Assignment 2:
Stochastic Variational Inference in the TrueSkill Model

STA414/STA2014 and CSC412/CSC2506 Winter 2020
David Duvenaud and Jesse Bettencourt
Hongbo Du (1003568089)

March 15, 2020

The goal of this assignment is to get you familiar with the basics of Bayesian inference in large models with continuous latent variables, and the basics of stochastic variational inference.

Background We'll implement a variant of the TrueSkill model, a player ranking system for competitive games originally developed for Halo 2. It is a generalization of the Elo rating system in Chess. For the curious, the original 2007 NIPS paper introducing the trueskill paper can be found here: http://papers.nips.cc/paper/3079-trueskilltm-a-bayesian-skill-rating-system.pdf

This assignment is based on one developed by Carl Rasmussen at Cambridge for his course on probabilistic machine learning: http://mlg.eng.cam.ac.uk/teaching/4f13/1920/

0.1 Model definition

We'll consider a slightly simplified version of the original trueskill model. We assume that each player has a true, but unknown skill $z_i \in \mathbb{R}$. We use N to denote the number of players.

The prior. The prior over each player's skill is a standard normal distribution, and all player's skills are *a prior* independent.

The likelihood. For each observed game, the probability that player i beats player j, given the player's skills z_A and z_B, is:

$$p(\text{A beat B}|z_A, z_B) = \sigma(z_i - z_j)$$

where

$$\sigma(y) = \frac{1}{1 + \exp(-y)}$$

There can be more than one game played between a pair of players, and in this case the outcome of each game is independent given the players' skills. We use M to denote the number of games.

The data. The data will be an array of game outcomes. Each row contains a pair of player indices. The first index in each pair is the winner of the game, the second index is the loser. If there were M games played, then the array has shape $M \times 2$.

1 Implementing the model [10 points]

(a) **[2 points]** Implement a function `log_prior` that computes the log of the prior over all player's skills. Specifically, given a $K \times N$ array where each row is a setting of the skills for all N players, it returns a $K \times 1$ array, where each row contains a scalar giving the log-prior for that set of skills.

```
function log_prior(zs)
  """
  Computes the log of the prior over all player’s skills.
  """
  return factorized_gaussian_log_density(0, log.(1), zs)
end
```

(b) **[3 points]** Implement a function `logp_a_beats_b` that, given a pair of skills z_a and z_b evaluates the log-likelihood that player with skill z_a beat player with skill z_b under the model detailed above. To ensure numerical stability, use the function `log1pexp` that computes $\log(1 + \exp(x))$ in a numerically stable way. This function is provided by `StatsFuns.jl` and imported already, and also by Python's numpy.

```
function logp_a_beats_b(za,zb)
  """
  Computes the log-likelihood that player with skill za beat player with skill zb.
  """
  return -(log1pexp.(zb .- za))
end
```

(c) **[3 points]** Assuming all game outcomes are i.i.d. conditioned on all players' skills, implement a function `all_games_log_likelihood` that takes a batch of player skills `zs` and a collection of observed games `games` and gives a batch of log-likelihoods for those observations. Specifically, given a $K \times N$ array where each row is a setting of the skills for all N players, and an $M \times 2$ array of game outcomes, it returns a $K \times 1$ array, where each row contains a scalar giving the log-likelihood of all games for that set of skills. Hint: You should be able to write this function without using for loops, although you might want to start that way to make sure what you've written is correct. If A is an array of integers, you can index the corresponding entries of another matrix B for every entry in A by writing `B[A]`.

```
function all_games_log_likelihood(zs,games)
  """
  Computes the log-likelihoods for those observed games.
  """
  #games = convert(ArrayInt64, games)
  zs_a = zs[games[:, 1], :]
  zs_b = zs[games[:, 2], :]
  likelihoods = logp_a_beats_b(zs_a, zs_b)
  return sum(likelihoods, dims = 1)
end
```

(d) **[2 points]** Implement a function `joint_log_density` which combines the log-prior and log-likelihood of the observations to give $p(z_1, z_2, \ldots, z_N, \text{all game outcomes})$

```
function joint_log_density(zs,games)
  """
  Combines the log-prior and log-likelihood of the observations.
  """
  return prod.(log_prior(zs) .+ all_games_log_likelihood(zs, games))
end
```

The following code contains the dimensional test for the above functions.

```
@testset "Test shapes of batches for likelihoods" begin
  B = 15 # number of elements in batch
```

```
  N = 4 # Total Number of Players
  test_zs = randn(4,15)
  test_games = [1 2; 3 1; 4 2] # 1 beat 2, 3 beat 1, 4 beat 2
  @test size(test_zs) == (N,B)
  #batch of priors
  @test size(log_prior(test_zs)) == (1,B)
  # loglikelihood of p1 beat p2 for first sample in batch
  @test size(logp_a_beats_b(test_zs[1,1],test_zs[2,1])) == ()
  # loglikelihood of p1 beat p2 broadcasted over whole batch
  @test size(logp_a_beats_b.(test_zs[1,:],test_zs[2,:])) == (B,)
  # batch loglikelihood for evidence
  @test size(all_games_log_likelihood(test_zs,test_games)) == (1,B)
  # batch loglikelihood under joint of evidence and
  @test size(joint_log_density(test_zs,test_games)) == (1,B)
end
```

```
Test Summary: | Pass Total
Test shapes of batches for likelihoods | 6 6
```

2 Examining the posterior for only two players and toy data [10 points]

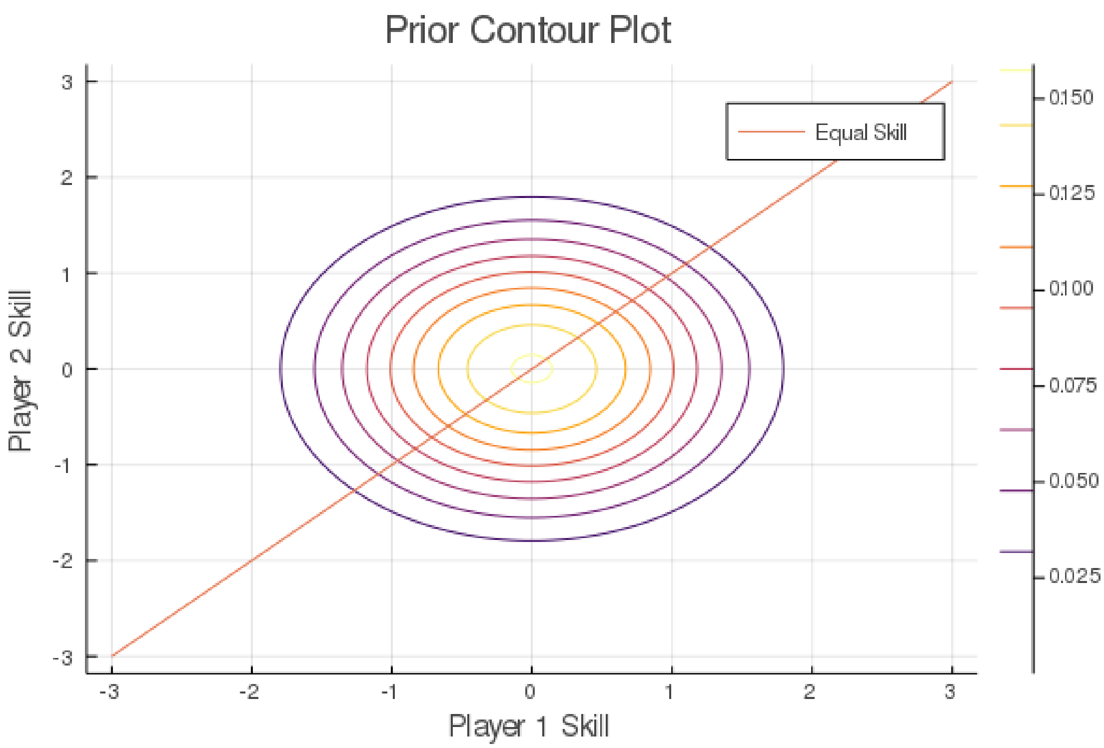
To get a feel for this model, we'll first consider the case where we only have 2 players, A and B. We'll examine how the prior and likelihood interact when conditioning on different sets of games.

Provided in the starter code is a function `skillcontour!` which evaluates a provided function on a grid of z_A and z_B's and plots the isocontours of that function. As well there is a function `plot_line_equal_skill!`. We have included an example for how you can use these functions.

We also provided a function `two_player_toy_games` which produces toy data for two players. I.e. `two_player_toy_games(5,3)` produces a dataset where player A wins 5 games and player B wins 3 games.

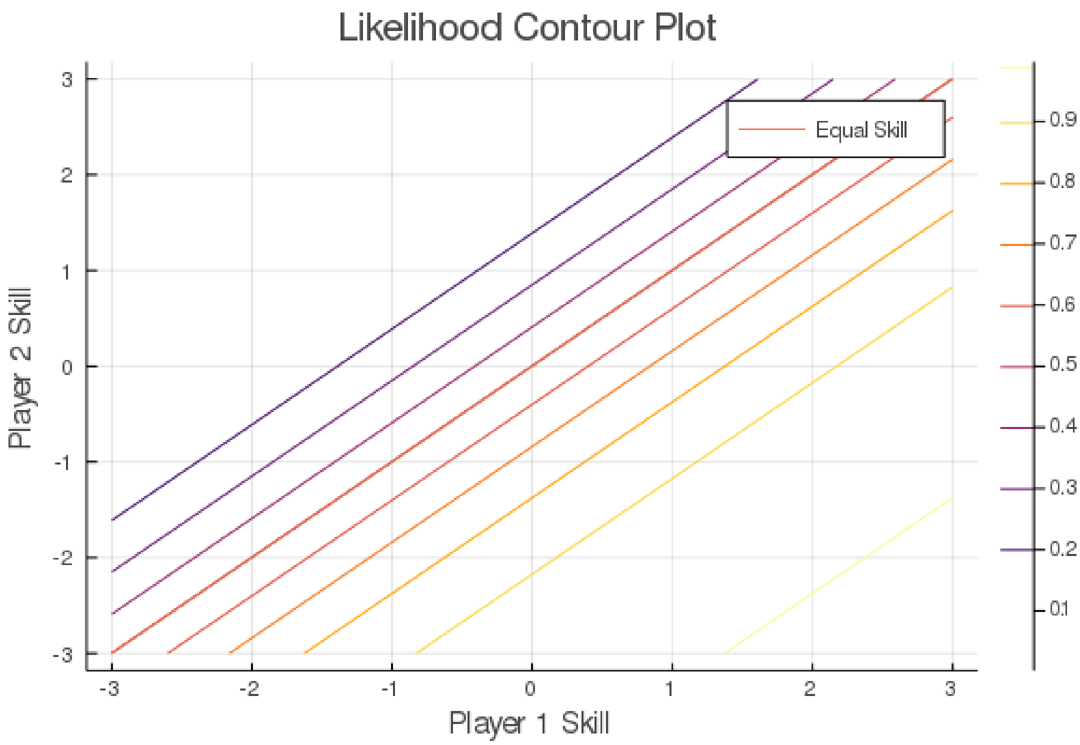
(a) **[2 points]** For two players A and B, plot the isocontours of the joint prior over their skills. Also plot the line of equal skill, $z_A = z_B$. Hint: you've already implemented the **log** of the likelihood function.

```
# plot prior contours
plot(title="Example prior Contour Plot",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )

log_prior_(zs) = exp(log_prior(zs))
skillcontour!(log_prior_)
plot_line_equal_skill!()
savefig(joinpath("plots","prior"))
```



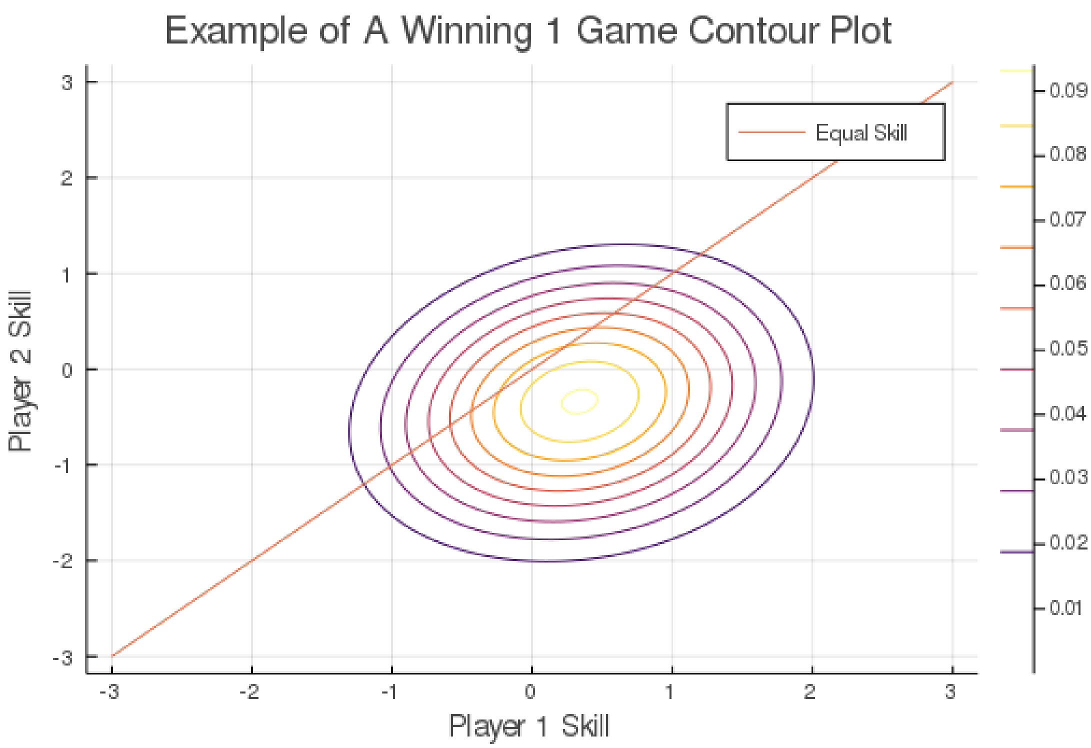
(b) **[2 points]** Plot the isocontours of the likelihood function. Also plot the line of equal skill, $z_A = z_B$.

```
# plot likelihood contours
plot(title="Likelihood Contour Plot",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )

logp_a_beats_b_(z_comb) = exp(logp_a_beats_b(z_comb[1], z_comb[2]))
skillcontour!(logp_a_beats_b_)
plot_line_equal_skill!()
savefig(joinpath("plots","likelihood"))
```



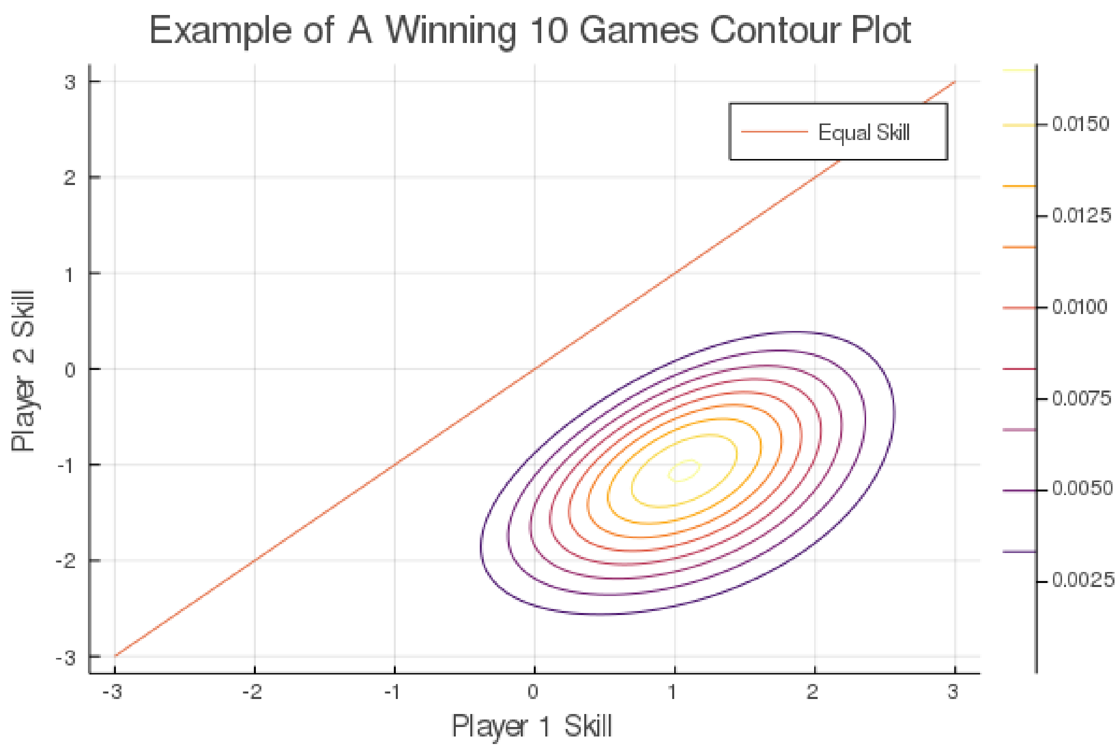
(c) **[2 points]** Plot isocountours of the joint posterior over z_A and z_B given that player A beat player B in one match. Since the contours don't depend on the normalization constant, you can simply plot the isocontours of the log of joint distribution of $p(z_A, z_B, \text{A beat B})$ Also plot the line of equal skill, $z_A = z_B$.

```
# Convenience function for producing toy games between two players.
two_player_toy_games(p1_wins, p2_wins) = vcat([repeat([1,2]',p1_wins), repeat([2,1]', p2_wins)]...)

# plot joint contours with player A winning 1 game
plot(title="Example of A Winning 1 Game Contour Plot",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )
game_outcome_1 = two_player_toy_games(1,0)
joint_log_density_(zs) = joint_log_density(zs, game_outcome_1)
game_1_0(zs) = exp.(joint_log_density_(zs))
skillcontour!(game_1_0)
plot_line_equal_skill!()
savefig(joinpath("plots","A_Wins_1_Game"))
```



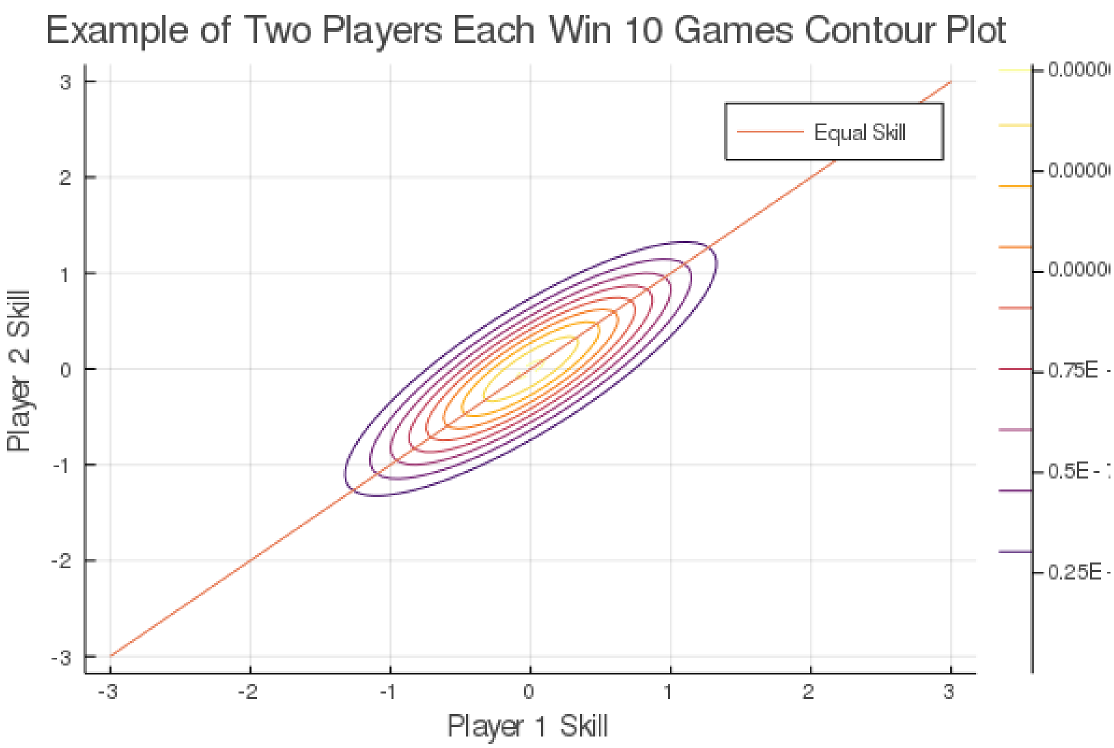
(d) **[2 points]** Plot isocountours of the joint posterior over z_A and z_B given that 10 matches were played, and player A beat player B all 10 times. Also plot the line of equal skill, $z_A = z_B$.

```
# plot joint contours with player A winning 10 games
plot(title="Example of A Winning 10 Games Contour Plot",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )
game_outcome_10 = two_player_toy_games(10,0)
game_10_0(zs) = exp(joint_log_density(zs, game_outcome_10))
skillcontour!(game_10_0)
plot_line_equal_skill!()
savefig(joinpath("plots","A_Wins_10_Games"))
```



(e) **[2 points]** Plot isocountours of the joint posterior over z_A and z_B given that 20 matches were played, and each player beat the other 10 times. Also plot the line of equal skill, $z_A = z_B$.

```
# plot joint contours with player A winning 10 games and player B winning 10 games
plot(title="Example of Two Players Each Win 10 Games Contour Plot",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )
game_outcome_10_10 = two_player_toy_games(10,10)
game_10_10(zs) = exp(joint_log_density(zs, game_outcome_10_10))
skillcontour!(game_10_10)
plot_line_equal_skill!()
savefig(joinpath("plots","Each_Wins_10_Games"))
```



3 Stochastic Variational Inference on Two Players and Toy Data [18 points]

One nice thing about a Bayesian approach is that it separates the model specification from the approximate inference strategy. The original Trueskill paper from 2007 used message passing. Carl Rasmussen's assignment uses Gibbs sampling, a form of Markov Chain Monte Carlo. We'll use gradient-based stochastic variational inference, which wasn't invented until around 2014.

In this question we will optimize an approximate posterior distribution with stochastic variational inference to approximate the true posterior.

(a) **[5 points]** Implement a function `elbo` which computes an unbiased estimate of the evidence lower bound. As discussed in class, the ELBO is equal to the KL divergence between the true posterior $p(z|\text{data})$, and an approximate posterior, $q_\phi(z|\text{data})$, plus an unknown constant. Use a fully-factorized Gaussian distribution for $q_\phi(z|\text{data})$. This estimator takes the following arguments:

- `params`, the parameters ϕ of the approximate posterior $q_\phi(z|\text{data})$.
- A function `logp`, which is equal to the true posterior plus a constant. This function must take a batch of samples of z. If we have N players, we can consider B-many samples from the joint over all players' skills. This batch of samples `zs` will be an array with dimensions (N, B).
- `num_samples`, the number of samples to take.

This function should return a single scalar. Hint: You will need to use the reparamterization trick when sampling `zs`.

```julia
function elbo(params,logp,num_samples)
  """
  Computes an unbiased estimate of the evidence lower bound.
  """
  samples = exp.(params[2]) .* randn(length(params[1]), num_samples) .+ params[1]
  logp_estimate = logp(samples)
  logq_estimate = factorized_gaussian_log_density(params[1], params[2], samples) # q(z|x)
  return mean(logp_estimate - logq_estimate)
end
```

(b) **[2 points]** Write a loss function called `neg_toy_elbo` that takes variational distribution parameters and an array of game outcomes, and returns the negative elbo estimate with 100 samples.

```julia
# Conveinence function for taking gradients
function neg_toy_elbo(params; games = two_player_toy_games(1,0), num_samples = 100)
  """
  Returns the -elbo estimate with num_samples many samples from q.
  """
  logp(zs) = joint_log_density(zs, games)
  return -elbo(params,logp, num_samples)
end
```

(c) **[5 points]** Write an optimization function called `fit_toy_variational_dist` which takes initial variational parameters, and the evidence. Inside it will perform a number of iterations of gradient descent where for each iteration :

(a) Compute the gradient of the loss with respect to the parameters using automatic differentiation.

(b) Update the parameters by taking an `lr`-scaled step in the direction of the descending gradient.

(c) Report the loss with the new parameters (using `@info` or print statements)

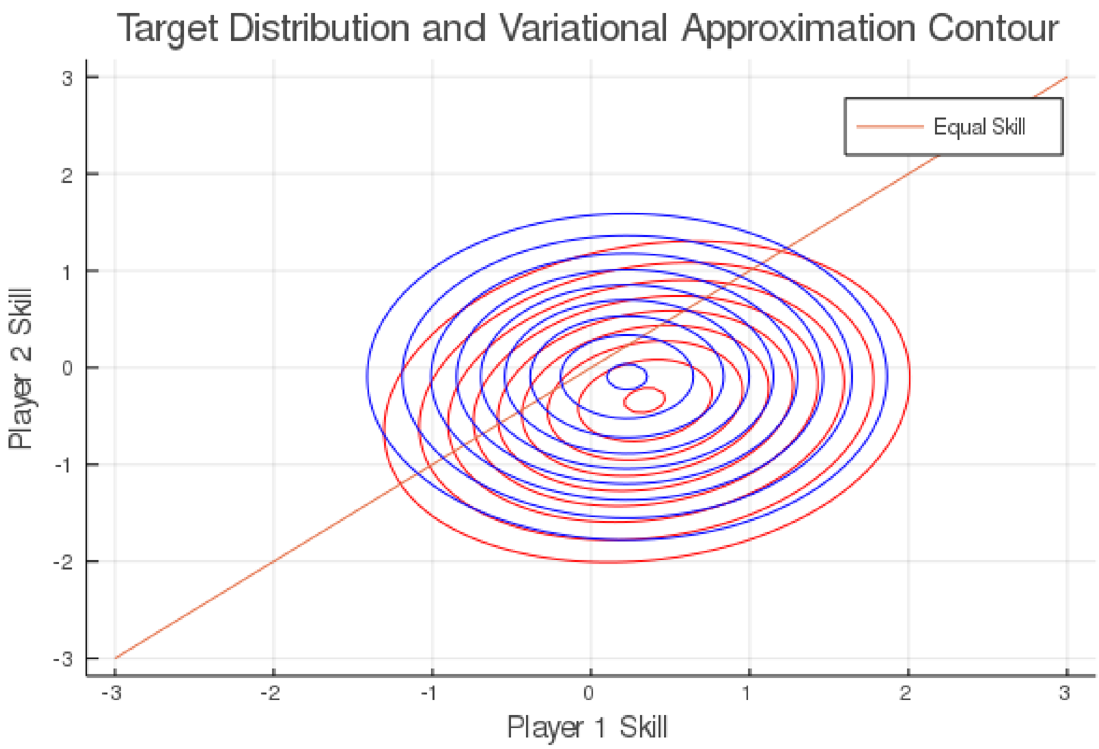
(d) On the same set of axes plot the target distribution in red and the variational approximation in blue.

Return the parameters resulting from training.

```julia
# Toy game
num_players_toy = 2
toy_mu = [-2., 3.] # Initial mu, can initialize randomly!
toy_ls = [0.5., 0.] # Initial log sigma, can initialize randomly!
toy_params_init = (toy_mu, toy_ls)

function fit_toy_variational_dist(init_params, toy_evidence; num_itrs=200, lr= 1e-2,
    num_q_samples = 10)
  params_cur = init_params

  for i in 1:num_itrs
    grad_params = gradient(params_cur -> neg_toy_elbo(params_cur; games = toy_evidence,
        num_samples = num_q_samples), params_cur)[1]
    params_cur = params_cur .- lr .* grad_params

    @info "loss: $(neg_toy_elbo(params_cur; games = toy_evidence, num_samples =
        num_q_samples))"

    plot(title="Target Distribution and Variational Approximation Contour",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill");

    samples = exp.(params_cur[2]) .* randn(length(params_cur[1]), num_q_samples) .+
        params_cur[1]

    target_dist(params_cur) = exp(joint_log_density(params_cur, toy_evidence))
    var_approx(samples) = exp(factorized_gaussian_log_density(params_cur[1], params_cur
        [2], samples))

    skillcontour!(target_dist, colour=:red)
    plot_line_equal_skill!()
    display(skillcontour!(var_approx, colour=:blue))
  end
  return params_cur
end
```

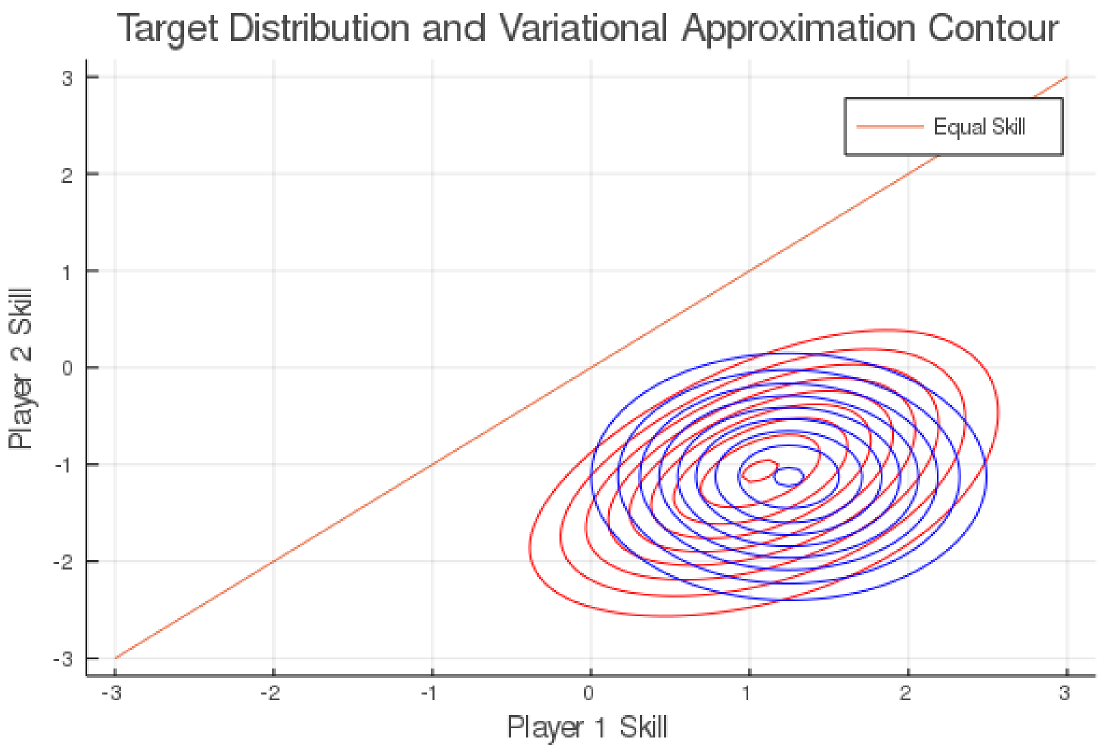
(d) **[2 points]** Initialize a variational distribution parameters and optimize them to approximate the joint where we observe player A winning 1 game. Report the final loss. Also plot the optimized variational approximation contours (in blue) aand the target distribution (in red) on the same axes.

```
# Fit q with SVI observing player A winning 1 game
fit_toy_variational_dist(toy_params_init, game_outcome_1; num_itrs=200, lr= 1e-2,
    num_q_samples = 10)
```



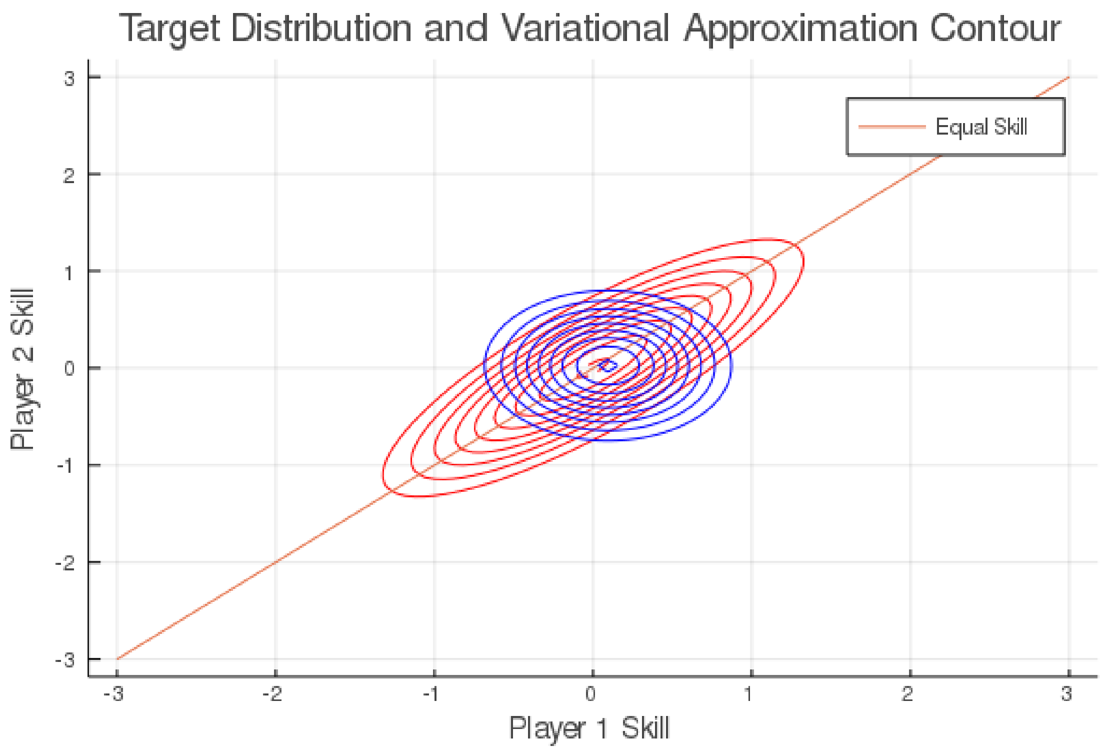
(e) **[2 points]** Initialize a variational distribution parameters and optimize them to approximate the joint where we observe player A winning 10 games. Report the final loss. Also plot the optimized variational approximation contours (in blue) aand the target distribution (in red) on the same axes.

```
# Fit q with SVI observing player A winning 10 games
param10 = fit_toy_variational_dist(toy_params_init, game_outcome_10; num_itrs=200, lr= 1
    e-2, num_q_samples = 10)
```



(f) **[2 points]** Initialize a variational distribution parameters and optimize them to approximate the joint where we observe player A winning 10 games and player B winning 10 games. Report the final loss. Also plot the optimized variational approximation contours (in blue) aand the target distribution (in red) on the same axes.

```
# Fit q with SVI observing player A winning 10 games and player B winning 10 games
param10_10 = fit_toy_variational_dist(toy_params_init, game_outcome_10_10; num_itrs=200,
     lr= 1e-2, num_q_samples = 10)
```



For all plots, label both axes.

4 Approximate inference conditioned on real data [24 points]

Load the dataset from `tennis_data.mat` containing two matrices:

- W is a 107 by 1 matrix, whose i'th entry is the name of player i.
- G is a 1801 by 2 matrix of game outcomes (actually tennis matches), one row per game. The first column contains the indices of the players who won. The second column contains the indices of the player who lost.

Compute the following using your code from the earlier questions in the assignment, but conditioning on the tennis match outcomes:

(a) **[1 point]** For any two players i and j, $p(z_i, z_j|\text{all games})$ is always proportional to $p(z_i, z_j, \text{all games})$. In general, are the isocontours of $p(z_i, z_j|\text{all games})$ the same as those of $p(z_i, z_j|\text{games between } i \text{ and } j)$? That is, do the games between other players besides i and j provide information about the skill of players i and j? A simple yes or no suffices.

Hint: One way to answer this is to draw the graphical model for three players, i, j, and k, and the results of games between all three pairs, and then examine conditional independencies. If you do this, there's no need to include the graphical models in your assignment.

Answer: Yes.

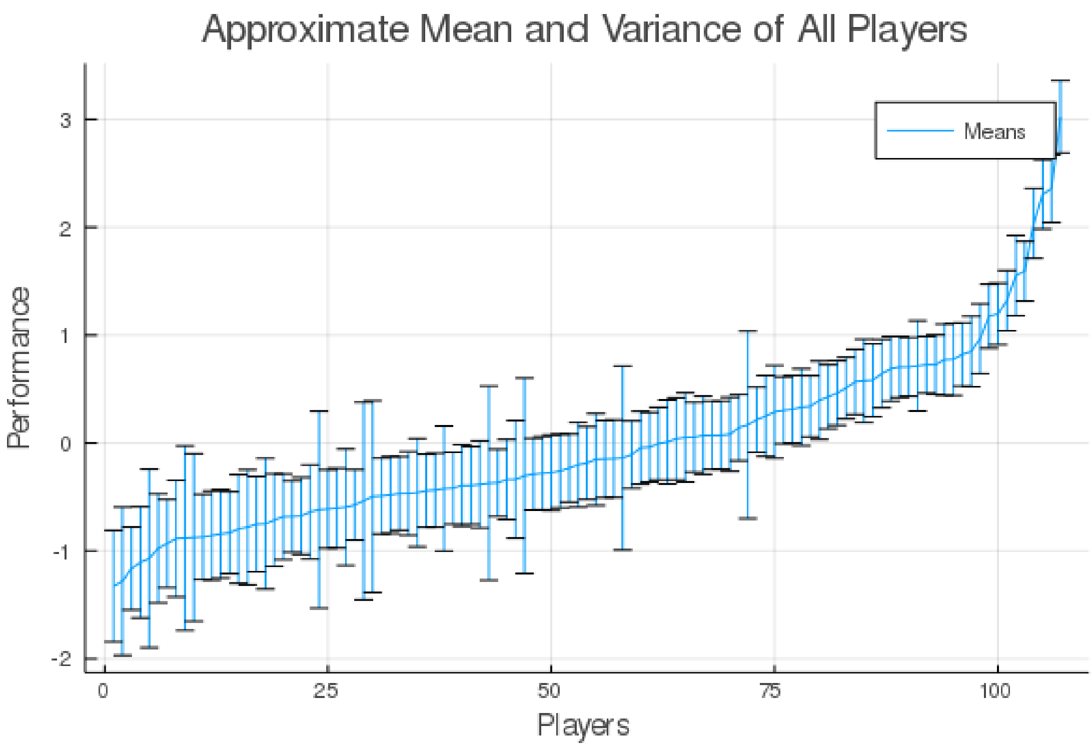
(b) **[5 points]** Write a new optimization function `fit_variational_dist` like the one from the previous question except it does not plot anything. Initialize a variational distribution and fit it to the joint distribution with all the observed tennis games from the dataset. Report the final negative ELBO estimate after optimization.

```julia
# Load the Data
using MAT
vars = matread("tennis_data.mat")
player_names = vars["W"]
tennis_games = Int.(vars["G"])
num_players = length(player_names)
print("Loaded data for $num_players players")


function fit_variational_dist(init_params, tennis_games; num_itrs=200, lr= 1e-2, num_q_samples = 10)
  params_cur = init_params

  for i in 1:num_itrs
    grad_params = gradient(params_cur -> neg_toy_elbo(params_cur; games = tennis_games, num_samples = num_q_samples), params_cur)[1]
    params_cur = params_cur .- lr .* grad_params

    @info "loss: $(neg_toy_elbo(params_cur; games = tennis_games, num_samples = num_q_samples))" # report objective value with current parameters
  end
  return params_cur
end


# nitialize variational family
init_mu = randn(107) #random initialziation
init_log_sigma = rand(107) # random initialziation
init_params = (init_mu, init_log_sigma)


# Train variational distribution
trained_params = fit_variational_dist(init_params, tennis_games)
```

The final negative ELBO estimate after optimization is 1143.492641999895

(c) **[2 points]** Plot the approximate mean and variance of all players, sorted by skill. For example, in Julia, you can use: `perm = sortperm(means); plot(means[perm], yerror=exp.(logstd[perm]))` There's no need to include the names of the players.

```
# (c) plot of the approx means and variances
mu_tennis = trained_params[1]
logstd = trained_params[2]
perm = sortperm(mu_tennis)

plot(mu_tennis[perm], yerror=exp.(logstd[perm]),
title="Approximate Mean and Variance of All Players",
xlabel = "Players",
ylabel = "Performance",
label = "Means")
savefig(joinpath("plots","PlayerSorted"))
```



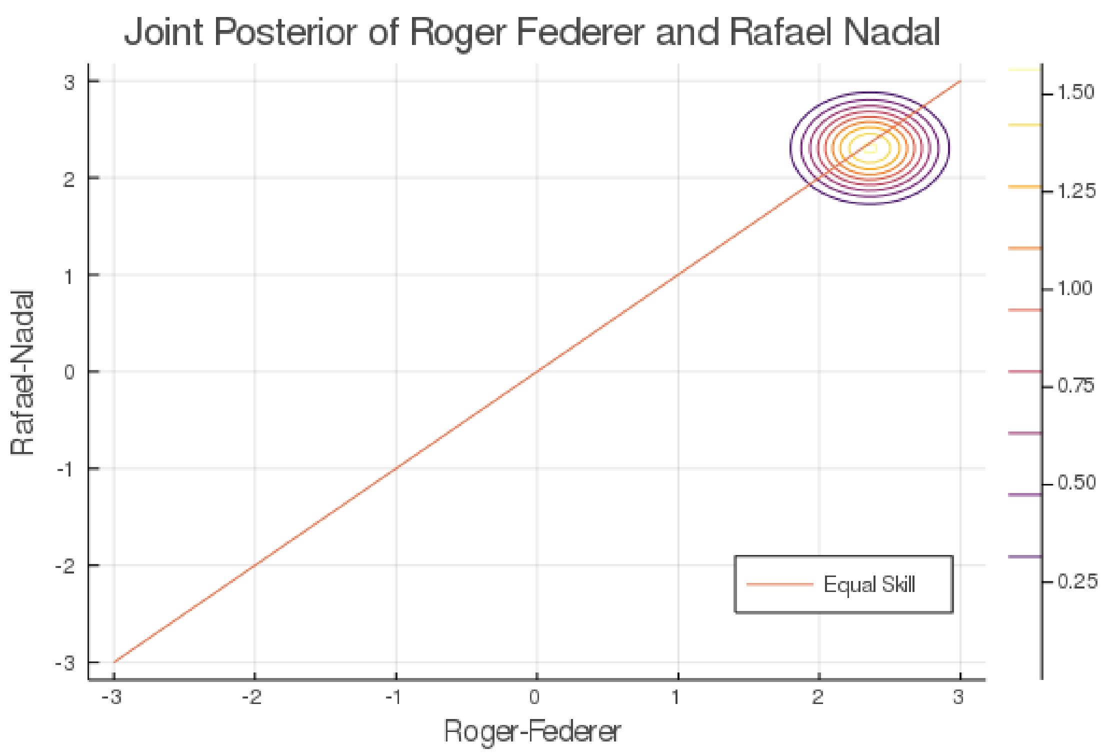
(d) **[2 points]** List the names of the 10 players with the highest mean skill under the variational model.

```
# 10 players with highest mean skill under variational model
top_ten = player_names[perm[98:107,]]

# Top 10 players from the highest mean to the lowest mean
julia> reverse(top_ten)
10-element Array{Any,1}:
 "Novak-Djokovic"
 "Roger-Federer"
 "Rafael-Nadal"
 "Andy-Murray"
 "David-Ferrer"
 "Robin-Soderling"
 "Jo-Wilfried-Tsonga"
 "Tomas-Berdych"
 "Juan-Martin-Del-Potro"
```

```
"Richard-Gasquet"
```

(e) **[3 points]** Plot the joint posterior over the skills of Roger Federer and Rafael Nadal.

```
RF = findall(x -> x == "Roger-Federer", player_names)[1][1]
RN = findall(x -> x == "Rafael-Nadal", player_names)[1][1]

mu_RF = trained_params[1][RF]
mu_RN = trained_params[1][RN]
mu_FN = [mu_RF, mu_RN]

sig_RF = trained_params[2][RF]
sig_RN = trained_params[2][RN]
sig_FN = [sig_RF, sig_RN]

param_FN = (mu_FN, sig_FN)

plot(title="Joint Posterior of Roger Federer and Rafael Nadal",
xlabel = "Roger-Federer",
ylabel = "Rafael-Nadal",
legend=:bottomright)
samples_FN = exp.(param_FN[2]) .* randn(length(param_FN[1]), 10) .+ param_FN[1]
var_approx_FN(samples_FN) = exp(factorized_gaussian_log_density(param_FN[1], param_FN[2], samples_FN))
skillcontour!(var_approx_FN)
plot_line_equal_skill!()
savefig(joinpath("plots","FedererNadal"))
```



(f) **[5 points]** Derive the exact probability under a factorized Guassian over two players' skills that one has higher skill than the other, as a function of the two means and variances over their skills.

- Hint 1: Use a linear change of variables $y_A, y_B = z_A - z_B, z_B$. What does the line of equal skill look like after this transformation?

- Hint 2: If $X \sim \mathcal{N}(\mu, \Sigma)$, then $AX \sim \mathcal{N}(A\mu, A\Sigma A^\top)$ where A is a linear transformation.
- Hint 3: Marginalization in Gaussians is easy: if $X \sim \mathcal{N}(\mu, \Sigma)$, then the ith element of X has a marginal distribution $X_i \sim \mathcal{N}(\mu_i, \Sigma_{ii})$

Answer: Take hint 1 and 2, consider that we apply a linear transformation A such that $A\boldsymbol{z} = \boldsymbol{y}$. Equivalently,

$$A\begin{bmatrix} z_A \\ z_B \end{bmatrix} = \begin{bmatrix} y_A \\ y_B \end{bmatrix} = \begin{bmatrix} z_A - z_B \\ z_B \end{bmatrix}$$

where

$$A = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}.$$

So if $\boldsymbol{z} \sim \text{MVN}(\mu_z, \Sigma_z)$, then $\boldsymbol{y} \sim \text{MVN}(A\mu_z, A\Sigma_z A^\top)$. By hint 3, we marginalize y_A such that $y_A \sim \mathcal{N}([A\mu_z]_1, [A\Sigma_z A^\top]_{11})$. Therefore, if we want to derive the exact probability under a factorized Guassian over two players' skills that one has higher skill than the other, that is, compute $\mathbb{P}(z_A - z_B > 0)$.

$$\begin{aligned} \mathbb{P}(z_A - z_B > 0) &= \mathbb{P}(y_A > 0) \\ &= 1 - \mathbb{P}(y_A = 0) \\ &= 1 - \Phi\left(\frac{y_A - [A\mu_z]_1}{\sqrt{[A\Sigma_z A^\top]_{11}}}\right) \\ &= 1 - \Phi\left(\frac{-[A\mu_z]_1}{\sqrt{[A\Sigma_z A^\top]_{11}}}\right) \end{aligned}$$

where Φ is the cumulative distribution function (CDF).

Therefore the exact probability is $\mathbb{P}(z_A - z_B > 0) = 1 - \Phi\left(\frac{-[A\mu_z]_1}{\sqrt{[A\Sigma_z A^\top]_{11}}}\right)$.

(g) **[2 points]** Compute the probability under your approximate posterior that Roger Federer has higher skill than Rafael Nadal. Compute this quantity exactly, and then estimate it using simple Monte Carlo with 10000 examples.

```
# CDF method
porb_CDF_F_wins = 1 - cdf(Normal(mean(mu_RF-mu_RN), 1), 0)

# MC simulation
num_sample_MC = 10000
sample_MC_FN = exp.(param_FN[2]) .* randn(length(param_FN[1]), num_sample_MC) .+
    param_FN[1]
sample_MC_RF = sample_MC_FN[1,:]
sample_MC_RN = sample_MC_FN[2,:]
prob_MC_F_wins = length(findall(x -> x > 0, sample_MC_RF .- sample_MC_RN)) /
    num_sample_MC

julia> prob_CDF_F_wins
0.5213126059566976
julia> prob_MC_F_wins
0.5481
```

The probability of CDF is 0.5213126059566976, and by using Monte Carlo simulation is estimated as 0.5481.

(h) **[2 points]** Compute the probability that Roger Federer is better than the player with the lowest mean skill. Compute this quantity exactly, and then estimate it using simple Monte Carlo with 10000 examples.

```
# CDF method
porb_CDF_N_wins = 1 - cdf(Normal(mean(mu_RN-mu_RF), 1), 0)

# MC simulation
num_sample_MC = 10000
sample_MC_FN = exp.(param_FN[2]) .* randn(length(param_FN[1]), num_sample_MC) .+ param_FN[1]
sample_MC_RF = sample_MC_FN[1,:]
sample_MC_RN = sample_MC_FN[2,:]
prob_MC_N_wins = length(findall(x -> x < 0, sample_MC_RF .- sample_MC_RN)) / num_sample_MC

julia> prob_CDF_N_wins
0.4786873940433024
julia> prob_MC_N_wins
0.4573
```

The probability of CDF is 0.4786873940433024, and by using Monte Carlo simulation is estimated as 0.4573.

(i) **[2 points]** Imagine that we knew ahead of time that we were examining the skills of top tennis players, and so changed our prior on all players to Normal(10, 1). Which answers in this section would this change? No need to show your work, just list the letters of the questions whose answers would be different in expectation.
Answer: (b), (c), and (e).