Assignment 2

Student Name Sheng Gao and #1002093584

March 22, 2020

A2_src.jl functions are moved to here.

```
using Plots
using StatsFuns: log1pexp

function factorized_gaussian_log_density(mu,logsig,xs)
  """
  mu and logsig either same size as x in batch or same as whole batch
  returns a 1 x batchsize array of likelihoods
  """
  σ = exp.(logsig)
  return sum((-1/2)*log.(2π*σ.^2) .+ -1/2 * ((xs .- mu).^2)./(σ.^2),dims=1)
end

function skillcontour!(f; colour=nothing, label="sample gaussian")
  n = 100
  x = range(-3,stop=3,length=n)
  y = range(-3,stop=3,length=n)
  z_grid = Iterators.product(x,y) # meshgrid for contour
  z_grid = reshape.(collect.(z_grid),:,1) # add single batch dim
  z = f.(z_grid)
  z = getindex.(z,1)'
  max_z = maximum(z)
  levels = [.99, 0.9, 0.8, 0.7,0.6,0.5, 0.4, 0.3, 0.2] .* max_z
  if colour==nothing
  p1 = contour!(x, y, z, fill=false, levels=levels)
  else
  p1 = contour!(x, y, z, fill=false, c=colour,levels=levels,colorbar=false)
  end
  plot!(p1)
end

function plot_line_equal_skill!()
  plot!(range(-3, 3, length=200), range(-3, 3, length=200), label="Equal Skill")
end
```

```
plot_line_equal_skill! (generic function with 1 method)
```

Question 1.

```
using Revise # lets you change A2funcs without restarting julia!
# includet("A2_src.jl")
using Plots
using Statistics: mean
using Zygote
using Test
```

```
using Logging
# using .A2funcs: log1pexp # log(1 + exp(x)) stable
# using .A2funcs: factorized_gaussian_log_density
# using .A2funcs: skillcontour!
# using .A2funcs: plot_line_equal_skill!
```

1a.

```
function log_prior(zs)
    # μ = mean(zs, dims=1)
    # @show size(μ)
    # logsig = log.(std(zs, dims=1))
    return factorized_gaussian_log_density(0, 0, zs)#TODO
end
```

```
log_prior (generic function with 1 method)
```

1b.

```
function logp_a_beats_b(za,zb)
    return log(1) - log1pexp(zb-za)#TODO
end
```

```
logp_a_beats_b (generic function with 1 method)
```

1c.

```
function all_games_log_likelihood(zs,games)
    w = games[:, 1]
    l = games[:, 2]
    zs_a = zs[w,:] #TODO
    zs_b = zs[l,:] #TODO
    likelihoods = logp_a_beats_b.(zs_a, zs_b)#TODO
    return sum(likelihoods, dims=1)#TODO
end
```

```
all_games_log_likelihood (generic function with 1 method)
```

1d.

```
function joint_log_density(zs,games)
    return all_games_log_likelihood(zs,games) + log_prior(zs)#TODO
end
```

```
joint_log_density (generic function with 1 method)
```

```
@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 prior
  @test size(joint_log_density(test_zs,test_games)) == (1,B)
end
```

```
Test Summary:                              | Pass  Total
Test shapes of batches for likelihoods |    6      6
Test.DefaultTestSet("Test shapes of batches for likelihoods", Any[], 6, fal
se)
```

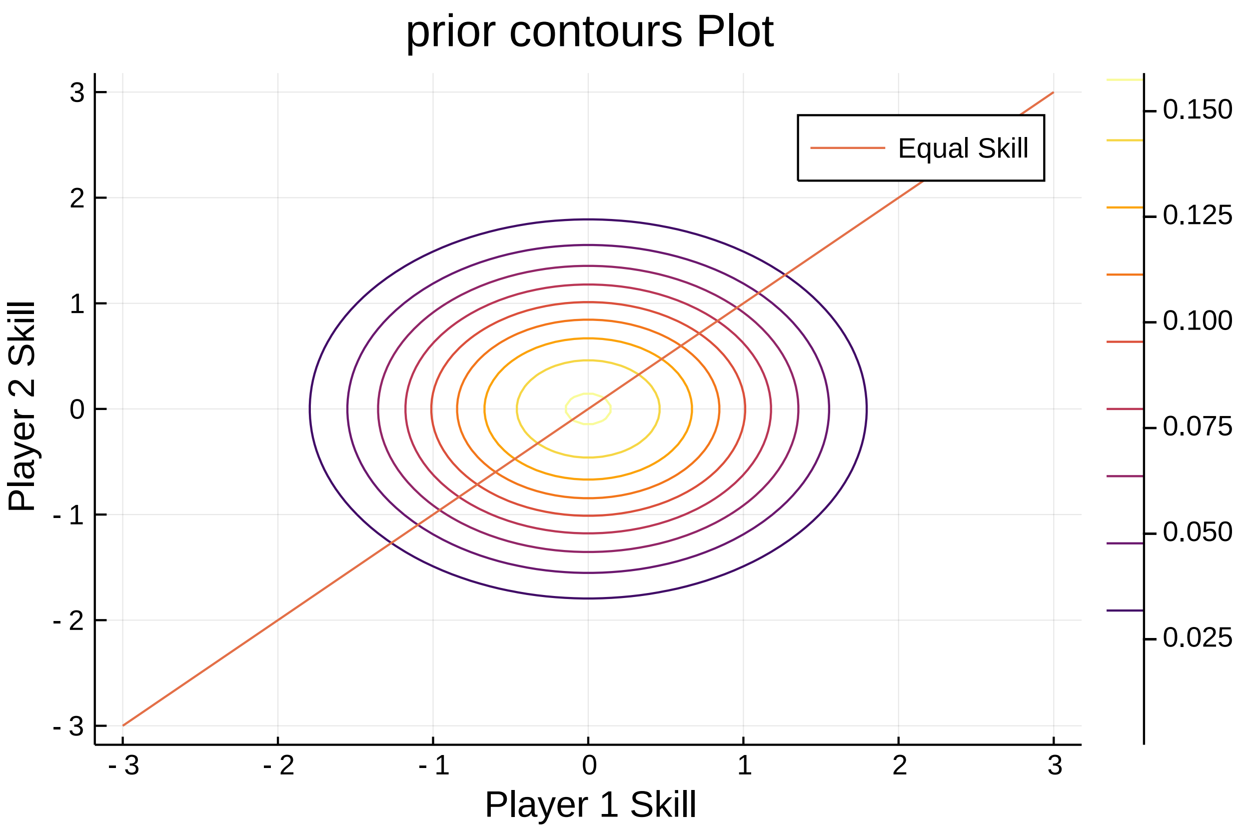
Question 2.

```
# 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)]...)

# Example for how to use contour plotting code
plot(title="Example Gaussian Contour Plot",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )
example_gaussian(zs) = exp(factorized_gaussian_log_density([-1.,2.],[0.,0.5],zs))
skillcontour!(example_gaussian; label="example gaussian")
plot_line_equal_skill!()
savefig(joinpath("plots","example_gaussian.pdf"))
```

2a.

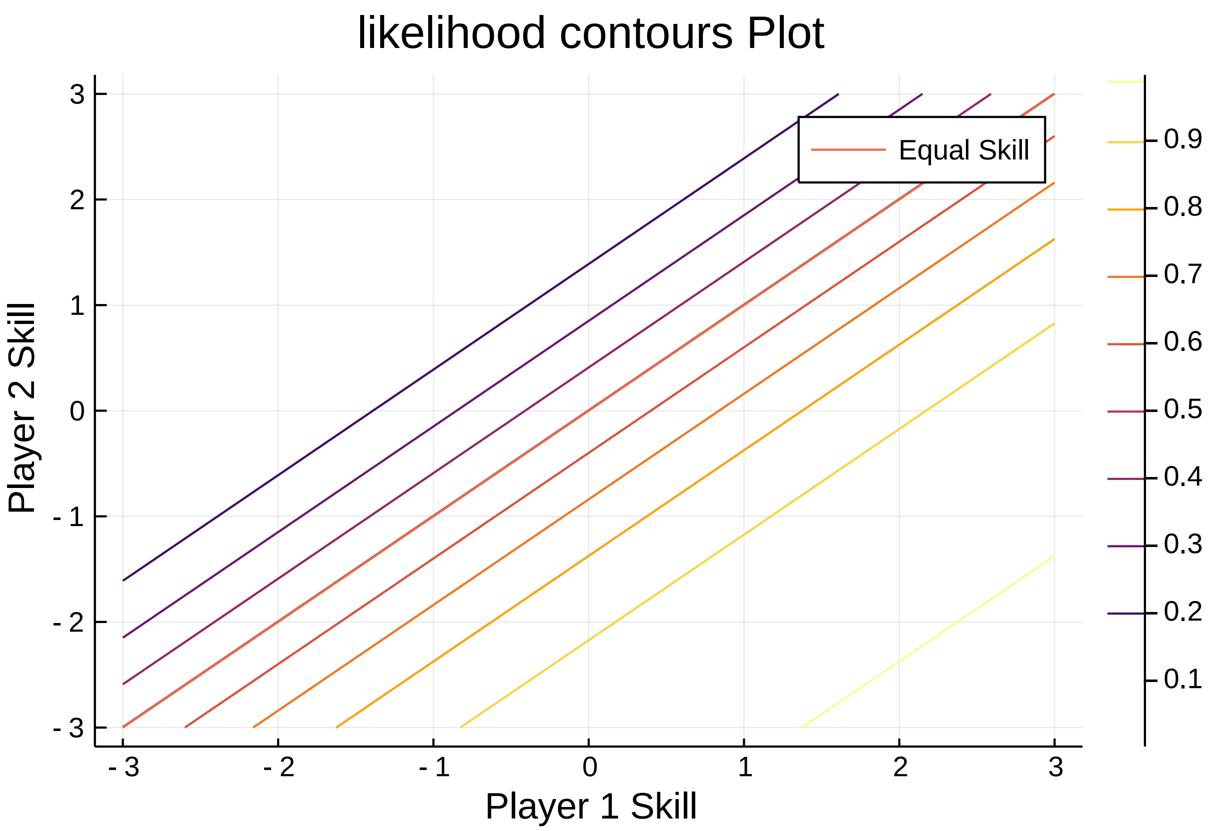
```
# TODO: plot prior contours
plot1 = plot(title="prior contours Plot", xlabel = "Player 1 Skill", ylabel = "Player 2
Skill")
joint_prior(zs) = exp(log_prior(zs))
skillcontour!(joint_prior)
plot_line_equal_skill!()
display(plot1)
```




```
savefig(joinpath("plots","joint_prior.pdf"))
```

2b.

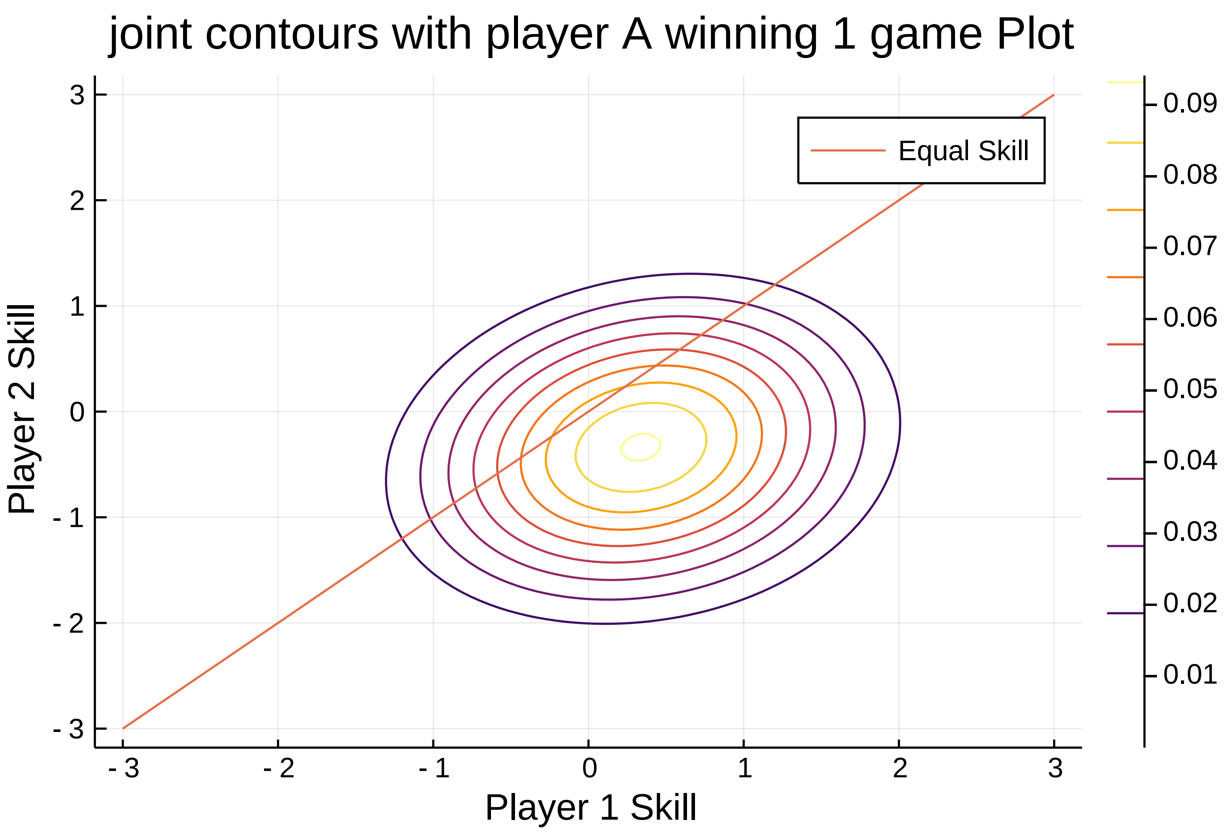
```
# TODO: plot likelihood contours
plot2 = plot(title="likelihood contours Plot", xlabel = "Player 1 Skill", ylabel =
"Player 2 Skill")
likelihood(zs) = exp(logp_a_beats_b(zs[1],zs[2]))
skillcontour!(likelihood)
plot_line_equal_skill!()
display(plot2)
```




```
savefig(joinpath("plots","likelihood.pdf"))
```

2c.

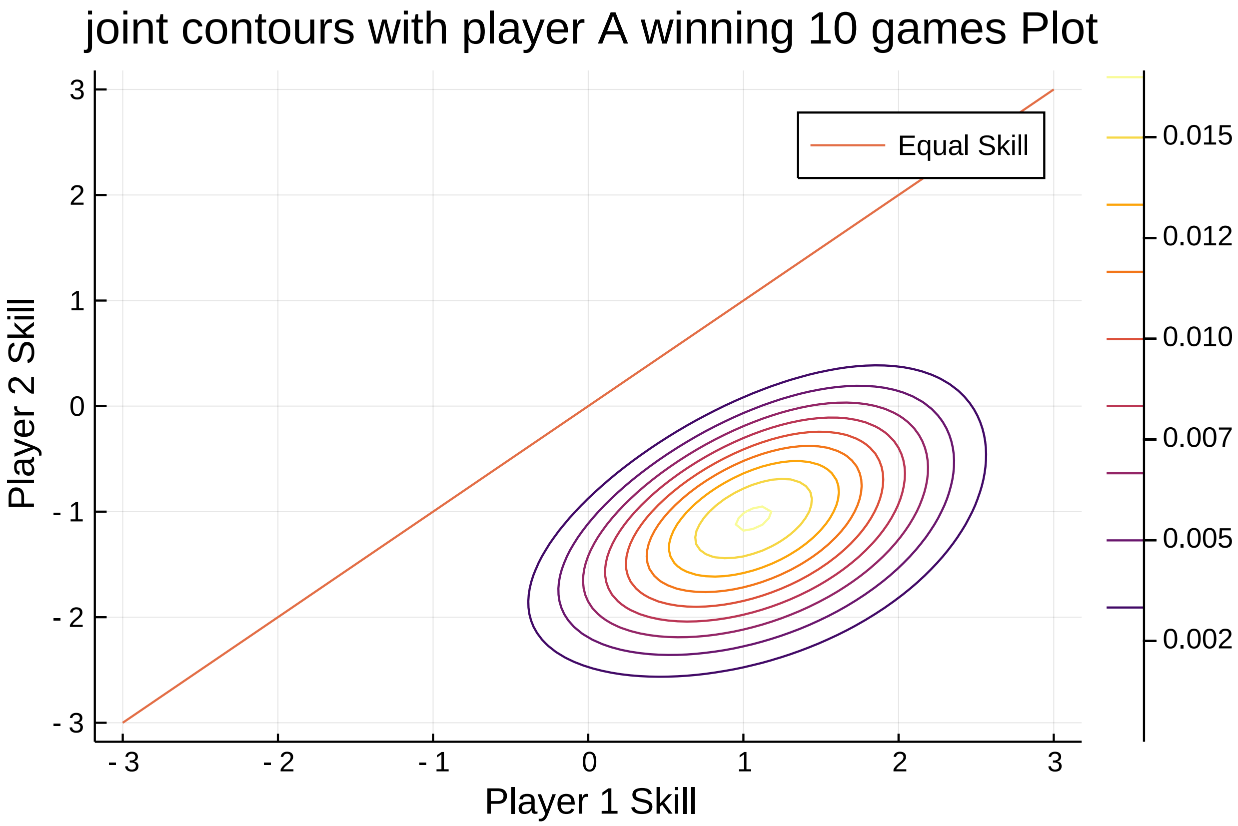
```
# TODO: plot joint contours with player A winning 1 game
plot3 = plot(title="joint contours with player A winning 1 game Plot", xlabel = "Player
1 Skill", ylabel = "Player 2 Skill")
game1  = two_player_toy_games(1, 0)
Awin1(zs) = exp(joint_log_density(zs,game1))
skillcontour!(Awin1)
plot_line_equal_skill!()
display(plot3)
```




```
savefig(joinpath("plots","Awin1.pdf"))
```

2d.

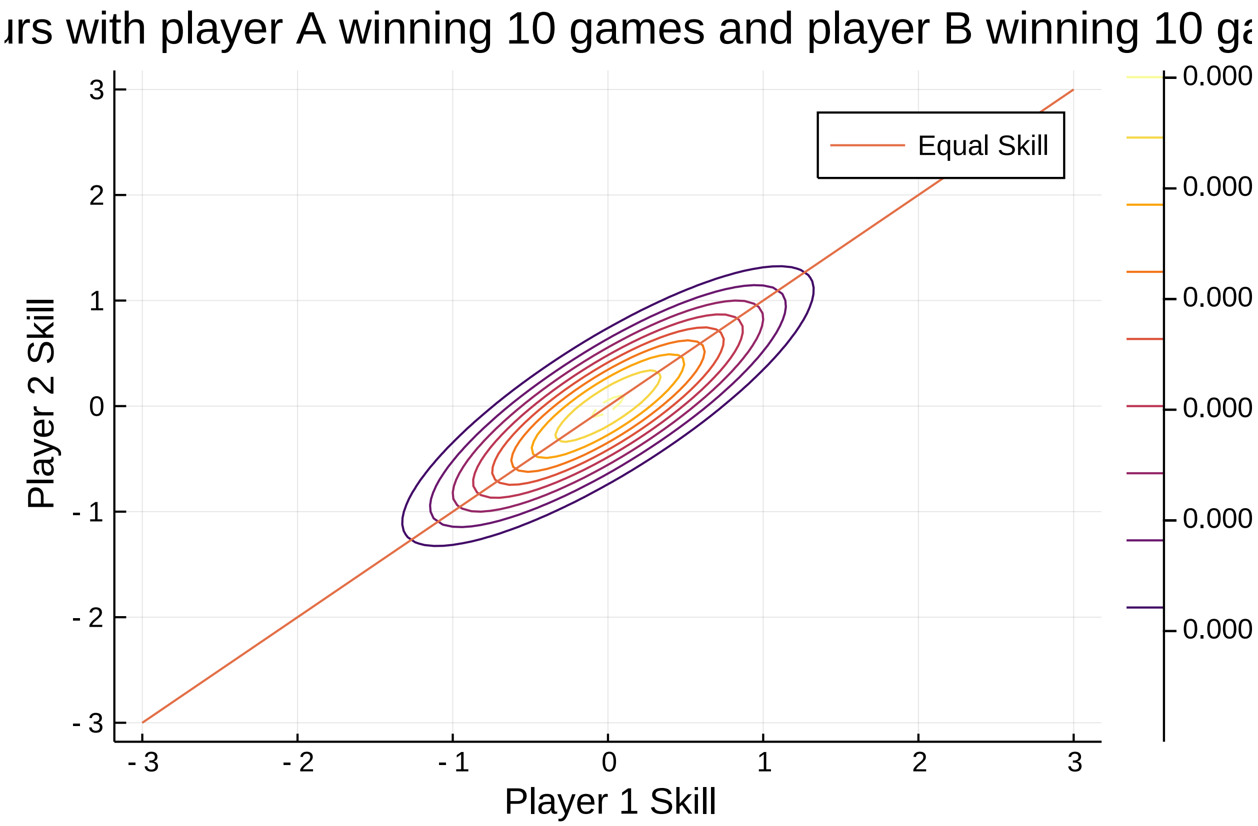
```
# TODO: plot joint contours with player A winning 10 games
plot4 = plot(title="joint contours with player A winning 10 games Plot", xlabel =
"Player 1 Skill", ylabel = "Player 2 Skill")
game10  = two_player_toy_games(10, 0)
Awin10(zs) = exp(joint_log_density(zs,game10))
skillcontour!(Awin10)
plot_line_equal_skill!()
display(plot4)
```




```
savefig(joinpath("plots","Awin10.pdf"))
```

2e.

```
#TODO: plot joint contours with player A winning 10 games and player B winning 10 games
plot5 = plot(title="joint contours with player A winning 10 games and player B winning
10 games Plot", xlabel = "Player 1 Skill", ylabel = "Player 2 Skill")
game1010  = two_player_toy_games(10, 10)
Awin1010(zs) = exp(joint_log_density(zs,game1010))
skillcontour!(Awin1010)
plot_line_equal_skill!()
display(plot5)
```




```
savefig(joinpath("plots","Awin1010.pdf"))
```

Question 3.

3a.

```
function elbo(params,logp,num_samples)
  # sample1 = randn(num_samples)
  # sample2 = randn(num_samples)
  # sample1 = sample1 * exp(params[2][1]) .+ params[1][1]
  # sample2 = sample2 * exp(params[2][2]) .+ params[1][2]
  # samples = transpose(hcat(sample1, sample2))
  sample1 = randn(num_samples)
  sample1 = sample1 .* exp.(params[2][1]) .+ params[1][1]
  for i in 2:1:(length(params[1]))
      # sample1 = randn(num_samples)
      sample2 = randn(num_samples)
      # sample1 = sample1 .* params[2][1] .+ params[1][1]
      sample2 = sample2 .* exp.(params[2][i]) .+ params[1][i]
      sample1 = hcat(sample1, sample2)
  end
  samples = transpose(sample1)
  logp_estimate = logp(samples)#TODO
  logq_estimate = factorized_gaussian_log_density(params[1], params[2], samples)#TODO
  return mean(logp_estimate - logq_estimate, dims=2)[1]#TODO: should return scalar
(hint: average over batch)
end
```

```
elbo (generic function with 1 method)
```

3b.

```
# Conveinence function for taking gradients
function neg_toy_elbo(params; games = two_player_toy_games(1,0), num_samples = 100)
```

```
  # TODO: Write a function that takes parameters for q,
  # evidence as an array of game outcomes,
  # and 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
```

```
neg_toy_elbo (generic function with 1 method)
```

```
# Toy game

num_players_toy = 2
toy_mu = [-2.,3.] # Initial mu, can initialize randomly!
toy_ls = [0.5,0.] # Initual log_sigma, can initialize randomly!
toy_params_init = (toy_mu, toy_ls)
# num_samples = 100
# params11 = toy_params_init
# sample1 = randn(num_samples)
# sample1 = sample1 .* exp.(params11[2][1]) .+ params11[1][1]
# for i in 2:1:(length(params11[1]))
#     # sample1 = randn(num_samples)
#     sample2 = randn(num_samples)
#     # sample1 = sample1 .* params[2][1] .+ params[1][1]
#     sample2 = sample2 .* exp.(params11[2][i]) .+ params11[1][i]
#     global sample1 = hcat(sample1, sample2)
# end
# samples = transpose(sample1)
# print(size(samples))
# print(samples)
```

3c.

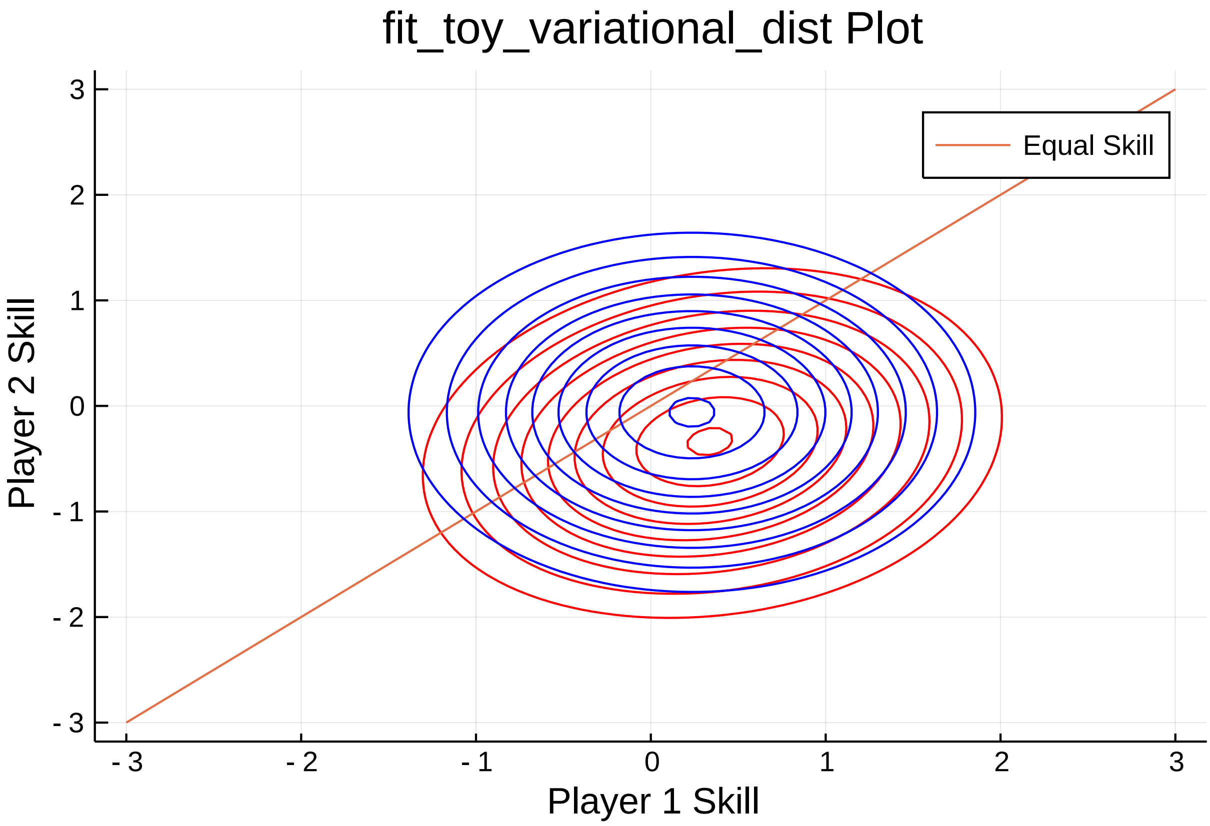
```
function fit_toy_variational_dist(init_params, toy_evidence; num_itrs=200, lr= 1e-2,
num_q_samples = 10)
  params_cur = init_params
  for i in 1:num_itrs
    grad_params = gradient(params_cur -> neg_toy_elbo(params_cur; games = toy_evidence,
num_samples = num_q_samples), params_cur)#TODO: gradients of variational objective with
respect to parameters
    params_cur[1] .=  params_cur[1] - lr*grad_params[1][1]
    params_cur[2] .=  params_cur[2] - lr*grad_params[1][2] #TODO: update paramters with
lr-sized step in descending gradient
    @info "loss: $(neg_toy_elbo(params_cur; games = toy_evidence, num_samples =
num_q_samples))"#TODO: report the current elbbo during training
    # TODO: plot true posterior in red and variational in blue
    # hint: call 'display' on final plot to make it display during training
    # plot();
  end
  plot(title="fit_toy_variational_dist Plot", xlabel = "Player 1 Skill", ylabel =
"Player 2 Skill");
  # zs = rand(2, num_q_samples)
  target_posterior(zs) = exp(joint_log_density(zs, toy_evidence))
  skillcontour!(target_posterior, colour=:red)
  plot_line_equal_skill!()
  # savefig(joinpath("plots","joint_prior.pdf"))
  variational_posterior(zs) = exp(factorized_gaussian_log_density(params_cur[1],
params_cur[2], zs))
  display(skillcontour!(variational_posterior, colour=:blue))
  #TODO: skillcontour!(...,colour=:red) plot likelihood contours for target posterior
  # plot_line_equal_skill()
```

```
  #TODO: display(skillcontour!(..., colour=:blue)) plot likelihood contours for
variational posterior
  return params_cur
end
```

```
fit_toy_variational_dist (generic function with 1 method)
```

3d.

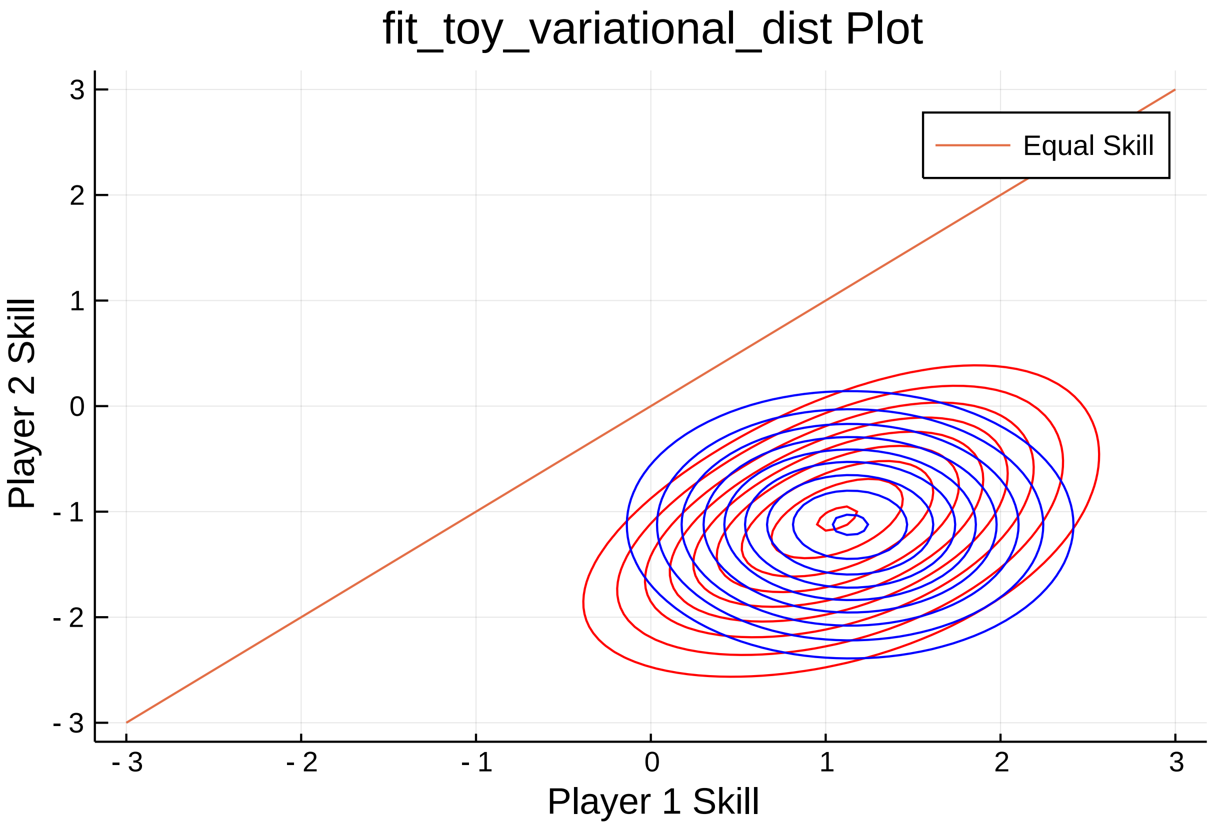
```
#TODO: fit q with SVI observing player A winning 1 game
toy_evidence = two_player_toy_games(1,0)
init_params = toy_params_init
fit_toy_variational_dist(init_params, toy_evidence; num_itrs=200, lr= 1e-2,
num_q_samples = 10)
```




```
savefig(joinpath("plots","toy_evidence(1,0).pdf"))
#TODO: save final posterior plots
```

3e.

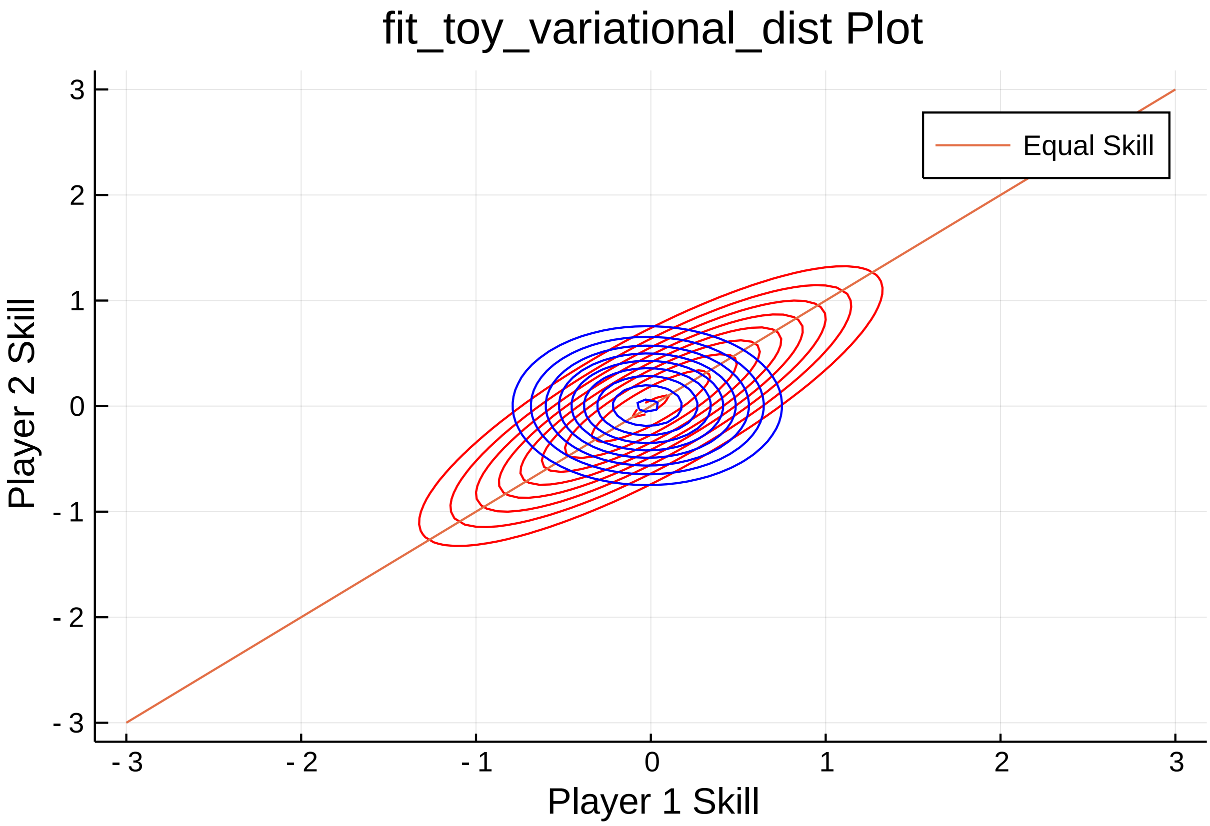
```
#TODO: fit q with SVI observing player A winning 10 games
toy_evidence = two_player_toy_games(10,0)
init_params = toy_params_init
fit_toy_variational_dist(init_params, toy_evidence; num_itrs=200, lr= 1e-2,
num_q_samples = 10)
```




```
savefig(joinpath("plots","toy_evidence(10,0).pdf"))
#TODO: save final posterior plots
```

3f.

```
#TODO: fit q with SVI observing player A winning 10 games and player B winning 10 games
toy_evidence = two_player_toy_games(10,10)
init_params = toy_params_init
fit_toy_variational_dist(init_params, toy_evidence; num_itrs=200, lr= 1e-2,
num_q_samples = 10)
```




```
savefig(joinpath("plots","toy_evidence(10,10).pdf"))
#TODO: save final posterior plots
```

Question 4.

```
# 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")
```

```
Loaded data for 107 players
```

4a.

Yes, games between other players besides i and j will provide information about the skill of players i and j.

4b.

```
function fit_variational_dist(init_params, tennis_games; num_itrs=2000, 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)#TODO: gradients of variational objective wrt
params
    params_cur[1] .=  params_cur[1] - lr*grad_params[1][1]
    params_cur[2] .=  params_cur[2] - lr*grad_params[1][2] #TODO: update parmaeters wite
lr-sized steps in desending gradient direction
    @info "loss: $(neg_toy_elbo(params_cur; games = tennis_games, num_samples =
num_q_samples))"#TODO: report objective value with current parameters
```

```julia
  end
  plot(title="fit_toy_variational_dist Plot", xlabel = "Player 1 Skill", ylabel =
"Player 2 Skill");
  # zs = rand(2, num_q_samples)
  # target_posterior(zs) = exp(joint_log_density(zs, tennis_games))
  # skillcontour!(target_posterior, colour=:red)
  plot_line_equal_skill!()
  # savefig(joinpath("plots","joint_prior.pdf"))
  # variational_posterior_roger(zs) =
exp(factorized_gaussian_log_density(params_cur[1][roger_index],
params_cur[2][roger_index], zs))
  # display(skillcontour!(variational_posterior_roger, colour=:blue))
  # variational_posterior_rafeal(zs) =
exp(factorized_gaussian_log_density(params_cur[1][rafeal_index],
params_cur[2][rafeal_index], zs))
  # display(skillcontour!(variational_posterior_rafeal, colour=:blue))
  return params_cur
end
```

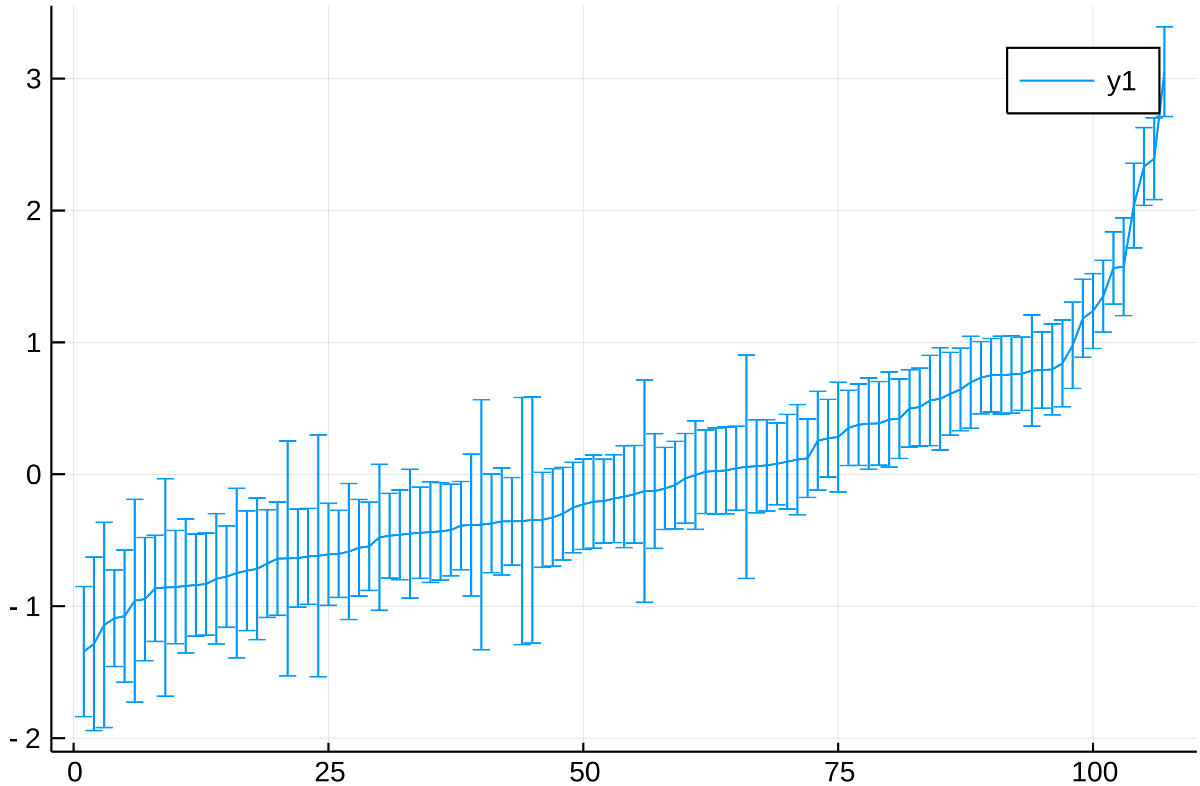
```
fit_variational_dist (generic function with 1 method)
```

```julia
# TODO: Initialize variational family
init_mu = 10 .* rand(num_players)#random initialziation
init_log_sigma = rand(num_players)# random initialziation
init_params = (init_mu, init_log_sigma)

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

4c.

```julia
#TODO: 10 players with highest mean skill under variational model
#hint: use sortperm
perm = sortperm(init_mu)
plot_4c = plot(init_mu[perm], yerror=exp.(init_log_sigma[perm]))
display(plot_4c)
```




```
savefig(joinpath("plots","approximate mean and variance of all players, sorted by
skill.pdf"))
```

4d.

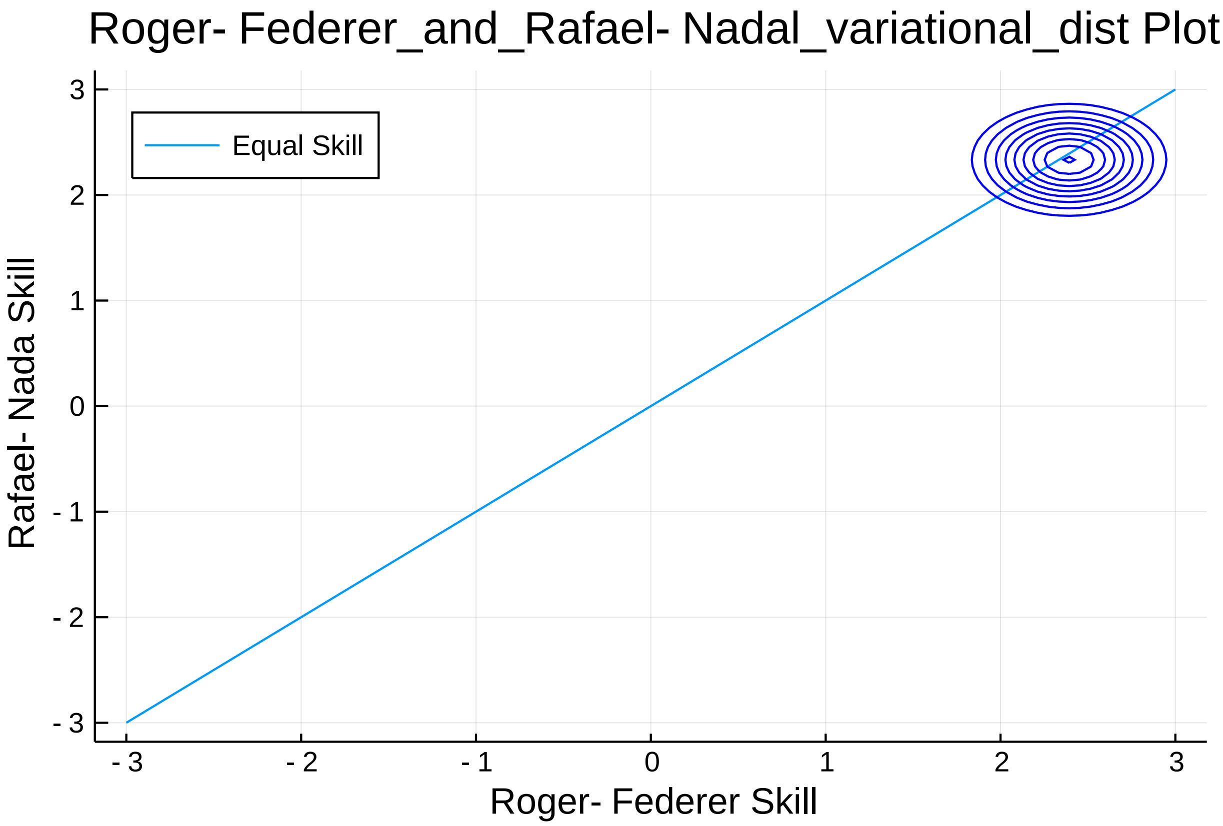
```
top10_player = reverse(player_names[perm][length(perm)-10+1: length(perm)])
print("names of the 10 players with the highest mean skill under the variational model
are ", top10_player)
```

```
names of the 10 players with the highest mean skill under the variational m
odel are Any["Novak-Djokovic", "Roger-Federer", "Rafael-Nadal", "Andy-Murra
y", "Robin-Soderling", "David-Ferrer", "Jo-Wilfried-Tsonga", "Tomas-Berdych
", "Juan-Martin-Del-Potro", "Richard-Gasquet"]
```

4e.

```
#TODO: joint posterior over "Roger-Federer" and ""Rafael-Nadal""
#hint: findall function to find the index of these players in player_names
roger_index = findall(x -> x == "Roger-Federer", vec(player_names))[1]
rafeal_index = findall(x -> x == "Rafael-Nadal", vec(player_names))[1]

plot(title="Roger-Federer_and_Rafael-Nadal_variational_dist Plot", xlabel =
"Roger-Federer Skill", ylabel = "Rafael-Nada Skill", legend=:topleft);
# zs = rand(2, num_q_samples)
# target_posterior(zs) = exp(joint_log_density(zs, toy_evidence))
# skillcontour!(target_posterior, colour=:red)
plot_line_equal_skill!()
# savefig(joinpath("plots","joint_prior.pdf"))
variational_posterior_roger(zs) =
exp(factorized_gaussian_log_density([trained_params[1][roger_index],trained_params[1][rafeal_index]],
[trained_params[2][roger_index],trained_params[2][rafeal_index]], zs))
display(skillcontour!(variational_posterior_roger, colour=:blue))
```




```
savefig(joinpath("plots","joint posterior over Roger-Federer and Rafael-Nadal.pdf"))
```

4f.

Let $X \sim \mathcal{N}(\mu, \Sigma)$, where $X = \begin{bmatrix} z_A \\ z_B \end{bmatrix}$, $\mu = \begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}$, $\Sigma = \begin{bmatrix} \sigma_A & 0 \\ 0 & \sigma_B \end{bmatrix}$, $A = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$, $Y = AX \sim \mathcal{N}(A\mu, A\Sigma A^T)$, where $X = \begin{bmatrix} z_A - z_B \\ z_B \end{bmatrix}$, $A\mu = \begin{bmatrix} \mu_A - \mu_B \\ \mu_B \end{bmatrix}$, $A\Sigma A^T = \begin{bmatrix} \sigma_A + \sigma_B & -\sigma_B \\ -\sigma_B & \sigma_B \end{bmatrix}$, After marginlization, $Y_1 \sim \mathcal{N}(\mu_A - \mu_B, \sigma_A + \sigma_B)$. And exact probability under a factorized Guassian over two players' skills that one has higher skill than the other is $1 - F_{Y_1}(0)$, where $F_{Y_1}(x)$ is CDF of Y_1.

4g.

```
using LinearAlgebra
init_mu1 = [trained_params[1][roger_index],trained_params[1][rafeal_index]]#random
initialziation
init_log_sigma1 =
([exp.(trained_params[2][roger_index]),exp.(trained_params[2][rafeal_index])])# random
initialziation
A = [1 -1; 0 1]
init_params1 = (A * init_mu1, A * Diagonal(init_log_sigma1) * transpose(A))
using Distributions
# variational_posterior_111(zs) = exp(factorized_gaussian_log_density(init_params1[1],
1/sqrt(init_params1[2][1]^2), zs))
# b = Normal(μ=init_params1[1][1], σ=init_params1[2][1])
q = init_params1[1][1]
w = init_params1[2][1]
b = Normal(q,w)
# b = b .* init_params1[2][1] + init_params1[1]
1 - cdf(b, 0)
print("exact probability under approximate posterior that Roger Federer has higher skill
than Rafael Nadal is ", 1 - cdf(b, 0))
```

```
exact probability under approximate posterior that Roger Federer has higher
 skill than Rafael Nadal is 0.5387941392390994
```

```
t = 0
for i in 1:10000
    p = rand(b, 1)
    if p[1] > 0
        global t = t + 1
    end
end
prob = t/10000
print("estimation of simple monte carlo with 10000 samples is ", prob)
```

```
estimation of simple monte carlo with 10000 samples is 0.5307
```

4h.

```
lowest_player = player_names[perm][1]
lowest_index = findall(x -> x == lowest_player, vec(player_names))[1]
init_mu2 = [trained_params[1][roger_index],trained_params[1][lowest_index]]#random
initialziation
init_log_sigma2 =
exp.([trained_params[2][roger_index],trained_params[2][lowest_index]])# random
initialziation
A = [1 -1; 0 1]
init_params2 = (A * init_mu2, A * Diagonal(init_log_sigma2) * transpose(A))
q2 = init_params2[1][1]
w2 = init_params2[2][1]
b2 = Normal(q2,w2)
# b = b .* init_params1[2][1] + init_params1[1]
1- cdf(b2, 0)
print("exact probability under approximate posterior that Roger Federer has higher skill
than the player with the lowest mean skill is ", 1 - cdf(b2, 0))
```

```
exact probability under approximate posterior that Roger Federer has higher
 skill than the player with the lowest mean skill is 0.9999983876405653
```

```
t2 = 0
for i in 1:10000
    p2 = rand(b2, 1)
    if p2[1] > 0
        global t2 = t2 + 1
    end
end
prob = t2/10000
print("estimation of simple monte carlo with 10000 samples is ", prob)
```

```
estimation of simple monte carlo with 10000 samples is 1.0
```

4i.

b, c and e.