 14]],
[[7, 15], [2, 13]],
[[16, 15, 9], [10, 9, 7]],
[[21, 19, 10, 15], [14, 12, 4, 9]],
[[21, 19, 10, 15], [14, 12, 4, 9]],
[[11, 15], [5, 13]],
[[11, 7, 9], [6, 3, 11]],
[[10, 7], [6, 2]],
[ [21, 10], [15, 4]],
[[10, 7, 5], [7, 3, 5]],
[[7, 10], [3, 7]],
[[17, 10 ], [13, 5]],
[[10, 7], [7, 3]],
[[10, 7, 9], [7, 3, 10]],
[[5, 7], [4, 2]],
[[7, 5], [2, 4]],
[[10, 7], [6, 2]],
[[19, 8], [10, 5]],
[[10, 9], [7, 11]],
[[7, 10, 11, 9], [2, 6, 5, 9]],
[[7, 10], [2, 6]],
[[7, 10 ], [3, 7]],
[[7, 10], [3, 7]],
[[7, 10, 11], [3, 7, 6]],
[[7, 10, 5, 9], [3, 7, 5, 11]],
[[7, 10], [3, 7]],
[[7, 10], [3, 7]],
[[11, 15], [3, 9]],
[ [19, 5], [13, 2]],
[[19, 10], [13, 4]],
[[10, 19], [4, 14]],
[[7, 11, 5], [3, 6, 5]]]
```

```
### Constants

alpha_val = 0.2

save = True
fp = '../plots/'
end = '.png'
dpi = 300

### Object orientation

nums = np.arange(1, 25)
n_players = 24

viridis = sns.color_palette('viridis', n_players)

class Members:
    def __init__(self, num):
        self.num = num
        self.color = viridis[num-1]

        # Games
        self.games = list()
        for game in games:
            if self.num in game[0]:
                self.games.append(game)


        # NumGames
        self.num_games = len(self.games)


        # Opponents
        self.opponents = set([item for sublist in [
                            game[0] for game in self.games
                                ] for item in sublist])
        self.opponents.remove(self.num)

        # NumOpponents
        self.num_opponents = len(self.opponents)


        # Times (game number, e.g. 68)
        self.times = list()
        for j in range(len(games)):
            game = games[j]
            if self.num in game[0]:
                self.times.append(j+1)
        self.times = np.array(self.times)
```

```
    # Seniority
    self.seniority = list()
    for game in self.games:
        self.seniority.append(game[1][game[0].index(self.num)])
    self.seniority = np.array(self.seniority)


    # Wins and Loses
    self.wins = {i:0 for i in nums}
    self.loses = {i:0 for i in nums}
    for i in self.opponents:
        for game in self.games:
            if i in game[0]:
                if game[0].index(num) < game[0].index(i):
                    self.wins[i] += 1
                else:
                    assert game[0].index(num) > game[0].index(i)
                    self.loses[i] += 1

    # NumWins and NumLoses
    self.num_wins = sum(self.wins.values())
    self.num_loses = sum(self.loses.values())

    # NumInteractions
    self.num_interactions = self.num_wins + self.num_loses

 self.num_interactions_per_opponent = self.num_interactions / self.num_opponents

    # Overall Win Percentage
    self.win_overall = self.num_wins / self.num_interactions

    # Net Wins
    self.net_wins = self.num_wins - self.num_loses

    # Avg Win Per Opponent
    self.avg_win = np.array([self.wins[i] / (self.wins[i] + self.loses[i]
     ) for i in self.opponents]).mean()




def get_info(self):
    print('Games [o, e]')
    print(self.games)
    print()
    print('Number of Games')
    print(self.num_games)
```

```
        print()
        print('Times (game index numbers)')
        print(list(self.times))
        print()
        print('Seniority (seniority values in each of the games)')
        print(list(self.seniority))
        print()
        print('Opponents')
        print(list(self.opponents))
        print()
        print('Wins to (member, this many times)')
        print([(i, self.wins[i]) for i in self.opponents])
        print()
        print('Loses to (member, this many times)')
        print([(i, self.loses[i]) for i in self.opponents])

mems = {i:Members(i) for i in nums}

mems[1].get_info()

### Number of players in each game

game_counts = np.array([len(game[0]) for game in games])
game_sizes = np.unique(game_counts, return_counts=True)

fig, ax = plt.subplots(1, 1)
ax.bar(game_sizes[0], game_sizes[1])
# ax.set_xticks()
ax.set(xticks = np.arange(game_sizes[0].min(), game_sizes[0].max()+1),
      xlabel='Number of Players in the Game', ylabel='Frequency')

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

ax.grid(alpha=alpha_val)
if save:
    plt.savefig(fp + 'num_mem' + end, dpi=dpi)
plt.show()

game_sizes

game_sizes[1][0] / game_sizes[1].sum()

# Mean
game_counts.mean()

# Median
```

```
np.quantile(game_counts, 0.5)

# IQR
st.iqr(game_counts)

### Number of games

num_games = np.array(
[mems[i].num_games for i in nums])

num_games2 = np.array(list(zip(nums, num_games)))

pd.Series(num_games, index=nums).sort_values(ascending=False).head()

pd.Series(num_games, index=nums).sort_values(ascending=False).tail(10)

num_games.mean()

np.quantile(num_games, [0.25, 0.5, 0.75])

st.iqr(num_games)

### Number of opponents

num_opponents = np.array(
[mems[i].num_opponents for i in nums])

pd.Series(num_opponents, index=nums).sort_values(ascending=False).head(10)

pd.Series(num_opponents, index=nums).sort_values(ascending=False).tail(10)

num_opponents.mean()

np.quantile(num_opponents, 0.5)

st.iqr(num_opponents)

#### Number of games and opponents

fig, axes = plt.subplots(1, 2, figsize=(10, 4))

ax = axes[0]

ax.bar(nums, num_games)
ax.set_xlabel('Player Number')
ax.set_ylabel('Number of Games Played')
ax.axhline(y=num_games.mean(),
color='red', linestyle='--', label='Mean')
ax.axhline(y=np.quantile(num_games, 0.5),
```

```
color='purple', linestyle='--', label='Median')
ax.legend(loc='upper left')
ax.grid(alpha=0.3)
ax.set_xticks(np.arange(1, 25, 3))

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

ax = axes[1]


ax.bar(nums, num_opponents)
ax.set_xlabel('Player Number')
ax.set_ylabel('Number of Opponents')
ax.axhline(y=num_opponents.mean(),
color='red', linestyle='--', label='Mean')
ax.axhline(y=np.quantile(num_opponents, 0.5),
color='purple', linestyle='--', label='Median')
ax.legend(loc='upper left')
ax.grid(alpha=0.3)
ax.set_xticks(np.arange(1, 25, 3))

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

plt.tight_layout()

if save:
    plt.savefig(fp + 'bar_games' + end, dpi=dpi)
plt.show()

### Number of interactions/games per opponent

num_interactions_per_opponent = np.array(
[mems[i].num_interactions_per_opponent for i in nums])

pd.Series(num_interactions_per_opponent, index=nums
).sort_values(ascending=False).head()

num_interactions_per_opponent.mean()

np.quantile(num_interactions_per_opponent, 0.5)
```

```
st.iqr(num_interactions_per_opponent)

plt.bar(nums, num_interactions_per_opponent)
plt.xlabel('Player Number')
plt.ylabel('Number of Games per Opponent')
plt.axhline(y=num_interactions_per_opponent.mean(),
color='red', linestyle='--', label='Mean')
plt.axhline(y=np.quantile(num_interactions_per_opponent, 0.5),
color='purple', linestyle='--', label='Median')
plt.legend(loc='upper left')
plt.grid(alpha=0.3)
plt.xticks(np.arange(1, 25, 3))

ax = plt.gca()
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

plt.show()

#### Seniority and member number

mean_sens = np.array([mems[i].seniority.mean() for i in nums])
median_sens = np.array([np.quantile(mems[i].seniority, 0.5) for i in nums])

plt.scatter(nums, mean_sens)
plt.xlabel('Player Number')
plt.ylabel('Mean Seniority')
plt.grid(alpha=0.3)
plt.xticks(np.arange(1, 25, 3))

ax = plt.gca()
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

plt.show()

st.pearsonr(nums, mean_sens)

### Number of interactions

num_interactions = np.array([mems[i].num_interactions for i in nums])

pd.Series(num_interactions, index=nums).sort_values(ascending=False).head()
```

```
num_interactions.mean()

np.quantile(num_interactions, 0.5)

st.iqr(num_interactions)

plt.bar(nums, num_interactions)
plt.xlabel('Player Number')
plt.ylabel('Number of Interactions')
plt.axhline(y=num_interactions.mean(), color='red', linestyle='--', label='Mean')
plt.axhline(y=np.quantile(num_interactions, 0.5),
color='purple', linestyle='--', label='Median')
plt.legend(loc='upper left')
plt.grid(alpha=0.3)
plt.xticks(np.arange(1, 25, 3))

ax = plt.gca()
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

plt.show()

### Movement in seniority between games

alpha_vird = 1
num_thres = 20
print((num_games < num_thres).sum())

sens_change = np.zeros(n_players)

for i in nums:
    temp_sens = mems[i].seniority
    sens_change[i-1] = temp_sens[-1] - temp_sens[0]

np.mean(np.abs(sens_change))

np.median(np.abs(sens_change))

st.iqr(np.abs(sens_change))

fig, axes = plt.subplots(1, 2, figsize=(10, 4))

ax = axes[0]

for i in nums:
```

```
    if mems[i].num_games < num_thres:
        ax.plot(np.arange(1, mems[i].num_games+1),
                mems[i].seniority, color=mems[i].color, alpha=alpha_vird)

ax.set_ylabel('Seniority')
ax.set_xlabel('Game number')

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

ax.set_xticks(np.arange(1, 18, 2))


ax = axes[1]

for i in nums:
    if mems[i].num_games < num_thres:
        ax.plot(np.arange(1, mems[i].num_games+1),
                mems[i].seniority - mems[i].seniority[0],
                 color=mems[i].color, alpha=alpha_vird, linestyle='-')

ax.set_ylabel('Change in Seniority')
ax.set_xlabel('Game number')

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

ax.set_xticks(np.arange(1, 18, 2))

plt.tight_layout()

if save:
    plt.savefig(fp + 'sen_move' + end, dpi=dpi)

plt.show()

fig, axes = plt.subplots(1, 2, figsize=(10, 4))

ax = axes[0]

for i in nums:
    if mems[i].num_games >= 20:
        ax.plot(np.arange(1, mems[i].num_games+1),
```

```
                mems[i].seniority, color=mems[i].color, alpha=alpha_vird)

ax.set_ylabel('Seniority')
ax.set_xlabel('Game number')

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

ax.set_xticks(np.arange(1, 18, 2))


ax = axes[1]

for i in nums:
    if mems[i].num_games < num_thres:
        ax.plot(np.arange(1, mems[i].num_games+1),
                mems[i].seniority - mems[i].seniority[0],
                 color=mems[i].color, alpha=alpha_vird, linestyle='-')

ax.set_ylabel('Change in Seniority')
ax.set_xlabel('Game number')

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

ax.set_xticks(np.arange(1, 18, 2))

plt.tight_layout()


plt.show()

#### Interactions between 2 and 8

mems[2].num_games

mems[2].games

### Correlate

def get_rs(game):
    rank = list(range(1, len(game[0]) + 1))
```

```
    sen = np.array(game[1])

    return rank, sen

new = [get_rs(game) for game in games]

spear_game = np.array([st.spearmanr(game[0], game[1]) for game in new])

spear_v = spear_game[:, 0]

spear = np.unique(spear_v, return_counts=True)

spear

np.nanmean(spear_v)

np.nanquantile(spear_v, 0.5)

np.nanquantile(spear_v, 0.499)

spear2 = spear[0][2:-4], spear[1][2:-4]



spear_v2 = spear_v.copy()

spear_v2 = spear_v[game_counts != 2]

np.mean(spear_v2)

np.median(spear_v2)

spear2 = np.unique(spear_v2, return_counts=True)

plt.bar(spear2[0], spear2[1], width=0.1)




fig, ax = plt.subplots(1, 1)
ax.bar(spear[0], spear[1], width=0.1)

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

ax.grid(alpha=alpha_val)
```

```
ax.set(xlabel='Spearman\'s Rho', ylabel='Frequency')
ax.axvline(x=np.nanmean(spear_v), color='red', linestyle='--', label='Mean')
ax.axvline(x=np.nanquantile(spear_v, 0.5),
color='purple', linestyle='--', label='Median')
plt.legend(loc='center right')

if save:
    plt.savefig(fp + 'spearman' + end, dpi=dpi)

plt.show()

### Create member matrix

matrix = np.zeros((n_players, n_players))
matrix2 = np.zeros((n_players, n_players))

for i in nums:
    for j in nums:
        # if i > j:
        if j in mems[i].opponents:
        matrix[i-1, j-1] = mems[i].wins[j] / (mems[i].wins[j] + mems[i].loses[j])
            matrix2[i-1, j-1] = mems[i].wins[j] - mems[i].loses[j]
        else:
            matrix[i-1, j-1] = np.nan
            matrix2[i-1, j-1] = np.nan
#         else:
#             matrix[i-1, j-1] = np.nan
#             matrix2[i-1, j-1] = np.nan

fig, ax = plt.subplots(1, 1, figsize=(10, 10))
im = ax.imshow(matrix, cmap='cividis')
plt.colorbar(im, ax=ax)
ax.set_xticks(np.arange(n_players))
ax.set_xticklabels(nums)
ax.set_yticks(np.arange(n_players))
ax.set_yticklabels(nums)
ax.tick_params(axis="x", bottom=True, top=True, labelbottom=True, labeltop=True)
ax.tick_params(axis="y", left=True, right=True, labelleft=True, labelright=True)
ax.grid(alpha=0.5)
plt.show()

fig, ax = plt.subplots(1, 1, figsize=(10, 10))
im = ax.imshow(matrix2, cmap='PRGn')
plt.colorbar(im, ax=ax)
ax.set_xticks(np.arange(n_players))
ax.set_xticklabels(nums)
ax.set_yticks(np.arange(n_players))
ax.set_yticklabels(nums)
```

```
ax.tick_params(axis="x", bottom=True, top=True, labelbottom=True, labeltop=True)
ax.tick_params(axis="y", left=True, right=True, labelleft=True, labelright=True)
ax.grid(alpha=0.5)
plt.show()

avg_wins = np.nanmean(matrix, axis=1)

avg_wins

np.array([mems[i].avg_win for i in nums])

assert np.allclose(avg_wins, np.array([mems[i].avg_win for i in nums]))

morder = np.argsort(avg_wins)[::-1]

nums_matrix_sort = nums[morder]

matrix_sort = matrix[morder]

net_wins = np.nansum(matrix2, axis=1)

assert (net_wins == np.array([mems[i].net_wins for i in nums])).all()

net_wins

m2order = np.argsort(net_wins)[::-1]

nums_matrix2_sort = nums[m2order]

matrix2_sort = matrix2[m2order]

fig, ax = plt.subplots(1, 1, figsize=(10, 10))
im = ax.imshow(matrix_sort, cmap='cividis')
plt.colorbar(im, ax=ax, label='Win Rate')
ax.set_xticks(np.arange(n_players))
ax.set_xticklabels(nums)
ax.set_yticks(np.arange(n_players))
ax.set_yticklabels(nums_matrix_sort)
ax.tick_params(axis="x", bottom=True, top=True, labelbottom=True, labeltop=True)
ax.tick_params(axis="y", left=True, right=True, labelleft=True, labelright=True)
ax.grid(alpha=0.3)
if save:
    plt.savefig(fp + 'matrix_avg' + end, dpi=dpi)
plt.show()

fig, ax = plt.subplots(1, 1, figsize=(10, 10))
im = ax.imshow(matrix2_sort, cmap='PRGn')
plt.colorbar(im, ax=ax, label='Net Wins')
ax.set_xticks(np.arange(n_players))
```

```
ax.set_xticklabels(nums)
ax.set_yticks(np.arange(n_players))
ax.set_yticklabels(nums_matrix2_sort)
ax.tick_params(axis="x", bottom=True, top=True, labelbottom=True, labeltop=True)
ax.tick_params(axis="y", left=True, right=True, labelleft=True, labelright=True)
ax.grid(alpha=0.3)
if save:
    plt.savefig(fp + 'matrix_net' + end, dpi=dpi)
plt.show()

plt.scatter(avg_wins, net_wins);

st.pearsonr(avg_wins, net_wins)

plt.scatter(mean_sens, net_wins);

st.pearsonr(mean_sens, net_wins)

plt.scatter(mean_sens, avg_wins);

st.pearsonr(mean_sens, avg_wins)



### Results

### Data manipulation
os = [np.array(game[0]) for game in games]
es = [np.array(game[1]) for game in games]

ois = [o-1 for o in os]  # The `os` for indexing in Python

def flatten(array):
    return np.array([item for sublist in array for item in sublist])

def get_beta(beta_sd):
    """Standardize and scale the e vector to betas.
        We need to multiply by negative one because low
        seniorities are considered better."""
   beta = [-1 * (e - flatten(es).mean()) * beta_sd / flatten(es).std() for e in es]
    assert np.allclose(flatten(beta).std(), beta_sd)
    return beta


### Constants
n = len(games)
n_players = 24
```

```
### Likelihood functions
def get_like(lam):
    """Calculate the likelihood using the likelihood equation."""
    gamma = 0
    for i in range(n):
        gamma += get_gamma(lam[ois[i]], beta[i])
    return np.exp(gamma)

def get_gamma(lamv, betav):
    """Get gamma."""
    return get_plus(lamv, betav) - get_minus(lamv, betav)

def get_minus(lamv, betav):
    """Get the subtracted part of gamma."""
    return np.sum(np.log(
                np.sum(axis=1,
                   a=np.triu(
                        np.tile(
                     np.exp(lamv + betav), lamv.size).reshape(lamv.size, -1)))))

def get_plus(lamv, betav):
    """Get the added part of gamma."""
    return np.sum(lamv + betav)


### Prior and posterior functions
def get_prior(lamx):
    return st.norm.pdf(lamx, loc=0, scale=prior_sd).prod()

def get_post(lamx):
    return get_like(lamx) * get_prior(lamx)


### Random variable funcitons
def init_lam():
    return np.random.normal(loc=0, scale=prior_sd, size=n_players)

def get_epsilon():
    return np.random.normal(loc=0, scale=d, size=n_players)


### Weight functions
def get_loc(lamx):
    return weight*lamx

def get_q(to_lam, from_lam):
    return st.norm.pdf(to_lam, loc=get_loc(from_lam), scale=d).prod()

### Parameters
```

```
prior_var = 0.75
beta_var_ratio = 0.25

prior_sd = np.sqrt(prior_var)
beta_sd = np.sqrt(prior_var * beta_var_ratio)

beta = get_beta(beta_sd)

d2 = 0.05
d = np.sqrt(d2)

total_var = prior_var * (1 + beta_var_ratio)

weight = 0.95
use_weight = True

n_iter = 500_000

### Initialize
lams = np.zeros((n_iter, n_players))
acc = np.zeros(n_iter, dtype=np.int8)

lams[0] = init_lam()
last_post = get_post(lams[0])

%%time

### Run MCMC
for i in range(1, n_iter):

    # Propose lambda and get its posterior
    if use_weight:
        lam2 = weight * lams[i-1] + get_epsilon()
    else:
        lam2 = lams[i-1] + get_epsilon()

    lam2_post = get_post(lam2)

    if use_weight:
        # Get conditional probabilities (to . from .)
        q21 = get_q(to_lam=lam2, from_lam=lams[i-1])
        q12 = get_q(to_lam=lams[i-1], from_lam=lam2)

        # Get ratio
        r = np.divide(lam2_post * q12, last_post * q21)
    else:
        r = np.divide(lam2_post, last_post)

    # Flip coin
```

```
    flip = np.random.binomial(n=1, p=min(r, 1))

    # If coin lands heads (acceptance)
    if flip == 1:
        # Update the last posterior
        last_post = lam2_post
        # Recalculate the total number of acceptances
        acc[i] = 1
        # Insert new lambda
        lams[i] = lam2

    # If coin lands tails (rejection)
    else:
        lams[i] = lams[i-1]

# np.savetxt('lams2.txt', lams)

# np.savetxt('acc2.txt', acc)

prior_var * beta_var_ratio

beta_sd / prior_sd

total_var

d

lams = np.loadtxt('lams2.txt')

acc = np.loadtxt('acc2.txt')

iters = np.arange(n_iter)

accsum = np.cumsum(acc)

acc_rate = acc.sum() / (n_iter-1)

n_show = 10_000

acc.sum()

acc_rate

lamsw = lams[-n_show:]
accw = acc[-n_show:]
itersw = iters[-n_show:]

lamswb = lams[:n_show]
accwb = acc[:n_show]
```

```
iterswb = iters[:n_show]

nums = np.arange(1, 25)

plt.plot(iterswb, lamswb[:,6],
         linewidth=0.5)


ax = plt.gca()
ax.set(xlabel='MCMC Iteration')
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

if save:
    plt.savefig(fp + 'burnin' + end, dpi=dpi)

plt.show()

plt.plot(itersw, lamsw[:,np.random.randint(n_players)],
         linewidth=0.5);

prop = np.int(n_iter * 0.8)

lamst = lams[-prop:]
acct = acc[-prop:]
iterst = iters[-prop:]

means = np.mean(lamst, axis=0)

nums_sort = nums[np.argsort(means)][::-1]

means_sort = np.sort(means)[::-1]

variances = np.var(lamst, axis=0, ddof=1)

variances_sort = variances[np.argsort(means)][::-1]



fig, ax = plt.subplots(1, 1, figsize=(8, 5))
ax.bar(nums, means_sort, yerr=np.sqrt(variances_sort),
       capsize=4, alpha=0.8, ecolor='black')
ax.set_xticklabels(nums_sort)
ax.set_xticks(nums)
ax.grid(alpha=alpha_val)
ax.tick_params(axis="x", bottom=True, top=True, labelbottom=True, labeltop=True)
```

```
ax.tick_params(axis="y", left=True, right=True, labelleft=True, labelright=True)
ax.set(ylabel='Estimated Skill', xlabel='Player Number')

if save:
    plt.savefig(fp + 'skills' + end, dpi=dpi)

plt.show()

fig, axes = plt.subplots(1, 2, figsize=(10, 4))

ax = axes[0]

ax.scatter(means, net_wins, alpha=0.8)
ax.set(ylabel='Net Wins', xlabel='Mean Skill')
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

texts = [ax.text(0.05 + means[i-1], 0.05 + net_wins[i-1],
            i, ha='left', va='bottom', fontsize=8)
                 for i in nums]
#adjust_text(texts)


ax = axes[1]
ax.scatter(means, avg_wins, alpha=0.8)
ax.set(ylabel='Win Rate', xlabel='Mean Skill')
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
ax.grid(alpha=alpha_val)

texts = [ax.text(0.05 + means[i-1], 0 + avg_wins[i-1],
            i, ha='left', va='bottom', fontsize=8)
                 for i in nums]

plt.tight_layout()

if save:
    plt.savefig(fp + 'skill_wins' + end, dpi=dpi)

plt.show()

st.pearsonr(net_wins, means)

st.pearsonr(avg_wins, means)
```

```
#### Means versus seniorities

plt.scatter(mean_sens, means);

st.pearsonr(mean_sens, means)

plt.scatter(median_sens, means);

st.pearsonr(median_sens, means)

fig, axes = plt.subplots(6, 4, figsize=(9, 12))

for i in range(n_players):
    ax = axes.ravel()[i]
    ax.hist(lamst[:, i], density=True, bins=30, alpha=0.8)
    ax.set(yticks=[], xlim=(-3, 3))
    ax.axvline(x=means[i], color='red', linestyle='--')
    ax.set_title('{}.'.format(i+1), fontsize=10, loc='left')

plt.tight_layout()

if save:
    plt.savefig(fp + 'dists' + end, dpi=dpi)

plt.show()

### Replay game 68

oiv = ois[67]
betav = beta[67]

oivp = np.array(list(
                itertools.permutations(oiv[oiv != 6])))
betap = np.array(list(
                itertools.permutations(betav[oiv != 6])))

assert oivp.shape[0] == betap.shape[0] == np.math.factorial(
        np.size(oiv[oiv != 6])) == 40320

n_perm = 40320

sixes = np.repeat(6, n_perm)
beta_sixes = np.repeat(betav[oiv == 6][0], n_perm)

oivp = np.column_stack((sixes, oivp))

betap = np.column_stack((beta_sixes, betap))
```

```
def get_likep():
    """Calculate the likelihood using the likelihood equation."""
    gamma = 0
    for i in range(n_perm):
        gamma += np.exp(get_gamma(means[oivp[i]], betap[i]))
    return gamma

np.size(oiv)

get_likep()

1 / np.size(oiv)

idxv = oiv[np.argsort(means[oiv])[::-1]]+1
lamv = means[oiv][np.argsort(means[oiv])][::-1]
numv = np.arange(1, np.size(oiv)+1)

fig, ax = plt.subplots(1, 1, figsize=(8, 5))
ax.bar(numv, lamv, alpha=0.8)

ax.set_xticklabels(idxv)
ax.set_xticks(numv)
ax.grid(alpha=alpha_val)
ax.tick_params(axis="x", bottom=True, top=True, labelbottom=True, labeltop=True)
ax.tick_params(axis="y", left=True, right=True, labelleft=True, labelright=True)

ax.set(ylabel='Mean Skill', xlabel='Player Number')

if save:
    plt.savefig(fp + 'game68' + end, dpi=dpi)

plt.show()


#### Testing

beta[:5]

ois[:5]

games[:5]

# Test the get_plus and get_minus functions
test_true = (np.exp(-2) / sum([np.exp(-2)])) * (np.exp(1) / sum([np.exp(1), np.exp(-2)]))

test_lamv = np.array([1, -2])
test_betav = np.array([0, 0])

test_calc = np.exp(get_plus(test_lamv, test_betav) - get_minus(test_lamv, test_betav))
```

```
print(test_calc)
print(test_true)

assert np.allclose(test_true, test_calc)
```