4F13 Coursework 2: Probabilistic Ranking

Word Count: 999

November 18, 2022

Problem a)

I run Gibbs sampling for 2100 iterations. Then I select the first 8 players and generate traceplots for all 2100 iterations and the first 100 iterations:

```
# select the first 8 members and plot their traceplots for all 2100
                                                iterations
player_index = [0,1,2,3,4,5,6,7]
fig, axs = plt.subplots(4,2,figsize=(20,20))
for i in range(4):
    for j in range(2):
        axs[i,j].plot(skill_samples[player_index[2*i+j],:],color='C'+str(2*i
                                                  +j))
        axs[i,j].set_title(W[player_index[2*i+j]][0])
        axs[i,j].set_ylabel("w")
        axs[i,j].set_xlabel("iteration")
```

```
# Code for plotting first 100 iteration
fig, axs = plt.subplots(4,2,figsize=(20,20))
for i in range(4):
    for j in range(2):
        axs[i,j].plot(skill_samples[player_index[2*i+j],:][0:101],color='C'+
                                                  str(2*i+j))
        axs[i,j].set_title(W[player_index[2*i+j]][0])
        axs[i,j].set_ylabel("w")
        axs[i,j].set_xlabel("iteration")
```

and the auto-correlation plot:

```
for n,p in enumerate(player_index):
    autocor = np.zeros(100)
    for i in range(100):
        autocor[i]=pandas.Series.autocorr(pandas.Series(skill_samples[p,:]),
                                                  lag=i)
    plt.plot(autocor,label=W[player_index[n]][0])
plt.legend(list(W[list(player_index)].reshape(-1,)))
```

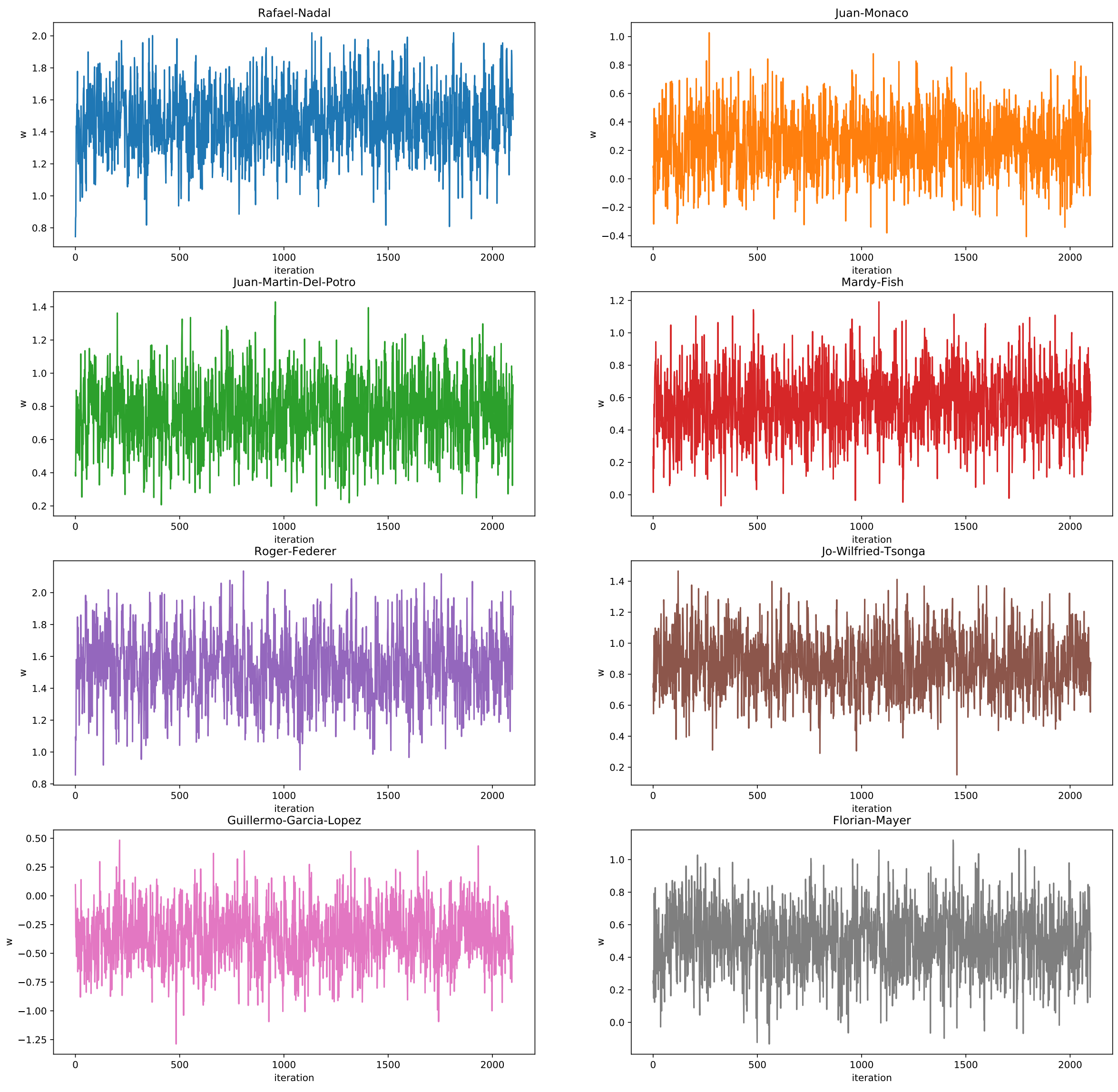

Figure 1: 2100 iterations of Gibbs sampling traceplots of the first 8 players on the list

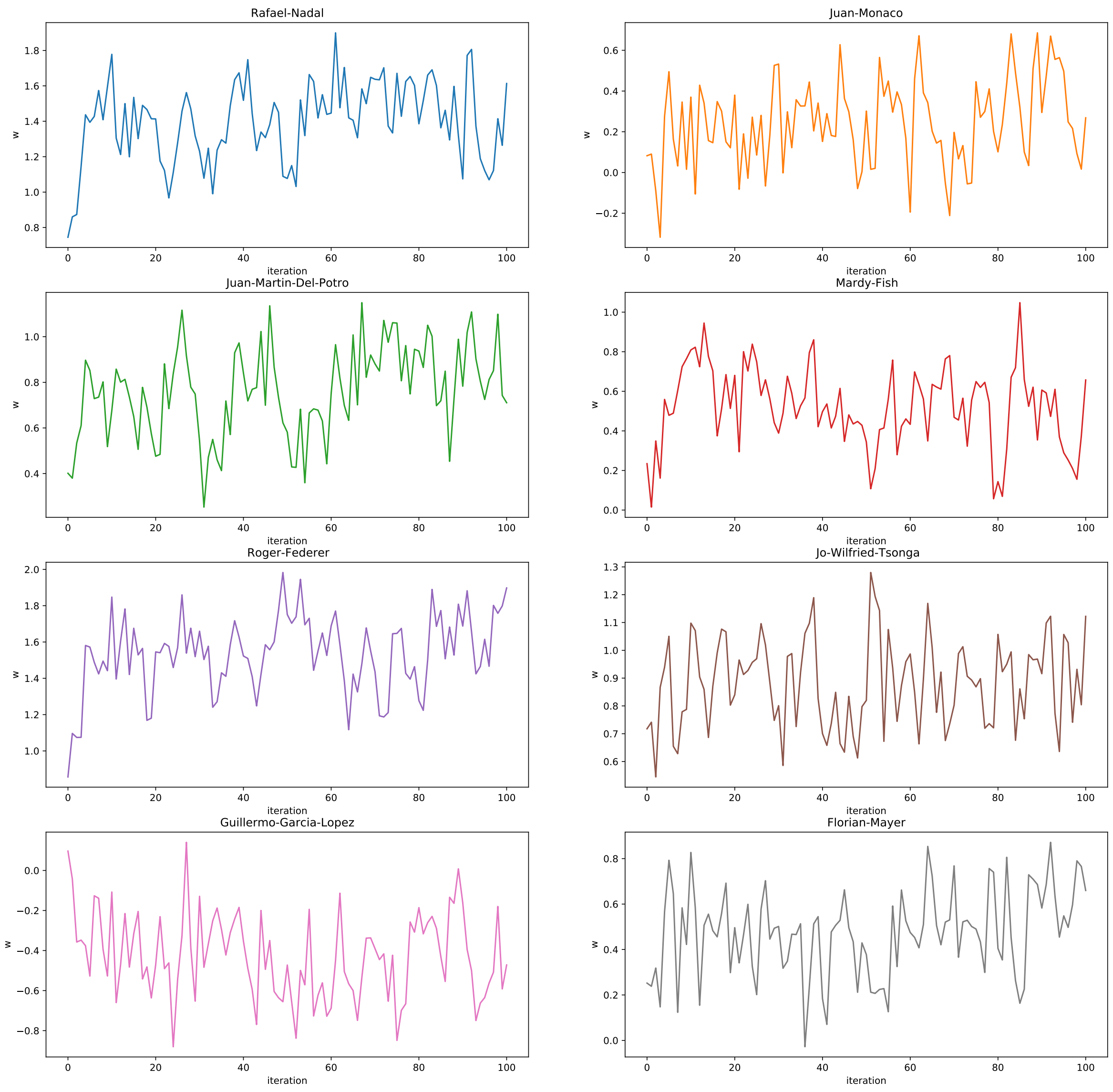

Figure 2: first 100 iterations of Gibbs sampling traceplots of the first 8 players on the list

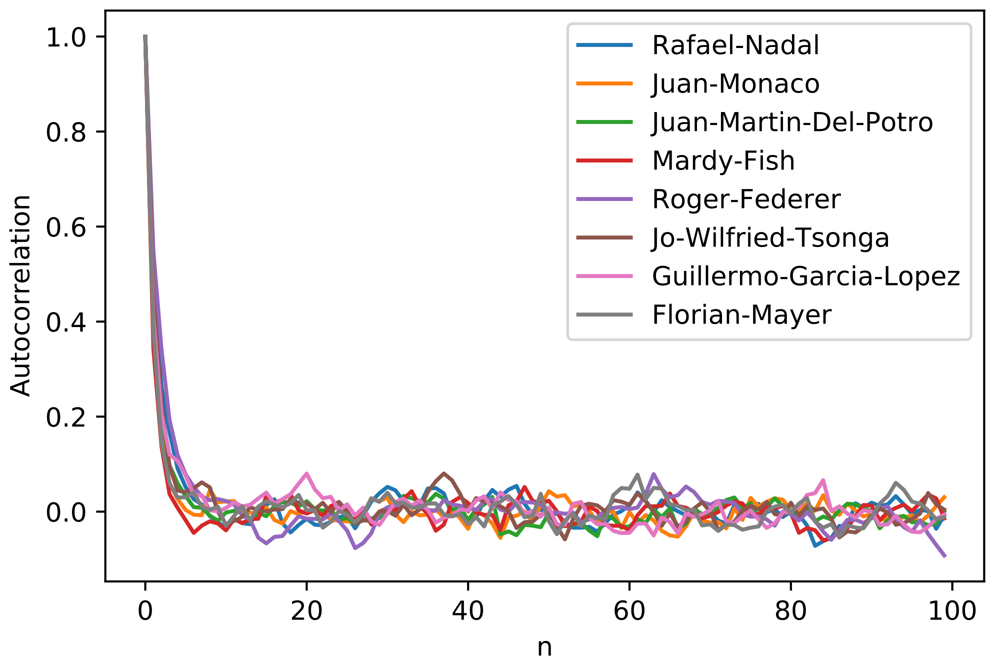

Figure 3: Auto-correlation plot

From Figure 2, the chains for Nadal and Federer start to stabilize after the 10th iteration while the chains for other 6 players stabilize even earlier, which indicates that burn-in happens at the 10th iteration.

The auto-correlation time can be estimated by first calculating the sum of the auto covariance coefficients for the first 100 time lags (start from 0) for each of the 107 players. Then time each of the 107 sum of auto covariance coefficients of the players by 2 and then subtract 1 from it, which generates 107 estimated auto-correlation times, one for each dimension. The final estimated auto-correlation time is taken to be the largest of the 107 values:

```
# Autocorrelation time calculation
Autocor = np.zeros((107,100))
for a in range(107):
    autocor = np.zeros(100)
    for i in range(100):
    autocor[i]=pandas.Series.autocorr(pandas.Series(skill_samples[a,:]),lag=
                                                                i)
    Autocor[a] = autocor

Autocor_time = (2*Autocor.sum(1)-1).max()
```

The estimated auto-correlation time is 9.43, which agrees with Figure 3 (auto-correlation drops to 0 for n > 10)

From Figure 1, 2100 iterations is enough to get reliable results as the chain burns in from the 10th iteration. If we wish to draw independent samples from the posterior skill distributions, we can sample for longer and do thinning on the Gibbs sampler.

Problem b)

I run message passing and EP for 100 iterations and record the mean and precision output of each player in each iteration:

```
# For loop, output mean and precision skills in each of the 100 iterations.
mean_player = []
precision_player = []
for i in range(1,101):
    mean_player_skills, precision_player_skills = eprank(G, M, i)
    mean_player.append(mean_player_skills)
    precision_player.append(precision_player_skills)
mean_player = np.array(mean_player)
precision_player = np.array(precision_player)
```

I plot the mean and precision of the 8 players chosen in a) for all 100 iterations:

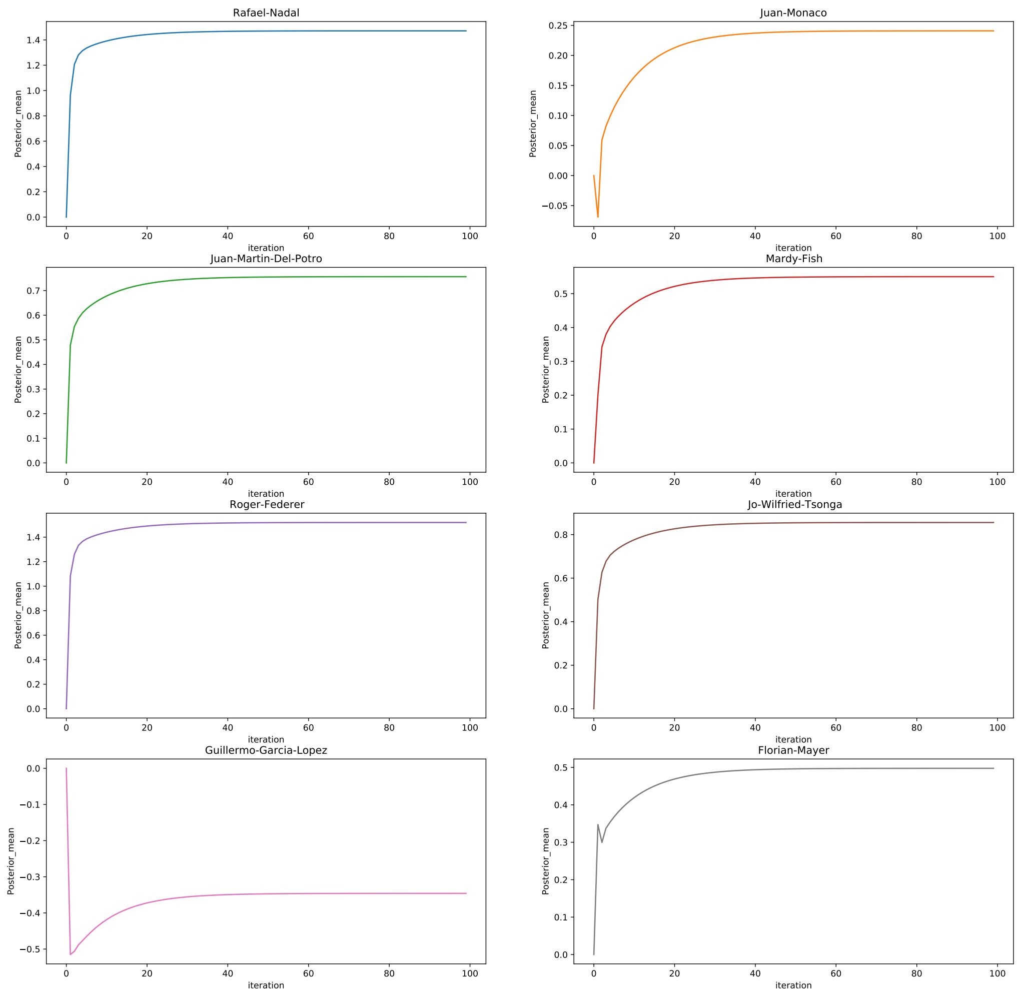

Figure 4: Posterior mean plot for 100 iterations of message passing and EP

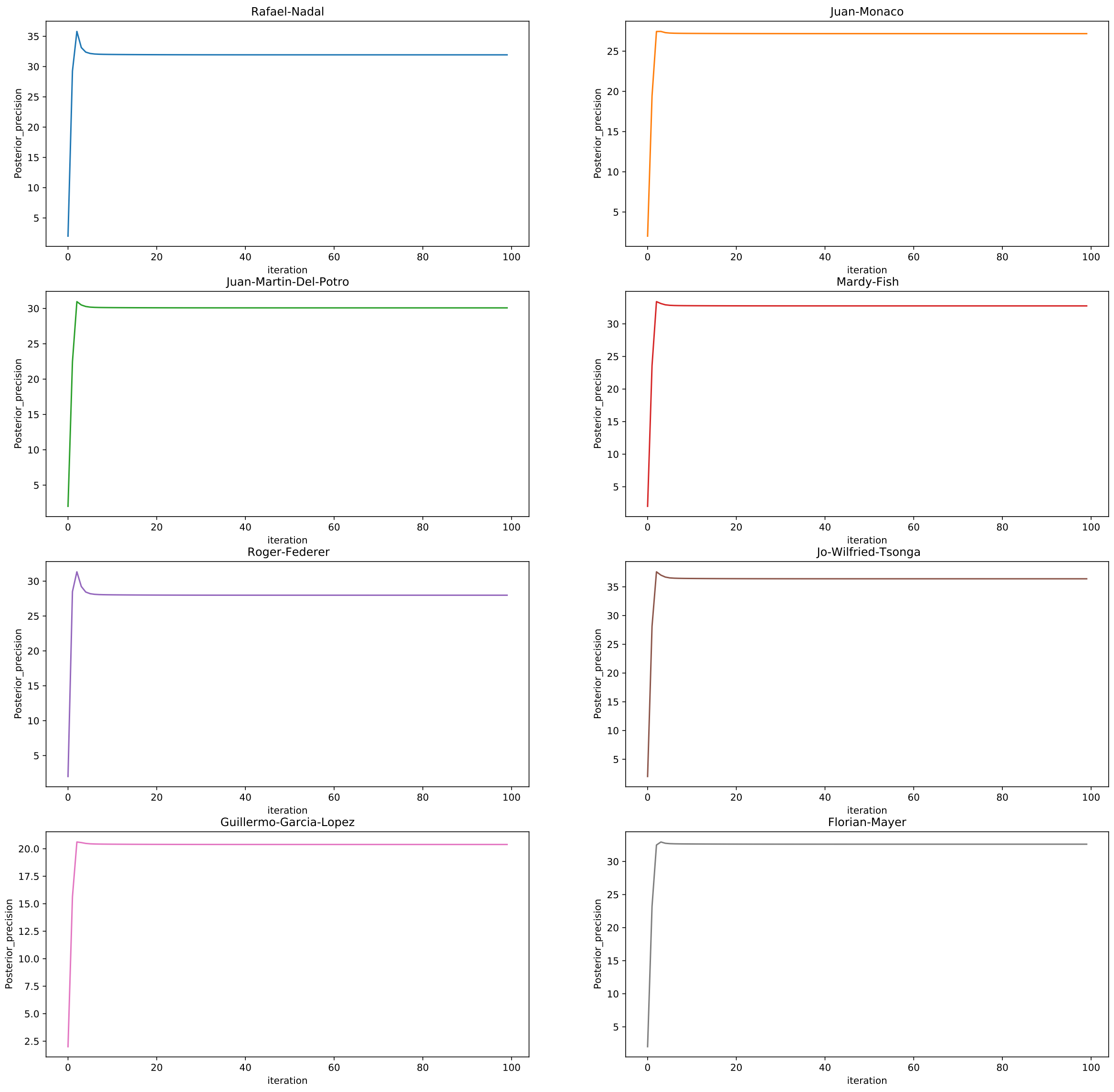

Figure 5: Posterior precision plot for 100 iterations of message passing and EP

For Gibbs sampling, convergence means that the underlying Markov Chain of Gibbs sampler has converged to the stationary (or limiting) distribution we want to sample from (in this case, the joint posterior skill distribution) such that the sample we draw from the chain are approximately distribution according to the posterior joint skill distribution. To judge convergence here, we can find the burn-in time by looking at traceplots or running Gibbs sampler several times with very different initializations and checking when do all the different chains converge to the same region. From Figure 1 and 2, 10 iterations are necessary for convergence.

For message passing, we are approximating the marginal posterior skill distribution of each player by a Gaussian distribution, so convergence means that the mean and precision parameters of the posterior-approximating Gaussian distribution for each player has converged to a fixed value, which returns a set of Gaussian distributions that best approximates the marginal posterior skill distribution. For convergence check, we can plot the mean and precision output of the message passing algorithm and check whether these two parameters converge. In Figure 4 and 5, 60 iterations are necessary for convergence.

Problem c)

Here is the code for finding the index of the top 4 player:

```python
# Code for find index of the top 4 player.
import scipy
top_4_index = []
for i in ["Novak-Djokovic","Rafael-Nadal","Roger-Federer","Andy-Murray"]:
    top_4_index.append(np.where(W==i)[0].item())
```

And the code for calculating the two tables of probabilities:

```python
# Probability that the skill of one player is higher than the other for the
                                        top 4 players
skill_prob = np.zeros([4,4])
for i in range(4):
    for j in range(4):
        skill_prob[i,j] = 1-scipy.stats.norm.cdf(0,mean_player[-1,
                                        top_4_index[i]]-mean_player[-1
                                        ,top_4_index[j]],np.sqrt(1/
                                        precision_player[-1,
                                        top_4_index[i]]+1/
                                        precision_player[-1,
                                        top_4_index[j]]))

# Probability that one player is beating the other for the top 4 players.
winning_prob=np.zeros([4,4])
for i in range(4):
    for j in range(4):
        winning_prob[i,j] = 1-scipy.stats.norm.cdf(0,mean_player[-1,
                                        top_4_index[i]]-mean_player[-1
                                        ,top_4_index[j]],np.sqrt(1/
                                        precision_player[-1,
                                        top_4_index[i]]+1/
                                        precision_player[-1,
                                        top_4_index[j]]+1))
```

P(skills of player on the row is higher)	Novak-Djokovic	Rafael-Nadal	Roger-Federer	Andy-Murray
Novak-Djokovic	-	0.93982221	0.90888527	0.98532149
Rafael-Nadal	0.06017779	-	0.42717016	0.7665184
Roger-Federer	0.09111473	0.57282984	-	0.81083525
Andy-Murray	0.01467851	0.2334816	0.18916475	-

Table 1: Table for the probabilities that the skill of one player is higher than the other, where each entry is the probability that the skill of the player on the row is higher than the skill of the player on the column.

P(player on the row beats the player on the column)	Novak-Djokovic	Rafael-Nadal	Roger-Federer	Andy-Murray
Novak-Djokovic	-	0.65536705	0.63802697	0.71982569
Rafael-Nadal	0.34463295	-	0.4816481	0.57310992
Roger-Federer	0.36197303	0.5183519	-	0.59087902
Andy-Murray	0.28017431	0.42689008	0.40912098	-

Table 2: Table for probability of one player winning a match between the two, where each entry is the probability that the player on the row beats the player on the column.

We can see that the probability that a low-ranked player beats a high-ranked player (Nadal beats Djokovic for example) is much larger than the probability that the skill of the low-ranked player is higher than the high-ranked player. The reason is that, for Table 1, we are calculating:

$$p(s = w_1 - w_2 > 0), \text{ where } s = w_1 - w_2 \sim \mathcal{N}(\mu_1 - \mu_2, \sigma_1^2 + \sigma_2^2) \text{ is the skill difference and } \mu_1, \mu_2, \sigma_1^2, \sigma_2^2 \text{ are the posterior mean and variance values of skills } w_1, w_2 \text{ respectively.}$$

While for Table 2 we are instead calculating:

$$p(t = w_1 - w_2 + \varepsilon > 0), \text{ where } t = w_1 - w_2 + \varepsilon \sim \mathcal{N}(\mu_1 - \mu_2, \sigma_1^2 + \sigma_2^2 + 1) \text{ is the performance difference.}$$

We have $p(s > 0) = \Phi(\frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}})$ and $p(t > 0) = \Phi(\frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2 + 1}})$ where Φ is the standard normal cdf. Thus, for $\mu_1 - \mu_2 < 0$ (as we are computing the probability that a low-rank player beats a high-ranked player), $\frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2 + 1}}$ is larger than $\frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}}$ and this indicates that $p(s > 0) = \Phi(\frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}}) < \Phi(\frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2 + 1}}) = p(t > 0)$. Figure 6 also shows the difference:

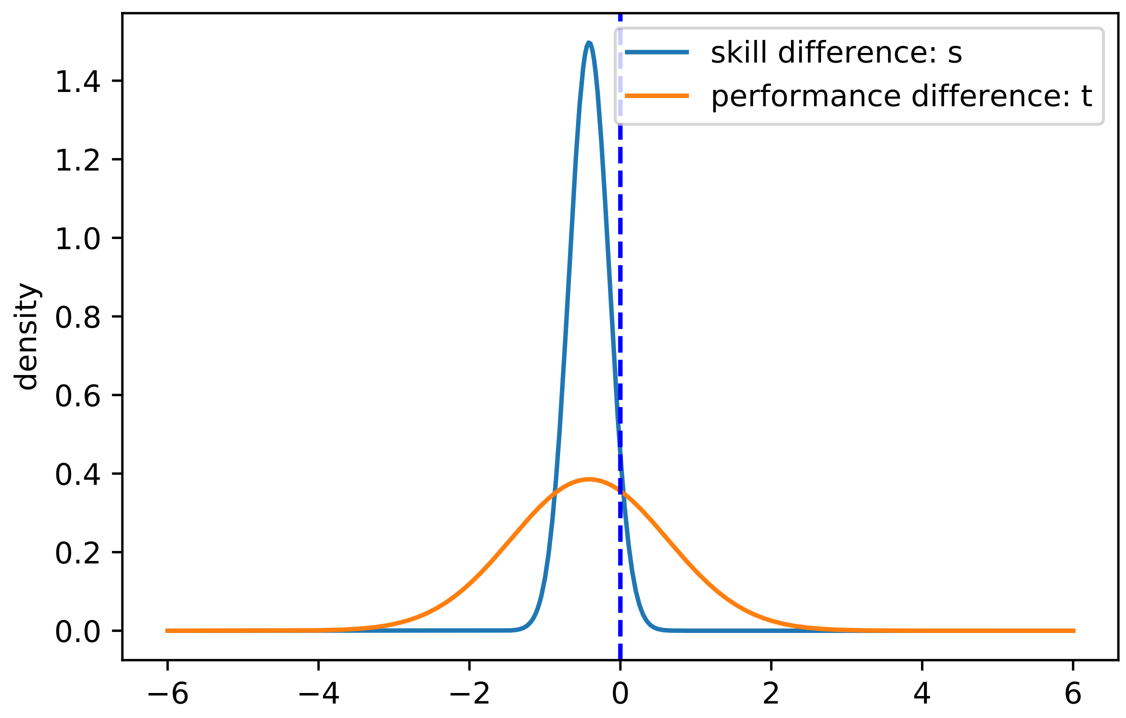

Figure 6: Density plot of skill difference and performance difference of Rafael-Nadal against Novak-Djokovic

Problem d)

I calculate the probability that the Nadal's skill, w_N, is higher than the Djokovic's skill, w_D, denoted as $P(w_N - w_D > 0)$ using three methods.

Firstly, I discard the first 10 samples of the 2100 samples:

```
# Discard first 10 samples, only use 2090 samples to make sure burn-in
skill_samples_top_4 = skill_samples[top_4_index][:,10:]
```

Method 1): Marginal Gaussian

Here, I calculate the Maximum Likelihood Estimate (MLE) of individual Gaussian mean ($\hat{\mu}$) and variance ($\hat{\sigma^2}$):

$$\hat{\mu} = \frac{\sum_{n=1}^{2090} w_n}{2090}$$
$$\hat{\sigma^2} = \frac{\sum_{n=1}^{2090} (w_n - \hat{\mu})^2}{2090}$$

```
## Using 2 marginal Gaussian for approximating marginal skills
# For Nadal:
mean_Nadal = np.mean(skill_samples_top_4[1])
```

```
var_Nadal = np.var(skill_samples_top_4[1])
# For Djokovic:
mean_Djokovic = np.mean(skill_samples_top_4[0])
var_Djokovic = np.var(skill_samples_top_4[0],ddof=0)
# Compute P(Nadal skill > Djokovic skill)
P_marginal = 1 - scipy.stats.norm.cdf(0,mean_Nadal-mean_Djokovic,np.sqrt(
                                     var_Nadal+var_Djokovic))
```

The marginal Gaussian density plots:

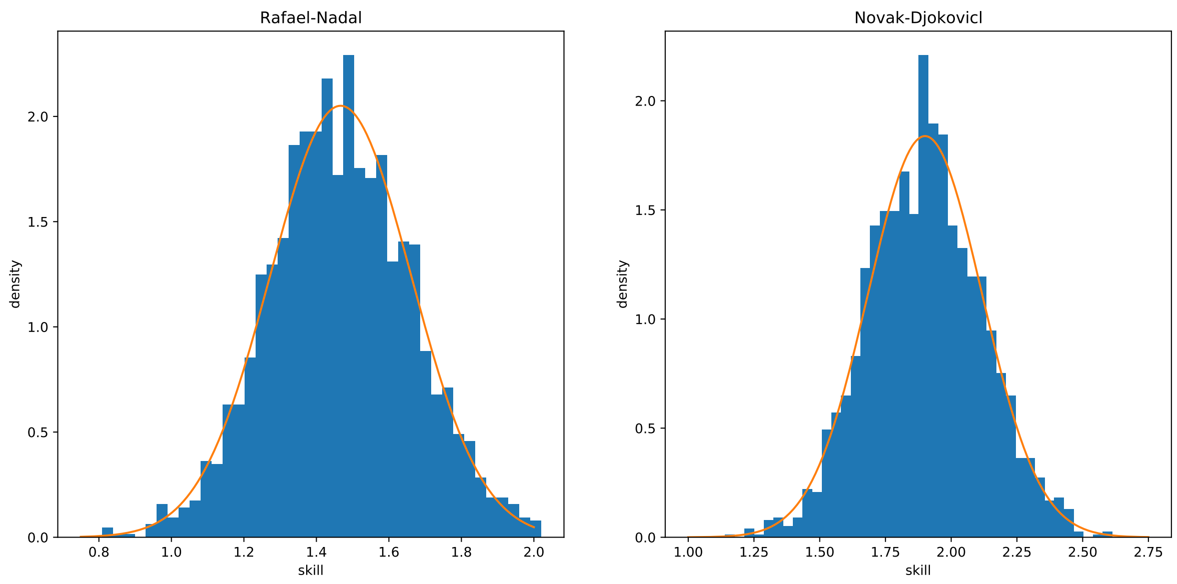

Figure 7: Density plots of the marginal distribution approximating the posterior skill distribution of Nadal and Djokovic together with histograms of the sampled skills from Gibbs sampler

Method 2): Joint Gaussian

Here, I still use the MLE of 2×1 mean vector ($\hat{\boldsymbol{\mu}}$) and covariance matrix ($\hat{\boldsymbol{\Sigma}}$) of the joint Gaussian:

$$\hat{\boldsymbol{\mu}} = \frac{\sum_{n=1}^{2090} \boldsymbol{w}_n}{2090}$$
$$\hat{\boldsymbol{\Sigma}} = \frac{\sum_{n=1}^{2090} (\boldsymbol{w}_n - \hat{\boldsymbol{\mu}})(\boldsymbol{w}_n - \hat{\boldsymbol{\mu}})^T}{2090}$$

```
## Using one Gaussian for approximating joint skill
mean_joint = np.mean(skill_samples_top_4[[0,1]],axis=1)
cov_mat = np.cov(skill_samples_top_4[[0,1]],ddof=0)
# Compute P(Nadal skill > Djokovic skill)
P_joint = 1-scipy.stats.norm.cdf(0,mean_joint[1]-mean_joint[0],np.sqrt(
                                     cov_mat[0,0]+cov_mat[1,1]-2*cov_mat[0,
                                     1]))
```

The joint Gaussian density plot:

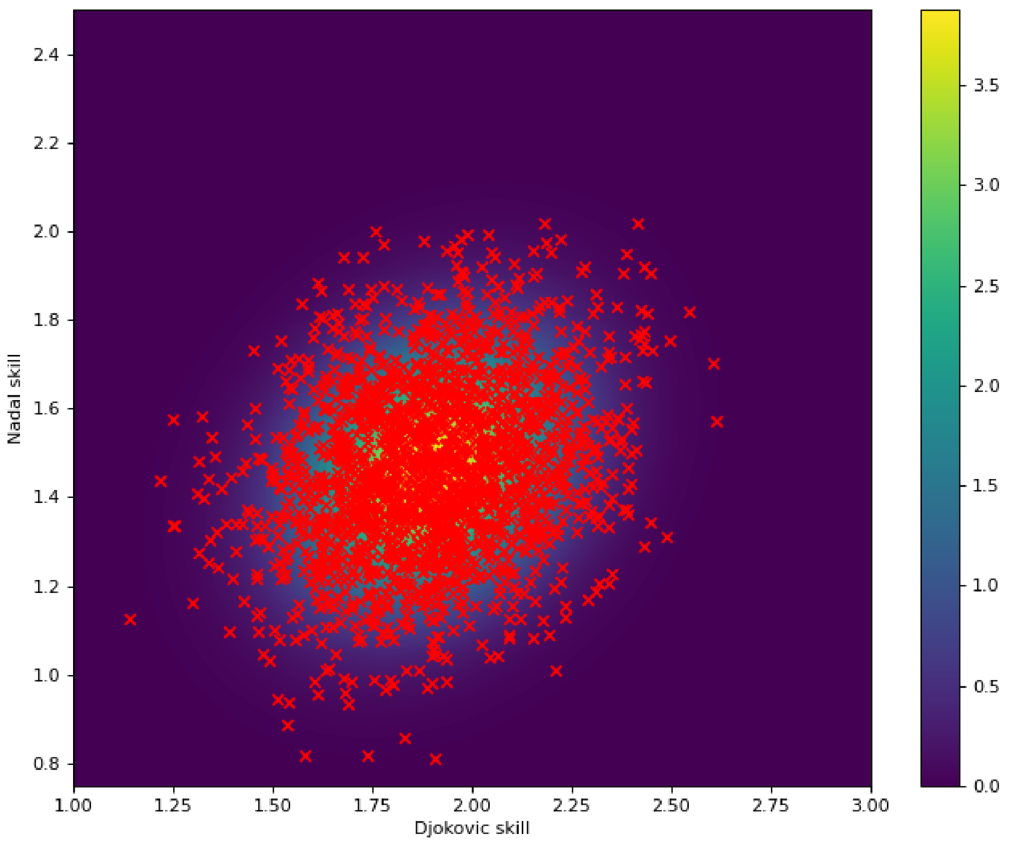

Figure 8: Density plot of the joint Gaussian distribution approximating the joint posterior skill distribution of Nadal and Djokovic together with sampled skills of Nadal and Djokovic from Gibbs sampler

Method 3): Directly from data

Here, I calculate the proportion of samples where Nadal's skill is higher than Djokovic's skill to estimate $P(w_N - w_D > 0)$:

```
## Using the data directly, discard the first 10 samples
# Compute P(Nadal skill > Djokovic skill)
P_direct = np.mean(skill_samples_top_4[1]>skill_samples_top_4[0])
```

Comparing skills of Nadal and Djokovic and all top 4 players

The estimated $P(w_N - w_D > 0)$ are:

	Marginal Gaussian	Joint Gaussian	Directly From Data
$P(w_N - w_D > 0)$	0.06841577	0.04448154	0.04401914

Table 3: Table for $P(w_N - w_D > 0)$ using three methods

From Table 3, method 2) returns a smaller $P(w_N - w_D > 0)$ than method 1) as $w_N - w_D$ under joint Gaussian approximation has smaller variance (an additional positive $2 * Cov(w_N, w_D)$ covariance term is subtracted) than in the marginal Gaussian case. Method 3) gives a similar $P(w_N - w_D > 0)$ as method 2) and method 3) is the best method because method 1) and 2) approximate the posterior skill distributions by Gaussians while the posterior skill distributions are not Gaussian. Also, method 3) is the Monte Carlo estimate of $P(w_N - w_D > 0)$, which is guaranteed to converge to the true value of $P(w_N - w_D > 0)$ when the number of samples gets larger.

The 4×4 skill table using method 3) is:

P(the skill of player on the row is higher)	Novak-Djokovic	Rafael-Nadal	Roger-Federer	Andy-Murray
Novak-Djokovic	-	0.95598086	0.91674641	0.9861244
Rafael-Nadal	0.04401914	-	0.39090909	0.76028708
Roger-Federer	0.08325359	0.60909091	-	0.81004785
Andy-Murray	0.0138756	0.23971292	0.18995215	-

Table 4: Table for the probabilities that the skill of one player is higher than the other using the third estimation method in d), where each entry is the probability that the skill of the player on the row is higher than the skill of the player on the column.

There are differences in all probabilities returned in Table 4 and Table 1 but the differences are not huge. The reason is that in message passing algorithm, the marginal posterior skills are approximated by Gaussians while there is no such approximation in method 3).

Problem e)

Method 1): empirical game outcome averages

For each player, I compute the ratio of the number of games won to total number of games played:

```
# empirical game outcome averages prediction
P_empirical = np.zeros(107)
for i in range(107):
    P_empirical[i] = np.sum(G[:,0]==i)/(np.sum(G[:,0]==i)+np.sum(G[:,1]==i))
sorted_barplot(P_empirical,W)
```

We get:

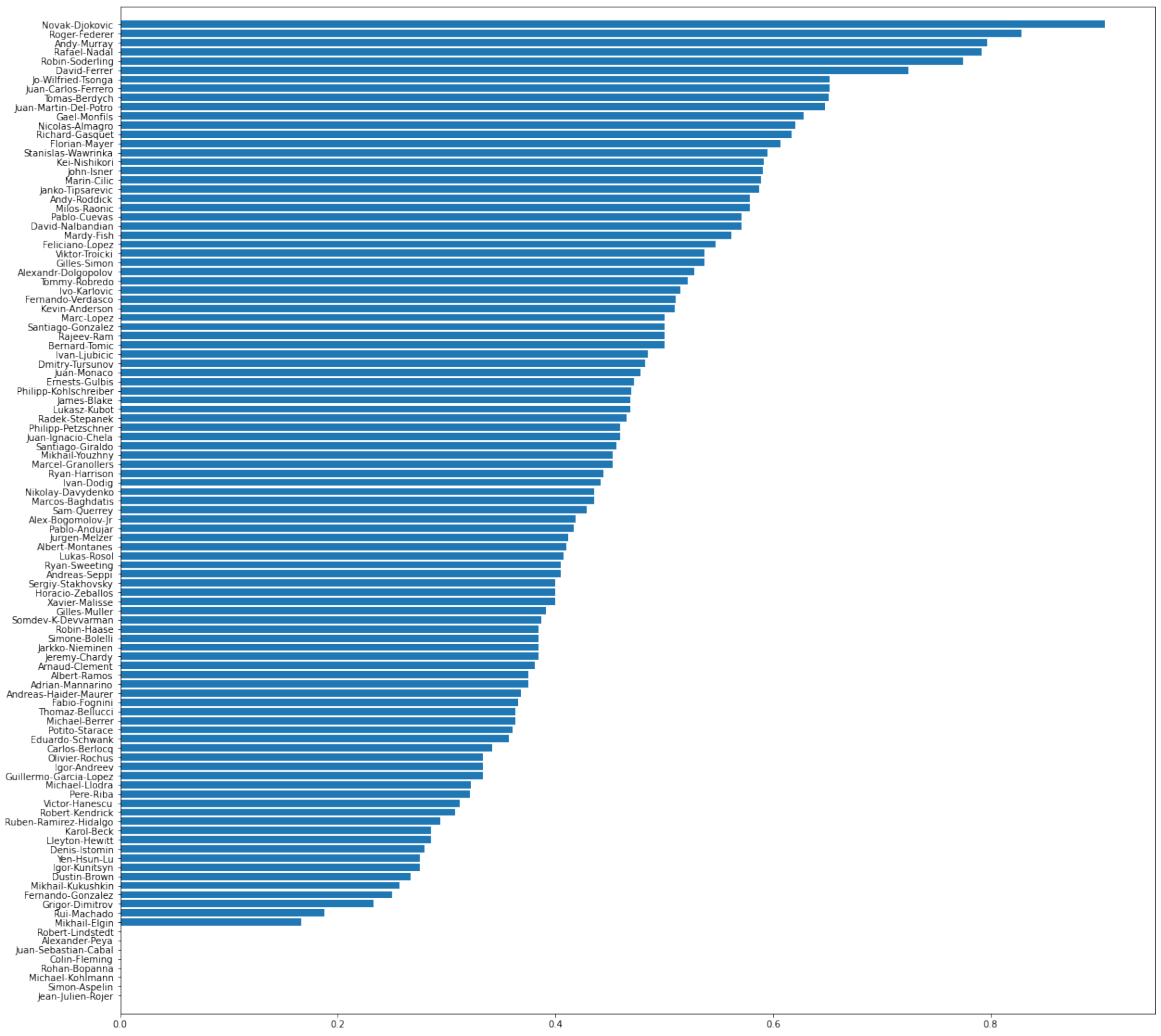

Figure 9: Predicted ranking using empirical game outcome averages

Method 2): Gibbs sampling prediction

For each player, I first calculate the winning probabilities of all 2090 samples (using $\Phi(w_i - w_j)$ for individual sample) of the player against another player and average the 2090 probabilities. Repeat this procedure to calculate the winning probability of this player against all other 106 players and average the 106 probabilities for final prediction:

```
# Gibbs sampling predictions discarding the first 10 samples from the gibbs
                                                    sampler
Gibbs_outcome = np.zeros((107,107))
skill_samples_burn_in = skill_samples[:,10:]
for i in range(107):
    for j in range(107):
        if i!=j:
            Gibbs_outcome[i,j] = np.mean(scipy.stats.norm.cdf(
                                                skill_samples_burn_in[i]-
                                                skill_samples_burn_in[j]))
        elif i==j:
            Gibbs_outcome[i,j] = 0

P_Gibbs = np.sum(Gibbs_outcome,axis=1)/106
sorted_barplot(P_Gibbs,W)
```

We get:

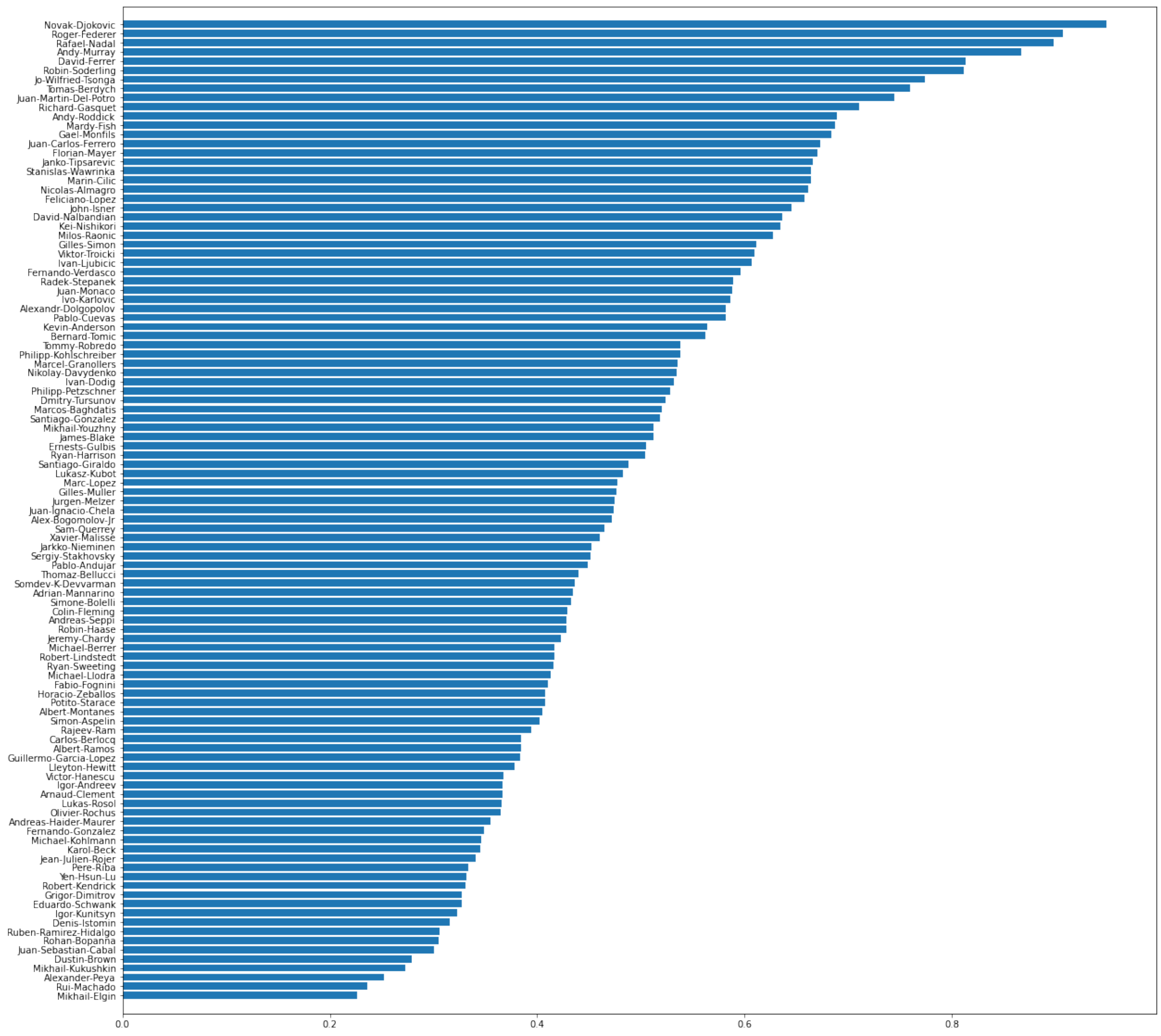

Figure 10: Predicted ranking using Gibbs sampling predictions

Method 3): message passing algorithm prediction

For each player, I use the same method in c) to calculate the winning probability of the player against all other 106 players and average the 106 winning probabilities to get the final prediction:

```
# message passing predictions
MP_outcome = np.zeros((107,107))
for i in range(107):
    for j in range(107):
        if i!=j:
            MP_outcome[i,j] = 1-scipy.stats.norm.cdf(0,mean_player[-1,i]-
                                                      mean_player[-1,j],np.sqrt(
                                                      1/precision_player[-1,i]+1
                                                      /precision_player[-1,j]+1)
                                                      )
        elif i==j:
            MP_outcome[i,j] = 0

P_MP = np.sum(MP_outcome,axis=1)/106
sorted_barplot(P_MP,W)
```

We get:

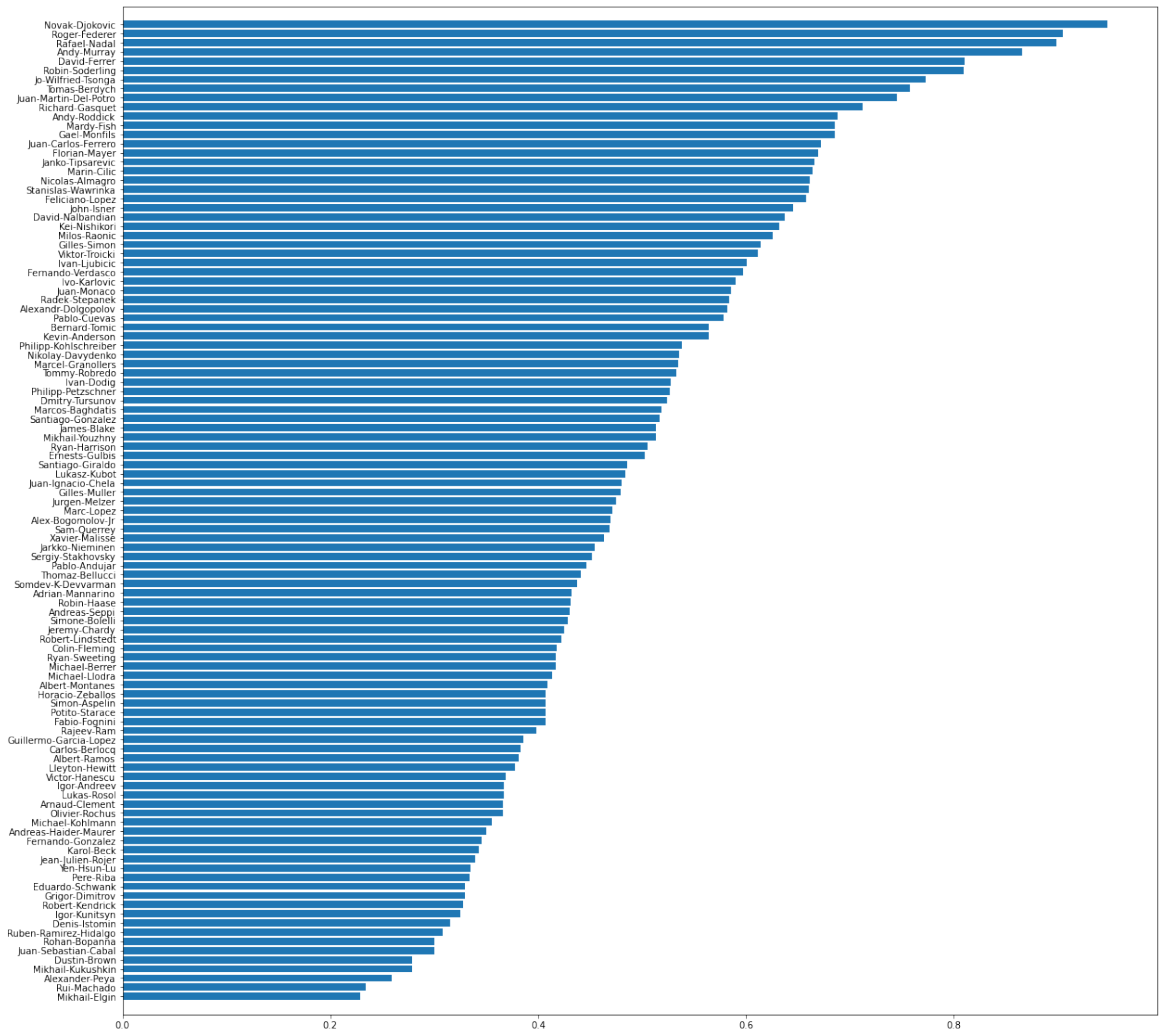

Figure 11: Predicted ranking using message passing algorithm predictions

Comparison

The bottom-ranked players returned by method 1) are different from the other two methods. Specifically, the bottom-ranked players are those with 0 winning probabilities. The reason is that method 1) does not account for the number of games a player played. For example, a player with medium skills may rank bottom simply because he only played 1 game and lost it, which is unfair. Instead, method 2) and 3) rank players by their posterior skills inferred, which is more reliable.

The ranking returned by methods 2) and 3) are overall similar with minor differences (for example, Stanislas-Wawrinka and Marin-Cilic). The difference is due to the inference methods: message passing algorithm approximates the marginal posterior skills via Gaussians while in Gibbs sampling we directly draw skill samples from the exact skill posterior.