Assignment 2 Stochastic Variational Inference in the TrueSkill Model

Gang Peng # 1002961921

March 14, 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.

1 0.1 Model definition

1.1 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 × 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

We are given The prior over each player′s skill is a standard normal distribution, and all player′s skills are a prior independent. That is each z_i is iid N(0,1).

```
using Statistics: mean
using Zygote
using Logging

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 log_prior(zs)
  factorized_gaussian_log_density(0,0,zs)
end
```

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

(b) [3 points] 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.

```
using StatsFuns: log1pexp
function logp_a_beats_b(za,zb)
  return log.(1 ./exp.(log1pexp.(-(za .- zb))))
end
```

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

(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 ×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. 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)
  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
```

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

(d) [2 points] 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)

Again by given indpendency, the joint density is the product of independent density while the joint log density is the sum of independent density.

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

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

```
using Test
@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)
```

1.2 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.

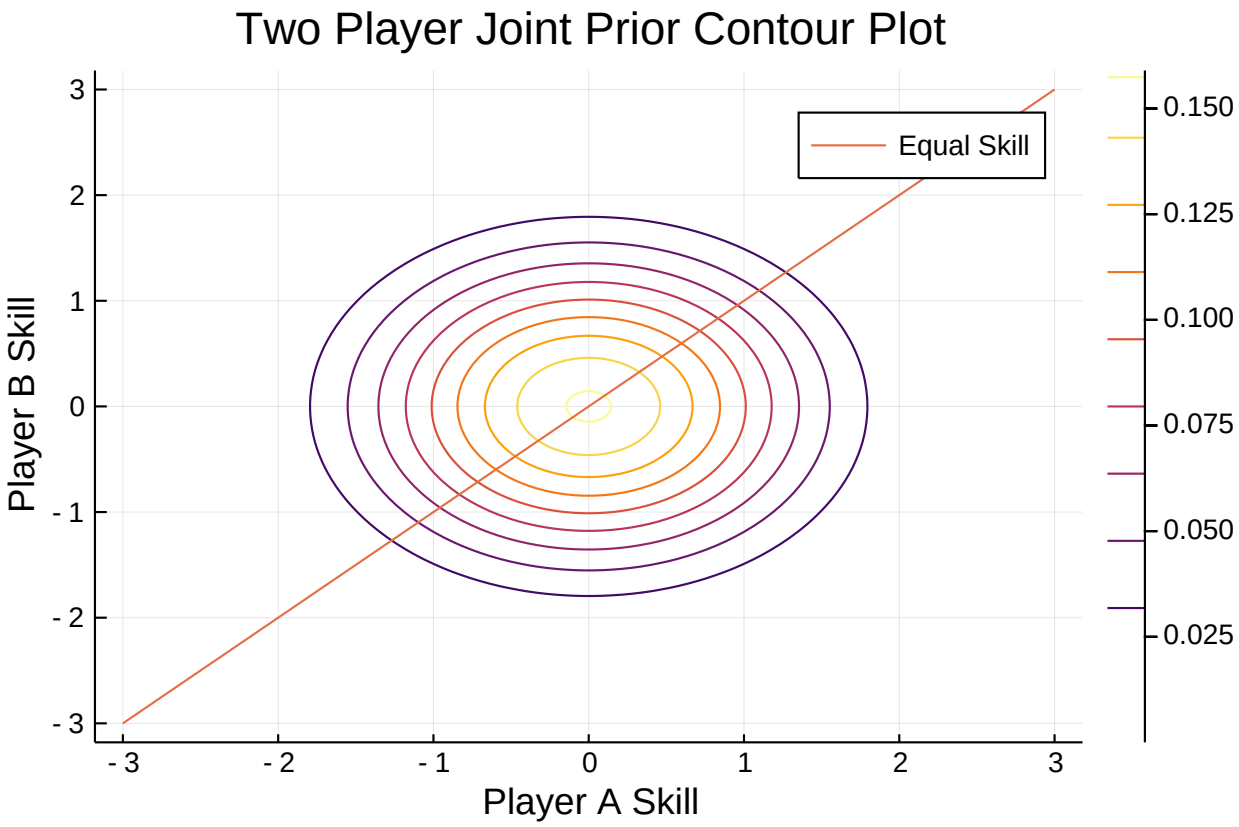
Provided in the starter code is a function skillcontour! which evaluates a provided function on a grid of zA and zB's and plots the isocontours of that function. As well there is a function plot line equal skill!. We have included an example for how you can use these functions.

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, zA = zB. Hint: you've already implemented the log of the likelihood function.

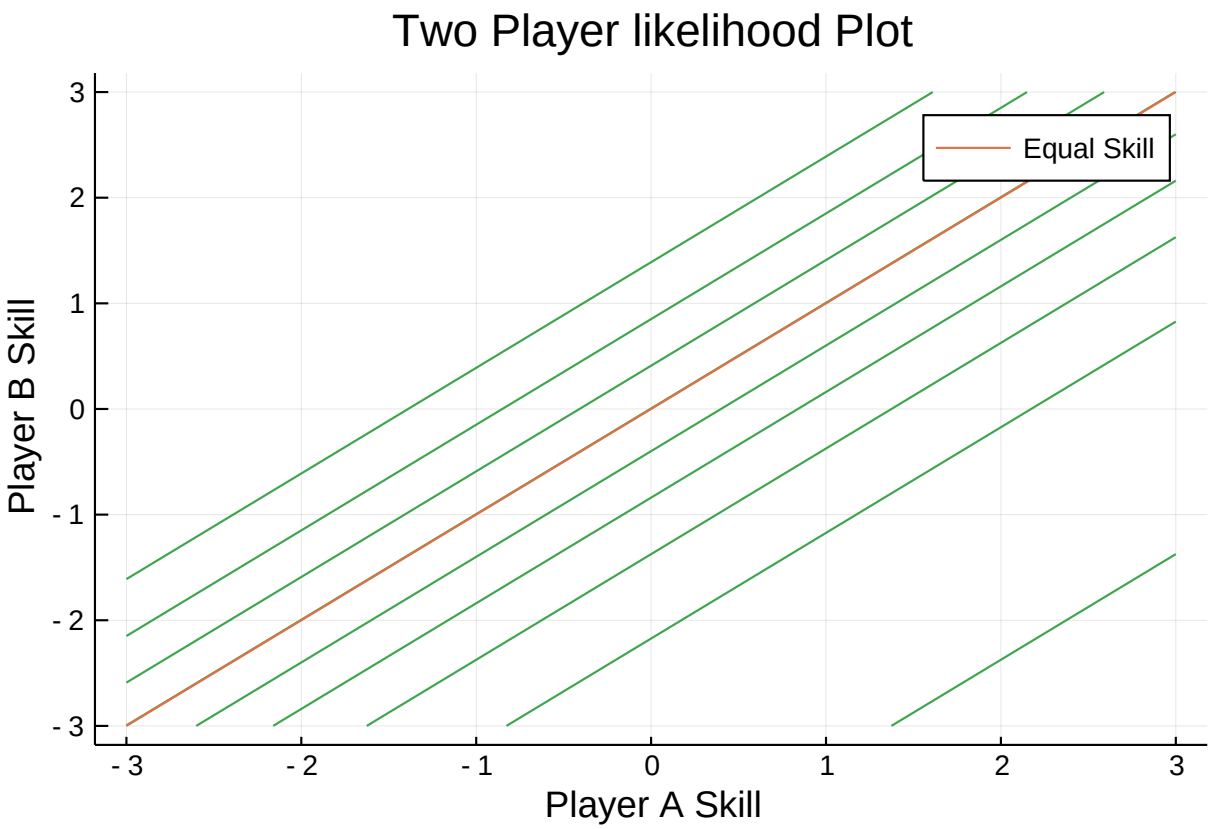
```
using Plots
function skillcontour!(f; colour=nothing)
  n = 100
  x = range(-3,stop=3,length=n)
  y = range(-3,stop=3,length=n)
  z_grid = Iterators.product(x,y) # meshgrid for contour
  z_grid = reshape.(collect.(z_grid),:,1) # add single batch dim
  z = f.(z_grid)
  z = getindex.(z,1)'
  max_z = 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

# 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)]...)

jointPrior(zs) = exp.(log_prior(zs))
plot(title="Two Player Joint Prior Contour Plot",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
```

```
skillcontour!(jointPrior)
plot_line_equal_skill!()
```



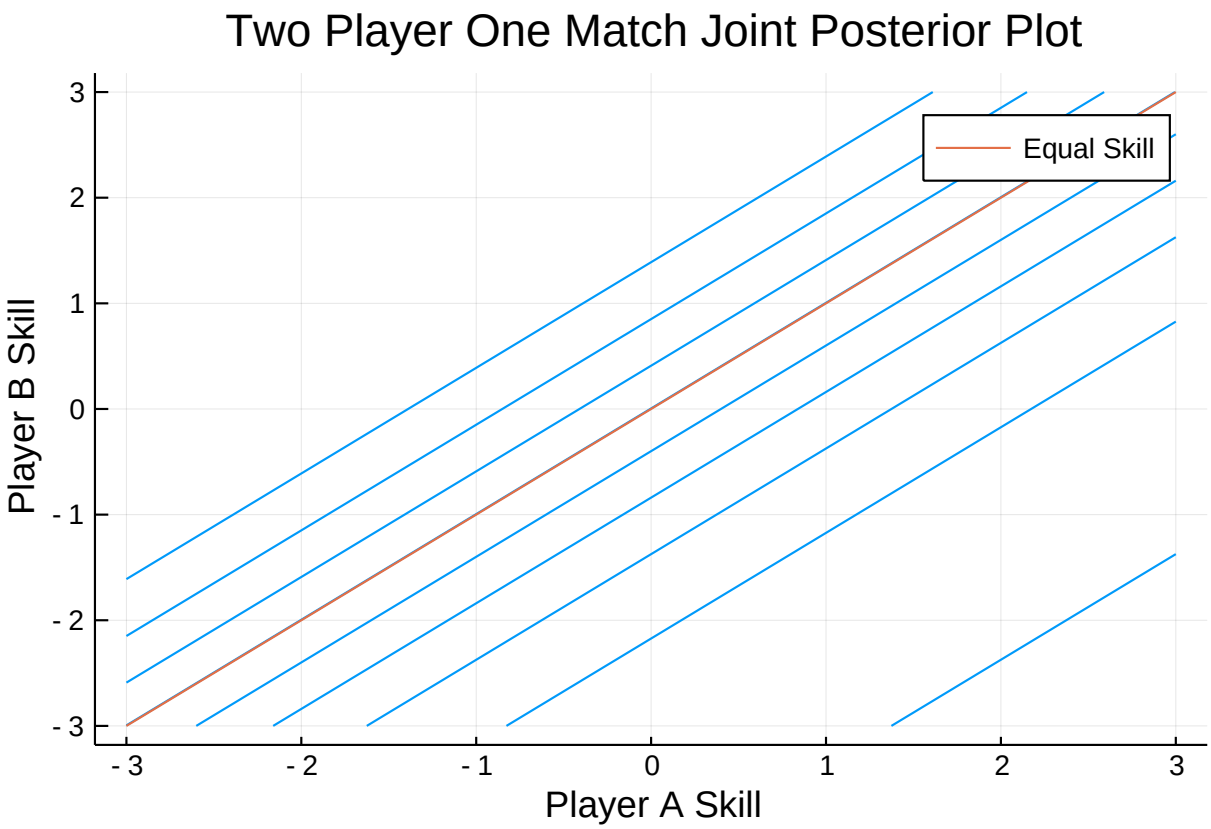
(b) [2 points] Plot the isocontours of the likelihood function. Also plot the line of equal skill, zA = zB.

```
likelihood(zs)=exp.(logp_a_beats_b(zs[1],zs[2]))
plot(title="Two Player likelihood Plot",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
skillcontour!(likelihood, colour=3)
plot_line_equal_skill!()
```



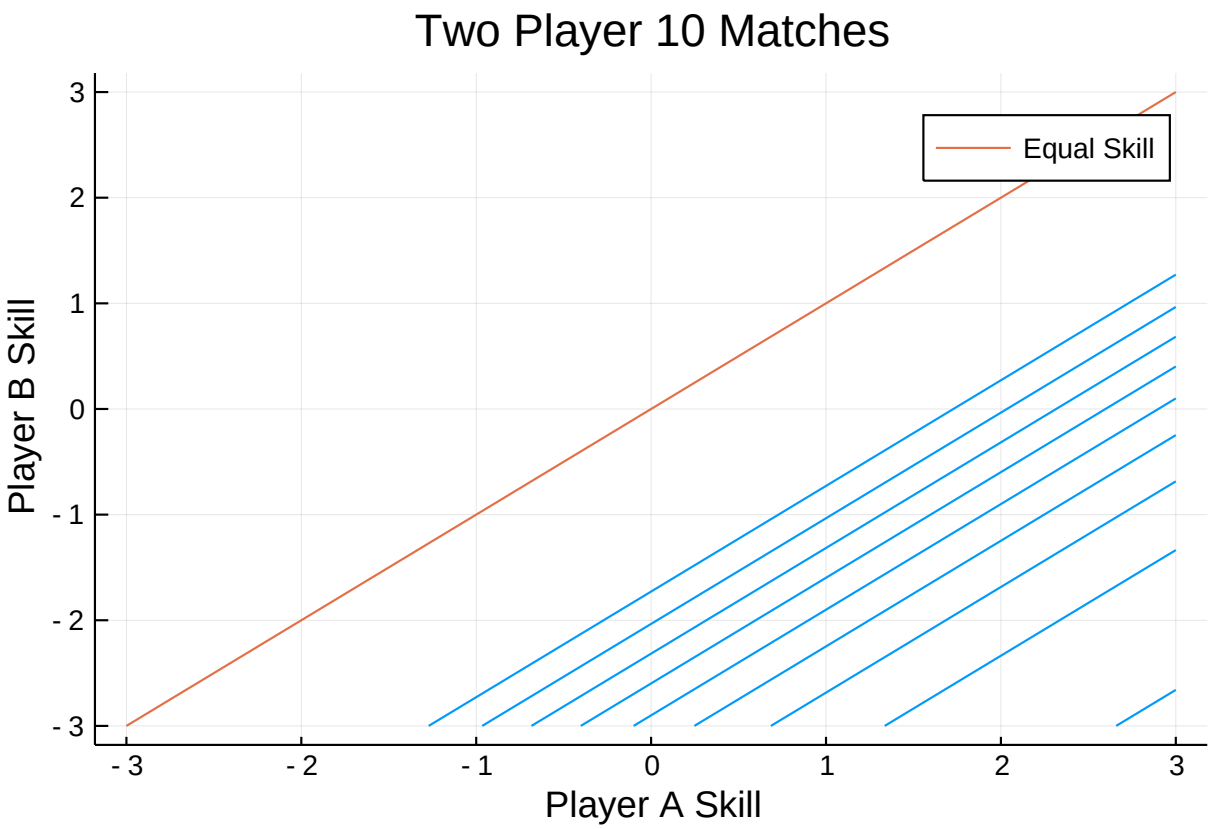
(c) [2 points] Plot isocountours of the joint posterior over zA and zB given that player A beat player B in one match. Since the contours don′t depend on the normalization constant, you can simply plot the isocontours of the log of joint distribution of p(zA,zB,A beat B) Also plot the line of equal skill, zA = zB.

```
games=two_player_toy_games(1, 0)
jt(zs)=exp(all_games_log_likelihood(zs,games))
plot(title="Two Player One Match Joint Posterior Plot",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
skillcontour!(jt,colour=1)
plot_line_equal_skill!()
```



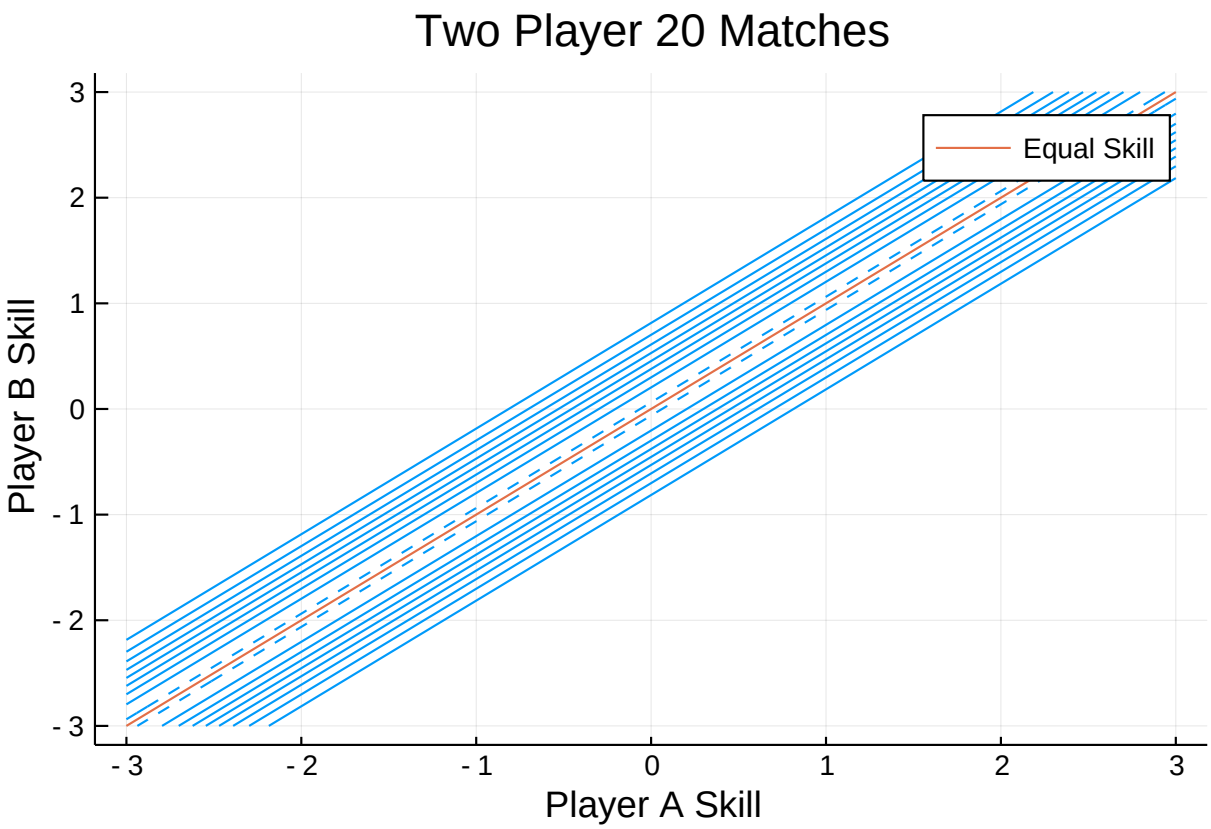
(d) [2 points] Plot isocountours of the joint posterior over zA and zB given that 10 matches were played, and player A beat player B all 10 times. Also plot the line of equal skill, zA = zB.

```
games=two_player_toy_games(10, 0)
jt10(zs)=exp.(all_games_log_likelihood(zs,games))
plot(title="Two Player 10 Matches",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
skillcontour!(jt10,colour=1)
plot_line_equal_skill!()
```



(e) [2 points] 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. For all plots, label both axe

```
games=two_player_toy_games(10, 10)
jt20(zs)=exp.(all_games_log_likelihood(zs,games))
plot(title="Two Player 20 Matches",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
skillcontour!(jt20,colour=1)
plot_line_equal_skill!()
```



2 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 speci

cation from the approxi- mate 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 infer- ence 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—data), and an approximate posterior, $q_\Phi(z|data)$, plus an unknown constant. Use a fully-factorized Gaussian distribution for $q_\Phi(z|data)$. This estimator takes the following arguments:

- params, the parameters ϕ of the approximate posterior $q_\Phi(z|data)$.
- A function logp, which is equal to the true posterior plus a constant. This function must take abatch 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.

```
function elbo(params,logp,num_samples)
  #Generate random samples from uniform distribution
  U=rand(size(params[1])[1],num_samples)
  #Reparametrization to genearte Gaussian of desired parameter
  zs = sqrt.(-2.0 .* log.(U)) .* cos.(2*pi .* U) .*params[2] .+ params[1]
  log_z=factorized_gaussian_log_density(0,0,zs)
  logp_estimate = logp
  log_data=logp .- log_z #Separate data from logp
  #estimate $q_{\Phi}(z|data)$
  logq_estimate = factorized_gaussian_log_density(params[1],params[2],zs) .+ log_data
  return sum(logp_estimate .- logq_estimate) ./ num_samples
end
```

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

(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.

```
function neg_toy_elbo(params; games = two_player_toy_games(1,0), num_samples = 100)
  zs=randn(size(params[1])[1],num_samples)
  logp = joint_log_density(zs,games)
  return -elbo(params,logp, num_samples)
end
```

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

(c) [5 points] Write an optimization function called fit toy variational dist which takes initial vari- ational 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.

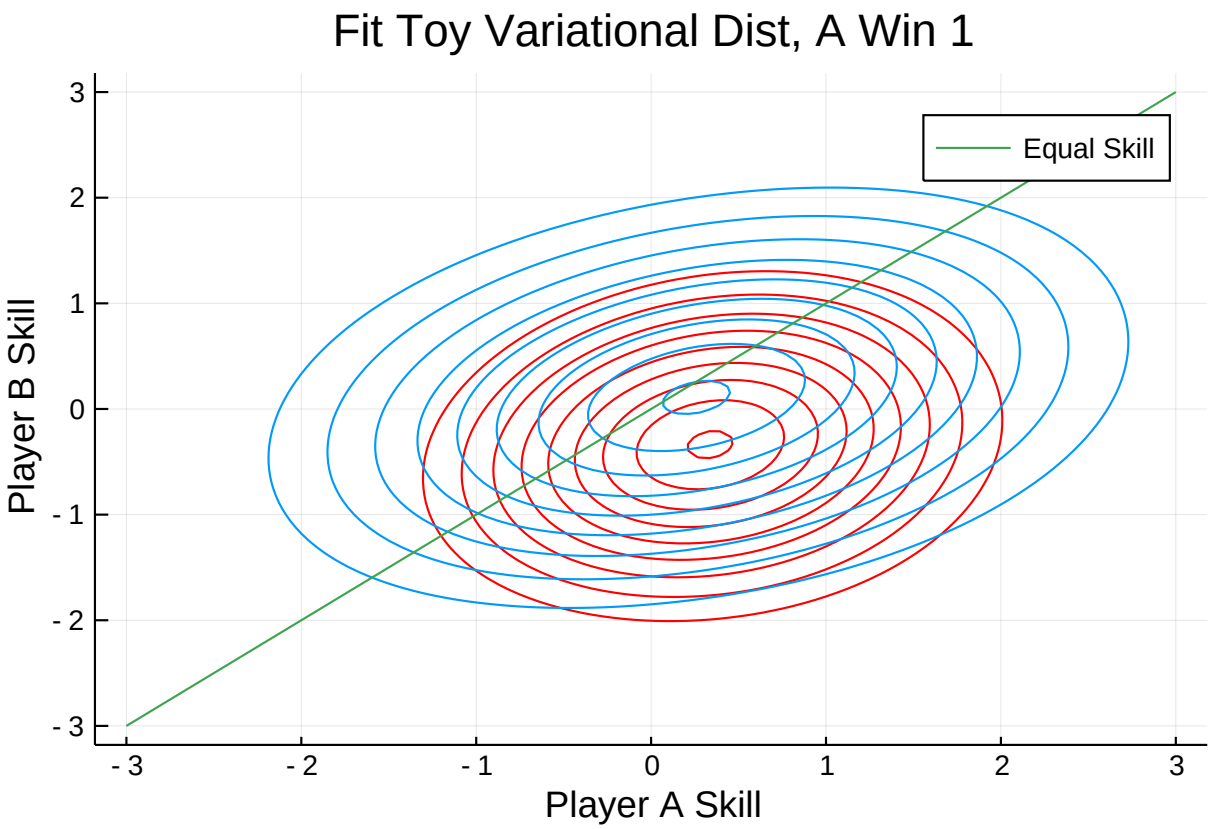
```
function fit_toy_variational_dist(init_params, toy_evidence; num_itrs=200, lr= 1e-6,
num_q_samples = 10)
  params_cur = init_params
  elbo_val = neg_toy_elbo(params_cur; games = toy_evidence, num_samples = num_q_samples)
  #Generate true prior
  pzs=randn(size(init_params[1])[1],num_q_samples)
  jointp(pzs)=exp.(joint_log_density(pzs,toy_evidence)) #function for contour plot

  #initialize plot comment out during compile jmd
  #plot(title="Fit Toy Variational Dist",
  #     xlabel = "Player A Skill",
  #     ylabel = "Player B Skill"
```

```
  #   )

  for i in 1:num_itrs
    f(params) = neg_toy_elbo(params; games = toy_evidence, num_samples = num_q_samples)
    grad_params = gradient(f, params_cur)[1]
    params_cur =  params_cur .- grad_params .* lr
    elbo_val = neg_toy_elbo(params_cur; games = toy_evidence, num_samples = num_q_samples)
    #@info "loss: $(elbo) "
    #Note: the following code do the required ploting but comment out during compile
jmd to avoid too much plots in the final file
    #U=rand(size(init_params[1])[1],num_q_samples)
    #qzs = sqrt.(-2.0 .* log.(U)) .* cos.(2*pi .* U) .*params_cur[2] .+ params_cur[1]
    #jointq(qzs)=exp.(factorized_gaussian_log_density(params_cur[1],params_cur[2],qzs) .+
    #all_games_log_likelihood(qzs,toy_evidence))
    #display(skillcontour!(jointq,colour=1))
  end
  #plot_line_equal_skill!()

  return params_cur, elbo_val
end

fit_toy_variational_dist (generic function with 1 method)
```

(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

nal loss. Also plot the optimized variational approximation contours (in blue) aand the target distribution (in red) on the same axes.

```
num_players_toy = 2
toy_mu = [-1.,0.8] # Initial mu, can initialize randomly!
toy_ls = [0.5,0.2] # Initual log_sigma, can initialize randomly!
toy_params_init = (toy_mu, toy_ls)
toy_evidence=two_player_toy_games(1,0)
fit=fit_toy_variational_dist(toy_params_init, toy_evidence; num_itrs=200, lr= 1e-8,
num_q_samples = 10)
opt_params=fit[1]
pzs=randn(size(toy_params_init[1])[1],10)
jointp(pzs)=exp.(joint_log_density(pzs,toy_evidence)) #function for contour plot
U=rand(size(toy_params_init[1])[1],10)
qzs = sqrt.(-2.0 .* log.(U)) .* cos.(2*pi .* U) .*opt_params[2] .+ opt_params[1]
jointq(qzs)=exp.(factorized_gaussian_log_density(opt_params[1],opt_params[2],qzs) .+
all_games_log_likelihood(qzs,toy_evidence))

# plot result
print("Final loss:",fit[2])

Final loss:0.2920520552856018

plot(title="Fit Toy Variational Dist, A Win 1",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
skillcontour!(jointp,colour="red")
skillcontour!(jointq,colour=1)
plot_line_equal_skill!()
```



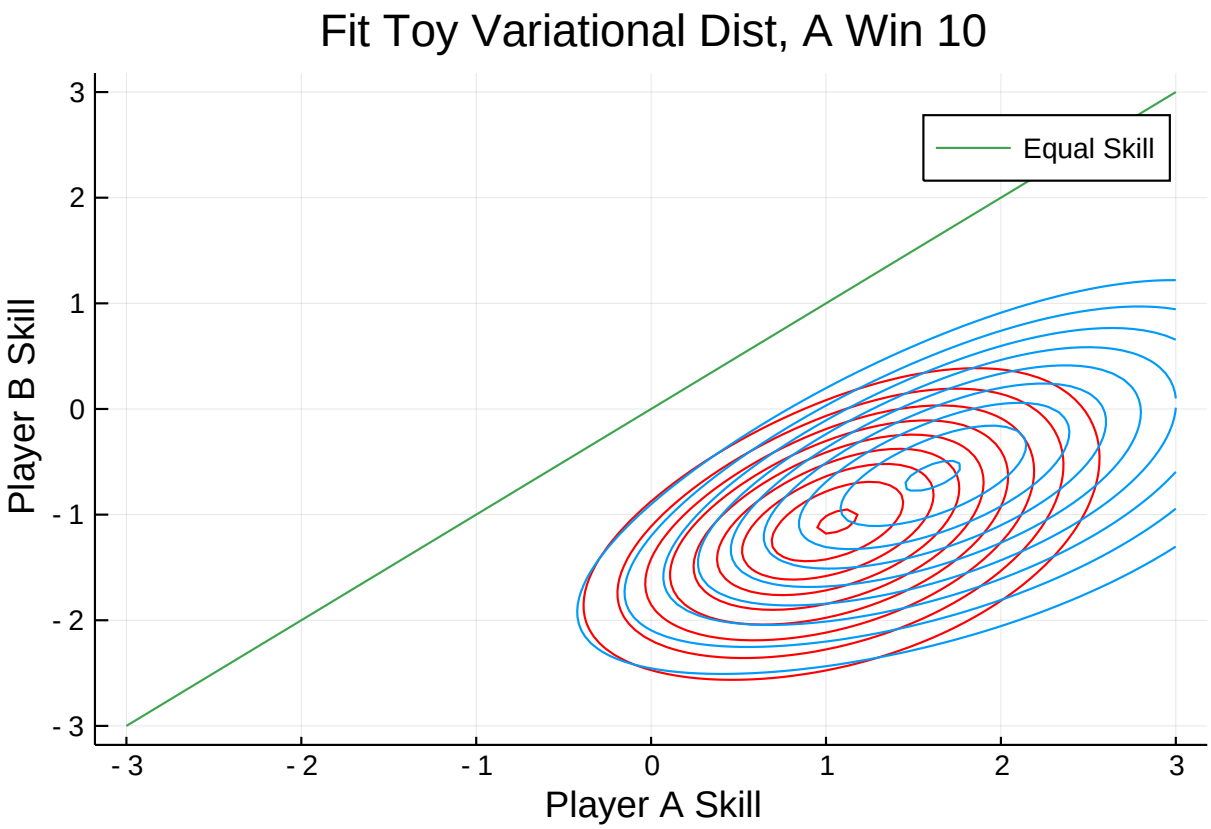
(e) [2 points] Initialize a variational distribution parameters and optimize them to approximate the joint where we observe player A winning 10 games. Report the

nal loss. Also plot the optimized variational approximation contours (in blue) aand the target distribution (in red) on the same axes.

```
num_players_toy = 2
toy_mu = [-1.,0.8] # Initial mu, can initialize randomly!
toy_ls = [0.5,0.2] # Initual log_sigma, can initialize randomly!
toy_params_init = (toy_mu, toy_ls)
toy_evidence=two_player_toy_games(10,0)
fit=fit_toy_variational_dist(toy_params_init, toy_evidence; num_itrs=200, lr= 1e-8,
num_q_samples = 10)
opt_params=fit[1]
pzs=randn(size(toy_params_init[1])[1],10)
jointp(pzs)=exp.(joint_log_density(pzs,toy_evidence)) #function for contour plot
U=rand(size(toy_params_init[1])[1],10)
qzs = sqrt.(-2.0 .* log.(U)) .* cos.(2*pi .* U) .*opt_params[2] .+ opt_params[1]
jointq(qzs)=exp.(factorized_gaussian_log_density(opt_params[1],opt_params[2],qzs) .+
all_games_log_likelihood(qzs,toy_evidence))

# plot result
print("Final loss:",fit[2])
```

```
Final loss:0.15529873242181208
```

```
plot(title="Fit Toy Variational Dist, A Win 10",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
skillcontour!(jointp,colour="red")
skillcontour!(jointq,colour=1)
plot_line_equal_skill!()
```



(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

nal loss. Also plot the optimized variational approximation contours (in blue) aand the target distribution (in red) on the same axes.

```
num_players_toy = 2
toy_mu = [-1.,0.8] # Initial mu, can initialize randomly!
toy_ls = [0.5,0.2] # Initual log_sigma, can initialize randomly!
toy_params_init = (toy_mu, toy_ls)
toy_evidence=two_player_toy_games(10,10)
fit=fit_toy_variational_dist(toy_params_init, toy_evidence; num_itrs=200, lr= 1e-8,
num_q_samples = 10)
opt_params=fit[1]
pzs=randn(size(toy_params_init[1])[1],10)
jointp(pzs)=exp.(joint_log_density(pzs,toy_evidence)) #function for contour plot
U=rand(size(toy_params_init[1])[1],10)
qzs = sqrt.(-2.0 .* log.(U)) .* cos.(2*pi .* U) .*opt_params[2] .+ opt_params[1]
jointq(qzs)=exp.(factorized_gaussian_log_density(opt_params[1],opt_params[2],qzs) .+
all_games_log_likelihood(qzs,toy_evidence))

# plot result
print("Final loss:",fit[2])
```

```
Final loss:0.07041095506797088
```

```
plot(title="Fit Toy Variational Dist, A,B Both Win 10",
    xlabel = "Player A Skill",
    ylabel = "Player B Skill"
   )
skillcontour!(jointp,colour="red")
skillcontour!(jointq,colour=1)
```

```
plot_line_equal_skill!()
```