Assignment 2: Variational Inference in the TrueSkill model

- Name: Tianyi Liu
- StudentNumber: 1005820827
- Collaborators: Lazar Atanackovic, Phil Fradkin

Goal

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

Outline

1. Introduce our problem and describe our model.
2. Implement our model.
3. Investigate the model on easy-to-visualize toy data.
4. Implement Stochastic Variational Inference (SVI) objective (ELBO) manually, using Automatic Differentiation (e.g. **Zygote.jl**).
5. Use our SVI implementation to learn an approximate distribution inferred from the toy data and visualize.
6. Use SVI to perform approximate inference on real data. A collection of games played by by Woman Grandmasters on chess.com. The data was modified from **this Kaggle Dataset**.
7. Use variational approximation to estimate inference questions using the model.

Background

We will 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.

Here are some readings you can familiarize yourself with the problem and their model. Note that we will use different inference methods than those discussed in these works (message passing):

- **The 2006 technical report at Microsoft Research**
- **The 2007 NeurIPS conference paper introducing TrueSkill**
- **The 2018 followup work TrueSkill 2**

Our assignment is based on **one developed by Carl Rasmussen at Cambridge for his course on probabilistic machine learning**. In their assignment they implement and utilize a Gibbs sampling method for approximate inference. We will not implement an MCMC sampling method, instead using SVI for our approximate inference.

1. Problem and Model Description

We will consider a simplified version of the TrueSkill model to model the skill of a individual players i at a 2-player game: chess.

We assume that each player has an unknown skill $z_i \in \mathbb{R}$. These are called latent variables to our model since we cannot observe the players' skill directly.

Given our list of players I, we have no way *a priori* (before we observe data) to infer the players' unknown skill. If we were familiar with the roster of Woman Grandmaster (WGM) chess players, though, we could incorporate our *prior information* about the players to suggest plausible skills. However, these initial priors can be biased, e.g. player fame, charisma, playstyle.

Instead, we will simply assume that all players' skill is distributed according to a standard Gaussian distribution.

We also assume that all players' skills are *a priori* independent.

Because we cannot monitor players' skill directly, we must collect other evidence. We observe the players' performance against other players in a series of games. We will use the observed game data to update our model of the players' skill.

Our model of the likelihood that player i beats player j in a game of chess depends on our model of their respective skills z_i, z_j. We will use the following likelihood model:

$$p(\text{game}(\text{i wins, j loses}) \mid z_i, z_j) = \sigma(z_i - z_j)$$

where

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

Note that $log(1 + exp(y))$ can suffer significant numerical instability if not implemented carefully. You should use a function like `StatsFuns.log1pexp` or its `numpy` equivalent.

We also assume that each game is independent from the others, given the players' skill.

Data

The observations of game outcomes are collected in an array. Each column contains a pair of player indices, i, j for the chess game. The first index, player i is the winner, and so j lost the game.

If we observe M games played then the data is collected into an array with shape $2 \times M$.

Note that data retains its dimension even as we observe more players. Additional players are indentified by their index. If we wanted to consider multiplayer games, like Halo 2, we could increase the first dimension, from our 2-player chess game.

```
using StatsFuns: log1pexp
```

2. Implementing the Model

- log-prior of skills zs
- log-likelihood of observing game with player i beating player j
- log-likelihood of observing collection of games with players' skills zs
- log-likelihood of joint distribution observed games and skills zs

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

```
function log_prior(zs)
    #TODO
    return sum(log(1 / √(2 * π)) .+ (- zs .^ 2 ./ 2), dims=2)
end
```

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

```
function logp_i_beats_j(zi,zj)
    #TODO
    return - log1pexp.(-(zi - zj))
end
```

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

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

```
end
```

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

```
function joint_log_density(zs,games)
    #hint: you have likelihood and prior, use Bayes
    #TODO
    return all_games_log_likelihood(zs, games) + log_prior(zs)
end
```

3. Visualize the Model on Toy Data

To check our understanding of the data and our model we consider a simple scenario.

Toy Data

Let's model the chess skills for *only* two players, A and B. Restricting to 2-dimensions allows us to visualize the *posterior* distribution of our model. Unlike the high-dimensional distributions we would model with real data, we can evaluate the model on a grid of points, like numerical integration, to produce a plot of contours for the possibly complicated posterior.

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

2D Posterior Visualization

You can use the function `skillcontour!` to perform this grid of evaluations over the 2D latent space $z_A \times z_B$. This will plot isocontours, curves of equal density, for the posterior distribution.

As well, `plot_line_equal_skill!` is provided to simply indicate the region of latent space corresponding to the players having equal skill, $z_A = z_B$.

(the default settings for these plotting functions may need modification.)

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

```
begin
    two_player_toy_games(p1_wins, p2_wins) = cat(collect.([repeat([1,2]',p1_wins)',
repeat([2,1]',p2_wins)']), dims=2)
end
```

```
2×8 Array{Int64,2}:
 1  1  1  1  1  2  2  2
 2  2  2  2  2  1  1  1
```

```
two_player_toy_games(5,3)
```

```
using Plots
```

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

```
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
    # modified for dim convention
    # z_grid = reshape.(collect.(z_grid),:,1) # add single batch dim
    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
```

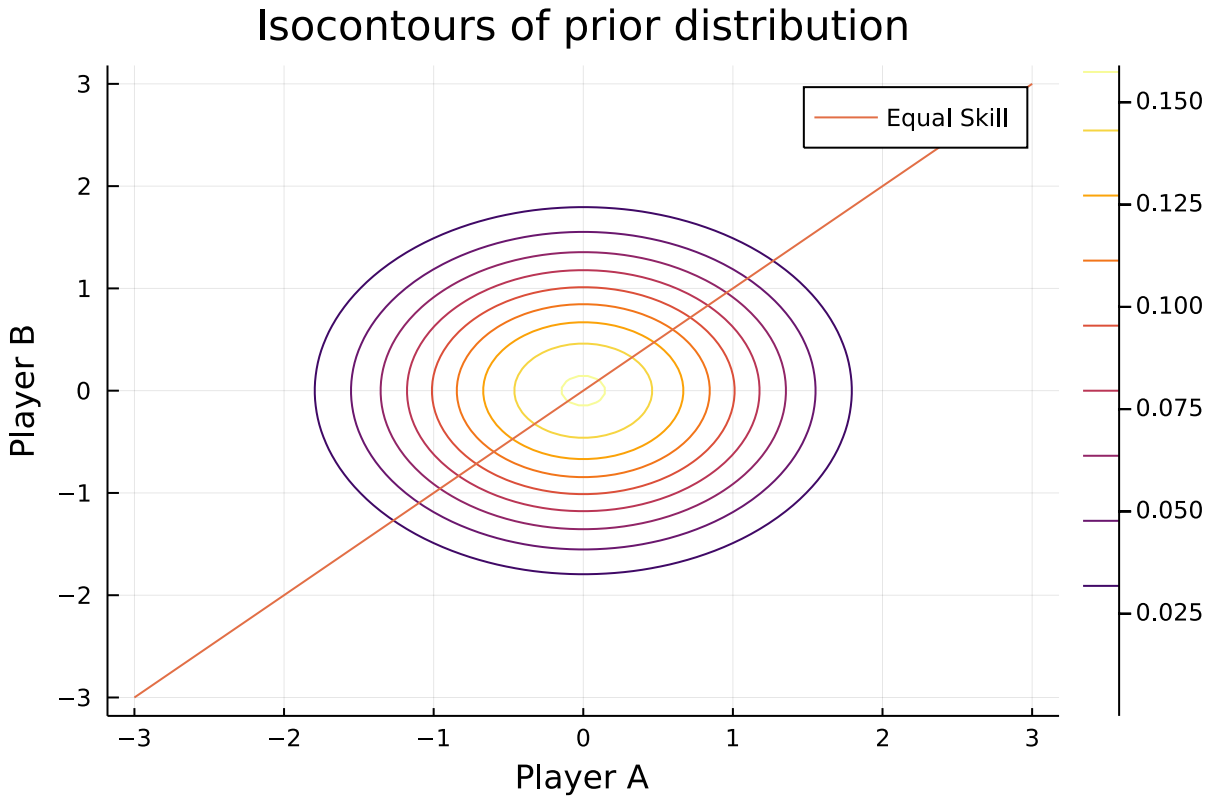
```
plot_line_equal_skill! (generic function with 1 method)
```

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

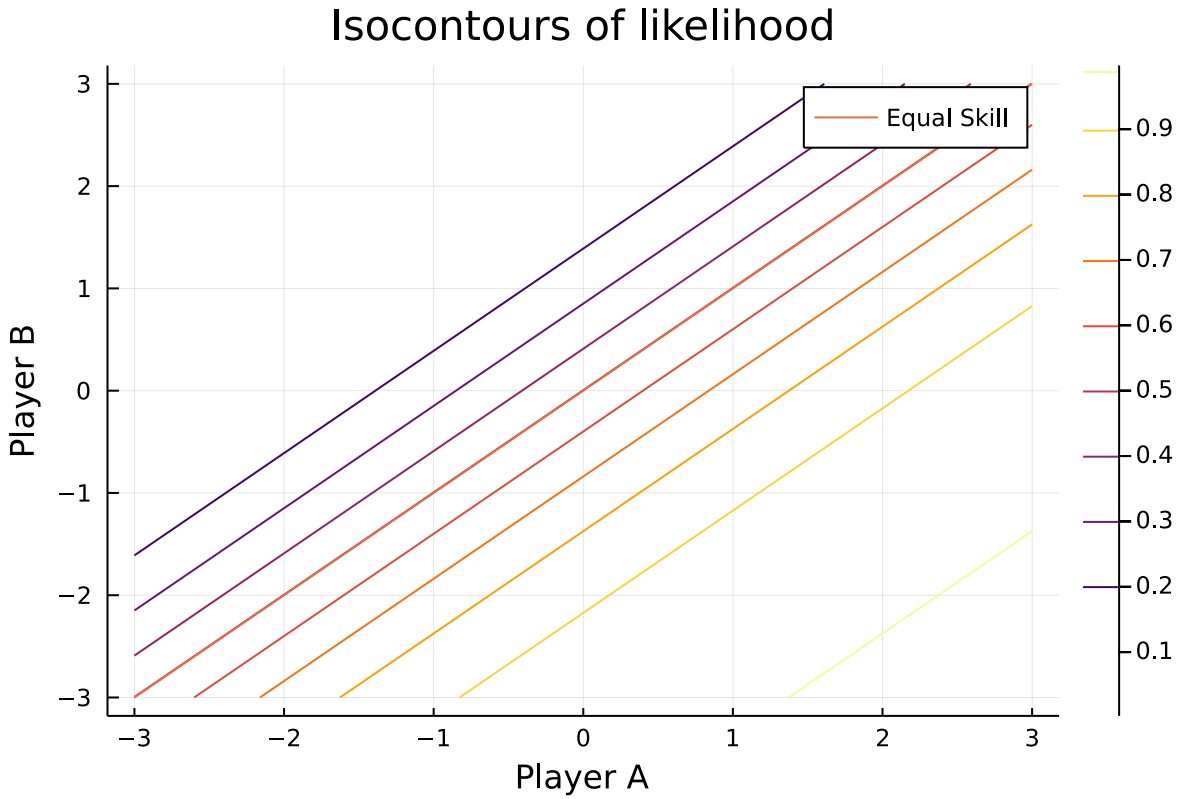
Plot the following distributions for two players A and B. In each, also plot the line of equal skill $z_A = z_B$.

Note, before plotting convert from log-density to density. Since there is only 1 distribution per plot, we are not comparing distributions, we do not need to worry about normalization of the contours.

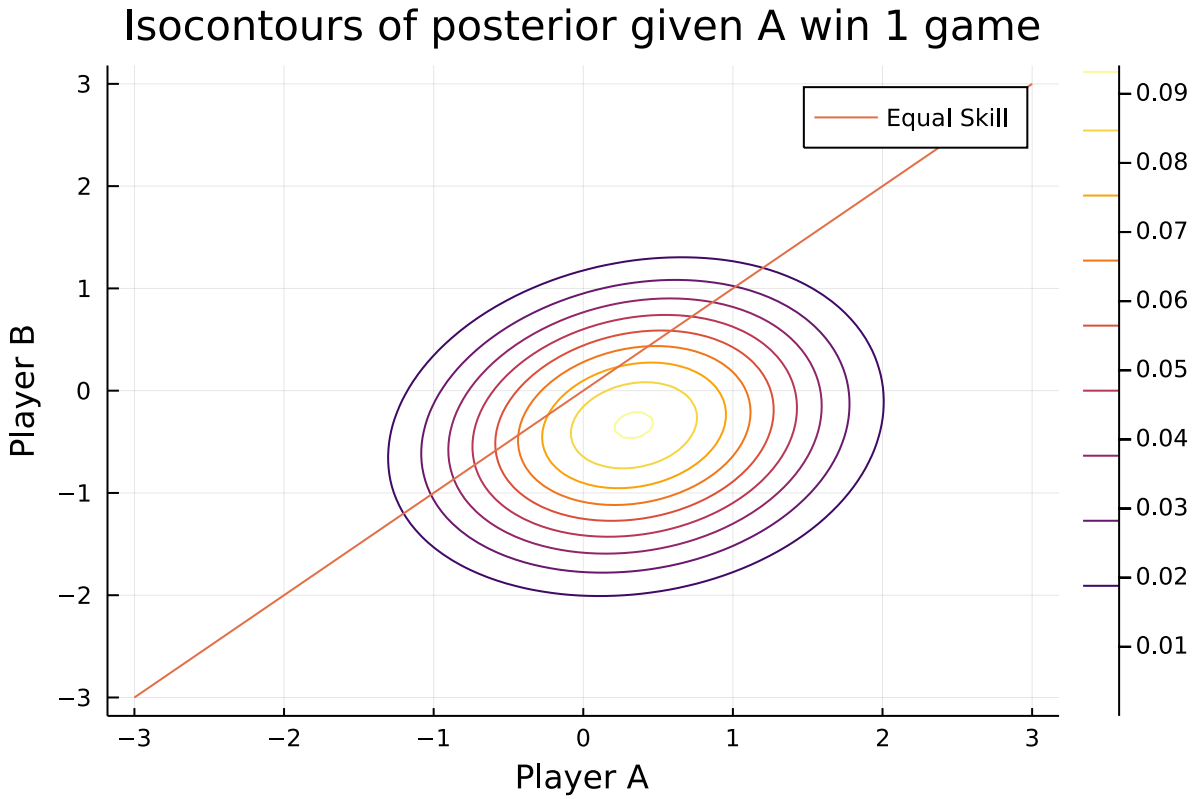
1. Isocontours of the prior distribution over players' skills.
2. Isocontours of the likelihood function.
3. Isocontours of the posterior over players' skills given the observaton: `player A beat player B in 1 game`.
4. Isocontours of the posterior over players' skills given the observation: `player A beat player B in 10 games`.
5. Isocontours of the posterior over players' skills given the observation: `20 games were played, player A beat player B in 10 games`.



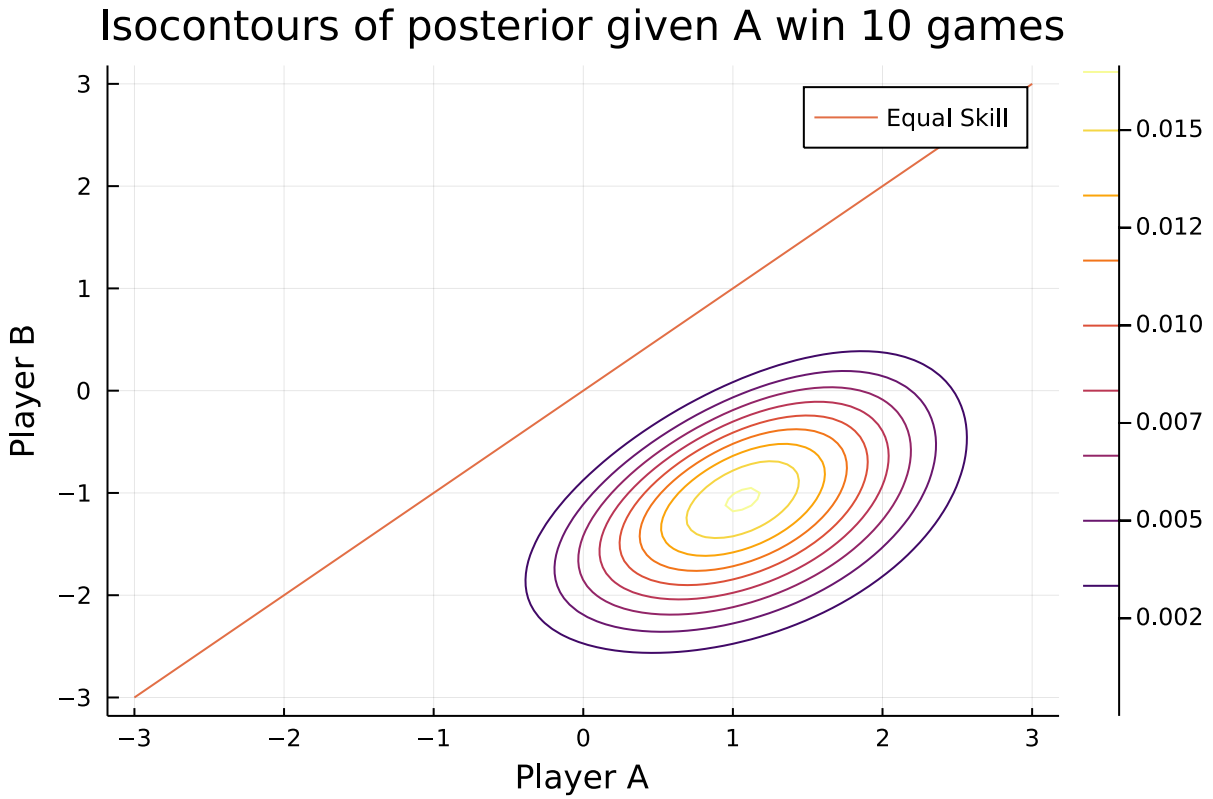

```
#1
# TODO: plot prior contours
begin
    plot() # Clear previous plot
    skillcontour!(zs->exp(log_prior(zs)))
    plot_line_equal_skill!()
    title!("Isocontours of prior distribution")
    xlabel!("Player A")
    ylabel!("Player B")
end
```



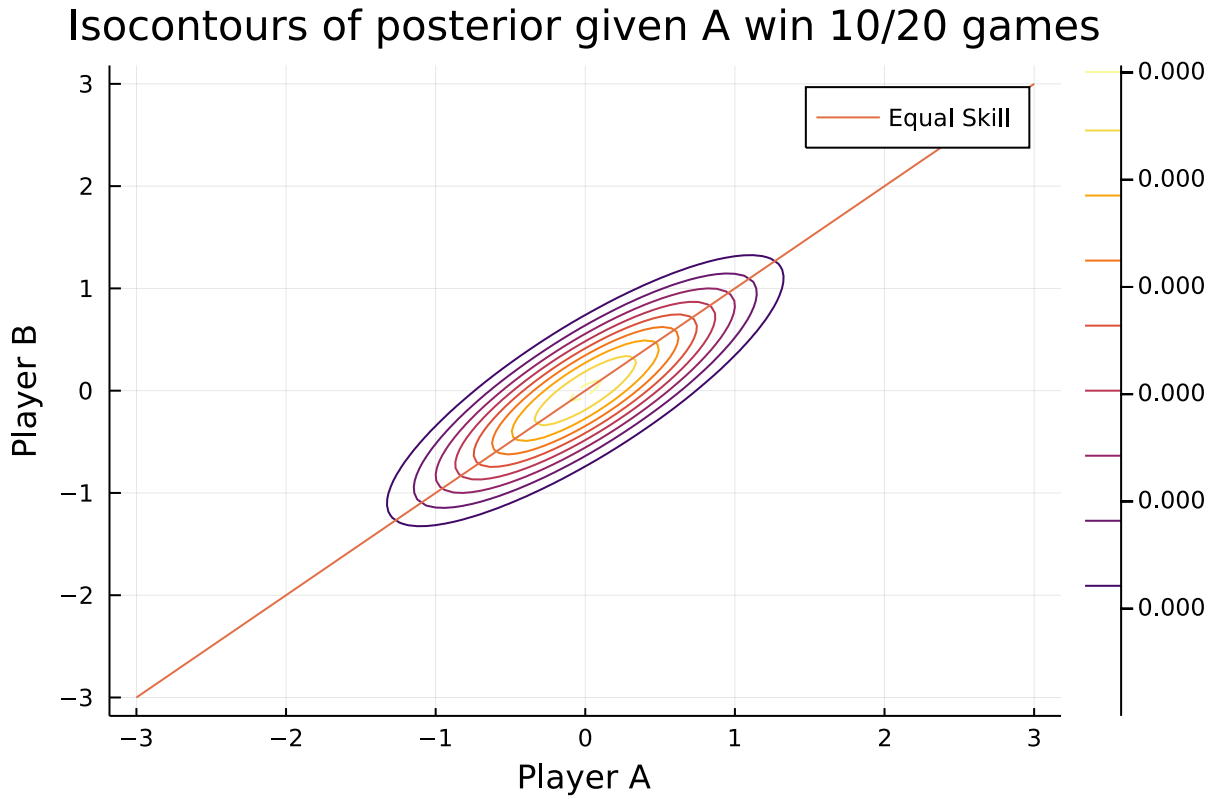

```
#2
# TODO: plot likelihood contours
begin
    plot() # Clear previous plot
    skillcontour!(zs -> exp.(logp_i_beats_j(zs[:,1],zs[:,2])))
    plot_line_equal_skill!()
    title!("Isocontours of likelihood")
    xlabel!("Player A")
    ylabel!("Player B")
end
```




```
#3
# TODO: plot posterior contours with player A winning 1 game
begin
    plot() # Clear previous plot
    skillcontour!(zs -> exp.(joint_log_density(zs, two_player_toy_games(1,0))))
    plot_line_equal_skill!()
    title!("Isocontours of posterior given A win 1 game")
    xlabel!("Player A")
    ylabel!("Player B")
end
```




```
#4
# TODO: plot joint contours with player A winning 10 games
begin
    plot() # Clear previous plot
    skillcontour!(zs -> exp.(joint_log_density(zs, two_player_toy_games(10, 0))))
    plot_line_equal_skill!()
    title!("Isocontours of posterior given A win 10 games")
    xlabel!("Player A")
    ylabel!("Player B")
end
```




```
#5
#TODO: plot joint contours with player A winning 10 games and player B winning 10 games
begin
    plot() # Clear previous plot
    skillcontour!(zs -> exp.(joint_log_density(zs, two_player_toy_games(10,10))))
    plot_line_equal_skill!()
    title!("Isocontours of posterior given A win 10/20 games")
    xlabel!("Player A")
    ylabel!("Player B")
end
```

4. Stochastic Variational Inference with Automatic Differentiation

A nice quality of our Bayesian approach is that we separate the model specification from the (approximate) method we use for inference.

In the original TrueSkill paper they described a message passing strategy for approximate inference. In Carl Rasmussen's assignment the students implement Gibbs sampling, a kind of Markov Chain Monte Carlo method.

We will use gradient-based stochastic variational inference (SVI), a recent technique that is extremely successful in our domain.

We will use SVI to produce an approximate posterior distribution to the, possibly complicated, posterior distributions specified by our model.

1. Implement the function `elbo` which computes an unbiased estimate of the Evidence Lower BOund. The ELBO is proportional to a KL divergence between the model posterior $p(z \mid data)$ and the approximate variational distribution $q_\phi(z \mid data)$.
2. Implement an objective function to optimize the parameters of the variational distribution. We will minimize a "loss" with gradient descent, the negative ELBO esitmated with 100 samples.
3. Write an optimization procedure `learn_and_vis_toy_variational_approx` that takes initial variational parameters and observed data and performs a number of iterations of gradient optimization where each iteration:
    1. compute the gradient of the loss with respect to the variational parameters using AD
    2. update the variational parameters taking a `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 one plot axis plot the target posterior contours in red and the variational approximation contours in blue. `display` the plot or save to a folder.

batchwise_gaussian_loglikelihood (generic function with 1 method)

```
function batchwise_gaussian_loglikelihood(params, samples)
    x = samples
    μ = params[1]
    σ = exp.(params[2])
    return sum(log.(1 ./ (2 * π * σ .^ 2) .^ 0.5) .+ (-0.5 * ((x .- μ) ./ σ) .^ 2), dims=1)'
end
```

elbo (generic function with 1 method)

```
function elbo(params,logp,num_samples)
    #TODO
    #size(samples) = N × batch size
    samples = params[1] .+ exp.(params[2]) .* randn((size(params[1])[1], num_samples))
    #TODO
    logp_estimate = logp(samples')
    #TODO
    logq_estimate = batchwise_gaussian_loglikelihood(params, samples)
    #TODO: should return scalar (hint: average over batch)
    return sum(logp_estimate - logq_estimate) / num_samples
end
```

neg_elbo (generic function with 1 method)

```
# Conveinence function for taking gradients
function neg_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
```

```
using Zygote: gradient
```

learn_and_vis_toy_variational_approx (generic function with 1 method)

```
function learn_and_vis_toy_variational_approx(init_params, toy_evidence; num_itrs=200, lr= 1e-2, num_q_samples = 10)
    params_cur = init_params
    p_return = 0
    loss = 1e6
    for i in 1:num_itrs
        #TODO: gradients of variational objective with respect to parameters
        grad_params = gradient(dparams -> neg_elbo(dparams, games=toy_evidence, num_samples=num_q_samples), params_cur)
        #TODO: update paramters with lr-sized step in descending gradient
        params_cur = params_cur .- lr .* grad_params[1]
        #TODO: report the current elbbo during training
        loss = neg_elbo(params_cur, games=toy_evidence, num_samples=num_q_samples)
        @info "iter: $i / $num_itrs \t $loss"
        # TODO: plot true posterior in red and variational in blue
        # hint: call 'display' on final plot to make it display during training
        p = plot();
        title!("Variational Approximation @ iter $i");
        xlabel!("Player A");
        ylabel!("Player B");
        #TODO: skillcontour!(...,colour=:red) plot likelihood contours for target posterior
        display(skillcontour!(zs -> exp.(joint_log_density(zs,toy_evidence)), colour=:red));
        # plot_line_equal_skill()
        plot_line_equal_skill!();
        #TODO: display(skillcontour!(..., colour=:blue)) plot likelihood contours for variational posterior
        display(skillcontour!(zs -> exp.(batchwise_gaussian_loglikelihood(params_cur, zs')), colour=:blue));
        if i == num_itrs
            p_return = p
        end
    end
    return params_cur, p_return, loss
end
```

5. Visualizing SVI on Two Player Toy

For the following collections of observed toy data of two player games learn an variational distribution to approximate the model posterior.

Using the plots produced in the training SVI procedure, one plot per iteration, create an animation that shows the variational distribution converging to its final approximation. In your final report, please simply plot the the learned variational distribution in at the final iteration.

Each of these require its own set of initial variational parameters and variational optimization procedure. The plots should have contours for two distributions, model posterior (red) and variational approximation (blue). To allow useful visualization do **not** normalize the contours to the same scale.

1. Report the final loss and plot the posteriors for the oberservation: `player A beats player B in 1 game`
2. Report the final loss and plot the posteriors for the oberservation: `player A beats player B in 10 games`

3. Report the final loss and plot the posteriors for the oberservation: `player A wins 10 games player B wins 10 games`

```
toy_mu =  Float64[0.88522, 1.44322]
```

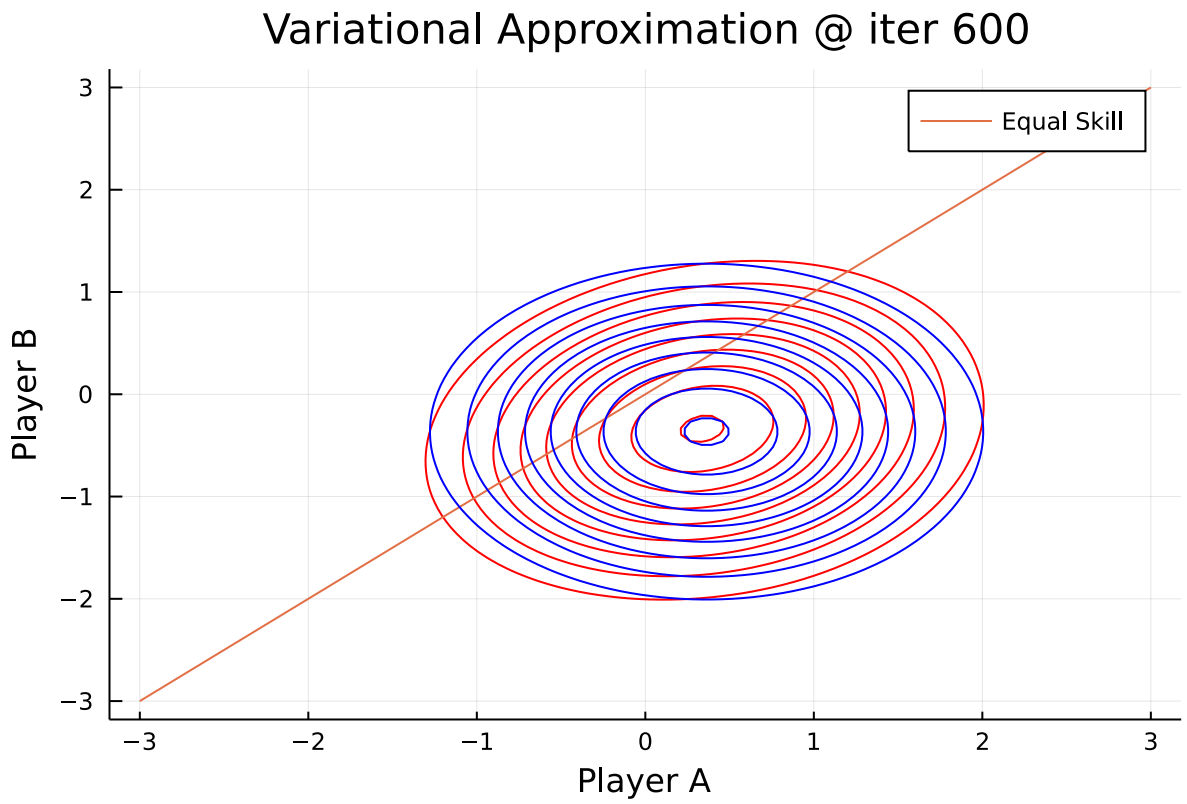
```
# Toy game
# toy_mu = [-2.,3.] # Initial mu, can initialize randomly!
toy_mu = randn(2)
```

```
toy_ls =  Float64[-0.585509, -0.0453019]
```

```
# toy_ls = [0.5,0.] # Initual log_sigma, can initialize randomly!
toy_ls = randn(2)
```

```
toy_params_init =  (Float64[0.88522, 1.44322], Float64[-0.585509, -0.0453019])
```

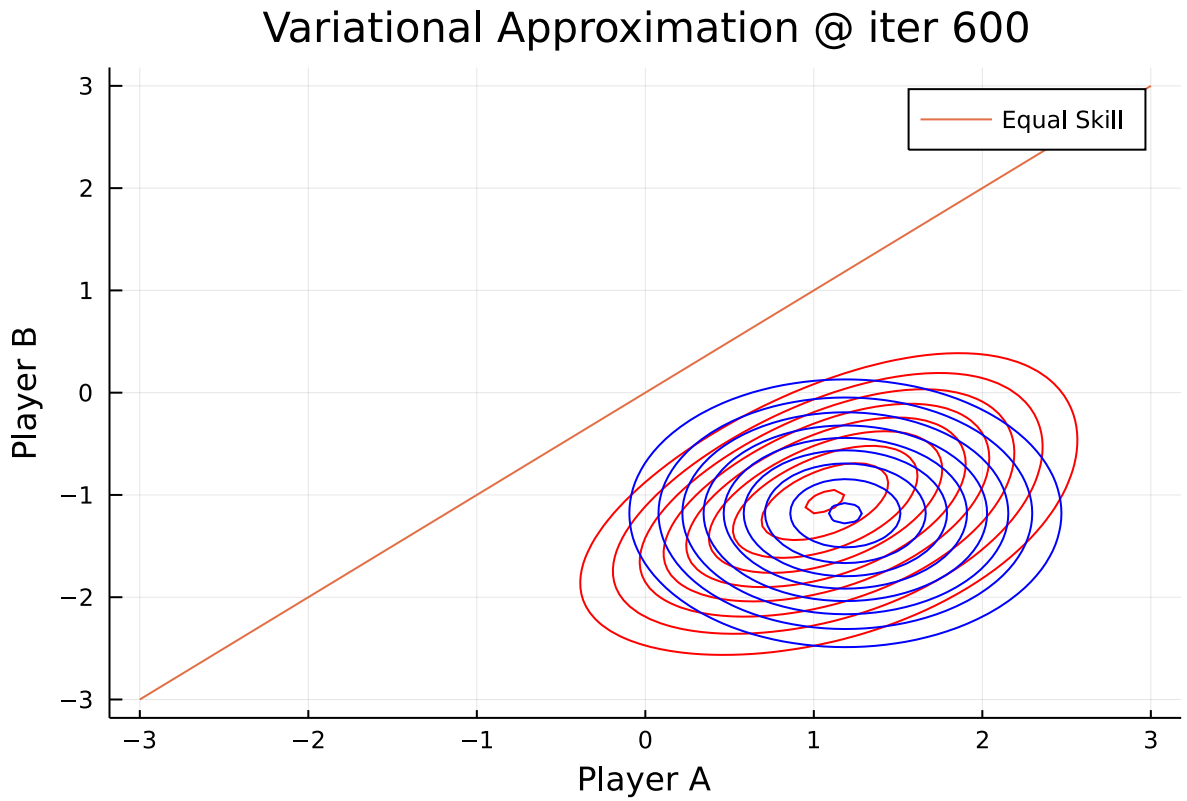
```
toy_params_init = (toy_mu, toy_ls)
```




```
begin
    #1
    #TODO: fit q with SVI observing player A winning 1 game
    params1, p1, loss1 = learn_and_vis_toy_variational_approx(toy_params_init, two_player_toy_games(1,0), num_itrs=600, num_q_samples=100)
    #TODO: save final posterior plots
    savefig("./plots/toy_svi_1.pdf")
    plot(p1)
end
```

A, B = (1,0)

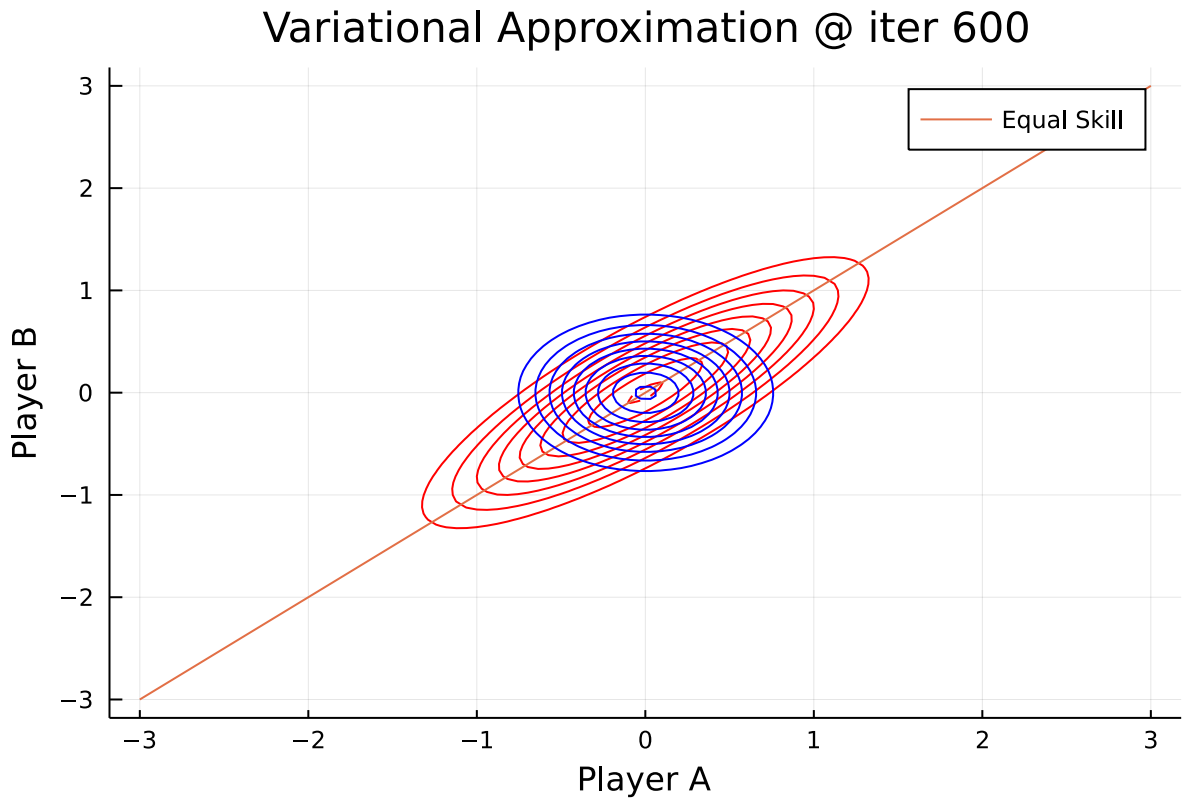
loss @ 600 iter = 0.6797202289919269




```
begin
    #2
    #TODO: fit q with SVI observing player A winning 10 games
    params2, p2, loss2 = learn_and_vis_toy_variational_approx(toy_params_init, two_player_toy_games(10,0), num_itrs=600, num_q_samples=100)
    #TODO: save final posterior plots
    savefig("./plots/toy_svi_10.pdf")
    plot(p2)
end
```

A, B = (10, 0)

loss @ 600 iter = 2.9186127656396916




```
begin
    #3
    #TODO: fit q with SVI observing player A winning 10 games and player B winning 10 games
    params3, p3, loss3 = learn_and_vis_toy_variational_approx(toy_params_init, two_player_toy_games(10,10), num_itrs=600, num_q_samples=100)
    #TODO: save final posterior plots
    savefig("./plots/toy_svi_10_20.pdf")
    plot(p3)
end
```

A, B = (10, 10)

loss @ 600 iter = 15.67333506972935

6. Approximate Inference on Real Data

Original Dataset

We will model the skills of players on chess.com from a collection of games played by Women GrandMasters since September 2009.

I have modified the dataset collected from the Kaggle Dataset: **Chess Games of Woman Grandmasters (2009 - 2021)**.

Note the extra information contained in the original dataset of game observations. If we were familiar with the rules of chess when we designed our model, and aware of this extra data collected in our

observations, we could describe a better model. For example,

- since White moves first, the White player has a considerable advantage
- not all games are observed as `win`,`lose`, some games end in draw due to repitition, or win by timeout.
- games have different timing rules.

Our model is simple and does not make use of that extra information in the original dataset. For this assignment it will not be necessary to modify our model beyond the simple case. If you are interested in extending this, consider how you could adapt the model specification to incorporate these observations.

Modified Dataset

My modifications to the data remove extra information about which player is White and simplifies the win conditions.

The data consists of two arrays:

- `games` is a $2 \times M$ collection of chess game outcomes, one game per column. The first row contains the indices of players who won, the second contains the indices of players who lost.
- `names` is a vector of player names, `names[i]` corresponds to the player chess.com username indicated by the values in `games`.

```
using CSV, DataFrames
```

```
games =
2×286889 Array{Int64,2}:
 1  3  2  5  6  8  7  10   7   7   7   7  15  …  43204  25115  2017  5193  2017  2017
 2  2  4  2  7  7  9   7  11  12  13  14   7      5193   5193  5193  2017  5193  5193
```

```
games = collect(Array(CSV.read("games.csv", DataFrame))')
```

```
names =
  String["getz-dal", "abrahamyan-la", "martirosov-bos", "sinanan-sea", "molner-arz", "r
```

```
names = vec(Array(CSV.read("names.csv", DataFrame)))
```

Variational Distribution

Use a fully-factorized Gaussian distribution for $q_\phi(z \mid data)$.

$$q_\phi(z \mid games) = \prod_i \mathcal{N}(z_i \mid \mu_i, \sigma_i)$$

Note, for numerical stability we work with the log_density. Also, for unconstrained optimization, we parameterize our Gaussian with $\log \sigma$, not variance or standard deviation.

You will need to implement this variational distribution with parameters $\phi = [\mu, \log \sigma]$ by defining `logq(z, params)` inside the `elbo`.

Using the model and SVI method, we will condition our posterior on observations from the dataset.

In the previous Two Player Toy we were informed about the players' skill by observing games between them. Now we have many players playing games among eachother. For any two players, i and j, answer yes or no to the following:

1. In general, is $p(z_i, z_j \mid \text{all games})$ proportional to $p(z_i, z_j, \text{all games})$?
2. In general, is $p(z_i, z_j \mid \text{all games})$ proportional to $p(z_i, z_j \mid \text{games between i and j})$? That is, do the games between player i and j provide all the information about the skills z_i and z_j?

Hint: consider the graphical model for three players, i, j, k and condition on results of games between all 3. Examine the conditional independeinces in the model. (graphical model is not required for marks)

3. write a new optimization procedure `learn_variational_approx` like the one from the previous question, but **does not produce any plots**. As well, have the optimization procedure save the loss (negative ELBO) for each iteration into an array. Initialize a variational distribution and optimize it to learn the model joint distribution observing the chess games in the dataset.
4. Plot the losses (negative ELBO estimate) obtained during variational optimization and report the loss of the final approximation.
5. We now have an approximate posterior over all our players' skill. Our variational family is simple, allowing us to easily compute an approximate mean and variance for all players' skill. Sort the players by mean skill and list the names of the 10 players with highest mean skill under our variational approximation. Use `sortperm`.
6. Plot the mean and variance of all players' skill, sorted by mean skill. There is no need to include the names of the players in the x-axis. Use `plot(means, yerror=exp.(logstds))` to add variance to the plot.

```
using Random
```

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

```
begin
    bs = 16384
    lr_decay_mb = 25
    lr_decay_fb = 40
    function learn_variational_approx(init_params, tennis_games; num_itrs=200, lr= 1e-2, num_q_samples = 10, mbsgd=true)
        params_cur = init_params
        losses = []
```

```
        tic_base = time()

        if mbsgd == true
            # for mini-batch SGD
            losses_b = []
            tennis_games = tennis_games[:, shuffle!(collect(1:size(tennis_games)[2]))]
            data_loader = [tennis_games[:, i:min(i + bs - 1, end)] for i in
1:bs:size(tennis_games)[2]]
            steps = size(data_loader)[1]
            @info "Start training with mini-batch SGD\n\tbs=$bs Σit=$num_itrs
Σst=$steps lr=$lr ld=$lr_decay_mb #q=$num_q_samples"
        else
            # for full-batch GD
            @info "Start training with full-batch GD\n\tΣit=$num_itrs lr=$lr
ld=$lr_decay_mb #q=$num_q_samples"
        end

        # training loop
        for i in 1:num_itrs
            # mini-batch SGD
            if mbsgd == true

                # lr scheduler
                if (i - 1) % lr_decay_mb == 0 && i != 1
                    if lr > 1e-4
                        @info "iter: $i\tlearning rate decays from $lr to $(lr * 0.2)"
                        lr *= 0.2
                    elseif lr > 1e-5
                        @info "iter: $i\tlearning rate decays from $lr to $(lr * 0.5)"
                        lr *= 0.5
                    end
                end

                loss_batch = 0
                # iter over data loader
                for (step, batch) in enumerate(data_loader)
                    tic = time()
                    #TODO: gradients of variational objective wrt params
                    grad_params = gradient(dparams -> neg_elbo(dparams, games=batch,
num_samples=num_q_samples), params_cur)
                    #TODO: update parmaeters wite lr-sized steps in desending gradient
direction
                    params_cur = params_cur .- lr .* grad_params[1]
                    #TODO: save objective value with current parameters
                    loss_mb = neg_elbo(params_cur, games=batch,
num_samples=num_q_samples)
                    loss_batch += loss_mb * size(batch)[2]
                    push!(losses, loss_mb)

                    toc = time()
                    eta = convert(Int, round((toc - tic_base) / ((i - 1) * steps +
step) * ((num_itrs - i) * steps + steps - step), digits=0))
                    @info "iter: $i / $num_itrs\tstep: $step / $steps\tloss:
$(round(loss_mb, digits=8))\ttime: $(round(toc - tic, digits=4)) s ETA: $eta s"
                end

                loss_b = loss_batch / size(tennis_games)[2]
                push!(losses_b, loss_b)
                @info "====================EPOCH SUMMARY==================\n\titer:
$i loss average over iteration: $(round(loss_b,
digits=8))\n=================================================="
            else
                # full-batch GD

                # lr scheduler
```

```
                if (i - 1) % lr_decay_fb == 0 && i != 1
                    if lr > 1e-4
                        @info "iter: $i\tlearning rate decays from $lr to $(lr * 0.2)"
                        lr *= 0.2
                    elseif lr > 1e-5
                        @info "iter: $i\tlearning rate decays from $lr to $(lr * 0.5)"
                        lr *= 0.5
                    end
                end

                tic = time()
                #TODO: gradients of variational objective wrt params
                grad_params = gradient(dparams -> neg_elbo(dparams, games=tennis_games, num_samples=num_q_samples), params_cur)
                #TODO: update parmaeters wite lr-sized steps in desending gradient direction
                params_cur = params_cur .- lr .* grad_params[1]
                #TODO: save objective value with current parameters
                log_loss = neg_elbo(params_cur, games=tennis_games, num_samples=num_q_samples)
                push!(losses, log_loss)
                toc = time()
                eta = convert(Int, round((toc - tic_base) / i * (num_itrs - i), digits=0))
                @info "iter: $i / $num_itrs\tloss: $(round(log_loss, digits=8))\ttime: $(round(toc - tic, digits=4)) s  ETA: $eta s"
            end
        end

        toc_base = time()
        @info "Total training time: $(round(toc_base - tic_base, digits=4)) s"

        if mbsgd == true
            return params_cur, losses, losses_b
        else
            return params_cur, losses
        end
    end
end
```

```
#1 Yes or No?
#2 Yes or No?
```

#1 YES.

#2 NO.

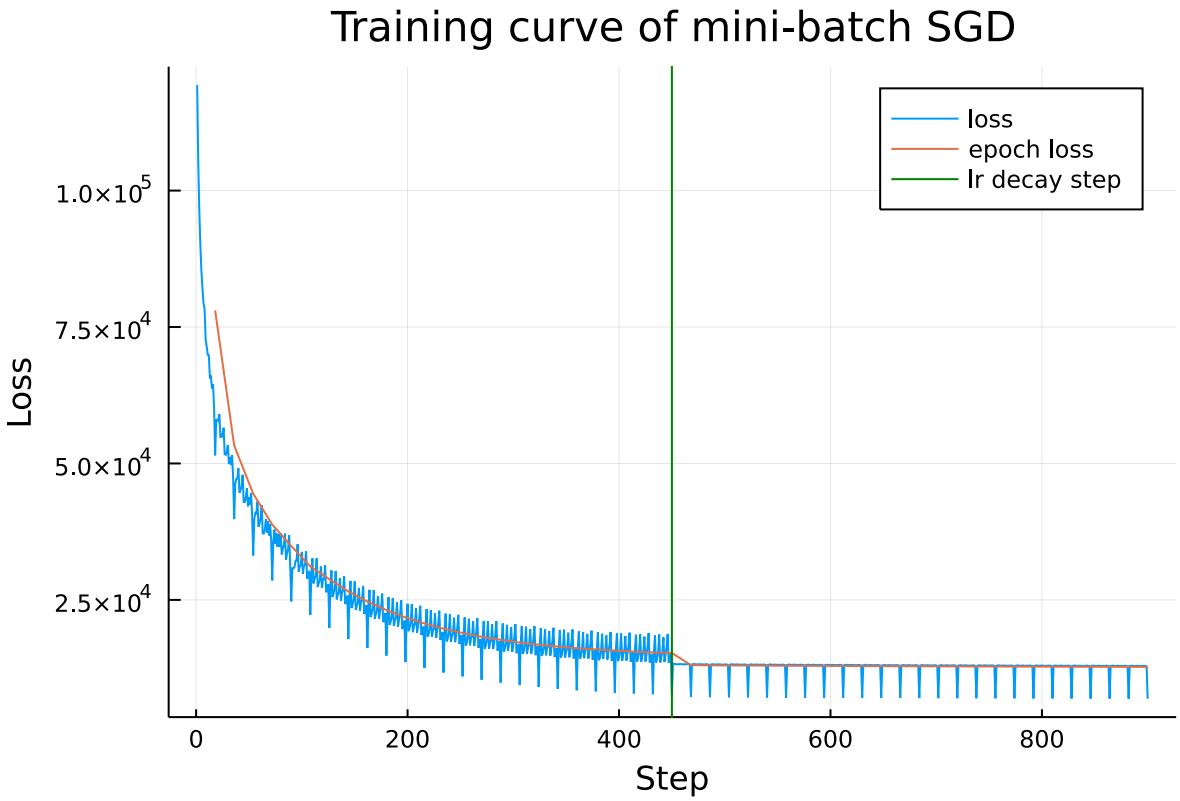
```
(Float64[0.517081, -0.633436, 0.934205, -0.453994, -2.02609, 0.176151, 0.636336, 0.0
```

```
begin
    #3
    # TODO: Initialize variational family
    init_mu = randn(size(names)[1])
    init_log_sigma = randn(size(names)[1])
    init_params = (init_mu, init_log_sigma) # random initialziation
end
```

Mini-batch SGD

```
((Float64[0.0607403,  -0.08495,  0.109076,  -0.0648334,  -0.112383,  0.0382971,  0.204682,
```

```
# mini-batch SGD
params, losses, losses_b = learn_variational_approx(init_params, games, lr=5e-3,
num_itrs=50, num_q_samples=100, mbsgd=true)
```




```
#4
#TODO plot losses during ELBO optimization and report final loss
begin
    plot(collect(1:size(losses)[1]), losses, label="loss")
    plot!(ceil(size(games)[2] / bs) * collect(1:size(losses_b)[1]), losses_b,
label="epoch loss")
    vline!([450], color=:green, label="lr decay step")
    title!("Training curve of mini-batch SGD")
    xlabel!("Step")
    ylabel!("Loss")
end
```

The final loss over the whole dataset (accumlated over mini-batches and averaged in an epoch): 12720.783948322794;

The final loss of the last mini-batch, whose size $\neq$ batch size: 6895.742670006433;

The final loss of the second last mini-batch, whose size $=$ batch size: 12836.176751549163;

```
10×2 Array{Any,2}:
 "liverpoolborn"       1.61739
 "sylvanaswindrunner"  1.09613
 "wonderfultime"       1.08132
 "aryantari"           1.05149
 "chessweebs"          1.02895
 "yileai"              0.864534
```

Remarks: Note that the all_games_log_density() implemented before sums over the likelihood of all observations. This means, different batch size gives all_games_log_density() in different magnitude, so as the joint_log_density(). I didn't normalize / scale up the loss per mini-batch since it requires reimplementing the joint_log_density(). Also, the current implementation just reports a loss in different magnitude and won't hurt the training. Therefore, the loss reported here may be different from other students who have done the scaling and normalization.

```
"volkovi"             0.853625
"atousa1"             0.84823
"kopeisk81"           0.834666
"nguyenxi"            0.83107
```

```
#5
#TODO: sort players by mean skill under our moidel and list top 10 players.
begin
    perm = sortperm(params[1], rev=true)
    cat(names[perm[1:10]], params[1][perm[1:10]], dims=2)
end
```

45

```
# ranks
findfirst(i -> i == "camillab", names[perm])
```

1

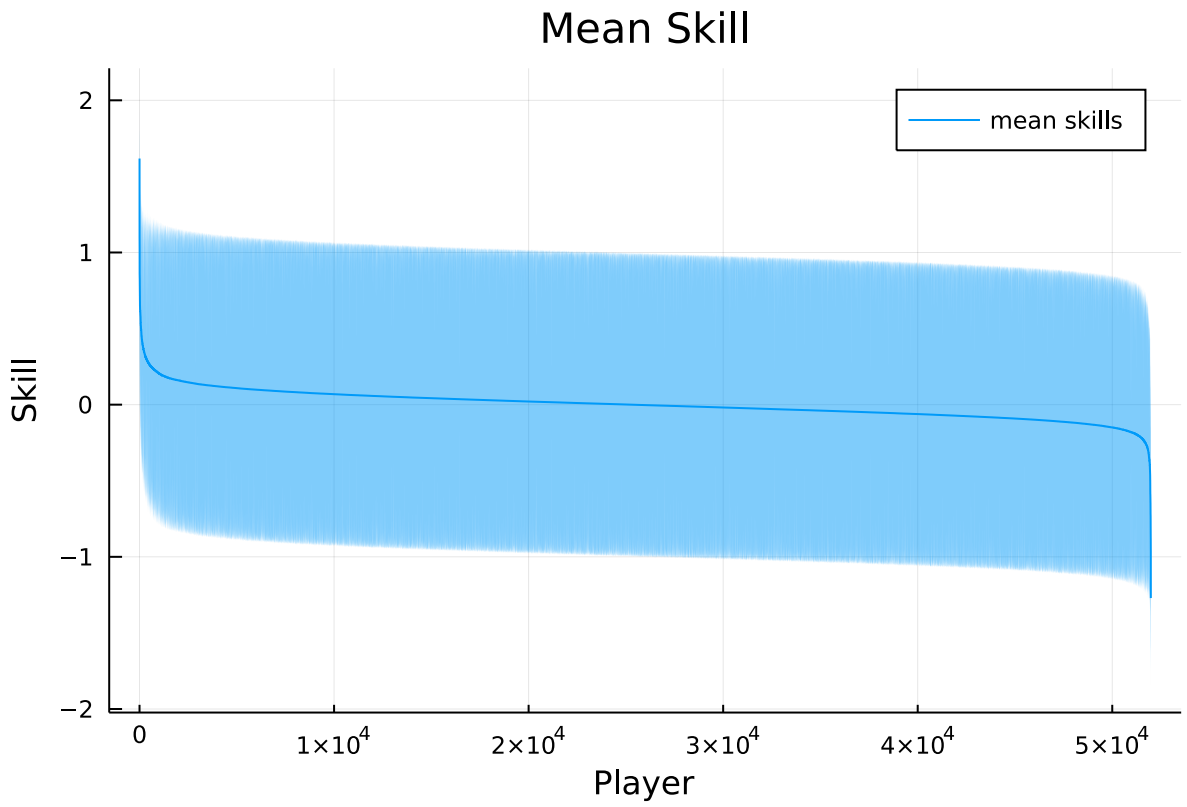
```
findfirst(i -> i == "liverpoolborn", names[perm])
```

227

```
findfirst(i -> i == "meri-arabidze", names[perm])
```

2

```
findfirst(i -> i == "sylvanaswindrunner", names[perm])
```




```
#6
#TODO: plot mean and variance of all players, sorted by skill
begin
    plot(params[1][perm], ribbon=exp.(params[2][perm]), label="mean skills")
    title!("Mean Skill")
    xlabel!("Player")
    ylabel!("Skill")
end
```

Full-batch GD (deprecated)

```
# full-batch GD
# params_fb, losses_fb = learn_variational_approx(init_params, games, lr=5e-4, num_itrs=240, num_q_samples=100, mbsgd=false)
```

```
#4
#TODO plot losses during ELBO optimization and report final loss
# begin
#   plot(collect(1:size(losses_fb)[1]), losses_fb, label="loss")
#   vline!([80,160], color=:red, label=false)
#   title!("Training curve of mini-batch SGD")
#   xlabel!("Step")
#   ylabel!("Loss")
#end
```

```
#5
#TODO: sort players by mean skill under our moidel and list top 10 players.
#begin
#   perm_fb = sortperm(params_fb[1], rev=true)
#   cat(names[perm_fb[1:10]], params_fb[1][perm_fb[1:10]], dims=2)
#end
```

```
# rank of liverpoolborn
#findfirst(i -> i == "liverpoolborn", names[perm_fb])
```

```
#6
#TODO: plot mean and variance of all players, sorted by skill
#begin
#   plot(params_fb[1][perm_fb], ribbon=exp.(params_fb[2][perm_fb]), label="mean skills")
#   title!("Mean Skill")
#   xlabel!("Player")
#   ylabel!("Skill")
#end
```

7. More Approximate Inference with our Model

Some facts from **The Queens of Chess Kaggle Notebook** motivate a couple inference questions.

- `camillab` Camilla Baginskaite is a Lithuanian and American WGM and is the most active player in the dataset, over 60K games played in the last 12 years!
- `liverpoolborn` Sheila Jackson is an English WGM and the most successful player (highest win %).
- `meri_arabidze` Meri Arabidze has achieved the highest rating by a WGM of 2763 on chess.com.
- `sylvanaswindrunner` Jovana Rapport is currently rated 2712, very close to Meri's all time high!

Here I find these players in the `names` data to determine their indices.

```
camillab = 4832
```

```
camillab = findfirst(i -> i == "camillab", names)
```

```
60655
```

```
length(findall(g -> g== 4832, vec(games))) # number of games camillab played!
```

```
liverpoolborn = 41410
```

```
liverpoolborn = findfirst(i -> i == "liverpoolborn", names)
```

```
meri_arabidze = 20805
```

```
meri_arabidze = findfirst(i -> i == "meri-arabidze", names)
```

```
sylvanaswindrunner = 37599
```

```
sylvanaswindrunner = findfirst(i -> i == "sylvanaswindrunner", names)
```

Use the optimized variational approximation to give estimates for the following inference problem:

What is the probability player A is better than player B?

Recall from our early plots, this is the mass under the distribution in the region above the line of equal skill, $z_A > z_B$.

We have chosen a very simple variational family, factorized Gaussians. This would allow us to derive the exact expression for the probability under the variational approximation that player A has higher skill than player B. This year we will not derive this value exactly.

We can estimate the answer to our inference question by sampling $N = 10000$ samples from the variational distribution and using a Simple Monte Carlo estimate.

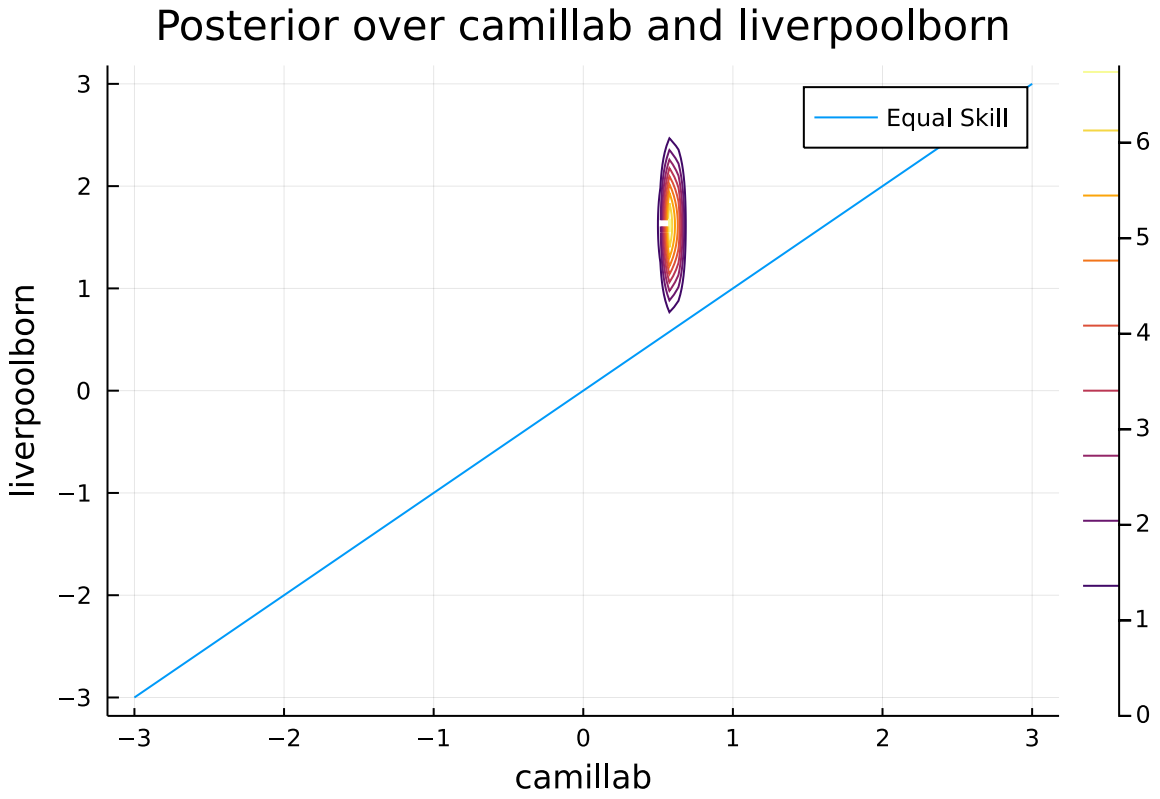
This inference question restricts us back to a two-player scenario. We know from previous questions that we can plot contours for distributions over 2 player skills.

most games vs most successful

1. Plot the **approximate** posterior over the skills of `camillab` and `liverpoolborn`.
2. Estimate the probability that `liverpoolborn` is more skillful than `camillab`.

all time high vs contender

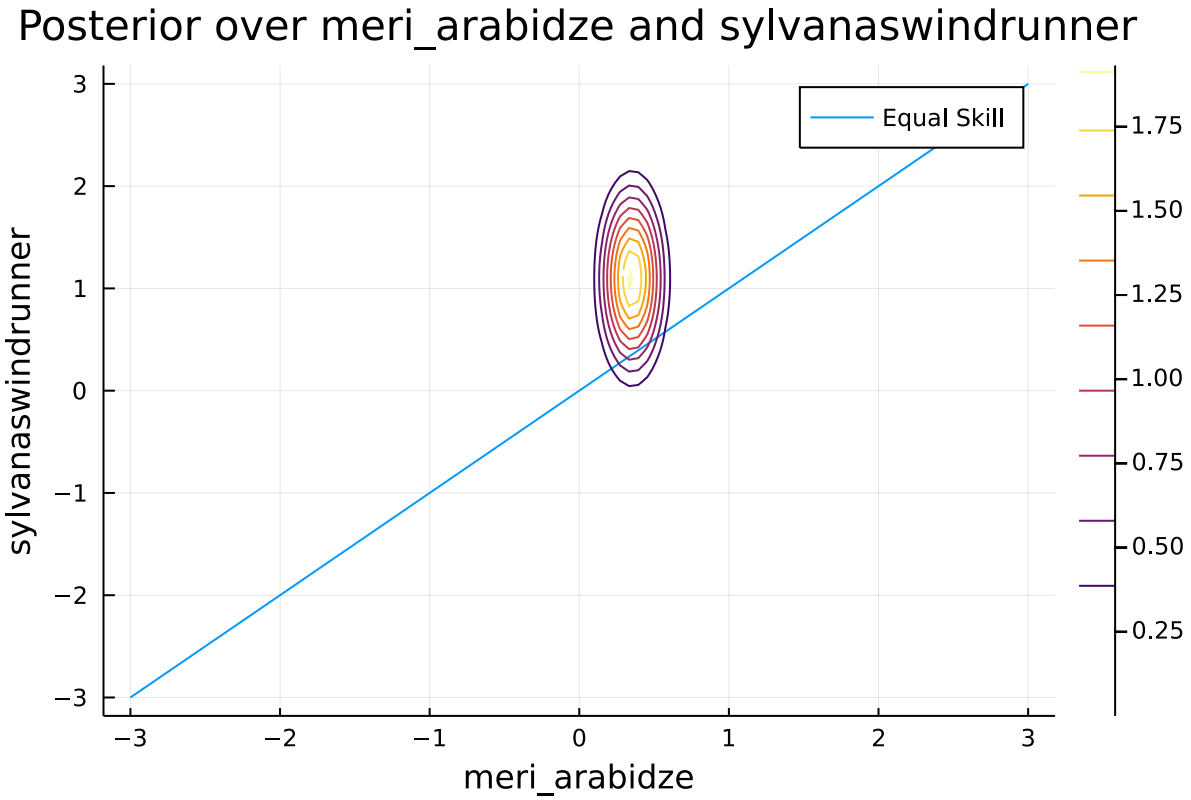
3. Plot the **approximate** posterior over the skills of `meri_arabidze` and `sylvanaswindrunner`.
4. Estimate the probability that `sylvanaswindrunner` is more skillful than `meri_arabidze`.




```
#1
#TODO: plot approximate posterior over two players
begin
    plot();
    μ_cl = [params[1][camillab], params[1][liverpoolborn]]
    log_σ_cl = [params[2][camillab], params[2][liverpoolborn]]
    plot_line_equal_skill!();
    skillcontour!(zs -> exp.(batchwise_gaussian_loglikelihood((μ_cl, log_σ_cl), zs')))
    title!("Posterior over camillab and liverpoolborn")
    xlabel!("camillab")
    ylabel!("liverpoolborn")
end
```

```
0.984
```

```
#2
# TODO: use Simple Monte Carlo to estimate probability player A is better
begin
    N = 10000
    z_camilab = randn(N) * exp(log_σ_cl[1]) .+ μ_cl[1]
    z_liverpoolborn = randn(N) * exp(log_σ_cl[2]) .+ μ_cl[2]
    count(z_camilab .< z_liverpoolborn) / N
end
```




```
#3
#TODO: plot approximate posterior over two players
begin
    plot();
    μ_ms = [params[1][meri_arabidze], params[1][sylvanaswindrunner]]
    log_σ_ms = [params[2][meri_arabidze], params[2][sylvanaswindrunner]]
    plot_line_equal_skill!();
    skillcontour!(zs -> exp.(batchwise_gaussian_loglikelihood((μ_ms, log_σ_ms), zs')))
    title!("Posterior over meri_arabidze and sylvanaswindrunner")
    xlabel!("meri_arabidze")
    ylabel!("sylvanaswindrunner")
end
```

```
0.8886
```

```
#4
# TODO: use Simple Monte Carlo to estimate probability player A is better
begin
    z_meri_arabidze = randn(N) * exp(log_σ_ms[1]) .+ μ_ms[1]
    z_sylvanaswindrunner = randn(N) * exp(log_σ_ms[2]) .+ μ_ms[2]
    count(z_meri_arabidze .< z_sylvanaswindrunner) / N
end
```