STA414_2024_Assignment_2_Solution

March 5, 2024

1 Probabilistic ML: Assignment 2 Solution

- **Deadline**: .
- **Submission**: You need to submit your solutions through Crowdmark, including all your derivations, plots, and your code. You can produce the files however you like (e.g. LaTeX, Microsoft Word, etc), as long as it is readable. Points will be deducted if we have a hard time reading your solutions or understanding the structure of your code.
- **Collaboration policy**: After attempting the problems on an individual basis, you may discuss and work together on the assignment with up to two classmates. However, **you must write your own code and write up your own solutions individually and explicitly name any collaborators** at the top of the homework.

2 Q1 - Image Denoising

In this problem, we will implement the sum-product Loopy **belief propagation** (Loopy-BP) method for denoising binary images which you have seen in tutorial 4. We will consider images as matrices of size $\sqrt{n} \times \sqrt{n}$. Each element of the matrix can be either 1 or -1, with 1 representing white pixels and -1 representing black pixels. This is different from the 0/1 representation commonly used for other CV tasks. This notation will be more convenient when multiplying with pixel values.

2.0.1 Data preparation

Below we provide you with code for loading and preparing the image data.

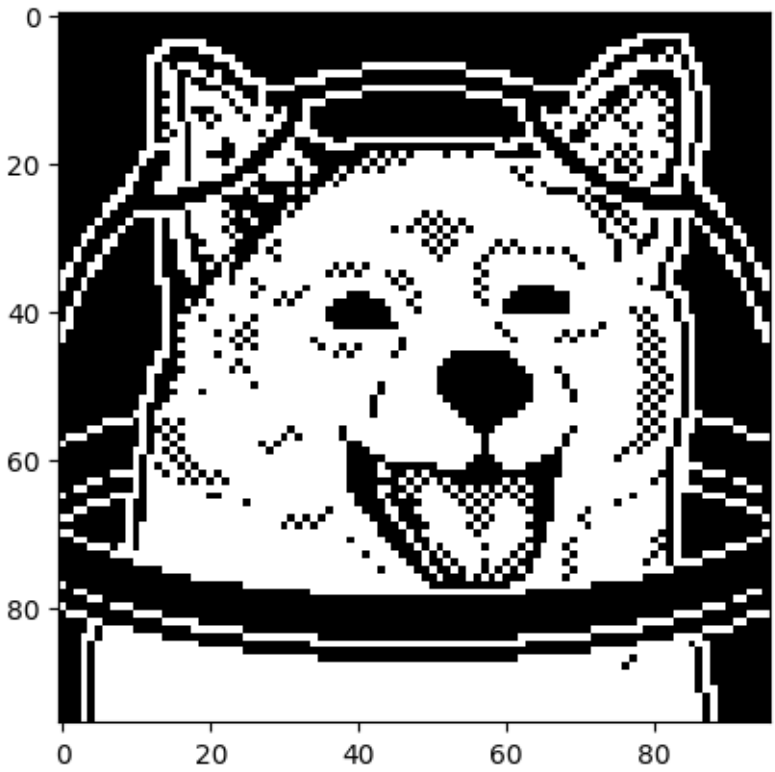
First, we load a black and white image of Lalika and convert it into a binary matrix of 1 and -1. So that white pixels have value 1 and black pixels have value -1.

[ ]:
```
!pip install wget

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import PIL.Image as Image
from os.path import exists
from wget import download
from tqdm import tqdm
filename, url = "3vaef0cog4f61.png", "https://i.redd.it/3vaef0cog4f61.png"
```

```
def load_img():
    if not exists(filename):
        download(url)

    with open(filename, 'rb') as fp:
        img2 = Image.open(fp).convert('L')
        img2 = np.array(img2)
    return (img2[:96,11:107] > 120) * 2.0 - 1

img_true = load_img()
plt.imshow(img_true, cmap='gray')
```

```
Collecting wget
  Downloading wget-3.2.zip (10 kB)
  Preparing metadata (setup.py) … done
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) … done
  Created wheel for wget: filename=wget-3.2-py3-none-any.whl size=9655
sha256=d260e9cd1ee411d3085a8207f58bcb112ddcdb8dd6d0b8c8fd07f85fe50a96ec
  Stored in directory: /root/.cache/pip/wheels/8b/f1/7f/5c94f0a7a505ca1c81cd1d92
08ae2064675d97582078e6c769
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
```

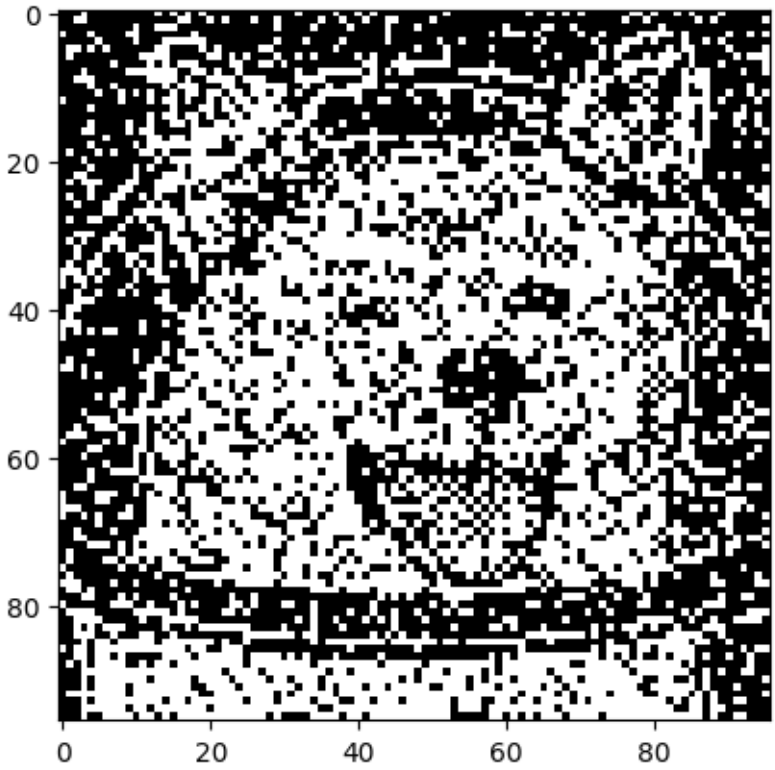
[ ]: <matplotlib.image.AxesImage at 0x794156f6a2c0>



To introduce noise into the image, for each pixel, swap its value between 1 and -1 with rate 0.2.

[ ]:
```
def gen_noisyimg(img, noise=.05):
    swap = np.random.binomial(1, noise, size=img.shape)
    return img * (2 * swap - 1)

noise = 0.2
img_noisy = gen_noisyimg(img_true, noise)
plt.imshow(-1 * img_noisy, cmap='gray')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x794155b59090>
```



2.0.2 The Loopy BP algorithm

Recall from lecture and tutorial, the Loopy-BP algorithm iteratively updates the messages of each node through a sum-product operation. The **sum-product** operation computes the joint inbound message through multiplication, and then marginalizes the factors through summation. This is in contrast to the **max-product** BP, which computes the maximum a-posteriori value for each variable through taking the maximum over variables.

Initialization:

For discrete node x_j with 2 possible states, $m_{i\to j}$ can be written as a 2 dimensional real vector $\mathrm{m}_{i,j}$ with $m_{i\to j}(x_j) = \mathrm{m}_{i,j}[index(x_j)]$. We initialize them uniformly to $m_{i\to j}(x_j) = 1/2$.

(Aside: for continuous cases, $m_{i\to j}(x_j)$ is a real valued function of x_j. We only need to deal with the discrete case here.)

For a number of iterations:

For node x_j in $\{x_s\}_{s=1}^n$:

1. Compute the product of inbound messages from neighbours of x_j:

$$\prod_{k\in N(j)\neq i} m_{k\to j}(x_j)$$

2. Compute potentials $\psi_j(x_j) = \exp(\beta x_j y_j)$ and $\psi_{ij}(x_i, x_j) = \exp(J x_i x_j)$. This expression specifically holds when $x \in \{-1, +1\}$.

3. Marginalize over $x_j = \{-1, +1\}$ to get $m_{j\to i}(x_i)$:

$$m_{j\to i}(x_i) = \sum_{x_j} \psi_j(x_j)\psi_{ij}(x_i, x_j) \prod_{k\in N(j)\neq i} m_{k\to j}(x_j)$$

4. Normalize messages for stability $m_{j\to i}(x_i) = m_{j\to i}(x_i)/\sum_{x_i} m_{j\to i}(x_i)$.

Compute beliefs after message passing is done.

$$b(x_i) \propto \psi_i(x_i) \prod_{j\in\mathcal{N}(i)} m_{j\to i}(x_i).$$

You'll be tasked to perform steps 1-3 in the iterations and computing the beliefs. We will provide you with helper functions for initialization, finding neighbours, and normalization.

2.0.3 Initialization

Initialize the message between neighbor pixels uniformly as $m_{j\to i}(x_i) = 1/k$. Since each pixel can only be 1 or -1, message has two values $m_{j\to i}(1)$ and $m_{j\to i}(-1)$. We also initialize hyperparameters J and β.

[ ]:
```
y = img_noisy.reshape([img_true.size, ])
num_nodes = len(y)
init_message = np.zeros([2, num_nodes, num_nodes]) + .5
J = 1.0
beta = 1.0
```

Find the neighboring pixels around a given pixel, which will be used for BP updates

[ ]:
```
def get_neighbors_of(node):
    """
    arguments:
     int node:  in [0,num_nodes) index of node to query
    globals:
     int num_nodes: number of nodes
    return: set(int) indices of neighbors of queried node
    """
    neighbors = []
    m = int(np.sqrt(num_nodes))
    if (node + 1) % m != 0:
        neighbors += [node + 1]
    if node % m != 0:
        neighbors += [node - 1]
    if node + m < num_nodes:
        neighbors += [node + m]
    if node - m >= 0:
        neighbors += [node - m]

    return set(neighbors)
```

2.1 Q1.1 Implement message passing in BP

Implement the function `get_message()` that computes the message passed from node j to node i:

$$m_{j\to i}(x_i) = \sum_{x_j} \psi_j(x_j)\psi_{ij}(x_i, x_j) \prod_{k\in N(j)\neq i} m_{k\to j}(x_j)$$

`get_message()` will be used by (provided below) `step_bp()` to perform one iteration of loopy-BP: it first normalizes the returned message from `get_message()`, and then updates the message with momentum `1.0 - step`.

[ ]:
```
def get_message(node_from, node_to, messages):
    """
    arguments:
     int node_from: in [0,num_nodes) index of source node
     int node_from: in [0,num_nodes) index of target node
     float array messages: (2, num_nodes, num_nodes), messages[:,j,i] is message
                           from node j to node i
    reads globals:
     float array y: (num_nodes,) observed pixel values
     float J: clique coupling strength constant
     float beta: observation to true pixel coupling strength constant
    return: array(float) of shape (2,) un-normalized message from node_from to
    node_to
    """
    #TODO: implement your function here
    # Solution:
    P = + J + beta * y[node_from]
    N = - J + beta * y[node_from]

    neighbors = get_neighbors_of(node_from)
    inMessage = np.prod(messages[:, list(neighbors.difference(set([node_to])))␣
 ↪node_from], axis=1)

    message = np.vstack([np.sum(np.exp(P * np.array([+1., -1.])) * inMessage),
                         np.sum(np.exp(N * np.array([+1., -1.])) * inMessage)]).
 ↪reshape([2, ])

    return message


def step_bp(step, messages):
    """
    arguments:
     float step: step size to update messages
    return
     float array messages: (2, num_nodes, num_nodes), messages[:,j,i] is message
                           from node j to node i
    """
```

```
    for node_from in range(num_nodes):
        for node_to in get_neighbors_of(node_from):
            m_new = get_message(node_from, node_to, messages)
            # normalize
            m_new = m_new / np.sum(m_new)

            messages[:, node_from, node_to] = step * m_new + (1. - step) * \
                messages[:, node_from, node_to]
    return messages
```

Then, run loopy BP update for 10 iterations:

[ ]:
```
num_iter = 10
step = 0.5
for it in range(num_iter):
    init_message = step_bp(step, init_message)
    print(it + 1,'/',num_iter)
```

```
1 / 10
2 / 10
3 / 10
4 / 10
5 / 10
6 / 10
7 / 10
8 / 10
9 / 10
10 / 10
```

2.2 Q1.2 Computing belief from messages

Now, calculate the unnormalized belief for each pixel

$$\tilde{b}(x_i) = \psi_i(x_i) \prod_{j \in N(i)} m_{j \to i}(x_i),$$

and normalize the belief across all pixels

$$b(x_i) = \frac{\tilde{b}(x_i)}{\sum_{x_j} \tilde{b}(x_j)}.$$

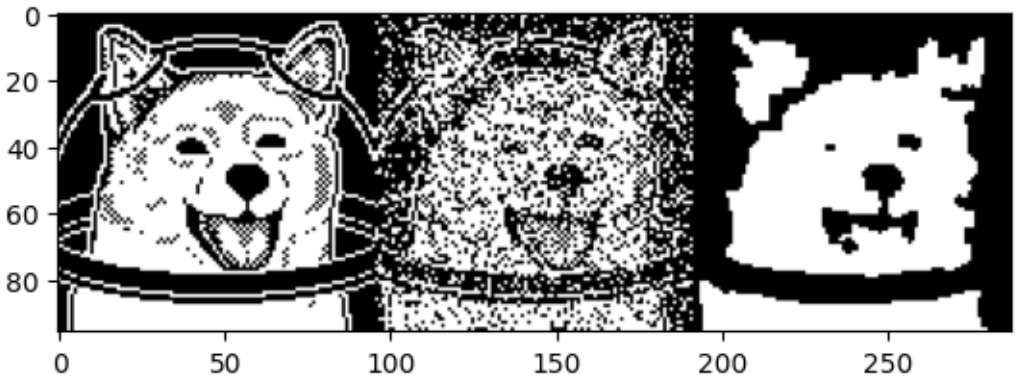
[ ]:
```
def update_beliefs(messages):
    """
    arguments:
    float array messages: (2, num_nodes, num_nodes), messages[:,j,i] is message
                          from node j to node i
    reads globals:
     float beta: observation to true pixel coupling strength constant
```

```
     float array y: (num_nodes,) observed pixel values
    returns:
     float array beliefs: (2, num_nodes), beliefs[:,i] is the belief of node i
    """
    beliefs = np.zeros([2, num_nodes])
    for node in range(num_nodes):
        #TODO: implement belief calculation here
        #Solution:
        neighbors = get_neighbors_of(node)
        inMessage = np.prod(messages[:, list(neighbors), node], axis=1)
        belief = np.exp(beta * y[node] * np.array([+1., -1.])) * inMessage
        beliefs[:, node] = belief / np.sum(belief)
    return beliefs

# call update_beliefs() once
beliefs = update_beliefs(init_message)
```

Finally, to get the denoised image, we use 0.5 as the threshold and consider pixel with belief less than threshold as black while others as white.

[ ]:
```
pred = 2. * ((beliefs[1, :] > .5) + .0) - 1.
img_out = pred.reshape(img_true.shape)

plt.imshow(np.hstack([img_true, -1*img_noisy, img_out]), cmap='gray')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x794155beb9a0>
```



2.3 Question 1.3 Momentum in belief propagation

In the sample code provided above, we performed message update with a momentum parameter `step`. In this question, you will experimentally investigate how momentum affects the characteristics of convergence.

2.3.1 Question 1.3.1

Complete the function `test_trajectory` below to obtain predicted image after each step of message passing. Return predicted images as list.

[ ]:
```
def test_trajectory(step_size, max_step=10):
    """
    step_size: step_size to update messages in each iteration
    max_step: number of steps
    """
    # re-initialize each time
    messages = np.zeros([2, num_nodes, num_nodes]) + .5
    images = []

    # solution:
    for it in range(max_step):
        messages = step_bp(step_size, messages)
        beliefs = update_beliefs(messages)
        pred = 2. * ((beliefs[1, :] > .5) + .0) - 1.
        img_out = pred.reshape(img_true.shape)
        images.append(img_out)

    return images
```

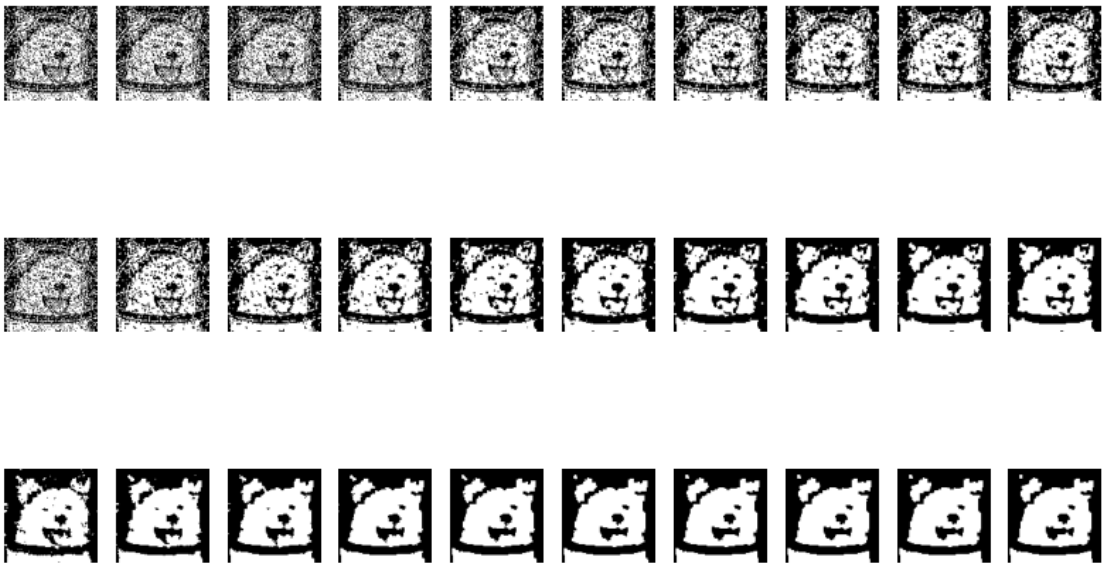
2.3.2 Question 1.3.2

Use test trajectory to create image serieses for `step size` 0.1, 0.3, and 1.0, each with 10 steps. Display these images with 'plot_series' provided below.

In the textbox below: 1. Comment on what happens when a large step size is used for too many iterations. 2. How would you adjust other hyperparameters to counteract this effect?

[ ]:
```
def plot_series(images):

  n = len(images)
  fig, ax = plt.subplots(1, n)
  for i in range(n):
    ax[i].imshow(images[i], cmap='gray')
    ax[i].set_axis_off()
  fig.set_figwidth(10)
  fig.show()

#Solution:
for step_size in tqdm([0.1, 0.3, 1.0]):
    imgs = test_trajectory(step_size, 10)
    plot_series(imgs)
```

```
100%|      | 3/3 [01:25<00:00, 28.45s/it]
```



2.3.3 Response to 1.3.2 (Enter your response below):

Solution 1. fine details in the image such as right eye are removed due to smoothing effect of message passing. 2. Use smaller J to reduce coupling strength -> reduce smoothing effect.

2.4 Question 1.4 Noise level and the hyperparameter J

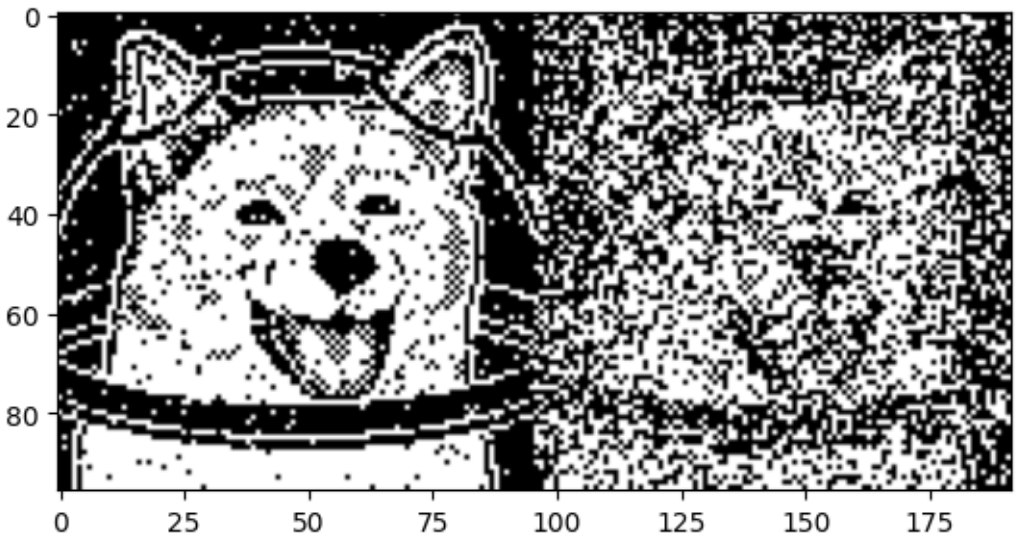
In this question, we will study how the level of noise in the image influences our choice in the hyperparameter J.

2.4.1 Question 1.4.1

First, generate and display images with noise of 0.05, 0.3. In the text box below, comment on what would happen if noise was set to 0.5 and 1.0

[ ]:
```
# Solution
imgs = []
for noise in [0.05, 0.3]:
  img_noisy = gen_noisyimg(img_true, noise)
  imgs.append(-1*img_noisy)
plt.imshow(np.hstack(imgs), cmap='gray')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x794154958610>
```



2.4.2 Response to 1.4.1 (enter your response below):

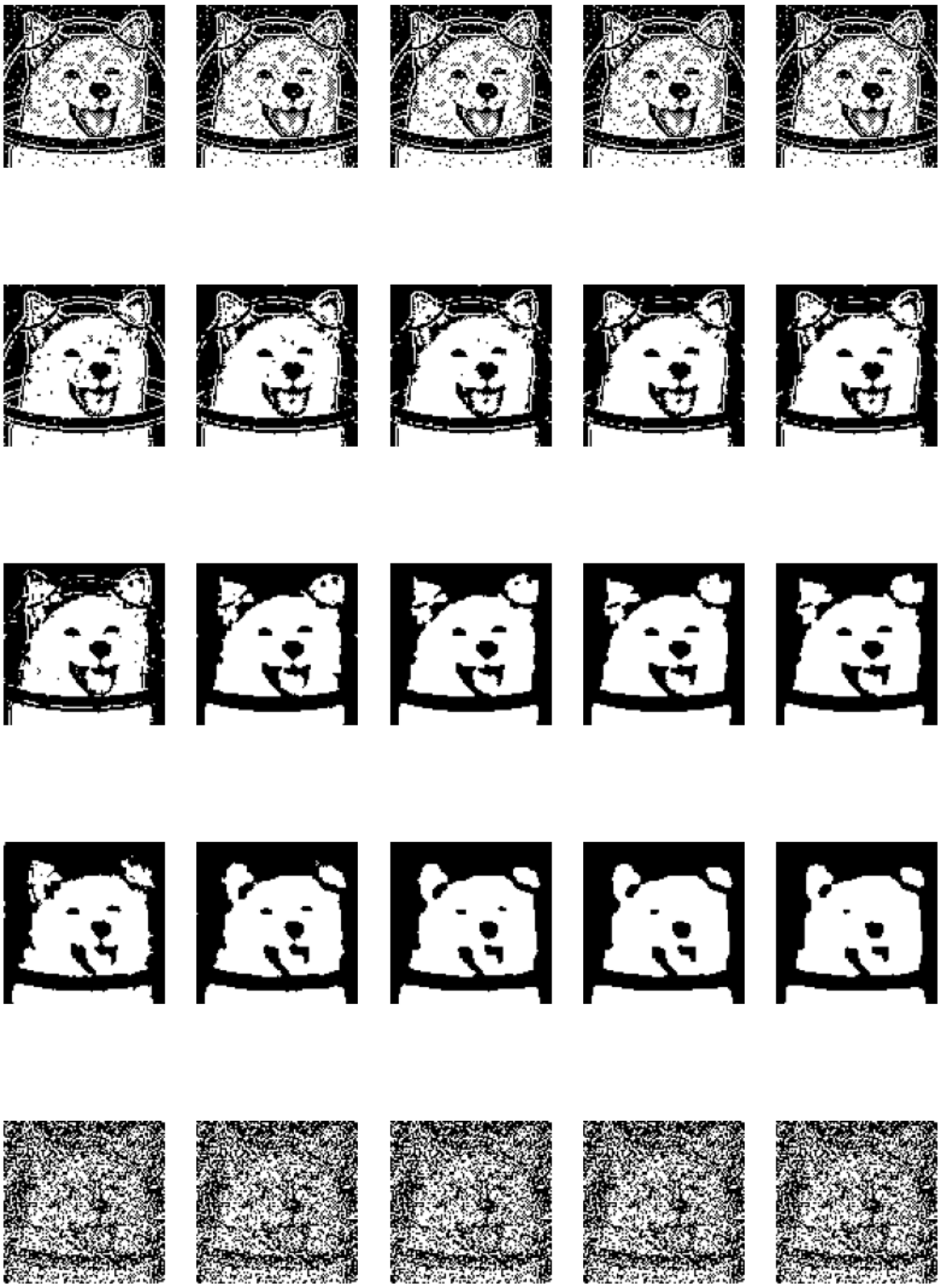
Solution: at 0.5, the image would be purely noise. at 1.0, the image would be inverted.

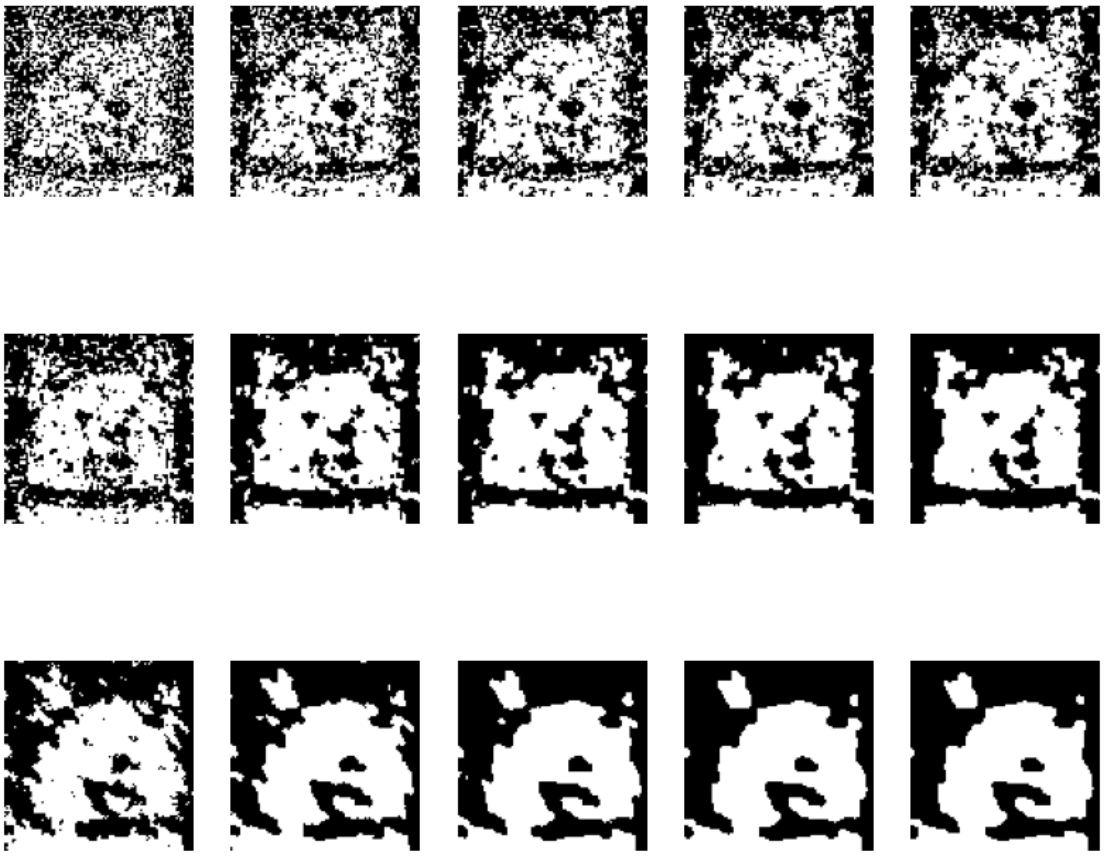
2.4.3 Question 1.4.2

Now, perform image denoising on images with noise levels 0.05 and 0.3 using $J = 0.1$, $J = 0.5$, $J = 1.0$, and $J = 5.0$. Set step size to 0.8 and max_step to 5. Plot the denoised images (if reusing `test_trajectory`, you should plot 8 image serieses). In text box below, comment on what you observe and provide a brief explanation on why this might occur.

[ ]:
```
from tqdm import tqdm
# Solution
beta = 1.0
for noise in [0.05, 0.3]:
    for J in tqdm([0.1, 0.5, 1.0, 5.0]):
        img_noisy = gen_noisyimg(img_true, noise)
        y = img_noisy.reshape([img_true.size, ])
        imgs = test_trajectory(0.8, 5)
        plot_series(imgs)
```

```
100%|      | 4/4 [01:00<00:00, 15.13s/it]
100%|      | 4/4 [01:01<00:00, 15.31s/it]
```





2.4.4 Response to 1.4.2 (enter your response below):

Solution: Higher values of J work better for higher noise level. Higher J represents a stronger belief that surrounding pixels must be similar, leading to a more smooth image. High values of J at small noise level results in "over-smoothing".

3 Question 2: Markov chain Monte Carlo in the TrueSkill model

The goal of this question is to get you familiar with the basics of Bayesian inference in medium-sized models with continuous latent variables, and the basics of Hamiltonian Monte Carlo.

3.1 Background

We'll 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.

This assignment is based on this one developed by Carl Rasmussen at Cambridge for his course on probabilistic machine learning.

3.2 Model definition

We'll consider a slightly simplified version of the original trueskill model. We assume that each player has a true, but unknown skill $z_i \in \mathbb{R}$. We use N to denote the number of players.

3.2.1 The prior:

The prior over each player's skill is a standard normal distribution, and all player's skills are *a priori* independent.

3.2.2 The likelihood:

For each observed game, the probability that player i beats player j, given the player's skills z_A and z_B, is:

$$p(A \text{ beat } B | z_A, z_B) = \sigma(z_A - z_B)$$

where

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

We chose this function simply because it's close to zero or one when the player's skills are very different, and equals one-half when the player skills are the same. This likelihood function is the only thing that gives meaning to the latent skill variables $z_1 \ldots z_N$.

There can be more than one game played between a pair of players. The outcome of each game is independent given the players' skills. We use M to denote the number of games.

[ ]:
```
!pip install wget
import os
import os.path

import matplotlib.pyplot as plt
import wget

import pandas as pd


import numpy as np
from scipy.stats import norm
import scipy.io
import scipy.stats
import torch
import random
from torch.distributions.normal import Normal

from functools import partial

import matplotlib.pyplot as plt
```

```
Collecting wget
  Downloading wget-3.2.zip (10 kB)
```

```
  Preparing metadata (setup.py) … done
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) … done
  Created wheel for wget: filename=wget-3.2-py3-none-any.whl size=9655
sha256=324f628ccf0f7b5cab81f98c8f7effa658de982803e4cc0ea98c6a85195370a6
  Stored in directory: /root/.cache/pip/wheels/8b/f1/7f/5c94f0a7a505ca1c81cd1d92
08ae2064675d97582078e6c769
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
```

3.3 Q 2.1 Implementing the TrueSkill Model [10 points]

3.3.1 Q 2.1.a [4 points]

Implement a function log_joint_prior that computes the log of the prior, jointly evaluated over all player's skills.

Specifically, given a $K \times N$ array where each row is a setting of the skills for all N players, it returns a $K \times 1$ array, where each row contains a scalar giving the log-prior for that set of skills.

[ ]:
```python
def log_joint_prior(zs_array):
  # TODO
  # Answer: DELETE ME BEFORE RELEASE
  m = Normal(0., 1.)
  return m.log_prob(zs_array).sum(axis=-1)
```

3.3.2 Q 2.1.b [6 points]

Implement two functions logp_a_beats_b and logp_b_beats_a.

Given a pair of skills z_a and z_b, logp_a_beats_b evaluates the log-likelihood that player with skill z_a beat player with skill z_b under the model detailed above, and logp_b_beats_a is vice versa.

To ensure numerical stability, use the function torch.logaddexp

[ ]:
```python
def logp_a_beats_b(z_a, z_b):
  # Hint: Use torch.logaddexp
  # TODO

  # Answer: DELETE ME BEFORE RELEASE
  return -torch.logaddexp(torch.tensor([0.0]), z_b - z_a)

def logp_b_beats_a(z_a, z_b):
  # Hint: Use torch.logaddexp
  # TODO

  # Answer: DELETE ME BEFORE RELEASE
  return -torch.logaddexp(torch.tensor([0.0]), z_a - z_b)
```

3.4 Q 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 plot_isocontours which evaluates a provided function on a grid of z_A and z_B's and plots the isocontours of that function. There is also a function plot_2d_fun. We have included an example for how you can use these functions.

[ ]:
```
# Plotting helper functions
def plot_isocontours(ax, func, steps=100):
    x = torch.linspace(-4, 4, steps=steps)
    y = torch.linspace(-4, 4, steps=steps)
    X, Y = torch.meshgrid(x, y, indexing="ij")
    Z = func(X, Y)
    cs = plt.contour(X, Y, Z )
    plt.clabel(cs, inline=1, fontsize=10)
    ax.set_yticks([])
    ax.set_xticks([])

def plot_2d_fun(f, x_axis_label="", y_axis_label="", scatter_pts=None):
    # This is the function your code should call.
    # f() should take two arguments.
    fig = plt.figure(figsize=(8,8), facecolor='white')
    ax = fig.add_subplot(111, frameon=False)
    ax.set_xlabel(x_axis_label)
    ax.set_ylabel(y_axis_label)
    plot_isocontours(ax, f)
    if scatter_pts is not None:
      plt.scatter(scatter_pts[:,0], scatter_pts[:, 1])
    plt.plot([4, -4], [4, -4], 'b--')   # Line of equal skill
    plt.show(block=True)
    plt.draw()
```

3.4.1 Q 2.2.a [2 point]

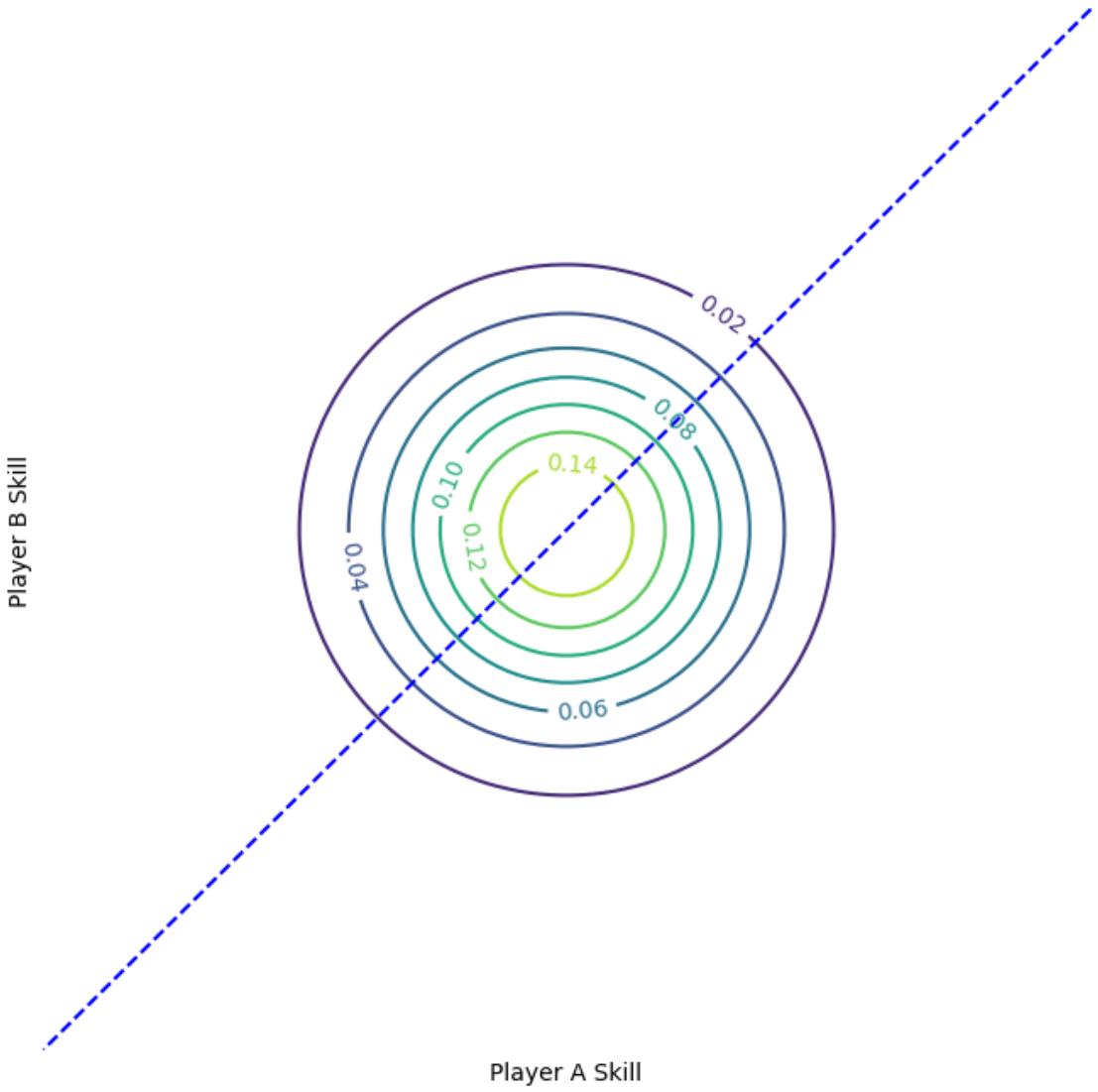
For two players A and B, plot the isocontours of the joint prior over their skills. Also plot the line of equal skill, $z_A = z_B$. Use the helper function `plot_2d_fun` above to plot.

According to the prior, what's the chance that player A is better than player B?

[ ]:
```
def log_prior_over_2_players(z1, z2):
  # TODO
  # Answer: DELETE ME BEFORE RELEASE
  m = Normal(0., 1.)
  return m.log_prob(z1) + m.log_prob(z2)

def prior_over_2_players(z1, z2):
  return torch.exp(log_prior_over_2_players(z1, z2))
```

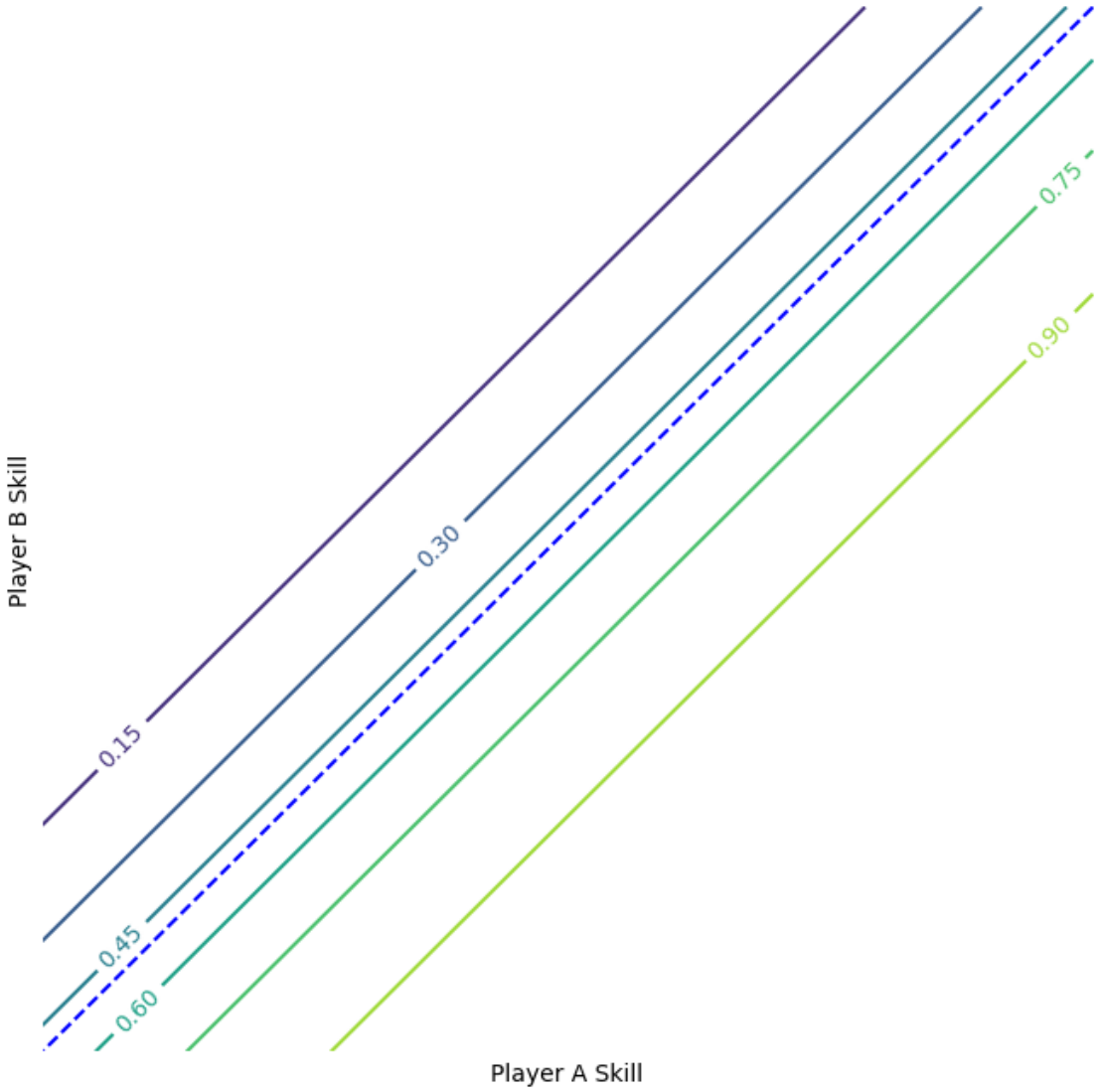
```
plot_2d_fun(prior_over_2_players, "Player A Skill", "Player B Skill")
```

```
torch.Size([100, 100]) torch.Size([100, 100])
```




```
<Figure size 640x480 with 0 Axes>
```

[ ]:
```
# Note:  This isn't part of the assignment
def likelihood_over_2_players(z1, z2):
  return torch.exp(logp_a_beats_b(z1, z2))

plot_2d_fun(likelihood_over_2_players, "Player A Skill", "Player B Skill")
```




```
<Figure size 640x480 with 0 Axes>
```

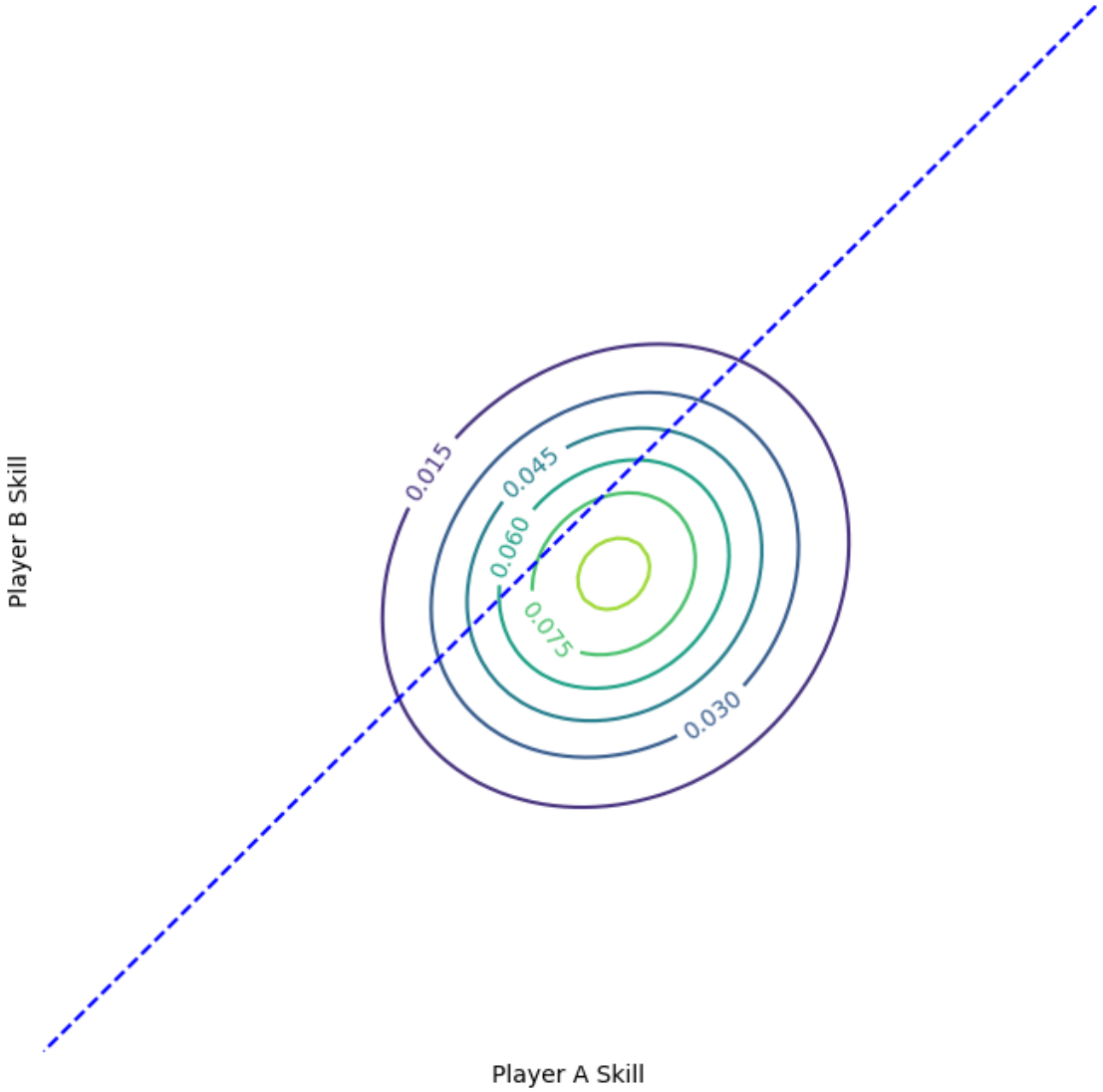
3.4.2 Q 2.2.b [3 points]

Plot isocountours of the joint posterior over z_A and z_B 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 $p(z_A, z_B|\text{A beat B})$. Also plot the line of equal skill, $z_A = z_B$.

To think about: According to this posterior, which player is likely to have higher skill?

[ ]:
```
def log_posterior_A_beat_B(z1, z2):
  # TODO: Combine the prior for two players with the likelihood for A beat B.
  # You might want to use the log_prior_over_2_players function from above.
```

```
  # Answer: DELETE ME BEFORE RELEASE
  return log_prior_over_2_players(z1, z2) + logp_a_beats_b(z1, z2)

def posterior_A_beat_B(z1, z2):
  return torch.exp(log_posterior_A_beat_B(z1, z2))

plot_2d_fun(posterior_A_beat_B, "Player A Skill", "Player B Skill")
# Note that the posterior probabilities shown are unnormalized
```



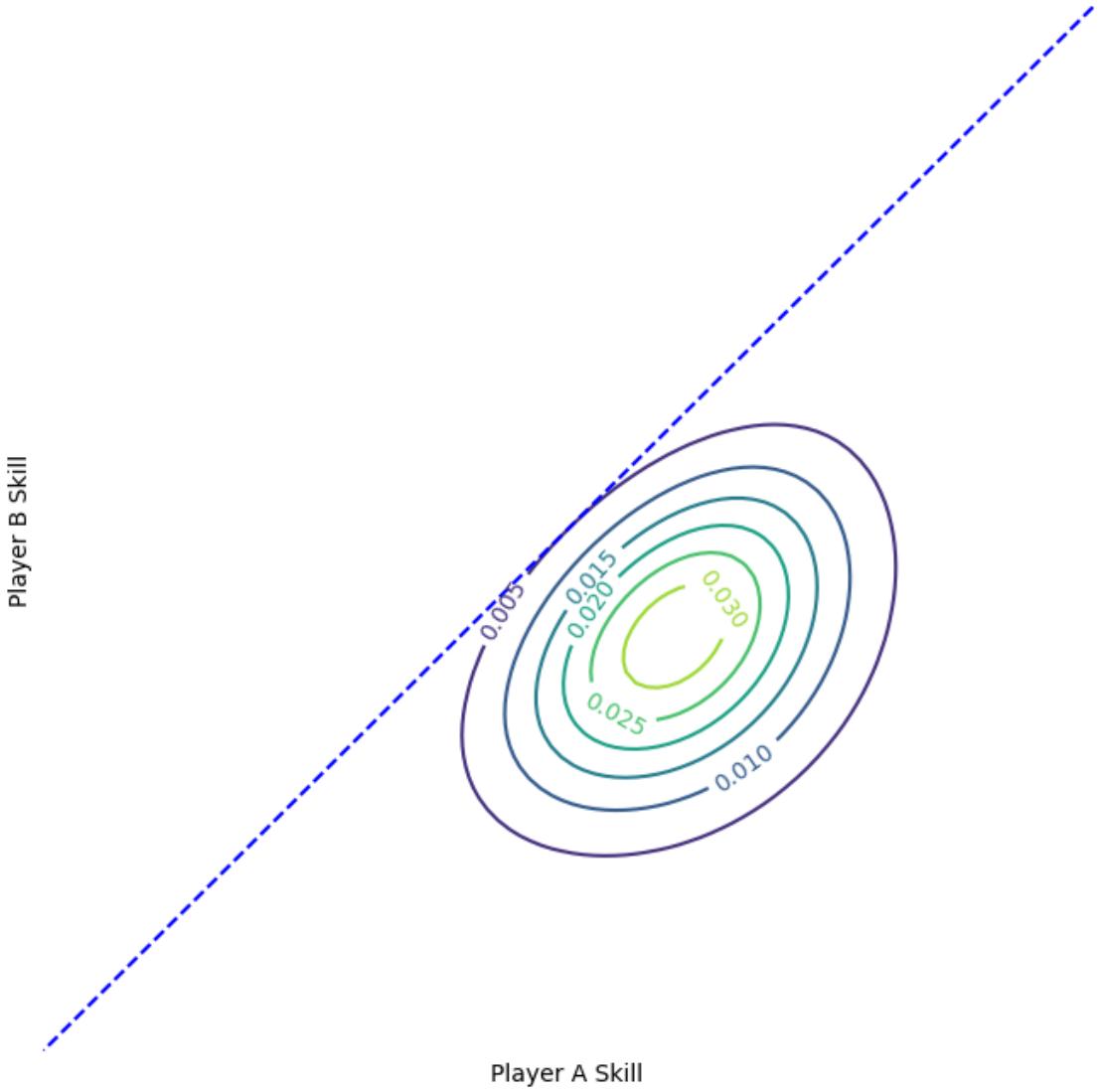

```
<Figure size 640x480 with 0 Axes>
```

3.4.3 Q 2.2.c [2 point]

Plot isocountours of the joint posterior over z_A and z_B given that 5 matches were played, and player A beat player B in all matches. Also plot the line of equal skill, $z_A = z_B$.

To think about: According to this posterior, is it plausible that player B is more skilled than player A?

[ ]:
```
def log_posterior_A_beat_B_5_times(z1, z2):
  # TODO: Combine the prior for two players with the likelihood for A beat B.
  # You might want to use your log_prior_over_2_players function.

  # Answer: DELETE ME BEFORE RELEASE
  return (log_prior_over_2_players(z1, z2) + 5.0 * logp_a_beats_b(z1, z2))

def posterior_A_beat_B_5_times(z1, z2):
  return torch.exp(log_posterior_A_beat_B_5_times(z1, z2))

plot_2d_fun(posterior_A_beat_B_5_times, "Player A Skill", "Player B Skill")
```




```
<Figure size 640x480 with 0 Axes>
```

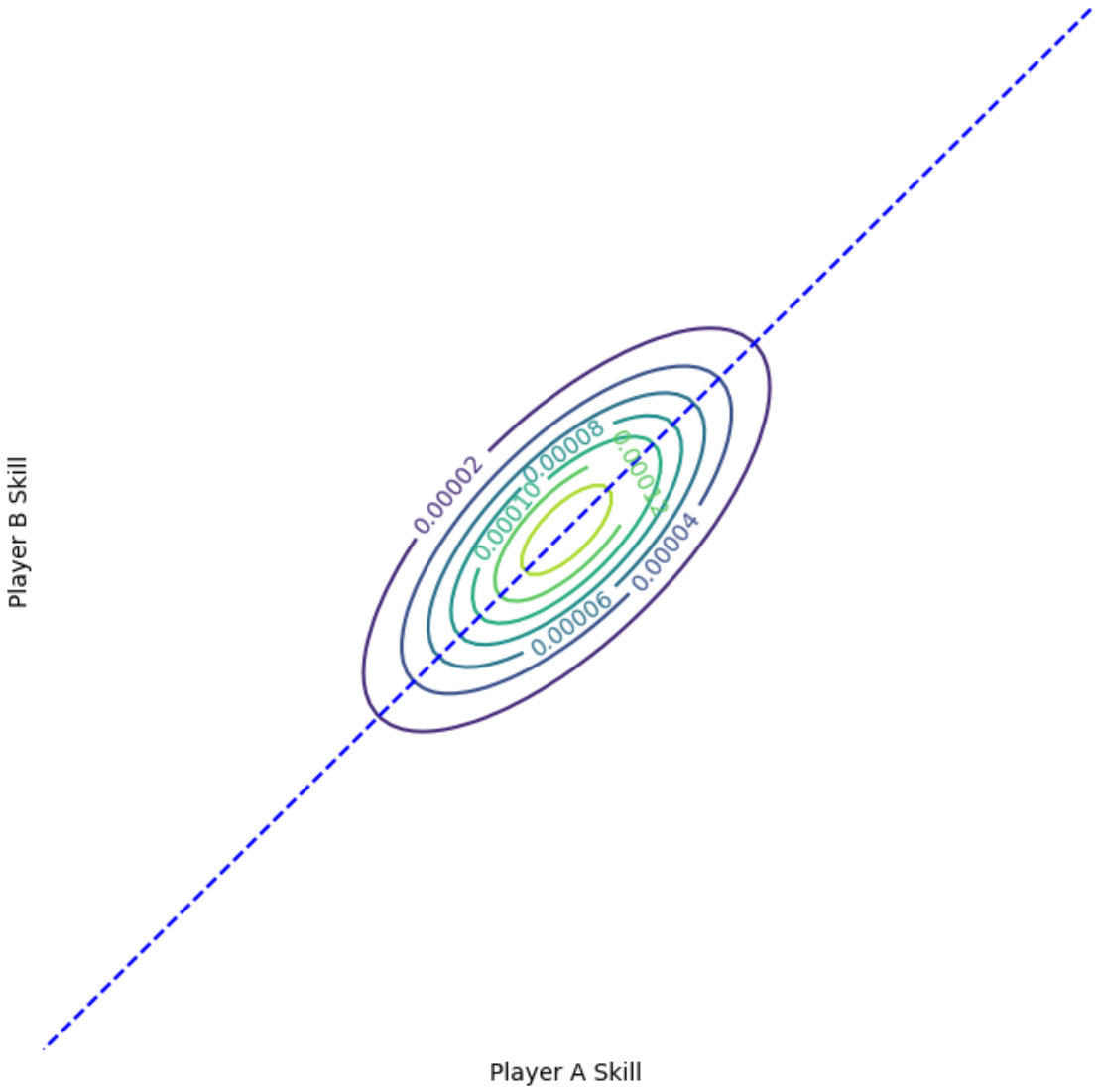
3.4.4 Q 2.2.d [3 point]

Plot isocontours of the joint posterior over z_A and z_B given that 10 matches were played, and each player beat the other 5 times. Also plot the line of equal skill, $z_A = z_B$.

To think about: According to this posterior, is it likely that one player is much better than another? Is it plausible that both players are better than average? Worse than average?

[ ]:
```python
def log_posterior_beat_each_other_5_times(z1, z2):
  # TODO: Combine the prior for two players with the likelihood for A beat B.
  # You might want to use your log_prior_over_2_players function from above.
```

```
  # Answer: DELETE ME BEFORE RELEASE
  return log_prior_over_2_players(z1, z2) \
    + 5.* logp_a_beats_b(z1, z2) \
    + 5.* logp_b_beats_a(z1, z2)

def posterior_beat_each_other_5_times(z1, z2):
  return torch.exp(log_posterior_beat_each_other_5_times(z1, z2))

plot_2d_fun(posterior_beat_each_other_5_times, "Player A Skill", "Player B␣
↪Skill")
```




```
<Figure size 640x480 with 0 Axes>
```

3.5 Q 2.3 Hamiltonian Monte Carlo on Two Players and Toy Data [4 points]

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.

In this question, we will approximate posterior distributions with gradient-based Hamiltonian Monte Carlo.

In the next assignment, we'll use gradient-based stochastic variational inference, which wasn't invented until around 2014.

[ ]:
```
random.seed(42)
torch.manual_seed(42)
```

[ ]:
```
<torch._C.Generator at 0x799e5bc2ab30>
```

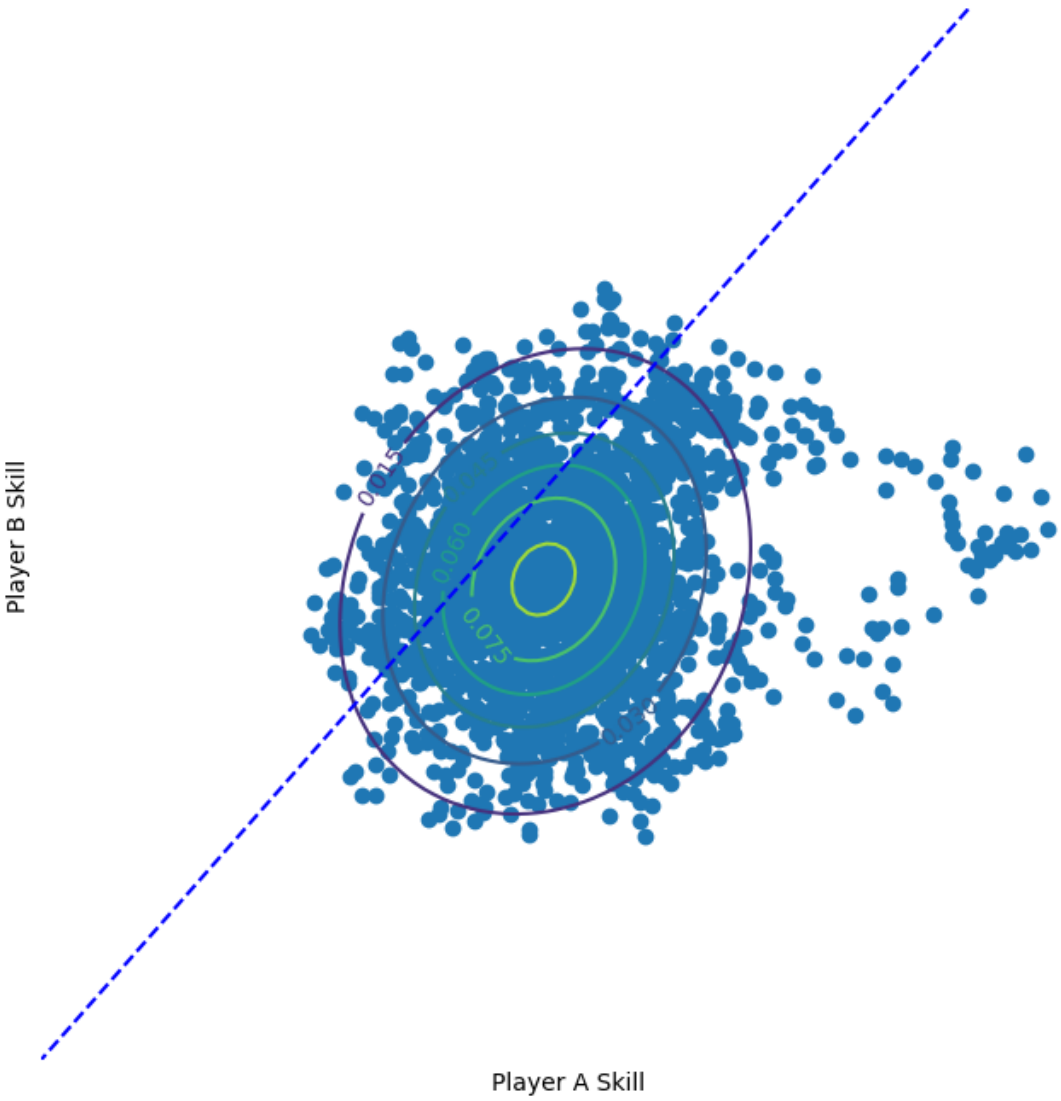
[ ]:
```
# Hamiltonian Monte Carlo
from tqdm import trange, tqdm_notebook  # Progress meters

def leapfrog(params_t0, momentum_t0, stepsize, logprob_grad_fun):
  # Performs a reversible update of parameters and momentum
  # See https://en.wikipedia.org/wiki/Leapfrog_integration
  momentum_thalf = momentum_t0    + 0.5 * stepsize * logprob_grad_fun(params_t0)
  params_t1 =      params_t0      +       stepsize * momentum_thalf
  momentum_t1 =    momentum_thalf + 0.5 * stepsize * logprob_grad_fun(params_t1)
  return params_t1, momentum_t1


def iterate_leapfrogs(theta, v, stepsize, num_leapfrog_steps, grad_fun):
  for i in range(0, num_leapfrog_steps):
    theta, v = leapfrog(theta, v, stepsize, grad_fun)
  return theta, v

def metropolis_hastings(state1, state2, log_posterior):
  # Compares the log_posterior at two values of parameters,
  # and accepts the new values proportional to the ratio of the posterior
  # probabilities.
  accept_prob = torch.exp(log_posterior(state2) - log_posterior(state1))
  if random.random() < accept_prob:
    return state2  # Accept
  else:
    return state1  # Reject

def draw_samples(num_params, stepsize, num_leapfrog_steps, n_samples,␣
 ↪log_posterior):
  theta = torch.zeros(num_params)

  def log_joint_density_over_params_and_momentum(state):
```

```
    params, momentum = state
    m = Normal(0., 1.)
    return m.log_prob(momentum).sum(axis=-1) + log_posterior(params)

  def grad_fun(zs):
    zs = zs.detach().clone()
    zs.requires_grad_(True)
    y = log_posterior(zs)
    y.backward()
    return zs.grad



  sampleslist = []
  for i in trange(0, n_samples):
    sampleslist.append(theta)

    momentum = torch.normal(0, 1, size = np.shape(theta))

    theta_new, momentum_new = iterate_leapfrogs(theta, momentum, stepsize,␣
 ↪num_leapfrog_steps, grad_fun)

    theta, momentum = metropolis_hastings((theta, momentum), (theta_new,␣
 ↪momentum_new), log_joint_density_over_params_and_momentum)
  return torch.stack((sampleslist))
```

Using samples generated by HMC, we cab approximate the joint posterior where we observe player A winning 1 game.

[ ]:
```
# Hyperparameters
num_players = 2
num_leapfrog_steps = 20
n_samples = 2500
stepsize = 0.01

def log_posterior_a(zs):
  z1, z2 = zs[0], zs[1]
  return log_posterior_A_beat_B(z1, z2)

samples_a = draw_samples(num_players, stepsize, num_leapfrog_steps, n_samples,␣
 ↪log_posterior_a)
plot_2d_fun(posterior_A_beat_B, "Player A Skill", "Player B Skill", samples_a)
```

```
100%|      | 2500/2500 [01:06<00:00, 37.37it/s]
```




```
<Figure size 640x480 with 0 Axes>
```

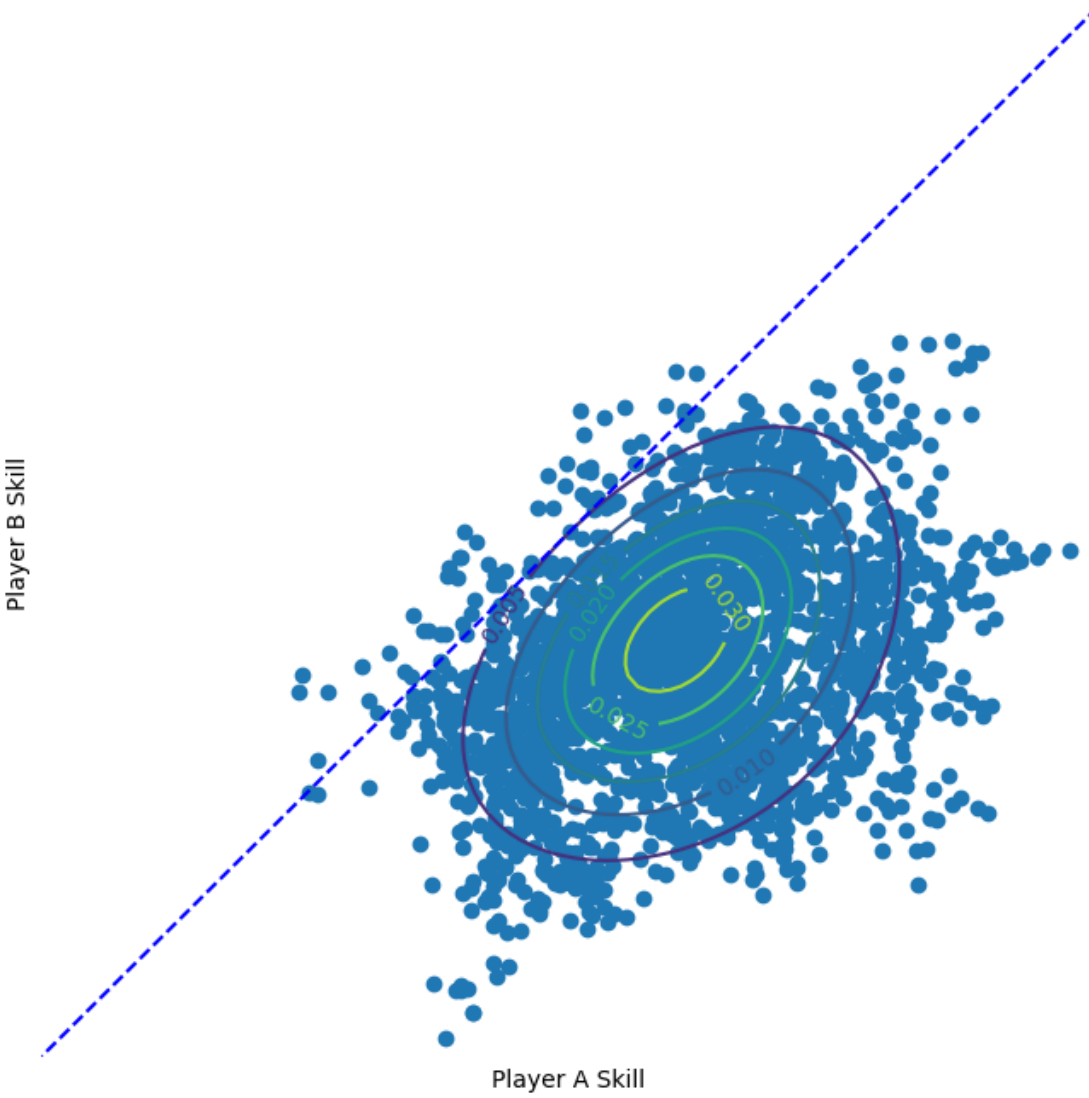
3.5.1 Q 2.3.a [2 point]

Using samples generated by HMC, approximate the joint posterior where we observe player A winning 5 games against player B. Hint: You can re-use the code from when you plotted the isocontours.

[ ]:
```
# Hyperparameters
num_players = 2
num_leapfrog_steps = 20
n_samples = 2500
stepsize = 0.01
key = 42
```

```
def log_posterior_b(zs):
  # TODO
  (z1, z2) = zs
  return log_posterior_A_beat_B_5_times(z1, z2)

samples_b = draw_samples(num_players, stepsize, num_leapfrog_steps, n_samples,␣
 ↪log_posterior_b)

# TODO Plot the posterior contour and the samples
ax = plot_2d_fun(posterior_A_beat_B_5_times, "Player A Skill", "Player B␣
 ↪Skill", samples_b)
```

```
100%|      | 2500/2500 [01:21<00:00, 30.74it/s]
```



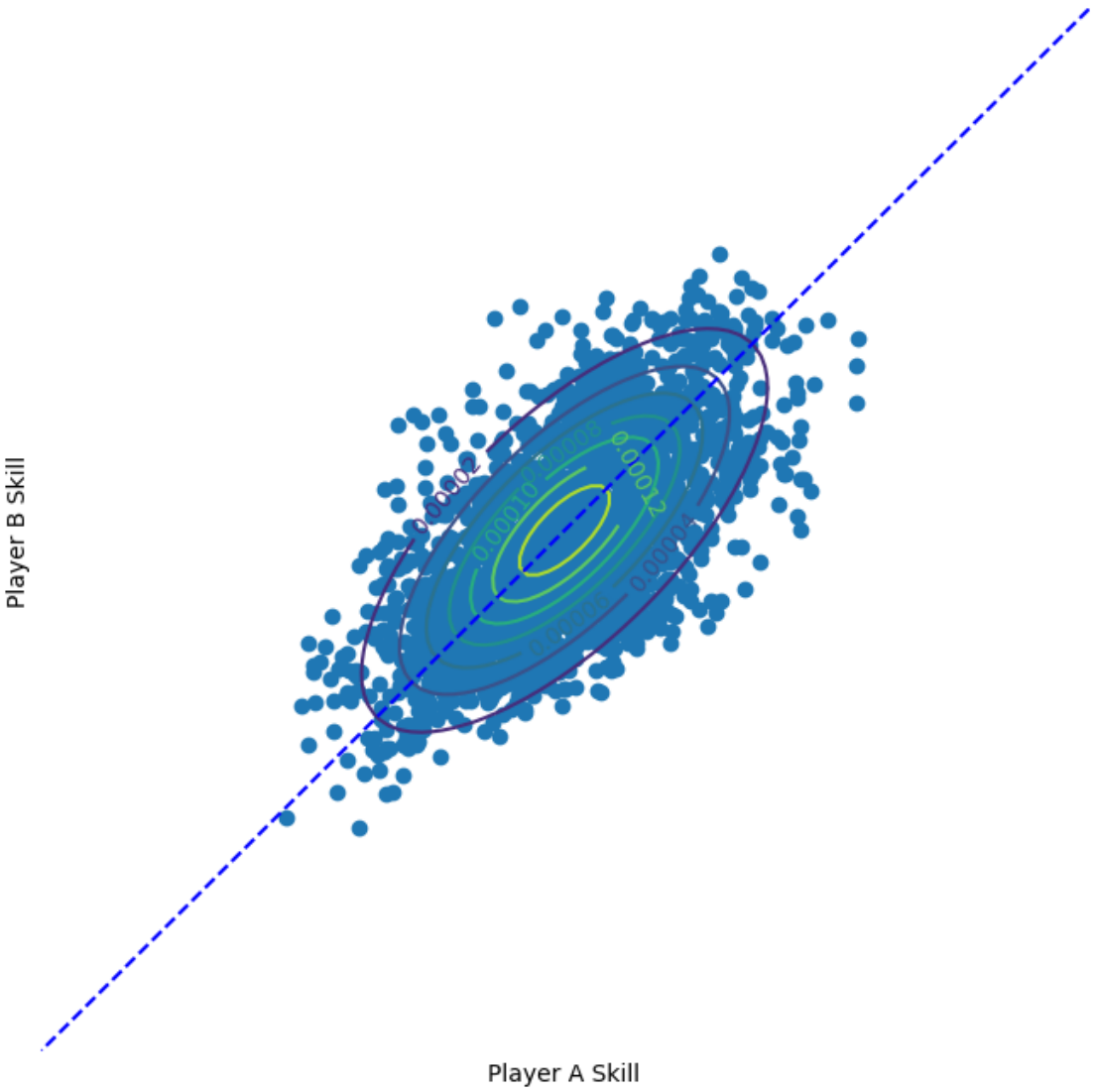
```
<Figure size 640x480 with 0 Axes>
```

3.5.2 Q 2.3.b [2 point]

Using samples generated by HMC, approximate the joint posterior where we observe player A winning 5 games and player B winning 5 games.

[ ]:
```
# Hyperparameters
num_players = 2
num_leapfrog_steps = 20
n_samples = 2500
stepsize = 0.01


def log_posterior_c(zs):
  # TODO
  return log_posterior_beat_each_other_5_times(zs[0], zs[1])

samples_c = draw_samples(num_players, stepsize, num_leapfrog_steps, n_samples,␣
 ↪log_posterior_c)

# TODO Plot the posterior contour and the samples
ax = plot_2d_fun(posterior_beat_each_other_5_times, "Player A Skill", "Player B␣
 ↪Skill", samples_c)
```

```
100%|      | 2500/2500 [01:27<00:00, 28.66it/s]
```




```
<Figure size 640x480 with 0 Axes>
```

3.6 Q 2.4 Approximate inference conditioned on real data [26 points]

The dataset contains data on 2546 chess games amongst 1434 players: - names is a 1163 by 1 matrix, whose i'th entry is the name of player i. - games is a 2543 by 2 matrix of game outcomes (actually chess 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.

It is based on the kaggle chess dataset: https://www.kaggle.com/datasets/datasnaek/chess

```
[ ]: wget.download("https://vahidbalazadeh.me/assets/datasets/chess_players.csv")
     wget.download("https://vahidbalazadeh.me/assets/datasets/chess_games.csv")
     games = pd.read_csv("chess_games.csv")[["winner_id", "loser_id"]].to_numpy()
     names = pd.read_csv("chess_players.csv")[["player_name"]].to_numpy().
      ↪astype('str')
```

```
[ ]: games = torch.LongTensor(games)
```

3.6.1 Q 2.4.a [5 points]

Assuming all game outcomes are i.i.d. conditioned on all players' skills, implement a function log_games_likelihood that takes a batch of player skills `zs` and a collection of observed games `games` and gives the total log-likelihoods for all those observations given all the 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]`.

```
[ ]: def log_games_likelihood(zs, games):
       # games is an array of size (num_games x 2)
       # zs is an array of size (num_players)
       #
       # Hint: With broadcasting, this function can be written
       # with no for loops.
       #
       winning_player_ixs = games[:,0]
       losing_player_ixs = games[:,1]
       # winning_player_skills =     #TODO: Look up the skill of the winning player␣
      ↪in each game.
       # losing_player_skills =      #TODO: Look up the skills of the losing player␣
      ↪in each game.
       # log_likelihoods =           #TODO: Compute the log_likelihood of each game␣
      ↪outcome.
       # return                      #TODO: Combine the log_likelihood of␣
      ↪independent events.

       # Answer: DELETE ME BEFORE RELEASE
       winning_player_skills = zs[winning_player_ixs]
       losing_player_skills = zs[losing_player_ixs]
       log_likelihoods = logp_a_beats_b(winning_player_skills, losing_player_skills)
       return torch.sum(log_likelihoods)
```

3.6.2 Q 2.4.b [3 points]

Implement a function joint_log_density which combines the log-prior and log-likelihood of the observations to give $p(z_1, z_2, \ldots, z_N, \text{all game outcomes})$

[ ]:
```
def log_joint_probability(zs, games):
  # Todo: Combine log_prior and log_likelihood

  # Answer: DELETE ME BEFORE RELEASE
  return log_joint_prior(zs) + log_games_likelihood(zs, games)
```

3.6.3 Q 2.4.c [5 points]

Run Hamiltonian Monte Carlo on the posterior over all skills conditioned on all the chess games from the dataset. Run for 10000 samples.

[ ]:
```
# Hyperparameters
num_players = 1163
num_leapfrog_steps = 20
n_samples = 10000
stepsize = 0.01

# TODO: all_games_samples = ...
def log_posterior(zs):
  return log_joint_probability(zs, games)

all_games_samples = draw_samples(num_players, stepsize, num_leapfrog_steps,␣
 ↪n_samples, log_posterior)
```

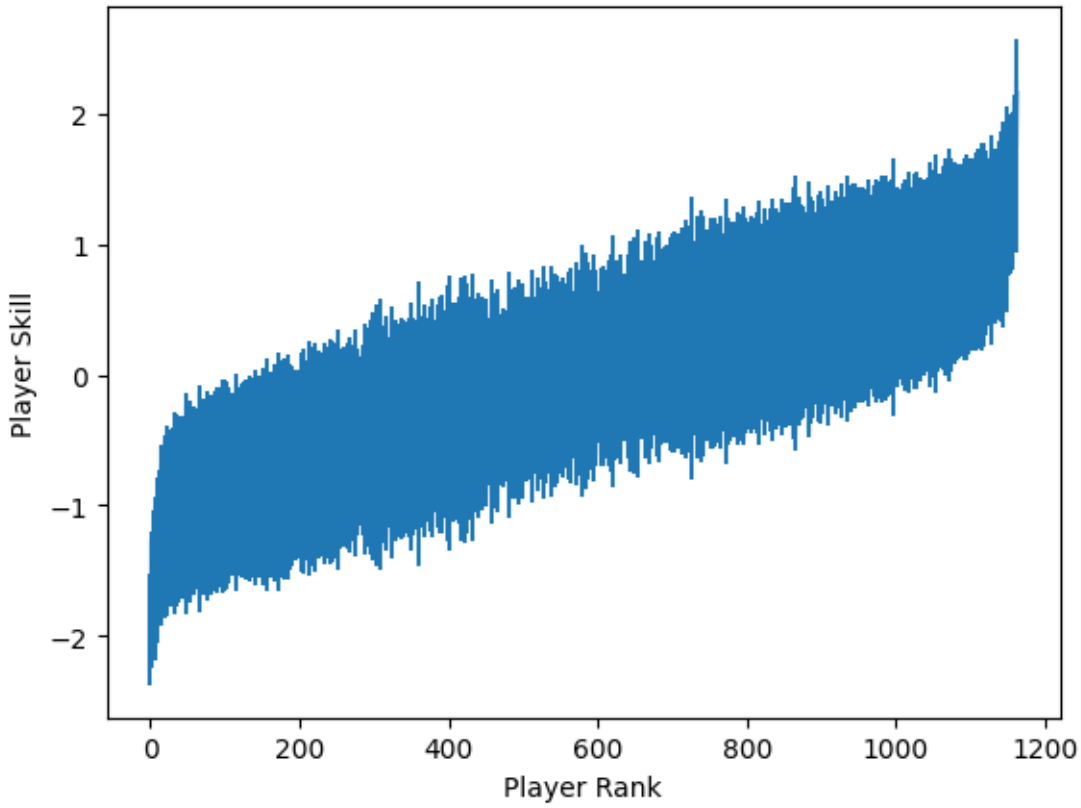
```
100%|      | 10000/10000 [05:57<00:00, 28.00it/s]
```

3.6.4 Q 2.4.d [3 points]

Based on your samples from the previous question, plot the approximate mean and variance of the marginal skill of each player, sorted by average skill. There's no need to include the names of the players. Label the axes "Player Rank", and "Player Skill".

[ ]:
```
# TODO
# mean_skills = ...
# var_skills = ...
mean_skills = torch.mean(all_games_samples, axis=0)
order = np.argsort(mean_skills)
mean_skills = mean_skills[order]
var_skills = torch.var(all_games_samples, axis=0)[order]

plt.xlabel("Player Rank")
plt.ylabel("Player Skill")
plt.errorbar(range(num_players), mean_skills, var_skills)
```

[ ]:
```
<ErrorbarContainer object of 3 artists>
```



3.6.5 Q 2.4.e [2 points]

List the names of the 5 players with the lowest mean skill and 5 players with the highest mean skill according to your samples.

[ ]:
```
print("Bottom 5")
# TODO: print the 5 players with the lowest mean skill
for i in range(0,5):
  print(names[order[i]])


print("Top 5")
# TODO: print the 5 players with the highest mean skill
for i in range(1,6):
  print(names[order[-i]])
```

```
Bottom 5
['josephelbouhessaini']
['thebestofthebad']
['vkmansftw']
['italiantranslator']
['kylarr']
```

```
Top 5
['doraemon61']
['mrzoom47']
['lzchips']
['smartduckduckcow']
['piroz_xucestih']
```

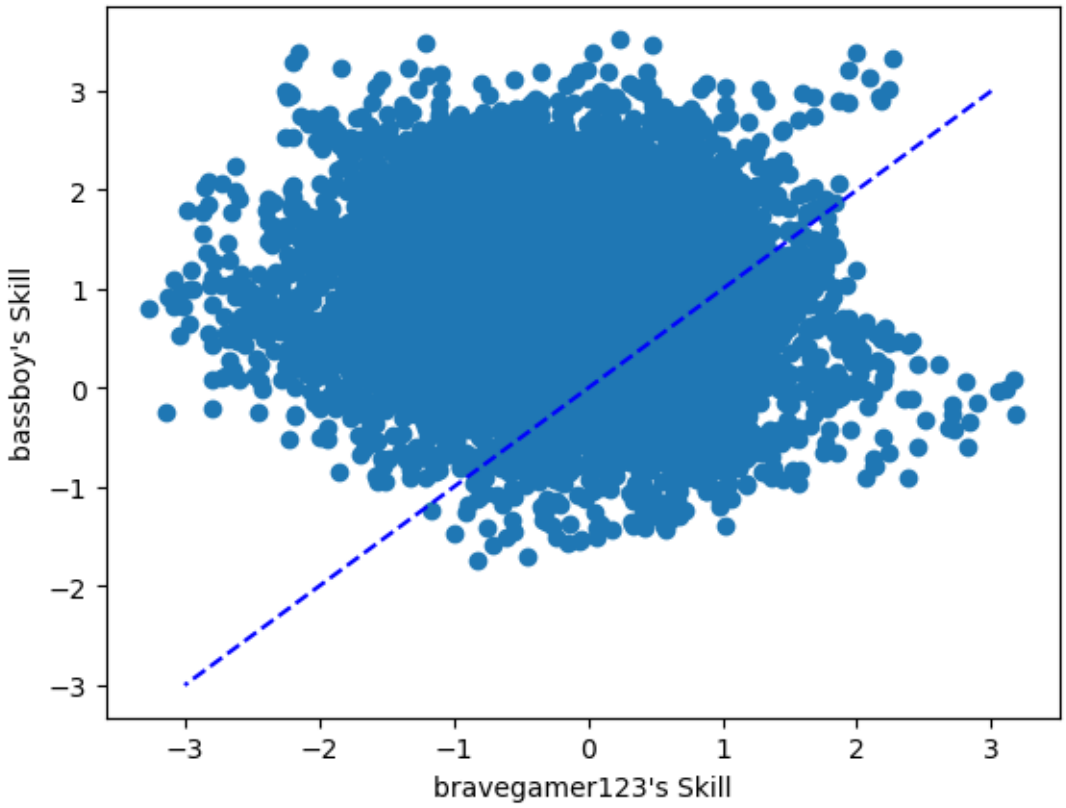
3.6.6 Q 2.4.f [2 points]

Use a scatterplot to show your samples over the joint posterior over the skills of bravegamer123 and bassboy. Include the line of equal skill. Hint: you can use `plt.scatter`.

[ ]:
```
# TODO
bravegamer123_ix = 120
bassboy_ix = 78
print(names[bravegamer123_ix])
print(names[bassboy_ix])

plt.xlabel("Player Rank")
plt.ylabel("Player Skill")
plt.plot([3, -3], [3, -3], 'b--')    # Line of equal skill
plt.scatter(all_games_samples[:, bravegamer123_ix], all_games_samples[:,␣
 ↪bassboy_ix])
plt.xlabel("bravegamer123's Skill")
plt.ylabel("bassboy's Skill")
```

```
['bravegamer123']
['bassboy']
```

[ ]:
```
Text(0, 0.5, "bassboy's Skill")
```



3.6.7 Q 2.4.g [3 points]

Using your samples, find the player that have the eleventh highest mean skill. Print an unbiased estimate of the probability that the player with the twelfth highest mean skill is not worse than bassboy, again as estimated from your samples. Hint: Probabilities of Bernoulli random variables can be written as the expectation that the Bernoulli takes value 1, so you can use simple Monte Carlo. The final formula will be very simple.

[ ]:
```
# TODO
eleventh_highest_player_ix = order[-11]
twelfth_highest_player_ix = order[-12]
print(names[eleventh_highest_player_ix])
print(torch.mean((all_games_samples[:, bassboy_ix] <= all_games_samples[:,␣
 ↪twelfth_highest_player_ix]).float()))
```

```
['vitaminex']
tensor(0.6208)
```

3.6.8 Q 2.4.h [3 points]

For any two players i and j, $p(z_i, z_j|\text{all games})$ is always proportional to $p(z_i, z_j, \text{all games})$, as a function of z_i and z_j.

In general, are the isocontours of $p(z_i, z_j|\text{all games})$ the same as those of $p(z_i, z_j|\text{games between } i \text{ and } j)$? 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.

Hint: One way to answer this is to draw the graphical model for three players, i, j, and k, and the results of games between all three pairs, and then examine conditional independencies. If you do this, include the graphical models in your assignment.

Your answer here: Yes or no?

[ ]: