Assignment 2

Raj Patel

March 22, 2020

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

```
include("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!
```

1 Implementing the model

(a) Implement a function log prior that computes the log of the prior over all player′s skills. Specifically, given a K × N array where each row is a setting of the skills for all N players, it returns a K × 1 array, where each row contains a scalar giving the log-prior for that set of skills.

```
function log_prior(zs)

  # using the factorized_gaussian_log_density from starter code
  l_prior = factorized_gaussian_log_density(0.,0.,zs)
  return l_prior

end
```

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

(b) Implement a function logp a beats b that, given a pair of skills za and zb evaluates the log-likelihood that player with skill za beat player with skill zb 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)

  # based on the trueskill model
  return -log1pexp.(-(za .- zb))

end
```

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

(c) 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 × N array where each row is a setting of the skills for all N players, and an M × 2 array of game outcomes, it returns a K × 1 array, where each row contains a scalar giving the log-likelihood of all games for that set of skills.

```
function all_games_log_likelihood(zs,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
```

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

(d) Implement a function joint log density which combines the log-prior and log-likelihood of the observations to give p(z1 , z2 , . . . , zN , all game outcomes)

```
function joint_log_density(zs,games)

  # joint log density using log-prior and log-likelihood of all observations
  return log_prior(zs) .+ all_games_log_likelihood(zs,games)

end
```

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

Now, we will perform tests to ensure that we are performing operations on right dimensions

```
@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)
```

2 Examining the posterior for only 2 players and toy data

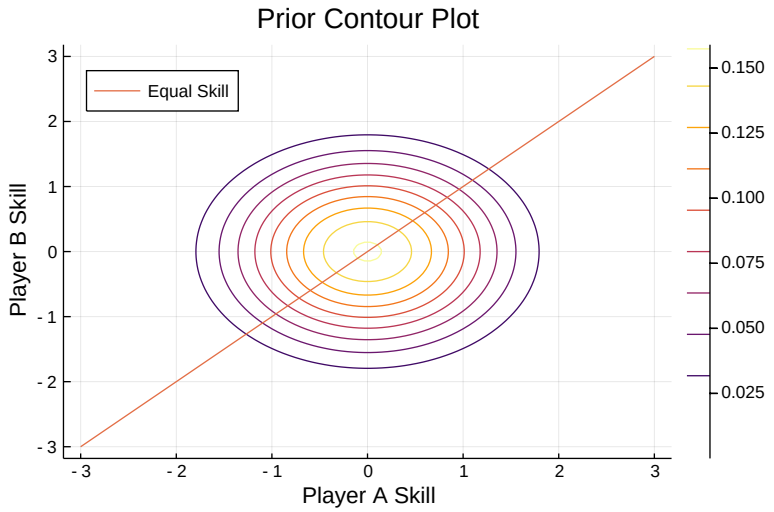
We first provide a function two player toy games which produces toy data for two players. I.e. two player toy games(5,3) produces a dataset where player A wins 5 games and player B wins 3 games.

```
# 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)]...)
```

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

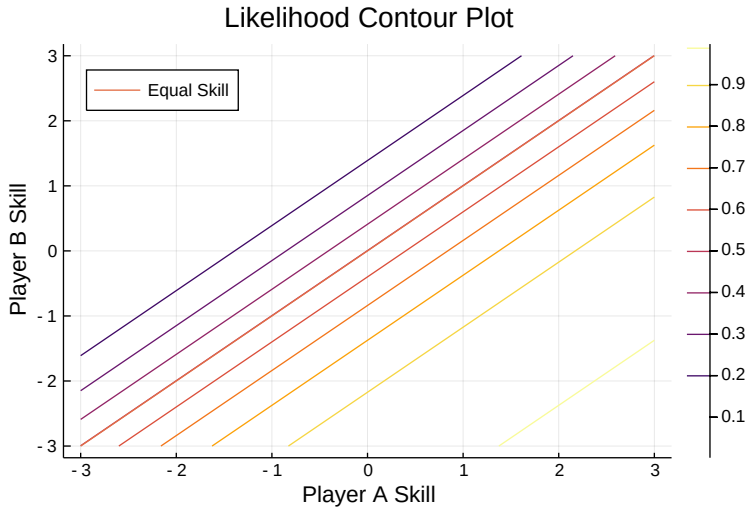
(a) For two players A and B, plot the isocontours of the joint prior over their skills. Also plot the line of equal skill, zA = zB.

```
using .A2funcs: skillcontour!

# plot prior contours
zs = randn(2,15)
plot(title="Prior Contour Plot",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
example_gaussian(zs) = exp(factorized_gaussian_log_density(0,0,zs))
skillcontour!(example_gaussian)
plot_line_equal_skill!()
savefig(joinpath("plots","prior_contour.pdf"))
```



(b) Plot the isocontours of the likelihood function. Also plot the line of equal skill, zA = zB

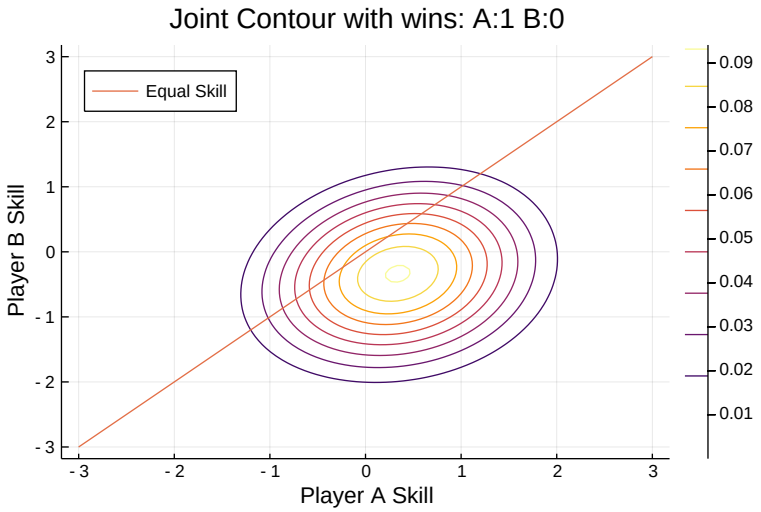
```
# plot likelihood contours
plot(reuse = false)
plot(title="Likelihood Contour Plot",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
example_gaussian(zs) = exp.(logp_a_beats_b(zs[1,:],zs[2,:]))
skillcontour!(example_gaussian)
plot_line_equal_skill!()
savefig(joinpath("plots","likelihood_contour.pdf"))
```



(c) Plot isocountours of the joint posterior over zA and zB 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(zA,zB,A beat B) Also plot the line of equal skill, zA =zB.

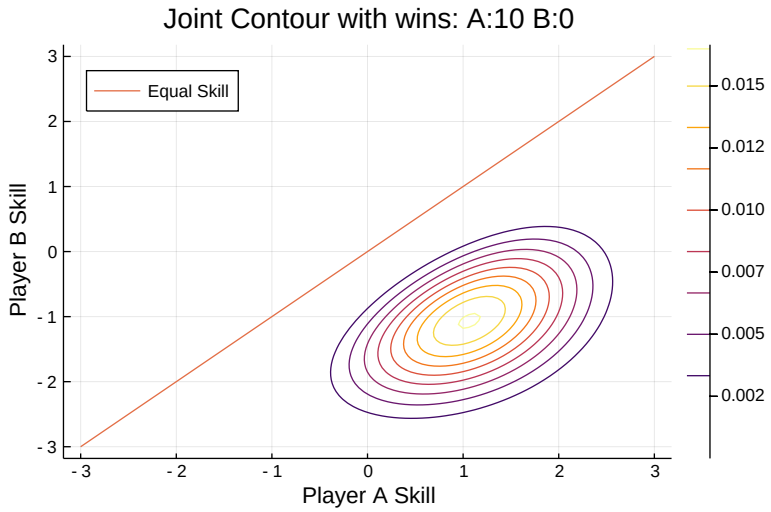
```
# plot isocountours of joint posterior over zA and zB given A beats B in 1 match
plot(reuse = false)
```

```
games_2c = two_player_toy_games(1, 0)
plot(title="Joint Contour with wins: A:1 B:0",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
example_gaussian(zs) = exp.(joint_log_density(zs,games_2c))
skillcontour!(example_gaussian)
plot_line_equal_skill!()
savefig(joinpath("plots","joint_contour_2c.pdf"))
```



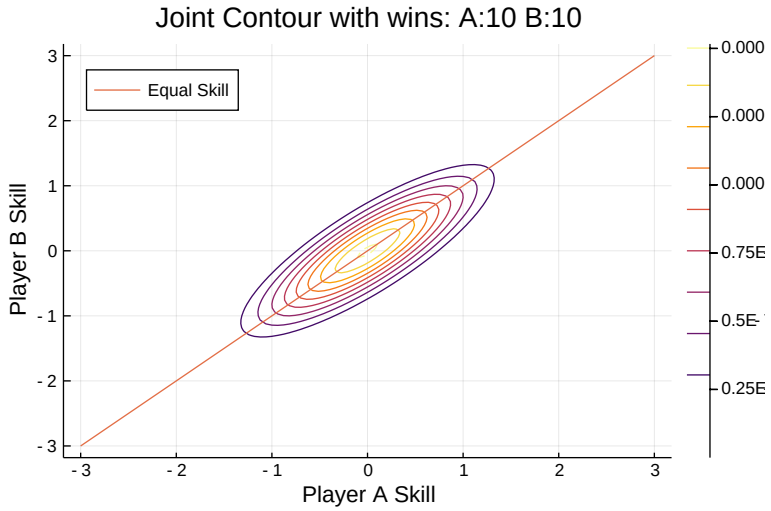
(d) Plot isocountours of the joint posterior over zA and zB given that 10 matches were played, and player A beat player B all 10 times. Also plot the line of equal skill, zA = zB.

```
# plot isocountours of joint posterior over zA and zB given A beats B in 10 matches
plot(reuse = false)
games_2d = two_player_toy_games(10, 0)
plot(title="Joint Contour with wins: A:10 B:0",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
example_gaussian(zs) = exp.(joint_log_density(zs,games_2d))
skillcontour!(example_gaussian)
plot_line_equal_skill!()
savefig(joinpath("plots","joint_contour_2d.pdf"))
```



(e) Plot isocountours of the joint posterior over zA and zB given that 20 matches were played, and each player beat the other 10 times. Also plot the line of equal skill, zA = zB.

```
# plot isocountours of joint posterior over zA and zB given both A and B beat each
other 10 times
plot(reuse = false)
games_2e = two_player_toy_games(10, 10)
plot(title="Joint Contour with wins: A:10 B:10",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
example_gaussian(zs) = exp.(joint_log_density(zs,games_2e))
skillcontour!(example_gaussian)
plot_line_equal_skill!()
savefig(joinpath("plots","joint_contour_2e.pdf"))
```



3 Stochastic Variational Inference on Two Players and Toy Data

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) Implement a function elbo which computes an unbiased estimate of the evidence lower bound. The ELBO is equal to the KL divergence between the true posterior p(z given data), and an approximate posterior, q(z given data), plus an unknown constant. Here we use a fully-factorized Gaussian distribution for q(z given data). This estimator takes the following arguments: (1) params, the parameters of the approximate posterior q(z given data). (2) 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). (3) num samples, the number of samples to take.

```
function elbo(params,logp,num_samples)

  # Using the reparametrization trick
  samples = (params[1] .+ ((exp.(params[2])) .* randn(length(params[1]), num_samples)))

  logp_estimate = logp(samples)

  # Using factorized Gaussian distribution for approximate posterior
  logq_estimate = factorized_gaussian_log_density(params[1],params[2],samples)

  return mean(logp_estimate .- logq_estimate)

end
```

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

(b) 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.

```
# 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
```

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

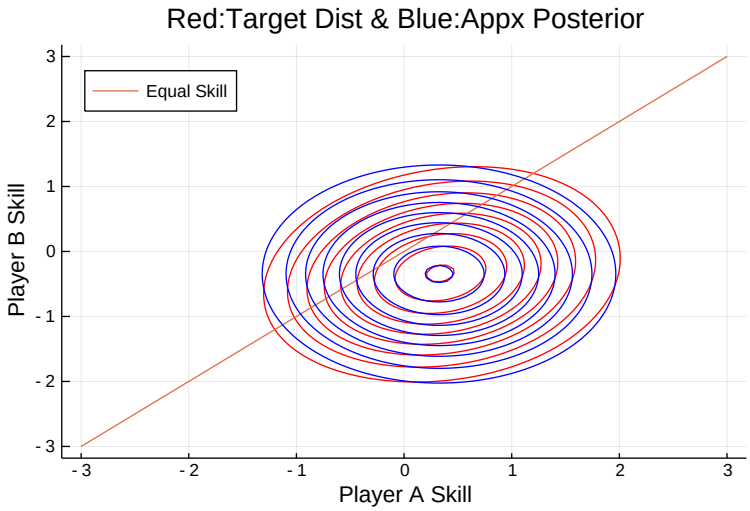
(c) Write an optimization function called fit toy variational dist which takes initial variational parameters, and the evidence. Inside it will perform a number of iterations of gradient descent where for each iteration : (1) Compute the gradient of the loss with respect to the parameters using automatic differentiation. (2) Update the parameters by taking an lr-scaled step in the direction of the descending gradient. (3) Report the loss with the new parameters (using @info or print statements) (4) On the same set of axes plot the target distribution in red and the variational approximation in blue. Return the parameters resulting from training.

```
function fit_toy_variational_dist(init_params, toy_evidence; num_itrs=200, lr= 1e-2,
num_q_samples = 100)

  # Initiating parameters
  params_cur = init_params

  for i in 1:num_itrs
    # gradients of variational objective with respect to parameters
    grad_params = gradient(params -> neg_toy_elbo(params, games = toy_evidence,
num_samples = num_q_samples),params_cur)[1]

    # update paramters with lr-sized step in descending gradient
    params_cur = params_cur .- (grad_params  .* lr)

    # report the current negative elbo during training
    @info "loss: $(neg_toy_elbo(params_cur, games = toy_evidence, num_samples =
num_q_samples))"

  end
  zs_new = (init_params[1] .+ ((exp.(init_params[2])) .* randn(length(init_params[1]),
num_q_samples)))

  plot(reuse = false)
   plot(title="Red:Target Dist & Blue:Appx Posterior",
      xlabel = "Player A Skill",
      ylabel = "Player B Skill"
     )

  # Plotting the target distribution
  example_gaussian(zs_new) = exp.(joint_log_density(zs_new,toy_evidence))
  display(skillcontour!(example_gaussian,colour=:red))
  display(plot_line_equal_skill!())

  # Plotting the variational approximation
  example_gaussian_2(zs_new) =
exp.(factorized_gaussian_log_density(params_cur[1],params_cur[2],zs_new))
  display(skillcontour!(example_gaussian_2,colour=:blue))


  # Return the parameters from training
  return (params_cur, neg_toy_elbo(params_cur, games = toy_evidence, num_samples =
num_q_samples))

end
```

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

(d) 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) and the target distribution (in red) on the same axes.

```
# Initial paramters
toy_mu = [-2.,3.]
toy_ls = [0.5,0.]
toy_params_init = (toy_mu, toy_ls)

model_1 = fit_toy_variational_dist(toy_params_init, two_player_toy_games(1, 0);
num_itrs=400, lr= 1e-2, num_q_samples = 10)
```




```
final_loss = model_1[2]
trained_mu = model_1[1][1]
trained_ls = model_1[1][2]
print("Final loss is $final_loss")
```

```
Final loss is 0.7296906745845444
```

```
print("Trained mu is $trained_mu")
```

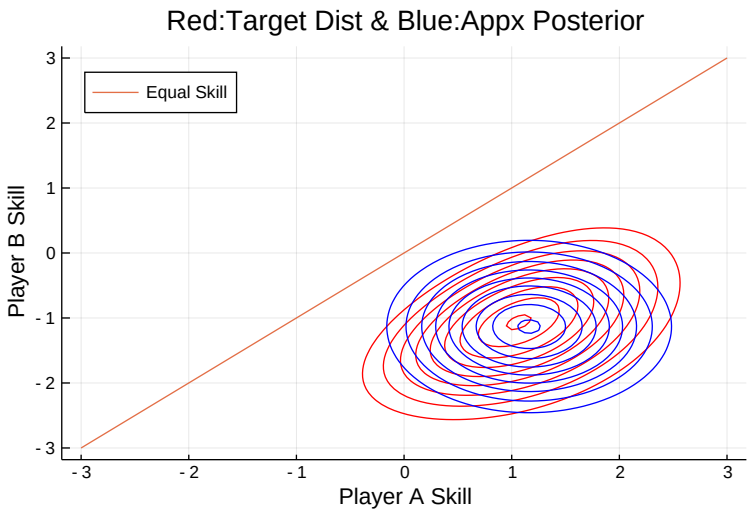
```
Trained mu is [0.3378400781655634, -0.34360368514090717]
```

```
print("Trained ls is $trained_ls")
```

```
Trained ls is [-0.048288492941839595, -0.060540884889007884]
```

(e) 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) and the target distribution (in red) on the same axes.

```
model_10 = fit_toy_variational_dist(toy_params_init, two_player_toy_games(10, 0);
num_itrs=400, lr= 1e-2, num_q_samples = 10)
```




```
final_loss = model_10[2]
trained_mu = model_10[1][1]
trained_ls = model_10[1][2]
print("Final loss is $final_loss")
```

```
Final loss is 3.132405928269167
```

```
print("Trained mu is $trained_mu")
```

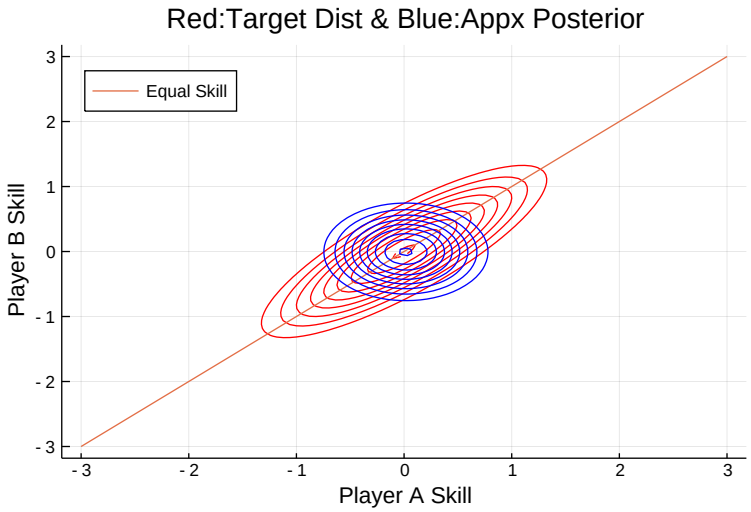
```
Trained mu is [1.1909792600394327, -1.1814301097313296]
```

```
print("Trained ls is $trained_ls")
```

```
Trained ls is [-0.28283390881373627, -0.34215892928753366]
```

(f) 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) and the target distribution (in red) on the same axes.

```
model_20 = fit_toy_variational_dist(toy_params_init, two_player_toy_games(10, 10);
num_itrs=400, lr= 1e-2, num_q_samples = 100)
```




```
final_loss = model_20[2]
trained_mu = model_20[1][1]
trained_ls = model_20[1][2]
print("Final loss is $final_loss")
```

```
Final loss is 15.585080569436094
```

```
print("Trained mu is $trained_mu")
```

```
Trained mu is [0.010135302840711954, 0.006877498940282886]
```

```
print("Trained ls is $trained_ls")
```

```
Trained ls is [-0.853606183640755, -0.8658075904301256]
```

4 Approximate inference conditioned on real data

Firstly, we load the dataset from tennis data.mat containing two matrices: (1) W is a 107 by 1 matrix, whose i′th entry is the name of player i. (2) 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.

```
# 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
```

(a) For any two players i and j,p(zi,zj given all games)is always proportional to p(zi,zj,all games). In general, are the isocontours of p(zi,zj given all games) the same as those of p(zi,zj given games between i 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.

Answer: Yes. The games between other players besides i and j does provide information about the skill of players i and j.

(b) 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.

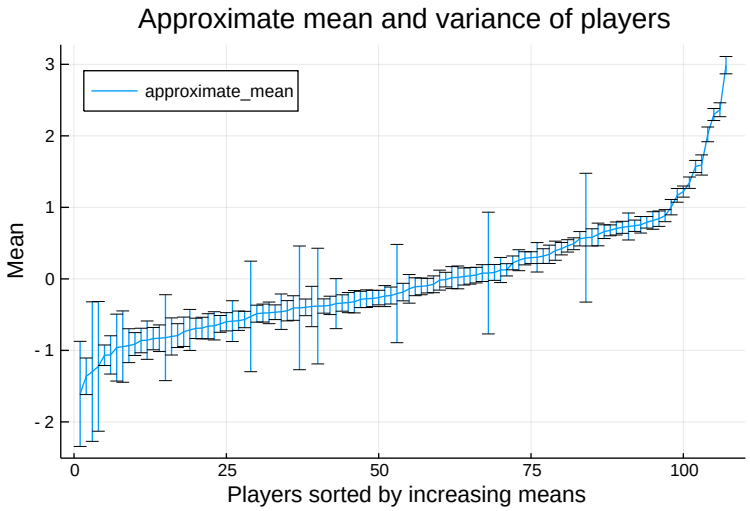
```
function fit_variational_dist(init_params, tennis_games; num_itrs=200, lr= 1e-2,
num_q_samples = 10)

  # Initializing parameters
  params_cur = init_params

  for i in 1:num_itrs
    # gradients of variational objective with respect to parameters
    grad_params = gradient(params -> neg_toy_elbo(params, games = tennis_games,
num_samples = num_q_samples),params_cur)[1]

    # update paramters with lr-sized step in descending gradient
    params_cur = params_cur .- (grad_params  .* lr)

    # reporting the loss in terms of neg_elbo
    @info "loss: $(neg_toy_elbo(params_cur, games = tennis_games, num_samples =
num_q_samples))"
  end

  # returning the trained parameters and final loss
  return (params_cur, neg_toy_elbo(params_cur, games = tennis_games, num_samples =
num_q_samples))

end

# Initializing parameters:
mu = rand(-10:10,107)
ls = sqrt.((rand(107)).^2)
params_init = (mu,ls)

# fitting the new optimization function
real_model = fit_variational_dist(params_init, tennis_games; num_itrs=200, lr= 1e-2,
num_q_samples = 10)
```

```
# Reporting final negative ELBO estimate after optimization
final_loss = real_model[2]
print("Final loss is $final_loss")
```

```
Final loss is 1145.2084258698974
```

(c) Plot the approximate mean and variance of all players, sorted by skill. There's no need to include the names of the players.

```
import Random
Random.seed!(10)

# Initializing parameters:
mu = rand(-10:10,107)
ls = sqrt.((rand(107)).^2)
params_init = (mu,ls)

opt_func = fit_variational_dist(params_init, tennis_games; num_itrs=200, lr= 1e-2,
num_q_samples = 10)

# means of players
mean_players = opt_func[1][1]
logstd_players = opt_func[1][2]
perm = sortperm(mean_players)

# Plotting approximate mean and variance of players
plot(mean_players[perm], yerror=((exp.(logstd_players[perm])).^2), title="Approximate
mean and variance of players",
xlabel = "Players sorted by increasing means",
ylabel = "Mean", legend = :topleft)
savefig(joinpath("plots","Q4.pdf"))
```



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

```
top_10 = player_names[perm[98:107]]
display(reverse(top_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 approximate posterior over the skills of Roger Federer and Rafael Nadal. Use the approximate posterior that you fit in question 4 part b.

```
# Initializing the initial parameters

init_mu = rand(-10:10,107)
init_log_sigma = sqrt.((rand(107)).^2)
init_params = (init_mu, init_log_sigma)

model_RFRN = fit_variational_dist(init_params, tennis_games; num_itrs=200, lr= 1e-2,
num_q_samples = 10)[1]

# Here, initially, we need to find the index of Roger Federer and Rafael Nadal to
get their paramters from our trained model

# Index of Roger Federer
RF_index = findall(player_names -> player_names == "Roger-Federer",player_names)[1][1]
print("Index of Roger Federer is $RF_index")
```

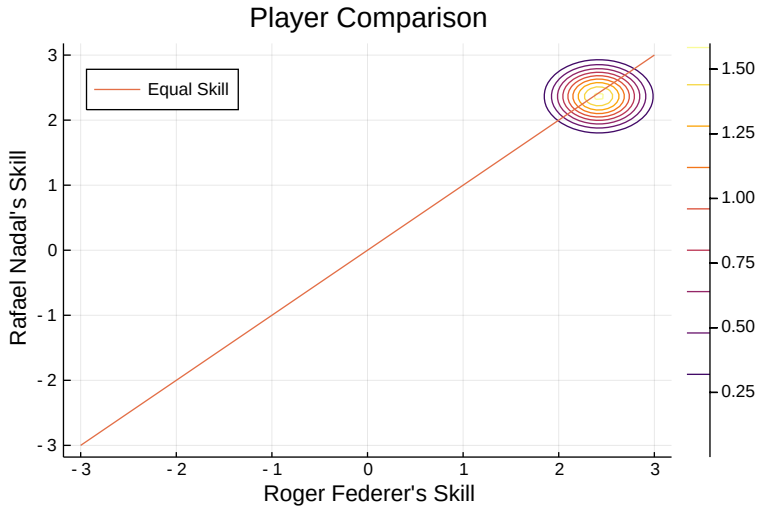
```
Index of Roger Federer is 5
```

```
RN_index = findall(player_names -> player_names == "Rafael-Nadal",player_names)[1][1]
print("Index of Rafael Nadal is $RN_index")
```

```
Index of Rafael Nadal is 1
```

```
# As the index of Roger Federer is 5 and Rafael Nadal is 1, we extract their
parameters from our trained model

RF_mu = model_RFRN[1][5] # Roger Federer's mean
RN_mu = model_RFRN[1][1] # Rafael Nadal's mean
RF_ls = (model_RFRN[2][5]) # Roger Federer's log-sigma
RN_ls = (model_RFRN[2][1]) # Rafael Nadal's log-sigma

RFRN_mu = [RF_mu,RN_mu] # trained mu
RFRN_ls = [RF_ls,RN_ls] # trained log_sigma

params_trained = (RFRN_mu, RFRN_ls)
```

```
zs_RFRN = (params_trained[1] .+ ((exp.(params_trained[2])) .*
randn(length(params_trained[1]), 10)))
plot(reuse = false)
plot(title="Player Comparison",
    xlabel = "Roger Federer's Skill",
    ylabel = "Rafael Nadal's Skill"
   )
example_gaussian_2(zs_RFRN) =
exp.(factorized_gaussian_log_density(params_trained[1],params_trained[2],zs_RFRN))
display(skillcontour!(example_gaussian_2))


display(plot_line_equal_skill!())


savefig(joinpath("plots","Q4_2.pdf"))
```



(f) 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. Express your answer in terms of the cumulative distribution function of a one-dimensional Gaussian random variable.

Here, we want to find $P(Z_a > Z_b)$.

As the first step, we do change of variables such that: $Y_a = Z_a - Z_b$ while $Y_b = Z_b$. So, essentially, we want to find matrix A such that:

$$AZ = A \begin{bmatrix} Z_a \\ Z_b \end{bmatrix} = \begin{bmatrix} Y_a \\ Y_b \end{bmatrix} = \begin{bmatrix} Z_a - Z_b \\ Z_b \end{bmatrix}$$

where,

$$\begin{pmatrix} Z_a \\ Z_b \end{pmatrix} \sim N \left[\begin{pmatrix} \mu_{z_a} \\ \mu_{z_b} \end{pmatrix}, \begin{pmatrix} \sigma^2_{z_a} & 0 \\ 0 & \sigma^2_{z_b} \end{pmatrix} \right]$$

Based on our requirement, A is:

$$\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

As,

$$\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} Z_a \\ Z_b \end{bmatrix} = \begin{bmatrix} Z_a - Z_b \\ Z_b \end{bmatrix} = \begin{bmatrix} Y_a \\ Y_b \end{bmatrix}$$

Moreover, we know that if:

$$Z \sim N(\mu, \sum)$$

then, if A is linear transformation, then:

$$AZ \sim N(A\mu, A \sum A^T)$$

As a result, based on the above result, our mean for AZ would be $A\mu$ which is:

$$A\mu = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mu_{z_a} \\ \mu_{z_b} \end{bmatrix} = \begin{bmatrix} \mu_{z_a} - \mu_{z_b} \\ \mu_{z_b} \end{bmatrix}$$

While, based on the above result, our variance for AZ would be $A \sum A^T$ which is:

$$A \sum A^T = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma^2_{z_a} & 0 \\ 0 & \sigma^2_{z_b} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} \sigma^2_{z_a} & -\sigma^2_{z_b} \\ 0 & \sigma^2_{z_b} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} \sigma^2_{z_a} + \sigma^2_{z_b} & -\sigma^2_{z_b} \\ -\sigma^2_{z_b} & \sigma^2_{z_b} \end{bmatrix}$$

As a result,

$$\begin{bmatrix} Y_a \\ Y_b \end{bmatrix} \sim N \left[\begin{pmatrix} \mu_{z_a} - \mu_{z_b} \\ \mu_{z_b} \end{pmatrix}, \begin{pmatrix} \sigma^2_{z_a} + \sigma^2_{z_b} & -\sigma^2_{z_b} \\ -\sigma^2_{z_b} & \sigma^2_{z_b} \end{pmatrix} \right]$$

Now, originally, we wanted $P(Z_a > Z_b)$. However, as:

$$\begin{bmatrix} Y_a \\ Y_b \end{bmatrix} = \begin{bmatrix} Z_a - Z_b \\ Z_b \end{bmatrix}$$

that is equivalent to finding $P(Y_a > 0)$, where

$$Y_a \sim N(\mu_{z_a} - \mu_{z_b}, \sigma^2_{z_a} + \sigma^2_{z_b})$$

As a result,

$$
\begin{aligned}
P(Y_a > 0) &= 1 - P(Y_a < 0) \\
&= 1 - \Phi_{Y_a}(0) \\
&= 1 - \Phi(\frac{0 - (\mu_{z_a} - \mu_{z_b})}{\sqrt{(\sigma^2_{z_a} + \sigma^2_{z_b})}}) \\
&= 1 - \Phi(\frac{\mu_{z_b} - \mu_{z_a}}{\sqrt{(\sigma^2_{z_a} + \sigma^2_{z_b})}}) \\
&= 1 - cdf(\frac{\mu_{z_b} - \mu_{z_a}}{\sqrt{(\sigma^2_{z_a} + \sigma^2_{z_b})}}) \\
&= cdf(\frac{\mu_{z_a} - \mu_{z_b}}{\sqrt{(\sigma^2_{z_a} + \sigma^2_{z_b})}}) \\
&= \boxed{\Phi(\frac{\mu_{z_a} - \mu_{z_b}}{\sqrt{(\sigma^2_{z_a} + \sigma^2_{z_b})}})}
\end{aligned}
$$

(g) Using the formula from part c, compute the exact probability under your approximate posterior that Roger Federer has higher skill than Rafael Nadal. Then, estimate it using simple Monte Carlo with 10000 examples, again using your approximate posterior.

```
# Finding z-score
z_score = (RN_mu-RF_mu) / (exp(RF_ls)^2 + exp(RN_ls)^2)^0.5

using Distributions

# Finding the exact probability based on (f)
display(1-cdf(Normal(0,1),z_score))
```

```
0.574279936355574
```

```
# Using Monte Carlo with 10000 samples
samples = 10000
skills = (params_trained[1] .+ ((exp.(params_trained[2])) .*
randn(length(params_trained[1]), samples)))
RF_skill = skills[1,:]
RN_skill = skills[2,:]
mc_est = count(RF_skill .> RN_skill)/samples

# Monte Carlo estimate
display(mc_est)
```

```
0.5711
```

Looking at exact probability under approximate posterior and comparing that with the simple Monte Carlo estimate, we observe that they are very similar.

(h) Using the formula from part c, 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, again using your approximate posterior.

```
low_mean_ind = perm[1] # as "perm" has sorted players according to their skills in
ascending order
print("Index of player with lowest mean is $low_mean_ind")
```

```
Index of player with lowest mean is 98
```

```
# Extracting the parameters of player with lowest mean skill
least_mu = model_RFRN[1][low_mean_ind]
least_ls = model_RFRN[2][low_mean_ind]

# Trained parameters
RFL_mu = [RF_mu,least_mu]
RFL_ls = [RF_ls,least_ls]
params_trained_RFL = (RFL_mu, RFL_ls)
```

```
# Finding z-score
z_score2 = (least_mu-RF_mu) / (exp(RF_ls)^2 + exp(least_ls)^2)^0.5

using Distributions

# Finding the exact probability based on (f)
display(1-cdf(Normal(0,1),z_score2))
```

```
0.9983087384446695
```

```
# Using Monte Carlo with 10000 samples
samples = 10000
skills_2 = (params_trained_RFL[1] .+ ((exp.(params_trained_RFL[2])) .*
randn(length(params_trained_RFL[1]), samples)))
RF_skill2 = skills_2[1,:]
least_skill = skills_2[2,:]
mc_est_2 = count(RF_skill2 .> least_skill)/samples

# Monte Carlo estimate
display(mc_est_2)
```

```
0.9984
```

Again, looking at exact probability under approximate posterior and comparing that with the simple Monte Carlo estimate, we observe that they are very similar.

(i) 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?

Answer: In this section, answers (c) and (e) would change. (b) can not change as it is just defining a new optimization function. (d) will not change as we are only changing the initialization of the prior and that too with the same variance. Essentially, we are just shifting

the mean for all players. So, the ranking we get will still be the same. For (g) and (h), the probabilities would be quite similar in expectation. So, only (c) and (e) would be the one to change. (where essentially we would just observe shift in the plot and not the shape)