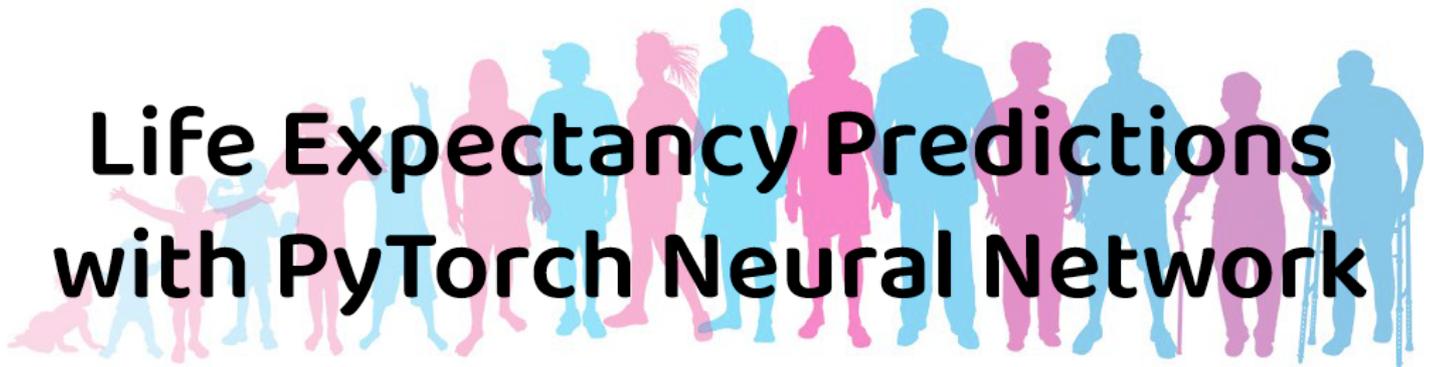


```
# Jovian Commit Essentials
# Please retain and execute this cell without modifying the contents for `jovian.commit`
!pip install jovian --upgrade -q
import jovian
jovian.set_project('neural-network-life-expectancy')
jovian.set_colab_id('1zGLFWjxddv4pI0trWiTsRTQAhHwFAW3h')
```

⇒ Life Expectancy Prediction with PyTorch



➔ INTRODUCTION:

This notebook will be a journey with PyTorch's neural network technology and a custom class linear regression model with the objective, or destination of this journey, of predicting life expectancy as accurately as possible using the quite thorough dataset from the World Health Organization with comprehensive data from 183 countries over 15 recent years.

Lets get started!

```
%%capture
!pip install jovian --upgrade --quiet
```

```
%%capture
!pip install opendatasets
```

```
%%capture
jovian.commit()
# eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsIm1hdCI6MTY2NzQxNTA1MSwianRp
```

➤ Importing Libraries

```
import torch
import jovian
import opendatasets as od
import torchvision
import torch.nn as nn
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
import random
import torch.nn.functional as F
from torchvision.datasets.utils import download_url
from torch.utils.data import DataLoader, TensorDataset, random_split
```

➤ Dataset Information:

[Dataset Source](#)

Dataset Description from Source:

CONTEXT: *Although there have been lot of studies undertaken in the past on factors affecting life expectancy considering demographic variables, income composition and mortality rates. It was found that affect of immunization and human development index was not taken into account in the past. Also, some of the past research was done considering multiple linear regression based on data set of one year for all the countries. Hence, this gives motivation to resolve both the factors stated previously by formulating a regression model based on mixed effects model and multiple linear regression while considering data from a period of 2000 to 2015 for all the countries. Important immunization like Hepatitis B, Polio and Diphtheria will also be considered. In a nutshell, this study will focus on immunization factors, mortality factors, economic factors, social factors and other health related factors as well. Since the observations this dataset are based on different countries, it will be easier for a country to determine the predicting factor which is contributing to lower value of life expectancy. This will help in suggesting a country which area should be given importance in order to efficiently improve the life expectancy of its population.*

CONTENT: *The project relies on accuracy of data. The Global Health Observatory (GHO) data repository under World Health Organization (WHO) keeps track of the health status as well as many other related factors for all countries The data-sets are made available to public for the purpose of health data analysis. The data-set related to life expectancy, health factors for 193 countries has been collected from the same WHO data repository website and its corresponding economic data was collected from United Nation website. Among all categories of health-related factors only those critical factors were chosen which are more representative. It has been observed that in the past 15 years, there has been a huge development in health sector resulting in improvement of human mortality rates especially in the developing nations in comparison to the past 30 years. Therefore, in this project we have considered data from year 2000-2015 for 193 countries for further analysis. The individual data files have been merged together into a single data-set. On initial visual inspection of the data showed some missing values. As the data-sets were from WHO, we found no evident errors. Missing data was handled in R software by using Missmap command. The result indicated that most of the missing data was for population, Hepatitis B and GDP. The missing data were from less known countries like Vanuatu, Tonga, Togo, Cabo Verde etc. Finding all data for these countries was difficult and hence, it was decided that we exclude these countries from the final model data-set. The final merged file(final dataset) consists of 22 Columns and 2938 rows which meant 20 predicting variables. All predicting variables was then divided*

into several broad categories: Immunization related factors, Mortality factors, Economical factors and Social factors.

➤ Downloading and Importing the Dataset:

```
%%capture
od.download('https://www.kaggle.com/datasets/kumarajarshi/life-expectancy-who/download?
# aa2c3d2d98029c9911c56873e95c3b94
```

```
data_raw = pd.read_csv("life-expectancy-who/Life Expectancy Data.csv")
```

➤ Basic Dataset Details:

```
print("This dataset has", data_raw.shape[0], "rows and", data_raw.shape[1], "columns, c
```

This dataset has 2938 rows and 22 columns, or features.

```
data_raw.sample(3)
```

	Country	Year	Status	Life expectancy	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	...	Polio
1319	Japan	2010	Developed	83.0	62.0	3	6.90	863.006149	NaN	450	...	98.0
2546	Syrian Arab Republic	2006	Developing	73.7	123.0	8	0.97	122.652333	83.0	517	...	83.0
2854	Vanuatu	2003	Developing	69.4	173.0	0	1.20	27.298391	64.0	165	...	67.0

3 rows × 22 columns

```
data_raw.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 2938 entries, 0 to 2937
```

```
Data columns (total 22 columns):
```

#	Column	Non-Null Count	Dtype
0	Country	2938 non-null	object
1	Year	2938 non-null	int64
2	Status	2938 non-null	object
3	Life expectancy	2928 non-null	float64
4	Adult Mortality	2928 non-null	float64
5	infant deaths	2938 non-null	int64
6	Alcohol	2744 non-null	float64
7	percentage expenditure	2938 non-null	float64

```

8 Hepatitis B          2385 non-null float64
9 Measles             2938 non-null int64
10 BMI                 2904 non-null float64
11 under-five deaths  2938 non-null int64
12 Polio               2919 non-null float64
13 Total expenditure  2712 non-null float64
14 Diphtheria         2919 non-null float64
15 HIV/AIDS           2938 non-null float64
16 GDP                 2490 non-null float64
17 Population         2286 non-null float64
18 thinness 1-19 years 2904 non-null float64
19 thinness 5-9 years  2904 non-null float64
20 Income composition of resources 2771 non-null float64
21 Schooling          2775 non-null float64

```

dtypes: float64(16), int64(4), object(2)

memory usage: 505.1+ KB

→ Cleaning Column Names:

The data came in a little messy, including column names that just won't do. So let's clean those up immediately.

```
data_raw.columns = data_raw.columns.str.strip().str.lower().str.replace(' ', '_')
```

→ data_raw.describe()

Getting an idea of the data we are working with and the distribution.

```
data_raw.describe()
```

	year	life_expectancy	adult_mortality	infant_deaths	alcohol	percentage_expenditure	hepatitis_
count	2938.000000	2928.000000	2928.000000	2938.000000	2744.000000	2938.000000	2385.000000
mean	2007.518720	69.224932	164.796448	30.303948	4.602861	738.251295	80.94046
std	4.613841	9.523867	124.292079	117.926501	4.052413	1987.914858	25.07001
min	2000.000000	36.300000	1.000000	0.000000	0.010000	0.000000	1.00000
25%	2004.000000	63.100000	74.000000	0.000000	0.877500	4.685343	77.00000
50%	2008.000000	72.100000	144.000000	3.000000	3.755000	64.912906	92.00000
75%	2012.000000	75.700000	228.000000	22.000000	7.702500	441.534144	97.00000
max	2015.000000	89.000000	723.000000	1800.000000	17.870000	19479.911610	99.00000

→ Separating the Data:

Here, I am accounting for the different types of data and separating the numerical data from the categorical. In this dataset, there is only one categorical column that will need encoding, the "status" column, developed or

developing. The "country" column will be useful in plotting and visualizing the data, but it will be a part of the inputs to the model. So we do not need to be concerned with it.

```
categorical = ['country', 'status']

numerical = ['year', 'life_expectancy', 'adult_mortality',
            'infant_deaths', 'alcohol', 'percentage_expenditure', 'hepatitis_b',
            'measles', 'bmi', 'under-five_deaths', 'polio', 'total_expenditure',
            'diphtheria', 'hiv/aids', 'gdp', 'population', 'thinness__1-19_years',
            'thinness_5-9_years', 'income_composition_of_resources', 'schooling']

# Switch total and percentage expenditure. They are wrong / swapped in the original data
data_raw['total_expenditure'], data_raw['percentage_expenditure'] = data_raw['percentage_expenditure'], data_raw['total_expenditure']
```

→ Removing NULL Values:

Because there are quite a few NULL values, I will be replacing them with the mean. This will help our model handle the data better and be more accurate.

```
data_raw.isnull().sum()
```

country	0
year	0
status	0
life_expectancy	10
adult_mortality	10
infant_deaths	0
alcohol	194
percentage_expenditure	0
hepatitis_b	553
measles	0
bmi	34
under-five_deaths	0
polio	19
total_expenditure	0
diphtheria	19
hiv/aids	0
gdp	448
population	652
thinness__1-19_years	34
thinness_5-9_years	34
income_composition_of_resources	167
schooling	163
dtype:	int64

```
# Replacing missing values with mean
for col in numerical:
    data_raw[col].fillna(data_raw[col].mean(), inplace=True)
```

→ One-Hot Encoding

The status column is the only categorical column in the dataset that will be a part of the inputs, so I will encode it and have two columns instead, one for "developed" countries and the other for "developing".

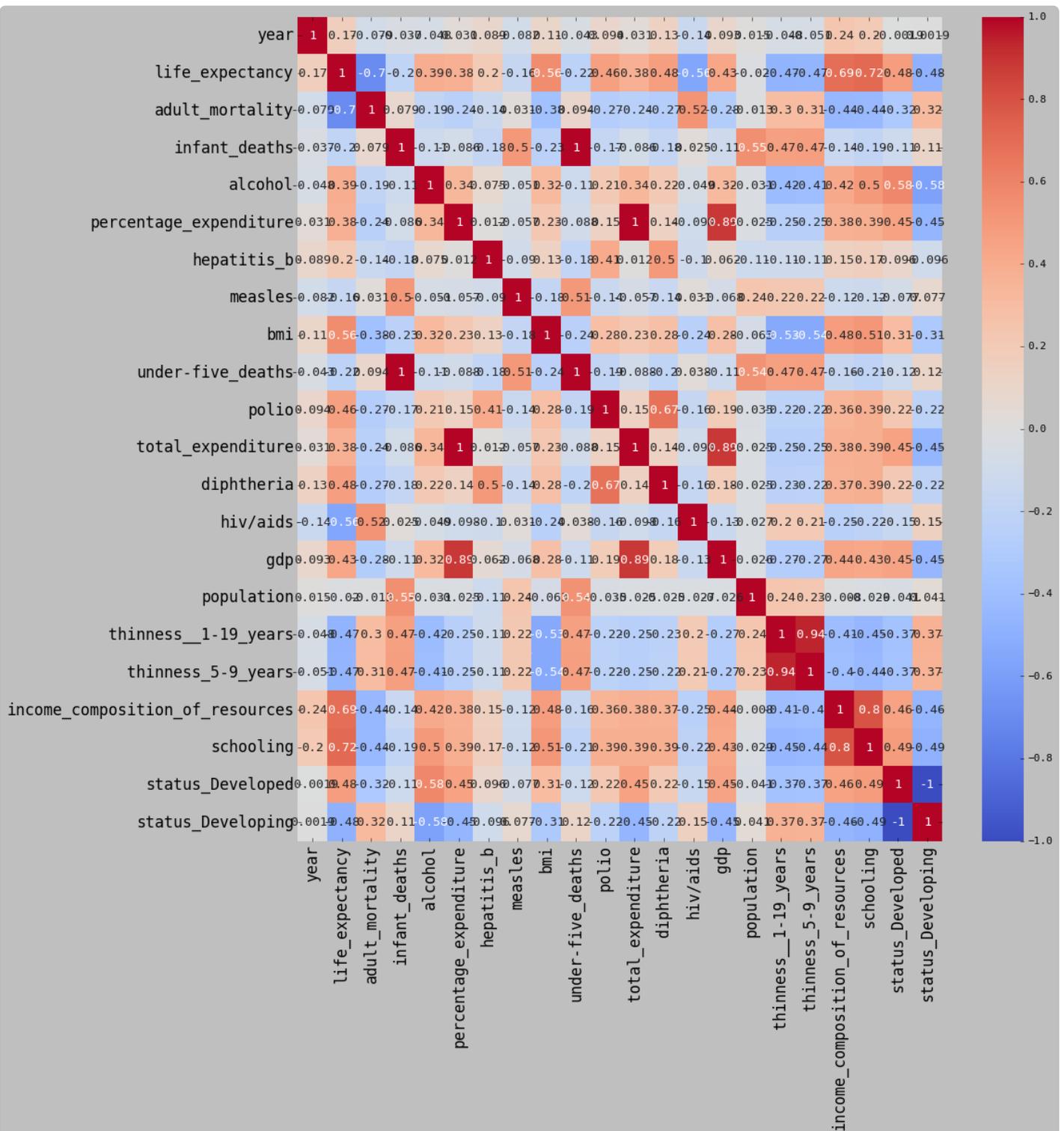
```
data_raw = pd.get_dummies(data_raw, columns=['status'])
```

➤ Correlation:

Here is our first look at how variables correlate on the raw data. Our target vector to the model will be the life expectancy column. Here we can see some interesting patterns. There are quite a few features that seem to directly correlate with the life expectancy to varying degrees. We will see those more specifically below.

Just past the heatmap, there are individual plots of each of the most impactful features and how they correlate to life expectancy, some more clearly than others.

```
# Make a heatmap of the correlation  
plt.figure(figsize=(14, 14))  
plt.xticks(fontsize = 15)  
plt.yticks(fontsize = 15, rotation=30)  
sns.heatmap(data_raw.corr(), annot=True, cmap='coolwarm');
```

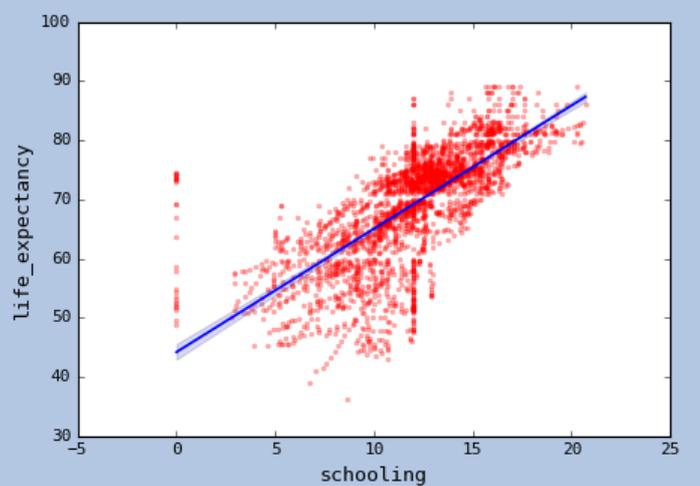
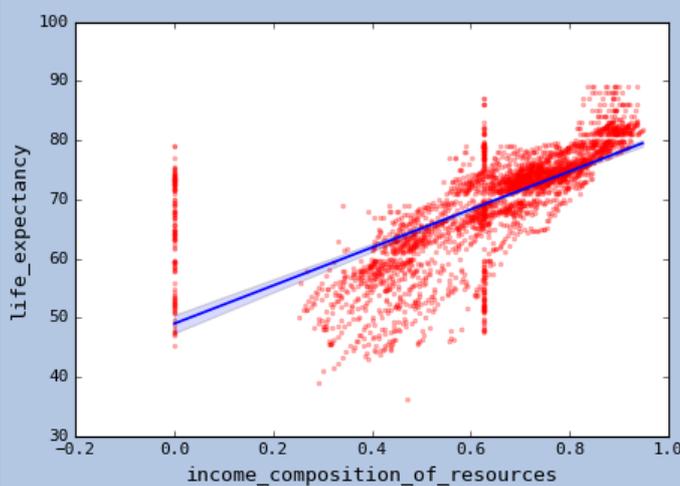
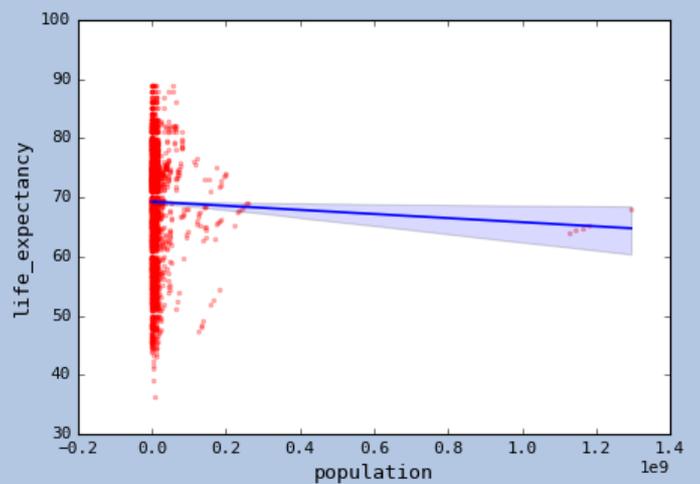
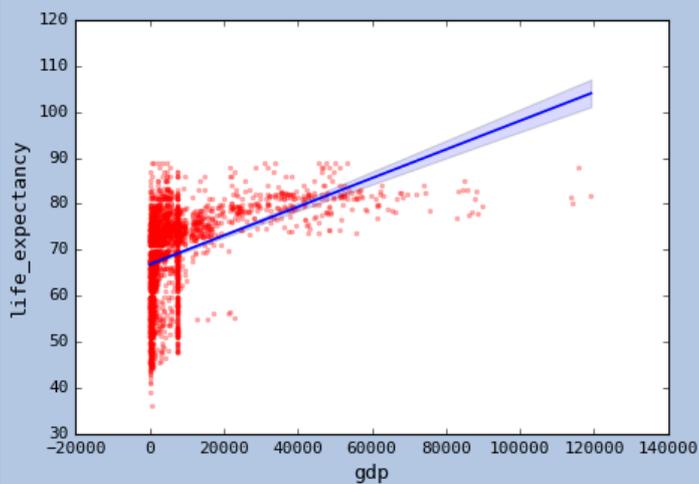
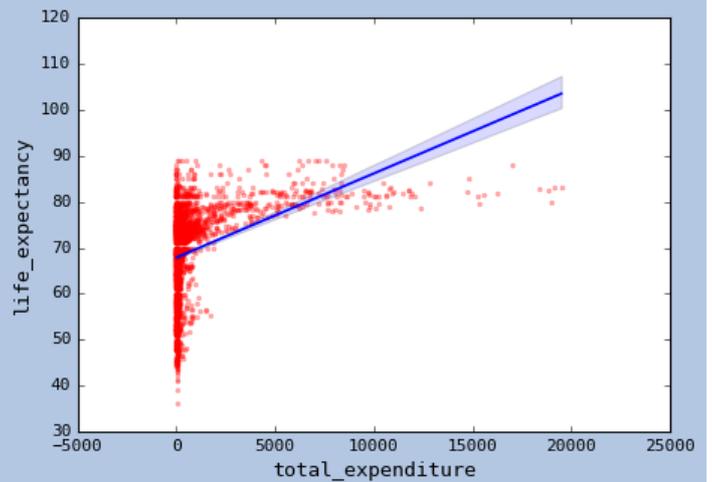
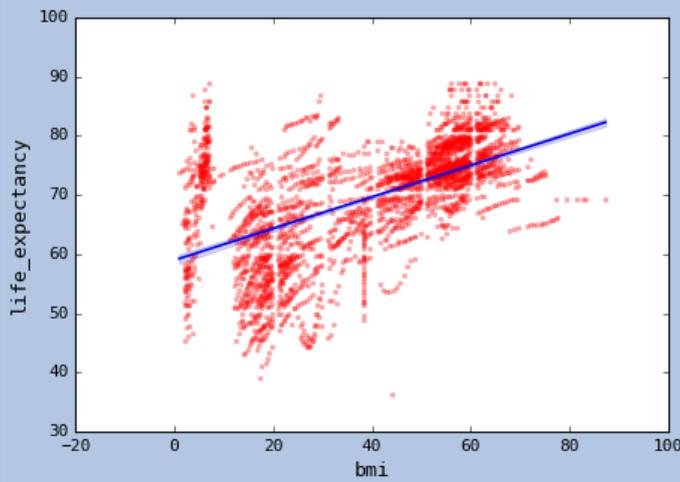
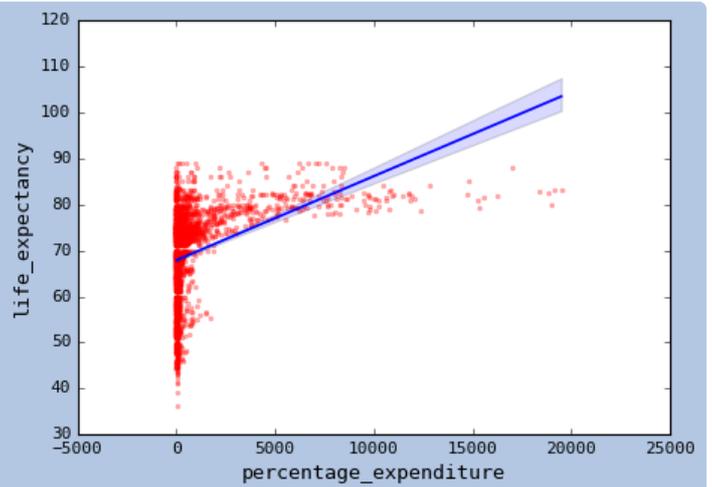
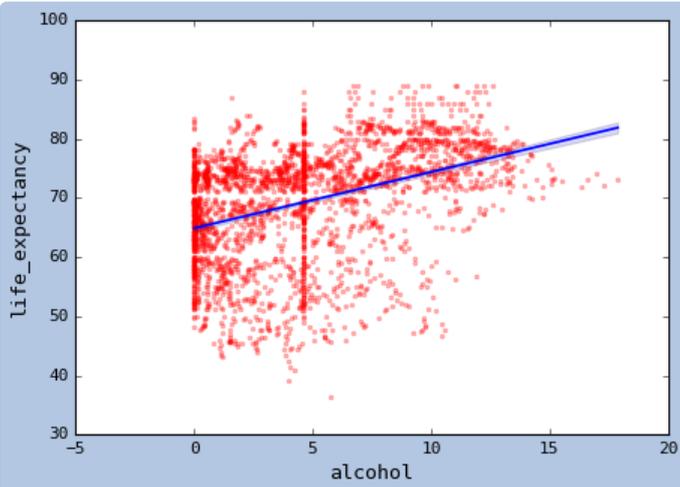


```

#@title → Features and Their Correlation to Life Expectancy: { display-mode: "form"
fig, ax = plt.subplots(4, 2, figsize=(14, 20), facecolor='#b3c7e3');
plt.style.use('classic');
plt.rcParams.update({'text.color': "black",
                    'axes.labelcolor': "black",
                    'axes.labelsize' : 12,
                    'xtick.labelsize' : 10,
                    'ytick.labelsize' : 10,
                    'font.family': 'monospace'});
sns.regplot(x=data_raw['alcohol'], y=data_raw['life_expectancy'], ax=ax[0, 0], scatter_kws=
sns.regplot(x=data_raw['percentage_expenditure'], y=data_raw['life_expectancy'], ax=ax[0, 1], scatter_kws=
sns.regplot(x=data_raw['bmi'], y=data_raw['life_expectancy'], ax=ax[1, 0], scatter_kws=
sns.regplot(x=data_raw['total_expenditure'], y=data_raw['life_expectancy'], ax=ax[1, 1], scatter_kws=

```

```
sns.regplot(x=data_raw['gdp'], y=data_raw['life_expectancy'], ax=ax[2, 0], scatter_kws=
sns.regplot(x=data_raw['population'], y=data_raw['life_expectancy'], ax=ax[2, 1], scatt
sns.regplot(x=data_raw['income_composition_of_resources'], y=data_raw['life_expectancy']
sns.regplot(x=data_raw['schooling'], y=data_raw['life_expectancy'], ax=ax[3, 1], scatte
```



```

#@title → Creating the input and target dataframes: { display-mode: "form" }
input_cols = ['adult_mortality', 'infant_deaths', 'alcohol',
              'percentage_expenditure', 'hepatitis_b',
              'measles', 'bmi', 'under-five_deaths', 'polio',
              'total_expenditure', 'diphtheria', 'hiv/aids', 'gdp',
              'population', 'thinness_1-19_years', 'thinness_5-9_years',
              'income_composition_of_resources', 'schooling',
              'status_Developed', 'status_Developing']

target_cols = ['life_expectancy']

# ----- Evaluation Data -----
# Create evaluation data for the end of the notebook
eval_data = data_raw.copy().sample(138, random_state=42)
# Remove the rows chosen for evaluation from raw data
data_raw = data_raw[~data_raw.index.isin(eval_data.index)]

# Get a list of the index numbers from eval_data
# eval_data_index = eval_data.index.tolist()

# Add a column called "original_index" to eval_data with the original index numbers for
#eval_data['original_index'] = eval_data_index

# Reset index values on eval_data
eval_data = eval_data.reset_index(drop=True)

# ----- Inputs & Targets -----
# Creating the input and target dataframes
inputs = data_raw[input_cols]
targets = data_raw[target_cols]

print("Input and target data information:")
print("- inputs: ", inputs.shape[0], 'rows and', inputs.shape[1], 'columns.')
print("- targets shape: ", targets.shape[0], 'rows and', targets.shape[1], 'columns.')
print('')
print("Evaluation data information:")
print("- eval_data: ", eval_data.shape[0], 'rows and', eval_data.shape[1], 'columns.')

```

Input and target data information:

- inputs: 2800 rows and 20 columns.
- targets shape: 2800 rows and 1 columns.

Evaluation data information:

- eval_data: 138 rows and 23 columns.

```

#@title → Converting columns to arrays, then tensors: { display-mode: "form" }
# Scale all the input data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
inputs_scaled = scaler.fit_transform(inputs)

```

```
inputs_tensor = torch.tensor(inputs_scaled, dtype=torch.float32)
targets_tensor = torch.tensor(targets.values, dtype=torch.float32)

print("Input and target tensor information:")
print("- inputs_tensor: ", inputs_tensor.shape)
print("- targets_tensor: ", targets_tensor.shape)

dataset = TensorDataset(inputs_tensor, targets_tensor)
```

Input and target tensor information:

```
- inputs_tensor: torch.Size([2800, 20])
- targets_tensor: torch.Size([2800, 1])
```

→ Dividing datasets into training and validation sets:

```
train_size = round(len(dataset) * 0.8)
validation_size = len(dataset) - train_size
train_data, validation_data = random_split(dataset, [train_size, validation_size])
```

→ Establishing batches for training:

```
batch_size = 70
train_loader = DataLoader(train_data, batch_size, shuffle=True)
validation_loader = DataLoader(validation_data, batch_size*2)
```

➤ Auditioning the best loss functions for this data:

→ Preliminary Model

```
# Input and output size definitions for Model
input_size = len(input_cols)
output_size = len(target_cols)
```

```
prelim_model = nn.Linear(input_size, output_size)
```

→ Loss Function Auditioning Function:

I wrote the following function so that I can try out a variety of loss functions available with PyTorch to see which is best for this dataset and model. I have narrowed them all down to the following two, with which I will use a list of learning rates to train these hyperparameters. Spoiler Alert! The **nn.L1Loss** definitely wins, although the **nn.MSELoss** was not bad.

```
##@title 🎧 Loss Function Audition: { display-mode: "form" }
# TRYING ALL LOSS FUNCTIONS AND LEARNING RATES
def get_lost_now(num_epochs, model, loss_fn, opt, train_dl):
    epoch_log = []
    for epoch in range(num_epochs):
```

```

    for xb, yb in train_dl:
        pred = model(xb)
        loss = loss_fn(pred, yb)
        loss.backward()
        opt.step()
        opt.zero_grad()
        if (epoch + 1) % 20 == 0:
            epoch_log.append([epoch + 1, loss.item()])
best_result = epoch_log.index(min(epoch_log, key=lambda x: x[1]))
best_epoch = epoch_log[best_result][0]
lowest_loss = epoch_log[best_result][1]
return lowest_loss, best_epoch

def loss_function_check(model, function, learning_rate_list,
                       train_loader):
    model = model
    function_log = []
    loss_fn = function
    for rate in learning_rate_list:
        optimizer = torch.optim.SGD(model.parameters(), lr=rate)
        current_lowest_loss, loss_epoch_number = get_loss_now(500, model,
                                                             loss_fn, optimizer, train_loader)
        print(f'Loss function: {function}, learning rate: {rate}, lowest loss: '
              f'{current_lowest_loss:,.4f}, epoch: {loss_epoch_number}')
        function_log.append([function, rate, current_lowest_loss, loss_epoch_number])
    function_log_df = pd.DataFrame(function_log, columns=['loss_function',
                                                       'learning_rate', 'lowest_loss', '
    epoch'])

    return function_log_df

```

→ nn.L1Loss() - Audition for the role of loss function

```

lrs_L1Loss = [0.0001, 0.001, 0.01, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.11, 0.12, 0.13]

loss_function_log_L1Loss = loss_function_check(model=prelim_model, function = nn.L1Loss,
                                              learning_rate_list=lrs_L1Loss,

```

```

Loss function: L1Loss(), learning rate: 0.0001, lowest loss: 63.8100, epoch: 360
Loss function: L1Loss(), learning rate: 0.001, lowest loss: 47.3107, epoch: 480
Loss function: L1Loss(), learning rate: 0.01, lowest loss: 2.1203, epoch: 360
Loss function: L1Loss(), learning rate: 0.05, lowest loss: 2.0949, epoch: 20
Loss function: L1Loss(), learning rate: 0.06, lowest loss: 1.8743, epoch: 120
Loss function: L1Loss(), learning rate: 0.07, lowest loss: 2.0206, epoch: 260
Loss function: L1Loss(), learning rate: 0.08, lowest loss: 2.0794, epoch: 20
Loss function: L1Loss(), learning rate: 0.09, lowest loss: 2.1819, epoch: 340
Loss function: L1Loss(), learning rate: 0.1, lowest loss: 2.0700, epoch: 60
Loss function: L1Loss(), learning rate: 0.11, lowest loss: 2.0611, epoch: 260
Loss function: L1Loss(), learning rate: 0.12, lowest loss: 2.0589, epoch: 420
Loss function: L1Loss(), learning rate: 0.13, lowest loss: 1.9579, epoch: 140

```

→ nn.MSELoss() - Audition for the role of loss function

```
lrs_MSELoss = [0.0001, 0.001, 0.002, 0.0025, 0.0026, 0.0027]
```

```
loss_function_log_MSELoss = loss_function_check(model=prelim_model, function = nn.MSELoss,
                                                learning_rate_list=lrs_MSELoss,
```

Loss function: MSELoss(), learning rate: 0.0001, lowest loss: 6.5231, epoch: 240

Loss function: MSELoss(), learning rate: 0.001, lowest loss: 7.1441, epoch: 280

Loss function: MSELoss(), learning rate: 0.002, lowest loss: 7.9606, epoch: 400

Loss function: MSELoss(), learning rate: 0.0025, lowest loss: 6.8553, epoch: 100

Loss function: MSELoss(), learning rate: 0.0026, lowest loss: 6.7872, epoch: 100

Loss function: MSELoss(), learning rate: 0.0027, lowest loss: 6.6377, epoch: 500

```
##@title > Defining Custom Class Model and Evaluation Functions: { display-mode: "form"
```

```
class LifeExpectancy_Model(nn.Module):
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.linear = nn.Linear(input_size, output_size)
```

```
    def forward(self, xb):
```

```
        out = self.linear(xb)
```

```
        return out
```

```
    def training_step(self, batch):
```

```
        inputs, targets = batch
```

```
        out = self(inputs)
```

```
        loss_function = nn.L1Loss()
```

```
        loss = loss_function(out, targets)
```

```
        return loss
```

```
    def validation_step(self, batch):
```

```
        inputs, targets = batch
```

```
        out = self(inputs)
```

```
        loss_function = nn.L1Loss()
```

```
        loss = loss_function(out, targets)
```

```
        return {'val_loss': loss.detach()}
```

```
    def validation_epoch_end(self, outputs):
```

```
        batch_losses = [x['val_loss'] for x in outputs]
```

```
        epoch_loss = torch.stack(batch_losses).mean()
```

```
        return {'val_loss': epoch_loss.item()}
```

```
    def epoch_end(self, epoch, result):
```

```
        if (epoch+1) % 50 == 0:
```

```
            print("Epoch [{}], val_loss: {:.2f}".format(epoch, result['val_loss']))
```

```
model = LifeExpectancy_Model()
```

```

# Training
def evaluate(model, val_loader):
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)

def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        # Training Phase
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation phase
        result = evaluate(model, val_loader)
        model.epoch_end(epoch, result)
        history.append(result)
        if (epoch+1) % 50 == 0:
            print('Epoch [{} / {}], Loss: {:.2f}'.format(epoch+1, epochs, loss.item()))

```

```

result = evaluate(model, validation_loader)
result

```

```
{'val_loss': 69.29302215576172}
```

```
history = fit(120, 0.06, model, train_loader, validation_loader)
```

```
Epoch [49], val_loss: 3.23
```

```
Epoch [50/120], Loss: 3.16
```

```
Epoch [99], val_loss: 3.20
```

```
Epoch [100/120], Loss: 3.08
```

➤ Evaluate Predictions:

Here, I am using single samples from the evaluation data that I separated out from the raw data at the beginning of the notebook. Let's have a look at the data we are working with. It looks like the random sample gave us a good selection of data for testing and evaluating the model.

```
eval_data['country'].value_counts()
```

```

Uganda      4
Bangladesh  3
Montenegro  3
Bahrain     3
Norway      3
..

```

```

South Africa    1
Burundi        1
Eritrea        1
France         1
Cyprus         1
Name: country, Length: 95, dtype: int64

```

The following is setting up our evaluation data so that we can see how well our model performs on real-world data that it has never seen before.

```

# Make evaluation data from the original raw data and reset the index:
eval_inputs = eval_data[input_cols]
eval_inputs = scaler.fit_transform(eval_inputs)
eval_targets = eval_data[target_cols]

print(f'-> The evaluation data has {eval_inputs.shape[0]} rows and {eval_inputs.shape[1]} columns of input data.
print(f'That is {eval_inputs.shape[0]} samples on which to evaluate the trained model.
print(f'-> The evaluation data has {eval_targets.shape[0]} rows and {eval_targets.shape[1]} columns of targets.
print("And the corresponding targets. ^^ Let's see how close the model gets.")

```

```

-> The evaluation data has 138 rows and 20 columns of input data.
That is 138 samples on which to evaluate the trained model.
-> The evaluation data has 138 rows and 1 column of targets.
And the corresponding targets. ^^ Let's see how close the model gets.

```

Here we see that we have 138 samples to work with for evaluation and testing. The function below will compare the predictions from our trained model with each sample and print out the target value, the prediction, the difference between the two, and the percentage accuracy of the model. At the end of the run, it will average the accuracy and give a final percentage.

```

#@title -> Determining Accuracy: (function) { display-mode: "form" }
def determine_accuracy(sample_count, model, eval_inputs, eval_targets):
    prediction_results = []
    for iteration in range(sample_count):
        # Choose random sample from the evaluation data
        random_row = np.random.randint(0, len(eval_inputs)-1)
        # select the inputs and targets at that row
        random_input = eval_inputs[[random_row]]
        random_target = np.array(eval_targets)[[random_row]]
        input = torch.tensor(random_input, dtype=torch.float32)
        target = torch.tensor(random_target, dtype=torch.float32)

        def prediction_versus_target(model, input, target):
            prediction = float(model(input))
            # prediction = prediction.detach().numpy()
            target = float(target.detach().numpy())
            accuracy = float((prediction / target) * 100) if target > prediction else float((target / prediction) * 100)
            return prediction, target, accuracy

```

```
prediction, target, accuracy = prediction_versus_target(model, input, target)
prediction_results.append([prediction, target, accuracy])
print(f"Iteration {iteration + 1} of {sample_count} \n - target age: {target: .2f} \n - prediction-target difference: {abs(prediction-target): .2f} \n - accuracy: {accuracy: .2f}%")
```

```
prediction_results_log = pd.DataFrame(prediction_results, columns=['prediction', 'target', 'accuracy'])

# Get the average accuracy of the model
average_accuracy = prediction_results_log['percentage_accuracy'].mean()

print(f"The average accuracy of the model is {average_accuracy: .2f}%.")

return prediction_results_log, average_accuracy
```

```
predictions, average_accuracy = determine_accuracy(sample_count=138, model=model, eval_data=eval_data)
```

Iteration 1 of 138

- target age: 64
- predicted age: 64.83
- prediction-target difference: 0.43
- accuracy: 99.34%

Iteration 2 of 138

- target age: 64
- predicted age: 64.64
- prediction-target difference: 0.84
- accuracy: 98.70%

Iteration 3 of 138

- target age: 52
- predicted age: 55.15
- prediction-target difference: 2.65
- accuracy: 95.20%

Iteration 4 of 138

- target age: 74
- predicted age: 68.77
- prediction-target difference: 4.93
- accuracy: 93.31%

Iteration 5 of 138

- target age: 73
- predicted age: 71.24
- prediction-target difference: 1.96
- accuracy: 97.32%

Iteration 6 of 138

- target age: 62
- predicted age: 63.38
- prediction-target difference: 0.88
- accuracy: 98.62%

Iteration 7 of 138

- target age: 73
- predicted age: 72.54
- prediction-target difference: 0.36
- accuracy: 99.50%

Iteration 8 of 138

- target age: 77
- predicted age: 75.69
- prediction-target difference: 1.51
- accuracy: 98.04%

Iteration 9 of 138

- target age: 59
- predicted age: 59.80
- prediction-target difference: 0.50
- accuracy: 99.16%

Iteration 10 of 138

- target age: 68
- predicted age: 62.20
- prediction-target difference: 6.10
- accuracy: 91.06%

Iteration 11 of 138

- target age: 50
- predicted age: 52.43
- prediction-target difference: 2.83
- accuracy: 94.60%

Iteration 12 of 138

- target age: 62
- predicted age: 59.15
- prediction-target difference: 2.85
- accuracy: 95.41%

Iteration 13 of 138

- target age: 64
- predicted age: 69.47
- prediction-target difference: 5.77
- accuracy: 91.69%

Iteration 14 of 138

- target age: 86
- predicted age: 83.52
- prediction-target difference: 2.48
- accuracy: 97.12%

Iteration 15 of 138

- target age: 64
- predicted age: 65.06
- prediction-target difference: 1.26
- accuracy: 98.07%

Iteration 16 of 138

- target age: 73
- predicted age: 73.79
- prediction-target difference: 0.69
- accuracy: 99.06%

Iteration 17 of 138

- target age: 70
- predicted age: 71.75
- prediction-target difference: 1.95
- accuracy: 97.29%

Iteration 18 of 138

- target age: 55
- predicted age: 51.38
- prediction-target difference: 3.92
- accuracy: 92.90%

Iteration 19 of 138

- target age: 58
- predicted age: 57.66
- prediction-target difference: 0.84
- accuracy: 98.56%

Iteration 20 of 138

- target age: 64
- predicted age: 65.06
- prediction-target difference: 1.26
- accuracy: 98.07%

Iteration 21 of 138

- target age: 80
- predicted age: 79.32
- prediction-target difference: 0.68
- accuracy: 99.15%

Iteration 22 of 138

- target age: 79
- predicted age: 77.35
- prediction-target difference: 1.95
- accuracy: 97.55%

Iteration 23 of 138

- target age: 64
- predicted age: 64.97
- prediction-target difference: 0.57
- accuracy: 99.12%

Iteration 24 of 138

- target age: 74
- predicted age: 74.70
- prediction-target difference: 1.10
- accuracy: 98.52%

Iteration 25 of 138

- target age: 75
- predicted age: 76.14
- prediction-target difference: 1.04
- accuracy: 98.64%

Iteration 26 of 138

- target age: 74
- predicted age: 77.08
- prediction-target difference: 2.68
- accuracy: 96.53%

Iteration 27 of 138

- target age: 73

- predicted age: 77.73
- prediction-target difference: 4.73
- accuracy: 93.91%

Iteration 28 of 138

- target age: 64
- predicted age: 64.64
- prediction-target difference: 0.84
- accuracy: 98.70%

Iteration 29 of 138

- target age: 83
- predicted age: 81.59
- prediction-target difference: 1.41
- accuracy: 98.30%

Iteration 30 of 138

- target age: 58
- predicted age: 56.36
- prediction-target difference: 1.54
- accuracy: 97.34%

Iteration 31 of 138

- target age: 74
- predicted age: 74.70
- prediction-target difference: 1.10
- accuracy: 98.52%

Iteration 32 of 138

- target age: 51
- predicted age: 53.28
- prediction-target difference: 1.88
- accuracy: 96.47%

Iteration 33 of 138

- target age: 80
- predicted age: 79.32
- prediction-target difference: 0.68
- accuracy: 99.15%

Iteration 34 of 138

- target age: 62
- predicted age: 59.15

- prediction-target difference: 2.85
- accuracy: 95.41%

Iteration 35 of 138

- target age: 69
- predicted age: 69.25
- prediction-target difference: 0.45
- accuracy: 99.35%

Iteration 36 of 138

- target age: 79
- predicted age: 81.50
- prediction-target difference: 2.30
- accuracy: 97.17%

Iteration 37 of 138

- target age: 59
- predicted age: 68.85
- prediction-target difference: 9.85
- accuracy: 85.69%

Iteration 38 of 138

- target age: 78
- predicted age: 76.63
- prediction-target difference: 1.37
- accuracy: 98.24%

Iteration 39 of 138

- target age: 59
- predicted age: 60.44
- prediction-target difference: 1.84
- accuracy: 96.96%

Iteration 40 of 138

- target age: 64
- predicted age: 64.48
- prediction-target difference: 0.08
- accuracy: 99.88%

Iteration 41 of 138

- target age: 61
- predicted age: 62.70
- prediction-target difference: 1.40

- accuracy: 97.76%

Iteration 42 of 138

- target age: 64

- predicted age: 69.47

- prediction-target difference: 5.77

- accuracy: 91.69%

Iteration 43 of 138

- target age: 62

- predicted age: 63.38

- prediction-target difference: 0.88

- accuracy: 98.62%

Iteration 44 of 138

- target age: 77

- predicted age: 80.42

- prediction-target difference: 3.72

- accuracy: 95.37%

Iteration 45 of 138

- target age: 79

- predicted age: 81.55

- prediction-target difference: 2.75

- accuracy: 96.63%

Iteration 46 of 138

- target age: 75

- predicted age: 76.44

- prediction-target difference: 1.34

- accuracy: 98.24%

Iteration 47 of 138

- target age: 73

- predicted age: 77.73

- prediction-target difference: 4.73

- accuracy: 93.91%

Iteration 48 of 138

- target age: 77

- predicted age: 75.69

- prediction-target difference: 1.51

- accuracy: 98.04%

Iteration 49 of 138

- target age: 79
- predicted age: 80.97
- prediction-target difference: 1.87
- accuracy: 97.69%

Iteration 50 of 138

- target age: 71
- predicted age: 68.73
- prediction-target difference: 2.27
- accuracy: 96.81%

Iteration 51 of 138

- target age: 73
- predicted age: 59.79
- prediction-target difference: 13.61
- accuracy: 81.46%

Iteration 52 of 138

- target age: 72
- predicted age: 70.93
- prediction-target difference: 1.17
- accuracy: 98.37%

Iteration 53 of 138

- target age: 76
- predicted age: 71.79
- prediction-target difference: 4.11
- accuracy: 94.58%

Iteration 54 of 138

- target age: 64
- predicted age: 64.83
- prediction-target difference: 0.43
- accuracy: 99.34%

Iteration 55 of 138

- target age: 67
- predicted age: 68.98
- prediction-target difference: 1.78
- accuracy: 97.42%

Iteration 56 of 138

- target age: 70
- predicted age: 71.75
- prediction-target difference: 1.95
- accuracy: 97.29%

Iteration 57 of 138

- target age: 74
- predicted age: 72.13
- prediction-target difference: 1.77
- accuracy: 97.61%

Iteration 58 of 138

- target age: 71
- predicted age: 68.73
- prediction-target difference: 2.27
- accuracy: 96.81%

Iteration 59 of 138

- target age: 58
- predicted age: 56.36
- prediction-target difference: 1.54
- accuracy: 97.34%

Iteration 60 of 138

- target age: 82
- predicted age: 79.95
- prediction-target difference: 1.55
- accuracy: 98.10%

Iteration 61 of 138

- target age: 78
- predicted age: 76.63
- prediction-target difference: 1.37
- accuracy: 98.24%

Iteration 62 of 138

- target age: 77
- predicted age: 79.62
- prediction-target difference: 2.82
- accuracy: 96.46%

Iteration 63 of 138

- target age: 79
- predicted age: 81.55
- prediction-target difference: 2.75
- accuracy: 96.63%

Iteration 64 of 138

- target age: 68
- predicted age: 62.20
- prediction-target difference: 6.10
- accuracy: 91.06%

Iteration 65 of 138

- target age: 67
- predicted age: 67.46
- prediction-target difference: 0.46
- accuracy: 99.32%

Iteration 66 of 138

- target age: 67
- predicted age: 68.98
- prediction-target difference: 1.78
- accuracy: 97.42%

Iteration 67 of 138

- target age: 66
- predicted age: 70.91
- prediction-target difference: 4.91
- accuracy: 93.07%

Iteration 68 of 138

- target age: 79
- predicted age: 77.35
- prediction-target difference: 1.95
- accuracy: 97.55%

Iteration 69 of 138

- target age: 59
- predicted age: 68.85
- prediction-target difference: 9.85
- accuracy: 85.69%

Iteration 70 of 138

- target age: 62

- predicted age: 62.66
- prediction-target difference: 0.56
- accuracy: 99.11%

Iteration 71 of 138

- target age: 77
- predicted age: 79.62
- prediction-target difference: 2.82
- accuracy: 96.46%

Iteration 72 of 138

- target age: 75
- predicted age: 71.75
- prediction-target difference: 3.25
- accuracy: 95.67%

Iteration 73 of 138

- target age: 50
- predicted age: 52.43
- prediction-target difference: 2.83
- accuracy: 94.60%

Iteration 74 of 138

- target age: 74
- predicted age: 76.58
- prediction-target difference: 2.38
- accuracy: 96.89%

Iteration 75 of 138

- target age: 74
- predicted age: 74.48
- prediction-target difference: 0.02
- accuracy: 99.97%

Iteration 76 of 138

- target age: 79
- predicted age: 77.35
- prediction-target difference: 1.95
- accuracy: 97.55%

Iteration 77 of 138

- target age: 73
- predicted age: 77.73

- prediction-target difference: 4.73
- accuracy: 93.91%

Iteration 78 of 138

- target age: 81
- predicted age: 82.88
- prediction-target difference: 1.88
- accuracy: 97.73%

Iteration 79 of 138

- target age: 64
- predicted age: 67.36
- prediction-target difference: 3.56
- accuracy: 94.71%

Iteration 80 of 138

- target age: 53
- predicted age: 63.27
- prediction-target difference: 9.97
- accuracy: 84.24%

Iteration 81 of 138

- target age: 69
- predicted age: 71.20
- prediction-target difference: 2.10
- accuracy: 97.05%

Iteration 82 of 138

- target age: 74
- predicted age: 72.13
- prediction-target difference: 1.77
- accuracy: 97.61%

Iteration 83 of 138

- target age: 74
- predicted age: 77.08
- prediction-target difference: 2.68
- accuracy: 96.53%

Iteration 84 of 138

- target age: 64
- predicted age: 64.97
- prediction-target difference: 0.57

- accuracy: 99.12%

Iteration 85 of 138

- target age: 72

- predicted age: 73.42

- prediction-target difference: 1.82

- accuracy: 97.52%

Iteration 86 of 138

- target age: 68

- predicted age: 70.32

- prediction-target difference: 2.72

- accuracy: 96.14%

Iteration 87 of 138

- target age: 50

- predicted age: 60.39

- prediction-target difference: 10.89

- accuracy: 81.97%

Iteration 88 of 138

- target age: 59

- predicted age: 59.80

- prediction-target difference: 0.50

- accuracy: 99.16%

Iteration 89 of 138

- target age: 75

- predicted age: 75.10

- prediction-target difference: 0.40

- accuracy: 99.47%

Iteration 90 of 138

- target age: 51

- predicted age: 60.25

- prediction-target difference: 9.25

- accuracy: 84.65%

Iteration 91 of 138

- target age: 74

- predicted age: 74.70

- prediction-target difference: 1.10

- accuracy: 98.52%

Iteration 92 of 138

- target age: 75
- predicted age: 75.10
- prediction-target difference: 0.40
- accuracy: 99.47%

Iteration 93 of 138

- target age: 76
- predicted age: 77.53
- prediction-target difference: 1.63
- accuracy: 97.90%

Iteration 94 of 138

- target age: 61
- predicted age: 62.70
- prediction-target difference: 1.40
- accuracy: 97.76%

Iteration 95 of 138

- target age: 56
- predicted age: 61.34
- prediction-target difference: 5.34
- accuracy: 91.29%

Iteration 96 of 138

- target age: 74
- predicted age: 68.77
- prediction-target difference: 4.93
- accuracy: 93.31%

Iteration 97 of 138

- target age: 73
- predicted age: 59.79
- prediction-target difference: 13.61
- accuracy: 81.46%

Iteration 98 of 138

- target age: 58
- predicted age: 57.66
- prediction-target difference: 0.84
- accuracy: 98.56%

Iteration 99 of 138

- target age: 76
- predicted age: 76.95
- prediction-target difference: 1.35
- accuracy: 98.25%

Iteration 100 of 138

- target age: 71
- predicted age: 68.73
- prediction-target difference: 2.27
- accuracy: 96.81%

Iteration 101 of 138

- target age: 74
- predicted age: 74.70
- prediction-target difference: 1.10
- accuracy: 98.52%

Iteration 102 of 138

- target age: 77
- predicted age: 80.42
- prediction-target difference: 3.72
- accuracy: 95.37%

Iteration 103 of 138

- target age: 71
- predicted age: 72.47
- prediction-target difference: 1.07
- accuracy: 98.52%

Iteration 104 of 138

- target age: 74
- predicted age: 71.01
- prediction-target difference: 3.49
- accuracy: 95.31%

Iteration 105 of 138

- target age: 78
- predicted age: 78.84
- prediction-target difference: 1.04
- accuracy: 98.68%

Iteration 106 of 138

- target age: 74
- predicted age: 76.58
- prediction-target difference: 2.38
- accuracy: 96.89%

Iteration 107 of 138

- target age: 79
- predicted age: 81.55
- prediction-target difference: 2.75
- accuracy: 96.63%

Iteration 108 of 138

- target age: 59
- predicted age: 64.14
- prediction-target difference: 5.24
- accuracy: 91.83%

Iteration 109 of 138

- target age: 46
- predicted age: 59.75
- prediction-target difference: 13.45
- accuracy: 77.49%

Iteration 110 of 138

- target age: 66
- predicted age: 70.91
- prediction-target difference: 4.91
- accuracy: 93.07%

Iteration 111 of 138

- target age: 64
- predicted age: 64.48
- prediction-target difference: 0.08
- accuracy: 99.88%

Iteration 112 of 138

- target age: 67
- predicted age: 69.68
- prediction-target difference: 2.48
- accuracy: 96.44%

Iteration 113 of 138

- target age: 64

- predicted age: 64.48
- prediction-target difference: 0.08
- accuracy: 99.88%

Iteration 114 of 138

- target age: 79
- predicted age: 81.50
- prediction-target difference: 2.30
- accuracy: 97.17%

Iteration 115 of 138

- target age: 51
- predicted age: 60.25
- prediction-target difference: 9.25
- accuracy: 84.65%

Iteration 116 of 138

- target age: 80
- predicted age: 79.31
- prediction-target difference: 0.69
- accuracy: 99.14%

Iteration 117 of 138

- target age: 79
- predicted age: 80.97
- prediction-target difference: 1.87
- accuracy: 97.69%

Iteration 118 of 138

- target age: 62
- predicted age: 62.66
- prediction-target difference: 0.56
- accuracy: 99.11%

Iteration 119 of 138

- target age: 76
- predicted age: 71.79
- prediction-target difference: 4.11
- accuracy: 94.58%

Iteration 120 of 138

- target age: 76
- predicted age: 77.53

- prediction-target difference: 1.63
- accuracy: 97.90%

Iteration 121 of 138

- target age: 59
- predicted age: 59.80
- prediction-target difference: 0.50
- accuracy: 99.16%

Iteration 122 of 138

- target age: 63
- predicted age: 69.28
- prediction-target difference: 5.98
- accuracy: 91.36%

Iteration 123 of 138

- target age: 71
- predicted age: 71.01
- prediction-target difference: 0.01
- accuracy: 99.98%

Iteration 124 of 138

- target age: 68
- predicted age: 62.20
- prediction-target difference: 6.10
- accuracy: 91.06%

Iteration 125 of 138

- target age: 59
- predicted age: 64.14
- prediction-target difference: 5.24
- accuracy: 91.83%

Iteration 126 of 138

- target age: 69
- predicted age: 71.54
- prediction-target difference: 2.14
- accuracy: 97.01%

Iteration 127 of 138

- target age: 72
- predicted age: 79.04
- prediction-target difference: 7.44

- accuracy: 90.58%

Iteration 128 of 138

- target age: 68

- predicted age: 70.32

- prediction-target difference: 2.72

- accuracy: 96.14%

Iteration 129 of 138

- target age: 58

- predicted age: 56.95

- prediction-target difference: 0.65

- accuracy: 98.87%

Iteration 130 of 138

- target age: 73

- predicted age: 73.79

- prediction-target difference: 0.69

- accuracy: 99.06%

Iteration 131 of 138

- target age: 77

- predicted age: 75.69

- prediction-target difference: 1.51

- accuracy: 98.04%

Iteration 132 of 138

- target age: 59

- predicted age: 60.44

- prediction-target difference: 1.84

- accuracy: 96.96%

Iteration 133 of 138

- target age: 72

- predicted age: 69.67

- prediction-target difference: 2.33

- accuracy: 96.76%

Iteration 134 of 138

- target age: 82

- predicted age: 81.65

- prediction-target difference: 0.15

- accuracy: 99.82%

Iteration 135 of 138

- target age: 53
- predicted age: 57.38
- prediction-target difference: 4.28
- accuracy: 92.53%

Iteration 136 of 138

- target age: 52
- predicted age: 47.98
- prediction-target difference: 3.92
- accuracy: 92.45%

Iteration 137 of 138

- target age: 72
- predicted age: 61.36
- prediction-target difference: 10.24
- accuracy: 85.70%

Iteration 138 of 138

- target age: 52
- predicted age: 47.98
- prediction-target difference: 3.92
- accuracy: 92.45%

The average accuracy of the model is 95.86%.

```
print(f'The average accuracy of the model is {average_accuracy: .2f}%')
```

The average accuracy of the model is 95.86%.

➤ Accuracy of 95.86%

So we can see here that the model did very well on predicting life expectancy using the various features in the dataset. We will now look at the samples that tested below the 95.86% average and see what it is about those samples that threw off our model. Perhaps it will shed some light on what might lead to more accuracy in the future.

```
predictions
```

	prediction	target	percentage_accuracy
0	64.828979	64.400002	99.338293
1	64.642342	63.799999	98.696919
2	55.145569	52.500000	95.202572

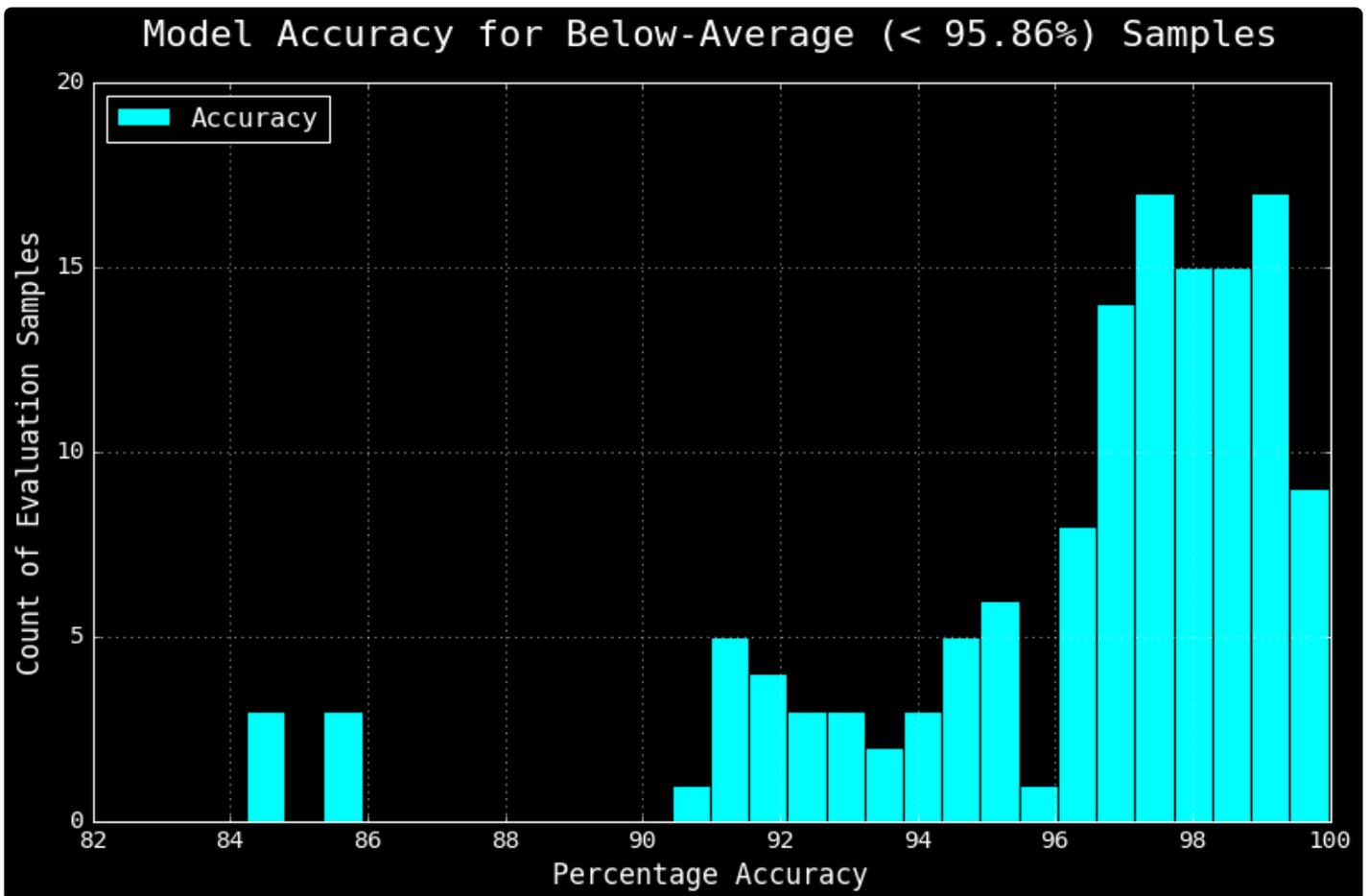
	prediction	target	percentage_accuracy
3	68.766945	73.699997	93.306578
4	71.237015	73.199997	97.318330
...
133	81.651512	81.800003	99.818471
134	57.384995	53.099998	92.532898
135	47.981987	51.900002	92.450839
136	61.361977	71.599998	85.701086
137	47.981987	51.900002	92.450839

138 rows × 3 columns

```

#@title → Model Accuracy for Below-Average (< 95.86%) Samples { display-mode: "form" }
fig = plt.figure(figsize=(12, 7))
plt.style.use('dark_background');
plt.rcParams.update({'text.color': "white",
                    'axes.labelcolor': "white",
                    'axes.labelsize' : 15,
                    'xtick.labelsize' : 13,
                    'ytick.labelsize' : 13,
                    'font.family': 'monospace'});
plt.hist(predictions['percentage_accuracy'], bins=40, label='Accuracy', color = "cyan")
plt.title('Model Accuracy for Below-Average (< 95.86%) Samples', size = 20, pad = 20)
plt.xlabel('Percentage Accuracy')
plt.ylabel('Count of Evaluation Samples')
plt.grid(color='white', linestyle=':')
plt.ylim(0, 20)
plt.xlim(82, 100)
plt.legend(loc='upper left')
plt.show()

```



➤ Review evaluation samples that score below average accuracy:

Not only did we get a really nice accuracy score of 95.86%. Even the test samples that came in below average have a pretty high level of accuracy, as you can see from the plot above. The vast majority are still in the high 90 percents!

```
def predictions_below_average(predictions, average_accuracy):
    below_average_predictions = []
    for index in predictions.index:
        if predictions['percentage_accuracy'][index] < average_accuracy:
            below_average_predictions.append(eval_data.iloc[index])
    below_average_predictions = pd.DataFrame(below_average_predictions)
    below_average_predictions = below_average_predictions.join(predictions)
    below_average_predictions = below_average_predictions[['percentage_accuracy', 'country',
                                                           'gdp', 'schooling', 'income_per_capita']]
    below_average_predictions = below_average_predictions.sort_values(by='percentage_accuracy')
    return below_average_predictions
```

```
below_average = predictions_below_average(predictions, average_accuracy)
```

→ 10 Examples from the Below Average Samples:

Let's look at the samples that came in below the accuracy average.

```
below_average.sample(10)
```

	percentage_accuracy	country	hiv/aids	population	bmi	gdp	schooling	income_compositi
137	92.450839	Cyprus	0.1	9.935630e+05	54.3	2293.478900	13.000000	
79	84.239519	Suriname	0.9	4.936300e+04	5.4	36.487730	11.100000	
107	91.827119	Bangladesh	0.1	1.413749e+07	12.0	46.757917	8.100000	
76	93.913364	Yemen	0.1	1.275338e+07	37.2	7483.158469	8.500000	
118	94.584628	Democratic Republic of the Congo	1.7	1.275338e+07	18.6	7483.158469	11.992793	
109	93.072781	Kenya	2.8	4.723626e+07	22.0	1349.971440	11.100000	
136	85.701086	Bahrain	0.1	1.275338e+07	57.6	2977.115300	14.400000	
71	95.670614	Chile	0.1	1.597378e+07	56.8	621.828325	14.300000	
135	92.450839	Turkmenistan	0.1	5.565284e+06	48.6	6432.668768	10.800000	
94	91.288003	Antigua and Barbuda	0.2	1.275338e+07	45.7	12565.441970	13.800000	

→ Overview of the Below Average Samples:

As you can see below, the average accuracy score for the 43 samples that came in below our average is still 90%. It looks like we have a pretty good model here!

```
below_average.describe()
```

	percentage_accuracy	hiv/aids	population	bmi	gdp	schooling	income_composition_of
count	43.000000	43.000000	4.300000e+01	43.000000	43.000000	43.000000	
mean	90.978760	1.579070	3.482307e+07	31.307471	5732.124192	10.987869	
std	4.668489	2.767202	1.734689e+08	20.811561	8190.669651	3.677475	
min	77.488213	0.100000	4.879500e+04	2.700000	34.516918	3.000000	
25%	90.823363	0.100000	8.063495e+05	14.050000	325.504643	8.250000	
50%	92.450839	0.400000	6.859482e+06	24.400000	1349.971440	11.200000	
75%	94.584628	1.950000	1.275338e+07	55.350000	7483.158469	13.350000	
max	95.670614	15.700000	1.144119e+09	63.100000	34165.934300	20.600000	

→ Below Average Samples by Accuracy Ascending:

```
below_average.sort_values(by='percentage_accuracy').head(20)
```

	percentage_accuracy	country	hiv/aids	population	bmi	gdp	schooling	income_comp
108	77.488213	Haiti	4.6	8.976552e+06	36.500000	329.782946	8.100000	
50	81.456362	Somalia	0.9	1.275338e+07	18.600000	7483.158469	11.992793	
96	81.456362	Slovakia	0.1	1.275338e+07	59.100000	7483.158469	15.000000	

	percentage_accuracy	country	hiv/aids	population	bmi	gdp	schooling	income_comp
86	81.967462	Saudi Arabia	0.1	1.275338e+07	59.400000	11138.874600	12.300000	
79	84.239519	Suriname	0.9	4.936300e+04	5.400000	36.487730	11.100000	
114	84.645148	India	0.3	1.144119e+09	13.500000	77.819000	9.400000	
89	84.645148	Greece	0.1	1.987314e+06	6.500000	22551.735740	15.600000	
68	85.688193	Eritrea	1.3	4.153332e+06	14.700000	317.329434	5.300000	
36	85.688193	Uganda	3.2	3.883334e+07	18.100000	719.172669	10.000000	
136	85.701086	Bahrain	0.1	1.275338e+07	57.600000	2977.115300	14.400000	
126	90.583037	Iran (Islamic Republic of)	0.1	1.275338e+07	46.900000	7483.158469	11.800000	
123	91.063689	Ghana	3.5	1.992452e+07	2.700000	39.484473	7.600000	
63	91.063689	Cabo Verde	0.6	4.879500e+04	24.900000	234.289920	11.900000	
9	91.063689	Swaziland	15.7	1.225258e+06	3.200000	3934.273250	11.200000	
94	91.288003	Antigua and Barbuda	0.2	1.275338e+07	45.700000	12565.441970	13.800000	
121	91.361891	Suriname	0.4	5.479280e+05	57.700000	9564.463830	12.700000	
41	91.687701	Togo	1.7	6.859482e+06	22.600000	563.689425	12.000000	
12	91.687701	Guinea	2.0	1.135170e+05	21.200000	459.291200	8.400000	
124	91.827119	Sudan	0.1	2.725535e+06	38.321247	361.358430	5.500000	
107	91.827119	Bangladesh	0.1	1.413749e+07	12.000000	46.757917	8.100000	

The graphs plotted below show us the most notable features and how they correspond to the accuracy of the model in predicting the life expectancy in samples that scored less than our average accuracy of 95.31%. As you can see from the excerpt from our below average samples above, most of the countries that the model had a slightly tougher time predicting are developing countries with heavy outliers.

```

#@title → Plotting Below Average Accuracy Samples: { display-mode: "form" }
def plot_below_average(dataframe, column_list):
    fig, axes = plt.subplots(8, 1, figsize=(13, 20))
    plt.style.use('dark_background');#Solarize_Light2
    #fig.suptitle(f'Model Accuracy in Samples <{average_accuracy: .2f}% / Average Accur
    for index, ax in enumerate(axes.flat):
        for column in column_list:
            ax.plot(dataframe[column_list[index]], dataframe['percentage_accuracy'], 'c
            ax.set_title(column_list[index].replace('_', ' '))
            ax.grid("both")
            # ax.set_xlabel(column_list[index])
            ax.set_ylabel('accuracy')
    plt.tight_layout(pad=0.5)
    plt.show()

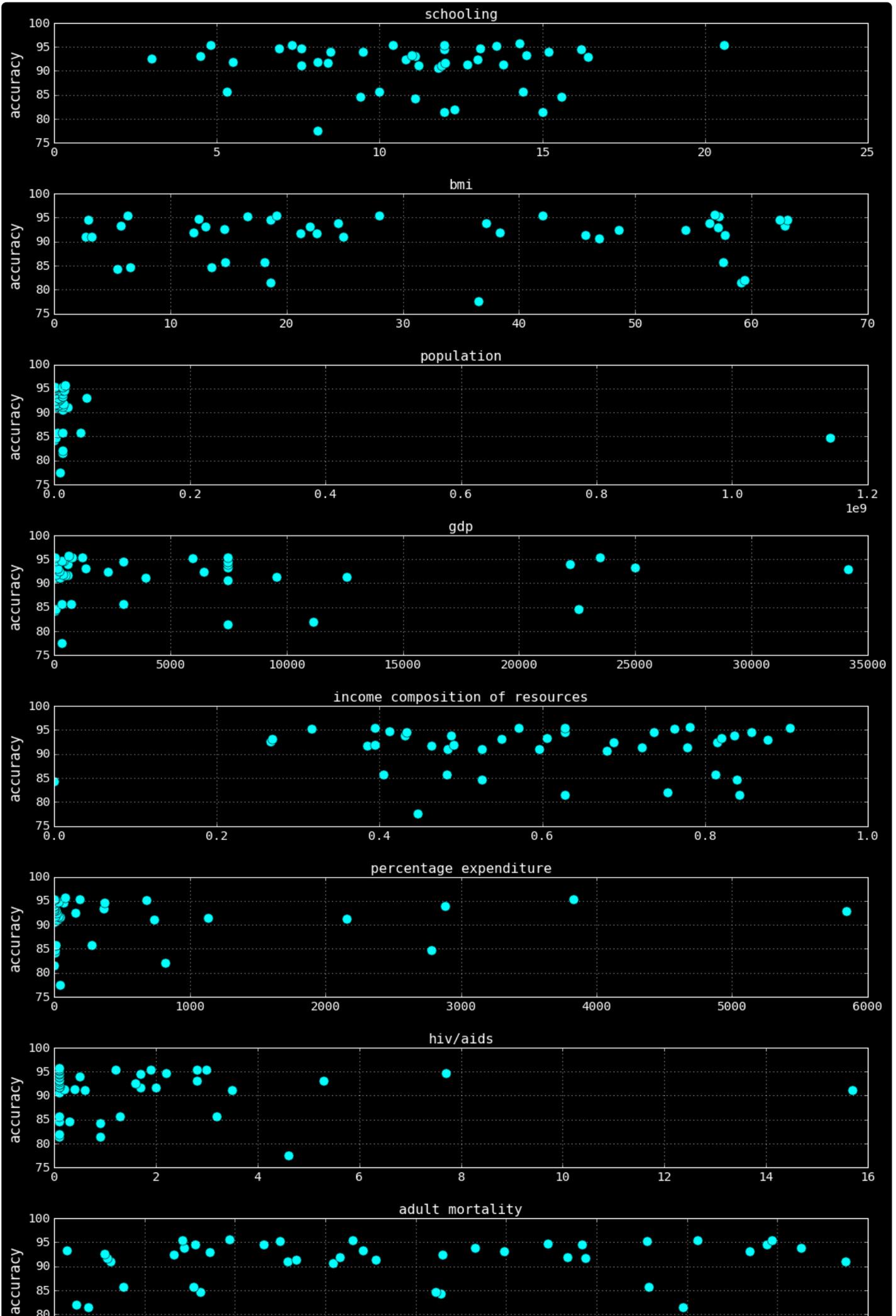
```

```

column_list = ['schooling', 'bmi', 'population', 'gdp', 'income_composition_of_resource
               'percentage_expenditure', 'hiv/aids', 'adult_mortality']

```

```
plot_below_average(below_average, column_list)
```





➤ Conclusion:

I honestly feel like I have only scratched the surface of what can be taken away from this dataset, but it was a very good start and a very successful implementation of a PyTorch linear regression neural network. I hope you enjoyed this journey with me!