Assignment 2

Chien-Che Hung 1004330164, Collaborated with Jeremy Shih 1003113043

March 15, 2020

```
# Load the package
using Revise # lets you change A2funcs without restarting julia!
includet("A2_src.jl")
using Plots
using Statistics: mean
using Zygote
using PyPlot
using Test
using Logging
using Distributions
using .A2funcs: log1pexp # log(1 + exp(x)) stable
using .A2funcs: factorized_gaussian_log_density
using .A2funcs: skillcontour!
using .A2funcs: plot_line_equal_skill!
```

1. Implementing the Model

(a) Log_prior

```
function log_prior(zs)
  prob_play = A2funcs.factorized_gaussian_log_density(0, 0, zs)
  return  prob_play
end
```

(b) Logp_a_beats_b

```
function logp_a_beats_b(za,zb)
  prob_beats = -A2funcs.log1pexp.(-(za-zb))
  return prob_beats
end
```

(c) All_games_log_likelihood

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

(d) Joint_log_density

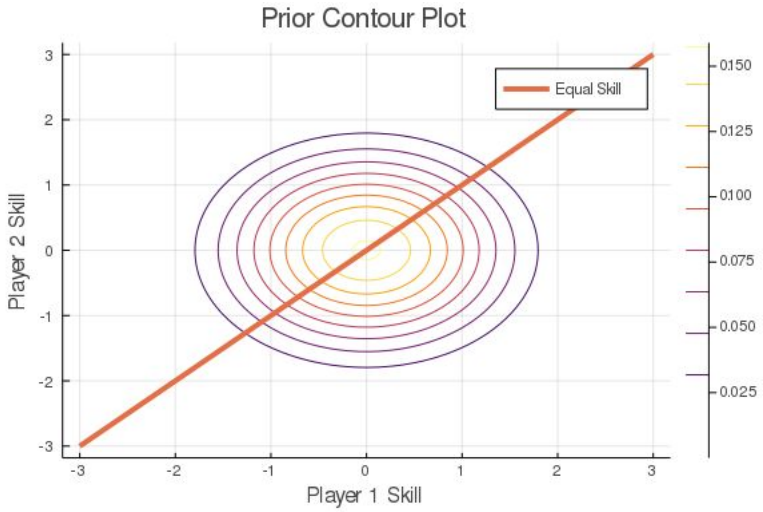
```
function joint_log_density(zs,games)
  prob_all = all_games_log_likelihood(zs, games) + log_prior(zs)
  return prob_all
```

```
end

@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

# 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)]...)
```

2. Examining the posterior for only two players and toy data

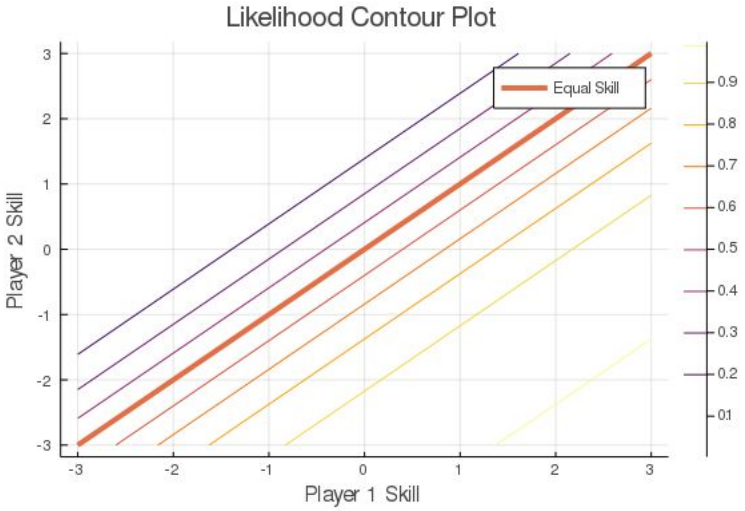
(a) Isocontours of the joint prior over their skills

```
# TODO: plot prior contours
PyPlot.clf()
plot(title="Prior Contour Plot",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )
prior_function(zs) = exp(log_prior(zs))
skillcontour!(prior_function)
plot_line_equal_skill!()
```



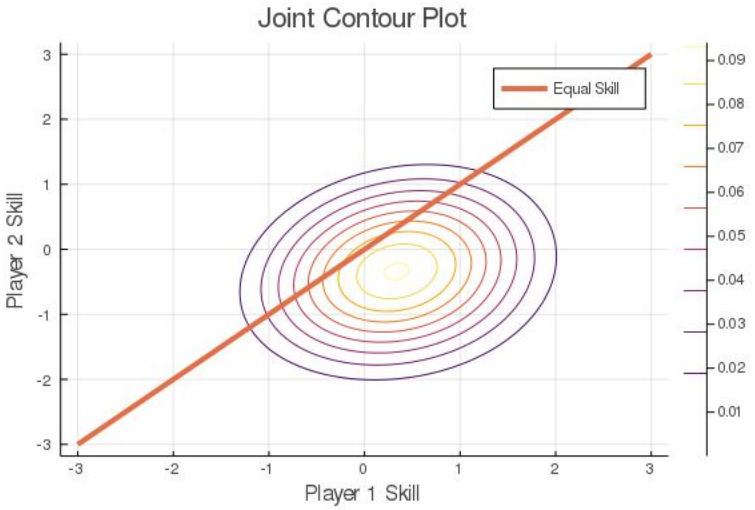
(b) Isocontours of the likelihood function

```
# TODO: plot likelihood
contours
PyPlot.clf()
plot(title="Likelihood Contour
Plot",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )
h(z) =
exp(logp_a_beats_b(z[1],
z[2]))
skillcontour!(h)
plot_line_equal_skill!()
```



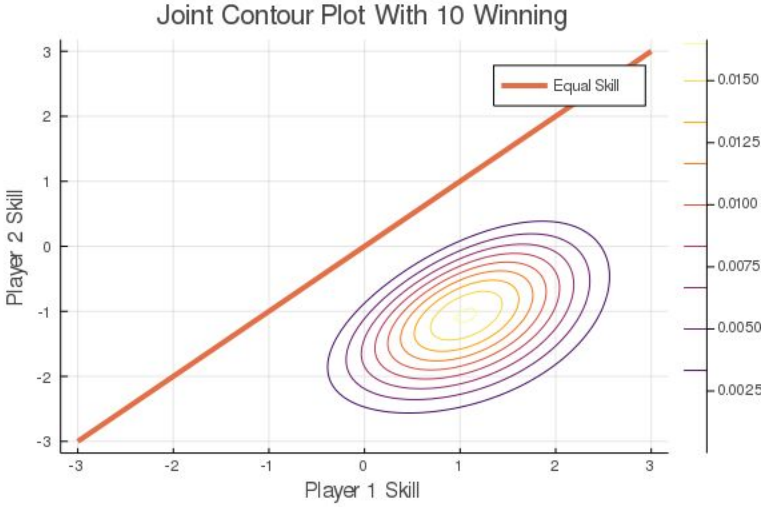
(c) Isocontours of the joint posterior over z_a and z_b given that player A beat player B in one match

```
# TODO: plot joint contours
with player A winning 1 game
PyPlot.clf()
plot(title="Joint Contour
Plot",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )
joint_game_1(zs) =
exp(joint_log_density(zs,
two_player_toy_games(1,0)))
skillcontour!(joint_game_1)
plot_line_equal_skill!()
```



(d) Isocontours 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

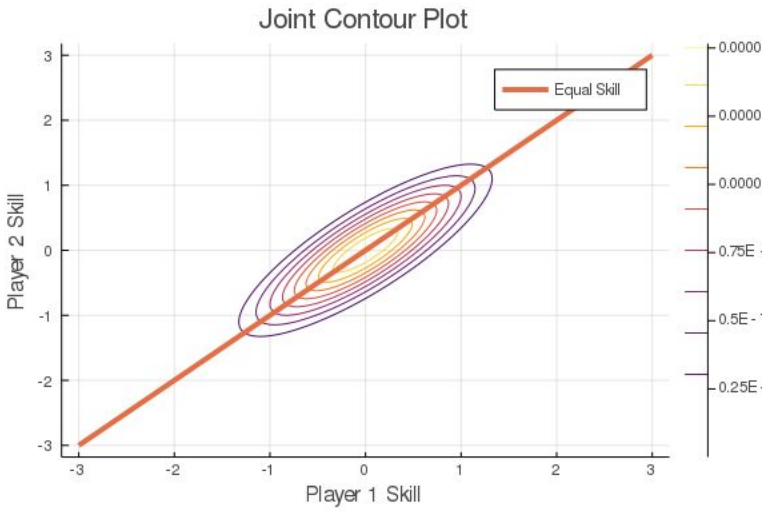
```
# TODO: plot joint contours
with player A winning 10 games
PyPlot.clf()
```



```
plot(title="Joint Contour Plot With 10 Winning",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )
games = two_player_toy_games(10,0)
joint_game_2(zs) = exp(joint_log_density(zs, games))
skillcontour!(joint_game_2)
plot_line_equal_skill!()
```

(e) Isocontours of the joint posterior over z_a and z_b given that 20 match were played and each player beat the other 10 times

```
#TODO: plot joint contours with player
A winning 10 games and player B
winning 10 games
PyPlot.clf()
plot(title="Joint Contour Plot",
    xlabel = "Player 1 Skill",
    ylabel = "Player 2 Skill"
   )
games = two_player_toy_games(10,10)
joint_game_2(zs) =
exp(joint_log_density(zs, games))
skillcontour!(joint_game_2)
plot_line_equal_skill!()
```




```
savefig(joinpath("plots","joint_contours_10_10.pdf"))
```

3. Stochastic Variational Inference on Two Players and Toy Data

(a) ELBO function

```
function elbo(params,logp,num_samples)
  samples = exp.(params[2]) .*  randn(length(params[1]), num_samples) .+
params[1]
  logp_estimate = logp(samples)
  logq_estimate = A2funcs.factorized_gaussian_log_density(params[1],
params[2], samples)
  return -mean(logq_estimate-logp_estimate)
end
```

(b) Neg_toy_elbo_function

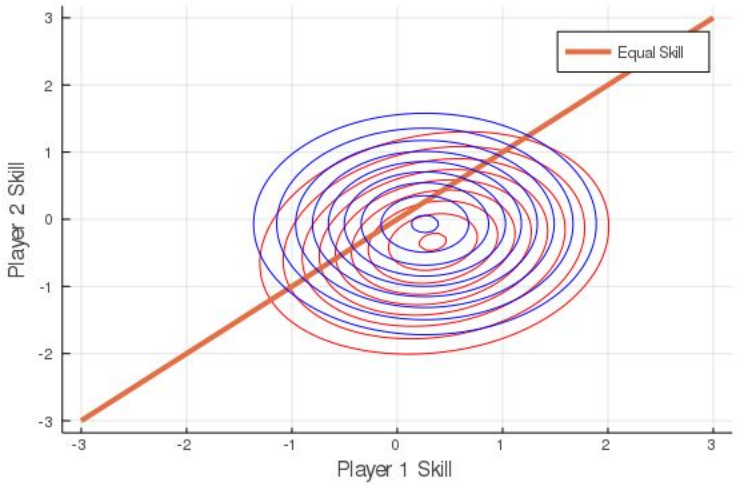
```
# Conveinence function for taking gradients
function neg_toy_elbo(params; games = two_player_toy_games(1,0), num_samples =
100)
  logp(zs) = joint_log_density(zs,games)
  return -elbo(params,logp, num_samples)
End
```

(c) Fit_variational_dist

```
# Toy game
num_players_toy = 2
toy_mu = [-2.,3.] # Initial mu, can initialize randomly!
toy_ls = [0.5,0.]
toy_params_init = (toy_mu, toy_ls) # Initual log_sigma, can initialize
randomly!
function fit_toy_variational_dist(init_params, toy_evidence; num_itrs=200, lr=
1e-2, num_q_samples = 10)
  params_cur = collect(init_params)
  player_num = length(init_params)[1]
  for i in 1:num_itrs
    grad_params = gradient((param)-> neg_toy_elbo(param, games=toy_evidence,
num_samples=num_q_samples), params_cur)[1]
    params_cur[1] = params_cur[1] - lr * grad_params[1]
    params_cur[2] = params_cur[2] - lr * grad_params[2]
    elbo = -neg_toy_elbo(params_cur, games = toy_evidence, num_samples =
num_q_samples)
    @info "elbo : $(elbo)"
    plot(xlabel = "Player 1 Skill", ylabel = "Player 2 Skill");
    log_estp(zs) = exp(joint_log_density(zs, toy_evidence))
    skillcontour!(log_estp, colour =:red)
    A2funcs.plot_line_equal_skill!()
    log_estq(zs) = exp(A2funcs.factorized_gaussian_log_density(params_cur[1],
params_cur[2], zs))
    display(skillcontour!(log_estq, colour=:blue))
  end
  return params_cur
end
```

(d) Fit q with SVI observing player A winning 1 game

```
#TODO: fit q with SVI observing player
A winning 1 game
#TODO: save final posterior plots
evidence = two_player_toy_games(1,0)
```

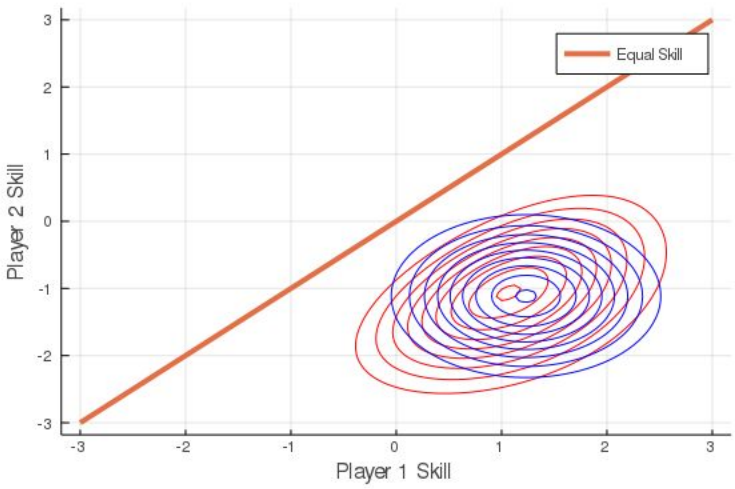


```
fit_toy_variational_dist(toy_params_init, evidence)
savefig(joinpath("plots","SVI_A_1game.pdf"))
```

Since we want to minimize neg_ELBO in order to have the minimized D_KL divergence

- Initial neg_ELBO: 10.079396782080003
- Final neg_ELBO: 0.9073487501064925

(e) Fit q with SVI observing player A winning 10 games



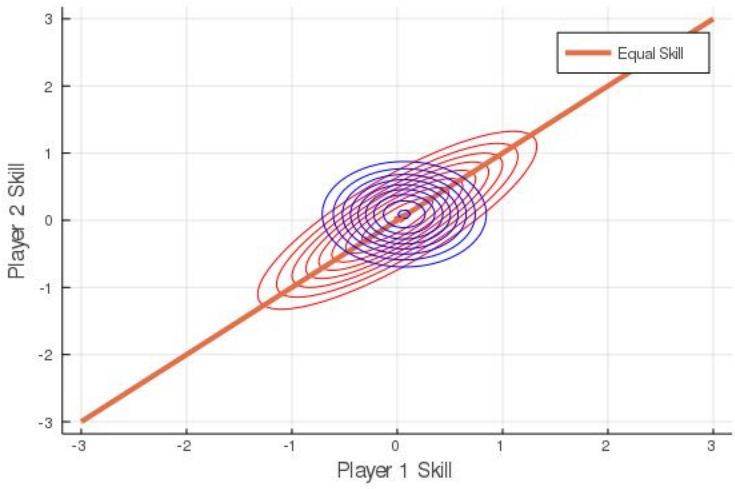

```
#TODO: fit q with SVI
observing player A winning
10 games
#TODO: save final posterior
plots
evidence =
two_player_toy_games(10,0)
```

```
fit_toy_variational_dist(toy_params_init, evidence)
savefig(joinpath("plots","SVI_A_10game.pdf"))
```

Since we want to minimize neg_ELBO in order to have the minimized D_KL divergence

- Initial neg_ELBO: 58.29685117007144
- Final neg_ELBO: 2.9588250420331152

(f) Fit q with SVI observing player A winning 10 games and player B winning 10 games




```
#TODO: fit q with SVI
observing player A winning
10 games and player B
winning 10 games
#TODO: save final posterior
plots
evidence =
two_player_toy_games(10,10)
```

```
fit_toy_variational_dist(toy_params_init, evidence)
savefig(joinpath("plots","SVI_A_10_B_10game.pdf"))
```

Since we want to minimize neg_ELBO in order to have the minimized D_KL divergence

- Initial neg_ELBO: 52.870125265776245
- Final neg_ELBO: 15.656824875306679

4. Approximate Inference Conditioned on Real Data

(a) Yes

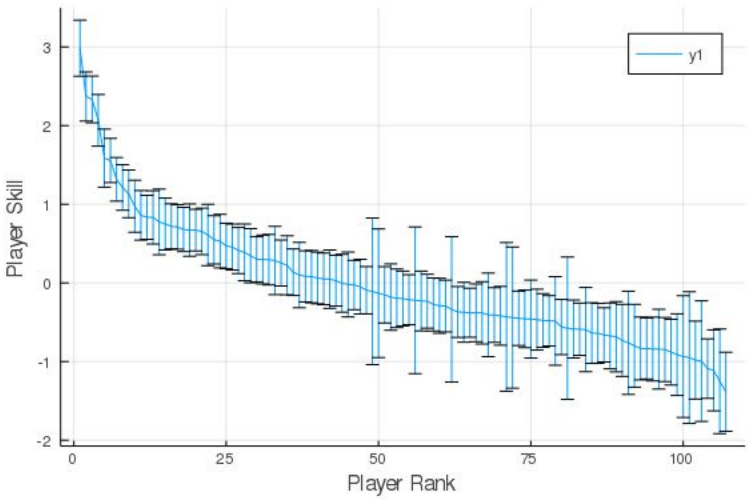
```
# Load the Data
using MAT
vars = matread("tennis_data.mat")
player_names = vars["W"]
tennis_games = Int.(vars["G"])
num_players = length(player_names)
print("Loaded data for $num_players players")
```

(b) Fit_variation_dist

```
function fit_variational_dist(init_params, tennis_games; num_itrs=200, lr=
1e-2, num_q_samples = 10)
  params_cur = collect(init_params)
  elbo = 0
  for i in 1:num_itrs
    grad_params = gradient((param)-> neg_toy_elbo(param, games=tennis_games,
num_samples=num_q_samples), params_cur)[1]#TODO: gradients of variational
objective with respect to parameters
    params_cur[1] = params_cur[1] - lr * grad_params[1]
    params_cur[2] = params_cur[2] - lr * grad_params[2]
    elbo = neg_toy_elbo(params_cur, games = tennis_games, num_samples =
num_q_samples)
    @info "elbo : $(elbo)"#TODO: report the current elbo during training
    plot(xlabel = "Player 1 Skill", ylabel = "Player 2 Skill");
  end
  @info "Final elbo: $(elbo)"
  return tuple(params_cur)
end
```

(c) Plot the approximate mean and variance of all players, sorted by skill

```
# TODO: Initialize variational
family
```



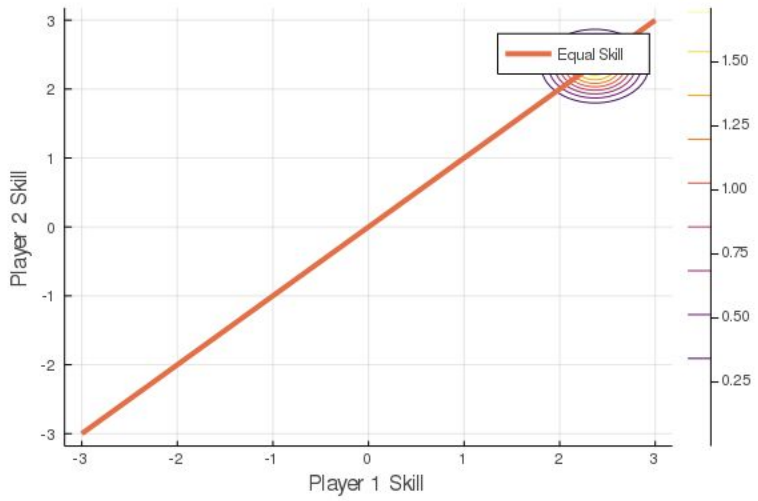
```
init_mu = randn(num_players)#random initialization
init_log_sigma = randn(num_players)/100 # random initialization
init_params = (init_mu, init_log_sigma)


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

(d) List the names of the 10 players with the highest mean skill under the variational model:

```
#TODO: 10 players with highest mean skill under variational model
#hint: use sortperm
mean_sort = reverse(sortperm(trained_params[1][1]))
plot(trained_params[1][1][mean_sort], yerror =
exp.(trained_params[1][2][mean_sort]), xlabel = "Player Rank", ylabel= "Player
Skill")
savefig(joinpath("plots","sort_players.pdf"))
player_names[mean_sort[1:10]]
```

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) Plot the joint posterior over the skills of Roger Federer and Rafel Nadal




```
#TODO: joint posterior over
"Roger-Federer" and
""Rafael-Nadal""
#hint: findall function to
find the index of these
players in player_names
```

```
roger = findall(x->x=="Roger-Federer", player_names)
rafael = findall(x->x=="Rafael-Nadal", player_names)
mean_trained =
[trained_params[1][1][roger][1],trained_params[1][1][rafael][1]]
var_trained = [trained_params[1][2][roger][1],trained_params[1][2][rafael][1]]

plot(xlabel = "Player 1 Skill", ylabel = "Player 2 Skill")
log_estp(zs) = exp(A2funcs.factorized_gaussian_log_density(mean_trained,
var_trained, zs))
A2funcs.skillcontour!(log_estp)
A2funcs.plot_line_equal_skill!()
savefig(joinpath("plots","RF_RN_joint.pdf"))
```

(f) Derive the exact probability under a factorized Gaussian over two players' skills that one has higher skill than the other, as a function of the two means and variances over their skills

$$Ax = b$$

$$A\begin{pmatrix} z_a \\ z_b \end{pmatrix} = \begin{pmatrix} y_a \\ y_b \end{pmatrix} = \begin{pmatrix} z_a - z_b \\ z_b \end{pmatrix}$$

From here, we can know that $A = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$

From Linear Transfomation $Aq(z_a, z_b) \sim N(AZ, A\Sigma A^T)$

$$\mu = \begin{pmatrix} \mu_a - \mu_b \\ \mu_b \end{pmatrix}, A\Sigma A^T = \begin{pmatrix} \Sigma_{11} - \Sigma_{21} - \Sigma_{12} + \Sigma_{22} & \Sigma_{12} - \Sigma_{22} \\ \Sigma_{21} - \Sigma_{22} & \Sigma_{22} \end{pmatrix}$$

$$\Rightarrow P(Y_a > 0) = P(\Sigma_{11} - \Sigma_{21} - \Sigma_{12} + \Sigma_{22} * x + (\mu_a - \mu_b) > 0)\blacksquare$$

(g) Exact Probability and Monte Carlo

(i) Exact Probablity: 0.5239767796227848

(ii) Monte Carlo Estimation: 0.5343

```
#TODO: 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.
tran_A = [1 -1; 0 1]
roger_mean = trained_params[1][1][roger]
roger_sig = exp.(trained_params[1][2][roger])
rafael_mean = trained_params[1][1][rafael]
```

```
rafael_sig = exp.(trained_params[1][2][rafael])
μa_μb = [roger_mean[1], rafael_mean[1]]
tran_mean = tran_A*μa_μb
σa_σb = [roger_sig[1] 0;0 rafael_sig[1]]
tran_sig = tran_A*σa_σb*tran_A'
# Exact Probability
1-cdf(Normal(tran_mean[1], tran_sig[1][1]),0)
samples_roger = roger_sig[1] .*  randn(1, 10000) .+ roger_mean[1]
samples_rafael = rafael_sig[1] .*  randn(1, 10000) .+ rafael_mean[1]
samples_roger[100]
function monte_carlo(sample1, sample2)
  count = 0
  for i in 1:10000
    if sample1[i] > sample2[i]
      count = count+1
    end
  end
  return count
End
# Monte Carlo
monte_carlo(samples_roger, samples_rafael)/10000
```

(h) Roger vs Lowest Mean Skill

(i) Exact Probability: 0.9999071166488139

(ii) Monte Carlo Estimation: 1.00

```
# Roger vs Lowest Mean Skill
tran_A = [1 -1; 0 1]
roger_mean = trained_params[1][1][roger]
roger_sig = exp.(trained_params[1][2][roger])
low_mean = trained_params[1][1][75]
low_sig = exp.(trained_params[1][2][75])
μa_μb = [roger_mean[1], rafael_mean[1]]
tran_mean = tran_A*μa_μb
σa_σb = [roger_sig[1] 0;0 rafael_sig[1]]
tran_sig = tran_A*σa_σb*tran_A'
# Exact Probability
1-cdf(Normal(tran_mean[1], tran_sig[1][1]),0)
# Monte Carlo
samples_roger = roger_sig[1] .*  randn(1, 10000) .+ roger_mean[1]
samples_low = low_sig[1] .*  randn(1, 10000) .+ low_mean[1]
monte_carlo(samples_roger, samples_low)/10000
```

(i) c, e will be different since the skill scores will be higher