

Biomedical Data Science 2026: Homework Assignment 2

CBB & CPSC & S&DS (programming) Assignment

- Release Date: Week of Feb 23rd, 2026
- Due: March 29th, 11:59pm

Name: write your name here (double click to edit)

Instructions

- You only need to write code with the `TODO` comments.
- You may write code outside of these blocks, but it will not be graded.
- If you make use of any online resource please cite the source in the code comments.
- You may use some small utility functions directly, but notice that directly copying large chunks of codes (even with variable name replacement) are not allowed and will be considered as plagiarism.
- After writing your code, you can run the cell by either pressing "SHIFT"+"ENTER" or by clicking on "Run Cell" (denoted by a play symbol) in the upper bar of the notebook.
- After you are finished, turn in both the **.ipynb** version and the **PDF** version to Canvas.

Part 1: Genome-Wide Association Study (GWAS) Visualization

Introduction

In Genome-Wide Association Studies (GWAS), we test millions of Single Nucleotide Polymorphisms (SNPs) across the genome for association with a trait (like height or disease risk). The results of these tests are compiled into "Summary Statistics."

Because we perform millions of independent statistical tests, using a standard α -value threshold would result in thousands of false positives. To account for this, we use a much stricter genome-wide significance threshold, typically the Bonferroni corrected threshold. In this assignment, you will analyze a thinned subset of the GIANT Consortium's GWAS on human height. You will calculate the appropriate significance thresholds and generate the two most common GWAS visualizations.

1.1 Data Loading and Preprocessing (5 pts)

```
In [ ]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

# 1. Load the dataset (assuming columns: CHR, POS, SNP, P)
# TODO: Read 'height_gwas_sample.txt' into a pandas DataFrame.
# Note: it is tab-separated.
gwas_df =

# 2. Drop any rows with missing P-values
# TODO
gwas_df =

# 3. Calculate the  $-\log_{10}(P\text{-value})$  for visualization
# TODO: Add a new column ' $-\log_{10}(p)$ ' to the dataframe
gwas_df[' $-\log_{10}(p)$ '] =

print(f"Loaded {len(gwas_df)} SNPs for analysis.")
```

1.2 Multiple Testing Correction (5 pts)

For 1.2, you must include the exact value of the corrected family-wise error rate (FWER) in your answer.

```
In [ ]: # To avoid false positives, calculate the Bonferroni significance threshold.

alpha = 0.05

# TODO: Calculate the Bonferroni threshold
bonferroni_threshold =

# TODO: Calculate the  $-\log_{10}$  of the Bonferroni threshold (we need this to draw the threshold line)
log_threshold =

# TODO: Count how many SNPs are strictly genome-wide significant
num_significant =

print(f"Bonferroni Threshold: {bonferroni_threshold:.2e}")
print(f"Number of significant SNPs: {num_significant}")
```

1.3 Visualizing the Manhattan Plot (10 pts)

```
In [ ]: # A Manhattan plot shows the  $-\log_{10}(P)$  of each SNP ordered by its position.
# For simplicity in this homework, we can plot the points
# sequentially based on their index.

plt.figure(figsize=(12, 5))

# TODO: Create a scatter plot of the  $-\log_{10}(P)$  values.
```

```

# TODO: Add a horizontal line (axhline) representing the log_threshold calcul
# Make it red and dashed.

plt.title("Manhattan Plot of Height GWAS")
plt.xlabel("SNP Index")
plt.ylabel("-log10(P-value)")
plt.show()

```

1.4 Visualizing the QQ Plot (10 pts)

```

In [ ]: # A QQ plot compares the observed p-values against what we would expect by c
# Expected p-values are uniformly distributed between 0 and 1.

# 1. Sort the observed p-values from smallest to largest
# TODO
observed_pvals =

# 2. Generate expected p-values
# The expected p-values are roughly (rank) / (total_number_of_SNPs + 1)
# TODO: Create an array of expected p-values
expected_pvals =

# 3. Plot the QQ plot
plt.figure(figsize=(6, 6))

# TODO: Scatter plot of expected (x-axis) vs observed (y-axis)

# TODO: Draw the diagonal line (y = x) representing the null hypothesis
# Hint: plot from [0, max(exp_log_p)] to [0, max(exp_log_p)]

plt.title("QQ Plot of GWAS P-values")
plt.xlabel("Expected -log10(P-value)")
plt.ylabel("Observed -log10(P-value)")
plt.show()

```

1.5 Calculating the Genomic Inflation Factor (5 Pts)

When conducting a GWAS, it is entirely possible that systematic biases—most commonly population stratification (e.g., mixing different ancestral populations)—can cause χ^2 -values to look artificially significant across the entire genome. We can quantify this global bias using the Genomic Inflation Factor (Lambda). It is defined as the ratio of the median observed (χ^2 -squared) test statistic to the expected median statistic under the null hypothesis.

The expected median of a distribution with 1 degree of freedom is approximately **0.4549**. Calculate and print your Genomic Inflation Factor value. Do you see the GWAS

shows signs of inflation? Can you explain why?

```
In [ ]: # 1. Find the median observed p-value
# TODO
median_pval =

# 2. Convert the median p-value to an observed chi-squared statistic
# TODO
chi2_observed =

# 3. Calculate lambda
# TODO
lambda_gc =

print(f"Genomic Inflation Factor (lambda): {lambda_gc:.3f}")
```

Explain your result here: (double click to edit)

Part 2: Variational Autoencoder on DermaMNIST

Introduction

A **Variational Autoencoder (VAE)** is a generative model that, like a standard autoencoder, consists of an **encoder** and a **decoder** network. The encoder $q_{\phi}(z|x)$ takes an input (such as an image) and maps it into a **latent space** – instead of producing a single latent vector, it produces a **probability distribution** (typically a Gaussian with parameters μ and σ) that represents the encoding of the input. The decoder $p_{\theta}(x|z)$ then takes a sample z from this latent distribution and tries to **reconstruct** the original input. The latent space is a compressed, abstract representation of the data where similar inputs are expected to lie close together. In a VAE, this latent space is **continuous and probabilistic**, which allows us not only to reconstruct inputs but also to generate **new** data by sampling from the latent space.

Key components of a VAE:

- **Encoder (x to z):** The encoder network transforms an input x into the parameters of a distribution over the latent variable z . Commonly, it outputs a vector of means μ and a vector of (log) variances $\log \sigma^2$ for a Gaussian distribution $q_{\phi}(z|x) = \mathcal{N}(z; \mu(x), \text{diag}(\sigma^2(x)))$. This distribution reflects uncertainty about how x is represented in latent space.
- **Latent space & reparameterization:** Instead of directly sampling $z \sim \mathcal{N}(\mu, \sigma^2)$ inside the network (which would break backpropagation due to the random sampling), VAEs use the **reparameterization**

trick. We sample an auxiliary noise $\epsilon \sim \mathcal{N}(0, I)$ and then **reparameterize** as $z = \mu + \sigma \odot \epsilon$, where $\sigma = \exp(0.5 \odot \log \sigma^2)$ and \odot is elementwise multiplication. This way, μ and σ are treated as deterministic outputs of the encoder, and the randomness is pushed to ϵ , which is independent of the network parameters. This trick allows gradients to flow through z via μ and σ during training.

- **Decoder ($z \rightarrow \hat{x}$):** The decoder network takes a latent sample z and produces a reconstruction \hat{x} in the original data space. The decoder is trained to make \hat{x} as close as possible to the original x . In a generative sense, if we sample z from the latent prior (often a standard Normal $\mathcal{N}(0, I)$), the decoder can generate *new* samples that resemble the training data.
- **Loss function (ELBO):** VAEs are trained by maximizing the Evidence Lower Bound (ELBO) on the data log-likelihood. In practice, this corresponds to minimizing a loss that has two terms: a **reconstruction loss** and a **KL divergence** loss. The reconstruction loss ensures the decoded output \hat{x} is similar to the input x (for example, using binary cross-entropy or mean squared error between x and \hat{x}). The **KL divergence** term $D_{\text{KL}}(q_{\phi}(z|x) \parallel p(z))$ acts as a regularizer, pushing the encoder's output distribution $q_{\phi}(z|x)$ to be close to the chosen prior $p(z)$ (usually $p(z) = \mathcal{N}(0, I)$). The KL term effectively encourages the latent space to be well-organized and prevents the encoder from just memorizing the inputs. Mathematically, for one data point x , the loss (to minimize) can be written as:

$$\mathcal{L}(x) = \underbrace{\mathbb{E}_{q_{\phi}(z|x)}[-\log p_{\theta}(x|z)]}_{\text{Reconstruction Loss}} + \underbrace{D_{\text{KL}}(q_{\phi}(z|x) \parallel p(z))}_{\text{KL Divergence}}.$$
In simpler terms, we want \hat{x} to be accurate (low reconstruction error) while keeping the latent z distributions close to $\mathcal{N}(0, I)$ (via a low KL divergence). Balancing these two forces prevents overfitting and yields a model that can generate realistic new data.

Conditional VAE (CVAE):

In a *conditional* VAE, we incorporate additional information (like a class label y) into the encoder and decoder. This means the encoder learns $q(z|x, y)$ and the decoder learns $p(x|z, y)$. In practice, we feed the class label into both networks (for example, by concatenating a one-hot encoded label to the input or latent vector). By doing so, the VAE learns a latent space that is structured *per class*, and we can generate new examples conditioned on a specific class. In this assignment, we will implement a **conditional VAE on the DermaMNIST dataset**, conditioning on the disease label for each dermatoscopic image. DermaMNIST is a part of the MedMNIST collection of

medical image datasets and contains 28×28 skin lesion images categorized into 7 classes of skin diseases (derived from the HAM10000 dataset).

Assignment Overview

In this assignment, you will build and train a conditional VAE on the **DermaMNIST** dataset using PyTorch. We will proceed through the following steps:

1. **Data Loading and Preprocessing:** Use the MedMNIST API to load DermaMNIST and prepare data loaders with appropriate transformations.
2. **Implementing the VAE Architecture:** Define the encoder and decoder networks for a conditional VAE. The architecture will incorporate image data and condition on class labels.
3. **Training Loop:** Train the VAE using the reconstruction + KL divergence loss. Implement the forward pass, reparameterization, loss computation, and backpropagation.
4. **Latent Space Visualization:** Use UMAP to project the learned latent space to 2D and visualize how the data clusters by class in the latent space.
5. **(Bonus: optional) Generation of New Samples:** Sample random latent vectors and pass them through the decoder to generate new skin lesion images for a given condition (class label).

Throughout the notebook, **fill in the code where indicated** (look for comments like `TODO` or blank placeholders). Short descriptions are provided to guide you for each part.

2.1: Data Loading and Preprocessing (5 pts)

In this section, we will load the DermaMNIST dataset and prepare it for training our VAE. DermaMNIST contains 28×28 color images of skin lesions across 7 classes. We will use the `medmnist` package to download and load the dataset easily. The tasks are:

1. **Define a transformation pipeline** for the images. At minimum, convert images to PyTorch tensors (and optionally normalize the pixel values).
2. **Load the DermaMNIST dataset** for training, validation, and testing splits using the provided MedMNIST classes.
3. **Create DataLoader objects** for each split to enable batching during training and evaluation.

Hint: Use `medmnist.DermaMNIST` class with the `split` argument as `'train'`, `'val'`, or `'test'`. The dataset returns images (as PIL images or arrays) and their labels. You can specify `transform=...` to apply the transformations defined. You might need to `pip install medmnist` if it's not already available.

```
In [ ]: import torch
from torch.utils.data import DataLoader
from torchvision import transforms

# 1. Define image transformations: convert images to tensor and (optionally)
# TODO
transform =

# 2. Load the DermaMNIST dataset for train, val, and test splits.
# If medmnist is not installed, install it via pip: !pip install medmnist
from medmnist import DermaMNIST

batch_size = 128 # feel free to adjust batch size
download = True # download dataset if not already downloaded

train_dataset = DermaMNIST(split='train', transform=transform, download=downr
val_dataset = DermaMNIST(split='val', transform=transform, download=downr
test_dataset = DermaMNIST(split='test', transform=transform, download=downr

# 3. Create DataLoader for each split.
# TODO

# 4. Set device (use GPU if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
In [ ]: # Quick sanity check: get one batch
images, labels = next(iter(train_loader))
print(f"Batch of images shape: {images.shape}") # expected: (batch_size, 3
print(f"Batch of labels shape: {labels.shape}") # expected: (batch_size,)
print(f"Unique labels in this batch: {labels.unique().tolist()}")
```

2.2: Implementing the Encoder and Decoder (15 pts)

Now, let's build the neural network architecture for our conditional VAE. We will create two modules: an **Encoder** that takes an image (and condition label) as input and produces the latent distribution parameters, and a **Decoder** that takes a latent vector (and condition label) and reconstructs the image. Because this is a *conditional* VAE, we need to incorporate the class label y at both stages:

- In the **Encoder**, y can be included by concatenating a one-hot representation of y with the image features before the final layers that produce μ and $\log \sigma^2$.
- In the **Decoder**, y can be included by concatenating a one-hot y with z before the first decoding layer.

We will use a simple convolutional neural network (CNN) architecture for both encoder and decoder (since the images are 28×28 , we can also use fully-connected layers, but CNN will better capture spatial structure). Parts of the architecture are given, and you will fill in the missing pieces.

Encoder Architecture (example): A possible design is:

- Convolutional layers to encode the $28 \times 28 \times 3$ image into a smaller feature map.
- Flatten and concatenate with the label's one-hot vector.
- Two linear layers to output `mu` and `log_var` (each of dimension `latent_dim`).

Decoder Architecture (example): A reverse process:

- Take the sampled latent vector and concatenate with label one-hot.
- A linear layer to expand into a feature map shape.
- Transposed convolution (or upsampling + conv) layers to reconstruct the image from the latent features.
- The final layer should produce $3 \times 28 \times 28$ output (the reconstructed image), likely followed by a sigmoid activation (to bound pixel values between 0 and 1).

TODO: Include the exact content of the given sample and use the variable name lat_dim as latent

Complete the code below by filling in the `TODO` sections for the encoder and decoder implementation. To improve training stability and prevent overfitting, you must include at least one regularization technique (e.g., BatchNorm or Dropout) in both your encoder and decoder pathways.

```
In [ ]: import torch.nn as nn
import torch.nn.functional as F

latent_dim = 128      # Dimensionality of the latent space (you can adjust)
num_classes = 7       # DermaMNIST has 7 classes
img_channels = 3      # RGB images

class Encoder(nn.Module):
    def __init__(self, latent_dim, num_classes):
        super(Encoder, self).__init__()
        # TODO
        # Convolutional encoder

        # Linear layers for mean and log-variance, taking into account label
        self.fc_mu =
        self.fc_logvar =

    def forward(self, x, y):
        # TODO

        return mu, log_var

class Decoder(nn.Module):
    def __init__(self, latent_dim, num_classes):
        super(Decoder, self).__init__()
        # TODO
```



```

        # Linear layer to expand z (with label) into a feature map

        # Transposed conv layers to upscale to 28x28

def forward(self, z, y):
    # TODO

    return recon_x

```

```

In [ ]: # Instantiate encoder and decoder
encoder = Encoder(latent_dim, num_classes).to(device)
decoder = Decoder(latent_dim, num_classes).to(device)

# Test the encoder and decoder with a dummy input
x_sample, y_sample = images[0:2].to(device), labels[0:2].to(device) # take a
mu, log_var = encoder(x_sample, y_sample)
print("Encoder output shapes:", mu.shape, log_var.shape) # each [2, latent_
z_sample = torch.randn_like(mu) # a random latent vector (for testing decod
x_recon = decoder(z_sample, y_sample)
print("Decoder output shape:", x_recon.shape) # expected [2, 3, 28, 28]

```

2.3: Training the VAE (15 pts)

Now that we have our encoder and decoder, the next step is to train the VAE. In training, we will:

- Iterate over batches of training data.
- For each batch, **encode** the images to get μ and $\log \sigma^2$.
- **Sample** a latent vector z using the reparameterization trick.
- **Decode** z to get reconstructed images.
- Compute the **loss**: sum of reconstruction loss and KL divergence loss.
- Backpropagate and update the model parameters.

The KL divergence term for a single data point is given by:

$$D_{\text{KL}}(q(z|x) \parallel p(z)) = -\frac{1}{2} \sum_{j=1}^d \left(1 + \log \sigma_j^2 - \mu_j^2 - \sigma_j^2 \right),$$

where d is the latent dimension. We will implement this formula.

Fill in the code below to complete the training loop. Look for places marked **TODO**:

- Implement the reparameterization to obtain `z` from `mu` and `log_var`.
- Compute the `recon_loss`.
- Compute the `kl_loss` using the formula above.
- Sum the losses to get `total_loss` and perform backpropagation.

```

In [ ]: import torch.optim as optim
# TODO
# Optimizer
optimizer =

# Training parameters (feel free to adjust)
num_epochs = 200
beta = 0.001

for epoch in range(num_epochs):
    encoder.train()
    decoder.train()
    train_loss = 0.0
    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)
        # Encode
        mu, log_var = encoder(images, labels)
        # Reparameterization: sample z from N(mu, sigma^2)
        # TODO: implement reparameterization trick

        # TODO
        # Compute reconstruction loss

        # Compute KL divergence loss

        # Total VAE loss

        # Backpropagation and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
    # Report average loss for the epoch (feel free to adjust)
    avg_loss = train_loss / len(train_loader)
    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch+1}/{num_epochs}], Average Loss: {avg_loss:.4f}")

```

After training for several epochs, you should observe the loss decreasing. During training, you might also consider monitoring the separate components of the loss (e.g., `print recon_loss.item()` and `kl_loss.item()` averages) to see the balance between reconstruction and KL. Over time, the encoder will learn to encode meaningful latent distributions and the KL term will balance with reconstruction quality.

Question: Propose a strategy to dynamically adjust the β parameter during the training process. Describe how your adjustment strategy works mathematically or programmatically across the `num_epochs`. Compared to using a static β , how do you expect the specific impact on the balance between `recon_loss` and `kl_loss` after implementing this strategy?

(Please write your answer in the Markdown cell here. You may use LaTeX for mathematical formulas.)

2.4 Evaluation: Visualize Reconstructions (15 pts)

Now that you have completed the training of your conditional VAE, it is time to evaluate its performance by visualizing how well it reconstructs images from the test set. In this section, you will:

- Set your encoder and decoder to evaluation mode.
- Retrieve a batch of test images along with their corresponding labels.
- Use the encoder to compute the latent representations (μ and $\log\sigma$), apply the reparameterization trick to obtain latent vectors, and then use the decoder to reconstruct the images.
- Plot a grid showing the original images in the first row and their corresponding reconstructions in the second row. Each original image should display its true label.

Implement this evaluation code in your notebook. Carefully compare the original images with the reconstructions to assess whether your VAE has captured the essential features of the data. If the reconstructions are poor, consider revisiting your model architecture or training procedure.

```
In [ ]: # -----
# Evaluation: Visualize Reconstructions
# -----
import numpy as np
import matplotlib.pyplot as plt

encoder.eval()
decoder.eval()
test_iter = iter(test_loader)
images, labels = next(test_iter)
images = images.to(device)
labels = labels.to(device).long()

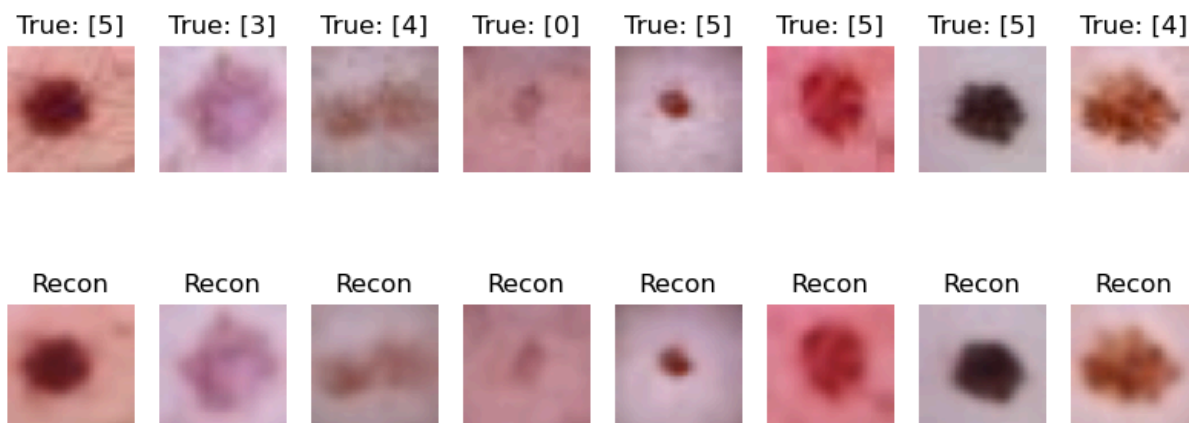
with torch.no_grad():
    # TODO: generate recon (reconstructed images)

plt.figure(figsize=(10, 4))
examples = 8

for i in range(examples):
    # Original image
    plt.subplot(2, examples, i + 1)
    plt.imshow(images[i].transpose(1, 2, 0))
    plt.title(f"True: {labels[i]}")
```

```
plt.axis('off')
# Reconstructed image
plt.subplot(2, examples, examples + i + 1)
plt.imshow(recon[i].transpose(1, 2, 0))
plt.title("Recon")
plt.axis('off')
plt.show()
```

In [11]: *## Expected Output:*



2.5: Visualize Latent Space (15 pts)

Once the model is trained, we want to inspect the **latent space** to see if it has learned a meaningful organization of the data. One way to do this is to take a bunch of images, encode them to get their latent vectors (specifically, we can use the mean μ as the representation for each image), and then visualize these vectors in 2D. Since our latent space might be higher-dimensional (e.g. 16D), we will use **UMAP** (Uniform Manifold Approximation and Projection) to reduce the dimensionality to 2 for visualization. UMAP often preserves local and global structure of the data better than t-SNE or PCA for many cases. **Steps:**

1. Use the trained encoder to compute μ for each image in the test set (concatenate all these latent vectors).
2. Use UMAP to transform the collection of latent vectors into 2D.
3. Create a scatter plot where each point is the 2D UMAP embedding of an image's latent vector, colored by its class label.

Note: You may need to install the UMAP library: `pip install umap-learn`. Also, ensure you have matplotlib for plotting.

```
In [ ]: # If UMAP is not installed, install it: !pip install umap-learn
import umap

encoder.eval() # set encoder to evaluation mode
all_latents = []
all_labels = []
```

```

# Encode all test images to get their latent means
with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        mu, log_var = encoder(images, labels)
        all_latents.append(mu.cpu().numpy())
        all_labels.append(labels.cpu().numpy())

# Stack all batches together
all_latents = np.concatenate(all_latents, axis=0) # shape [N_test_samples,
all_labels = np.concatenate(all_labels, axis=0)   # shape [N_test_samples]

# TODO
# Use UMAP to reduce latent dimensions to 2 for visualization

# Plot the 2D latent space
# TODO: create a scatter plot of latents_2d, colored by all_labels

```

Bonus (Optional): Generating New Samples Conditioned on a Class (10 pts)

As a bonus exercise, you can use the trained VAE to **generate new images** of skin lesions by sampling from the latent space. Because our VAE is conditional, we can direct the generation by choosing a class label.

What to do:

Pick a label (0 through 6, corresponding to one of the skin lesion categories in DermaMNIST), sample random latent vectors from the standard Normal prior $p(z) = \mathcal{N}(0, I)$, and feed them into the decoder along with the chosen label. This will produce novel images that (hopefully) resemble lesions of that class.

This part is open-ended – feel free to experiment by generating multiple samples and visualizing them.

In []: