Changelog: (Last Updated 2023-03-09)

Statistical Methods in ML II: Assignment 3 Solutions

- **Deadline**: 2023-03-27 (March 27th 2023)
- **Submission**: You need to submit your solutions through MarkUs, 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.

1. [40pts] Stochastic Variational Inference in the TrueSkill Model

Background

We'll continue working with TrueSkill model, a player ranking system for competitive games originally developed for Halo 2. Recall the model:

Model definition

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.

The prior:

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

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:

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

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.

In [ ]:
```
!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 import nn
from torch.distributions.normal import Normal

from functools import partial
from tqdm import trange, tqdm_notebook

import matplotlib.pyplot as plt

# Helper function
def diag_gaussian_log_density(x, mu, std):
    # axis=-1 means sum over the last dimension.
    m = Normal(mu, std)
```

```
    return torch.sum(m.log_prob(x), axis=-1)
```

```
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=9656 sha25
6=b71eec5cb4d2084a148888f948f0d9c055940f3a4a8153e54bb4d232acbbbfe7
  Stored in directory: /root/.cache/pip/wheels/8b/f1/7f/5c94f0a7a505ca1c81cd
1d9208ae2064675d97582078e6c769
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
```

Implementing the TrueSkill Model

This part was mostly done in Assignment 2. We will recall some useful functions.

a) The function `log_joint_prior` computes the log of the prior, jointly evaluated over all player's skills.

In [ ]:
```
def log_joint_prior(zs_array):
    return diag_gaussian_log_density(zs_array, torch.tensor([0.0]), torch.te
```

b) The function `logp_a_beats_b` that, given a pair of skills z_a and z_b, evaluates the log-likelihood that player with skill z_a beat player with skill z_b under the model detailed above.

To ensure numerical stability, we use the function `np.log1p` that computes $\log(1+x)$ in a numerically stable way. Or even better, use `np.logaddexp`.

In [ ]:
```
def logp_a_beats_b(z_a, z_b):
    return -torch.logaddexp(torch.tensor([0.0]), z_b - z_a)

def log_prior_over_2_players(z1, z2):
    m = Normal(torch.tensor([0.0]), torch.tensor([[1.0]]))
    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))

def log_posterior_A_beat_B(z1, z2):
    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))
```

```
def log_posterior_A_beat_B_10_times(z1, z2):
    return log_prior_over_2_players(z1, z2) + 10.0 * logp_a_beats_b(z1, z2)

def posterior_A_beat_B_10_times(z1, z2):
    return torch.exp(log_posterior_A_beat_B_10_times(z1, z2))

def log_posterior_beat_each_other_10_times(z1, z2):
    return log_prior_over_2_players(z1, z2) \
        + 10.* logp_a_beats_b(z1, z2) \
        + 10.* logp_a_beats_b(z2, z1)

def posterior_beat_each_other_10_times(z1, z2):
    return torch.exp(log_posterior_beat_each_other_10_times(z1, z2))
```

The following functions will be used for plotting. Note that `plot_2d_fun` can now take an optional second function, so you can compare two functions.

In [ ]:
```
# Plotting helper functions for free
def plot_isocontours(ax, func, xlimits=[-4, 4], ylimits=[-4, 4], steps=101,
    x = torch.linspace(*xlimits, steps=steps)
    y = torch.linspace(*ylimits, steps=steps)
    X, Y = torch.meshgrid(x, y)
    Z = func(X, Y)
    plt.contour(X, Y, Z, cmap=cmap)
    ax.set_yticks([])
    ax.set_xticks([])

def plot_2d_fun(f, x_axis_label="", y_axis_label="", f2=None, scatter_pts=No
    # 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 f2 is not None:
      plot_isocontours(ax, f2, cmap='winter')

    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()
```

1.1 [14pts] Stochastic Variational Inference on Two Players and Toy Data

One nice thing about a Bayesian approach is that it separates the model specification from the approximate inference strategy. The original Trueskill paper from 2007 used message passing.

In this question we will approximate posterior distributions with gradient-based stochastic variational inference.

The parameters are $\phi = (\mu, \log(\sigma))$. Notice that instead of σ (which is constrained to be positive), we work with $\log(\sigma)$, removing the constraint. This way, we can do unconstrained gradient-based optimization.

a) [6pts] Implement the missing lines in the below code, to complete the evidence lower bound function and the reparameterized sampler for the approximate posterior.

Hint 1: You must use the reparametrization trick in your sampler if you want your gradients to be unbiased.

Hint 2: If you're worried you got these wrong, you can check that the sampler matches the log pdf by plotting a histogram of samples against a plot of the pdf.

In [ ]:
```
def diag_gaussian_samples(mean, log_std, num_samples):
    # mean and log_std are (D) dimensional vectors
    # Return a (num_samples, D) matrix, where each sample is
    # from a diagonal multivariate Gaussian.

    # REMOVE SOLUTION
    return mean + torch.exp(log_std) * torch.randn(num_samples, mean.shape[-
    # TODO.  You might want to use torch.randn(). Remember
    # you must use the reparameterization trick.  Also remember that
    # we are parameterizing the _log_ of the standard deviation.

def diag_gaussian_logpdf(x, mean, log_std):
    # Evaluate the density of a batch of points on a
    # diagonal multivariate Gaussian. x is a (num_samples, D) matrix.
    # Return a tensor of shape (num_samples)

    # REMOVE SOLUTION
    return diag_gaussian_log_density(x, mean, torch.exp(log_std))

def batch_elbo(logprob, mean, log_std, num_samples):
    # TODO: Use simple Monte Carlo to estimate ELBO
    # on a batch of size num_samples
    # REMOVE SOLUTION
    samples = diag_gaussian_samples(mean, log_std, num_samples)
    return torch.mean(logprob(samples) - diag_gaussian_logpdf(samples, mean,
```

b) [3pts] Write a loss function called `objective` that takes variational distribution parameters, and returns an unbiased estimate of the negative elbo using `num_samples_per_iter` samples, to approximate the joint posterior over skills conditioned on observing player A winning 10 games.

Note: We want a *negative* ELBO estimate, because the convention in optimization is to minimize functions, and we want to maximize the ELBO.

In [ ]:
```
# Hyperparameters
num_players = 2
n_iters = 800
stepsize = 0.0001
num_samples_per_iter = 50

def log_posterior_A_beat_B_10_times_1_arg(z1z2):
  return log_posterior_A_beat_B_10_times(z1z2[:,0], z1z2[:,1]).flatten()

def objective(params):  # The loss function to be minimized.
  # TODO.  Hint:  This can be done in one line.
  # REMOVE SOLUTION
  return -batch_elbo(log_posterior_A_beat_B_10_times_1_arg, params[0], param
```

c) [1pt] Initialize a set of variational parameters and optimize them to approximate the joint where we observe player A winning 10 games. Report the final loss. Also plot the optimized variational approximation contours and the target distribution on the same axes.

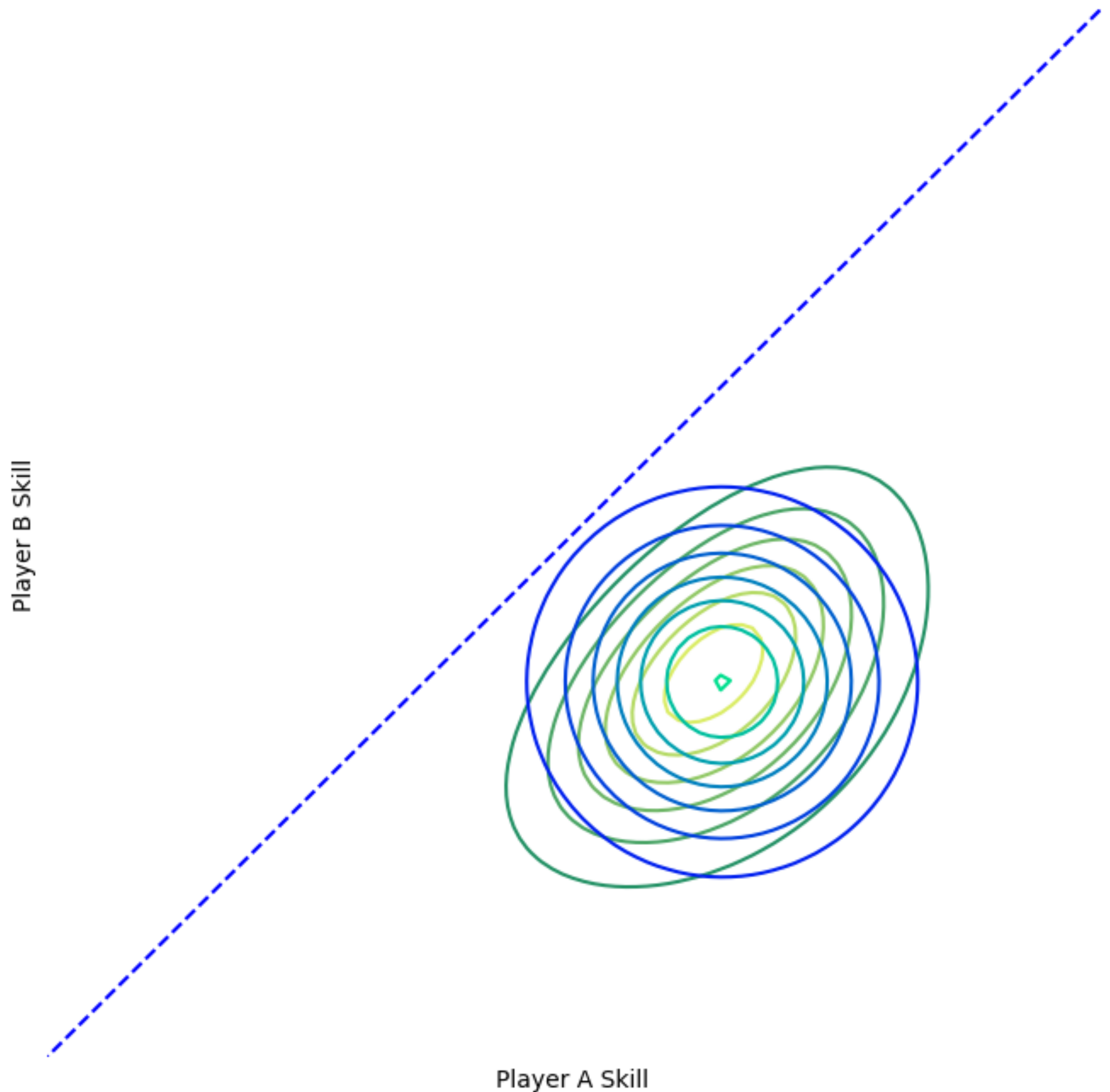
Hint: Any initialization should be fine. How many variational parameters do you need?

In [ ]:
```
def callback(params, t):
  if t % 25 == 0:
    print("Iteration {} lower bound {}".format(t, objective(params)))

# Set up optimizer.
D = 2
# init_log_std  = # TODO.
# init_mean = # TODO
# REMOVE SOLUTION
init_log_std  = torch.ones(D, requires_grad=True)
init_mean = torch.zeros(D, requires_grad=True)
params = (init_mean, init_log_std)
optimizer = torch.optim.SGD(params, lr=stepsize, momentum=0.9)

def update():
    optimizer.zero_grad()
    loss = objective(params)
```

```
    loss.backward()
    optimizer.step()

# Main loop.
print("Optimizing variational parameters...")
for t in trange(0, n_iters):
    update()
    callback(params, t)


def approx_posterior_2d(z1, z2):
    # The approximate posterior
    mean, logstd = params[0].detach(), params[1].detach()
    return torch.exp(diag_gaussian_logpdf(torch.stack([z1, z2], dim=2), mear

plot_2d_fun(posterior_A_beat_B_10_times, "Player A Skill", "Player B Skill",
            f2=approx_posterior_2d)
```

```
Optimizing variational parameters...
  0%|          | 1/800 [00:00<02:16,  5.87it/s]
Iteration 0 lower bound 27.328237533569336
  3%|▎         | 21/800 [00:00<00:08, 93.39it/s]
Iteration 25 lower bound 11.493878364562988
  9%|▉         | 73/800 [00:00<00:03, 199.86it/s]
Iteration 50 lower bound 11.567449569702148
Iteration 75 lower bound 5.994995594024658
 12%|█▏        | 97/800 [00:00<00:03, 212.03it/s]
Iteration 100 lower bound 6.4192986488342285
 20%|██        | 162/800 [00:00<00:02, 273.94it/s]
Iteration 125 lower bound 6.423530101776123
Iteration 150 lower bound 5.441266059875488
Iteration 175 lower bound 5.085821151733398
 27%|██▋       | 219/800 [00:00<00:02, 271.66it/s]
Iteration 200 lower bound 3.814467430114746
Iteration 225 lower bound 3.722540855407715
 31%|███       | 247/800 [00:01<00:02, 267.79it/s]
Iteration 250 lower bound 4.256308078765869
 39%|███▊      | 309/800 [00:01<00:01, 289.59it/s]
Iteration 275 lower bound 3.13362455368042
Iteration 300 lower bound 4.327754974365234
Iteration 325 lower bound 3.589164972305298
 46%|████▌     | 368/800 [00:01<00:01, 253.53it/s]
Iteration 350 lower bound 3.152064800262451
 49%|████▉     | 395/800 [00:01<00:01, 248.32it/s]
Iteration 375 lower bound 3.117180824279785
Iteration 400 lower bound 3.000610589981079
 53%|█████▎    | 424/800 [00:01<00:01, 258.21it/s]
```

```
Iteration 425 lower bound 3.125608444213867
 56%|█████▋    | 451/800 [00:01<00:01, 212.72it/s]
Iteration 450 lower bound 2.9839229583740234
 59%|█████▉    | 475/800 [00:02<00:01, 218.20it/s]
Iteration 475 lower bound 3.198651075363159
 62%|██████▏   | 499/800 [00:02<00:01, 208.62it/s]
Iteration 500 lower bound 3.5829315185546875
 70%|███████   | 560/800 [00:02<00:00, 251.49it/s]
Iteration 525 lower bound 2.7823500633239746
Iteration 550 lower bound 2.9266357421875
Iteration 575 lower bound 2.8685829639434814
 78%|███████▊  | 621/800 [00:02<00:00, 276.34it/s]
Iteration 600 lower bound 3.080533504486084
Iteration 625 lower bound 2.9182214736938477
 84%|████████▍ | 676/800 [00:02<00:00, 219.39it/s]
Iteration 650 lower bound 2.9631388187408447
Iteration 675 lower bound 2.902109146118164
 91%|█████████▏| 731/800 [00:03<00:00, 244.25it/s]
Iteration 700 lower bound 2.907780408859253
Iteration 725 lower bound 3.1057589054107666
Iteration 750 lower bound 3.0081300735473633
100%|██████████| 800/800 [00:03<00:00, 232.72it/s]
Iteration 775 lower bound 3.001633405685425
/usr/local/lib/python3.10/dist-packages/torch/functional.py:507: UserWarnin
g: torch.meshgrid: in an upcoming release, it will be required to pass the i
ndexing argument. (Triggered internally at ../aten/src/ATen/native/TensorSha
pe.cpp:3549.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```




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

d) [2pts] Write a loss function called `objective` that takes variational distribution parameters , and returns a negative ELBO estimate using simple Monte Carlo with `num_samples_per_iter` samples, to approximate the joint distribution where we observe player A winning 10 games and player B winning 10 games.

Hint: You can find analogous functions in the code above.

In [ ]:
```
# Hyperparameters
n_iters = 100
stepsize = 0.0001
num_samples_per_iter = 50
```

```
def log_posterior_beat_each_other_10_times_1_arg(z1z2):
    # z1z2 is a tensor with shape (num_samples x 2)
    # Return a tensor with shape (num_samples)
    # REMOVE SOLUTION
    return log_posterior_beat_each_other_10_times(z1z2[:,0], z1z2[:,1]).flat

def objective(params):
    # REMOVE SOLUTION
    return -batch_elbo(log_posterior_beat_each_other_10_times_1_arg, params[
```

e) [2pts] Run the code below to optimize, and report the final loss. Also plot the optimized variational approximation contours and the target distribution on the same axes.

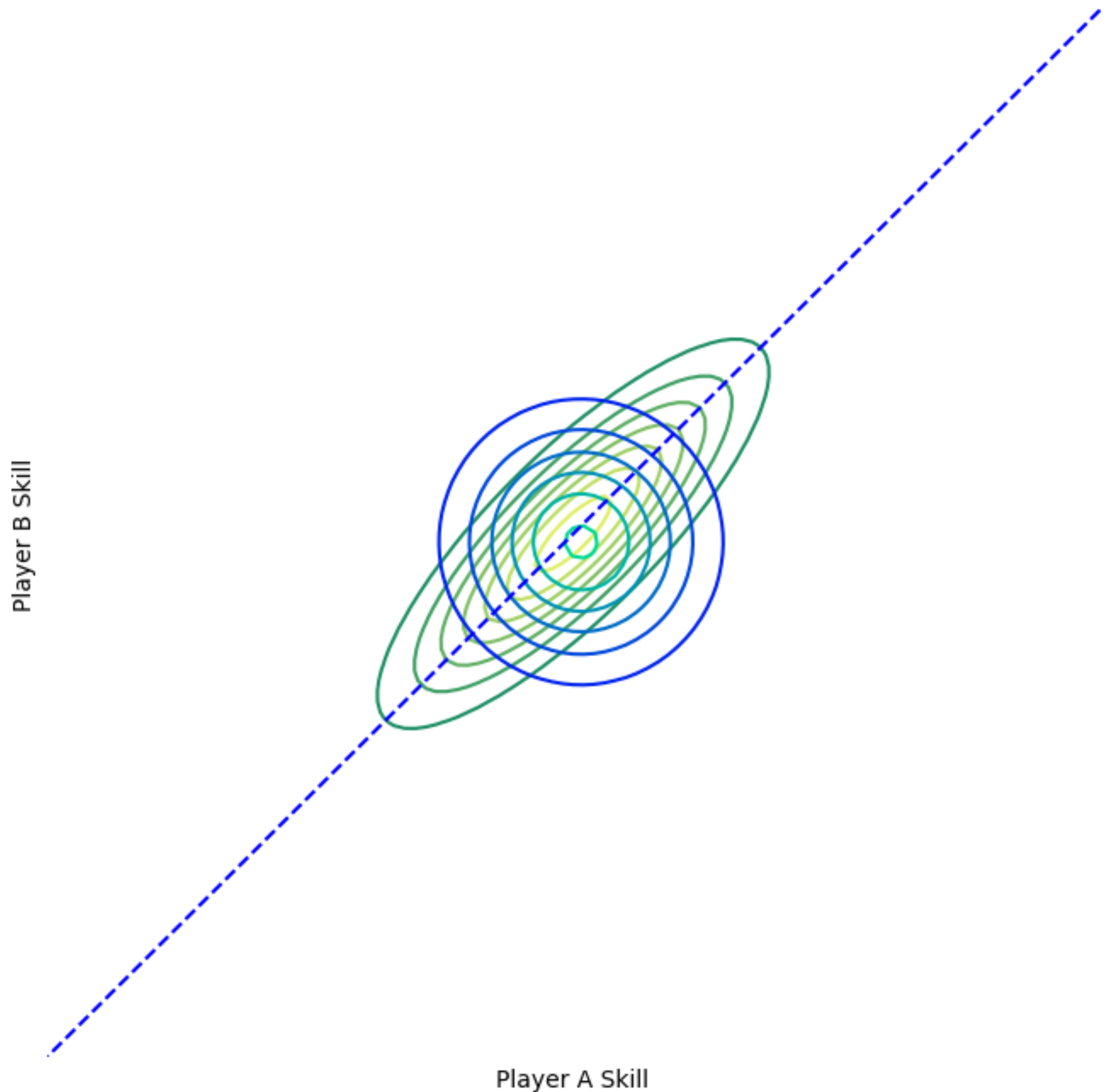
Write one or two sentences describing the joint settings of skills that are plausible under the true posterior, but which are not plausible under the approximate posterior.

Finally, answer with one or two sentences: Would changing the variational approximate posterior from a fully-factorized (diagonal covariance) Gaussian to a non-factorized (fully parameterized covariance) Gaussian make a better approximation in this instance?

In [ ]:

```
# Main loop.
init_log_std  = torch.ones(D, requires_grad=True)
init_mean = torch.zeros(D, requires_grad=True)

print("Optimizing variational parameters...")
for t in trange(0, n_iters):
    update()
    callback(params, t)

plot_2d_fun(posterior_beat_each_other_10_times, "Player A Skill", "Player B
            f2=approx_posterior_2d)
```

```
Optimizing variational parameters...
 14%|█         | 14/100 [00:00<00:00, 137.30it/s]
Iteration 0 lower bound 16.687257766723633
Iteration 25 lower bound 16.49291229248047
 76%|███████▋  | 76/100 [00:00<00:00, 179.70it/s]
Iteration 50 lower bound 15.672494888305664
Iteration 75 lower bound 16.054340362548828
100%|██████████| 100/100 [00:00<00:00, 184.20it/s]
```



<Figure size 640x480 with 0 Axes>

1.2 [26pts] Approximate inference conditioned on real data

The dataset contains data on 2546 chess games amongst 1434 players:

- 'names' is a 1434 by 1 matrix, whose i'th entry is the name of player i.
- 'games' is a 2546 by 2 matrix of game outcomes (actually chess matches), one row per game.

The first column of 'games' contains the indices of the players who won. The second

column of 'games' contains the indices of the player who lost.

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

In [ ]:
```
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(
```

a) [0pt] Assuming all game outcomes are i.i.d. conditioned on all players' skills, the function `log_games_likelihood` below takes a batch of player skills `zs` and a collection of observed games `games` and gives the total log-likelihood for all those observations given all the skills. (You do not need to code anything here.)

In [ ]:
```
def log_games_likelihood(zs, games):
    winning_player_ixs = games[:,0]
    losing_player_ixs = games[:,1]

    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_sk
    return torch.sum(log_likelihoods, dim=1)
```

In [ ]:
```
def log_joint_probability(zs):
    return log_joint_prior(zs) + log_games_likelihood(zs, games)
```

b) [2pt] Write a new objective function like the one from the previous question.

Below, we initialize a variational distribution and fit it to the joint distribution with all the observed tennis games from the dataset.

In [ ]:
```
# Hyperparameters
num_players = 1163
n_iters = 500
stepsize = 0.0001
num_samples_per_iter = 150

def objective(params):
    # REMOVE SOLUTION
    return -batch_elbo(log_joint_probability, params[0], params[1], num_samp
```

c) [2pts] Optimize, and report the final loss.

In [ ]:
```
# Set up optimizer.
```

```
init_mean = torch.zeros(num_players, requires_grad=True)
init_log_std  = torch.zeros(num_players, requires_grad=True)
params = (init_mean, init_log_std)
optimizer = torch.optim.SGD(params, lr=stepsize, momentum=0.9)

def update():
    optimizer.zero_grad()
    loss = objective(params)
    loss.backward()
    optimizer.step()

# Optimize and print loss in a loop
# HINT: you can use the callback() function to report loss
# REMOVE SOLUTION
print("Optimizing variational parameters...")
for t in trange(0, n_iters):
    update()
    callback(params, t)

callback(params, n_iters)
```

```
Optimizing variational parameters...
  0%|          | 2/500 [00:00<00:25, 19.16it/s]
Iteration 0 lower bound 2290.423583984375
  6%|▌         | 30/500 [00:00<00:13, 35.75it/s]
Iteration 25 lower bound 2201.52294921875
 11%|█         | 56/500 [00:01<00:11, 38.82it/s]
Iteration 50 lower bound 2080.553466796875
 16%|█▌        | 80/500 [00:02<00:12, 35.00it/s]
Iteration 75 lower bound 1996.75732421875
 21%|██        | 105/500 [00:02<00:11, 35.42it/s]
Iteration 100 lower bound 1931.35205078125
 26%|██▌       | 129/500 [00:03<00:10, 34.13it/s]
Iteration 125 lower bound 1881.376220703125
 31%|███▏      | 157/500 [00:04<00:10, 34.01it/s]
Iteration 150 lower bound 1840.0404052734375
 37%|███▋      | 184/500 [00:05<00:08, 37.86it/s]
Iteration 175 lower bound 1807.953369140625
 41%|████      | 205/500 [00:05<00:08, 36.75it/s]
Iteration 200 lower bound 1767.9404296875
 46%|████▌     | 230/500 [00:06<00:07, 35.70it/s]
Iteration 225 lower bound 1749.2520751953125
 51%|█████     | 254/500 [00:07<00:08, 27.94it/s]
Iteration 250 lower bound 1727.832763671875
 56%|█████▌    | 278/500 [00:08<00:08, 24.88it/s]
Iteration 275 lower bound 1713.6834716796875
```

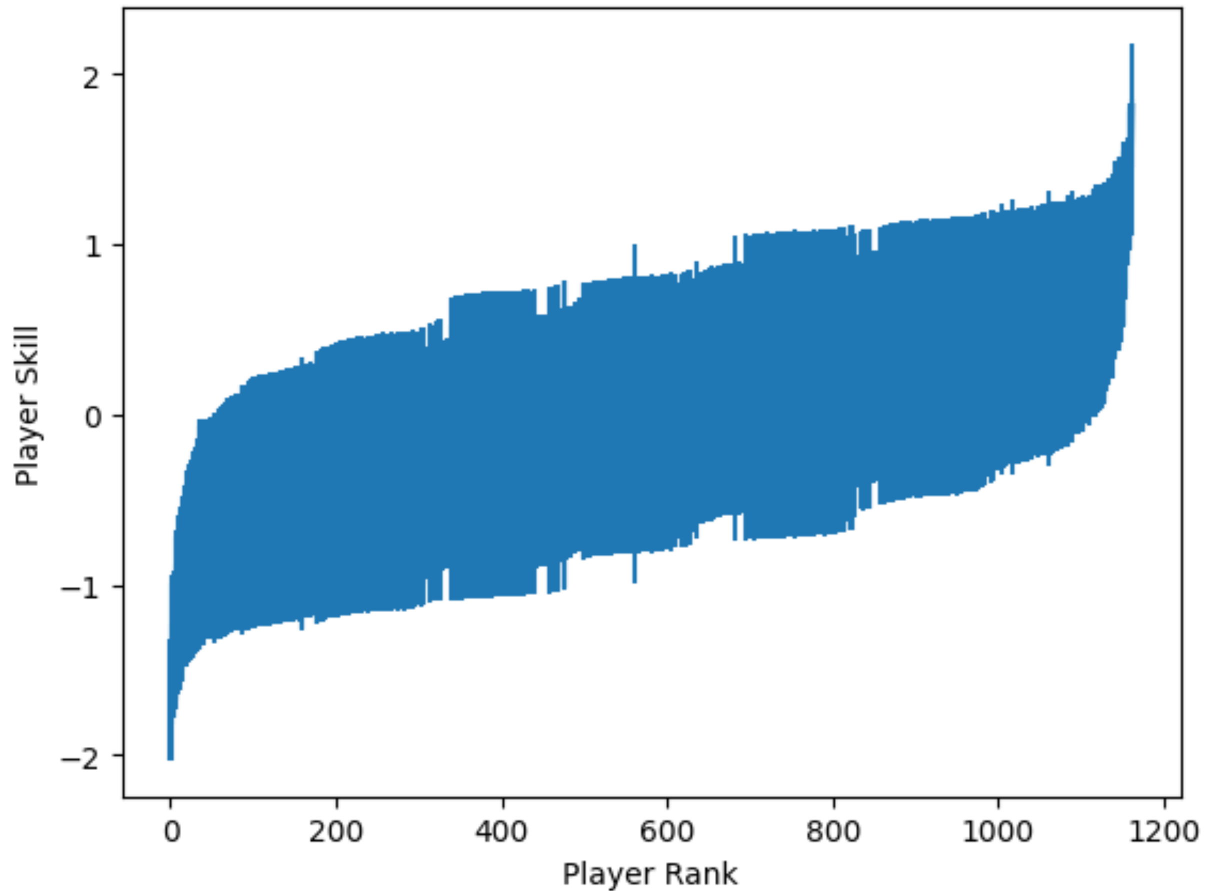
```
 61%|██████    | 305/500 [00:09<00:07, 24.97it/s]
Iteration 300 lower bound 1696.6085205078125
 66%|██████▌   | 331/500 [00:10<00:04, 34.77it/s]
Iteration 325 lower bound 1680.5787353515625
 71%|███████   | 356/500 [00:10<00:03, 36.57it/s]
Iteration 350 lower bound 1664.3497314453125
 76%|███████▌  | 380/500 [00:11<00:03, 35.26it/s]
Iteration 375 lower bound 1651.856201171875
 82%|████████▏ | 408/500 [00:12<00:02, 35.41it/s]
Iteration 400 lower bound 1644.0809326171875
 86%|████████▋ | 432/500 [00:12<00:01, 36.01it/s]
Iteration 425 lower bound 1637.434326171875
 92%|█████████▏| 459/500 [00:13<00:01, 39.17it/s]
Iteration 450 lower bound 1630.0926513671875
 96%|█████████▌| 480/500 [00:14<00:00, 37.06it/s]
Iteration 475 lower bound 1623.0010986328125
100%|██████████| 500/500 [00:14<00:00, 34.18it/s]
Iteration 500 lower bound 1617.30029296875
```

d) [1pt] Plot the approximate mean and variance of all players, sorted by skill.

In [ ]:

```
# mean_skills, logstd_skills = # TODO.  Hint: You don't need to do simple Mo
# Hint: You should use .detach() before you do anything with the params tens
# REMOVE SOLUTION
mean_skills, logstd_skills = params[0].detach(), params[1].detach()
variance = torch.square(torch.exp(logstd_skills))
order = torch.argsort(mean_skills)

plt.xlabel("Player Rank")
plt.ylabel("Player Skill")
plt.errorbar(range(num_players), mean_skills[order], variance[order])
```

Out[ ]: <ErrorbarContainer object of 3 artists>



e) [1pt] List the names of the 10 players with the highest mean skill under the variational model.

In [ ]:
```
for i in range(1,11):
    # REMOVE SOLUTION
    print(names[order[-i]]) # TODO
```

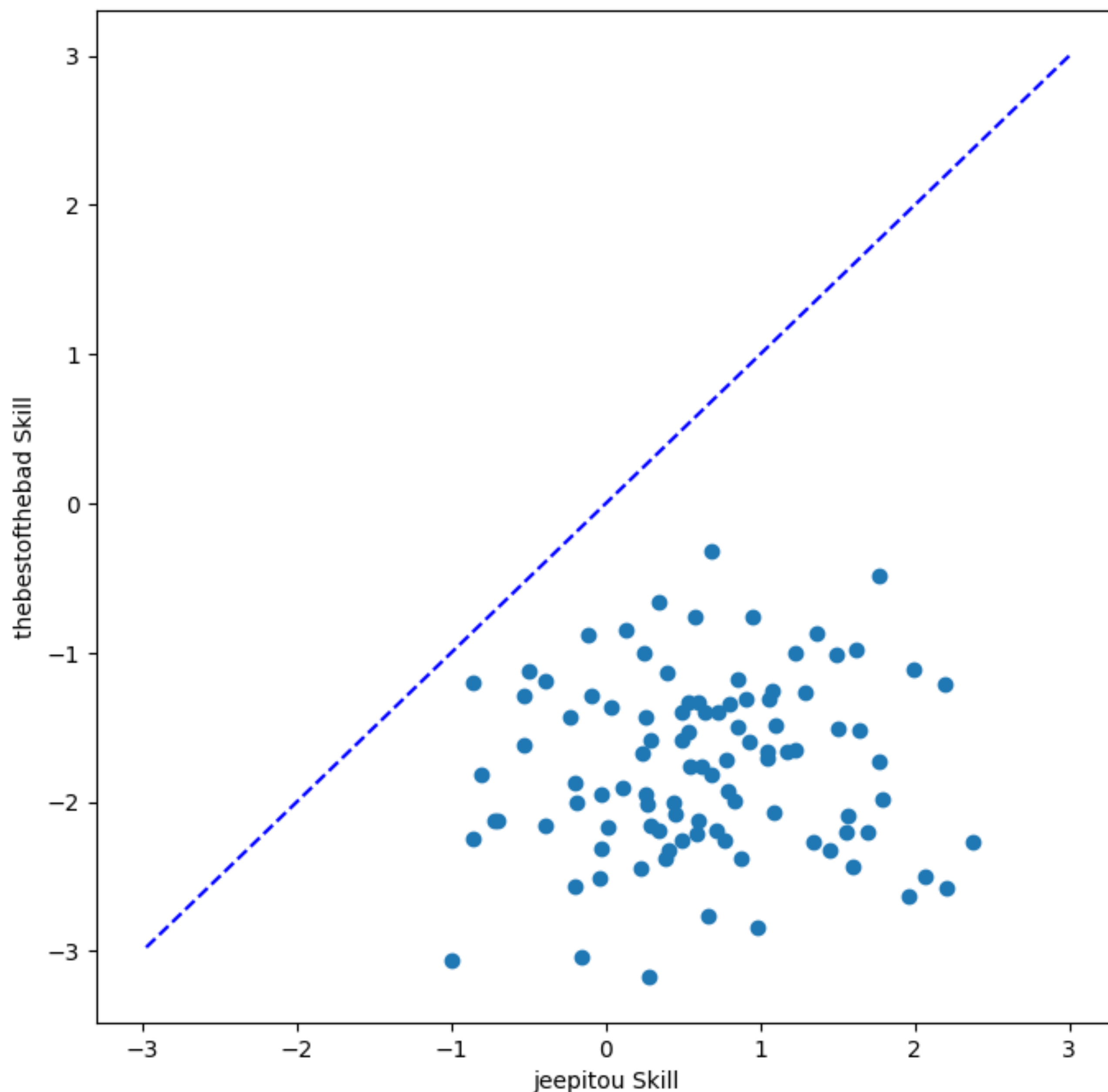
```
['doraemon61']
['smartduckduckcow']
['projetoxadrez']
['mrzoom47']
['chesswithmom']
['lance5500']
['lzchips']
['chess-brahs']
['cdvh']
['chiggen']
```

f) [3pt] Plot samples from the joint posterior over the skills of 'lelik3310' and 'thebestofthebad'. Based on your samples, describe in a sentence the relationship between the skills of the players. (Is one better than the other? Are they approximately even?)

In [ ]:
```
lelik3310_ix = np.where(names=='lelik3310')[0].item()
thebestofthebad_ix = np.where(names=='thebestofthebad')[0].item()

print(lelik3310_ix, thebestofthebad_ix)
print(names[lelik3310_ix])
print(names[thebestofthebad_ix])

fig = plt.figure(figsize=(8,8), facecolor='white')

# Label each with "<player> Skill"
plt.xlabel("jeepitou Skill")
plt.ylabel("thebestofthebad Skill")

plt.plot([3, -3], [3, -3], 'b--') # Line of equal skill

samples = diag_gaussian_samples(mean_skills, logstd_skills, 100)

# TODO:  Hint:  Use plt.scatter()
# REMOVE SOLUTION
plt.scatter(samples[:, lelik3310_ix], samples[:, thebestofthebad_ix])
```

```
568 1051
['lelik3310']
['thebestofthebad']
```

Out[ ]:
```
<matplotlib.collections.PathCollection at 0x7d630a27c250>
```



g) [6pts] Derive the exact probability under a factorized Gaussian over two players' skills that one has higher skill than the other, as a function of the two means and variances over their skills. Express your answer in terms of the cumulative distribution function of a one-dimensional Gaussian random variable.

- Hint 1: Use a linear change of variables $y_A, y_B = z_A - z_B, z_B$. What does the line of equal skill look like after this transformation?
- Hint 2: If $X \sim \mathcal{N}(\mu, \Sigma)$, then $AX \sim \mathcal{N}(A\mu, A\Sigma A^\top)$ where A is a linear transformation.
- Hint 3: Marginalization in Gaussians is easy: if $X \sim \mathcal{N}(\mu, \Sigma)$, then the ith element of X has a marginal distribution $X_i \sim \mathcal{N}(\mu_i, \Sigma_{ii})$.

Suppose $p(z_A > z_B) = p(z_A - z_B)$. Consider the following variables.

$$y_A = z_A - z_B$$
$$y_B = z_B$$
$$A = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

Since $z \sim \mathcal{N}(\mu, \Sigma)$, we have the following

$$\begin{aligned} y &\sim \mathcal{N}(A\mu, A\Sigma A^T) \\ \begin{bmatrix} y_A \\ y_B \end{bmatrix} &\sim \mathcal{N}\left(\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma_2^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}^T \right) \\ &\sim \mathcal{N}\left(\begin{bmatrix} \mu_1 - \mu_2 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \sigma_1^2 + \sigma_2^2 & -\sigma_2^2 \\ -\sigma_2^2 & \sigma_2^2 \end{bmatrix}^T \right) \\ y_A &\sim \mathcal{N}(\mu_1 - \mu_2, \sigma_1^2 + \sigma_2^2) \end{aligned}$$

Then infer the following.

$$\begin{aligned} p(y_A > 0) &= 1 - p(y_A < 0) \\ &= 1 - \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(\frac{\mu - x}{2\sigma^2}\right) dx \\ &= 1 - \Phi\left(\frac{\mu_2 - \mu_1}{\sqrt{\sigma_1^2 + \sigma_2^2}}\right) \\ &= \Phi\left(\frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}}\right) \end{aligned}$$

h) [3pts] Compute the probability under your approximate posterior that lelik3310 has higher skill than thebestofthebad. Compute this quantity exactly using the formula you just derived above, and also estimate it using simple Monte Carlo with 10000 examples.

Hint: You might want to use `Normal(0,1).cdf()` for the exact formula.

In [ ]:
```
# TODO
# REMOVE SOLUTION
def prob_A_superior_B(N, A_ix, B_ix):
    mu_diff = mean_skills[A_ix] - mean_skills[B_ix]
    formula_est = Normal(0,1).cdf(torch.tensor([mu_diff/torch.sqrt(variance[
```

```
    samples = diag_gaussian_samples(mean_skills, logstd_skills, N)
    A_wins = samples[:,A_ix] > samples[:,B_ix]
    mc_est = np.count_nonzero(A_wins) / samples.shape[0]

    return formula_est, mc_est

formula_est, mc_est = prob_A_superior_B(10000, lelik3310_ix, thebestofthebac
print(f"Exact CDF Estimate: {formula_est}")
print(f"Simple MC Estimate: {mc_est}")

formula_est, mc_est = prob_A_superior_B(10000, lelik3310_ix, thebestofthebac
print(f"Exact CDF Estimate: {formula_est}")
print(f"Simple MC Estimate: {mc_est}")
```

```
Exact CDF Estimate: tensor([0.9882])
Simple MC Estimate: 0.9884
Exact CDF Estimate: tensor([0.9882])
Simple MC Estimate: 0.9878
```

i) [2pts] Compute the probability that lelik3310 is better than the player with the 5th lowest mean skill. Compute this quantity exactly, and then estimate it using simple Monte Carlo with 10000 examples.

In [ ]:
```
# TODO
# REMOVE SOLUTION
fifth_worst_ix = order[4]
print(fifth_worst_ix)
cdf_est, mc_est = prob_A_superior_B(10000, lelik3310_ix, fifth_worst_ix)
print(f"Exact CDF Estimate: {cdf_est}")
print(f"Simple MC Estimate: {mc_est}")
```

```
tensor(935)
Exact CDF Estimate: tensor([0.9716])
Simple MC Estimate: 0.9686
```

j) [3pts] Imagine that we knew ahead of time that we were examining the skills of top chess players, and so changed our prior on all players to Normal(10, 1) and re-ran our approximate inference from scratch. Would that change the answer of either of the previous 2 questions, in expectation?

Your answer here.

REMOVE SOLUTION

Shifting all the priors to 10 might change the initialization environment. But it would be unlikely to affect the relative skills between players that are close, especially if they're at

the lower or upper extrema of the spectrum. Relative skills shouldn't see a difference in either CDF or MC estimates.

k) [3pts] Based on all the plots and results in this assignment and HW2, which approximate inference method do you suspect is producing a better overall approximation to the true posterior over all skills conditioned on all games? Give a short explanation.

Your answer here.

REMOVE SOLUTION

As discussed earlier, the SVI model's diagonal covariance matrix doesn't represent relative skills between players very well. Even the trained approximations don't model the true posterior distribution well. Recalling back to the HMC model in HW2, it's apparent that it is more effective at approximating the true posterior.

2. [22pts] Question 2: Variational Auto Encoder (VAE) with synthetic data

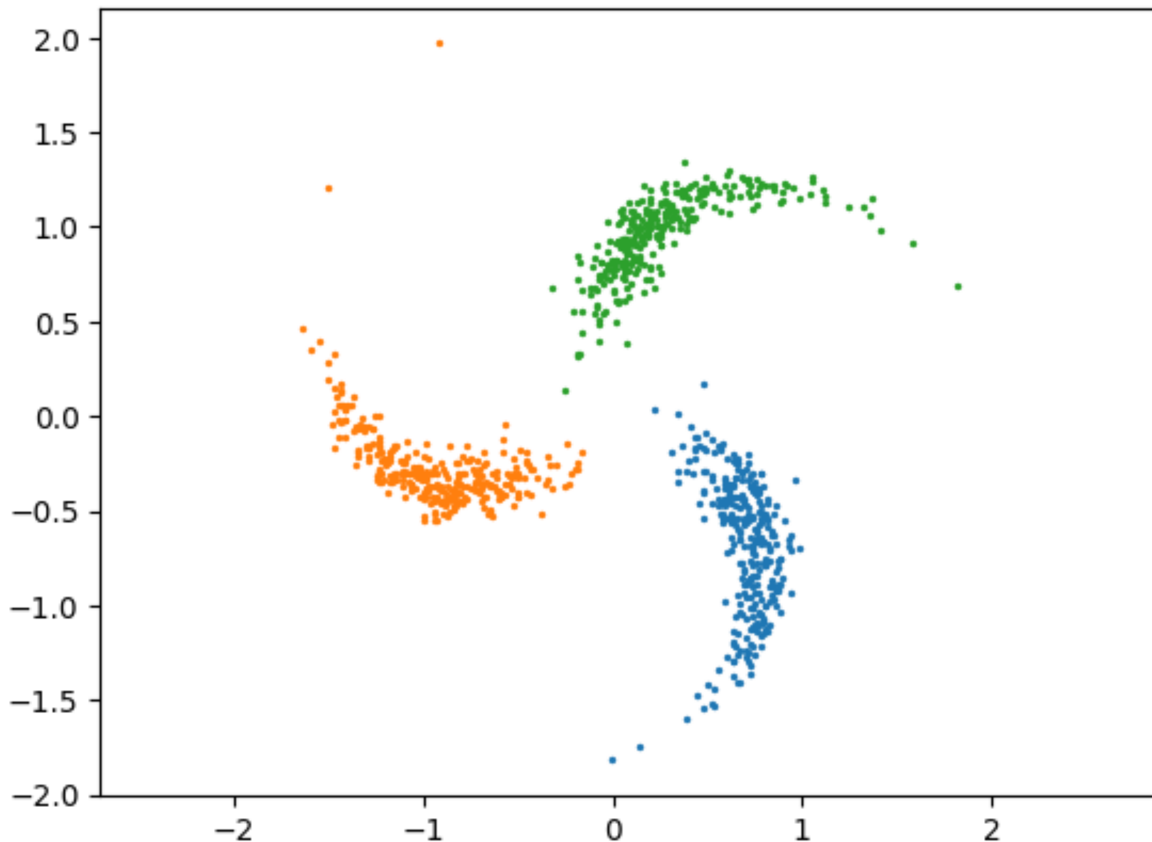
In this question, we will train a VAE on a synthetic data which resembles spirals in 2d. The code below generates the synthetic spiral data and visualizes the data generated. Notice that there are 3 clusters in the input space, each colored with a different color. The VAE will not see the cluster assignments, but we hope to recover this structure in the latent space.

In [ ]:
```
# Code to generate the pinwheel dataset.
# Taken from [Johnson et al (2016)], updated by Zhao & Linderman.
def make_pinwheel_data(radial_std, tangential_std, num_classes, num_per_clas
    rads = torch.linspace(0, 2*torch.pi, num_classes + 1)

    features = torch.randn(num_classes*num_per_class, 2) * torch.tensor([rac
    print(features)
    features[:, 0] = features[:, 0] + 1.0
    labels = torch.repeat_interleave(torch.arange(num_classes), num_per_clas

    angles = rads[labels] + rate * torch.exp(features[:,0])
    rotations = torch.stack([np.cos(angles), -np.sin(angles), np.sin(angles)
    rotations = torch.reshape(rotations.T, (-1, 2, 2))

    perm_ix = torch.randperm(labels.shape[0])
    return labels[perm_ix], torch.einsum('ti,tij->tj', features, rotations)[
```

```
num_clusters = 3
samples_per_cluster = 300
labels, data = make_pinwheel_data(0.3, 0.1, num_clusters, samples_per_cluste

for k in range(num_clusters):
    plt.scatter(data[labels == k, 0], data[labels == k, 1], s=2)

plt.axis("equal")
```

```
tensor([[-0.2623, -0.0052],
        [ 0.3945,  0.0854],
        [-0.0944, -0.1206],
        ...,
        [ 0.4239, -0.2133],
        [ 0.0347, -0.1301],
        [-0.0747,  0.0777]])
```

Out[ ]:
```
(-1.8121850848197938,
 1.9869072794914246,
 -2.007331818342209,
 2.1582182347774506)
```



2.1 [12pts] Implement the missing lines in the below code, to complete the `elbo` function for a variational

autoencoder.

The model we consider is as follows:

- **Prior:** The prior over each items's latent representation is a multivariate standard normal distribution. We'll set the dimension of the latent space to 2.
- **Likelihood:** Given the latent representation z for a point, its likelihhod distribution is two-dimensional Gaussian distribution, whose mean and variance are given by the output of a neural network $f_\theta(z)$:

  $$x|(z,\theta) \sim \mathcal{N}\left(f_\theta(z)_{mean}, f_\theta(z)_{var}\right)$$

  The neural network f_θ is called the decoder, and its parameters θ will be optimized to fit the data.

In [ ]:
```
# Generic VAE functions.

def log_prior(zs_array):
    return diag_gaussian_log_density(zs_array, torch.tensor([0.0]), torch.te

def diag_gaussian_samples(mean, log_std, num_samples):
    return mean + torch.exp(log_std) * torch.randn(num_samples, mean.shape[-

def diag_gaussian_logpdf(x, mean, log_std):
    return diag_gaussian_log_density(x, mean, torch.exp(log_std))


def batch_elbo(  # Simple Monte Carlo estimate of the variational lower bour
    recognition_net,    # takes a batch of datapoints, outputs mean and log_
    decoder_net,        # takes a batch of latent samples, outputs mean and 
    log_joint,          # takes decoder_net, a batch of latent samples, and 
    data                # a.k.a. x
    ):
    # REMOVE SOLUTION
    q_params = recognition_net(data)  # TODO. Call recognition net.
    samples = diag_gaussian_samples(*q_params, len(data)) # TODO.  Sample z 
    log_joint_value = log_joint(decoder_net, samples, data) # TODO. Call dec
    log_post_value = diag_gaussian_logpdf(samples, *q_params)  # TODO.  Eval
    elbo_hat = log_joint_value - log_post_value  # TODO. Produce an unbiasec
    return torch.mean(elbo_hat)
```

The below code trains a VAE where the encoder and decoder are both neural networks. The parameters are specified in the starter code. You don't need to do anything here, this is just to help you debug.

In [ ]:
```
from torch.utils.data import TensorDataset, DataLoader
# Now define a specific VAE for the spiral dataset

data_dimension = 2
latent_dimension = 2

# Define the recognition network.
class RecognitionNet(nn.Module):
    def __init__(self, data_dimension, latent_dimension):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(data_dimension, 100),
            nn.ReLU(),
            nn.Linear(100, 50),
            nn.ReLU()
        )
        self.mean_net = nn.Linear(50, latent_dimension) # Output mean of q(z
        self.std_net = nn.Linear(50, latent_dimension) # Output log_std of q

    def forward(self, x):
        interm = self.net(x)
        mean, log_std = self.mean_net(interm), self.std_net(interm)
        return mean, log_std

recognition_net = RecognitionNet(data_dimension, latent_dimension)

# Define the decoder network.
# Note that it has two outputs, a mean and a variance, because
# this model has a Gaussian likelihood p(x|z).
class Decoder(nn.Module):
    def __init__(self, latent_dimension, data_dimension):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(latent_dimension, 100),
            nn.ReLU(),
            nn.Linear(100, 50),
            nn.ReLU()
        )
        self.mean_net = nn.Linear(50, data_dimension) # Output mean of p(x|z
        self.std_net = nn.Linear(50, data_dimension) # Output log_std of p(x

    def forward(self, x):
        interm = self.net(x)
        mean, log_std = self.mean_net(interm), self.std_net(interm)
        return mean, log_std

decoder_net = Decoder(latent_dimension, data_dimension)

# Set up log likelihood function.
def log_likelihood(decoder_net, latent, data):
```

```
    mean, log_std = decoder_net(latent)
    return diag_gaussian_logpdf(data, mean,
                                np.log(0.1) + 0. * log_std)  # Note: we are

def log_joint(decoder_net, latent, data):
    return log_prior(latent) + log_likelihood(decoder_net, latent, data)

# Run optimization
optimizer = torch.optim.Adam([{'params': recognition_net.parameters()},
                              {'params': decoder_net.parameters()}], lr=1e-3)
n_iters = 2000
minibatch_size = 300

dataset = TensorDataset(torch.tensor(data))
dataloader = DataLoader(dataset, batch_size=minibatch_size, shuffle=True)

def objective(recognition_net, decoder_net):  # The loss function to be mini
    minibatch = next(iter(dataloader))[0]
    return -batch_elbo(
    recognition_net,
    decoder_net,
    log_joint,
    minibatch)

def callback(t):
    if t % 100 == 0:
        print("Iteration {} lower bound {}".format(t, -objective(recognitior

def update():
    optimizer.zero_grad()
    loss = objective(recognition_net, decoder_net)
    loss.backward()
    optimizer.step()

# Main loop.
print("Optimizing variational parameters...")
for t in trange(0, n_iters):
    update()
    callback(t)
```

```
<ipython-input-5-26fe4ed13568>:64: UserWarning: To copy construct from a ten
sor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.
clone().detach().requires_grad_(True), rather than torch.tensor(sourceTenso
r).
  dataset = TensorDataset(torch.tensor(data))
Optimizing variational parameters...
  0%|          | 1/2000 [00:00<03:35,  9.27it/s]
Iteration 0 lower bound -58.10840606689453
  6%|▌         | 118/2000 [00:01<00:17, 106.10it/s]
```

```
Iteration 100 lower bound -2.4775567054748535
 11%|█         | 220/2000 [00:02<00:14, 126.20it/s]
Iteration 200 lower bound -1.9697253704071045
 16%|█▌        | 314/2000 [00:02<00:13, 127.81it/s]
Iteration 300 lower bound -1.739956021308899
 21%|██        | 422/2000 [00:03<00:12, 130.17it/s]
Iteration 400 lower bound -1.4823447465896606
 26%|██▌       | 520/2000 [00:04<00:14, 104.58it/s]
Iteration 500 lower bound -1.5135951042175293
 31%|███       | 615/2000 [00:05<00:13, 105.51it/s]
Iteration 600 lower bound -1.403287410736084
 36%|███▌      | 720/2000 [00:06<00:09, 131.38it/s]
Iteration 700 lower bound -1.4258238077163696
 41%|████      | 815/2000 [00:07<00:14, 79.01it/s]
Iteration 800 lower bound -1.307396650314331
 45%|████▌     | 908/2000 [00:08<00:16, 66.39it/s]
Iteration 900 lower bound -1.2096158266067505
 51%|█████     | 1018/2000 [00:09<00:09, 108.41it/s]
Iteration 1000 lower bound -1.3110040426254272
 56%|█████▌    | 1118/2000 [00:10<00:08, 103.25it/s]
Iteration 1100 lower bound -1.19325590133667
 61%|██████    | 1220/2000 [00:12<00:07, 100.14it/s]
Iteration 1200 lower bound -1.1395703554153442
 66%|██████▋   | 1325/2000 [00:13<00:05, 114.97it/s]
Iteration 1300 lower bound -1.101892113685608
 71%|███████   | 1416/2000 [00:13<00:04, 121.51it/s]
Iteration 1400 lower bound -1.1155719757080078
 76%|███████▌  | 1522/2000 [00:14<00:03, 122.04it/s]
Iteration 1500 lower bound -1.2160181999206543
 81%|████████  | 1619/2000 [00:15<00:03, 111.64it/s]
Iteration 1600 lower bound -1.1516125202178955
 86%|████████▌ | 1723/2000 [00:16<00:02, 123.08it/s]
Iteration 1700 lower bound -1.066819667816162
 91%|█████████ | 1816/2000 [00:17<00:01, 126.27it/s]
Iteration 1800 lower bound -1.0098801851272583
 96%|█████████▌| 1921/2000 [00:18<00:00, 125.40it/s]
Iteration 1900 lower bound -1.0342038869857788
100%|██████████| 2000/2000 [00:18<00:00, 106.69it/s]
```

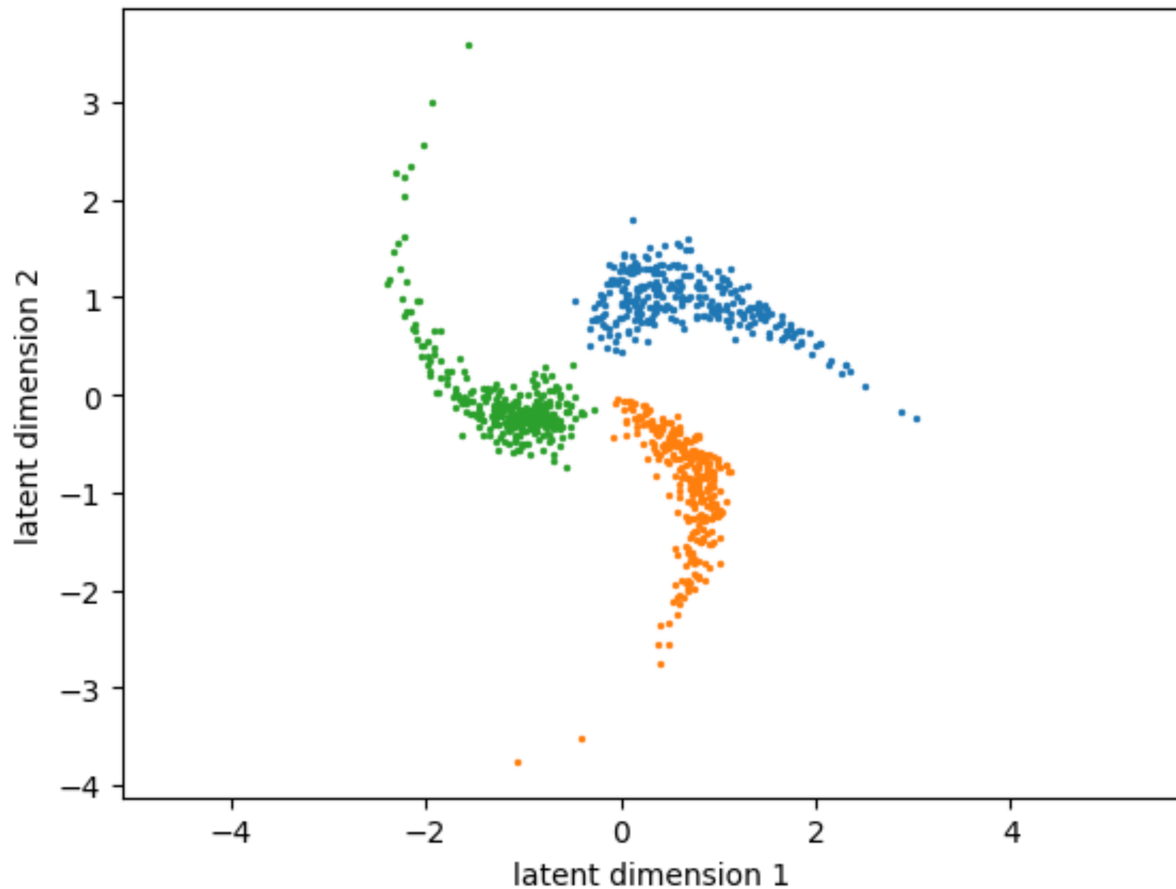
2.2 [5pts] In this part, we visualize how the data looks like in the latent space. We simply use the trained recognition network (the encoder) to map each input

to latent space.

In [ ]:
```
# Show the means of the encoded data in a 2D latent space.
# Don't worry if this doesn't look much like a Gaussian.

for k in range(num_clusters):
    # cur_data =  # TODO get all the data from this cluster.
    # transformed =  # TODO find the mean of q(z|x) for each x.
    # REMOVE SOLUTION
    cur_data = data[labels == k]
    transformed = recognition_net(cur_data)[0].detach()
    plt.scatter(transformed[:, 0], transformed[:, 1], s=2)
    print(transformed.shape)

plt.axis("equal")
plt.xlabel("latent dimension 1")
plt.ylabel("latent dimension 2")
```

```
torch.Size([300, 2])
torch.Size([300, 2])
torch.Size([300, 2])
```

Out[ ]: `Text(0, 0.5, 'latent dimension 2')`



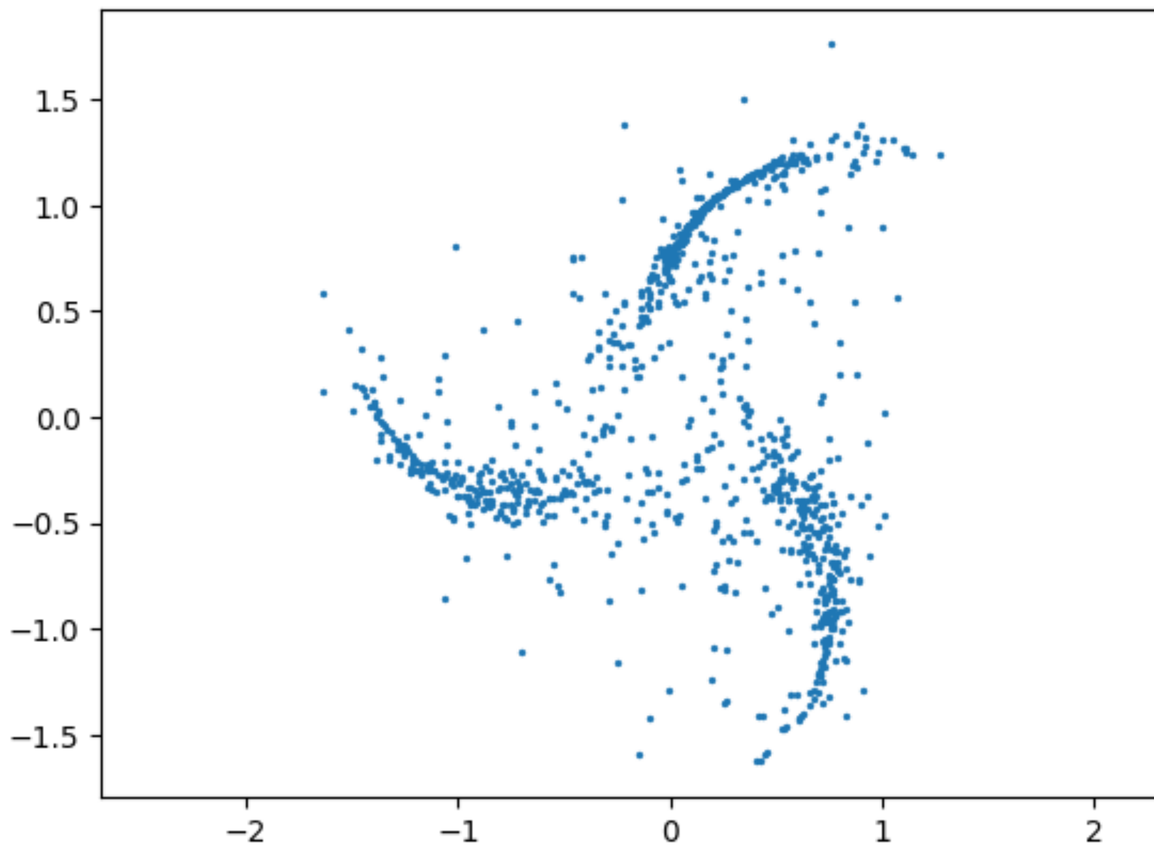
2.3 [5pts] Generate new data using the decoder and the generative model we just trained.

For this, we simply generate 1000 latent variables in the latent space from the prior and pass it through the decoder network.

You shouldn't expect this to match the data exactly, just to get the overall shape and number of clusters roughly correct.

In [ ]:
```
# Sample data from the trained generative model to see if it
# roughly matches the data.  # Note: This doesn't add the likelihood noise,
# although it should if we want it to match the data.

num_samples = 1000
# samples =  # TODO
# transformed =  # TODO
# REMOVE SOLUTION
samples = torch.randn(num_samples, latent_dimension) # TODO
transformed = decoder_net(samples)[0].detach() # TODO
plt.scatter(transformed[:, 0], transformed[:, 1], s=2)
plt.axis("equal")
```

Out[ ]:
```
(-1.7869500756263732,
 1.420520508289337,
 -1.7971095979213714,
 1.9274159610271453)
```

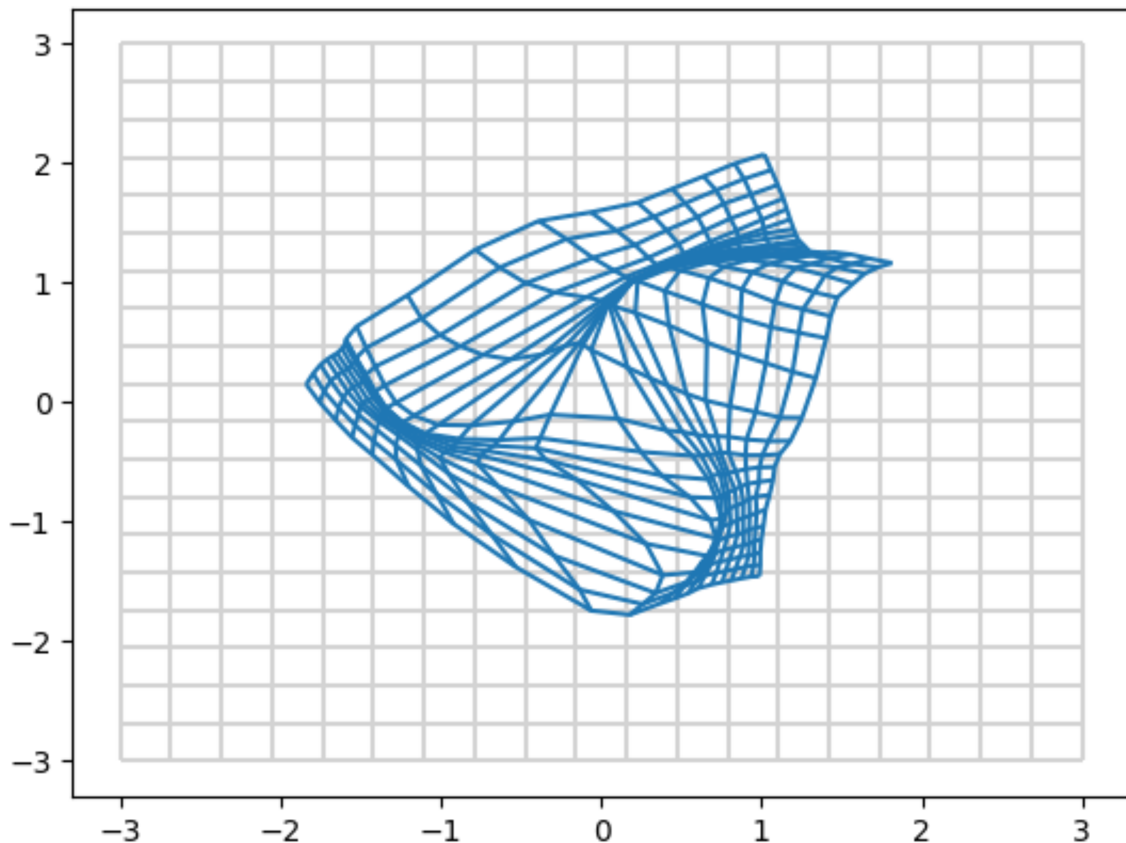


Here's a debugging tool only available when both the latent space and the data are both 2-dimensional. We can show the function being learned by the encoder by showing how it warps a 2D grid into the latent space.

In [ ]:
```
from matplotlib.collections import LineCollection

def plot_grid(x,y, ax=None, **kwargs):
    ax = ax or plt.gca()
    segs1 = np.stack((x,y), axis=2)
    segs2 = segs1.transpose(1,0,2)
    ax.add_collection(LineCollection(segs1, **kwargs))
    ax.add_collection(LineCollection(segs2, **kwargs))
    ax.autoscale()

def f(x,y):
    xy = torch.stack([x.flatten(), y.flatten()], dim=1)
    return decoder_net(xy)

fig, ax = plt.subplots()

grid_x,grid_y = torch.meshgrid(torch.linspace(-3,3,20),torch.linspace(-3,3,2
plot_grid(grid_x,grid_y, ax=ax,  color="lightgrey")

distx, disty = f(grid_x,grid_y)
```

```
distx, disty = distx.reshape(20, 20, 2), disty.reshape(20, 20, 2)
plot_grid(distx[:, :, 0].detach().numpy(), distx[:, :, 1].detach().numpy(),

plt.show()
```

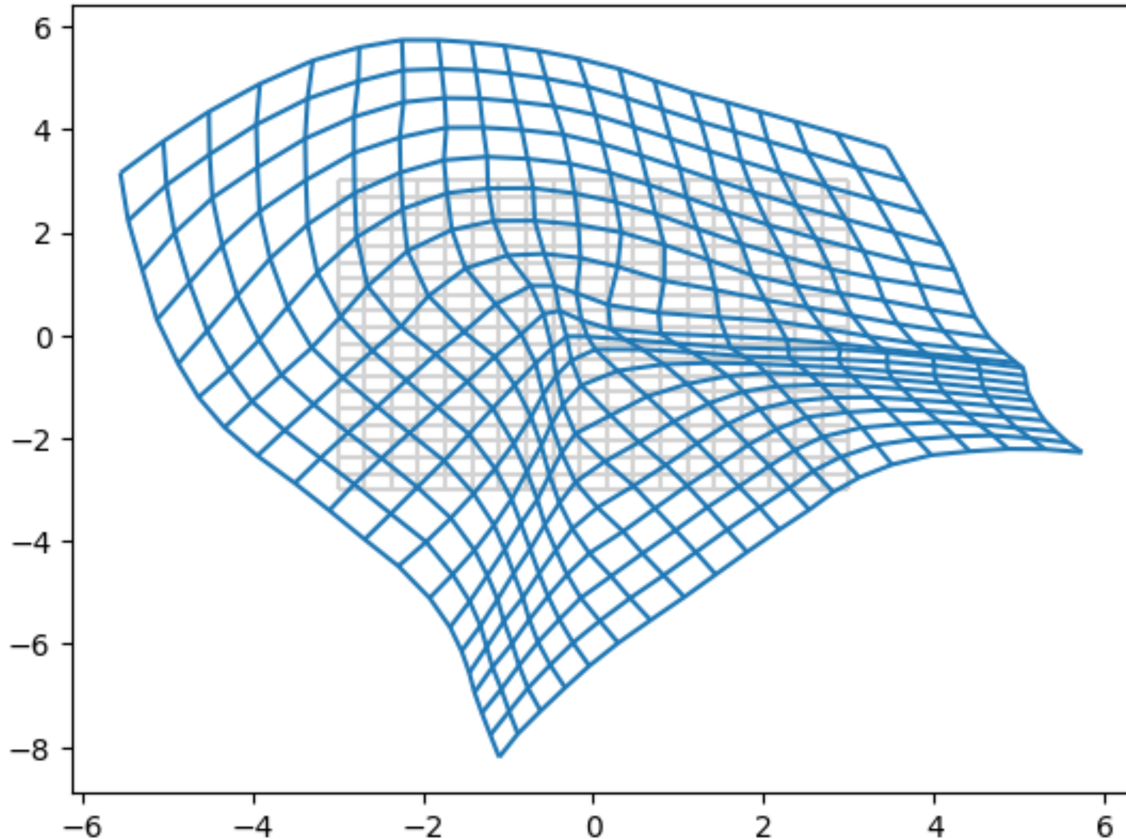
```
/usr/local/lib/python3.10/dist-packages/torch/functional.py:507: UserWarnin
g: torch.meshgrid: in an upcoming release, it will be required to pass the i
ndexing argument. (Triggered internally at ../aten/src/ATen/native/TensorSha
pe.cpp:3549.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Here, we show the function being learned by the decoder by showing how it warps a 2D grid into the observed space.

In [ ]:
```
def f(x,y):
    xy = torch.stack([x.flatten(), y.flatten()], dim=1)
    return recognition_net(xy)

fig, ax = plt.subplots()

grid_x,grid_y = torch.meshgrid(torch.linspace(-3,3,20),torch.linspace(-3,3,2
plot_grid(grid_x,grid_y, ax=ax,  color="lightgrey")

distx, disty = f(grid_x,grid_y)
distx, disty = distx.reshape(20, 20, 2), disty.reshape(20, 20, 2)
```

```
plot_grid(distx[:, :, 0].detach().numpy(), distx[:, :, 1].detach().numpy(),

plt.show()
```



[38pts] Question 3: Expectation-Maximization (EM) algorithm

In [ ]:
```
%matplotlib inline
import scipy
import numpy as np
import itertools
import matplotlib.pyplot as plt
```

3.1 [5pts] Generating the Data

a) [4pts] First, we will generate some data for this problem. Set the number of points $N = 400$, their dimension $D = 2$, and the number of clusters $K = 2$, and generate data from the distribution $p(x|z = k) = N(\mu_k, \Sigma_k)$. Sample 200 data points for $k = 1$ and 200 for $k = 2$, with

$$\mu_1 = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}, \mu_2 = \begin{bmatrix} 6.0 \\ 0.1 \end{bmatrix} \text{ and } \Sigma_1 = \Sigma_2 = \begin{bmatrix} 10 & 7 \\ 7 & 10 \end{bmatrix}$$

Here, $N = 400$. If you generate the data, you already know which sample comes from which class. Complete the following statements and run to generate.

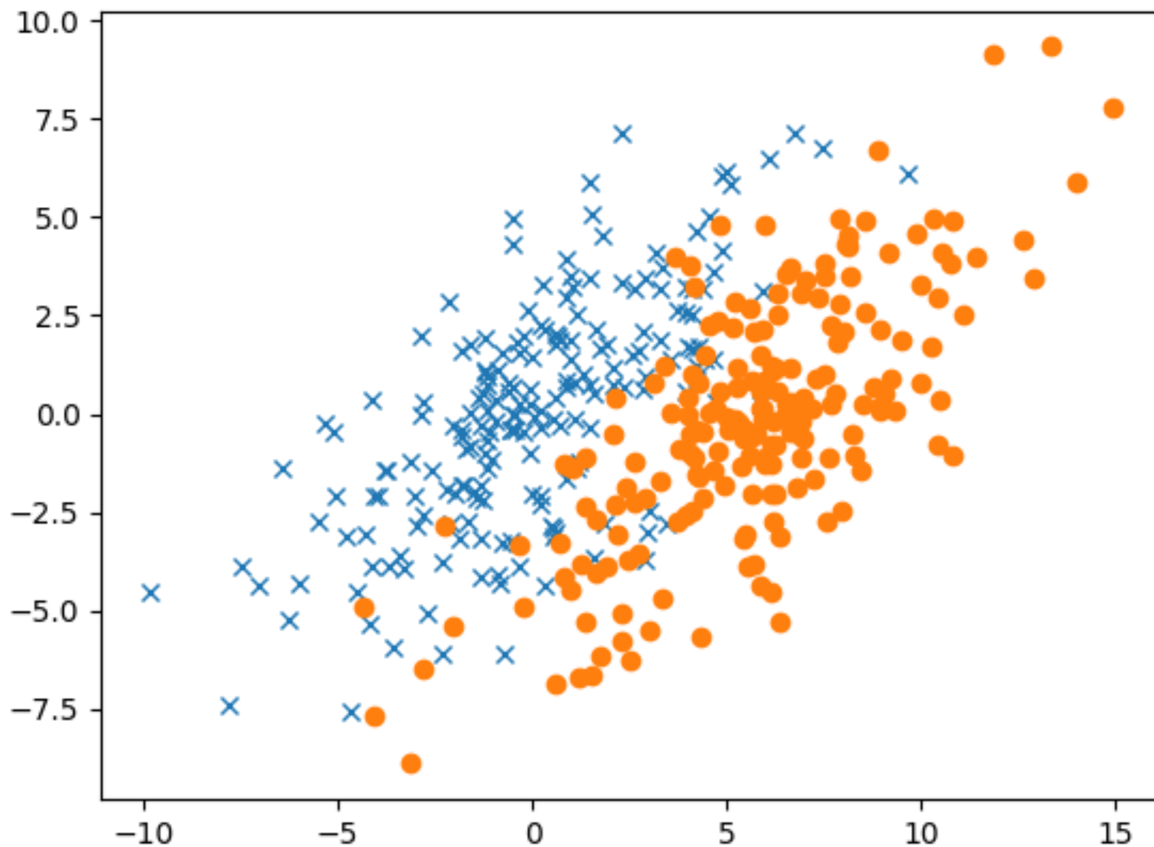
Hint: you can use `np.random.multivariate_normal`.

In [ ]:
```
num_samples = 400
cov = np.array([[1., .7], [.7, 1.]]) * 10
mean_1 = [.1, .1]
mean_2 = [6., .1]

x_class1 = np.random.multivariate_normal(mean_1, cov, num_samples // 2)
x_class2 = np.random.multivariate_normal(mean_2, cov, num_samples // 2)
xy_class1 = np.column_stack((x_class1, np.zeros(num_samples // 2)))
xy_class2 = np.column_stack((x_class2, np.ones(num_samples // 2)))

data_full = np.row_stack([xy_class1, xy_class2])
np.random.shuffle(data_full)
data = data_full[:, :2]
labels = data_full[:, 2]
```

b) [1pt] Make a scatter plot of the data points showing the true cluster assignment of each point using different color codes and shape (x for first class and circles for second class):

In [ ]:
```
plt.plot(x_class1[:,0], x_class1[:, 1], 'x')
plt.plot(x_class2[:,0], x_class2[:, 1], 'o')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7d2af183b9a0>]
```



3.2 [11pt] Implement and Run K-Means algorithm

Now, we assume that the true class labels are not known. Implement the k-means algorithm for this problem. Write two functions: `km_assignment_step`, and `km_refitting_step` as given in the lecture (Here, `km_` means k-means). Identify the correct arguments, and the order to run them. Initialize the algorithm with

$$\hat{\mu}_1 = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}, \hat{\mu}_2 = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix}$$

and run it until convergence. Show the resulting cluster assignments on a scatter plot either using different color codes or shape or both. Also plot the cost (distortion in lecture slides) vs. the number of iterations. Report your misclassification error.

a) [2pts] Complete the `cost` function.

In [ ]:
```
def cost(data, R, Mu):
    """ Compute the K-Means cost function

    Args:
        data: an NxD matrix for the data points
```

```
        Mu: a DxK matrix for the cluster means locations
        R: an NxK matrix of responsibilities (assignments)

    Returns:
        J: the K-Means cost
    """

    N, D = data.shape
    K = Mu.shape[1]
    J = 0
    for k in range(K):
        J += np.dot(np.linalg.norm(data - np.array([Mu[:, k], ] * N), axis=1

    return J
```

b) [3pts] K-Means assignment step.

In [ ]:
```
def km_assignment_step(data, Mu):
    """ Compute K-Means assignment step

    Args:
        data: an NxD matrix for the data points
        Mu: a DxK matrix for the cluster means locations

    Returns:
        R_new: an NxK matrix of responsibilities
    """

    N, D = data.shape
    K = Mu.shape[1]
    r = np.zeros([N, K])

    for k in range(K):
        r[:, k] = np.linalg.norm(data - np.array([Mu[:, k], ] * N), axis=1)*
        arg_min = np.argmin(r, axis=1)
        R_new = np.zeros([N, K])
        R_new[range(N), arg_min] = 1
    return R_new
```

c) [2pts] K-Means refitting step.

In [ ]:
```
def km_refitting_step(data, R, Mu):
    """ Compute K-Means refitting step.

    Args:
        data: an NxD matrix for the data points
        R: an NxK matrix of responsibilities
        Mu: a DxK matrix for the cluster means locations
```

```
    Returns:
        Mu_new: a DxK matrix for the new cluster means locations
    """

    N, D = data.shape
    K = Mu.shape[1]
    Mu_new = (R.T.dot(data) / np.sum(R, axis=0)[:, None]).T
    return Mu_new
```

d) [3pts] Run this cell to call the K-Means algorithm.

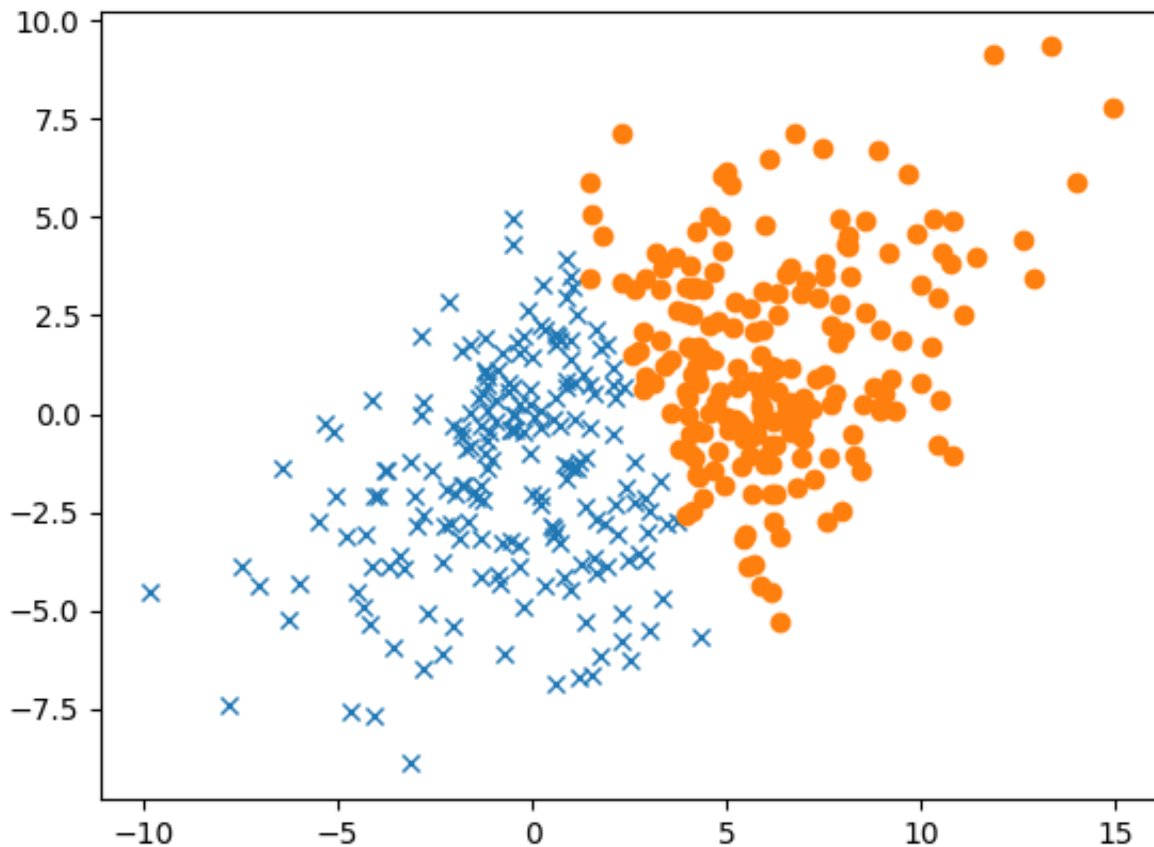
In [ ]:
```
N, D = data.shape
K = 2
max_iter = 10
class_init = np.random.binomial(1., .5, size=N)
R = np.vstack([class_init, 1 - class_init]).T

Mu = np.zeros([D, K])
Mu[:, 1] = 1.
R.T.dot(data), np.sum(R, axis=0)

for it in range(max_iter):
    R = km_assignment_step(data, Mu)
    Mu = km_refitting_step(data, R, Mu)
    print(it, cost(data, R, Mu))

class_1 = np.where(R[:, 0])
class_2 = np.where(R[:, 1])
```

```
0 5434.650128337574
1 5226.087177453153
2 5187.427646748891
3 5176.875690450033
4 5174.288919653969
5 5171.712244966692
6 5169.228867734544
7 5168.40426640519
8 5168.40426640519
9 5168.40426640519
```

e) [1pt] Make a scatterplot for the data points showing the K-Means cluster assignments of each point.

In [ ]:
```
plt.plot(data[class_1, 0][0], data[class_1, 1][0], 'x')
plt.plot(data[class_2, 0][0], data[class_2, 1][0], 'o')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7d2af1766590>]
```



3.3 [16pts] Implement EM algorithm for Gaussian mixtures

Next, implement the EM algorithm for Gaussian mixtures. Write three functions: `log_likelihood`, `gm_e_step`, and `gm_m_step` as given in the lecture.

- Identify the correct arguments, and the order to run them.
- Initialize the algorithm with the same initialization as in Q2.1 for the means, and with $\hat{\Sigma}_1 = \hat{\Sigma}_2 = I$, and $\hat{\pi}_1 = \hat{\pi}_2$.

Run the algorithm until convergence and show the resulting cluster assignments on a scatter plot either using different color codes or shape or both. Also plot the log-likelihood vs. the number of iterations. Report your misclassification error.

In [ ]:
```
def normal_density(x, mu, Sigma):
    return np.exp(-.5 * np.dot(x - mu, np.linalg.solve(Sigma, x - mu))) \
        / np.sqrt(np.linalg.det(2 * np.pi * Sigma))
```

a) [4pts] Log-Likelihood.

In [ ]:
```
def log_likelihood(data, Mu, Sigma, Pi):
    """ Compute log likelihood on the data given the Gaussian Mixture Parame

    Args:
        data: an NxD matrix for the data points
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
        Sigma: a list of size K with each element being DxD covariance matri
        Pi: a vector of size K for the mixing coefficients

    Returns:
        L: a scalar denoting the log likelihood of the data given the Gaussi
    """

    N, D = data.shape
    K = Mu.shape[1]
    L = 0.
    for n in range(N):
        T = 0.
        for k in range(K):
            T += Pi[k] * normal_density(data[n, :], Mu[:, k], Sigma[k])
        L += np.log(T)
    return L
```

b) [4pts] Gaussian Mixture Expectation Step.

In [ ]:
```
def gm_e_step(data, Mu, Sigma, Pi):
    """ Gaussian Mixture Expectation Step.

    Args:
        data: an NxD matrix for the data points
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
        Sigma: a list of size K with each element being DxD covariance matri
        Pi: a vector of size K for the mixing coefficients

    Returns:
        Gamma: an NxK matrix of responsibilities
    """

    N, D = data.shape
    K = Mu.shape[1]
    Gamma = np.zeros([N, K])

    for n in range(N):
        for k in range(K):
            Gamma[n, k] = Pi[k] * normal_density(data[n, :], Mu[:, k], Sigma
        Gamma[n, :] /= np.sum(Gamma[n, :])

    return Gamma
```

c) [4pts] Gaussian Mixture Maximization Step.

```
In [ ]: def gm_m_step(data, Gamma):
            """ Gaussian Mixture Maximization Step.

            Args:
                data: an NxD matrix for the data points
                Gamma: an NxK matrix of responsibilities

            Returns:
                Mu: a DxK matrix for the means of the K Gaussian Mixtures
                Sigma: a list of size K with each element being DxD covariance matri
                Pi: a vector of size K for the mixing coefficients
            """

            N, D = data.shape
            K = Gamma.shape[1]
            Nk = np.sum(Gamma, axis=0)
            Mu = (np.dot(Gamma.T, data) / Nk[: None]).T
            Sigma = [np.eye(2), np.eye(2)]

            for k in range(K):
                data_white = data - np.array([Mu[:, k], ] * N)
                Sigma[k] = data_white.T.dot(np.diag(Gamma[:, k]).dot(data_white)) /

            Pi = Nk / N
            return Mu, Sigma, Pi
```

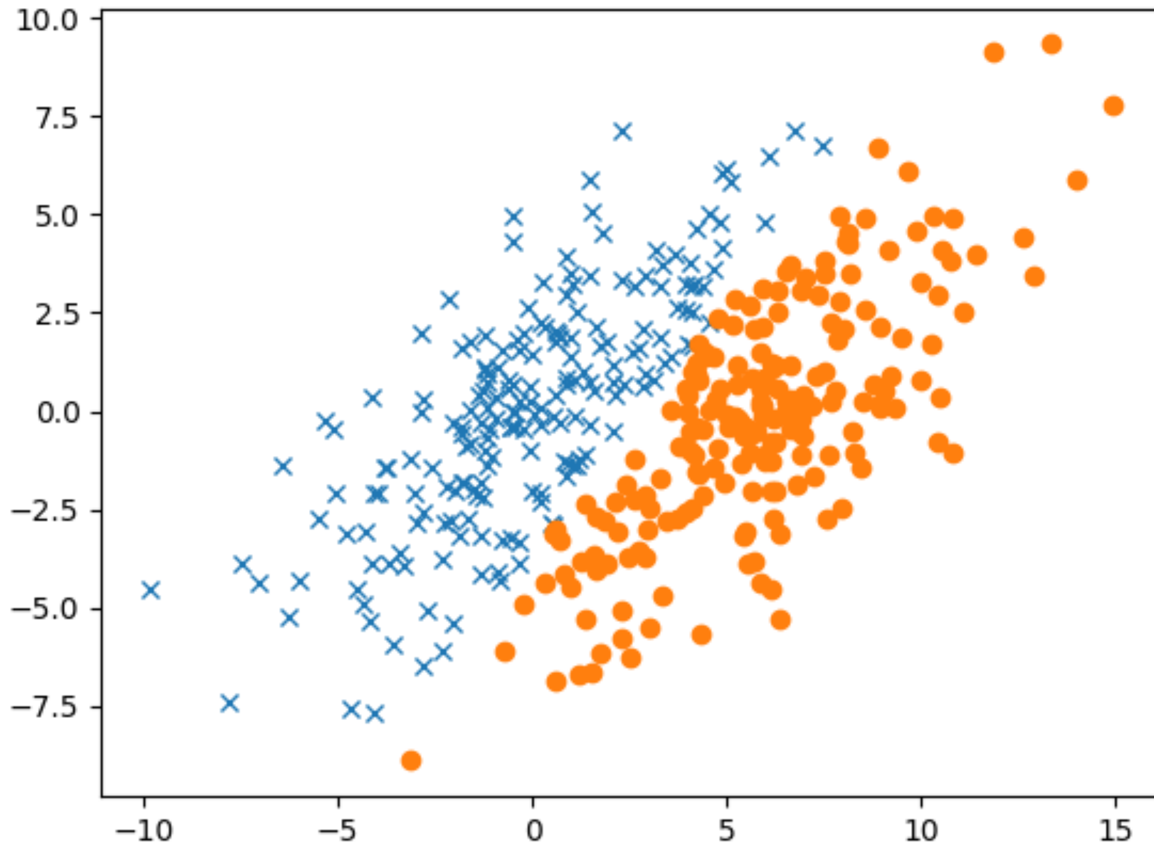
d) [3pts] Run this cell to call the Gaussian Mixture EM algorithm.

```
In [ ]: N, D = data.shape
        K = 2
        Mu = np.zeros([D, K])
        Mu[:, 1] = 1.
        Sigma = [np.eye(2), np.eye(2)]
        Pi = np.ones(K) / K
        Gamma = np.zeros([N, K]) # Gamma is the matrix of responsibilities

        max_iter  = 200

        for it in range(max_iter):
            Gamma = gm_e_step(data, Mu, Sigma, Pi)
            Mu, Sigma, Pi = gm_m_step(data, Gamma)
            # print(it, log_likelihood(data, Mu, Sigma, Pi)) # This function makes t

        class_1 = np.where(Gamma[:, 0] >= .5)
        class_2 = np.where(Gamma[:, 1] >= .5)

        # print()
```

e) [1pt] Make a scatterplot for the data points showing the Gaussian Mixture cluster assignments of each point

In [ ]:
```
plt.plot(data[class_1, 0][0], data[class_1, 1][0], 'x')
plt.plot(data[class_2, 0][0], data[class_2, 1][0], 'o')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7d2af15460e0>]
```



3.4 [6pts] Comment on findings + additional experiments

Comment on the results:

- Compare the performance of k-Means and EM based on the resulting cluster assignments. [2pts]
- Compare the performance of k-Means and EM based on their convergence rate. What is the bottleneck for which method? [2pts]
- Experiment with 5 different data realizations (generate new data), run your algorithms, and summarize your findings. Does the algorithm performance depend on different realizations of data? [3pts]

- Comment on what might happen as you increase the number K of clusters. [1pt]

Your answer here.

In [ ]:
```
# your code here, if you have any
```