A2

March 25, 2021

1 Assignment 2

Name: Mohammadreza Salehi Student#: 1006890081

[1]:
```
from math import pi
import jax.numpy as jnp
from itertools import product
from matplotlib import pyplot as plt
from scipy.stats import norm
from jax.scipy.special import logsumexp
import numpy as np
from jax import grad
from PIL import Image
import pandas as pd
from matplotlib.animation import FuncAnimation
```

1.1 2. Implementing the Model

[2]:
```
def log1pexp(x):
    """
    x: ndarray with shape (d1, d2, ..., dk)

    returns log1pexp of x (elementwise)

    As I didn't find an equivalent for log1pexp in python, I used logsumexp to␣
↪implement it.
    Note that log1pexp(x) = log(exp(0) + exp(x)) = logsumexp([0, x])
    """
    zeros = jnp.zeros_like(x)
    z = jnp.concatenate((x[..., jnp.newaxis], zeros[..., jnp.newaxis]), axis=-1)

    return logsumexp(z, axis=-1)
```

[3]:
```
def log_prior(zs):
    """
    zs: ndarray with shape (K, N)
```

```
    """
    return -0.5 * jnp.sum(jnp.log(2 * pi) + zs ** 2, axis=1, keepdims=True)  #␣
↪shape: (K, 1)


def logp_i_beats_j(zi, zj):
    return -log1pexp(jnp.array(zj - zi)).item()


def all_games_log_likelihood(zs, games):
    """
    zs: ndarray with shape (K, N)
    games: ndarray with shape (M, 2)
    """
    zs_a = zs[:, games[:, 0]]  # skills of winners with shape: (K, M)
    zs_b = zs[:, games[:, 1]]  # skills of losers with shape: (K, M)

    likelihoods = jnp.sum(-log1pexp(zs_b - zs_a), axis=1, keepdims=True)  #␣
↪shape: (K, 1)

    return likelihoods


def joint_log_density(zs, games):
    """
    zs: ndarray with shape (K, N)
    games: ndarray with shape (M, 2)
    """
    return log_prior(zs) + all_games_log_likelihood(zs, games)  # shape: (K, 1)
```

1.2 3. Visualize the Model on Toy Data

1.2.1 3.1. Toy Data

[4]:
```
def two_player_toy_games(p1_wins, p2_wins):
    return jnp.vstack((jnp.array([[0, 1]]).repeat(p1_wins, axis=0),
                       jnp.array([[1, 0]]).repeat(p2_wins, axis=0)))
```

[5]:
```
two_player_toy_games(5, 3)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0
and rerun for more info.)
```

[5]:
```
DeviceArray([[0, 1],
             [0, 1],
             [0, 1],
```

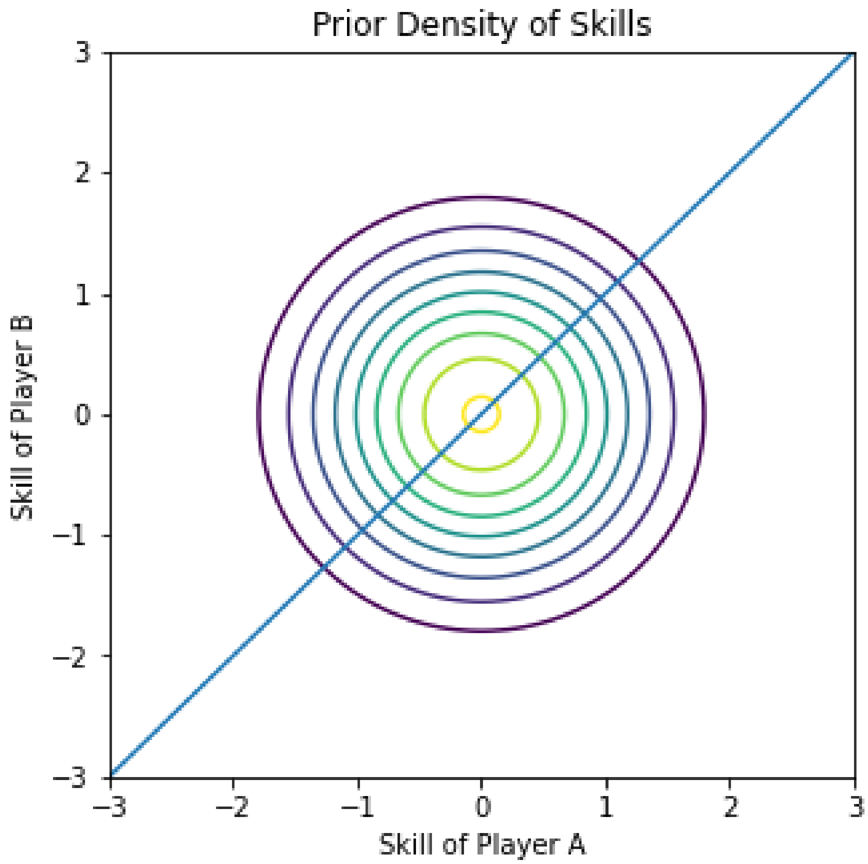
```
        [0, 1],
        [0, 1],
        [1, 0],
        [1, 0],
        [1, 0]], dtype=int32)
```

1.2.2 3.2. 2D Posterior Visualization

[6]:
```
def skill_countour(f, colour=None):
    n = 100
    x = jnp.linspace(-3, 3, num=n).tolist()
    y = jnp.linspace(-3, 3, num=n).tolist()
    z_grid = jnp.array(list(product(x, y)))  # shape: (n**2, 2)
    z = f(z_grid)  # shape: (n**2, 1)
    z = z[:,0]  # shape: (n**2,)
    max_z = max(z)
    levels = [level * max_z for level in [0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.
 ↪9, 0.99]]

    if colour is None:
        p1 = plt.contour(x, y, z.reshape(n, n).T, levels=levels)
    else:
        p1 = plt.contour(x, y, z.reshape(n, n).T, colors=colour, levels=levels)

def plot_line_equal_skill():
    plt.plot(jnp.linspace(-3, 3, num=200), jnp.linspace(-3, 3, num=200),␣
 ↪label='Equal skill')
```

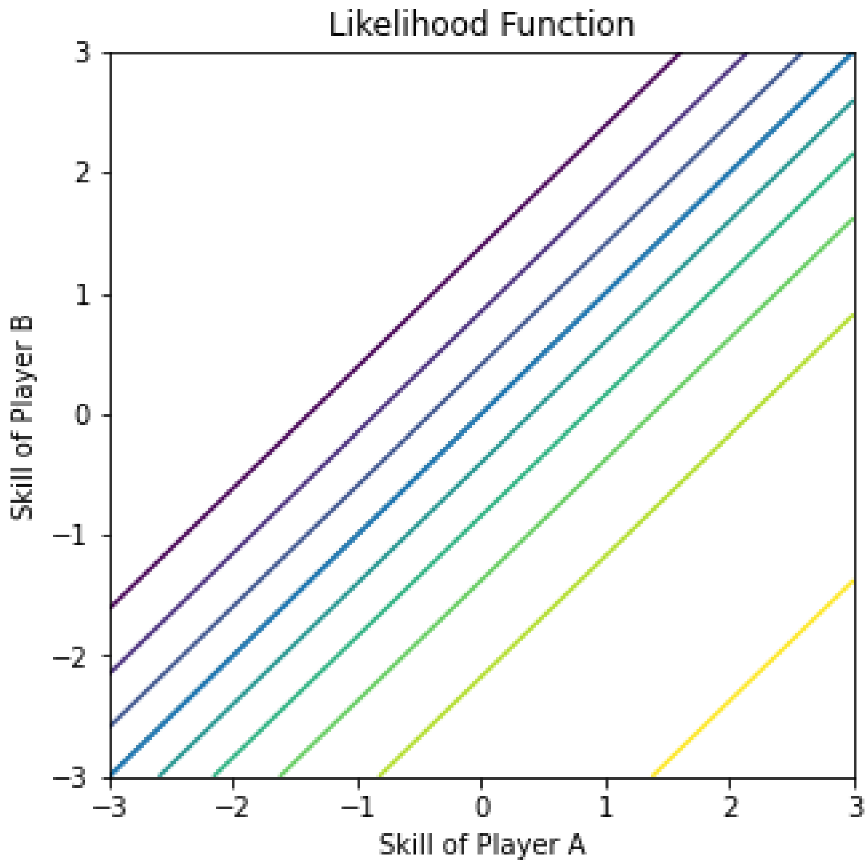
1. Isocontours of the prior distribution over players' skills:

[7]:
```
f = lambda zs: jnp.exp(log_prior(zs))
fig = plt.figure(figsize=(5, 5))
plot_line_equal_skill()
plt.title('Prior Density of Skills')
plt.xlabel('Skill of Player A')
plt.ylabel('Skill of Player B')
skill_countour(f);
```



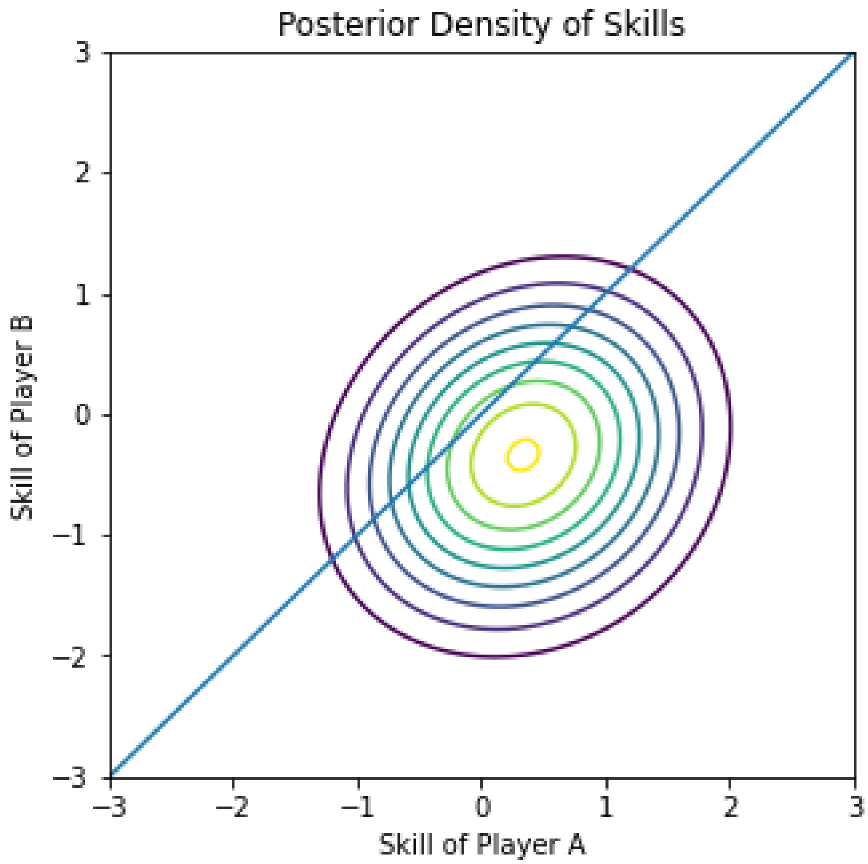
2. Isocontours of the likelihood function:

[450]:
```
f = lambda zs: jnp.exp(jnp.array([[logp_i_beats_j(el[0], el[1])] for el in zs]))
fig = plt.figure(figsize=(5, 5))
plot_line_equal_skill()
plt.title('Likelihood Function')
plt.xlabel('Skill of Player A')
plt.ylabel('Skill of Player B')
skill_countour(f)
```



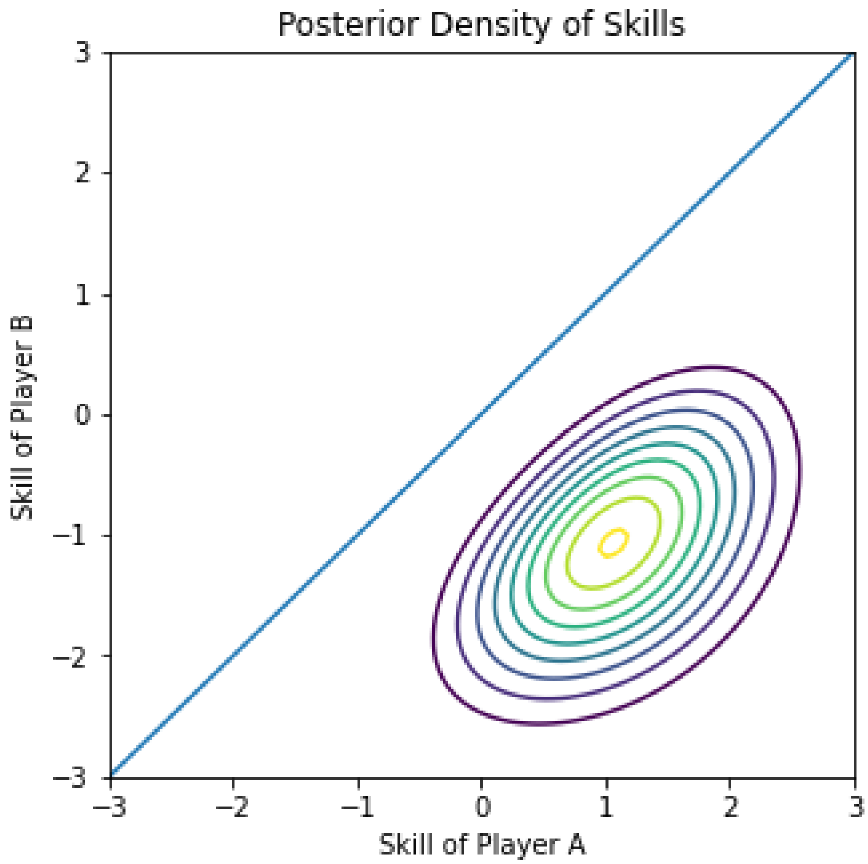
3. Isocontours of the posterior over players' skills given the observaton: player A beat player B in 1 game.

[8]:
```
def f(zs):
    games = two_player_toy_games(1,0)
    return jnp.exp(joint_log_density(zs, games))
fig = plt.figure(figsize=(5, 5))
plot_line_equal_skill()
plt.title('Posterior Density of Skills')
plt.xlabel('Skill of Player A')
plt.ylabel('Skill of Player B')
skill_countour(f)
```



4. Isocontours of the posterior over players' skills given the observation: player A beat player B in 10 games .

[9]:
```
def f(zs):
    games = two_player_toy_games(10,0)
    return jnp.exp(joint_log_density(zs, games))
fig = plt.figure(figsize=(5, 5))
plot_line_equal_skill()
plt.title('Posterior Density of Skills')
plt.xlabel('Skill of Player A')
plt.ylabel('Skill of Player B')
skill_countour(f)
```



5. Isocontours of the posterior over players' skills given the observation: 20 games were played, player A beat player B in 10 games .

[10]:
```
def f(zs):
    games = two_player_toy_games(10,10)
    return jnp.exp(joint_log_density(zs, games))
fig = plt.figure(figsize=(5, 5))
plot_line_equal_skill()
plt.title('Posterior Density of Skills')
plt.xlabel('Skill of Player A')
plt.ylabel('Skill of Player B')
skill_countour(f)
```

1.3 4. Stochastic Variational Inference with Automatic Differentiation

[28]:
```
def elbo(params, logp, num_samples):
    mu, ls = params['mu'], params['ls']  # mu & ls are arrays with shepe (N,)

    # reparametrization
    epsilon = jnp.array(np.random.randn(num_samples, mu.shape[0]))  # shape: (B,␣
↪N)
    samples = epsilon * ls + mu  # shape: (B, N)

    logp_estimate = jnp.mean(logp(samples))

    """
    q is a factorized Gaussian distribution. Note that logq is exactly equal to␣
↪the
    likelihoods of samples before reparametrization since the reparametrization␣
↪does
```

```
    not change the probabilities! Therefore we can compute the log likelihoos of␣
↪epsilons
    instead of samples.
    """
    logq_estimate = jnp.mean(jnp.sum(-0.5 * (jnp.log(2 * pi) + epsilon ** 2),␣
↪axis=1))

    return logp_estimate - logq_estimate


def neg_elbo(params, games=two_player_toy_games(1, 0), num_samples=100):
    def logp(zs):
        return joint_log_density(zs, games)

    return -elbo(params, logp, num_samples)
```

[29]:
```
def learn_and_vis_toy_variational_approx(init_params, toy_evidence,␣
↪num_iters=200, lr=1e-2, num_q_samples=10,
                                          print_every=10):

    params_list = []  # a list to save parameters which will be used to create␣
↪an animation
    losses = []
    grad_fn = grad(neg_elbo, argnums=0)
    params_cur = init_params

    def f_true_posterior(zs):
        return jnp.exp(all_games_log_likelihood(zs, toy_evidence) +␣
↪joint_log_density(zs, toy_evidence))

    def f_var(zs):
        mu, ls = params_cur['mu'], params_cur['ls']
        out = jnp.prod((1/((2*pi) ** 0.5 * ls)) * jnp.exp(-0.5 * ((zs - mu) /␣
↪ls) ** 2), axis=1, keepdims=True)
        return out

    for i in range(num_iters):
        grad_params = grad_fn(params_cur, toy_evidence, num_q_samples)
        neg_elbo_cur = neg_elbo(params_cur, toy_evidence, num_q_samples)
        losses.append(neg_elbo_cur)

        params_list.append(params_cur.copy())

        if i % print_every == 0:
            print(f'Iteration {i}, Loss: {neg_elbo_cur}')
```

```
        for p in params_cur:
            params_cur[p] -= grad_params[p] * lr

    skill_countour(f_true_posterior, colour='red');
    skill_countour(f_var, colour='blue');

    return params_list, losses
```

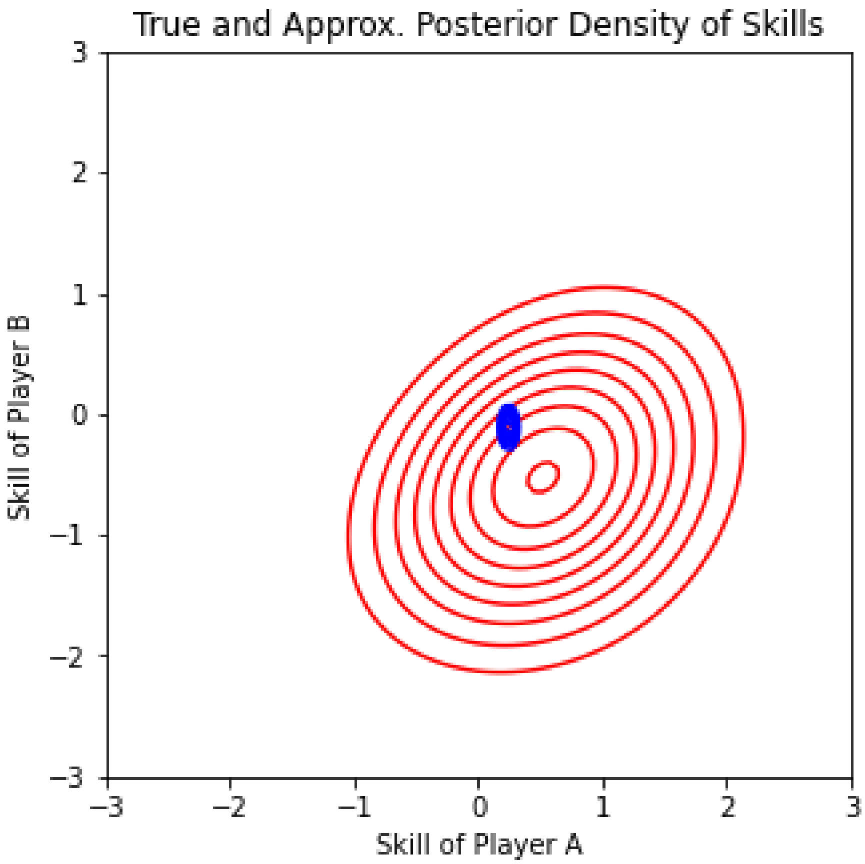
1.4 5. Visualizing SVI on Two Player Toy

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

[30]:
```
init_params = {'mu': jnp.array([-2.,3.]), 'ls': jnp.array([0.5, 1.])}
```

[31]:
```
toy_evidence = two_player_toy_games(1, 0)

fig = plt.figure(figsize=(5, 5));
params_list_1, losses = learn_and_vis_toy_variational_approx(init_params,
                                                             toy_evidence,
                                                             num_iters=200,
                                                             lr=1e-2,
                                                             num_q_samples=100,
                                                             print_every=20)
print('Final loss', losses[-1])
plt.title('True and Approx. Posterior Density of Skills')
plt.xlabel('Skill of Player A')
plt.ylabel('Skill of Player B');
```

```
Iteration 0, Loss: 11.34329891204834
Iteration 20, Loss: 6.605831146240234
Iteration 40, Loss: 4.025387763977051
Iteration 60, Loss: 2.288309335708618
Iteration 80, Loss: 1.1001198291778564
Iteration 100, Loss: 0.46373748779296875
Iteration 120, Loss: -0.06800341606140137
Iteration 140, Loss: -0.24183988571166992
Iteration 160, Loss: -0.24112820625305176
Iteration 180, Loss: -0.22714972496032715
Final loss -0.30454612
```

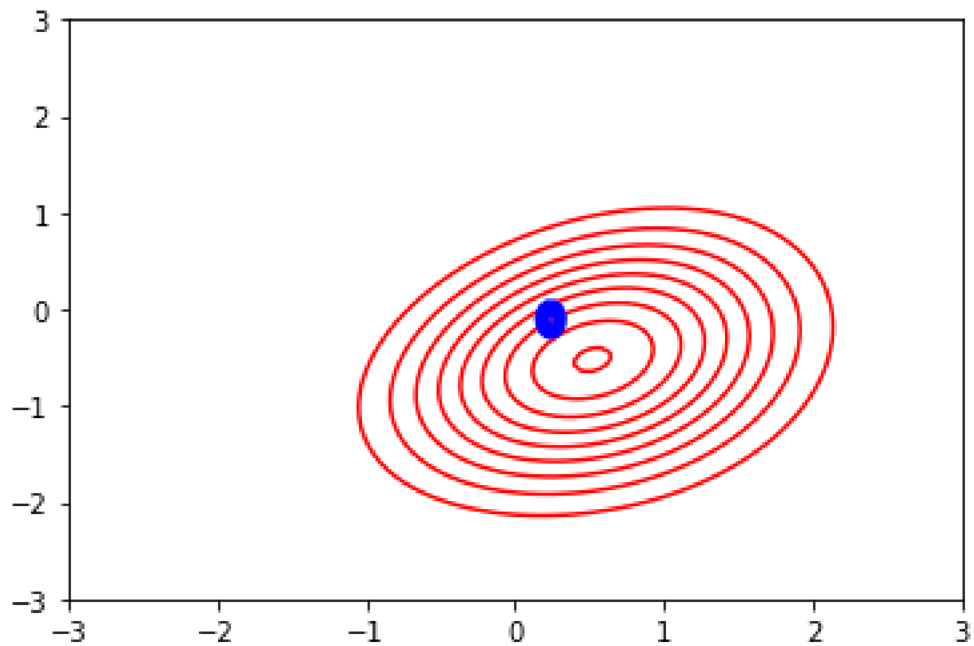


1.4.1 Animation

[21]:
```
CONT = None

def create_animation(params_list, fname='anim.gif'):
    global CONT
    n = 100
    x = jnp.linspace(-3, 3, num=n).tolist()
    y = jnp.linspace(-3, 3, num=n).tolist()
    z_grid = jnp.array(list(product(x, y)))

    def f_true_posterior(zs):
        return jnp.exp(all_games_log_likelihood(zs, toy_evidence) +␣
 ↪joint_log_density(zs, toy_evidence))

    def f_var(zs, i):
        mu, ls = params_list[i]['mu'], params_list[i]['ls']
```

```
        out = jnp.prod((1/((2*pi) ** 0.5 * ls)) * jnp.exp(-0.5 * ((zs - mu) /␣
↪ls) ** 2), axis=1, keepdims=True)
        return out

    fig, ax = plt.subplots()

    z_approx = f_var(z_grid, 0)
    z_approx = z_approx[:, 0]

    z_true = f_true_posterior(z_grid)
    z_true = z_true[:, 0]

    max_z_approx = max(z_approx)
    levels_approx = [level * max_z_approx for level in [0.2, 0.3, 0.4, 0.5, 0.6,␣
↪0.7, 0.8, 0.9, 0.99]]

    max_z_true = max(z_true)
    levels_true = [level * max_z_true for level in [0.2, 0.3, 0.4, 0.5, 0.6, 0.
↪7, 0.8, 0.9, 0.99]]

    ax.contour(x, y, z_true.reshape(n, n).T , colors='red', levels=levels_true)

    CONT = ax.contour(x, y, z_approx.reshape(n, n).T , colors='blue',␣
↪levels=levels_approx)

    def skill_contour_animate(i):
        global CONT
        z_approx = f_var(z_grid, i)
        z_approx = z_approx[:, 0]
        max_z = max(z_approx)
        levels = [level * max_z for level in [0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,␣
↪0.9, 0.99]]

        for coll in CONT.collections:
            coll.remove()

        CONT = ax.contour(x, y, z_approx.reshape(n, n).T , colors='blue',␣
↪levels=levels)

        return CONT


    anim = FuncAnimation(fig, skill_contour_animate, frames=200, interval=1)

    anim.save(fname);
```

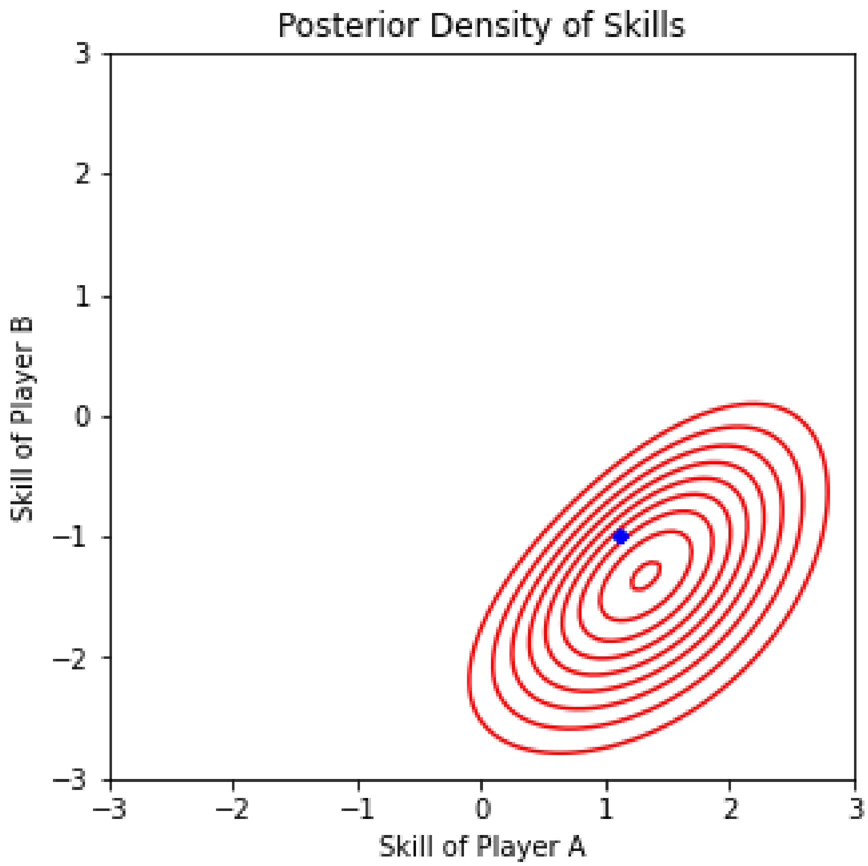
[22]:
```python
create_animation(params_list_1, 'anim1.gif')
```



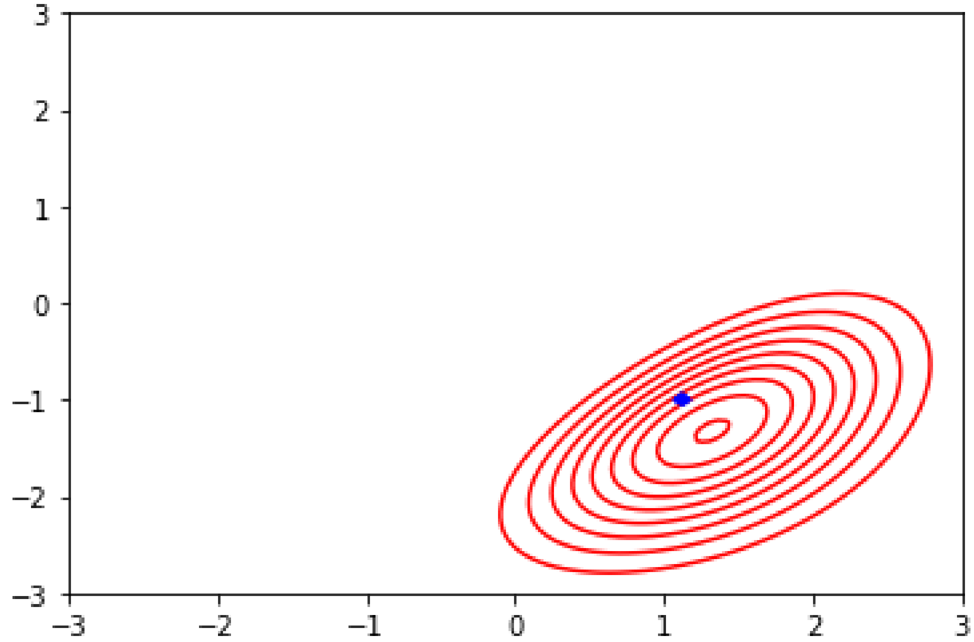
[34]:
```python
init_params = {'mu': jnp.array([-2.,3.]), 'ls': jnp.array([2, 2.3])}

toy_evidence = two_player_toy_games(10, 0)

fig = plt.figure(figsize=(5, 5));
params_list_2, losses = learn_and_vis_toy_variational_approx(init_params,␣
 ↪toy_evidence, num_iters=200,
                                                             lr=1e-2,␣
 ↪num_q_samples=100, print_every=20)

print('Final loss', losses[-1])
plt.title('Posterior Density of Skills')
plt.xlabel('Skill of Player A')
plt.ylabel('Skill of Player B');
```

```
Iteration 0, Loss: 60.0381965637207
Iteration 20, Loss: 18.666484832763672
Iteration 40, Loss: 4.732278823852539
Iteration 60, Loss: 2.2469639778137207
Iteration 80, Loss: 1.5858526229858398
Iteration 100, Loss: 1.392362356185913
Iteration 120, Loss: 1.0595269203186035
Iteration 140, Loss: 1.2278976440429688
Iteration 160, Loss: 1.3229739665985107
```

```
Iteration 180, Loss: 1.3807885646820068
Final loss 1.334738
```

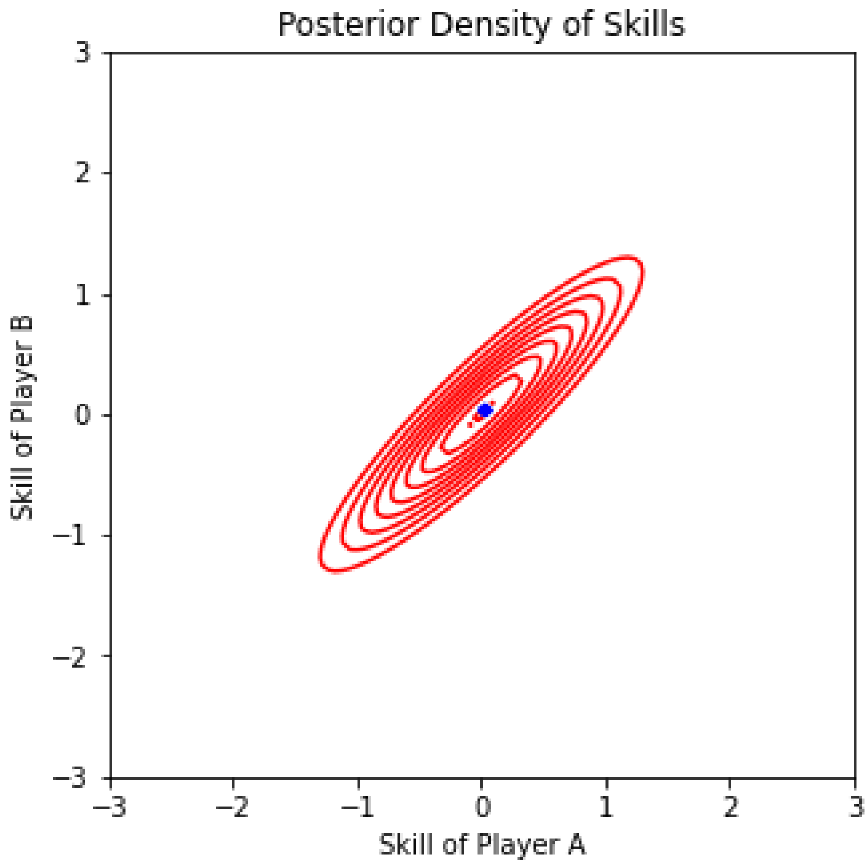


[24]:
```
create_animation(params_list_2, 'anim2.gif')
```

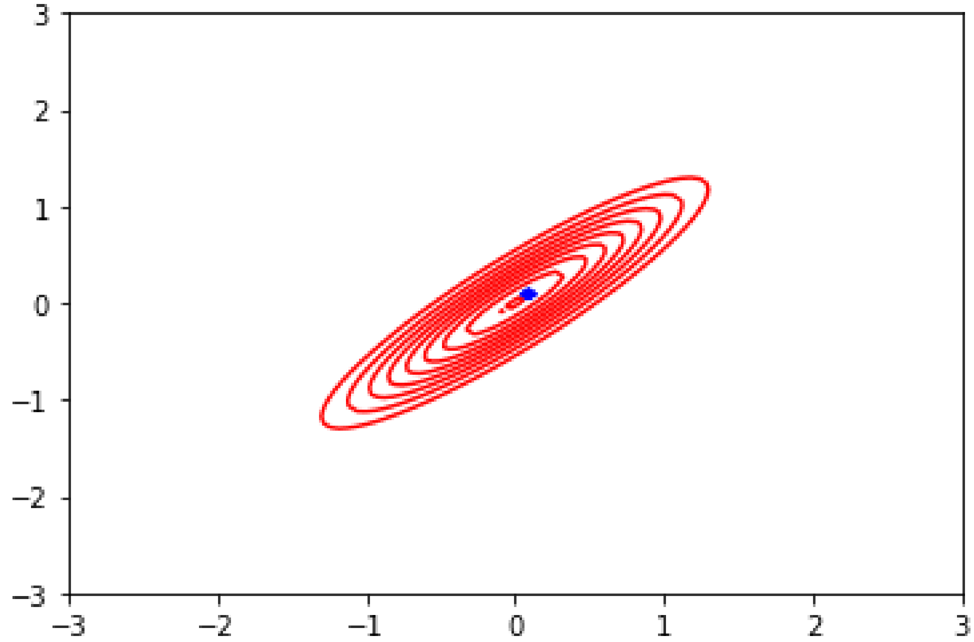


[35]:
```
init_params = {'mu': jnp.array([-2.,3.]), 'ls': jnp.array([20., 11.])}

toy_evidence = two_player_toy_games(10, 10)

fig = plt.figure(figsize=(5, 5));

params_list_3, losses = learn_and_vis_toy_variational_approx(init_params,␣
↪toy_evidence, num_iters=200, lr=1e-2, num_q_samples=100,
                                    print_every=20)

print('Final loss', losses[-1])
plt.title('Posterior Density of Skills')
plt.xlabel('Skill of Player A')
plt.ylabel('Skill of Player B');
```

```
Iteration 0, Loss: 428.0752868652344
Iteration 20, Loss: 301.5541076660156
Iteration 40, Loss: 176.51268005371094
Iteration 60, Loss: 142.41168212890625
Iteration 80, Loss: 64.97075653076172
Iteration 100, Loss: 37.80109786987305
Iteration 120, Loss: 21.494976043701172
Iteration 140, Loss: 14.050697326660156
Iteration 160, Loss: 13.077592849731445
Iteration 180, Loss: 12.945093154907227
Final loss 12.758126
```



[26]:
```
create_animation(params_list_3, 'anim3.gif')
```



1.5 6. Approximate inference on real data

1.5.1 Reading the data

[39]:
```
df_games = pd.read_csv('games.csv')
df_names = pd.read_csv('names.csv')

games = df_games.values
names = df_names['name'].values.tolist()

games.shape, len(names)
```

[39]: ((286889, 2), 51983)

1.5.2 Dataloader

[40]:
```
from math import ceil

def dataloader(data, batch_size, shuffle=True):
    if shuffle:
        np.random.shuffle(data)

    num_iters = ceil(data.shape[0] / batch_size)
```

```
    for i in range(num_iters):
        yield data[i * batch_size:(i+1) * batch_size]
```

1.5.3 Variational Distribution

1. In general is $p(z_i, z_j|\text{all games})$ proportional to $p(z_i, z_j, \text{all games})$? Yes
2. In general is $p(z_i, z_j|\text{all games})$ proportional to $p(z_i, z_j, \text{games between i and j})$? That is do the games between player i and j provide all the information about the skills z_i and z_j? No.

[41]:

```
def elbo(params, logp, num_samples):
    mu, log_ls = params['mu'], params['log_ls']  # mu & sigma are arrays with␣
↪shepe (N,)

    epsilon = jnp.array(np.random.randn(num_samples, mu.shape[0]))  # shape: (B,␣
↪N)

    samples = epsilon * jnp.exp(log_ls) + mu  # shape: (B, N)
    logp_estimate = jnp.mean(logp(samples))

    '''
    To calculate log_q we can compute the pdfs before transforming the samples␣
↪since the transformation
    does not change the likelihood!
    '''
    logq_estimate = jnp.mean(jnp.sum(-0.5 * (jnp.log(2 * pi) + epsilon ** 2),␣
↪axis=1))

    return logp_estimate - logq_estimate


def neg_elbo(params, games, num_samples=100):
    def logp(zs):
        return joint_log_density(zs, games)

    return -elbo(params, logp, num_samples)
```

1.5.4 Train loop

[42]:

```
def learn_variational_approx(init_params, games, num_epochs, lr=1e-2,␣
↪num_q_samples=10, print_every=100):
    params_cur = init_params

    grad_fn = grad(neg_elbo, argnums=0)
```

```
    iters = 0
    losses = []

    for epoch in range(num_epochs):
        print(f'Epoch {epoch+1}')
        for X_batch in dataloader(games, batch_size=256, shuffle=True):
            grad_params = grad_fn(params_cur, X_batch, num_q_samples)

            loss = neg_elbo(params_cur, X_batch, num_q_samples)
            losses.append(loss)

            if iters % print_every == 0:
                print(f'\tIteration {iters}, Loss: {loss}')

            for p in params_cur:
                params_cur[p] -= grad_params[p] * lr

            iters += 1

    return losses, params_cur
```

[43]:

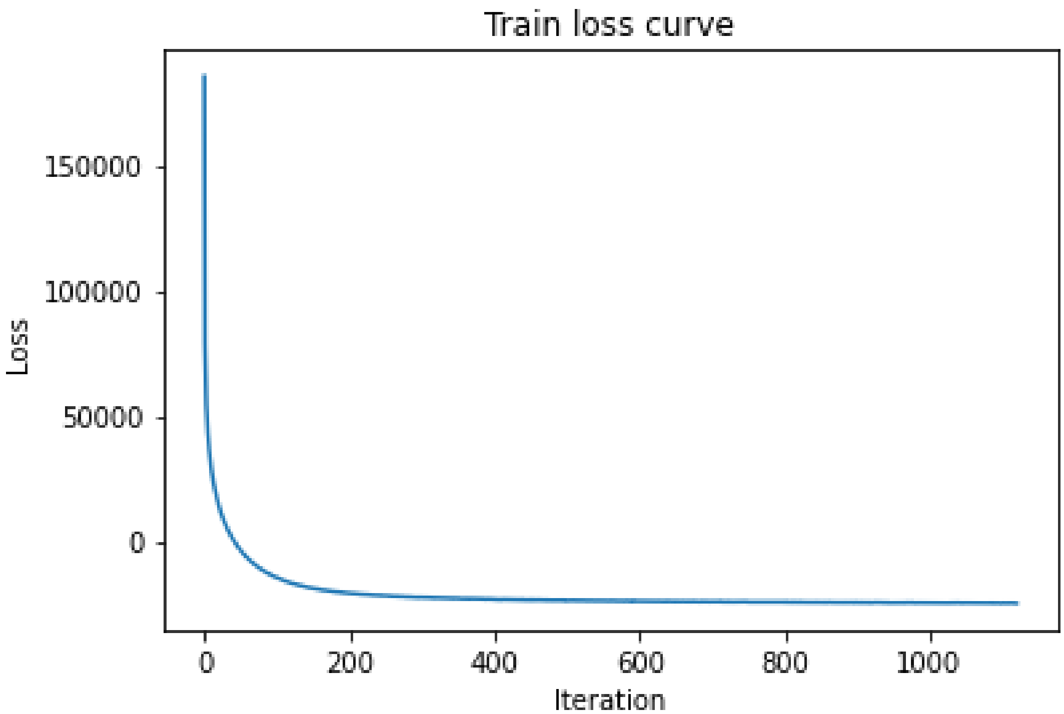
```
%%time

init_params = {'mu': jnp.array(np.random.randn(len(names),)), 'log_ls': jnp.
 ↪array(np.random.randn(len(names),))}

losses, params_cur = learn_variational_approx(init_params, games, num_epochs=1,␣
 ↪lr= 1e-2, num_q_samples = 10)
```

```
Epoch 1
        Iteration 0, Loss: 185436.46875
        Iteration 100, Loss: -14645.2421875
        Iteration 200, Loss: -20814.5078125
        Iteration 300, Loss: -22475.73828125
        Iteration 400, Loss: -23309.79296875
        Iteration 500, Loss: -23743.0078125
        Iteration 600, Loss: -23995.4375
        Iteration 700, Loss: -24210.94140625
        Iteration 800, Loss: -24420.62890625
        Iteration 900, Loss: -24555.7578125
        Iteration 1000, Loss: -24689.10546875
        Iteration 1100, Loss: -24726.390625
CPU times: user 1min 53s, sys: 7.82 s, total: 2min 1s
Wall time: 1min 32s
```

plot losses during ELBO optimization and report final loss:

[47]:
```
print('Final loss:', losses[-1])
plt.plot(losses)
plt.title('Train loss curve')
plt.xlabel('Iteration')
plt.ylabel('Loss');
```

```
Final loss: -24911.562
```



sort players by mean skill under our model and list top 10 players.

[48]:
```
params_names_joint = list(zip(params_cur['mu'], params_cur['log_ls'], names))
players_sorted = sorted(params_names_joint, key = lambda item: item[0],␣
 ↪reverse=True)
```

[49]:
```
print('mu \t log_ls \t name')
players_sorted[:10]
```

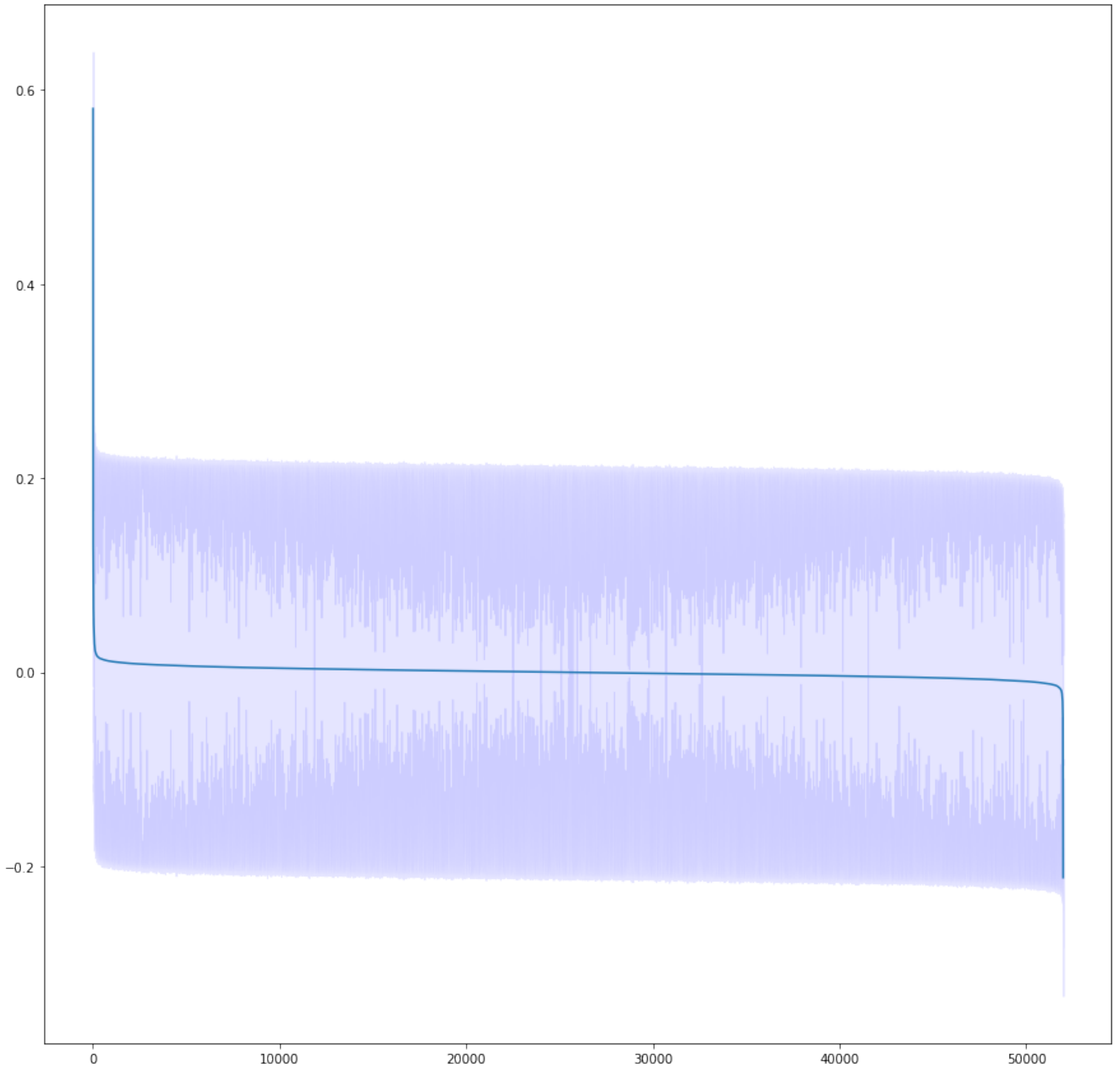
```
mu       log_ls          name
```

[49]:
```
[(0.58064646, -2.839418, 'sandro_mikanovic'),
 (0.24126501, -2.114113, 'usuba'),
 (0.23032774, -1.9641535, 'girijanr'),
 (0.19324963, -1.9165764, 'vesper2018'),
 (0.18054536, -1.9194374, 'sacsthexchange'),
```

```
 (0.17740448, -1.6515046, 'alex-brien'),
 (0.15707992, -2.0566213, 'malvinlim'),
 (0.13506743, -2.0798047, 'flexda'),
 (0.13156505, -2.5806167, 'bonjourbonjour'),
 (0.1295882, -1.808963, 'snakedad')]
```

plot mean and variance of all players, sorted by skill:

[50]:
```python
plt.figure(figsize=(15, 15))

ys = [item[0] for item in players_sorted]
error_ys = [jnp.exp(item[1]) for item in players_sorted]

plt.plot(ys, '-')

plt.fill_between(list(range(len(players_sorted))),
                 [y - error_y for y , error_y in zip(ys, error_ys)],
                 [y + error_y for y , error_y in zip(ys, error_ys)],
                 color='blue',
                 alpha=0.1);
```



1.6 7. More Approximate Inference with our Model

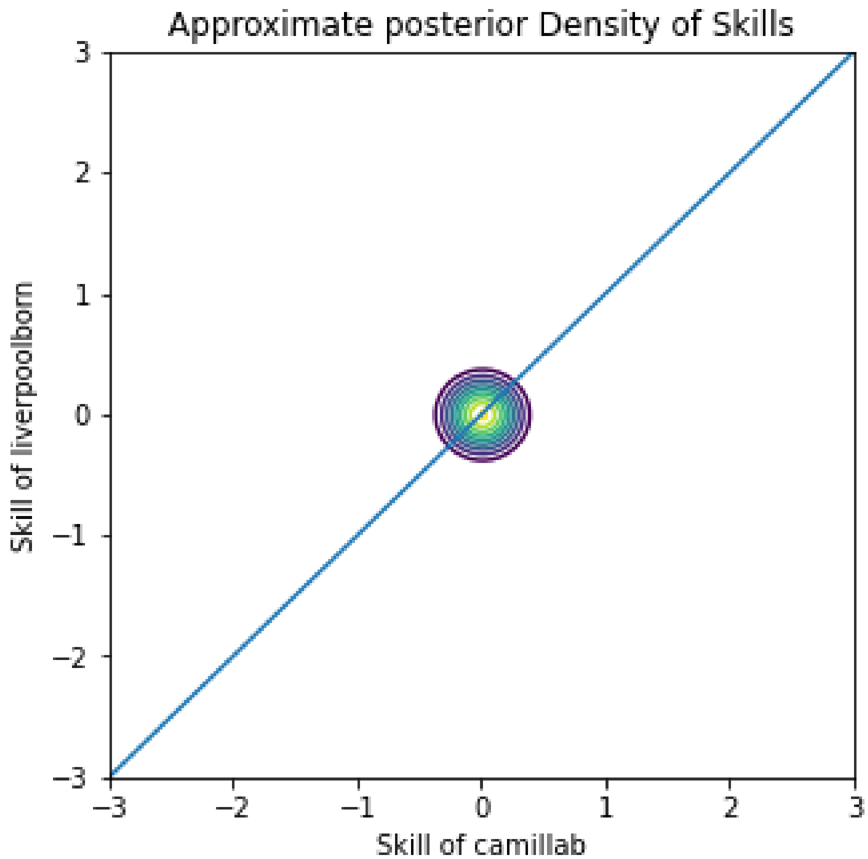
[57]:
```
camillab, liverpoolborn, meri_arabidze, sylvanaswindrunner = None, None, None,␣
↪None

for i, item in enumerate(players_sorted):
    name = item[2]
    if name == 'camillab':
        camillab = i
    elif name == 'liverpoolborn':
        liverpoolborn = i
    elif name == 'meri-arabidze':
        meri_arabidze = i
```

```
    elif name == 'sylvanaswindrunner':
        sylvanaswindrunner = i
```

1.6.1 most games vs most successful

Plot the approximate posterior over the skills of camillab and liverpoolborn.

[58]:
```
camillab_mean, camillab_std  = players_sorted[camillab][0], jnp.
 ↪exp(players_sorted[camillab][1])
liverpoolborn_mean, liverpoolborn_std  = players_sorted[liverpoolborn][0], jnp.
 ↪exp(players_sorted[liverpoolborn][1])

means = jnp.array([camillab_mean, liverpoolborn_mean])
stds = jnp.array([camillab_std, liverpoolborn_std])

f = lambda zs: jnp.exp(jnp.sum(-0.5 * jnp.log(2 * pi) - jnp.log(stds) - 0.5 *␣
 ↪((zs - means) / stds) ** 2,
                                      axis=1,
                                      keepdims=True))
fig = plt.figure(figsize=(5, 5))
plot_line_equal_skill()
plt.title('Approximate posterior Density of Skills')
plt.xlabel('Skill of camillab')
plt.ylabel('Skill of liverpoolborn')
skill_countour(f)
```




```
[59]: n_sample = 10000
skills = np.random.randn(n_sample, 2) * stds + means
print('Prob. of liverpoolborn more skillful than camillab:', (jnp.sum(skills[:,␣
↪0] < skills[:, 1]) / n_sample).item())
```

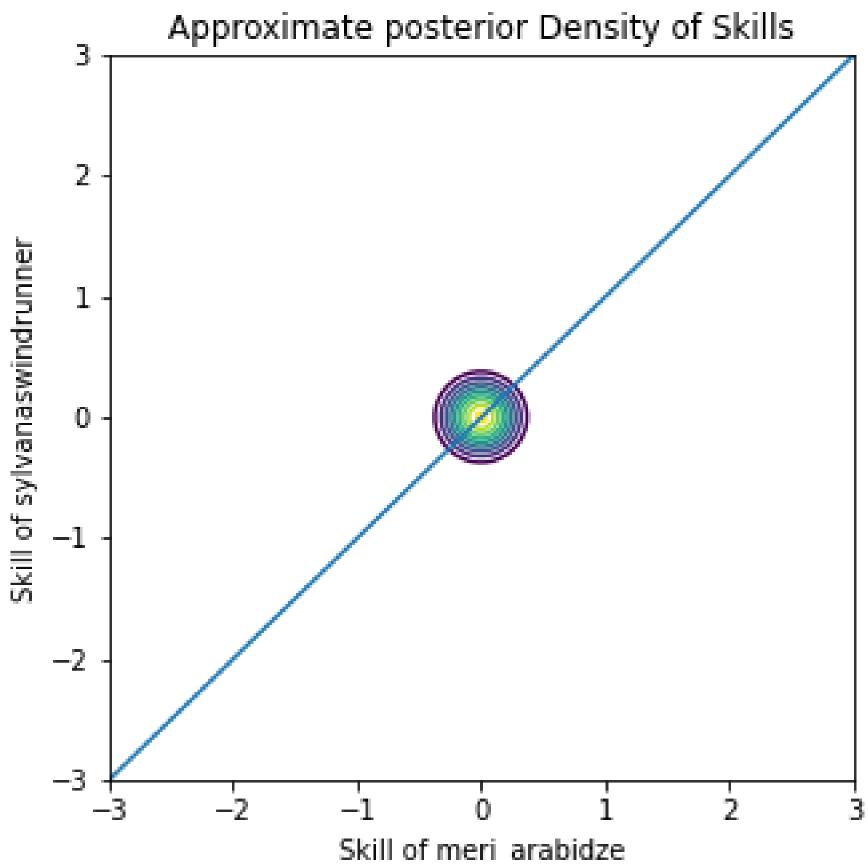
```
Prob. of liverpoolborn more skillful than camillab: 0.48829999566078186
```

1.6.2 all time high vs contender

Plot the approximate posterior over the skills of meri_arabidze and sylvanaswindrunner .

```
[60]: meri_arabidze_mean, meri_arabidze_std  = players_sorted[meri_arabidze][0], jnp.
↪exp(players_sorted[meri_arabidze][1])
sylvanaswindrunner_mean, sylvanaswindrunner_std  =␣
↪players_sorted[sylvanaswindrunner][0], jnp.
↪exp(players_sorted[sylvanaswindrunner][1])

means = jnp.array([meri_arabidze_mean, sylvanaswindrunner_mean])
stds = jnp.array([meri_arabidze_std, sylvanaswindrunner_std])
```

```
f = lambda zs: jnp.exp(jnp.sum(-0.5 * jnp.log(2 * pi) - jnp.log(stds) - 0.5 *
↪((zs - means) / stds) ** 2,
                                axis=1,
                                keepdims=True))
fig = plt.figure(figsize=(5, 5))
plot_line_equal_skill()
plt.title('Approximate posterior Density of Skills')
plt.xlabel('Skill of meri_arabidze')
plt.ylabel('Skill of sylvanaswindrunner')
skill_countour(f)
```



Estimate the probability that sylvanaswindrunner is more skillful than meri_arabidze .

[62]:
```
n_sample = 10000
skills = np.random.randn(n_sample, 2) * stds + means
print('Prob. of sylvanaswindrunner more skillful than meri_arabidze:', (jnp.
↪sum(skills[:, 0] < skills[:, 1]) / n_sample).item())
```

```
Prob. of sylvanaswindrunner more skillful than meri_arabidze: 0.5083000063896179
```