1RT705: Project Assignment
Advanced Probabilistic Machine Learning 2020

Paul Häusner
Nathaniel Ahy
Patrick Petersson

Abstract

The goal of this project is to implement the by Microsoft developed Trueskill model in order to rank teams or players based on results of the games they played against each other. In the implementation two different approaches are tested out. The first one is based on Bayesian networks and uses Gibbs sampling in order to estimate the posterior distribution. The second approach uses factor graphs and is based on their belief propagation with moment-matching in order to estimate non-Gaussian distributions.

1 Q.1 Modeling

The four random variables in our model are shown with their distribution in equation (1) to (3). The Bayesian model itself is given in equation 4.

$$s_1 \sim \mathcal{N}(s_1; \mu_1, \sigma_1^2), s_2 \sim \mathcal{N}(s_2; \mu_2, \sigma_2^2) \tag{1}$$

$$t|s_1, s_2 \sim \mathcal{N}(t; s_1 - s_2, \sigma_t^2), \tag{2}$$

$$y|t = \begin{cases} 1 \text{ if } t \geq 0, \\ -1 \text{ else} \end{cases} \tag{3}$$

$$p(s_1, s_2, t, y) = p(s_1)p(s_2)p(t|s_1, s_2)p(y|t) \tag{4}$$

Thus, the model contains 5 hyper-parameters: The mean and standard deviation for each of the skills $\mu_1, \mu_2, \sigma_1^2, \sigma_2^2$ respectively, as well as the standard deviation for the result of the match σ_t^2.

Q.2 Bayesian Network

The Bayesian model that was introduced in the last section is represented graphically as a Bayesian network as shown in Figure 1.

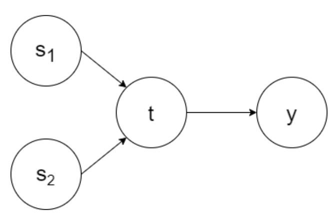


Figure 1: Bayesian Network of the model

From this Bayesian network we can identify the following independent sets of random variables using the rules of D-separation:

- $s_1 \perp\!\!\!\perp s_2 \mid \emptyset$: The variables that show the skills of a player are independent when the remaining variables in the network are non-observed. This follows from the head-to-head rule.

- $(s_1, s_2) \perp\!\!\!\perp y \mid t$: The skills of the player depend solely on the result of the game if it is observed and not the games outcome. This follows from the head-to-tail node with intermediate observed variables.

Q.3 Computing with the Model

For this section, all relevant matrices are found in the appendix. The first distribution to compute is $p(s_1, s_2|t, y)$. This is equivalent to computing $p(s_1, s_2|t)$ as can be seen in the Bayesian network in the previous section with the conditional independence of y and s_1, s_2 derived by the head-to-tail node with an observed variable t in between them. This enables us to apply Corollary 1 from Lecture 2. By this corollary we get that

$$p(s_1, s_2|t) \stackrel{def}{=} p(\boldsymbol{s}|t) = \mathcal{N}(\boldsymbol{s}; \boldsymbol{\mu}_{\boldsymbol{s}|t}, \boldsymbol{\Sigma}_{\boldsymbol{s}|t}) \tag{5}$$

where we compute $\boldsymbol{\mu}_{\boldsymbol{s}|t}$ and $\boldsymbol{\Sigma}_{\boldsymbol{s}|t}$ using Corollary 1 along with the matrices in Appendix B leading us to the closed form solution of the mean and the variance.

$$\boldsymbol{\mu}_{\boldsymbol{s}|t} = \boldsymbol{\Sigma}_{\boldsymbol{s}|t} \begin{pmatrix} \sigma_1^{-2}\mu_1 + \sigma_t^{-2}t \\ \sigma_2^{-2}\mu_2 - \sigma_t^{-2}t \end{pmatrix} \tag{6}$$

$$\boldsymbol{\Sigma}_{\boldsymbol{s}|t} = \frac{1}{(\sigma_2^{-2} + \sigma_t^{-2})(\sigma_1^{-2} + \sigma_t^{-2}) - \sigma_t^{-4}} \begin{pmatrix} \sigma_2^{-2} + \sigma_t^{-2} & \sigma_t^{-2} \\ \sigma_t^{-2} & \sigma_1^{-2} + \sigma_t^{-2} \end{pmatrix}. \tag{7}$$

For finding the full conditional distribution of the outcome, we utilize Bayes' Theorem which states the proportionality relation shown in Equation 8:

$$p(t|s_1, s_2, y) \propto p(y|t)p(t|s_1, s_2). \tag{8}$$

Here we know from the equations introduced in section Q.1 that the factor $p(t|s_1, s_2)$ is simply given from $\mathcal{N}(t; s_1 - s_2, \sigma_t^2)$, and as such we need to evaluate the factor $p(y|t)$. Since the random variable y is defined as $sgn(t)$, we know that $y = 1 \Leftrightarrow t \geq 0$. Thus, $p(y = 1|t) = \mathbb{I}_{\{y=sign(t)\}}$, where $\mathbb{I}$ is an indicator function. Put together, we get:

$$p(t|s_1, s_2, y) = \begin{cases} \mathbb{I}_{\{y=sign(t)\}}\mathcal{N}(t; s_1 - s_2, \sigma_t) \cdot \frac{1}{\int_0^\infty \mathcal{N}(t; s_1 - s_2, \sigma_t)dt}, & y = 1 \\ \mathbb{I}_{\{y=sign(t)\}}\mathcal{N}(t; s_1 - s_2, \sigma_t) \cdot \frac{1}{\int_{-\infty}^0 \mathcal{N}(t; s_1 - s_2, \sigma_t)dt}, & y = -1, \end{cases} \tag{9}$$

i.e. this probability will take on a truncated Gaussian which we will denote as $\mathcal{TG}(t; a, b, \mu, \sigma)$ where a, and b, are the lower and upper bounds of truncation respectively, μ, and σ are the mean and standard deviation parameters of the underlying *normal* distribution.

The marginal probability that player 1 wins the game, $p(y = 1)$, is given by $p(t > 0)$. This probability is obtained by marginalizing out s_1 and s_2 from the joint distribution $p(s_1, s_2, t)$ using Corollary 2 from the lectures. For this corollary we use the known distributions $p(\boldsymbol{s}) = \mathcal{N}(\boldsymbol{s}; \boldsymbol{\mu_s}, \boldsymbol{\Sigma_s})$ and $p(t|\boldsymbol{s}) = \mathcal{N}(t; \boldsymbol{A}\boldsymbol{s} + \boldsymbol{b}, \boldsymbol{\Sigma_{t|s}})$, To find $p(t)$ from this corollary we compute $\boldsymbol{\mu_{s|t}}$ and $\boldsymbol{\Sigma_s}$ with the equations given in Corollary 2. Therefore, we can as such obtain the following marginal distribution:

$$p(t) = \mathcal{N}(\boldsymbol{t} : \boldsymbol{\mu_t}, \boldsymbol{\Sigma_t}) \stackrel{def}{=} \mathcal{N}(t; \mu_1 - \mu_2, \sigma_1^2 + \sigma_2^2 + \sigma_t^2) \tag{10}$$

Finally, the probability that player 1 wins is calculated as

$$p(y = 1) = p(t > 0) = \int_0^\infty \mathcal{N}(t; \mu_1 - \mu_2, \sigma_1^2 + \sigma_2^2 + \sigma_t^2)dt \tag{11}$$

Q.4 A first Gibbs sampler

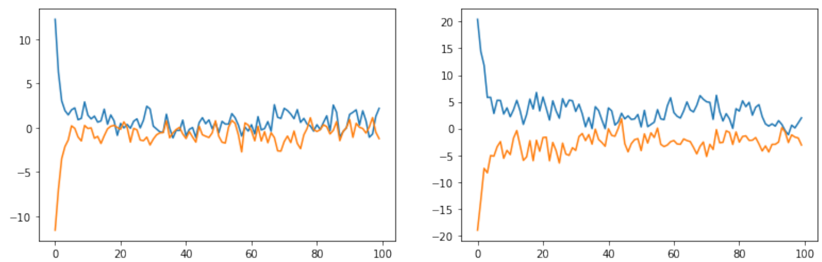
In order to implement a Gibbs sampler with an observed variable y it is required that we can sample from both of the conditional probabilities:

$$p(\boldsymbol{s}|t, y = 1) = \mathcal{N}(\boldsymbol{s}; \boldsymbol{\mu}_{\boldsymbol{s}|t}, \boldsymbol{\Sigma}_{\boldsymbol{s}|t}) \tag{12}$$

$$p(t|s_1, s_2, y = 1) = \mathcal{TG}(t; 0, \infty, s_1 - s_2; \sigma_t^2) \tag{13}$$

These equations and parameters have been evaluated in the previous section: Q.3. For the prior distribution $s_1 \sim \mathcal{N}(s_1; 0, 1)$ is used. The second skill variable, s_2, uses the same prior distribution and $\sigma_t^2 = 2$ is chosen as a hyper-parameter.

In Figure 2a the samples for s_1 and s_2 are shown after observing the first game where $y = 1$. As an initial value for the sampling $t_* = 20$ has been chosen (this has intentionally been chosen to be slightly more off than we expect it to be in order to visualize the burn in). In Figure 2b sampling is shown again where the new prior that was obtained from the first sampling is shown. It seems take longer in the second run to reach the stationary distribution compared to the initial run we executed. We select the burning period therefore to have a size of $b = 20$ since then we have reached the stationary part in both cases and can be sure to only sample from the stationary distribution.



(a) Sampling using two identical priors (b) Sampling using the posterior as a prior

Figure 2: Gibbs sampling

In Figure 3 different choices of samples drawn with their respective required times are shown. We can see that when choosing a very small number of samples like 50 or 100, the estimated Gaussian curve doesn't fit the sample data too well. For higher numbers like $1,000$ to $10,000$ the Gaussian approximation looks a lot better. In order to obtain an overall algorithm with a justifiable runtime, we choose the sample size $K = 1,000$. In Figure 4 the prior and posterior distribution of the skill

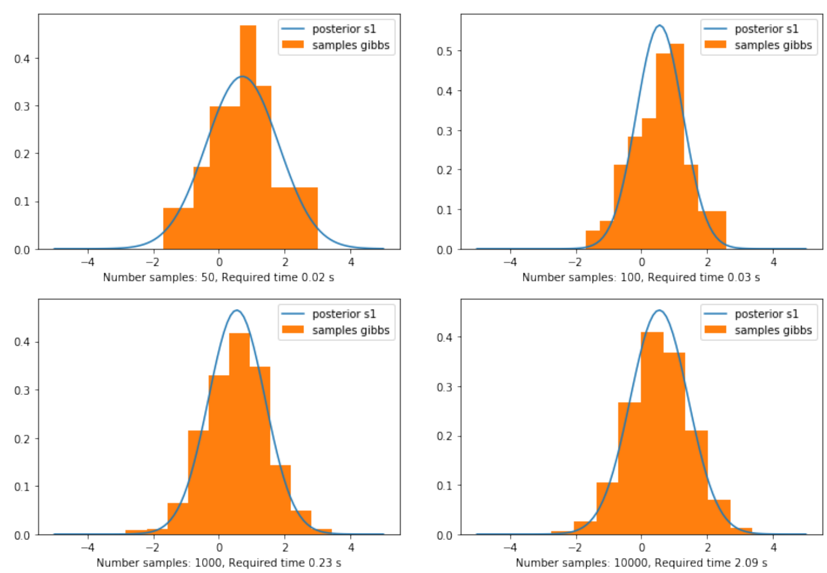


Figure 3: Different number of gibbs samples used to approximate the posterior distribution

variables s_1 and s_2 are shown. The prior distribution is just normally distributed with mean 0 and a standard deviation of 1 for both random variables. The posterior distribution is obtained after observing one data point where $y = 1$. This means that $t > 0$ and thus, player 1 won the actual game. To reflect this the mean of the skill for player 1 is increased and the mean for player 2 is decreased. This can be seen in the graph as the posterior distribution for s_1 is shifted to the right, while the one for s_2 is shifted to the left. Since now we have more certainty about the actual skill of each player the standard deviation is decreased for each variable.

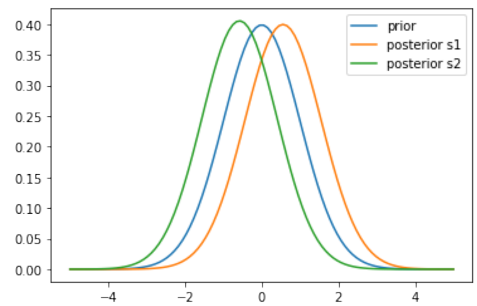


Figure 4: The prior distribution of s_1 and s_2 as well as their respective posterior distribution after observing a game with the outcome $y = 1$

Q.5 Assumed Density filtering

For each team we set the prior distribution of the skill to for each player: $p(s_x) \sim N(s_x; 0, 1)$, where $x \in \{1, 2\}$. The hyper-parameter σ_t is arbitrarily set to 2.

The final ranking of the teams is given in Figure 5 where the different distributions for the teams are shown after the whole dataset has been analyzed. We can see that the first and the second ranked team have a distinctively larger mean than the other teams. Those two teams are `Juventus` and `Napoli`. Those correspond to the champion and the vice-champion of the league. In the figure the ranking of the teams based on their mean is also given as a list. We can see that all the teams have a fairly similar standard deviation. This indicates how certain we are how that team performs and how likely that is that they will win or loose a game against a team that is either a lot better or worse respectively.

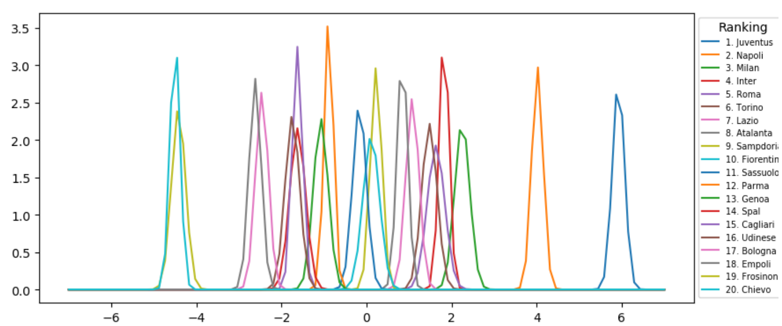


Figure 5: The distribution of the posterior skill for all teams from the dataset

When the ordering of the matches in the dataset is changed the result is most likely different. The reason for this is that with the ordered matches every team has seen an equal amount of matches after each game day. Thus, all the posterior distributions that represent the skill of the player have a similar standard deviation. When randomly shuffling the matches, this is not the case any more and we might have to estimate more often how teams perform about which we don't know the performance yet.

Q.6 Using the model for predictions

When using the predictor that is based on our TrueSkill model the obtained prediction rate is $r = 0.65$. When using a completely random guesser we would expect to achieve $r = 0.5$. However, the provided dataset is slightly biased. It is more likely for a team to win when it plays at home (it is listed as `team1` in the dataset). Thus, when we predict for every match that the home team wins we obtain a prediction rate of $r = 0.61$. Our model doesn't currently take into account this data bias. Furthermore, we can see that the performance of our predictor still outperforms the random predictors slightly.

Q.7 Factor graph

We begin by simply drawing the factor graph corresponding to the random variables and their connections in the project, shown in Figure 6.

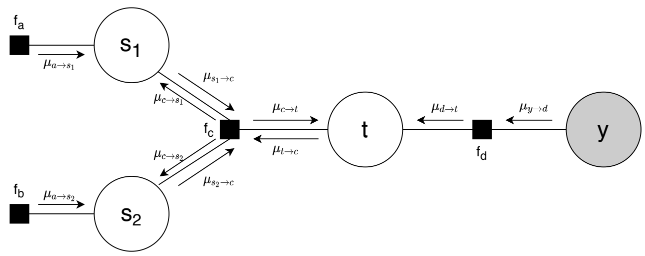


Figure 6: The random variables and the messages presented in a factor graph.

To compute the distributions $p(s_1|y)$ and $p(s_2|y)$ we use message passing, through observing the factor graph of this section. We let

$$
\begin{aligned}
f_a(s_1) &= \mathcal{N}(s_1; \mu_1, \sigma_1^2), \qquad f_b(s_2) = \mathcal{N}(s_2; \mu_2, \sigma_2^2) \\
f_c(t) &= \mathcal{N}(t; s_1 - s_2, \sigma_t^2), \quad f_d(y) = \delta(y),
\end{aligned}
$$

where δ is an indicator function. This, along with the formulas provided in Lecture 5, allows us to derive the following messages in a trivial manner:

$$
\begin{aligned}
\mu_{a\to s_1}(s_1) &= f_a(s_1), \quad \mu_{b\to s_2}(s_2) = f_b(s_2) \\
\mu_{s_1\to c}(s_1) &= f_a(s_1), \quad \mu_{s_2\to c}(s_2) = f_b(s_2)
\end{aligned}
$$

$$
\begin{aligned}
\mu_{c\to t}(t) &= \iint \mathcal{N}(t; s_1 - s_2, \sigma_t^2)\mathcal{N}(s_1; \mu_1, \sigma_1^2)\mathcal{N}(s_2; \mu_2, \sigma_2^2) ds_2 ds_1 && (14) \\
&\overset{Corrollary\ 2}{=} \mathcal{N}(t; \mu_1 - \mu_2, \sigma_t^2 + \sigma_1^2 + \sigma_2^2), && (15)
\end{aligned}
$$

where the identities we use for the above corollary are in Appendix C.

Q.8 A Message passing algorithm

We have that

$$\mu_{c\to t}(t)\mu_{y\to t}(t) = \delta(y = sign(t))\mathcal{N}(t; \mu_1 - \mu_2, \sigma_t^2 + \sigma_1^2 + \sigma_2^2),$$

and moment match $\hat{q}(t; \hat{\mu}_q, \hat{\sigma}_q^2)$ to this distribution. Moreover, we make the definition:

$$
\begin{aligned}
\hat{p}(t; \hat{\mu}, \hat{\sigma}^2) &\overset{def}{=} \frac{\hat{q}(t; \hat{\mu}_q, \hat{\sigma}_q^2)}{\mu_{c\to t}(t)} \\
&= \mathcal{N}(t; \hat{\mu}, \hat{\sigma}^2) \approx \mu_{t\to c}(t).
\end{aligned}
$$

Finally

$$
\begin{aligned}
\mu_{c\to s_1}(s_1) &\approx \int \mathcal{N}(t; \hat{\mu}, \hat{\sigma}^2) \int \left[\mathcal{N}(t; s_1 - s_2, \sigma_t^2)\mathcal{N}(s_2; \mu_2, \sigma_2^2) ds_2\right] dt && (16) \\
&\overset{Corollary\ 2}{=} \int \mathcal{N}(t; \hat{\mu}, \hat{\sigma}^2)\mathcal{N}(t; s_1 - \mu_2, \sigma_t^2 + \sigma_2^2) dt, && (17) \\
&= \int \mathcal{N}(s_1; t + \mu_2, \sigma_t^2 + \sigma_2^2)\mathcal{N}(t; \hat{\mu}, \hat{\sigma}^2) && (18)
\end{aligned}
$$

where the matrices in the Corollary 2 are found in the Appendix C.2. Applying Corollary 2 with matrices found in Appendix C.3 with the following: gives us that

$$\mu_{c\rightarrow s_1}(s_1) \approx \mathcal{N}(s_1; \hat{\mu} + \mu_2, \sigma_t^2 + \sigma_2^2 + \hat{\sigma}^2). \tag{19}$$

Finally, the desired posterior probability is computed by the equality below:

$$p(s_1|y) \propto \mu_{c\rightarrow s_1}(s_1)\mathcal{N}(s_1; \mu_1, \sigma_1^2).$$

Similarly we get that the posterior distribution for s_2 is:

$$p(s_2|y) \propto \mathcal{N}(s_2; \hat{\mu} - \mu_1, \sigma_t^2 + \sigma_1^2 + \hat{\sigma}^2)\mathcal{N}(s_2; \mu_2, \sigma_2^2).$$

With this we can now estimate the posterior distribution using moment-matching and compare this distribution with the one obtained from the Gibbs sampling as shown in Figure 7.

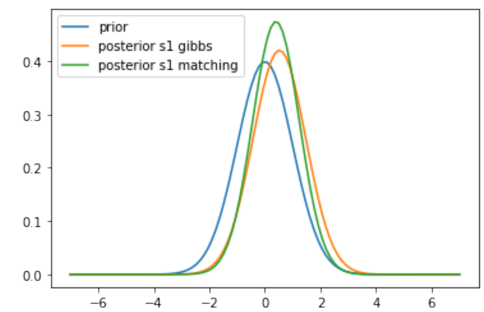


Figure 7: Prior distribution and posterior distribution estimated using a gibbs sampler and moment-matching

Q.9 Our own data

The developed Trueskill method is tested on a dataset containing tennis matches from the ATP tournament.[1] We only take a look at the matches that have taken place in the last 5 years in order to have a manageable amount of different players and matches between them.

Our model achieves a prediction accuracy of 65% on this dataset. We disregarded Gibbs sampling as the run time of the algorithm increases significantly due to the relatively high number of needed samples and a high number of total matches.

Q.10 Project extension

As already described in section Q.6 where we try to predict the outcome of the upcoming matches the data includes a small bias towards the chance of the home team winning. In reality we can see that the probability that the home team wins is about 60%. We slightly adjust our model to take that factor into account. Therefore, when we predict the outcome of the next game we shift the mean of distribution to the right by a factor $\Delta\mu$ such that $p(y = 1) = 0.6$ given that both teams have the same mean. Thus we chose an approximate $\Delta\mu$ such that the following holds for teams with an equal mean

$$p(y = 1) = p(t > 0) = \int_0^\infty N(t; \Delta\mu; \sigma_1^2 + \sigma_2^2 + \sigma_t^2) = 0.6 \tag{20}$$

The value $\Delta\hat{\mu}$ can be looked up in a table for the Gaussian normal distribution. We choose $\Delta\hat{\mu} = 0.25\sqrt{(\sigma_1^2 + \sigma_2^2 + \sigma_t^2)}$. The marginal distribution of y is now given by

$$p(y = 1) = p(t > 0) = \int_0^\infty N(t; \mu_1 - \mu_2 + \Delta\hat{\mu}; \sigma_1^2 + \sigma_2^2 + \sigma_t^2). \tag{21}$$

Adding this a-priori advantage for the home team to our model increases the performance of the predictions from 65% to approximately 70%.

[1] `https://www.kaggle.com/sijovm/atpdata`

A Code

A.1 Code for Quiz

```
#project test
m1=1
sigma1=1#sigma is the variance not the volatility here!
m2=-1
sigma2=4
sigma3=5
#task 1
div=(sigma2**(-1)+sigma3**(-1))*(sigma1**(-1)+sigma3**(-1))-sigma3**(-2)
sigma_st=np.array([[sigma2**(-1)+sigma3**(-1), sigma3**(-1)],
          [sigma3**(-1), sigma1**(-1)+sigma3**(-1)]])/div
t=3
print(sigma_st)
multiplier=np.array([[sigma1**(-1)*m1+sigma3**(-1)*t],[sigma2**(-1)*m2-sigma3**(-1)*t]])
mu_st=np.matmul(sigma_st,multiplier)

print(f"mu_st is:{mu_st} and sigma_st is:{sigma_st}")

#task 2
py_1=1-norm.cdf(0,m1-m2,np.sqrt(sigma1+sigma2+sigma3))#takes in square root
print(f"p(y=1) is:{py_1}")
```

A.2 Script for report results

```
import pandas as pd
import numpy as np
from scipy.stats import norm, truncnorm, multivariate_normal
from matplotlib import pyplot as plt
import time


def sample_s(m1, m2, sigma1, sigma2, sigma3, t):
    div=(sigma2**(-1)+sigma3**(-1))*(sigma1**(-1)+sigma3**(-1))-sigma3**(-2)
    sigma_st=np.array([[sigma2**(-1)+sigma3**(-1), sigma3**(-1)],
              [sigma3**(-1), sigma1**(-1)+sigma3**(-1)]])/div
    multiplier=np.array([[sigma1**(-1)*m1+sigma3**(-1)*t],[sigma2**(-1)*m2-sigma3**(-1)*t]])
    mu_st=np.matmul(sigma_st,multiplier)
    # sample s1, s2
    return multivariate_normal.rvs(mean=mu_st.flatten(), cov=sigma_st)


# compare question 2 in the quiz
def sample_t(s1, s2, y, sigma_t):
    if y == 1:
        return truncnorm.rvs(0, np.inf, loc=(s1-s2), scale=np.sqrt(sigma_t))
    else:
        return truncnorm.rvs(np.NINF, 0, loc=(s1-s2), scale=np.sqrt(sigma_t))


# used to predict new games
def marginal_y(mu1, mu2, sigma1, sigma2, sigmat):
    return 1-norm.cdf(0,mu1-mu2, np.sqrt(sigma1+sigma2+sigmat))


# calculate the marginal of y includeing the data bias
def marginal_y_biased(mu1, mu2, sigma1, sigma2, sigmat):
    # 0.3 standard deviations equals to 11.79%
    offset_mu = 0.25 * np.sqrt(sigma1+sigma2+sigmat)
    return 1-norm.cdf(0,mu1-mu2+offset_mu, np.sqrt(sigma1+sigma2+sigmat))
```

```python


def Gauss_mult(m1, m2, s1, s2):
    s = 1 / (1 / s1 + 1 / s2)
    m = (m1 / s1 + m2 / s2) * s
    return m, s


def Gauss_div(m1, m2, s1, s2):
    m, s = Gauss_mult(m1, m2, s1, -s2)
    return m, s


def Gauss_trunc(a, b, m1, s1):
    b_norm = (b - m1) / np.sqrt(s1)
    a_norm = (a - m1) / np.sqrt(s1)
    m= truncnorm.mean(a_norm, b_norm, loc=m1,scale=np.sqrt(s1))
    s=truncnorm.var(a_norm,b_norm,loc=m1,scale=np.sqrt(s1))
    return m, s


# the gibbs sampler should return mean1, std1, mean2 and std2 for the two players
# burn in and number of samples empirically chosen
def gibbs_sampler(prior_1, prior_2, y, sigma_t=2, t_=50, b=20, K=1000):
    # lists to store the samples
    s1 = []
    s2 = []

    # here is where the magic happens
    for i in range(K):
        s = sample_s(prior_1[0], prior_2[0], prior_1[1], prior_2[1], sigma_t, t_)
        s1_ = s[0]
        s1.append(s1_)
        s2_ = s[1]
        s2.append(s2_)
        t_ = sample_t(s1_, s2_, y, sigma_t)[0]

    # return the sample values without the burn period
    return s1[b:K], s2[b:K]


# Q4.2
def estimate_posterior(prior_1, prior_2, y, sigma_t=2):
    s1, s2 = gibbs_sampler(prior_1, prior_2, y, sigma_t=sigma_t)
    # estimate mean and std for the two players
    s1_m = estimate_mean(s1)
    s1_s = estimate_std(s1, s1_m)
    s2_m = estimate_mean(s2)
    s2_s = estimate_std(s2, s2_m)
    # return results posterior_1 and posterior_2
    return (s1_m, s1_s), (s2_m, s2_s)


# Q4.2
def estimate_mean(X):
    return 1/len(X) * sum(X)


def estimate_std(X, mean):
    return 1/len(X) * sum([(x - mean) ** 2 for x in X])


def q4():
    prior = (0.0, 1.0)
    k = 100
```

```
    s1, s2 = gibbs_sampler(prior, prior, y=1, b=0, K=k)
    s1_m = estimate_mean(s1)
    s1_s = estimate_std(s1, s1_m)
    s2_m = estimate_mean(s2)
    s2_s = estimate_std(s2, s2_m)
    plt.plot(range(0, k), s1,
             range(0, k), s2)
    plt.show()
    sigma_t = 2
    k = 100
    s1, s2 = gibbs_sampler((s1_m, s1_s), (s2_m, s2_s), 1, b=0, K=k)
    plt.plot(range(0, k), s1,
             range(0, k), s2)
    plt.show()
    prior = (0.0, 1.0)
    # test out different k values and for each plot the distribution and the
         estimated gaussian
    for k_ in [50, 100, 150, 500, 1000, 5000, 10000, 25000]:
        start = time.time()
        s1, s2 = gibbs_sampler(prior, prior, y=1, K=(k_+20))
        s1_m = estimate_mean(s1)
        s1_s = estimate_std(s1, s1_m)
        s2_m = estimate_mean(s2)
        s2_s = estimate_std(s2, s2_m)
        end = time.time()
        elapsed = end - start
        x = np.linspace(-5, 5, 100)
        plt.plot(x, norm.pdf(x, s1_m, s1_s), label="posterior s1")
        plt.hist(s1, density=1, label="samples gibbs")
        plt.xlabel("Number of gibbs samples: {}, Required time {:.2f} s".format(k_,
             elapsed))
        plt.legend()
        plt.show()
    prior = (0.0, 1.0)
    # run gibbs sampler and estimate posterior gaussians
    s1, s2 = gibbs_sampler(prior, prior, y=1, K=1000)
    s1_m = estimate_mean(s1)
    s1_s = estimate_std(s1, s1_m)
    s2_m = estimate_mean(s2)
    s2_s = estimate_std(s2, s2_m)
    # plot 4 gaussian distributions
    x = np.linspace(-5, 5, 100)
    plt.plot(x, norm.pdf(x, 0, 1), label="prior")
    plt.plot(x, norm.pdf(x, s1_m, s1_s), label="posterior s1")
    plt.plot(x, norm.pdf(x, s2_m, s2_s), label="posterior s2")
    plt.legend()
    plt.show()


def output_result(x):
    # output the binary result based on a winning probability
    if  x >= 0.5:
        y_ = 1
    else:
        y_ = -1
    return y_


def get_y(row):
    if row["score1"] == row["score2"]:
        return 0
    elif row["score1"] > row["score2"]:
        return 1
    else:
        return -1
```

```python


def q56(shuffle=False, gibbs=True, advanced_predictor=False):
    print("Using Seria A data")
    data = pd.read_csv('SerieA.csv')
    # create new row with y values
    data['y'] = data.apply(lambda x: get_y(x), axis=1)
    # filter out the games where the teams draw
    data = data.loc[data['y'] != 0]
    # create a list of all the teams and store their skill representation
    d_teams = data['team1'].unique()
    teams = pd.DataFrame(index=d_teams)
    # set starting values for the mean and the std of the skill representation
    teams['mean'] = 0.0
    teams['std'] = 1.0
    if shuffle:
        print("Shuffle data in advance for Q.5")
        data = data.sample(frac=1).reset_index(drop=True)
    correct_predictions = 0
    random_predictions = 0
    # run through all the matches
    for index, row in data.iterrows():
        # fetch current values for team1 and team2
        s1 = teams.loc[row['team1']]
        s2 = teams.loc[row['team2']]
        # get priors
        prior_1 = (s1['mean'], s1['std'])
        prior_2 = (s2['mean'], s2['std'])
        sigma_t = 2
        # Q6: predict who should win based on the model
        if advanced_predictor:
            y_ = output_result(marginal_y_biased(prior_1[0], prior_2[0], prior_1[1],
                 prior_2[1], sigma_t))
        else:
            y_ = output_result(marginal_y(prior_1[0], prior_2[0], prior_1[1],
                 prior_2[1], sigma_t))
        # fetch the actual result
        y = row['y']
        if y == 1:
            random_predictions += 1
        if y == y_:
            correct_predictions += 1
        # run sampling
        if gibbs:
            posterior_1, posterior_2 = estimate_posterior(prior_1, prior_2, y,
                 sigma_t=sigma_t)
        else:
            posterior_1, posterior_2 = moment_matching(prior_1, prior_2, y,
                 st=sigma_t)
        # rewrite values mean and std
        s1['mean'] = posterior_1[0]
        s1['std'] = posterior_1[1]
        s2['mean'] = posterior_2[0]
        s2['std'] = posterior_2[1]

    if advanced_predictor:
        print("Using advanced predictor")
    if gibbs:
        print("Using gibbs sampling")
    else:
        print("Using moment matching")
    print(f"The prediction rate is {correct_predictions / data.shape[0]}\nUsing
     random guessing it is {random_predictions / data.shape[0]}")
    teams = teams.sort_values(["mean", "std"], ascending=False)
    x = np.linspace(-10, 10, 100)
```

```
    for num, (index, row) in enumerate(teams.iterrows()):
        l = str(num + 1) + ". " + index
        plt.plot(x, norm.pdf(x, row['mean'], row['std']), label=l)
    plt.legend(title="Ranking", loc="upper left", bbox_to_anchor=(1,1),
         fontsize='x-small')
    plt.show()


def moment_matching(prior1, prior2, y, st=2):
    m1 = prior1[0]
    s1 = prior1[1]
    m2 = prior2[0]
    s2 = prior2[1]
    #we call this c
    mus2_ft_m=m2
    mus2_ft_s=s2
    mus1_ft_m=m1
    mus1_ft_s=s1
    muft_t_m=mus1_ft_m-mus2_ft_m
    muft_t_s=mus1_ft_s+mus2_ft_s+st
    # truncated gaussian approximation via moment-matching
    if y == -1:
        a = np.NINF
        b = 0
    else:
        a = 0
        b = 1000
    # q/mufxt_t approximates mut_fxt
    q_m, q_s = Gauss_trunc(a, b, muft_t_m, muft_t_s)
    p_m, p_s = Gauss_div(q_m, muft_t_m , q_s, muft_t_s)
    #see notes
    muft_s1_m=m2+p_m
    muft_s1_s=p_s+st+s2
    s1_m, s1_s = Gauss_mult(muft_s1_m, m1, muft_s1_s, s1)
    muft_s2_m=m1-p_m
    muft_s2_s=p_s+st+s1
    s2_m, s2_s = Gauss_mult(muft_s2_m, m2, muft_s2_s, s2)
    return (s1_m, s1_s), (s2_m, s2_s)


def q78():
    prior = (0.0, 1.0)
    # run gibbs sampler and estimate posterior gaussians
    s1, s2 = gibbs_sampler(prior, prior, y=1, K=1000)
    s1_m = estimate_mean(s1)
    s1_s = estimate_std(s1, s1_m)
    s2_m = estimate_mean(s2)
    s2_s = estimate_std(s2, s2_m)
    # run matching
    s1_, s2_ = moment_matching(prior, prior, y=1)
    # plot 4 gaussian distributions
    x = np.linspace(-4, 4, 100)
    plt.plot(x, norm.pdf(x, 0, 1), label="prior")
    plt.plot(x, norm.pdf(x, s1_m, s1_s), label="posterior s1 gibbs")
    plt.plot(x, norm.pdf(x, s1_[0], s1_[1]), label="posterior s1 moment-matching")
    # plt.plot(x, norm.pdf(x, s2_m, s2_s), label="posterior s2")
    plt.legend()
    plt.show()
    plt.plot(x, norm.pdf(x, 0, 1), label="prior")
    plt.plot(x, norm.pdf(x, s1_[0], s1_[1]), label="posterior s1 moment-matching")
    plt.plot(x, norm.pdf(x, s2_[0], s2_[1]), label="posterior s2 moment-matching")
    plt.legend()
    plt.show()
```

```python
def q9():
    print("Using the ATP dataset")
    data = pd.read_csv('ATP.csv')
    # only consider events from the last 5 years
    data = data.loc[data["tourney_date"] > 20150000, ["winner_id", "winner_name",
        "loser_id", "loser_name", "tourney_date"]]
    d_teams = np.unique(np.append(data['winner_id'].unique(),
        data['loser_id'].unique()))
    teams = pd.DataFrame(index=d_teams)
    # set starting values for the mean and the std of the skill representation
    teams['mean'] = 0.0
    teams['std'] = 1.0
    correct_predictions = 0
    # run through all the matches
    for index, row in data.iterrows():
        # Switch betwwen Gibbs and moment-matching
        gibbs = False
        # fetch current values for team1 and team2
        s1 = teams.loc[row['winner_id']]
        s2 = teams.loc[row['loser_id']]
        # get priors
        prior_1 = (s1['mean'], s1['std'])
        prior_2 = (s2['mean'], s2['std'])
        sigma_t = 2
        # predict next game
        y_ = output_result(marginal_y(prior_1[0], prior_2[0], prior_1[1],
            prior_2[1], sigma_t))
        # in this dataet there is no home or away s1 is per definition the winner
            of the game
        y = 1
        if y == y_:
            correct_predictions += 1
        # run sampling
        if gibbs:
            posterior_1, posterior_2 = estimate_posterior(prior_1, prior_2, y,
                sigma_t=sigma_t)
        else:
            posterior_1, posterior_2 = moment_matching(prior_1, prior_2, y,
                st=sigma_t)
        # rewrite values mean and std
        s1['mean'] = posterior_1[0]
        s1['std'] = posterior_1[1]
        s2['mean'] = posterior_2[0]
        s2['std'] = posterior_2[1]
    print(f"The prediction rate is {correct_predictions / data.shape[0]}")


def main():
    q4()
    q56()
    print("\n")
    q56(shuffle=True)
    print("\n")
    q56(gibbs=False)
    print("\n")
    q56(gibbs=False, advanced_predictor=True)
    print("\n")
    q78()
    q9()


if __name__ == '__main__':
    main()
```

B Corollary 1 Matrices

The matrices used for the application of Corollary 1 in Section Q.3 are given in standard notation as:

$$\boldsymbol{A} = \begin{pmatrix} 1 & -1 \end{pmatrix} \tag{22}$$

$$\boldsymbol{\Sigma_s} = \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix} \tag{23}$$

$$\boldsymbol{\Sigma}_{t|s} = \sigma_t^2 \tag{24}$$

$$\boldsymbol{\mu}_s = \begin{pmatrix} \mu_1 & \mu_2 \end{pmatrix}. \tag{25}$$

C Corollary 2 Matrices

C.1 First usage of Corollary 2

The first usage of Corollary 2 was in (14) and we used the following matrices:

$$\boldsymbol{\mu_s} = (\mu_1, \mu_2)^T, \quad \boldsymbol{\Sigma_s} \overset{s_1 \perp\!\!\!\perp s_2}{=} \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix} \tag{26}$$

$$\boldsymbol{A} = (1, -1), \quad \boldsymbol{b} = 0, \quad \boldsymbol{\Sigma_{t|s}} = \sigma_t^2 \tag{27}$$

C.2 Second usage of Corollary 2

The matrices created for the second application of Corollary 2 in (16) is given in standard notation as:

$$\boldsymbol{A} = -1, \quad \boldsymbol{b} = s_1 \tag{28}$$

$$\boldsymbol{\mu_a} = \mu_2, \quad \boldsymbol{\Sigma_{b|a}} = \sigma_t^2. \tag{29}$$

C.3 Third usage of Corollary 2

The third usage of Corollary 2 was in (19) and utilized the following vectors.

$$A = 1, \quad b = \mu_2 \tag{30}$$

$$\mu_a = \hat{\mu}, \quad \Sigma_{b|a} = \sigma_t^2 + \sigma_2^2. \tag{31}$$