Global Life Expectancy Regression with TensorFlow

The World Health Organization (WHO)'s Global Health Observatory (GHO) data repository tracks life expectancy for countries worldwide by following health status and many other related factors.

Although there have been a lot of studies undertaken in the past on factors affecting life expectancy considering demographic variables, income composition, and mortality rates, it was found that the effects of immunization and human development index were not taken into account.

This dataset covers a variety of indicators for all countries from 2000 to 2015 including:

- immunization factors
- mortality factors
- economic factors
- social factors
- other health-related factors

However, all indicators will be uncovered in the next section.

Ideally, this data will eventually inform countries concerning which factors to change in order to improve the life expectancy of their populations. If we can predict life expectancy well given all the factors, this is a good sign that there are some important patterns in the data. Life expectancy is expressed in years, and hence it is a number. This means that in order to build a predictive model one needs to use regression.

In this project, the main focus is to design, train, and evaluate a neural network model performing the task of regression to predict the life expectancy of countries using the dataset.

1. Exploratory Data Analysis

This dataset is sourced from Life Expectancy Kaggle Dataset (https://www.kaggle.com/kumarajarshi/life-expectancy-who), submitted by KumarRajarshi (https://www.kaggle.com/kumarajarshi), originally scraped from WHO and United Nations website with the help of Deeksha Russell and Duan Wang.

First, I load the `life_expectancy.csv` dataset into a pandas DataFrame by first importing pandas, and then using the `pandas.read_csv()` function to load the file and assign the resulting DataFrame to a variable called dataset.

In [45]:
```
import pandas as pd
import numpy as np

# Set options for printing
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)

# Load the dataset
df = pd.read_csv("life_expectancy.csv")
```

The next step is to observe the data by printing the first entries in the DataFrame dataset by using the `df.head()` function.

In [46]:
```
df.head()
```

Out[46]:

	Country	Year	Status	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	BM
0	Afghanistan	2015	Developing	263.0	62	0.01	71.279624	65.0	1154	19.
1	Afghanistan	2014	Developing	271.0	64	0.01	73.523582	62.0	492	18.
2	Afghanistan	2013	Developing	268.0	66	0.01	73.219243	64.0	430	18.
3	Afghanistan	2012	Developing	272.0	69	0.01	78.184215	67.0	2787	17.
4	Afghanistan	2011	Developing	275.0	71	0.01	7.097109	68.0	3013	17.

In [56]:
```
# Sanity check whether the dataset has any missing values
print(df.isnull().values.any())
```

```
False
```

Column Definition

From here, we know what these columns represent:

- `Country` represents the name of the country
- `Year` represents the year of observation in that country.
- `Status` is categorical, either `Developing` or `Developed`
- `Adult Mortality` is a probability of dying between 15 and 60 years for both sexes (per 1000 population).
- `infant deaths` is the number of Infant Deaths per 1000 population
- `Alcohol` is alcohol consumption, recorded per capita (age 15+) (in litres of pure alcohol)
- `Percentage Expenditure` is the expenditure on health as a percentage of Gross Domestic Product per capita (%)
- `Hepatitis B` is Hepatitis B (HepB) immunization coverage among 1-year-olds (%)
- `Measles -` number of reported measles cases (per 1000 population)
- `BMI` is the average Body Mass Index of entire population, of that country and year
- `under-five-deaths` is the number of under-five deaths (per 1000 population)
- `Polio` is the rate of Polio (Pol3) immunization coverage among 1-year-olds (%)
- `Total expenditure` is the general government expenditure on health as a percentage of total government expenditure (%)
- `Diphtheria` the rate of Diphtheria tetanus toxoid and pertussis (DTP3) immunization coverage among 1-year-olds (%)
- `HIV-AIDS` is the amount of deaths because of HIV/AIDS (0-4 years) (per 1000 live births)
- `GDP` Gross Domestic Product per capita (in USD)
- `Population` is the population of the country
- `thinness 1-19 years` is the prevalence of thinness among children and adolescents for Age 10 to 19 (%)
- `thinness 5-9 years` is the prevalence of thinness among children for Age 5 to 9 (%)
- `income composition of resources` is the Human Development Index in terms of income composition of resources (index ranging from 0 to 1)
- `Schooling` is the number of years of Schooling (years)
- `Life expectancy` is the life expectancy in age

2. Data Pre-Processing

Dropping the `Country` and `Year` column from the DataFrame using the DataFrame `drop` method. Why? To create a predictive model, knowing from which country data comes can be confusing and it is not a column we can generalize over. The goal is to learn a general pattern for all the countries, and not only those dependent on specific countries and/or year.

In [3]:
```
df.drop(['Country', 'Year'], axis=1, inplace=True)
df
```

Out[3]:

	Status	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	BMI	under-five deaths	Polic
0	Developing	263.0	62	0.01	71.279624	65.0	1154	19.1	83	6.(
1	Developing	271.0	64	0.01	73.523582	62.0	492	18.6	86	58.(
2	Developing	268.0	66	0.01	73.219243	64.0	430	18.1	89	62.(
3	Developing	272.0	69	0.01	78.184215	67.0	2787	17.6	93	67.(
4	Developing	275.0	71	0.01	7.097109	68.0	3013	17.2	97	68.(
...	...	...	...	...	...	...	...	...	...	..
2933	Developing	723.0	27	4.36	0.000000	68.0	31	27.1	42	67.(
2934	Developing	715.0	26	4.06	0.000000	7.0	998	26.7	41	7.(
2935	Developing	73.0	25	4.43	0.000000	73.0	304	26.3	40	73.(
2936	Developing	686.0	25	1.72	0.000000	76.0	529	25.9	39	76.(
2937	Developing	665.0	24	1.68	0.000000	79.0	1483	25.5	39	78.(

2938 rows × 20 columns

Label and Feature Selection

After dropping the aforementioned columns, I'll be splitting the data into labels and features. Labels are contained in the `Life expectancy` column, which is the final column in the DataFrame. As such, one method is to use `iloc` indexing to assign the final column of dataset to it.

In [4]:
```
labels = df.iloc[:, -1]
labels
```

Out[4]:
```
0       65.0
1       59.9
2       59.9
3       59.5
4       59.2
        ...
2933    44.3
2934    44.5
2935    44.8
2936    45.3
2937    46.0
Name: Life expectancy, Length: 2938, dtype: float64
```

Features span from the first column up until the last column (not including it, because `Life Expectancy` is considered a label). Like before, features are assigned using the `iloc` indexing to assign a subset of columns.

In [5]:
```
features = df.iloc[:, np.r_[0:19]]
features
```

Out[5]:

	Status	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	BMI	under-five deaths	Polic
0	Developing	263.0	62	0.01	71.279624	65.0	1154	19.1	83	6.(
1	Developing	271.0	64	0.01	73.523582	62.0	492	18.6	86	58.(
2	Developing	268.0	66	0.01	73.219243	64.0	430	18.1	89	62.(
3	Developing	272.0	69	0.01	78.184215	67.0	2787	17.6	93	67.(
4	Developing	275.0	71	0.01	7.097109	68.0	3013	17.2	97	68.(
...	...	...	...	...	...	...	...	...	...	..
2933	Developing	723.0	27	4.36	0.000000	68.0	31	27.1	42	67.(
2934	Developing	715.0	26	4.06	0.000000	7.0	998	26.7	41	7.(
2935	Developing	73.0	25	4.43	0.000000	73.0	304	26.3	40	73.(
2936	Developing	686.0	25	1.72	0.000000	76.0	529	25.9	39	76.(
2937	Developing	665.0	24	1.68	0.000000	79.0	1483	25.5	39	78.(

2938 rows × 19 columns

One-Hot Encoding

One particular column is categorical in this dataset, namely the `Status` column. Categorical columns need to be converted into numerical columns using methods such as one-hot-encoding. To tackle this approach, using `pandas.get_dummies(DataFrame)` to apply one-hot-encoding on the categorical column is a prominent method. After that, I'll assign the result of the encoding back to the `features` variable.

In [6]:
```
features = pd.get_dummies(data = features, columns= ['Status'])
features # Check whether one-hot encoding works as intended
```

Out[6]:

	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	BMI	under-five deaths	Polio	Tot expenditu
0	263.0	62	0.01	71.279624	65.0	1154	19.1	83	6.0	8.
1	271.0	64	0.01	73.523582	62.0	492	18.6	86	58.0	8.
2	268.0	66	0.01	73.219243	64.0	430	18.1	89	62.0	8.
3	272.0	69	0.01	78.184215	67.0	2787	17.6	93	67.0	8.
4	275.0	71	0.01	7.097109	68.0	3013	17.2	97	68.0	7.
...	...	...	...	...	...	...	...	...	...	
2933	723.0	27	4.36	0.000000	68.0	31	27.1	42	67.0	7.
2934	715.0	26	4.06	0.000000	7.0	998	26.7	41	7.0	6.
2935	73.0	25	4.43	0.000000	73.0	304	26.3	40	73.0	6.
2936	686.0	25	1.72	0.000000	76.0	529	25.9	39	76.0	6.
2937	665.0	24	1.68	0.000000	79.0	1483	25.5	39	78.0	7.

2938 rows × 20 columns

Training and Test Set Splitting

Now that the data has been cleaned, they are split into training set and test sets using the `sklearn.model_selection.train_test_split()` function.

Variables to assign:

- the training features to a variable called `features_train`
- training labels to a variable called `labels_train`
- test data to a variable called `features_test`
- test labels to a variable called `labels_test` .

For this project, the data is split randomly into 33% test data and 67% training data.

Note that 67% of the training data will be split again into 80% training set and 20% validation set when the model is instantiated.

In [7]:
```
from sklearn.model_selection import train_test_split

features_train, features_test, labels_train, labels_test = train_test_split(features, labels, test_size=0.33, random_state=42)

features_train
```

Out[7]:

	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	BMI	under-five deaths	Polio	To expenditu
2285	177.0	0	9.72	1121.475547	99.0	0	34.2	0	99.0	3.
601	262.0	2	0.25	77.028396	69.0	85	19.6	2	69.0	4.
360	159.0	65	7.19	394.932130	99.0	0	49.4	73	99.0	8.
1616	129.0	0	1.98	41.306888	98.0	0	15.6	0	97.0	5.
303	312.0	1	0.17	93.358728	98.0	418	13.9	1	98.0	6.
...	...	...	...	...	...	...	...	...	...	
1638	59.0	0	6.91	3601.287457	82.0	3	68.0	0	96.0	9.
1095	287.0	4	3.21	53.307827	83.0	26	23.1	7	82.0	6.
1130	266.0	17	6.08	56.778587	92.0	0	4.7	23	62.0	5.
1294	72.0	2	9.30	3519.258515	95.0	10982	57.0	3	97.0	8.
860	34.0	7	1.07	5.064689	96.0	19	13.9	9	96.0	2.

1968 rows × 20 columns

Z-Score Normalization

The next step is to normalize the data's numerical features.

As a wrapper, normalization with z-score is done using the `sklearn.compose.ColumnTransformer` method. The variable `ct` will be used to set up the normalization procedure.

Keep in mind that only numerical columns are legitimate arguments for the `ColumnTransformer` object. Also, for the columns not specified when the object is created, `sklearn` will automatically ignore them when the parameter `remainder='passthrough'` is specified.

In [8]:
```
print("Feature columns are: ", features.columns.tolist())
```

```
Feature columns are:  ['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']
```

In [9]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer

ct = ColumnTransformer([
    (
        'scaler',
        StandardScaler(),
        ['Adult Mortality', 'infant deaths', 'Alcohol', 'percentage expenditur
e', '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', 'Schoolin
g', 'Status_Developed', 'Status_Developing']
    )
], remainder='passthrough')
```

After instantiating the object `ct` of `ColumnTransformer`, fit and transform training data by using the `ColumnTransformer.fit_transform()` method. The result is assigned to a variable called `features_train_scaled`.

By similar method, transform also the test data instance `features_test` using the trained `ColumnTransformer` instance `ct`. The result is assigned into a variable called `features_test_scaled`.

In [57]:
```
features_train_scaled = pd.DataFrame(ct.fit_transform(features_train), columns
= features_train.columns)
features_test_scaled = pd.DataFrame(ct.fit_transform(features_test), columns =
features_test.columns)

features_test_scaled
```

Out[57]:

	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	BMI	under-five deaths	
0	-0.362574	-0.188911	-0.928037	-0.316057	-0.001886	-0.188249	0.556401	-0.199456	0.(
1	-0.441418	-0.252366	1.879750	0.374488	0.390150	-0.236200	1.042862	-0.258584	0.5
2	-0.346806	-0.252366	0.108724	-0.048191	-3.225294	-0.236293	0.997256	-0.258584	0.4
3	-1.277159	-0.252366	-0.772910	-0.198185	0.651507	-0.232019	1.286093	-0.258584	0.(
4	2.042152	0.358392	-0.788423	-0.374833	0.390150	1.715404	-1.171551	0.456863	-3.
...	...	...	...	...	...	...	...	...	
965	0.220867	-0.220639	-0.273921	-0.374153	0.085233	-0.236293	0.338507	-0.229020	0.1
966	0.788541	-0.228570	-1.176239	-0.375161	0.651507	-0.236293	-0.614147	-0.229020	0.(
967	0.906806	-0.188911	0.307803	-0.369085	0.520829	-0.227558	0.348641	-0.199456	0.5
968	1.001418	-0.236502	1.114460	-0.270026	0.390150	-0.234899	-0.477330	-0.240845	-1.
969	-0.394112	-0.252366	0.806793	-0.295164	0.433710	-0.236293	0.439853	-0.258584	0.4

970 rows × 20 columns

3. Network Building

In this project, I'll be training and building a neural network from scratch using the `Sequential()` method from `tensorflow.keras.models`. Afterwards, if the evaluation metrics are satisfactory, I'll export the model with the computed weights in `.json` and `.pb`.

Baseline Instantiation

I start with a dummy model before performing the regression task. This is done to provide a baseline, which I'll use to evaluate whether the model I'll make performs reasonably or not.

For this project, the baseline is calculated with the `mean` method along the evaluation metric **Mean Absolute Error**.

Feel free to use `median` or `quantile` for the reproducibility of the result.

In [63]:
```
from sklearn.dummy import DummyRegressor
from sklearn.metrics import mean_absolute_error

dummy_regr = DummyRegressor(strategy="mean")
dummy_regr.fit(features_train_scaled, labels_train)
y_pred = dummy_regr.predict(features_test_scaled)
MAE_baseline = mean_absolute_error(labels_test, y_pred)
print("The model must have a validation error lower than", round(MAE_baseline,
3))
```

```
The model must have a validation error lower than 7.833
```

Making the Model

For reproducibility, I'll wrap my model around functions instead of making instances of them every time. Such functions are free to tweak for your own purposes and/or datasets, but for this project, model specifications are as follows:

- 1 output layer for each datapoint;
- 2 hidden layers, with 64 and 32 nodes respectively, and with `relu` activation;
- 2 dropout layers, with 20% dropout fraction of total outputs;
- Adam optimizer;
- *Mean Squared Error* loss function;
- *Mean Absolute Error* evaluation metrics.
- `Verbose` set to `True` ;
- `EarlyStopping` enabled, monitoring validation loss with `patience` set to `5` .

In the case of `EarlyStopping` , `patience` parameter is set to a small number despite a large amount of initial epochs to prevent overfitting.

The model designed is wrapped around the `design_model` function.

In [61]:
```
## Define functions as wrappers
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
def design_model(X, learning_rate):
    """Function to instantiate empty model

    Input: 2-dimensional NumPy array or Pandas DataFrame, and specified learni
ng rate.

    Returns: Empty model with 20% dropout between hidden layers"""
    model = Sequential(name="second_model")
    input = tf.keras.Input(shape=(X.shape[1],)) # Automatically detect feature
s as input nodes
    model.add(input)

    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dropout(0.2))
    model.add(layers.Dense(32, activation='relu'))
    model.add(layers.Dropout(0.2))

    model.add(layers.Dense(1)) # output layer
    opt = tf.keras.optimizers.Adam(learning_rate = learning_rate)
    model.compile(loss='mse', metrics=['mae'], optimizer = opt)
    return model

def plot(history, path=None):
    """Function to plot learning curves.

    Input: model metrics per epoch

    Returns: Learning curves for both training and validation sets"""
    fig, axs = plt.subplots(1, 2, gridspec_kw={'hspace': 1, 'wspace': 0.5})
    (ax1, ax2) = axs
    ax1.plot(history.history['loss'], label='train')
    ax1.plot(history.history['val_loss'], label='validation')
    ax1.legend(loc="upper right")
    ax1.set_xlabel("# of epochs")
    ax1.set_ylabel("loss (mse)")

    ax2.plot(history.history['mae'], label='train')
    ax2.plot(history.history['val_mae'], label='validation')
    ax2.legend(loc="upper right")
    ax2.set_xlabel("# of epochs")
    ax2.set_ylabel("MAE")

def fit_model(model, f_train, l_train, learning_rate, num_epochs):
    """Function to train the model with stochastic gradient descent.

    Input: 2-dimensional NumPy array or Pandas DataFrame

    Returns: Trained model with calculated weights"""
    es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=5)
    model.fit(f_train, l_train, epochs=num_epochs, batch_size= 1, verbose=1, v
```

```
alidation_split = 0.2, callbacks=[es])
    return model

def save_model(model):
    """Function to save pre-trained model.

    Input: pre-trained model"""
    return model.save('model')
```

Before starting training, I'll double check whether we get the layers right by passing the `Sequential.summary(model)` method.

In [64]:
```
learning_rate = 0.01 # Specify learning rate
num_epochs = 200 # Specify number of forward-backward passes along the network

model = design_model(features_train_scaled, learning_rate)
model.summary()
```

```
Model: "second_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 64)                1344
_________________________________________________________________
dropout_8 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_13 (Dense)             (None, 32)                2080
_________________________________________________________________
dropout_9 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_14 (Dense)             (None, 1)                 33
=================================================================
Total params: 3,457
Trainable params: 3,457
Non-trainable params: 0
_________________________________________________________________
```

In [65]:
```
fit_model(model, features_train_scaled, labels_train, learning_rate, num_epochs)
print("------ TRAINING FINISHED! ------".center(110))
```

```
Epoch 1/200
1574/1574 [==============================] - 2s 975us/step - loss: 673.9647 -
mae: 18.8983 - val_loss: 102.5363 - val_mae: 8.6541
Epoch 2/200
1574/1574 [==============================] - 1s 872us/step - loss: 191.4294 -
mae: 10.9800 - val_loss: 38.4667 - val_mae: 4.6537
Epoch 3/200
1574/1574 [==============================] - 1s 880us/step - loss: 199.7355 -
mae: 11.0939 - val_loss: 63.1065 - val_mae: 6.5446
Epoch 4/200
1574/1574 [==============================] - 1s 873us/step - loss: 174.3048 -
mae: 10.4190 - val_loss: 44.7044 - val_mae: 4.5733
Epoch 5/200
1574/1574 [==============================] - 1s 899us/step - loss: 156.6945 -
mae: 9.8157 - val_loss: 41.9230 - val_mae: 4.8199
Epoch 6/200
1574/1574 [==============================] - 1s 888us/step - loss: 122.8386 -
mae: 8.9154 - val_loss: 144.7928 - val_mae: 10.5800
Epoch 7/200
1574/1574 [==============================] - 1s 888us/step - loss: 112.9416 -
mae: 8.3536 - val_loss: 29.0418 - val_mae: 4.1900
Epoch 8/200
1574/1574 [==============================] - 1s 909us/step - loss: 109.6132 -
mae: 8.1137 - val_loss: 24.0315 - val_mae: 3.8315
Epoch 9/200
1574/1574 [==============================] - 1s 889us/step - loss: 93.0972 -
mae: 7.4063 - val_loss: 16.9380 - val_mae: 3.1740
Epoch 10/200
1574/1574 [==============================] - 1s 917us/step - loss: 75.8575 -
mae: 6.7003 - val_loss: 28.3784 - val_mae: 4.3769
Epoch 11/200
1574/1574 [==============================] - 1s 924us/step - loss: 75.3744 -
mae: 6.7350 - val_loss: 18.4234 - val_mae: 2.9702
Epoch 12/200
1574/1574 [==============================] - 1s 904us/step - loss: 74.5407 -
mae: 6.6582 - val_loss: 21.3164 - val_mae: 3.4439
Epoch 13/200
1574/1574 [==============================] - 1s 916us/step - loss: 64.3847 -
mae: 6.0667 - val_loss: 55.5443 - val_mae: 6.7846
Epoch 14/200
1574/1574 [==============================] - 1s 887us/step - loss: 54.0934 -
mae: 5.7897 - val_loss: 22.3867 - val_mae: 3.4954
Epoch 00014: early stopping
                                        ------ TRAINING FINISHED! ------
```

In [72]:
```
mse, mae = model.evaluate(features_test_scaled, labels_test)
print("----------------- EVALUATION FINISHED! -----------------".center(115
))
print("Final Mean Squared Error (loss func.) is {}\nFinal Mean Absolute Error
 (eval. metric) is {}".format(mse, mae))
```

```
31/31 [==============================] - 0s 1ms/step - loss: 21.8280 - mae:
3.5451
                              ----------------- EVALUATION FINISHED! --------
----------
Final Mean Squared Error (loss func.) is 21.828018188476562
Final Mean Absolute Error (eval. metric) is 3.5451462268829346
```

4. Conclusion

The observed baseline is **`7.833`** for the model's evaluation metric. After training, results show that this simple yet effective network has an error of **`3.54`** on test dataset.

This shows that the model actually performs more than 2x better the baseline, and thus we can draw a conclusion that the result is satisfactory.

We can use this model to predict unseen data from this dataset, and if a new data comes regardless of the country and/or year, the model can right away predict the life expectancy of that data point with minimal error.

In [75]:
```
# Save the model
save_model(model)
```

```
INFO:tensorflow:Assets written to: model\assets
```