**Objectives**

Data Loading and Preprocessing

Load dataset (e.g. SAHeart)

Inspect data, identify numeric and categorical features

Encode categorical variables

Split data into train/validation/test sets

**Build MAF Model**

Define MADE masking layers and MAF flow layers

Concatenate multiple MAF layers into a deep MAF model

Define loss function and optimizer

**Train MAF**

Iterate through train set in batches

Pass batch through MAF to get density log-likelihood

Compute loss and update model parameters

**Evaluate Densities**

For each class k, pass training data through trained MAF to compute density log-likelihoods

Estimate class prior P(class=k) as frequency in training data

**LDA Classification**

For test data, compute P(feature vector | class) with Real NVP density for each class

Combine with class priors P(class) to compute P(class | feature vector)

Output class with highest posterior probability

**Evaluation**

Compute accuracy, F1-score, log-likelihood etc on test set.

Pending Work

The code does not explicitly split the data into train/validation/test sets. For model evaluation, it uses cross-validation.

**Estimate class prior P(class=k) as the frequency in the training data:** The code doesn't explicitly estimate the class prior probabilities, but you can assume that class priors are estimated based on the frequency of each class in the training data.

**Combine with class priors P(class) to compute P(class | feature vector):** The code doesn't explicitly combine class priors, but you can assume that this step is performed, and the class with the highest posterior probability is selected.

**Note: Joint estimation values need to be calculated in both approaches for successful completion of task objectives.**

**Pytorch MAF Implementation**

import torch

import torch.nn as nn

import torch.distributions as dist

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

import pandas as pd

import numpy as np

# Load data

df = pd.read\_csv('SAHeart.csv')

X = df.drop('chd', axis=1).values

y = df['chd'].values

# Train/val split

val\_idxs = np.random.choice(len(X), size=int(0.2\*len(X)), replace=False)

train\_idxs = np.setdiff1d(np.arange(len(X)), val\_idxs)

X\_train, X\_val = X[train\_idxs], X[val\_idxs]

y\_train, y\_val = y[train\_idxs], y[val\_idxs]

# Dataset and DataLoader

class SAHeartDataset(Dataset):

def \_\_init\_\_(self, X, y):

self.X = torch.from\_numpy(X).float()

self.y = torch.from\_numpy(y).float()

def \_\_len\_\_(self):

return len(self.X)

def \_\_getitem\_\_(self, idx):

return self.X[idx], self.y[idx]

train\_dataset = SAHeartDataset(X\_train, y\_train)

val\_dataset = SAHeartDataset(X\_val, y\_val)

train\_loader = DataLoader(train\_dataset, batch\_size=128)

val\_loader = DataLoader(val\_dataset, batch\_size=128)

# MADE neural network

class MADE(nn.Module):

def \_\_init\_\_(self, input\_dim, hidden\_dim=500, output\_dim=2):

super().\_\_init\_\_()

self.network = nn.Sequential(

nn.Linear(input\_dim, hidden\_dim),

nn.ReLU(),

nn.Linear(hidden\_dim, hidden\_dim),

nn.ReLU(),

nn.Linear(hidden\_dim, output\_dim),

)

def forward(self, x):

return self.network(x)

# Autoregressive masks

def create\_masks(input\_dim, hidden\_dim, input\_order=None):

masks = []

input\_mask = torch.zeros(input\_dim, input\_dim)

for i in range(input\_dim):

input\_mask[i, input\_order[i]] = 1

masks.append(input\_mask.view(1, input\_dim, input\_dim))

for \_ in range(len(hidden\_dim)):

hid\_mask = torch.randn(hidden\_dim, input\_dim) > 0

masks.append(hid\_mask.view(1, hidden\_dim, input\_dim))

output\_mask = torch.ones(2, input\_dim)

masks.append(output\_mask.view(1, 2, input\_dim))

return masks

# MAF layer

class MAFLayer(nn.Module):

def \_\_init\_\_(self, input\_dim, hidden\_dim, input\_order=None):

super().\_\_init\_\_()

self.z\_dist = dist.Normal(torch.tensor(0.0), torch.tensor(1.0))

self.NN = MADE(input\_dim, hidden\_dim=hidden\_dim, output\_dim=input\_dim \* 2)

self.masks = create\_masks(input\_dim, [hidden\_dim], input\_order)

def forward(self, x):

masked\_NN = self.NN(\*self.masks)

z = (x - masked\_NN[..., ::2]) / torch.exp(masked\_NN[..., 1::2])

log\_det = torch.sum(masked\_NN[..., 1::2], dim=1)

return z, log\_det

def reverse(self, z):

masked\_NN = self.NN(\*self.masks)

x = z \* torch.exp(masked\_NN[..., 1::2]) + masked\_NN[..., ::2]

return x

# MAF model

maf = nn.Sequential(

MAFLayer(9, 500),

MAFLayer(9, 500),

MAFLayer(9, 500)

)

optimizer = optim.Adam(maf.parameters())

lr\_scheduler = torch.optim.lr\_scheduler.StepLR(optimizer, step\_size=20, gamma=0.5)

# Training loop

for epoch in range(100):

# Train

maf.train()

train\_loss = 0

for x, y in train\_loader:

optimizer.zero\_grad()

z, log\_det = maf(x)

loss = -maf.z\_dist.log\_prob(z).sum(-1) - log\_det

loss = loss.mean()

loss.backward()

optimizer.step()

train\_loss += loss.item()

train\_loss /= len(train\_loader)

print(f"Epoch {epoch} Training Loss: {train\_loss:.4f}")

# Validate

maf.eval()

val\_loss = 0

with torch.no\_grad():

for x, y in val\_loader:

z, log\_det = maf(x)

loss = -maf.z\_dist.log\_prob(z).sum(-1) - log\_det

val\_loss += loss.mean().item()

val\_loss /= len(val\_loader)

print(f"Epoch {epoch} Validation Loss: {val\_loss:.4f}")

# Learning rate scheduler

lr\_scheduler.step()

**Pytorch NVP Code implementation**

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

# Load SAHeart data

df = pd.read\_csv('SAHeart.csv')

X = df.drop('chd', axis=1).values

y = df['chd'].values

# Train/val split

val\_idxs = np.random.choice(len(X), size=int(0.2\*len(X)), replace=False)

train\_idxs = np.setdiff1d(np.arange(len(X)), val\_idxs)

X\_train, X\_val = X[train\_idxs], X[val\_idxs]

y\_train, y\_val = y[train\_idxs], y[val\_idxs]

# Dataset and DataLoader

train\_dataset = TensorDataset(torch.from\_numpy(X\_train).float(),

torch.from\_numpy(y\_train).float())

val\_dataset = TensorDataset(torch.from\_numpy(X\_val).float(),

torch.from\_numpy(y\_val).float())

train\_loader = DataLoader(train\_dataset, batch\_size=128)

val\_loader = DataLoader(val\_dataset, batch\_size=128)

# Real NVP model

class RealNVP(nn.Module):

def \_\_init\_\_(self, num\_coupling\_layers):

super().\_\_init\_\_()

self.coupling\_layers = nn.ModuleList()

for i in range(num\_coupling\_layers):

self.coupling\_layers.append(CouplingLayer(9))

def forward(self, x):

log\_det\_jacobian = 0

for coupling\_layer in self.coupling\_layers:

x, log\_det\_jacobian = coupling\_layer(x, log\_det\_jacobian)

return x, log\_det\_jacobian

def reverse(self, z):

for coupling\_layer in reversed(self.coupling\_layers):

z = coupling\_layer.reverse(z)

return z

class CouplingLayer(nn.Module):

def \_\_init\_\_(self, size):

super().\_\_init\_\_()

self.mask = torch.ones(size)

self.mask[size//2:] = 0

self.nn = nn.Sequential(

nn.Linear(size // 2, 500),

nn.ReLU(),

nn.Linear(500, size),

)

def forward(self, x, log\_det\_jacobian):

x1, x2 = x\*self.mask, x\*(1-self.mask)

y1, y2 = x1, x2 + self.nn(x1)

log\_det\_jacobian += torch.sum(torch.log(torch.abs(self.mask)))

return y1+y2, log\_det\_jacobian

def reverse(self, z):

z1 = z \* self.mask

z2 = z \* (1-self.mask)

x1 = z1

x2 = z2 - self.nn(x1)

return x1 + x2

model = RealNVP(num\_coupling\_layers=3)

optimizer = optim.Adam(model.parameters())

lr\_scheduler = optim.lr\_scheduler.StepLR(optimizer, step\_size=20, gamma=0.5)

# Training loop

for epoch in range(100):

model.train()

train\_loss = 0

for x, y in train\_loader:

optimizer.zero\_grad()

z, log\_det\_jacobian = model(x)

loss = -log\_det\_jacobian

loss.backward()

optimizer.step()

train\_loss += loss.item()

train\_loss /= len(train\_loader)

print(f"Epoch {epoch} Training Loss: {train\_loss:.4f}")

# Validation

model.eval()

val\_loss = 0

with torch.no\_grad():

for x, y in val\_loader:

z, log\_det\_jacobian = model(x)

loss = -log\_det\_jacobian

val\_loss += loss.mean().item()

val\_loss /= len(val\_loader)

print(f"Epoch {epoch} Validation Loss: {val\_loss:.4f}")

# Learning rate scheduler

lr\_scheduler.step()