Tutorial for a Multi-Label Classification Neural Network

By: Caitlyn Nguyen

Date: April 17th, 2023

All code is available at this [Github Repository](https://github.com/caitlynmn/Multilabel_Classification_NN_Tutorial): <https://github.com/caitlynmn/Multilabel_Classification_NN_Tutorial>

Table of Contents

[1. Introduction 2](#_Toc132737227)

[2. Data Generation and Splitting 4](#_Toc132737228)

[3. Neural Network Background 6](#_Toc132737229)

[4. Neural Network Set-Up 7](#_Toc132737230)

[5. Training and Validating the Neural Network 10](#_Toc132737231)

[6. Testing the Neural Network 17](#_Toc132737232)

[7. Evaluating the Neural Network 18](#_Toc132737233)

[8. SHAP Values and Feature Importance 19](#_Toc132737234)

[9. Compiled Code 23](#_Toc132737235)

# Introduction

Traditional modelling frameworks are designed to predict a single label, l, from a set of disjoint labels, L. This type of learning problem is known as single-label classification (SLC). When predicting from only = 2 set of disjoint labels, this SLC problem can be called binary classification (BC). When predicting for only one label from a set of mutually exclusive labels where > 2, this SLC is known as multi-class classification (MCC). Both BC and MCC are focused on predicting a single label from the disjoint labels, L. Alternatively, one could predict for a set of multiple related, non-exclusive classes, , where . This type of prediction is known as multi-label classification (MLC).

This tutorial will review how to create a multi-label classification neural network using torch. This tutorial was created using Python 3.10.9, numpy 1.23.5, pandas 1.5.2, matplotlib 3.7.1, scikit-learn 1.2.2, torch 2.0.0, and shap 0.41.0, and jupyter 1.0.0.

Import the necessary libraries for this tutorial using the code below.

**Code:**

import numpy as np

import pandas as pd

import random

import os

import time

import matplotlib.pyplot as plt

from sklearn.datasets import make\_multilabel\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import roc\_auc\_score, precision\_recall\_curve, auc

import torch

import torch.nn as nn

from torch.utils.data import Dataset, DataLoader

from torch.autograd import Variable

import shap

Set the seed for all sources of randomness using the code below for reproducibility. We will be using seed 123 in this tutorial.

**Code:**

def seed\_everything(seed=123):

  random.seed(seed)

  os.environ['PYTHONHASHSEED'] = str(seed)

  np.random.seed(seed)

  torch.manual\_seed(seed)

  torch.backends.cudnn.deterministic = True

  torch.backends.cudnn.benchmark = False

seed\_everything()

There are three custom functions used in this tutorial: sigmoid, bootstrap\_auc and bootstrap\_pr. The sigmoid function will be used to convert the predicted logits to predicted probabilities. The two functions of bootstrap\_auc and bootstrap\_pr will be used to compute the 95% confidence interval for the area under the curve (AUC) of the Receiver-Operator Characteristic Curve and the Precision-Recall Curve, respectfully. This will be done using bootstrapping methods. If ever the sample size is less than 100 and the outcome is rare, the functions will ensure that both cases of the outcome are represented in the dataset by sampling from both negative and positive cases.

**Code:**

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def bootstrap\_auc(y\_true, y\_score, iters):

    orig\_auc = roc\_auc\_score(y\_true, y\_score)

    y\_true0\_ind = list(np.where(y\_true == 0)[0])

    y\_true1\_ind = list(np.where(y\_true == 1)[0])

    samp\_size = len(y\_score)

    estimated\_auc = []

    for i in range(iters):

        if samp\_size < 100:

            samp\_indices = random.choices(y\_true0\_ind, k = len(y\_true0\_ind))

            samp\_indices.extend(random.choices(y\_true1\_ind, k = len(y\_true1\_ind)))

        else:

            samp\_indices = random.choices(range(0, samp\_size), k = samp\_size)

        samp\_score = [y\_score[i] for i in samp\_indices]

        samp\_true = [y\_true[i] for i in samp\_indices]

        estimated\_auc.append(roc\_auc\_score(samp\_true, samp\_score) - orig\_auc)

    lower = np.percentile(estimated\_auc, 2.5)

    upper = np.percentile(estimated\_auc, 97.5)

    return (orig\_auc + lower, orig\_auc + upper)

def bootstrap\_pr(y\_true, y\_score, iters):

    precision, recall, thresholds = precision\_recall\_curve(y\_true, y\_score)

    orig\_auc = auc(recall, precision)

    y\_true0\_ind = list(np.where(y\_true == 0)[0])

    y\_true1\_ind = list(np.where(y\_true == 1)[0])

    samp\_size = len(y\_score)

    estimated\_auc = []

    for i in range(iters):

        if samp\_size < 100:

            samp\_indices = random.choices(y\_true0\_ind, k = len(y\_true0\_ind))

            samp\_indices.extend(random.choices(y\_true1\_ind, k = len(y\_true1\_ind)))

        else:

            samp\_indices = random.choices(range(0, samp\_size), k = samp\_size)

        samp\_score = [y\_score[i] for i in samp\_indices]

        samp\_true = [y\_true[i] for i in samp\_indices]

        precision, recall, thresholds = precision\_recall\_curve(samp\_true, samp\_score)

        auc\_precision\_recall = auc(recall, precision)

        estimated\_auc.append(auc\_precision\_recall - orig\_auc)

    lower = np.percentile(estimated\_auc, 2.5)

    upper = np.percentile(estimated\_auc, 97.5)

    return (orig\_auc + lower, orig\_auc + upper)

# Data Generation and Splitting

We will generate a dataset with multiple labels using the make\_multilabel\_classification()function from scikit-learn. The code below creates a synthetic X and y dataset with 1,000 samples, 30 features, and 2 classes (outputs) each with 2 labels (0 or 1). The random state is set so that this dataset generation is reproducible. I like to convert the arrays to pandas dataframes as they are easier to work with. Printing the X and y dataset shapes confirm that there are 1,000 samples, 30 features, and 2 classes.

**Code:**

X, y = make\_multilabel\_classification(n\_samples = 1000, n\_features = 30, n\_classes = 2, n\_labels = 2, random\_state = 1)

X = pd.DataFrame(X)

y = pd.DataFrame(y, columns = ['Output1', 'Output2'])

print(X.shape, y.shape)

**Output:**

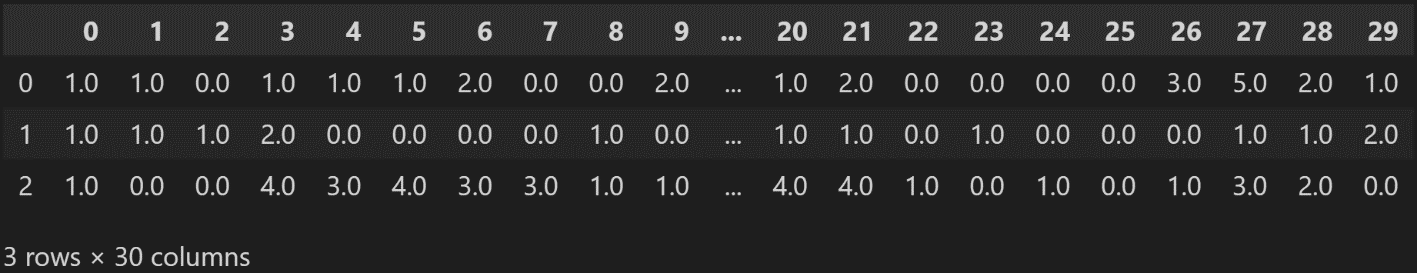
(1000, 30) (1000, 2)

We can further check the first 3 rows of the X and y subsets to take a look at the data.

**Code:**

X.head(3)

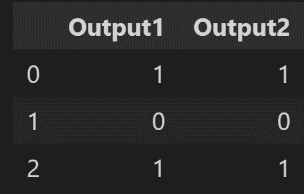
**Output:**



**Code:**

y.head(3)

**Output:**



We can look at the counts of the outcome to determine if there is class imbalance. For this tutorial, we will not be going over how to address class imbalance. Please look at the documentation of the loss functions that are later mentioned to see how to incorporate weights for addressing class imbalance.

**Code:**

print(y.iloc[:,0].value\_counts())

print(y.iloc[:,1].value\_counts())

**Output:**

1 572

0 428

Name: Output1, dtype: int64

1 641

0 359

Name: Output2, dtype: int64

We will now split the data into a training, validation, and testing set in the ratio of 80:10:10 for training:validation:testing. First, we will split the data by a 80:20 to get the 80% training set, then split the remaining 20% in half to get a 10% validation and 10% testing set. The printed shapes of the dataset splits confirms that there was correct 80:10:10 splitting.

**Code:**

X\_train, X\_rem, y\_train, y\_rem = train\_test\_split(X, y, test\_size = 0.20, random\_state = 1)

X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_rem, y\_rem, test\_size = 0.50, random\_state = 1)

print(X\_train.shape, y\_train.shape)

print(X\_val.shape, y\_val.shape)

print(X\_test.shape, y\_test.shape)

**Output:**

(800, 30) (800, 2)

(100, 30) (100, 2)

(100, 30) (100, 2)

# Neural Network Background

A neural network designed to predict for multi-label classification differs from those designed for binary classification and multi-class classification by its number of output nodes, activation function in the output layer, and loss function.

The number of nodes in the output layer should be equal to the number of output classes one is trying to predict. In our example, we have two output classes we are trying to predict, so there should be two nodes in the output layer. A real-world example is trying to predict both mortality status and hospitalization status. Predicting for both mortality and hospitalization would require two output nodes.

The activation function in the output layer depends on the type of data one has in the output node. Multi-class classification problems use a softmax activation function. The softmax activation function takes the data and produces a predicted probability for each class under the assumption that the classes are mutually exclusive. The softmax activation function is , where is the data for the individual’s output nodes, is the number of mutually exclusive output classes, and is the data for the class. The function takes in an input vector,, of dimension for classes, and normalizes the values into predicted probabilities proportional to the exponentials of the input values. The predicted probabilities across classes when using the softmax activation function add to 1. In the case of binary outcome variables (where values are 0 or 1), one should use a sigmoid activation function. A sigmoid activation function takes the data and outputs the predicted probability of the outcome following the formula: . The sigmoid activation function is applied to each output node individually, such that the predicted probability for each class is outputted independently from the outputted probability of other classes.

For a binary classification problem, one would use a binary cross-entropy function. It is advised to use the torch.nn.BCEWithLogitsLoss function (1), where is the batch size, is the weight of the positive class, is the input, and is the output. The BCE with logits loss combines the usage of a sigmoid activation function and a binary cross entropy loss into a single layer, which has proven to be more numerically stable than traditionally using a sigmoid layer followed by the calculation of the binary cross entropy loss. Another advantage of BCE with logits loss is that one could use the pos\_weight argument to weight positive cases higher for imbalanced datasets. If using this loss function as well, **do not apply the sigmoid function to your output layer**. This loss function will automatically apply the sigmoid function to the final outputs when calculating the loss, so adding a sigmoid activation function to the architecture yourself will lead to errors in calculating the loss. To obtain predicting probabilities, apply the sigmoid function to the output.

Binary cross-entropy with logits loss (1):

()

For a multi-class problem, one would use a categorical cross-entropy loss function in a combined LogSoftmax and negative log-likelihood loss layer described in (2), where is the batch size, is the number of classes, is the class weight, is the input, and is the output. This can be incorporated using torch.nn.CrossEntropyLoss. **Do not apply the softmax activation function in your output layer if using this loss function**, as it is applied in the calculation of the loss. Class weights can be used using the weight argument. To obtain predicted probabilities, apply the softmax function to the outputted predicted logits.

Categorical cross-entropy loss function (2):

(2)

In multi-label classification, the binary cross-entropy loss value in (3) is calculated for each label, where is the batch size, is the label number, is the weight of the positive class, is the input, and is the outpu. The total loss would then be the summation of each label’s binary-cross entropy loss, . In this way, the total loss can be backpropagated for optimization of parameter weights in the network, seeking to minimize the total loss from both labels.

Multi-label binary cross-entropy loss function (3):

(3)

For this tutorial, we will be using the torch.nn.BCEWithLogitsLoss function to calculate the loss of each individual binary output class. We will not

# Neural Network Set-Up

For torch, you need to first create a class to convert the dataset into the torch Dataset type.

**Code:**

class Tutorial\_DS(Dataset):

    def \_\_init\_\_(self, X, y):

        self.x = X.copy()

        self.labels = y.copy()

    # Shape function

    def shape(self):

        return self.x.shape, self.labels.shape

    # Length function, needed for DataLoader

    def \_\_len\_\_(self):

        return len(self.labels.iloc[:,0])

    # Return function

    def \_\_getitem\_\_(self, idx):

        return self.x.iloc[idx].values.astype(np.float32), self.labels.iloc[idx].values.astype(np.float32)

train\_ds = Tutorial\_DS(X\_train, y\_train)

val\_ds = Tutorial\_DS(X\_val, y\_val)

test\_ds = Tutorial\_DS(X\_test, y\_test)

print(train\_ds.shape())

print(val\_ds.shape())

print(test\_ds.shape())

**Output:**

((800, 30), (800, 2))

((100, 30), (100, 2))

((100, 30), (100, 2))

Load the data into DataLoader, which will shuffle the data into batches of our desired size. We will be using a batch size of 64.

**Code:**

batch\_size = 64

train\_dl = DataLoader(train\_ds, batch\_size = batch\_size, shuffle = True)

val\_dl = DataLoader(val\_ds, batch\_size = batch\_size, shuffle = True)

test\_dl = DataLoader(test\_ds, batch\_size = batch\_size, shuffle = True)

We will then create a class for our model. Our model takes in the inputs of the number of input features, desired number of hidden units, and the number of labels for each class. The neural network has a fully-connected input layer, a ReLU activation function, and two output nodes, 1 for each output. The predicted logits for predicting each class is returned.

**Code:**

class MLC(nn.Module):

    def \_\_init\_\_(self, input\_dim, hidden\_units, n\_labels1, n\_labels2):

        # Inherit from torch.nn.Module

        super(MLC, self).\_\_init\_\_()

        # Assign parameters to model attributes

        self.input\_dim = input\_dim

        self.n\_units = hidden\_units

        self.output1\_labels = n\_labels1

        self.output2\_labels = n\_labels2

        # Define network layers

        ## First fully-connected layer

        self.fc1 = nn.Linear(in\_features = self.input\_dim, out\_features = self.n\_units) # input: input\_dim, output: hidden\_units

        ## ReLU activation function

        self.relu = nn.ReLU()

        ## Output fully-connected layers

        self.fc\_output1 = nn.Linear(in\_features = self.n\_units, out\_features = self.output1\_labels) # input: hidden\_units, output: n\_labels1

        self.fc\_output2 = nn.Linear(in\_features = self.n\_units, out\_features = self.output2\_labels) # input: hidden\_units, output: n\_labels2

        # Forward pass method

    def forward(self, X):

        X = self.fc1(X)

        X = self.relu(X)

        logits\_1 = self.fc\_output1(X)

        logits\_2 = self.fc\_output2(X)

        return logits\_1, logits\_2

We will now define the parameters to be loaded into the model.

**Code:**

# Define model parameters

input\_dim = X\_train.shape[1]

hidden\_units = int(2/3\*input\_dim)

n\_labels1 = np.unique(y\_train['Output1'].values.ravel()).max()

n\_labels2 = np.unique(y\_train['Output2'].values.ravel()).max()

print(input\_dim, hidden\_units, n\_labels1, n\_labels2)

**Output:**

30 20 1 1

Next, we instantiate the model, loss function, optimizer, number of epochs, and early stopping min\_delta and patience. We will use the loss function of torch.nn.BCEWithLogitsLoss for both of our outputs as they are both binary. A learning rate of 0.001 will be used with the torch.optim.Adam optimizer. For early stopping, a minimum delta of 0.001 and a patience of 10 will be used.

**Code:**

# Instantiate model

model = MLC(input\_dim = input\_dim, hidden\_units = hidden\_units, n\_labels1 = n\_labels1, n\_labels2 = n\_labels2)

# Number of epochs

n\_epochs = 200

# Loss functions

loss\_func1 = torch.nn.BCEWithLogitsLoss()

loss\_func2 = torch.nn.BCEWithLogitsLoss()

# Optimizer and learning rate

lr = 0.001

optimizer = torch.optim.Adam(model.parameters(), lr = lr)

# Early stopping

min\_delta = 0.001

patience = 10

print(model)

**Output:**

MLC(

(fc1): Linear(in\_features=30, out\_features=20, bias=True)

(relu): ReLU()

(fc\_output1): Linear(in\_features=20, out\_features=1, bias=True)

(fc\_output2): Linear(in\_features=20, out\_features=1, bias=True)

)

# Training and Validating the Neural Network

We first need to initialize an empty early stopping count, train and validation score and loss vectors that will be used in each loop. Then, we will iterate through each epoch. For each epoch, we will iterate through each batch. For each batch, the optimizer will be zeroed and there will be a forward pass of the data which will produce the summed loss of the predictions on each outcome. Then, the summed loss will be propagated backwards and the model parameters will be updated accordingly. After all training batches have been passed through the model, we will evaluate the model performance by passing the batches of the validation set through the model and obtaining resultant validation loss and area under the curve (AUC). Early stopping will be applied if at the end of the epoch, there have been 10 consecutive decreases (with a minimum decrease of 0.001) from the minimum validation loss.

**Code:**

# Initialize zeroed count and vectors

early\_stopping\_count = 0

auc\_train\_scores = []

auc\_val\_scores = []

loss\_train\_scores = []

loss\_val\_scores = []

# Start time

time\_start = time.time()

print('\nTraining model...\n')

for epoch in range(n\_epochs):

    # Initialize batch actual values, predictions, and loss

    actual\_train\_output1 = []

    pred\_train\_output1 = []

    actual\_train\_output2 = []

    pred\_train\_output2 = []

    actual\_val\_output1 = []

    pred\_val\_output1 = []

    actual\_val\_output2 = []

    pred\_val\_output2 = []

    train\_loss1\_sum = 0

    train\_loss2\_sum = 0

    val\_loss1\_sum = 0

    val\_loss2\_sum = 0

    # Place model in training model

    model.train()

    # Iterate through training data batches

    for x, y in train\_dl:

        # Zero out the gradients

        optimizer.zero\_grad()

        # Forward pass

        pred\_y1, pred\_y2 = model(x)

        # Sum losses

        loss1 = loss\_func1(pred\_y1[:, 0], y[:, 0])

        loss2 = loss\_func2(pred\_y2[:, 0], y[:, 1])

        loss = loss1 + loss2

        # Backward pass

        loss.backward()

        optimizer.step()

        # Update vectors with real values and predictions

        actual\_train\_output1.extend(pd.DataFrame(y.detach().numpy())[0].to\_numpy())

        pred\_train\_output1.extend(sigmoid(pred\_y1[:, 0].detach().numpy()))

        actual\_train\_output2.extend(pd.DataFrame(y.detach().numpy())[1].to\_numpy())

        pred\_train\_output2.extend(sigmoid(pred\_y2[:, 0].detach().numpy()))

        # Update individual loss for display

        train\_loss1\_sum += loss1

        train\_loss2\_sum += loss2

    # Calculate epoch training loss and AUC

    train\_loss\_final = train\_loss1\_sum + train\_loss2\_sum

    train\_auc1 = roc\_auc\_score(actual\_train\_output1, pred\_train\_output1)

    train\_auc2 = roc\_auc\_score(actual\_train\_output2, pred\_train\_output2)

    # Add final training loss and AUC to vectors

    loss\_train\_scores.append([train\_loss\_final.detach().numpy(), train\_loss1\_sum.detach().numpy(), train\_loss2\_sum.detach().numpy()])

    auc\_train\_scores.append([train\_auc1, train\_auc2])

    # Put model into evaluation mode for validation

    model.eval()

    # Iterate through validation data batches

    for x, y in val\_dl:

        # Zero out the gradients

        optimizer.zero\_grad()

        # Forward pass

        pred\_y1, pred\_y2 = model(x)

        # Sum losses

        loss1 = loss\_func1(pred\_y1[:, 0], y[:, 0])

        loss2 = loss\_func2(pred\_y2[:, 0], y[:, 1])

        loss = loss1 + loss2

        # Update vectors with real values and predictions

        actual\_val\_output1.extend(pd.DataFrame(y.detach().numpy())[0].to\_numpy())

        pred\_val\_output1.extend(sigmoid(pred\_y1[:, 0].detach().numpy()))

        actual\_val\_output2.extend(pd.DataFrame(y.detach().numpy())[1].to\_numpy())

        pred\_val\_output2.extend(sigmoid(pred\_y2[:, 0].detach().numpy()))

        # Update individual loss for display

        val\_loss1\_sum += loss1

        val\_loss2\_sum += loss2

    # Calculate epoch validation loss and AUC

    val\_loss\_final = val\_loss1\_sum + val\_loss2\_sum

    val\_auc1 = roc\_auc\_score(actual\_val\_output1, pred\_val\_output1)

    val\_auc2 = roc\_auc\_score(actual\_val\_output2, pred\_val\_output2)

    # Add final validation loss and AUC to vectors

    loss\_val\_scores.append([val\_loss\_final.detach().numpy(), val\_loss1\_sum.detach().numpy(), val\_loss2\_sum.detach().numpy()])

    auc\_val\_scores.append([val\_auc1, val\_auc2])

    # Display epoch progress

    prog\_disp\_freq = 1

    if (((epoch + 1) % prog\_disp\_freq) == 0) | (epoch == 0) | ((epoch + 1) == n\_epochs):

        print("EPOCH: %s | TRAIN. (Output1 AUC: %5.3f  Output2 AUC: %5.3f   Loss: %5.3f) | VAL. (Output1 AUC: %5.3f  Output2 AUC: %5.3f   Loss: %5.3f)" % \

            (epoch, train\_auc1, train\_auc2, train\_loss\_final, val\_auc1, val\_auc2, val\_loss\_final))

    # Early stopping

    if epoch == 0:

        best\_loss = loss\_val\_scores[0][0]

    else:

        if loss\_val\_scores[epoch][0] >= (best\_loss - min\_delta):

            early\_stopping\_count += 1

        else:

            best\_loss = loss\_val\_scores[epoch][0]

            early\_stopping\_count = 0

    if early\_stopping\_count == patience:

        break

time\_end = time.time()

print('\nModel training complete.\n')

print('Time to train model: %.1f s\n' % (time\_end - time\_start))

**Output:**

Output exceeds the [size limit](command:workbench.action.openSettings?%5B%22notebook.output.textLineLimit%22%5D). Open the full output data [in a text editor](command:workbench.action.openLargeOutput?065c3b86-d9d6-41e3-9e12-674eef187724)

Training model...

EPOCH: 0 | TRAIN. (Output1 AUC: 0.796 Output2 AUC: 0.715 Loss: 16.061) | VAL. (Output1 AUC: 0.878 Output2 AUC: 0.668 Loss: 2.313)

EPOCH: 1 | TRAIN. (Output1 AUC: 0.897 Output2 AUC: 0.766 Loss: 14.600) | VAL. (Output1 AUC: 0.915 Output2 AUC: 0.714 Loss: 2.162)

EPOCH: 2 | TRAIN. (Output1 AUC: 0.933 Output2 AUC: 0.806 Loss: 13.395) | VAL. (Output1 AUC: 0.934 Output2 AUC: 0.764 Loss: 2.006)

EPOCH: 3 | TRAIN. (Output1 AUC: 0.955 Output2 AUC: 0.848 Loss: 12.140) | VAL. (Output1 AUC: 0.947 Output2 AUC: 0.803 Loss: 1.807)

EPOCH: 4 | TRAIN. (Output1 AUC: 0.965 Output2 AUC: 0.882 Loss: 10.843) | VAL. (Output1 AUC: 0.955 Output2 AUC: 0.839 Loss: 1.675)

EPOCH: 5 | TRAIN. (Output1 AUC: 0.973 Output2 AUC: 0.915 Loss: 9.877) | VAL. (Output1 AUC: 0.966 Output2 AUC: 0.882 Loss: 1.524)

EPOCH: 6 | TRAIN. (Output1 AUC: 0.978 Output2 AUC: 0.937 Loss: 8.876) | VAL. (Output1 AUC: 0.973 Output2 AUC: 0.904 Loss: 1.358)

EPOCH: 7 | TRAIN. (Output1 AUC: 0.981 Output2 AUC: 0.954 Loss: 8.042) | VAL. (Output1 AUC: 0.974 Output2 AUC: 0.928 Loss: 1.257)

EPOCH: 8 | TRAIN. (Output1 AUC: 0.983 Output2 AUC: 0.961 Loss: 7.342) | VAL. (Output1 AUC: 0.975 Output2 AUC: 0.935 Loss: 1.177)

EPOCH: 9 | TRAIN. (Output1 AUC: 0.985 Output2 AUC: 0.965 Loss: 6.791) | VAL. (Output1 AUC: 0.979 Output2 AUC: 0.939 Loss: 1.100)

EPOCH: 10 | TRAIN. (Output1 AUC: 0.985 Output2 AUC: 0.968 Loss: 6.338) | VAL. (Output1 AUC: 0.983 Output2 AUC: 0.944 Loss: 1.020)

EPOCH: 11 | TRAIN. (Output1 AUC: 0.987 Output2 AUC: 0.970 Loss: 5.996) | VAL. (Output1 AUC: 0.983 Output2 AUC: 0.946 Loss: 0.964)

EPOCH: 12 | TRAIN. (Output1 AUC: 0.987 Output2 AUC: 0.973 Loss: 5.674) | VAL. (Output1 AUC: 0.984 Output2 AUC: 0.949 Loss: 0.962)

EPOCH: 13 | TRAIN. (Output1 AUC: 0.988 Output2 AUC: 0.975 Loss: 5.391) | VAL. (Output1 AUC: 0.985 Output2 AUC: 0.952 Loss: 0.924)

EPOCH: 14 | TRAIN. (Output1 AUC: 0.988 Output2 AUC: 0.976 Loss: 5.198) | VAL. (Output1 AUC: 0.986 Output2 AUC: 0.955 Loss: 0.858)

EPOCH: 15 | TRAIN. (Output1 AUC: 0.988 Output2 AUC: 0.978 Loss: 4.965) | VAL. (Output1 AUC: 0.987 Output2 AUC: 0.956 Loss: 0.854)

EPOCH: 16 | TRAIN. (Output1 AUC: 0.988 Output2 AUC: 0.979 Loss: 4.850) | VAL. (Output1 AUC: 0.987 Output2 AUC: 0.958 Loss: 0.878)

EPOCH: 17 | TRAIN. (Output1 AUC: 0.989 Output2 AUC: 0.980 Loss: 4.721) | VAL. (Output1 AUC: 0.988 Output2 AUC: 0.960 Loss: 0.821)

EPOCH: 18 | TRAIN. (Output1 AUC: 0.989 Output2 AUC: 0.981 Loss: 4.577) | VAL. (Output1 AUC: 0.988 Output2 AUC: 0.961 Loss: 0.800)

EPOCH: 19 | TRAIN. (Output1 AUC: 0.989 Output2 AUC: 0.982 Loss: 4.399) | VAL. (Output1 AUC: 0.989 Output2 AUC: 0.962 Loss: 0.746)

EPOCH: 20 | TRAIN. (Output1 AUC: 0.989 Output2 AUC: 0.983 Loss: 4.386) | VAL. (Output1 AUC: 0.989 Output2 AUC: 0.964 Loss: 0.775)

EPOCH: 21 | TRAIN. (Output1 AUC: 0.989 Output2 AUC: 0.984 Loss: 4.191) | VAL. (Output1 AUC: 0.989 Output2 AUC: 0.967 Loss: 0.771)

...

Model training complete.

Time to train model: 14.5 s

We can visualize the training and validation AUC and loss for each output over epoch.

**Code:**

# AUC Plot

fig = plt.figure()

fig.set\_facecolor('white')

auc\_train\_scores = pd.DataFrame(auc\_train\_scores)

auc\_val\_scores = pd.DataFrame(auc\_val\_scores)

plot\_lines = []

l1, = plt.plot(auc\_train\_scores[0], color = 'blue', linewidth = 2, linestyle = '-')

l2, = plt.plot(auc\_val\_scores[0], color = 'blue', linewidth = 2, linestyle = '--')

l3, = plt.plot(auc\_train\_scores[1], color = 'red', linewidth = 2, linestyle = '-')

l4, = plt.plot(auc\_val\_scores[1], color = 'red', linewidth = 2, linestyle = '--')

plot\_lines.append([l1, l2, l3, l4])

legend1 = plt.legend(plot\_lines[0], ["Training", "Validation"], loc = "upper center", bbox\_to\_anchor = (0.75, -0.15), fancybox = True)

plt.legend([plot\_lines[0][0], plot\_lines[0][2]], ["Output1 AUC", "Output2 AUC"], loc = "upper center", bbox\_to\_anchor = (0.25, -0.15), fancybox = True)

plt.gca().add\_artist(legend1)

plt.title("AUC over epochs", fontsize = 14)

plt.xlabel("Epochs")

plt.show()

**Output**

Chart

Description automatically generated with medium confidence

**Code:**

# Loss Plot

fig = plt.figure()

fig.set\_facecolor('white')

loss\_train\_scores = pd.DataFrame(loss\_train\_scores)

loss\_val\_scores = pd.DataFrame(loss\_val\_scores)

plot\_lines = []

l1, = plt.plot(loss\_train\_scores[0], color = 'blue', linewidth = 2, linestyle = '-')

l2, = plt.plot(loss\_val\_scores[0], color = 'blue', linewidth = 2, linestyle = '--')

l3, = plt.plot(loss\_train\_scores[1], color = 'red', linewidth = 2, linestyle = '-')

l4, = plt.plot(loss\_val\_scores[1], color = 'red', linewidth = 2, linestyle = '--')

plot\_lines.append([l1, l2, l3, l4])

legend1 = plt.legend(plot\_lines[0], ["Training", "Validation"], loc = "upper center", bbox\_to\_anchor = (0.75, -0.15), fancybox = True)

plt.legend([plot\_lines[0][0], plot\_lines[0][2]], ["Output1 Loss", "Output2 Loss"], loc = "upper center", bbox\_to\_anchor = (0.25, -0.15), fancybox = True)

plt.gca().add\_artist(legend1)

plt.title("Loss over epochs", fontsize = 14)

plt.xlabel("Epochs")

plt.show()

**Output:**

Chart

Description automatically generated

# Testing the Neural Network

Run the testing data through the neural network in the evaluation mode to obtain predicted probabilities on the testing dataset.

**Code:**

# Running testing set through model

# Initialize test set vectors

actual\_test\_output1 = []

pred\_test\_output1 = []

actual\_test\_output2 = []

pred\_test\_output2 = []

# Place model in evaluation mode

model.eval()

# Zero out the gradients

optimizer.zero\_grad()

# Iterate through test data batches

for x, y in test\_dl:

    # Forward pass

    pred\_y1, pred\_y2 = model(x)

    # Update vectors with real values and predictions

    actual\_test\_output1.extend(pd.DataFrame(y.detach().numpy())[0].to\_numpy())

    pred\_test\_output1.extend(sigmoid(pred\_y1[:, 0].detach().numpy()))

    actual\_test\_output2.extend(pd.DataFrame(y.detach().numpy())[1].to\_numpy())

    pred\_test\_output2.extend(sigmoid(pred\_y2[:, 0].detach().numpy()))

# Evaluating the Neural Network

We can evaluate the predictions on the testing set from the neural network by finding the area under the curve (AUC) for the Receiver-Operator Characteristic Curve and the Precision-Recall curve. We can find the 95% bootstrapped confidence intervals using our bootstrapping functions defined previously.

**Code:**

# Calculate test AUC for Receiver-Operator Characteristic Curve

test\_auc\_y1 = roc\_auc\_score(actual\_test\_output1, pred\_test\_output1)

test\_auc\_y2 = roc\_auc\_score(actual\_test\_output2, pred\_test\_output2)

[lower1, upper1] = bootstrap\_auc(np.array(actual\_test\_output1), pred\_test\_output1, 1000)

[lower2, upper2] = bootstrap\_auc(np.array(actual\_test\_output2), pred\_test\_output2, 1000)

print("Testing set 95%% AUC CI for Output 1: %5.3f [%5.3f, %5.3f]" % (test\_auc\_y1, lower1, upper1))

print("Testing set 95%% AUC CI for Output 2: %5.3f [%5.3f, %5.3f]" % (test\_auc\_y2, lower2, upper2))

**Output:**

Testing set 95% AUC CI for Output 1: 0.986 [0.967, 0.998]

Testing set 95% AUC CI for Output 2: 0.979 [0.950, 0.997]

**Code:**

# Calculate test AUC for Precision-Recall Curve

precision, recall, thresholds = precision\_recall\_curve(actual\_test\_output1, pred\_test\_output1)

auc\_pr1 = auc(recall, precision)

precision, recall, thresholds = precision\_recall\_curve(actual\_test\_output2, pred\_test\_output2)

auc\_pr2 = auc(recall, precision)

[lower1, upper1] = bootstrap\_pr(np.array(actual\_test\_output1), pred\_test\_output1, 1000)

[lower2, upper2] = bootstrap\_pr(np.array(actual\_test\_output2), pred\_test\_output2, 1000)

print("Testing set 95%% AUPRC CI for Output 1: %5.3f [%5.3f, %5.3f]" % (auc\_pr1, lower1, upper1))

print("Testing set 95%% AUPRC CI for Output 2: %5.3f [%5.3f, %5.3f]" % (auc\_pr2, lower2, upper2))

**Output:**

Testing set 95% AUPRC CI for Output 1: 0.986 [0.965, 0.999]

Testing set 95% AUPRC CI for Output 2: 0.985 [0.965, 0.999]

# SHAP Values and Feature Importance

The SHAP explainer does not integrate well with tensors, so we have to use a function to wrap the model to convert the tensors to numpy. We then convert the training and testing dataset to numpy. With large datasets, using a large number of observations in the background of the explainer may lead to large computational time. In this tutorial, we will only use the first 100 observations for simplicity.

**Code:**

# SHAP values for outcome 1

# Define function to wrap model to transform data to numpy

f = lambda x: model(Variable(torch.from\_numpy(x)))[0].detach().numpy()

# Convert pandas dataframe to numpy

train\_vals = X\_train.to\_numpy(dtype = np.float32)

test\_vals = X\_test.to\_numpy(dtype = np.float32)

# SHAP explainer, requires conversion from tensor to numpy using function f

explainer = shap.KernelExplainer(f, train\_vals[:100])

shap\_values\_y1 = explainer.shap\_values(test\_vals)

From these values, we can get a violin plot to display the impact of each variable on each individual. We can also get a boxplot which shows the average importance (mean(|SHAP value|)) for each feature.

**Code:**

# Getting predictor names

feature\_names = X\_train.columns

# SHAP violin plot for outcome 1

fig = plt.figure()

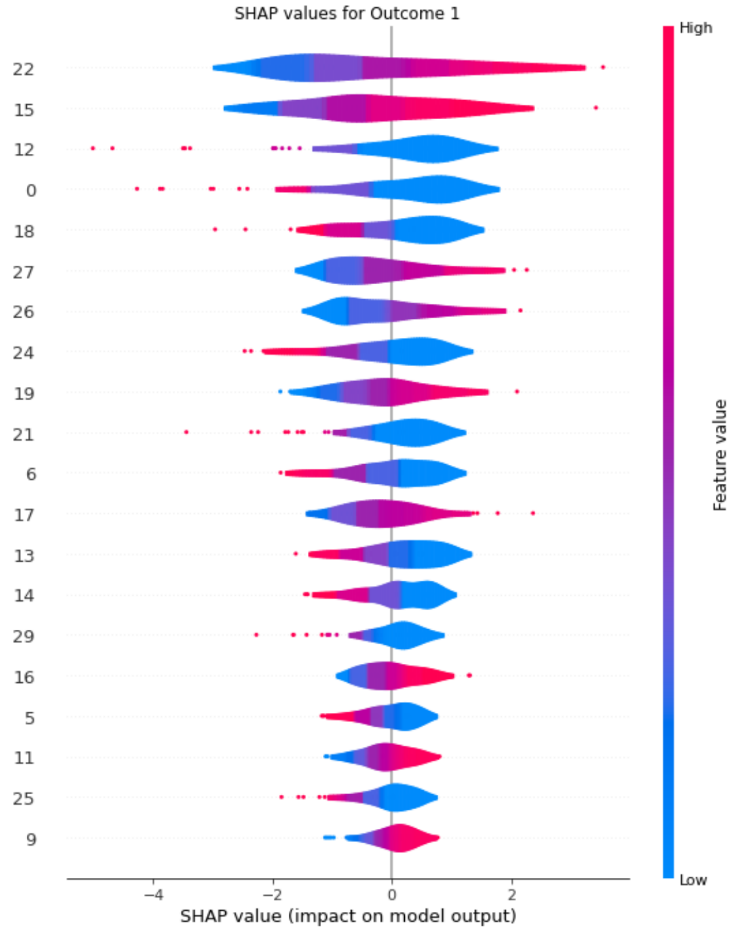
fig.set\_facecolor('white')

shap.summary\_plot(shap\_values\_y1[0], X\_test, plot\_type = "violin", feature\_names = feature\_names, show = False)

plt.title("SHAP values for Outcome 1")

plt.show()

**Output:**



**Code:**

# SHAP barplot for outcome 1

fig = plt.figure()

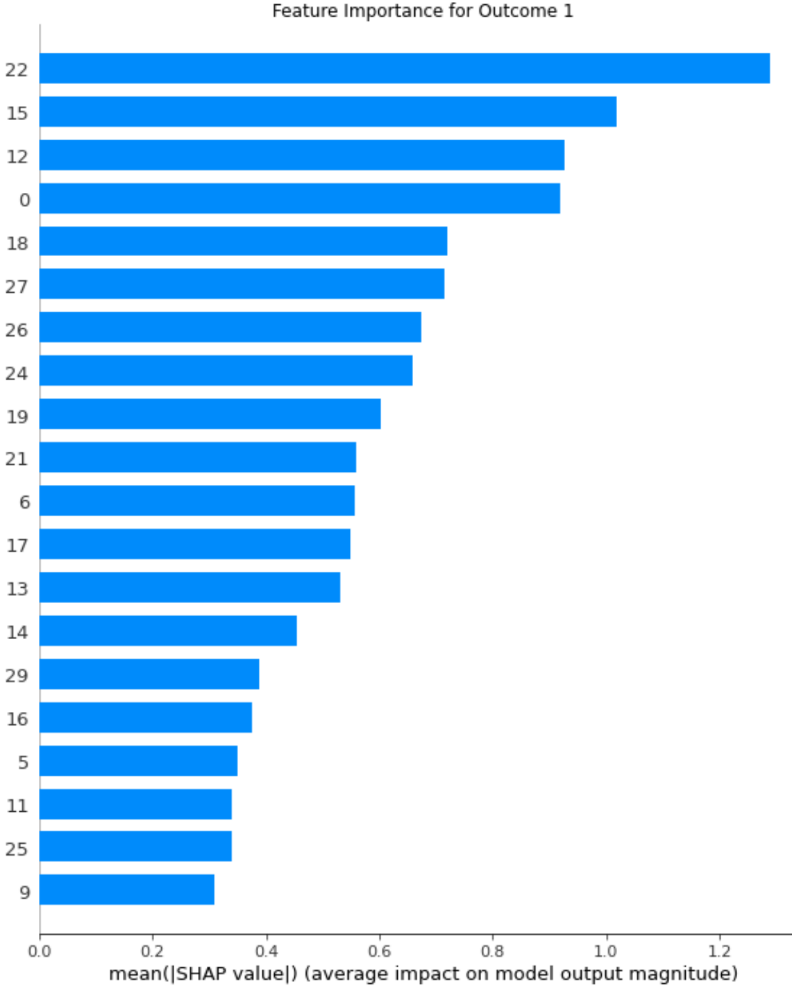
fig.set\_facecolor('white')

shap.summary\_plot(shap\_values\_y1[0], X\_test, plot\_type = "bar", feature\_names = feature\_names, show = False)

plt.title("Feature Importance for Outcome 1")

plt.show()

**Output:**



We can repeat the same steps for outcome 2.

**Code:**

# SHAP values for outcome 2

# Define function to wrap model to transform data to numpy

f = lambda x: model(Variable(torch.from\_numpy(x)))[1].detach().numpy()

# Convert pandas dataframe to numpy

train\_vals = X\_train.to\_numpy(dtype = np.float32)

test\_vals = X\_test.to\_numpy(dtype = np.float32)

# SHAP explainer, requires conversion from tensor to numpy using function f

explainer = shap.KernelExplainer(f, train\_vals[:100])

shap\_values\_y2 = explainer.shap\_values(test\_vals)

**Code:**

# SHAP violin plot for outcome 2

fig = plt.figure()

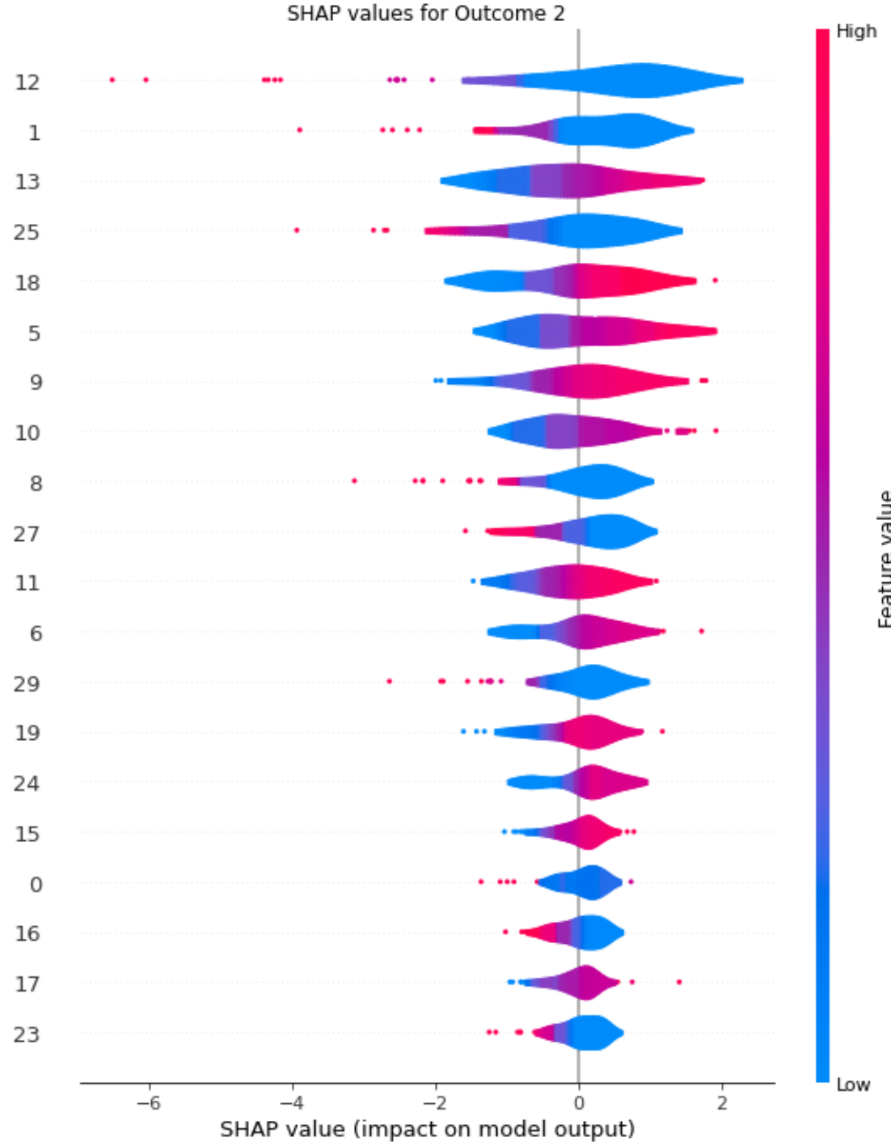
fig.set\_facecolor('white')

shap.summary\_plot(shap\_values\_y2[0], X\_test, plot\_type = "violin", feature\_names = feature\_names, show = False)

plt.title("SHAP values for Outcome 2")

plt.show()

**Output:**



**Code:**

# SHAP barplot for outcome 2

fig = plt.figure()

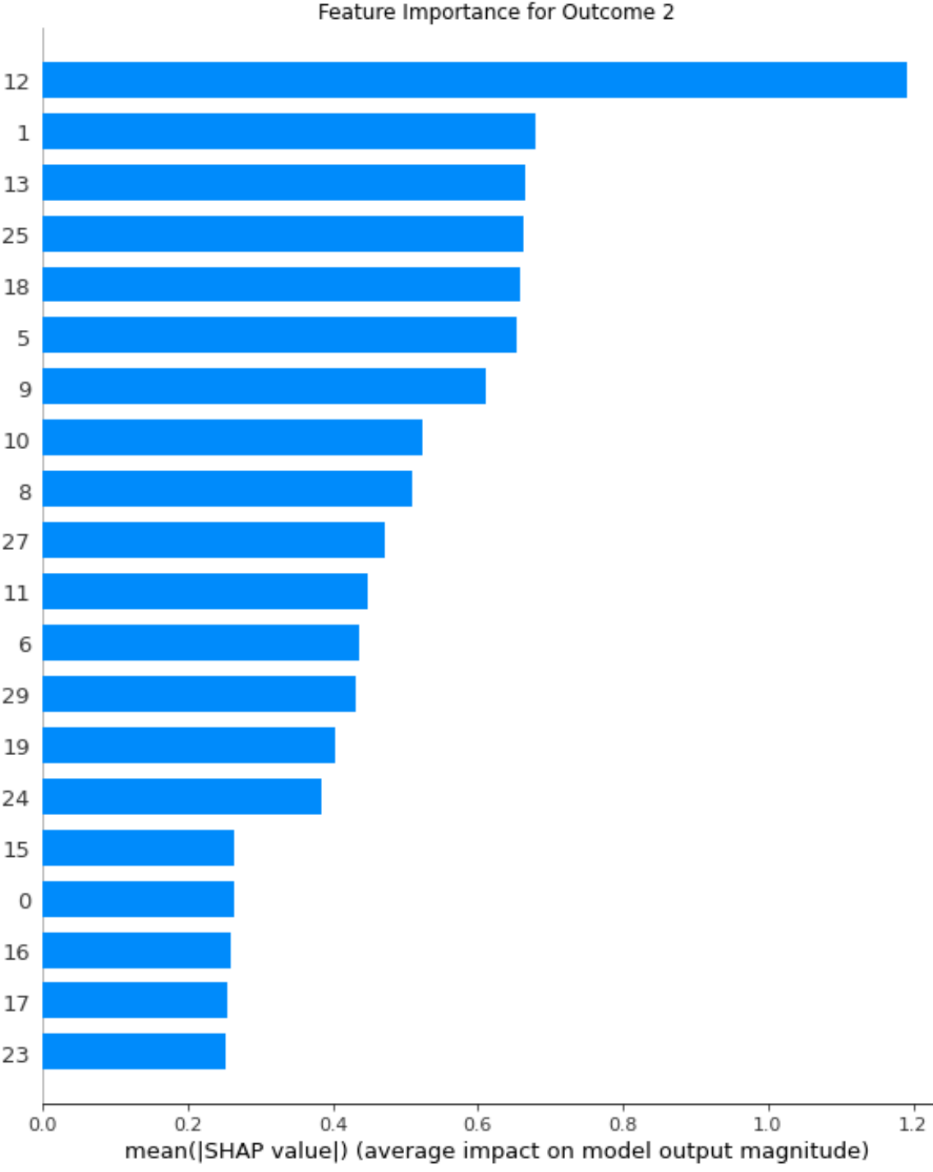
fig.set\_facecolor('white')

shap.summary\_plot(shap\_values\_y2[0], X\_test, plot\_type = "bar", feature\_names = feature\_names, show = False)

plt.title("Feature Importance for Outcome 2")

plt.show()

**Output:**



# Compiled Code

Below is the code used in this tutorial all compiled together.

**Code:**

import numpy as np

import pandas as pd

import random

import os

import time

import matplotlib.pyplot as plt

from sklearn.datasets import make\_multilabel\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import roc\_auc\_score, precision\_recall\_curve, auc

import torch

import torch.nn as nn

from torch.utils.data import Dataset, DataLoader

from torch.autograd import Variable

import shap

def seed\_everything(seed=123):

  random.seed(seed)

  os.environ['PYTHONHASHSEED'] = str(seed)

  np.random.seed(seed)

  torch.manual\_seed(seed)

  torch.backends.cudnn.deterministic = True

  torch.backends.cudnn.benchmark = False

seed\_everything()

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def bootstrap\_auc(y\_true, y\_score, iters):

    orig\_auc = roc\_auc\_score(y\_true, y\_score)

    y\_true0\_ind = list(np.where(y\_true == 0)[0])

    y\_true1\_ind = list(np.where(y\_true == 1)[0])

    samp\_size = len(y\_score)

    estimated\_auc = []

    for i in range(iters):

        if samp\_size < 100:

            samp\_indices = random.choices(y\_true0\_ind, k = len(y\_true0\_ind))

            samp\_indices.extend(random.choices(y\_true1\_ind, k = len(y\_true1\_ind)))

        else:

            samp\_indices = random.choices(range(0, samp\_size), k = samp\_size)

        samp\_score = [y\_score[i] for i in samp\_indices]

        samp\_true = [y\_true[i] for i in samp\_indices]

        estimated\_auc.append(roc\_auc\_score(samp\_true, samp\_score) - orig\_auc)

    lower = np.percentile(estimated\_auc, 2.5)

    upper = np.percentile(estimated\_auc, 97.5)

    return (orig\_auc + lower, orig\_auc + upper)

def bootstrap\_pr(y\_true, y\_score, iters):

    precision, recall, thresholds = precision\_recall\_curve(y\_true, y\_score)

    orig\_auc = auc(recall, precision)

    y\_true0\_ind = list(np.where(y\_true == 0)[0])

    y\_true1\_ind = list(np.where(y\_true == 1)[0])

    samp\_size = len(y\_score)

    estimated\_auc = []

    for i in range(iters):

        if samp\_size < 100:

            samp\_indices = random.choices(y\_true0\_ind, k = len(y\_true0\_ind))

            samp\_indices.extend(random.choices(y\_true1\_ind, k = len(y\_true1\_ind)))

        else:

            samp\_indices = random.choices(range(0, samp\_size), k = samp\_size)

        samp\_score = [y\_score[i] for i in samp\_indices]

        samp\_true = [y\_true[i] for i in samp\_indices]

        precision, recall, thresholds = precision\_recall\_curve(samp\_true, samp\_score)

        auc\_precision\_recall = auc(recall, precision)

        estimated\_auc.append(auc\_precision\_recall - orig\_auc)

    lower = np.percentile(estimated\_auc, 2.5)

    upper = np.percentile(estimated\_auc, 97.5)

    return (orig\_auc + lower, orig\_auc + upper)

X, y = make\_multilabel\_classification(n\_samples = 1000, n\_features = 30, n\_classes = 2, n\_labels = 2, random\_state = 1)

X = pd.DataFrame(X)

y = pd.DataFrame(y, columns = ['Output1', 'Output2'])

print(X.shape, y.shape)

X.head(3)

y.head(3)

print(y.iloc[:,0].value\_counts())

print(y.iloc[:,1].value\_counts())

X\_train, X\_rem, y\_train, y\_rem = train\_test\_split(X, y, test\_size = 0.20, random\_state = 1)

X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_rem, y\_rem, test\_size = 0.50, random\_state = 1)

print(X\_train.shape, y\_train.shape)

print(X\_val.shape, y\_val.shape)

print(X\_test.shape, y\_test.shape)

class Tutorial\_DS(Dataset):

    def \_\_init\_\_(self, X, y):

        self.x = X.copy()

        self.labels = y.copy()

    # Shape function

    def shape(self):

        return self.x.shape, self.labels.shape

    # Length function, needed for DataLoader

    def \_\_len\_\_(self):

        return len(self.labels.iloc[:,0])

    # Return function

    def \_\_getitem\_\_(self, idx):

        return self.x.iloc[idx].values.astype(np.float32), self.labels.iloc[idx].values.astype(np.float32)

train\_ds = Tutorial\_DS(X\_train, y\_train)

val\_ds = Tutorial\_DS(X\_val, y\_val)

test\_ds = Tutorial\_DS(X\_test, y\_test)

print(train\_ds.shape())

print(val\_ds.shape())

print(test\_ds.shape())

batch\_size = 64

train\_dl = DataLoader(train\_ds, batch\_size = batch\_size, shuffle = True)

val\_dl = DataLoader(val\_ds, batch\_size = batch\_size, shuffle = True)

test\_dl = DataLoader(test\_ds, batch\_size = batch\_size, shuffle = True)

class MLC(nn.Module):

    def \_\_init\_\_(self, input\_dim, hidden\_units, n\_labels1, n\_labels2):

        # Inherit from torch.nn.Module

        super(MLC, self).\_\_init\_\_()

        # Assign parameters to model attributes

        self.input\_dim = input\_dim

        self.n\_units = hidden\_units

        self.output1\_labels = n\_labels1

        self.output2\_labels = n\_labels2

        # Define network layers

        ## First fully-connected layer

        self.fc1 = nn.Linear(in\_features = self.input\_dim, out\_features = self.n\_units) # input: input\_dim, output: hidden\_units

        ## ReLU activation function

        self.relu = nn.ReLU()

        ## Output fully-connected layers

        self.fc\_output1 = nn.Linear(in\_features = self.n\_units, out\_features = self.output1\_labels) # input: hidden\_units, output: n\_labels1

        self.fc\_output2 = nn.Linear(in\_features = self.n\_units, out\_features = self.output2\_labels) # input: hidden\_units, output: n\_labels2

        # Forward pass method

    def forward(self, X):

        X = self.fc1(X)

        X = self.relu(X)

        logits\_1 = self.fc\_output1(X)

        logits\_2 = self.fc\_output2(X)

        return logits\_1, logits\_2

# Define model parameters

input\_dim = X\_train.shape[1]

hidden\_units = int(2/3\*input\_dim)

n\_labels1 = np.unique(y\_train['Output1'].values.ravel()).max()

n\_labels2 = np.unique(y\_train['Output2'].values.ravel()).max()

print(input\_dim, hidden\_units, n\_labels1, n\_labels2)

# Instantiate model

model = MLC(input\_dim = input\_dim, hidden\_units = hidden\_units, n\_labels1 = n\_labels1, n\_labels2 = n\_labels2)

# Number of epochs

n\_epochs = 200

# Loss functions

loss\_func1 = torch.nn.BCEWithLogitsLoss()

loss\_func2 = torch.nn.BCEWithLogitsLoss()

# Optimizer and learning rate

lr = 0.001

optimizer = torch.optim.Adam(model.parameters(), lr = lr)

# Early stopping

min\_delta = 0.001

patience = 10

print(model)

# Initialize zeroed count and vectors

early\_stopping\_count = 0

auc\_train\_scores = []

auc\_val\_scores = []

loss\_train\_scores = []

loss\_val\_scores = []

# Start time

time\_start = time.time()

print('\nTraining model...\n')

for epoch in range(n\_epochs):

    # Initialize batch actual values, predictions, and loss

    actual\_train\_output1 = []

    pred\_train\_output1 = []

    actual\_train\_output2 = []

    pred\_train\_output2 = []

    actual\_val\_output1 = []

    pred\_val\_output1 = []

    actual\_val\_output2 = []

    pred\_val\_output2 = []

    train\_loss1\_sum = 0

    train\_loss2\_sum = 0

    val\_loss1\_sum = 0

    val\_loss2\_sum = 0

    # Place model in training model

    model.train()

    # Iterate through training data batches

    for x, y in train\_dl:

        # Zero out the gradients

        optimizer.zero\_grad()

        # Forward pass

        pred\_y1, pred\_y2 = model(x)

        # Sum losses

        loss1 = loss\_func1(pred\_y1[:, 0], y[:, 0])

        loss2 = loss\_func2(pred\_y2[:, 0], y[:, 1])

        loss = loss1 + loss2

        # Backward pass

        loss.backward()

        optimizer.step()

        # Update vectors with real values and predictions

        actual\_train\_output1.extend(pd.DataFrame(y.detach().numpy())[0].to\_numpy())

        pred\_train\_output1.extend(sigmoid(pred\_y1[:, 0].detach().numpy()))

        actual\_train\_output2.extend(pd.DataFrame(y.detach().numpy())[1].to\_numpy())

        pred\_train\_output2.extend(sigmoid(pred\_y2[:, 0].detach().numpy()))

        # Update individual loss for display

        train\_loss1\_sum += loss1

        train\_loss2\_sum += loss2

    # Calculate epoch training loss and AUC

    train\_loss\_final = train\_loss1\_sum + train\_loss2\_sum

    train\_auc1 = roc\_auc\_score(actual\_train\_output1, pred\_train\_output1)

    train\_auc2 = roc\_auc\_score(actual\_train\_output2, pred\_train\_output2)

    # Add final training loss and AUC to vectors

    loss\_train\_scores.append([train\_loss\_final.detach().numpy(), train\_loss1\_sum.detach().numpy(), train\_loss2\_sum.detach().numpy()])

    auc\_train\_scores.append([train\_auc1, train\_auc2])

    # Put model into evaluation mode for validation

    model.eval()

    # Iterate through validation data batches

    for x, y in val\_dl:

        # Zero out the gradients

        optimizer.zero\_grad()

        # Forward pass

        pred\_y1, pred\_y2 = model(x)

        # Sum losses

        loss1 = loss\_func1(pred\_y1[:, 0], y[:, 0])

        loss2 = loss\_func2(pred\_y2[:, 0], y[:, 1])

        loss = loss1 + loss2

        # Update vectors with real values and predictions

        actual\_val\_output1.extend(pd.DataFrame(y.detach().numpy())[0].to\_numpy())

        pred\_val\_output1.extend(sigmoid(pred\_y1[:, 0].detach().numpy()))

        actual\_val\_output2.extend(pd.DataFrame(y.detach().numpy())[1].to\_numpy())

        pred\_val\_output2.extend(sigmoid(pred\_y2[:, 0].detach().numpy()))

        # Update individual loss for display

        val\_loss1\_sum += loss1

        val\_loss2\_sum += loss2

    # Calculate epoch validation loss and AUC

    val\_loss\_final = val\_loss1\_sum + val\_loss2\_sum

    val\_auc1 = roc\_auc\_score(actual\_val\_output1, pred\_val\_output1)

    val\_auc2 = roc\_auc\_score(actual\_val\_output2, pred\_val\_output2)

    # Add final validation loss and AUC to vectors

    loss\_val\_scores.append([val\_loss\_final.detach().numpy(), val\_loss1\_sum.detach().numpy(), val\_loss2\_sum.detach().numpy()])

    auc\_val\_scores.append([val\_auc1, val\_auc2])

    # Display epoch progress

    prog\_disp\_freq = 1

    if (((epoch + 1) % prog\_disp\_freq) == 0) | (epoch == 0) | ((epoch + 1) == n\_epochs):

        print("EPOCH: %s | TRAIN. (Output1 AUC: %5.3f  Output2 AUC: %5.3f   Loss: %5.3f) | VAL. (Output1 AUC: %5.3f  Output2 AUC: %5.3f   Loss: %5.3f)" % \

            (epoch, train\_auc1, train\_auc2, train\_loss\_final, val\_auc1, val\_auc2, val\_loss\_final))

    # Early stopping

    if epoch == 0:

        best\_loss = loss\_val\_scores[0][0]

    else:

        if loss\_val\_scores[epoch][0] >= (best\_loss - min\_delta):

            early\_stopping\_count += 1

        else:

            best\_loss = loss\_val\_scores[epoch][0]

            early\_stopping\_count = 0

    if early\_stopping\_count == patience:

        break

time\_end = time.time()

print('\nModel training complete.\n')

print('Time to train model: %.1f s\n' % (time\_end - time\_start))

# AUC Plot

fig = plt.figure()

fig.set\_facecolor('white')

auc\_train\_scores = pd.DataFrame(auc\_train\_scores)

auc\_val\_scores = pd.DataFrame(auc\_val\_scores)

plot\_lines = []

l1, = plt.plot(auc\_train\_scores[0], color = 'blue', linewidth = 2, linestyle = '-')

l2, = plt.plot(auc\_val\_scores[0], color = 'blue', linewidth = 2, linestyle = '--')

l3, = plt.plot(auc\_train\_scores[1], color = 'red', linewidth = 2, linestyle = '-')

l4, = plt.plot(auc\_val\_scores[1], color = 'red', linewidth = 2, linestyle = '--')

plot\_lines.append([l1, l2, l3, l4])

legend1 = plt.legend(plot\_lines[0], ["Training", "Validation"], loc = "upper center", bbox\_to\_anchor = (0.75, -0.15), fancybox = True)

plt.legend([plot\_lines[0][0], plot\_lines[0][2]], ["Output1 AUC", "Output2 AUC"], loc = "upper center", bbox\_to\_anchor = (0.25, -0.15), fancybox = True)

plt.gca().add\_artist(legend1)

plt.title("AUC over epochs", fontsize = 14)

plt.xlabel("Epochs")

plt.show()

# Loss Plot

fig = plt.figure()

fig.set\_facecolor('white')

loss\_train\_scores = pd.DataFrame(loss\_train\_scores)

loss\_val\_scores = pd.DataFrame(loss\_val\_scores)

plot\_lines = []

l1, = plt.plot(loss\_train\_scores[0], color = 'blue', linewidth = 2, linestyle = '-')

l2, = plt.plot(loss\_val\_scores[0], color = 'blue', linewidth = 2, linestyle = '--')

l3, = plt.plot(loss\_train\_scores[1], color = 'red', linewidth = 2, linestyle = '-')

l4, = plt.plot(loss\_val\_scores[1], color = 'red', linewidth = 2, linestyle = '--')

plot\_lines.append([l1, l2, l3, l4])

legend1 = plt.legend(plot\_lines[0], ["Training", "Validation"], loc = "upper center", bbox\_to\_anchor = (0.75, -0.15), fancybox = True)

plt.legend([plot\_lines[0][0], plot\_lines[0][2]], ["Output1 Loss", "Output2 Loss"], loc = "upper center", bbox\_to\_anchor = (0.25, -0.15), fancybox = True)

plt.gca().add\_artist(legend1)

plt.title("Loss over epochs", fontsize = 14)

plt.xlabel("Epochs")

plt.show()

# Running testing set through model

# Initialize test set vectors

actual\_test\_output1 = []

pred\_test\_output1 = []

actual\_test\_output2 = []

pred\_test\_output2 = []

# Place model in evaluation mode

model.eval()

# Zero out the gradients

optimizer.zero\_grad()

# Iterate through test data batches

for x, y in test\_dl:

    # Forward pass

    pred\_y1, pred\_y2 = model(x)

    # Update vectors with real values and predictions

    actual\_test\_output1.extend(pd.DataFrame(y.detach().numpy())[0].to\_numpy())

    pred\_test\_output1.extend(sigmoid(pred\_y1[:, 0].detach().numpy()))

    actual\_test\_output2.extend(pd.DataFrame(y.detach().numpy())[1].to\_numpy())

    pred\_test\_output2.extend(sigmoid(pred\_y2[:, 0].detach().numpy()))

# Calculate test AUC for Receiver-Operator Characteristic Curve

test\_auc\_y1 = roc\_auc\_score(actual\_test\_output1, pred\_test\_output1)

test\_auc\_y2 = roc\_auc\_score(actual\_test\_output2, pred\_test\_output2)

[lower1, upper1] = bootstrap\_auc(np.array(actual\_test\_output1), pred\_test\_output1, 1000)

[lower2, upper2] = bootstrap\_auc(np.array(actual\_test\_output2), pred\_test\_output2, 1000)

print("Testing set 95%% AUC CI for Output 1: %5.3f [%5.3f, %5.3f]" % (test\_auc\_y1, lower1, upper1))

print("Testing set 95%% AUC CI for Output 2: %5.3f [%5.3f, %5.3f]" % (test\_auc\_y2, lower2, upper2))

# Calculate test AUC for Precision-Recall Curve

precision, recall, thresholds = precision\_recall\_curve(actual\_test\_output1, pred\_test\_output1)

auc\_pr1 = auc(recall, precision)

precision, recall, thresholds = precision\_recall\_curve(actual\_test\_output2, pred\_test\_output2)

auc\_pr2 = auc(recall, precision)

[lower1, upper1] = bootstrap\_pr(np.array(actual\_test\_output1), pred\_test\_output1, 1000)

[lower2, upper2] = bootstrap\_pr(np.array(actual\_test\_output2), pred\_test\_output2, 1000)

print("Testing set 95%% AUPRC CI for Output 1: %5.3f [%5.3f, %5.3f]" % (auc\_pr1, lower1, upper1))

print("Testing set 95%% AUPRC CI for Output 2: %5.3f [%5.3f, %5.3f]" % (auc\_pr2, lower2, upper2))

# SHAP values for outcome 1

# Define function to wrap model to transform data to numpy

f = lambda x: model(Variable(torch.from\_numpy(x)))[0].detach().numpy()

# Convert pandas dataframe to numpy

train\_vals = X\_train.to\_numpy(dtype = np.float32)

test\_vals = X\_test.to\_numpy(dtype = np.float32)

# SHAP explainer, requires conversion from tensor to numpy using function f

explainer = shap.KernelExplainer(f, train\_vals[:100])

shap\_values\_y1 = explainer.shap\_values(test\_vals)

# Getting predictor names

feature\_names = X\_train.columns

# SHAP violin plot for outcome 1

fig = plt.figure()

fig.set\_facecolor('white')

shap.summary\_plot(shap\_values\_y1[0], X\_test, plot\_type = "violin", feature\_names = feature\_names, show = False)

plt.title("SHAP values for Outcome 1")

plt.show()

# SHAP barplot for outcome 1

fig = plt.figure()

fig.set\_facecolor('white')

shap.summary\_plot(shap\_values\_y1[0], X\_test, plot\_type = "bar", feature\_names = feature\_names, show = False)

plt.title("Feature Importance for Outcome 1")

plt.show()

# SHAP values for outcome 2

# Define function to wrap model to transform data to numpy

f = lambda x: model(Variable(torch.from\_numpy(x)))[1].detach().numpy()

# Convert pandas dataframe to numpy

train\_vals = X\_train.to\_numpy(dtype = np.float32)

test\_vals = X\_test.to\_numpy(dtype = np.float32)

# SHAP explainer, requires conversion from tensor to numpy using function f

explainer = shap.KernelExplainer(f, train\_vals[:100])

shap\_values\_y2 = explainer.shap\_values(test\_vals)

# SHAP violin plot for outcome 2

fig = plt.figure()

fig.set\_facecolor('white')

shap.summary\_plot(shap\_values\_y2[0], X\_test, plot\_type = "violin", feature\_names = feature\_names, show = False)

plt.title("SHAP values for Outcome 2")

plt.show()

# SHAP barplot for outcome 2

fig = plt.figure()

fig.set\_facecolor('white')

shap.summary\_plot(shap\_values\_y2[0], X\_test, plot\_type = "bar", feature\_names = feature\_names, show = False)

plt.title("Feature Importance for Outcome 2")

plt.show()