In [11]:
```
import csv
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MaxAbsScaler
from scipy.stats import pearsonr
from matplotlib import pyplot as plt
import torch
from torch import nn
```

In [12]:
```
DATASET_PATH: str = "./School_Absenteeism_Cleaned.csv"
```

In [13]:
```
# DEFINING INDEXES INTO THE X AND Y DATASETS
X_FEMALE_PERCENTAGE = 0
X_MALE_PERCENTAGE = 1
X_ASIAN_PERCENTAGE = 2
X_BLACK_PERCENTAGE = 3
X_HISPANIC_PERCENTAGE = 4
X_MULTIRACIAL_PERCENTAGE = 5
X_NATIVE_AMERICAN_PERCENTAGE = 6
X_WHITE_PERCENTAGE = 7
#X_MISSING_RACE_PERCENTAGE = 8
X_DISABILITIES_PERCENTAGE = 8
X_ENGLISH_LANGUAGE_LEARNERS_PERCENTAGE = 9
X_POVERTY_PERCENTAGE = 10
X_ECONOMIC_PERCENTAGE = 11

Y_ATTENDANCE_PERCENTAGE = 0
Y_CHRONICALLY_ABSENT_PERCENTAGE = 1
Y_MEAN_SCALE_SCORE_E = 2
Y_MEAN_SCALE_SCORE_M = 3
```

In [14]:
```
# Extracts the dataset from the csv into a numpy.ndarray (this is just a fancy array)
# This type of array is just easy to use for pytorch and other stats libraries
def extract_dataset(dataset_path: str) -> tuple[np.ndarray, np.ndarray]:

    X_list: list[list] = []
    Y_list: list[list] = []
    with open(dataset_path) as dataaset_file:

        csvreader = csv.reader(dataaset_file)
        next(csvreader)


        for line in csvreader:
            index = line[0]
            dbn = line[1]
            year = line[2]
            mean_scale_score_e = float(line[3])
            mean_scale_score_m = float(line[4])
            total_enrolement = line[5]
            female_percentage = float(line[6])
            male_percentage = float(line[7])
            asian_percentage = float(line[8])
            black_percentage = float(line[9])
            hispanic_percentage = float(line[10])
            multiracial_percentage = float(line[11])
            native_american_percentage = float(line[12])
            white_percentage = float(line[13])
            missing_race_data_percentage = float(line[14])
            disabilities_percentage = float(line[15])
            english_language_learners = float(line[16])
            poverty_percentage = float(line[17])
            economic_need_index = float(line[18])
            num_total_days_a = line[19]
            num_days_absent_a = line[20]
            num_days_present_a = line[21]
            attendance_percentage_a = float(line[22]) / 100
            num_contributing_total_pres_day_a = line[23]
            num_chronically_absent_a = line[24]
            chronically_absent_percentage_a = float(line[25]) / 100
            tested_percentage_e = float(line[26])
            tested_percentage_m = float(line[27])

            x_row: list[float] = [
```

```
                female_percentage,
                male_percentage,
                asian_percentage,
                black_percentage,
                hispanic_percentage,
                multiracial_percentage,
                native_american_percentage,
                white_percentage,
                disabilities_percentage,
                english_language_learners,
                poverty_percentage,
                economic_need_index
            ]

            y_row: list[float] = [
                attendance_percentage_a,
                chronically_absent_percentage_a,
                mean_scale_score_e,
                mean_scale_score_m
            ]


            X_list.append(x_row)
            Y_list.append(y_row)


    return np.array(X_list), np.array(Y_list)
```

In [15]:
```
# This calculates pearson (linear) correlation coefficient for each
# of the X features against each of the y features
def pearson_correlation(X: np.ndarray, Y: np.ndarray) -> np.ndarray:
    # Assuming X and Y are your datasets
    num_cols_X = X.shape[1]
    num_cols_Y = Y.shape[1]

    # Initialize a matrix to hold the Pearson correlation coefficients
    pearson_correlation_matrix = np.zeros((num_cols_X, num_cols_Y))

    # Calculate the Pearson correlation coefficient for each column in X against
    # each column in Y
    for i in range(num_cols_X):
        for j in range(num_cols_Y):
            # Compute the Pearson correlation coefficient
            correlation, _ = pearsonr(X[:, i], Y[:, j])
            pearson_correlation_matrix[i, j] = correlation

    return pearson_correlation_matrix

if __name__ == "__main__":
    X: np.ndarray
    Y: np.ndarray

    X, Y = extract_dataset(DATASET_PATH)

    correlation_coeffs: np.ndarray = pearson_correlation(X, Y)

    # An example of how to get the correlation coefficients from the correlation coeffs matrix
    #print(f"Linear correlation between asian percentage and attendance is {correlation_coeffs[X_ASIAN_PERCENT
    print(f"Linear correlation between asian percentage and attendance is {correlation_coeffs[X_ASIAN_PERCENTA
    print(correlation_coeffs)

    #plt.scatter(X[:,X_ASIAN_PERCENTAGE], Y[:, Y_ATTENDANCE_PERCENTAGE])
    #plt.xlabel("Asian Percentage")
    #plt.ylabel("Attendance")
    #plt.show()


    X_train, X_val, Y_train, Y_val = train_test_split( # split the dataset into a train and test split
        X,
        Y,
        test_size = 0.2,
```

```
    random_state = 42
)
X_train = torch.tensor(X_train, dtype=torch.float32) # convert to tensors (similar to numpy.ndarray but fo
X_val = torch.tensor(X_val, dtype=torch.float32)
Y_train = torch.tensor(Y_train, dtype=torch.float32)
Y_val = torch.tensor(Y_val, dtype=torch.float32)


model: nn.Sequential = nn.Sequential( # Seqiential models just pass data through each of the following moc
    nn.Linear(X.shape[1], Y.shape[1]), # (fully connected layer that learns linear relationships between X
    nn.Sigmoid() # Maps output to range (0, 1) since we are predicting values in this range
)

loss_fn: nn.Module = nn.L1Loss() # This loss is just the difference between prediction and true value e.g
optimiser: torch.optim.Optimizer = torch.optim.Adam(model.parameters(), lr=0.01) # Controls how the weigh

# Number of epochs (how many times we are training on the same dataset)
epochs = 5000
for epoch in range(epochs):

    # Set model to training mode
    model.train()

    # Zero the gradients
    optimiser.zero_grad()

    # Forward pass
    Y_pred = model(X_train)

    # Compute loss between model prediction and the true value
    loss: nn.Module = loss_fn(Y_pred, Y_train)

    # Backward pass (calculates what updates need to be made to the model)
    loss.backward()

    # Update weights of the model
    optimiser.step()

    # Print loss every 100 epochs
    if epoch % 100 == 0:
        print(f'Epoch {epoch}, Train Loss: {loss.item()}')

# Evaluate the model on validation dataset
```

```
model.eval()
# torch.no_grad is just saying we don't need to track values for backpropagation (weight updating)
with torch.no_grad():
    Y_val_pred = model(X_val) # pass the X validation dataset into the model to get the predictions for th
    val_loss = loss_fn(Y_val_pred, Y_val) # calculate the loss of the validation dataset (ou want this clo

print(f'====================\nValidation Loss: {val_loss.item()}')




######## Extracting the weights from the model so we can interpret them #############
linear_layer = model[0]
weights = linear_layer.weight.data
bias = linear_layer.bias.data

print(f"====================\nWeights of the model \n=============\nYou can interpret this as how much t
print(f"\nAttendance Weights: \n{weights[0]}")
print(f"Chronically Absent Weights: \n{weights[1]}")
print(f"ELA Test Scores: \n{weights[2]}")
print(f"Math Test Scores: \n{weights[3]}")
print(f"====================\nTo interpret this, a positive weight for an X feature means that this featu
print(f"====================")
```

```
Linear correlation between asian percentage and attendance is 0.4693007420084773
[[ 0.03000492 -0.02874846  0.01736541  0.01361121]
 [-0.0299081   0.02871256 -0.01700019 -0.01323784]
 [ 0.46930074 -0.46509995  0.0658102   0.0962609 ]
 [-0.45525613  0.45025482 -0.02634468 -0.05276537]
 [-0.2017007   0.24215728 -0.08679253 -0.09952265]
 [ 0.2443545  -0.28569156  0.01417946  0.02678091]
 [-0.0862014   0.10258708 -0.02175912 -0.0238714 ]
 [ 0.40298859 -0.45029748  0.09743478  0.11886302]
 [-0.44366626  0.43747134 -0.03538222 -0.05660722]
 [ 0.05190869 -0.00126451 -0.0965947  -0.08736029]
 [-0.47794021  0.53856617 -0.1176475  -0.138295  ]
 [-0.52488779  0.58279953 -0.13803483 -0.16007572]]
Epoch 0, Train Loss: 0.31733566522598267
Epoch 100, Train Loss: 0.1572992503643036
Epoch 200, Train Loss: 0.1522122323513031
Epoch 300, Train Loss: 0.15040172636508942
Epoch 400, Train Loss: 0.1494378298521042
Epoch 500, Train Loss: 0.1487985998392105
Epoch 600, Train Loss: 0.14832232892513275
Epoch 700, Train Loss: 0.14793266355991364
Epoch 800, Train Loss: 0.14764003455638885
Epoch 900, Train Loss: 0.14741484820842743
Epoch 1000, Train Loss: 0.1472376137971878
Epoch 1100, Train Loss: 0.14709503948688507
Epoch 1200, Train Loss: 0.1469791829586029
Epoch 1300, Train Loss: 0.14688704907894135
Epoch 1400, Train Loss: 0.14681583642959595
Epoch 1500, Train Loss: 0.1467563658952713
Epoch 1600, Train Loss: 0.14670409262180328
Epoch 1700, Train Loss: 0.14666396379470825
Epoch 1800, Train Loss: 0.1466306746006012
Epoch 1900, Train Loss: 0.14660309255123138
Epoch 2000, Train Loss: 0.14657822251319885
Epoch 2100, Train Loss: 0.14655590057373047
Epoch 2200, Train Loss: 0.14653736352920532
Epoch 2300, Train Loss: 0.14651980996131897
Epoch 2400, Train Loss: 0.14650505781173706
Epoch 2500, Train Loss: 0.14649271965026855
Epoch 2600, Train Loss: 0.14648258686065674
Epoch 2700, Train Loss: 0.14647354185581207
Epoch 2800, Train Loss: 0.1464654803276062
Epoch 2900, Train Loss: 0.14645837247371674
```

```
Epoch 3000, Train Loss: 0.14645209908485413
Epoch 3100, Train Loss: 0.1464458703994751
Epoch 3200, Train Loss: 0.14644008874893188
Epoch 3300, Train Loss: 0.14643464982509613
Epoch 3400, Train Loss: 0.14642882347106934
Epoch 3500, Train Loss: 0.1464230865240097
Epoch 3600, Train Loss: 0.14641740918159485
Epoch 3700, Train Loss: 0.14641132950782776
Epoch 3800, Train Loss: 0.1464054137468338
Epoch 3900, Train Loss: 0.14639894664287567
Epoch 4000, Train Loss: 0.14639241993427277
Epoch 4100, Train Loss: 0.14638520777225494
Epoch 4200, Train Loss: 0.1463780403137207
Epoch 4300, Train Loss: 0.1463708132505417
Epoch 4400, Train Loss: 0.14636288583278656
Epoch 4500, Train Loss: 0.14635515213012695
Epoch 4600, Train Loss: 0.14634743332862854
Epoch 4700, Train Loss: 0.14633871614933014
Epoch 4800, Train Loss: 0.1463300585746765
Epoch 4900, Train Loss: 0.14632122218608856
=====================
Validation Loss: 0.15155984461307526
=====================
Weights of the model
==============
You can interpret this as how much they contribute to each of the Y features, the indexes of the weights cor
responds to each of the X features in order, so female percentage, male percentage, asian percentage, black
percentage, ... etc are the first index, second index, ...

Attendance Weights:
tensor([ 0.9942,  1.2001,  1.6885,  0.9230,  1.2785, -1.9402, -0.9272,  0.9352,
        -0.5909,  0.0878,  0.1204, -1.3391])
Chronically Absent Weights:
tensor([-0.8232, -1.1229, -1.4071, -0.2247, -0.8149,  5.4163,  3.7141, -0.2246,
         1.1131, -0.1324, -0.1642,  2.6137])
ELA Test Scores:
tensor([ 2.2100e-01, -2.5276e-01,  3.0046e+00,  2.3062e+00,  2.5033e+00,
        -2.3344e+00,  1.1404e+00,  2.5097e+00,  1.2465e-03, -6.4549e-01,
        -8.0624e-02, -9.9773e-01])
Math Test Scores:
tensor([-0.2423, -0.3120,  3.3460,  2.5608,  2.7978, -3.4622,  1.1597,  2.8130,
        -0.0614, -0.4977, -0.2106, -0.9525])
=====================
```

```
To interpret this, a positive weight for an X feature means that this feature contributes positively to that
Y feature. So for my training, Asian Percentage (the third value), had a positive weight for attendance whic
h means that the model learned that a large Asian percentage contributes positively to a larger attendance
(this technically isn't linear so we can't say the model learned a linear correlation)
=====================
```

In [ ]: