Report on the Mini Project

Topic: Life Expectancy Prediction

Algorithms: Linear Clustering, Decision Trees, Random Forests, Boosting

Team 2

Nguyễn Tống Minh – 20204885
Nguyễn Nhật Quang – 20204925
Nguyễn Công Đạt – 20200137
Nguyễn Xuân Thái Hòa – 20204878
Ngô Thị Thu Huyền - 20200289

Course: Machine Learning – 2022.1
19/07/2022

I. Problem Description

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 datasets, including Life Expectancy (WHO), are made available to public for the purpose of health data analysis. The dataset Life Expectancy (WHO) related to life expectancy and 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.

Given the dataset where each record contains 21 explanatory variables, which can affect the life expectancy of each country on the globe, and a response attribute which tells us about the average age of the country recorded at that time. Our task is to determine the life expectancy of a country recorded at a certain period given all the 22 observed features of that country.

The dataset contains around 2000 records. There are 21 attributes in each record of the dataset. They can be found in Life Expectancy (WHO).

II. Exploratory Data Analysis

Firstly, we took a glance at the data to get a general understanding of the kind of data we were dealing with. Since our training set has just around 2000 instances, we have just worked directly on the full set. The exploratory data analysis was carried in the notebook of the same name placed at the root of the project folder.

II.1. Data Collection

One issue at first glance was that the column names of the dataset had no convention, which posed an obstacle for the data users. Therefore, we prepared a new name set for the dataset columns.

```
In [4]:
# Show columns
df.columns
```

```
Out[4]:
Index(['Country', 'Year', 'Status', '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'],
      dtype='object')
```

Figure 2.1.1. The naming does not have any convention.

```
In [5]:
# Rename some columns as their names contain trailing spaces
df.rename(columns={" BMI ":"BMI","Life expectancy ":"Life_Expectancy","Adult Mortality":"Adult_Mort
                   "infant deaths":"Infant_Deaths","percentage expenditure":"Percentage_Exp","Hepat
                  "Measles ":"Measles"," BMI ":"BMI","under-five deaths ":"Under_Five_Deaths","Diph
                  " HIV/AIDS":"HIV/AIDS"," thinness  1-19 years":"thinness_1to19_years"," thinness
                   "Total expenditure":"Tot_Exp"},inplace=True)
df.columns
```

```
Out[5]:
Index(['Country', 'Year', 'Status', 'Life_Expectancy', 'Adult_Mortality',
       'Infant_Deaths', 'Alcohol', 'Percentage_Exp', 'HepatitisB', 'Measles',
       'BMI', 'Under_Five_Deaths', 'Polio', 'Tot_Exp', 'Diphtheria',
       'HIV/AIDS', 'GDP', 'Population', 'thinness_1to19_years',
       'thinness_5to9_years', 'Income_Comp_Of_Resources', 'Schooling'],
      dtype='object')
```

Figure 2.1.2. The naming was changed. After that we also lowered the case of the column names.

Then, we went through the structure of the data, which showed that there were only two categorical columns (country, status) and the remaining columns were numerical.

```
In [7]: # Show the schema and state of the dataset
        df.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_exp           2938 non-null   float64
 8   hepatitisb               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  tot_exp                  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_1to19_years     2904 non-null   float64
 19  thinness_5to9_years      2904 non-null   float64
 20  income_comp_of_resources 2771 non-null   float64
 21  schooling                2775 non-null   float64
dtypes: float64(16), int64(4), object(2)
memory usage: 505.1+ KB
```

Caution: Two columns (`country` and `status`) are categorical.

Figure 2.1.3. The data structure.

Most machine learning algorithms prefer to work with numbers, so the two categories should be converted from text to numbers. One way is that each of them is encoded into one single, ordinal attribute.
However, the issue is that machine learning algorithms will assume that two nearby values are more similar than two distant values. This may be fine in some cases (e.g., “bad”, “good”), but it is obviously not the case for status and country.

```
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

Figure 2.1.4. One single, ordinal array

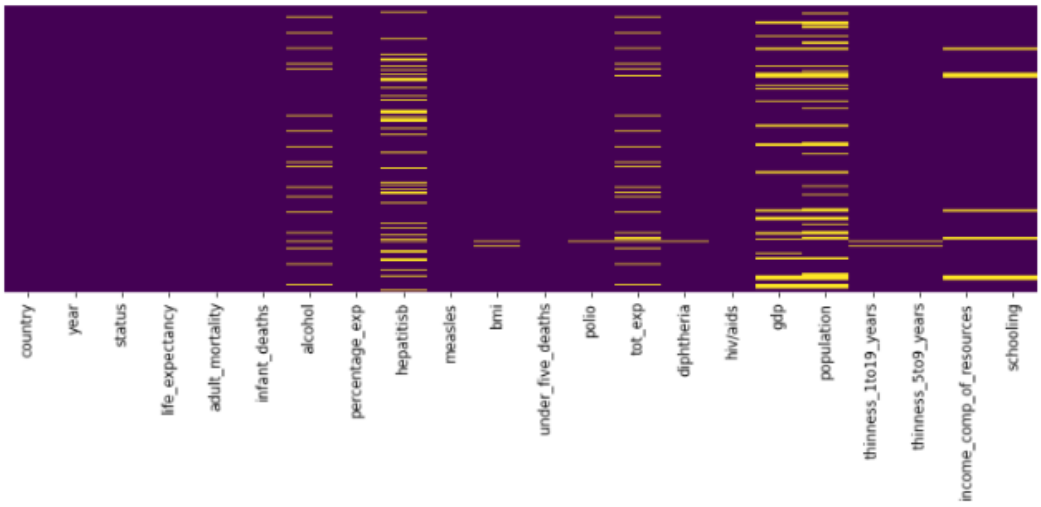
A common solution is to create one binary attribute per category: One attribute equal to 1 when the category is “Developing” (and 0 otherwise), another attribute equal to 1 when the category is “Developed” (and 0 otherwise), and so on. Therefore, dummy variables (implemented from pandas) were intended to replace the classes of each of the original categorical attributes during the preprocessing phase. However, for country, one problem left was that it has nearly 200 distinct countries, which can result in up to 200 dummy features. This could lead to several unexpected events and slow down the models’ development dramatically. Therefore, we had to convert country into a corresponding attribute with fewer classes, then encode it. This step will be represented at the end of this section.

II.2. Data Missing

One observed that the dataset has been poorly collected as some attributes (e.g., population) had lots of null values.

```
In [16]:
# Show the distribution of missing values
plt.figure(figsize=(12,6))
msno = sns.heatmap(df.isnull(), yticklabels=False, cbar=False, cmap = 'viridis')
save_fig("msno_matrix")
```

```
Saving figure msno_matrix
```



Note: The yellow spaces signify missing entries.

Figure 2.2.1. The dense presence and distribution of missing data.

Particularly, some attributes (e.g., `population`) had even over 15% of records missing. The variable of interest life expectancy also had missing values. Our action here was to drop the records having their label missing and fill the others by linear interpolation grouped by `country` and ordered by `year`. This technique is to fill the current missing value by the next value and the previous value in the sequence of data. In case of that either the former or the latter is also missing, the current data is filled by one of non-null record. Rarely, if both adjacent data is missing, it will consider the second-adjacent data and so on.

Since there were a lot of missing values across features, this way of fixing null data would ensure that the pseudo data will be more reasonable, compared to the simple imputer with mean value of each column.

In [19]:

```
# Fill NaN using linear interpolation method
num_cols = df.drop(["status", "country"], axis=1).columns.tolist()
for country in df.country.unique().tolist():
    df.loc[df["country"]==country, num_cols] = df.loc[df["country"]==country, num_cols].sort_values
df.isnull().sum()
```

Out[19]:

```
country                          0
year                             0
status                           0
life_expectancy                  0
adult_mortality                  0
infant_deaths                    0
alcohol                          0
percentage_exp                   0
hepatitisb                       0
measles                          0
bmi                              0
under_five_deaths                0
polio                            0
tot_exp                          0
diphtheria                       0
hiv/aids                         0
gdp                              0
population                       0
thinness_1to19_years             0
thinness_5to9_years              0
income_comp_of_resources         0
schooling                        0
dtype: int64
```

Figure 2.2.2. Fill `NaN` by linear interpolation method.

II.3. Outlier Detection

The outlier observation was held on numerical features, using box plot technique, to get a deeper insight into the characteristics of the dataset.

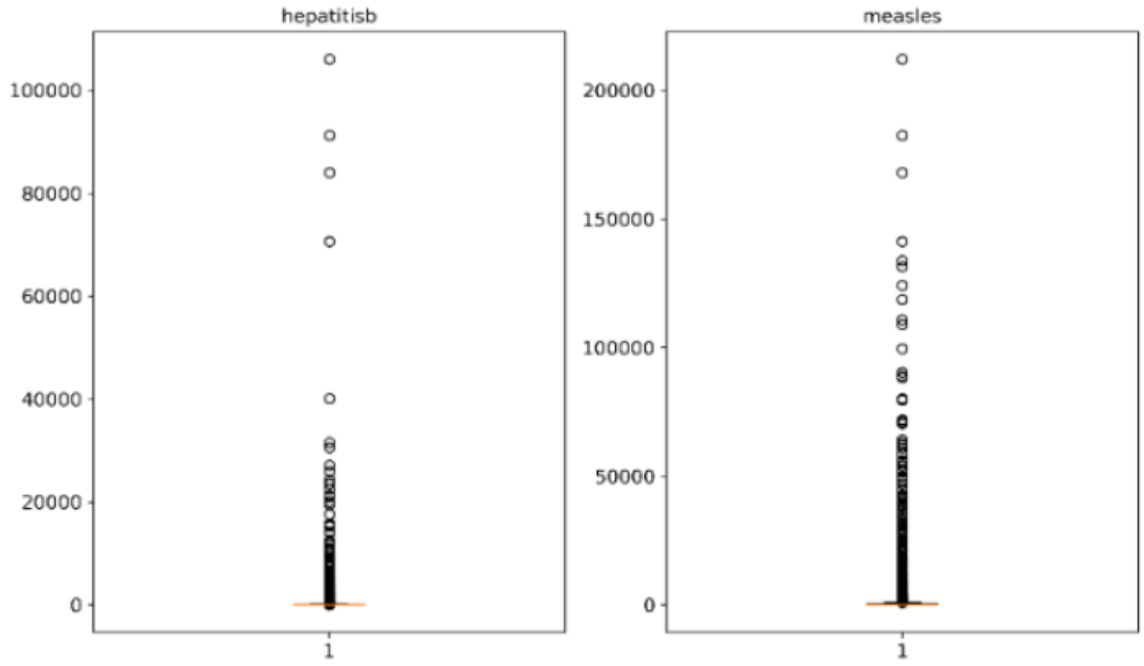


Figure 2.3.1. The box plots of `hepatitisb` and `measles`. These instances are calling for the fact that the data on each feature is spreading. The box plots of the other attributes can be found in folder `images` or the notebook `01_exploratory_data_analysis.ipynb` at the root of the project.

Speaking, the box plot technique can be described shortly through the below figure:

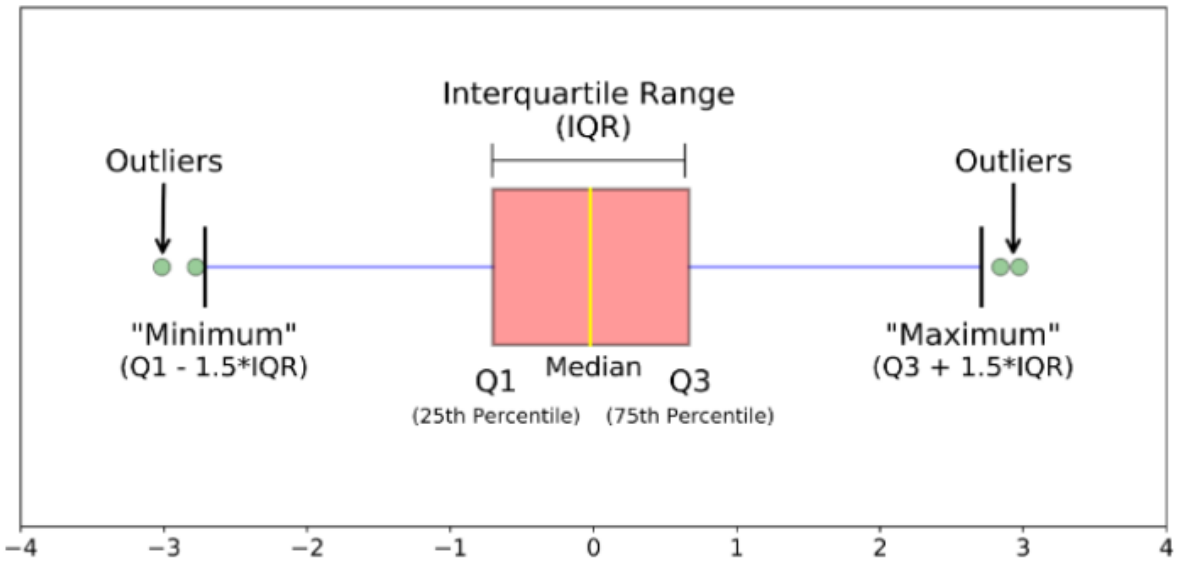


Figure 2.3.2. The structure of box plot.

A value is considered to be an outlier if it falls out of the range from the minimum percentile to the maximum percentile. By such measurements, we estimated the number and the percentage of the values across features considered as outliers.

Out[23]:

	number	percentage	upper_quantile	lower_quantile
hepatitisb	706	24.112022	131.0	43.0
measles	542	18.510929	905.625	-543.375
hiv/aids	542	18.510929	1.85	-0.95
population	438	14.959016	11461365.625	-6854871.375
gdp	424	14.480874	12480.144348	-7141.756107
under_five_deaths	394	13.456284	70.0	-42.0
percentage_exp	388	13.251366	1099.254858	-651.786572
diphtheria	316	10.79235	125.5	49.5
infant_deaths	315	10.758197	55.0	-33.0
polio	270	9.221311	127.0	47.0
income_comp_of_resources	240	8.196721	1.23	0.062
tot_exp	233	7.95765	13.91	-1.37
thinness_5to9_years	127	4.337432	15.85	-6.95
thinness_1to19_years	114	3.893443	15.85	-6.95
schooling	98	3.346995	20.85	2.85
adult_mortality	82	2.800546	459.0	-157.0
alcohol	32	1.092896	18.4	-9.76
bmi	20	0.68306	111.65	-35.95
life_expectancy	10	0.34153	94.6	44.2

Figure 2.3.3. The table of quantitative outlier observation using box plot technique.

One could deduce that the data collected was quite irregular and spreading on each feature. Some suggestions were to drop the anomalies in the training data, but they, on the other hand, were able to be the cases of reality. Consequently, dropping a large quantity of such records was likely to affect the models' predictive ability to deal with special cases, especially when their occurrence was quite frequent.

Therefore, the decision would be made based on which model was chosen and how the model performed at the time of machine learning.

II.4. Exploratory Data Analysis

We observed first the distribution and the range of each numerical feature through histograms.

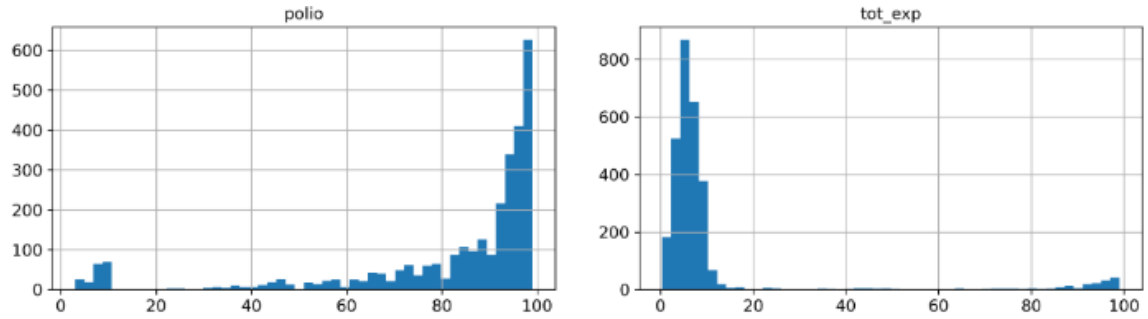


Figure 2.4.1. Histograms of two features (`polio` and `tot_exp`). The histograms of the other attributes can be found in folder `images` or the notebook `01_exploratory_data_analysis.ipynb` at the root of the project.

Many histograms were skewed; they extended much farther to one side of the median than to the another. This may make it a bit harder for some machine learning algorithms to detect the patterns. One measure was to transform these attributes later to have more bell-shaped distributions using normalization (or standardization). The transformation would then be included in the pipeline if a certain model really needed scaled data.

On the other hand, there were more records for the developing countries than those for the developed ones. The ratio of records of developing countries was around 82.5%. However, according to worlddata.info , 85.22% of countries were developing all over the world. Hence, the sample was describing the population rightly in this aspect.

The relationships among features were also worth exploiting. Since the models used were not just a simple linear regression and the characteristics of numerical features were visibly non-Gaussian and irregular, we explored the monotonic relationship among variables (Spearman's correlation) instead of just the linear relationship (Pearson's correlation).

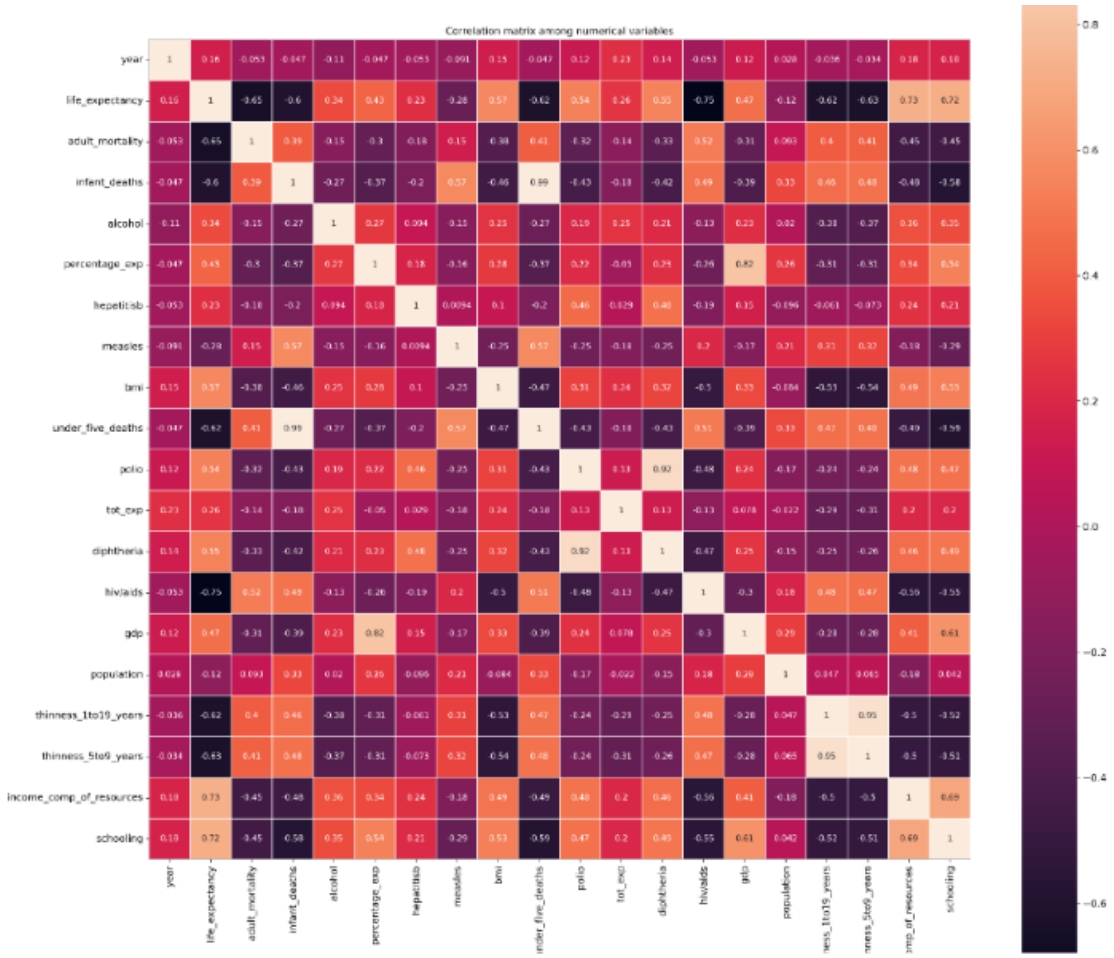


Figure 2.4.2. The correlation matrix among numerical variables.

Having acquired from the correlation matrix, we deduced the following highly correlated pairs:

```
In [37]: # Get the most correlated pairs of variables
tri_corr_mat = corr_mat.unstack()
cells_to_drop = get_redundant_pairs(df.drop(columns=["country", "status"]))
tri_corr_mat = tri_corr_mat.drop(labels=cells_to_drop).sort_values(ascending=False)

print("Top 10 positive correlated feature pairs:\n" + str(tri_corr_mat[0:10]))
print("\n----------------------------------------------------------------------")
print("\nTop 10 negative correlated feature pairs:\n"+ str(tri_corr_mat[-10:]))
print("\n----------------------------------------------------------------------")
print("\nTop 10 feature pairs:\n" + str(tri_corr_mat.abs().sort_values(ascending=False)[:10]))
```

```
Top 10 positive correlated feature pairs:
infant_deaths             under_five_deaths           0.993221
thinness_1to19_years      thinness_5to9_years         0.948435
polio                     diphtheria                  0.921455
percentage_exp            gdp                         0.821022
life_expectancy           income_comp_of_resources    0.729473
                          schooling                   0.722382
income_comp_of_resources  schooling                   0.691046
gdp                       schooling                   0.606093
measles                   under_five_deaths           0.572276
infant_deaths             measles                     0.571157
dtype: float64

----------------------------------------------------------------------

Top 10 negative correlated feature pairs:
hiv/aids           schooling                  -0.554349
                   income_comp_of_resources   -0.559156
infant_deaths      schooling                  -0.578362
under_five_deaths  schooling                  -0.587731
life_expectancy    infant_deaths              -0.600727
                   thinness_1to19_years       -0.618098
                   under_five_deaths          -0.618611
                   thinness_5to9_years        -0.627867
                   adult_mortality            -0.650007
                   hiv/aids                   -0.753642
dtype: float64

----------------------------------------------------------------------

Top 10 feature pairs:
infant_deaths             under_five_deaths           0.993221
thinness_1to19_years      thinness_5to9_years         0.948435
polio                     diphtheria                  0.921455
percentage_exp            gdp                         0.821022
life_expectancy           hiv/aids                    0.753642
                          income_comp_of_resources    0.729473
                          schooling                   0.722382
income_comp_of_resources  schooling                   0.691046
life_expectancy           adult_mortality             0.650007
                          thinness_5to9_years         0.627867
dtype: float64
```

Figure 2.4.3. The highly correlated feature pairs.

The three features hiv/aids, income_comp_of_resources and schooling correlated highly to our variable of interest. They seemed to be potential, significant predicators for the average age of a country. Therefore, the distribution of these attributes should be preserved on the part of dataset where developed model would be tested on, with regard of the assumption that our sample describes the population quite well. This might ensure that the result of the final test possibly approximates the true performance.

Besides, one can observe that there were many features extremely correlated (or even can be considered the same). Likewise, most features were pairwise correlated, which hinted an unexpected method to handle the missing values and the outliers more properly, compared to the linear interpolation method mentioned.

Life Expectancy and Status

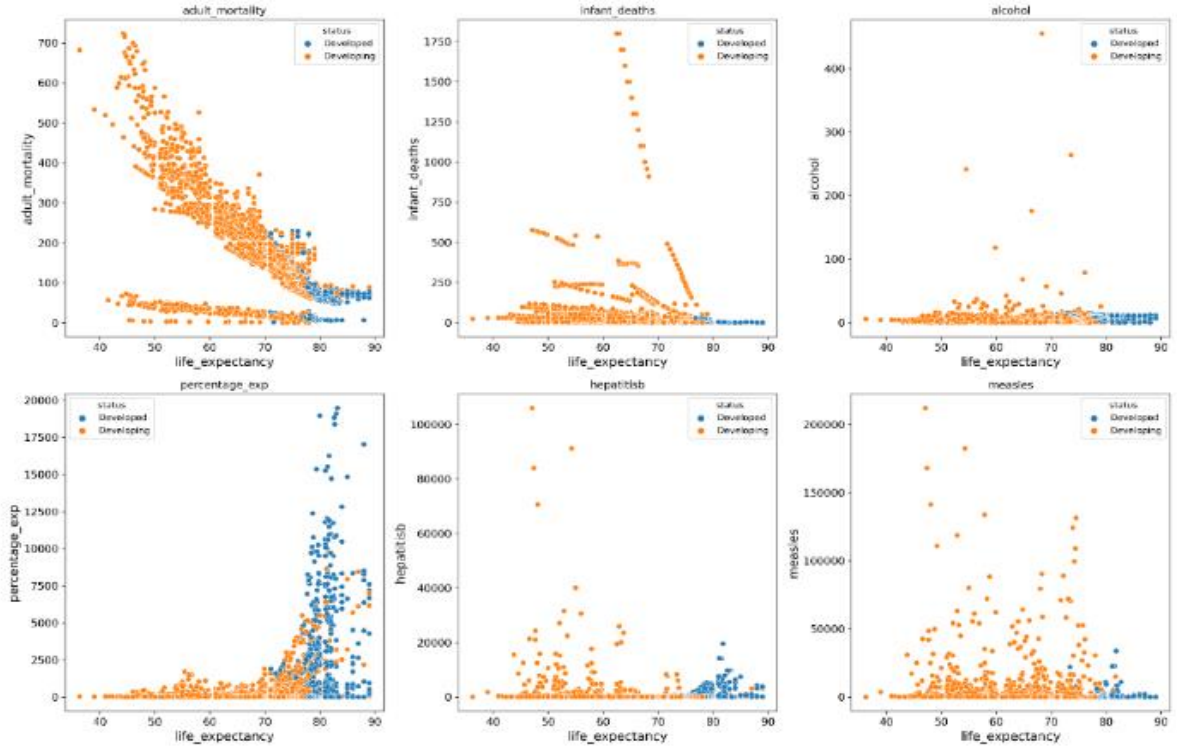


Figure 2.4.4. Life expectancy vs. other features, labeled by status. The full plots can be found in folder `images` or the notebook `01_exploratory_data_analysis.ipynb` at the root of the project.

The developing and developed countries were separated clearly on each scatter plot. This observation pointed out that status could also be an important factor to predict the life expectancy.

Countries and Continents

We have mentioned that country was problematic since there were hundreds of countries over the world, which were included in the dataset. A simple idea then ignited was to convert each country to the corresponding continent.

```
In [44]: df_continent["continent"].unique()

Out[44]: ['Asia', 'Europe', 'Africa', 'Americas', 'Oceania']
         Categories (5, object): ['Asia', 'Europe', 'Africa', 'Americas', 'Oceania']
```

Figure 2.4.5. The five continents considered.

With just five continents on consideration, the number of dummy variables reduced from hundreds to units, which tackled possible unexpected issues occurring during the model development.

III. Data Preprocessing

The exploratory data analysis shared clues about the data. The preprocessing step here will proceed the most general tasks to produce usable data for machine learning. Some specific tasks such as scaling will be carried out during the machine learning phase regarding the elements of models.

III.1. Create a Test Set

Initially, we separated the dataset into two subsets, one for training and the rest for the final testing. In our opinion, a standalone test set is necessary since the last benchmark of any model should not be held on the data where it has been developed. The model has already learned the patterns of the sample it was trained on, which makes it fit as well as possible the sample. Therefore, if the final benchmark is on the training sample, the model will be on a vantage point and the result is hence not appropriate. It is the case in practice when a model must deal with the unknown data stream.

More importantly, our data splitting has relied on stratified sampling. As analyzed, hiv/aids, income_comp_of_resources, schooling and status strongly correlate with life expectancy. In other words, they are significant to determine the average age of a country. Accordingly, regarding the assumption that the sample can describe the population decently, we designed the test set so that it could remain the characteristics of the sample, for the purpose that the final testing's results could fairly approach the true performance of the models. As a consequence, the result would be more credible.

Often, the stratified sampling is applied on categorical attributes. However, it does not mean this technique cannot be applied on continuous variables, especially when we can transform them into discrete variables. For example, income composition of resources (HDI) was categorized into four corresponding groups, according to the global convention (source attached in the notebook 02_data_preprocessing.ipynb). We therefore categorized the remaining two continuous features in the same manner and performed stratified sampling on the four attributes chosen.

In [137…
```
# Categorize income composition of resources (HDI)
# source: https://en.wikipedia.org/wiki/Human_Development_Index
bins = pd.IntervalIndex.from_tuples([(-0.009, 0.549), (0.549, 0.699), (0.699, 0.799), (0.799, 1)])
df["income_composition_of_resources_cat"] = pd.cut(df["income_composition_of_resources"], bins=bins
df["income_composition_of_resources_cat"].isnull().sum()
```
Out[137…
```
160
```
In [138…
```
# Categorize hiv/aids (infant deaths per 1000)
# no general criterion found, so let's categorize based on box plot in EDA
bins = pd.IntervalIndex.from_tuples([(0.099, 5.15), (5.15, 20), (20, 50.6)])
df["hiv/aids_cat"] = pd.cut(df["hiv/aids"], bins=bins)
df["hiv/aids_cat"].isnull().sum()
```
Out[138…
```
0
```
In [139…
```
# Categorize schooling (number of years of schooling)
# source: https://en.wikipedia.org/wiki/Educational_stage
bins = pd.IntervalIndex.from_tuples([(-1, 3), (3, 5), (5, 12), (12, 19), (19, 21)])
df["schooling_cat"] = pd.cut(df["schooling"], bins=bins)
df["schooling_cat"].isnull().sum()
```
Out[139…
```
160
```

Figure 3.1.1. Categorize the three continuous features.

As shown on Figure 3.1.1, there were missing values for categorized features, because the data splitting was held before any other preprocessing steps, including null handling. We then chose to drop them, totally dropping just 160 records (around 5.5% of the data). Due to the small size of the dataset, the dropped rows were then added back to the training set and the test set through common random sampling. This ensured that we did not waste any assets.

In [145…

```
# Check whether the proportions of status on the whole data and test set are close
def status_proportion(df):
    return df["status"].value_counts() / len(df)

compare_props = pd.DataFrame({
    "Overall": status_proportion(df_origin),
    "Stratified": status_proportion(test_set)
}).sort_index()

compare_props["Strat. %error"] = 100 * compare_props["Stratified"] / compare_props["Overall"] - 100
compare_props # good! :>
```

Out[145…

	Overall	Stratified	Strat. %error
Developed	0.174863	0.173497	-0.781250
Developing	0.825137	0.826503	0.165563

Figure 3.1.2. The re-check showed that the proportions of status on the whole data and test set were the same.

III.2. Preprocessing the Data

The whole preprocessing has been packaged into pipeline, which would help us reuse and transform the data conveniently across the different segments of the dataset. From the exploratory notebook, we utilized the flexible organizations of scikit-learning to build our own imputer (linear interpolation method) and feature engineer (convert country into continent), but they were still compatible to the framework's pipeline.

Afterwards, we designed the pipeline with the two custom transformers to transform the different datasets (training set and test set) and then get the dummy variables for `continent` and `status` (excluded `country`). At last, we saved all the preprocessed data for later usage.

IV. Clustered Linear Regression

We loaded the preprocessed data including the training set and the test set. The training set was used for developing the model (e.g., hyperparameter tuning, feature selection…), then the test set was employed once to estimate the true performance of the models in use. Regardless of the results, the model would not be tuned then in order to avoid the behaviors of fitting it to the test set.

Initially, we tried to train just a simple linear regression to the training data and the result of the fit on the training data poorly met our expectation. Therefore, the idea was to increase the number of linear regression models fitting the data simultaneously, but how to arrange these regressors on just one fit remained problematic. One method was proposed, that was clustering. In the development phase, a clustering algorithm is trained on the training set, so that the data is clustered into the different areas, each of which contains instances that can have relationships. After that, a linear regression model is assigned on each cluster, fitting the instances of that cluster and then predicting the future records that belong to the corresponding cluster. In our case, we used K-Means as the clustering model and Elastic net as the regression one (the primary predictor).

K-Means is a common method for clustering. Since the final performance was expected to minimize the generalized error of the whole model, appropriate clustering, in our opinion, was not necessary but a clustering that could bring a better performance during the training and validation. Therefore, K-Means, which was fast and efficient, demonstrated itself as a suitable candidate for this position.

Subsequently, elastic net was preferred over the other linear regression models (e.g., Ridge regression, Lasso regression…). Compared to the classical linear regression, a regularized linear regression like elastic net prevents the best fit on the training data of ordinary least squares method, which can even drag itself fitting outliers. In other words, ordinary least square is a low bias model, then it is well-suited to have its variance traded off a higher bias, which may result in higher overall predictive ability.

For the case of Lasso and Ridge regression, elastic net showed itself a greater choice. The critical point here was that the data, as analyzed, consisted of highly correlated features. The pattern of Ridge regressor is to share the weight shrinkage equally across the attributes, while that of Lasso's technique extremely eliminates some useless predicators by reducing their weights to zero. We have preferred Lasso regression because of the feature selection it could perform, our decision was however changed due to the severe collinearity among the attributes, as analyzed before. Particularly, for example, x and z are the two highly correlated variables and suppose both are centered and scaled (to mean zero, variance one). Then the ridge penalty on the parameter vector is $\beta_1^2 + \beta_2^2$ while the lasso penalty term is $|\beta_1| + |\beta_2|$. since the model is supposed to be highly colinear, so that x and z more or less can substitute each other in predicting Y, so many linear combinations of x , z where we simply substitute in part x for z , will work very similarly as predictors, for example $0.2x + 0.8z$, $0.3x + 0.7z$, or $0.5x + 0.5z$ will be about equally good as predictors. In this case, the lasso penalty in all three cases is equal, it is 1, while the ridge penalty differs, it is respectively 0.68, 0.58, 0.5, so the ridge penalty will prefer equal weighting of colinear variables while lasso penalty will break down. This was the reason we considered that ridge would work better with colinear predictors: When the data give little reason to choose between different linear combinations of colinear predictors, lasso will just "wander" while ridge tends to choose equal weighting. That last might be a better guess for use with future data.

As mentioned, Ridge was a good option, but its penalty term could not perform feature selection "automatically" like Lasso. Since the collinearity occurs, a foreseen expectation was to exclude the redundant, noisy columns of data. Therefore, we desired to bring back the behavior of Lasso, that opened an idea about a combination between the two penalty terms. One then could resolve the case of collinearity and one handle the feature selection. To sum up, elastic net became the rising star for this concern. Figure 4.1 will reveal geometrically the patterns of each regularized regression.

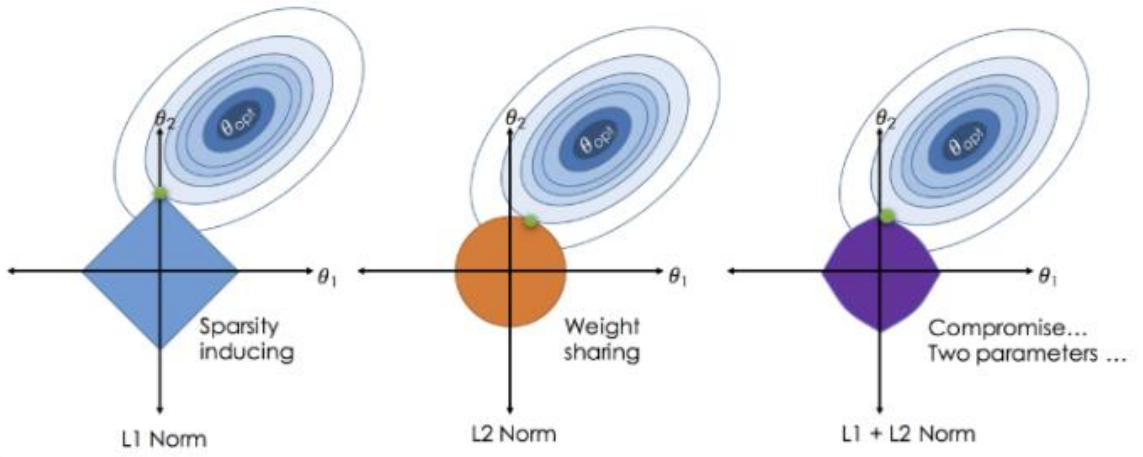


Figure 4.1. An image visualizing the pattern of the Lasso, the Ridge and the Elastic Net Regressors. Image Citation: Zou, H., & Hastie, T. (2005). Regularization and variable selection via the elastic net.

One more important detail was data scaling. Regularized linear regressors always require data to be scaled before training. The reason is that the penalty terms (i.e., L1 norm, L2 norm) are set on the magnitude of coefficients. Consequently, if there are two variables having the same impact on response but extremely different scale, the tiny scale one will be penalized more when the another is insignificantly constrained. This phenomenon on the unscaled data may cause an additional penalty on small-scale attributes by accident. Therefore, scaling (standardization) was included in our specific preprocessing for this section.

Since scikit-learning did not have a built-in clustered linear regression, we customized one, based on the skeleton of an estimator of the framework. The work aimed to design a sklearn-supported predictor so that we could utilize its cross validation to tune the model afterwards.

In [21]:

```
# Build a model of clustered Linear Regressions on clusters of K-Means
class ClusteredLinearRegression(BaseEstimator, TransformerMixin):

    def __init__(self,
                 clusterer=KMeans(n_clusters=4, random_state=42),
                 estimator=ElasticNet(alpha=0),
                 classifier=None):

        self.clusterer = clusterer
        self.estimator = estimator
        self.classifier = classifier # if the clusterer cannot predict for the new instances

    def fit(self, X, y=None):
        # cluster the data
        self.clusterer.fit(X)

        # fit the data of each cluster using a elastic net
        lin_regs = {}
        clusters = np.unique(self.clusterer.labels_)
        for cluster in clusters:
            lin_reg = clone(self.estimator)
            lin_reg.fit(X[self.clusterer.labels_ == cluster], y[self.clusterer.labels_ == cluster])
            lin_regs[cluster] = lin_reg

        self.lin_regs_ = lin_regs

        # train classifier to classify the new instances (if needed)
        if self.classifier != None:
            self.classifier.fit(X, self.clusterer.labels_)

        return self.lin_regs_

    def transform(self, X):
        return self.clusterer.transform(X)

    def predict(self, X):
        y_hat = np.zeros(X.shape[0])

        # predict the cluster of new instances
        if self.classifier != None:
            clusters_pred = self.classifier.predict(X)
        else:
            clusters_pred = self.clusterer.predict(X)

        # predict the labels of new instances based on clusters
        for cluster in self.lin_regs_.keys():
            if X[clusters_pred == cluster].shape[0] > 0:
                y_hat_cluster = self.lin_regs_[cluster].predict(X[clusters_pred == cluster])
                y_hat[clusters_pred == cluster] = y_hat_cluster

        return y_hat
```

Figure 4.2. The implementation of clustered linear regression.

One may notice that there is a classifier. The purpose the classifier plays is that it will predict the cluster of the new instances instead of the clustering model itself. The reason could simply be whether such algorithms are unable to label the cluster of the new instances, or it just simply improves the cluster assignment. In our case, we assessed that K-Means may wrongly label the upcoming data that would be on the boundaries among the clusters. The term revealed an issue that such points were far away from its centroid and then unreliable to belong that cluster. By this way, the classifier would be helpful and so it did. Intuitively, we used KNN as the classifier.

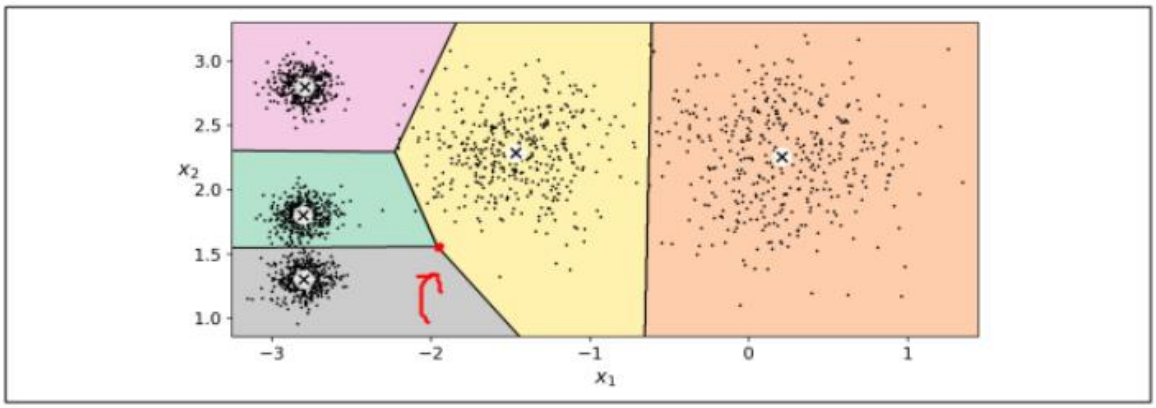


Figure 4.3. The red point, for example, is an issue to the clustering model's labeling.

We trained both models (one with KNN, one without) and cross-validated (using randomized search) them to find decent sets of hyperparameters for each of them.

In [37]:
```
# Evaluate score
cvres = lins_rndsearch.cv_results_
min_loss = np.inf
opt_param = None
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    if min_loss > np.sqrt(-mean_score):
        min_loss = np.sqrt(-mean_score)
        opt_param = params
print("Best score:", min_loss)
print("Best param:", opt_param)
```

```
Best score: 3.224143382222099
Best param: {'clusterer__n_clusters': 42, 'estimator__alpha': 0.04676566321361543, 'estimator__l1_rati
o': 0.09763799669573131}
```

In [40]:
```
# Show the best estimator
lins_rndsearch.best_estimator_
```

Out[40]:
```
ClusteredLinearRegression(clusterer=KMeans(n_clusters=42, random_state=42),
                          estimator=ElasticNet(alpha=0.04676566321361543,
                                               l1_ratio=0.09763799669573131))
```

Figure 4.4. The cross-validation results for K-Means & Elastic net.

In [38]:
```
# Evaluate score
cvres = kmeans_lins_knn_rndsearch.cv_results_
min_loss = np.inf
opt_param = None
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    if min_loss > np.sqrt(-mean_score):
        min_loss = np.sqrt(-mean_score)
        opt_param = params
print("Best score:", min_loss)
print("Best param:", opt_param)
```

```
Best score: 3.2030714717532476
Best param: {'classifier__n_neighbors': 13, 'classifier__weights': 'distance', 'clusterer__n_cluster
s': 42, 'estimator__alpha': 0.04676566321361543, 'estimator__l1_ratio': 0.09763799669573131}
```

In [39]:
```
# Show the best estimator
kmeans_lins_knn_rndsearch.best_estimator_
```

Out[39]:
```
ClusteredLinearRegression(classifier=KNeighborsClassifier(n_neighbors=13,
                                                           weights='distance'),
                          clusterer=KMeans(n_clusters=42, random_state=42),
                          estimator=ElasticNet(alpha=0.04676566321361543,
                                               l1_ratio=0.09763799669573131))
```

Figure 4.5. The cross-validation results for K-Means, Elastic net & KNN.

As shown on Figure 4.4 and 4.5, the performance of the clustered linear regressor with K-Means, Elastic net and KNN was better than that of the another. It was then put on the test set to deduce the final performance of the model.

In [47]:
```
# Evaluate the model on test set
y_test_pred = final_kmeans_lins_knn.predict(X_test_prep)
evaluate(y_test_prep, y_test_pred)
```

```
RMSE:  3.1814351000713206
MAE:  10.121529295965814
R2:  0.8894194732863653
```

In [48]:
```
# Compute 95% confidence interval for generalization error (rmse only)
from scipy import stats
confidence = 0.95

squared_errors = (y_test_pred - y_test_prep) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
                         loc=squared_errors.mean(),
                         scale=stats.sem(squared_errors))) # t-distribution
```

Out[48]:
```
array([2.85096809, 3.48066654])
```

Figure 4.6. Observe the performance of the best model on the test set.

Last to remind, we did not forget to transform the test set (scaling) like what we have done before. Also, the testing was held on a pure model that has been re-trained just once before testing.

V. Feature Correlation Imputation and Ridge Regression

FEATURE CORRELATION BASED MISSING DATA & RIDGE REGRESSION.

4.1. Algorithmn Outline

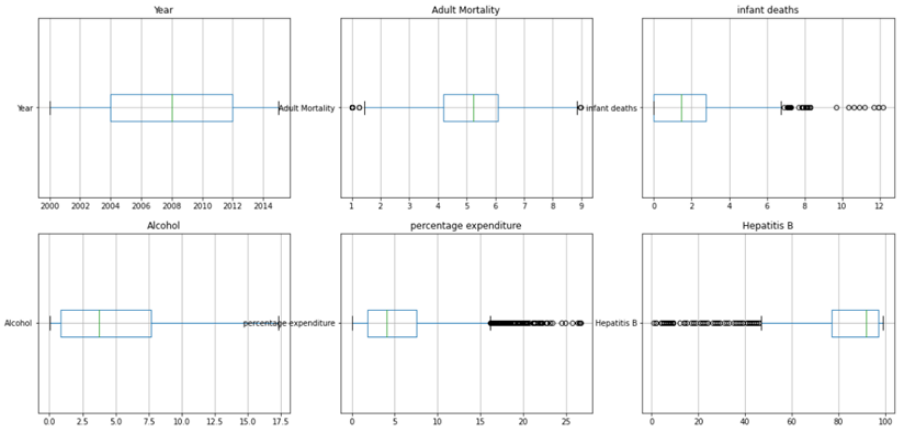
- 1st step: Outlier detection and reduction by using Boxplot and transformation method

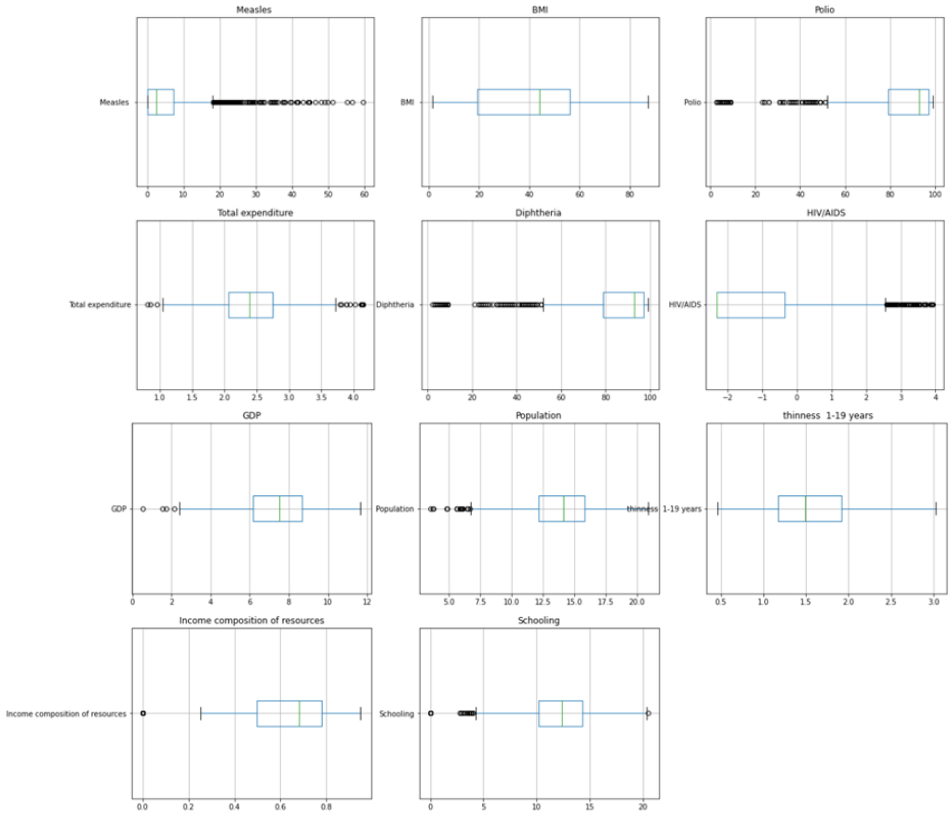
- 2nd step: Missing value handling

- 3rd step: Apply Ridge Regression model

4.2. Outlier detection and reduction

Fistly, we used Boxplot to know the distribution of numerical data and skewness through displaying the data quartiles (or percentiles) and averages.

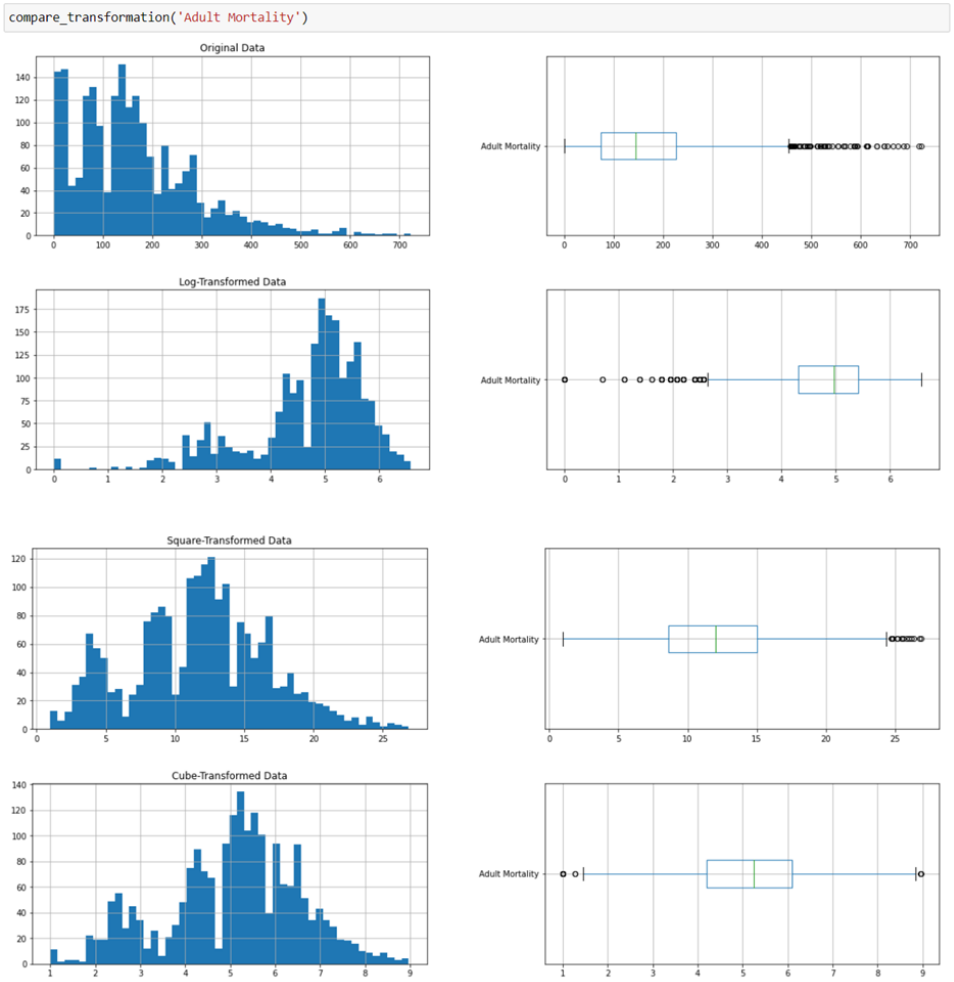




Box plot of 17 features

As we can see, there are a lot of outliers. Besides, the data is unevenly distributed, or it can be said to be very skewed. So we rescaled on features which have skew data, such as “Adult Mortality”, “infant deaths”… . Therefore, we decided to use Tranformation which is a replacement that changes the shape of a distribution or relationship. There are many reasons for transformation :Convenience, Reducing skewness, Equal spreads, Linear relationships, Additive relationships comprehensive…

We compared 3 methods: Log Transformation(Transform the response variable from y to log(y)) , Square Root Transformation(Transform the response variable from y to $\sqrt{y}$.) and Cube Root Transformation(Transform the response variable from y to $y^{1/3}$), then, we chose the best transformation to reduce the number of outliers. Some features can not use Log Transformation because of range of data, for these features, we just compared 2 methods. We give an example below:



For Adult Mortality, the original data had a lot of outliers. Among 3 methods, we can see easily that after using Cube Root Transformation, the data just had few outliers left. So we chose this method to transform data.

However, we can not see the difference when using tranformation method on some features, they could not reduce outliers significantly. So, for these features, we did not affect their data.

4.3 Handling missing data

First, with the test set, the null value of a feature will be filled by its feature's average value. Since it's the test set and we'll later use it for algorithmn comparison so we're not gonna do anything else to this set.

As the train set performs how well our Ridge Regression would do, to improve the accuracy, we're gonna handle these null value in a different way. Rather than filling the missing values with their *whole feature's mean value* (like what we have done to the test set), we can handle better these missing values by data which is more "precise" using "Feature Correlation based Missing Data Imputation" (FCMI).

4.3.1 Method approach:

FCMI is a missing-value-handling method in which the null values of a feature will be imputed based on its correlation with other feature. Thoroughly speaking, if the correlation between them is high, we say that they capture information about each other. If one column happens to have a missing value in one of its instances, we can use the other column to predict the missing value. That is our strategy.

4.3.2 Application of FCMI in our data

For a better understanding of this approach, we take “Schooling” and “Life expectancy” for example

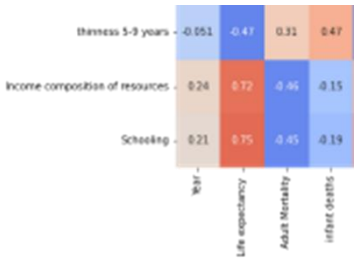
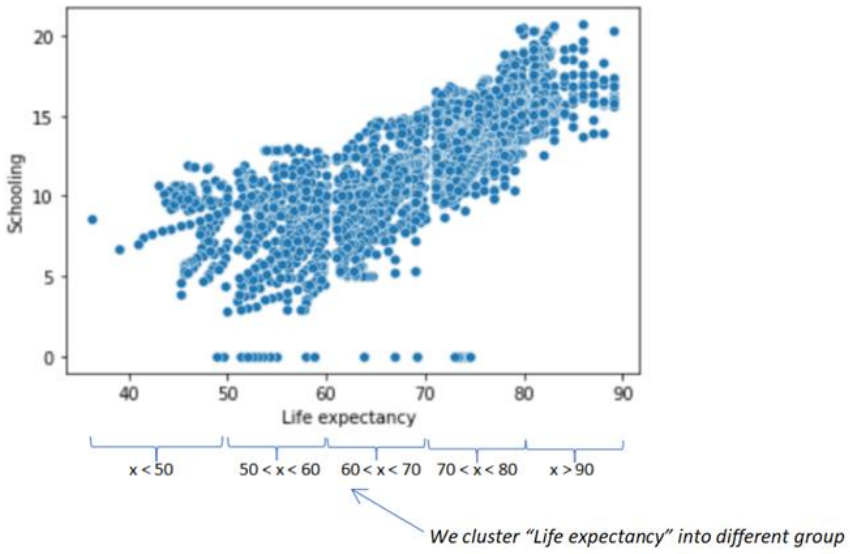


Fig 4.4.2.1: Correlation heatmap among some features in our data

For the heatmap above, it’s clearly that “Schooling” has a great correlation with “Life expectancy”, and since “Life expectancy” is non-null so we’re gonna use it to impute missing values in “Schooling”. To do this, first we build a scatter plot to visualize these two features, and then we cluster “Life expectancy” into different groups.



After that, for each missing value of “Schooling”, we consider which group does its “Life expectancy” value belong to, and then we’re gonna impute the missing data by the average “Schooling” values which has the corresponding “Life expectancy” belong to the same group. For example:

Belongs to group 70 < x < 80

Life expectancy	Schooling
45	7
62	12
78	null
68	17

This null value will be filled by the average value of 'Schooling' where its corresponding "Life expectancy" value belongs to (70;80)

We're keep doing this until all missing data have been imputed. Then our data is ready for our Ridge Regression.

Let's see how our Ridge Regression do with our model:

```
ridge_reg = linear_model.Ridge(alpha=10, max_iter=1000, tol=0.01)
ridge_reg.fit(X_train, y_train)
ridge_reg.score(X_test,y_test)
```

✓ 0.2s

```
0.9116354349012894
```

The score is not bad, but maybe we can even make it better by choosing the appropriate value for each parameter.

4.4 Ridge Regression parameters determination

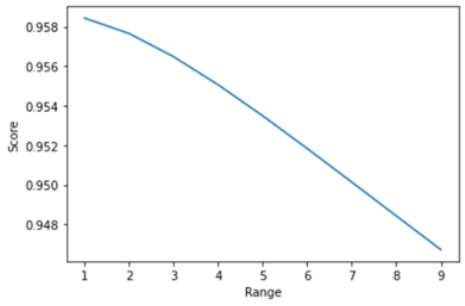
Since Python supports Ridge Regression, let's have a look at its function:

sklearn.linear_model.Ridge

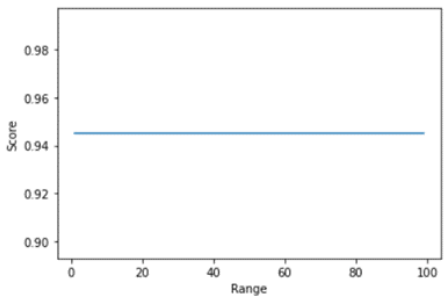
class **sklearn.linear_model.Ridge**(*alpha=1.0, *, fit_intercept=True, normalize='deprecated', copy_X=True, max_iter=None, tol=0.001, solver='auto', positive=False, random_state=None*) [source]

We can see there are pretty much parameters for this, but we're gonna consider three most important: *alpha*, *max_iter* and *tol.*

Let's consider which value of these parameter can maximize our R2 score. First, we're gonna plot a figure with the value of alpha's range runs from 0 to 1, the same with tol.



Our score as "alpha" value changes



Our score as "toll" value changes

By above graphs :

- As *alpha* gets close to 0, our R2 score increase, so we choose *alpha* = 0.001 (we're not taking 0 because the objective is equivalent to ordinary least squares, solved by the Linear Regression object)

- As *tol* changes, our R2 score stays still, so we take *tol* = 0.01 because it's common.

- As *max_iter* defines maximum number of iterations, we just take a great number: 100000.

Until this point, our Ridge Regression model is finished.

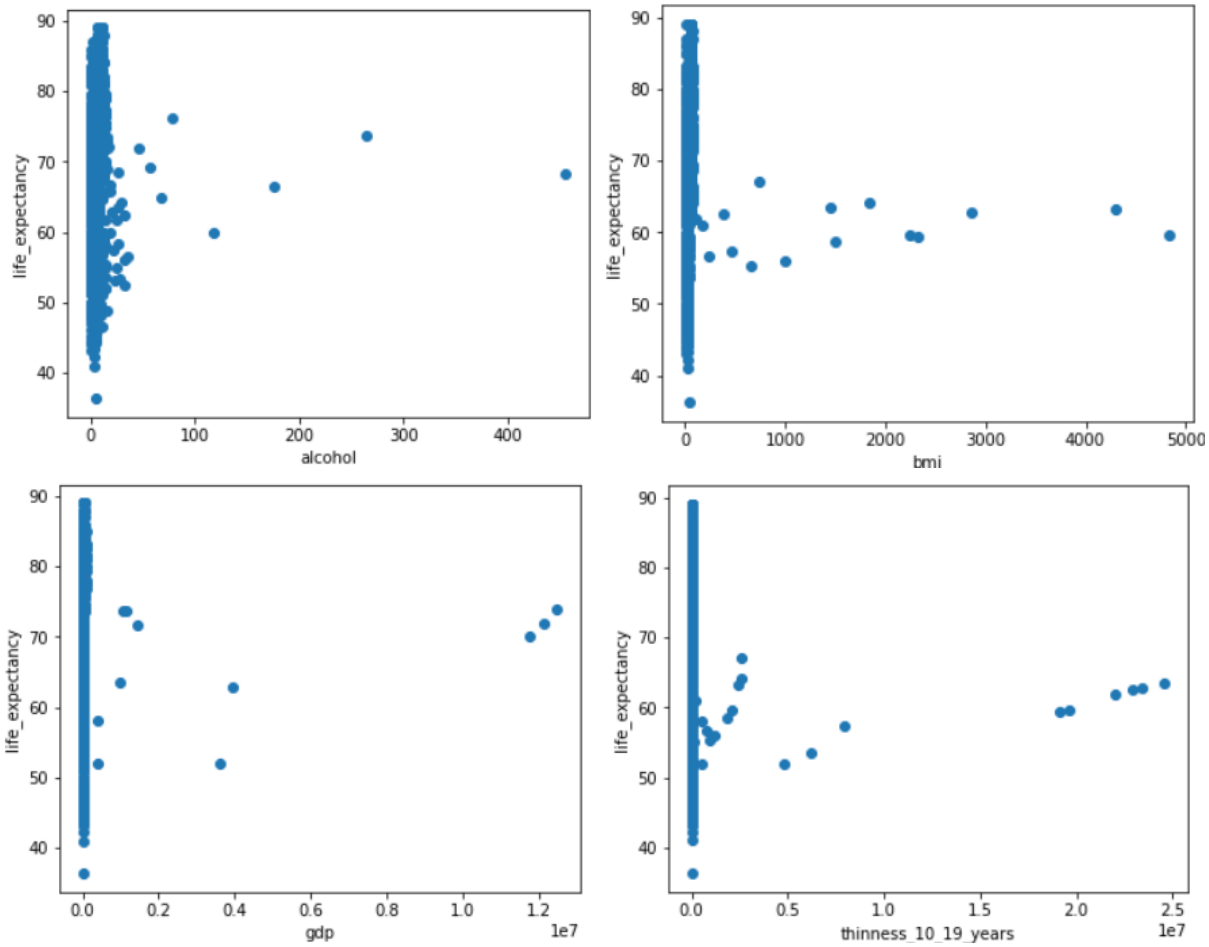
VI. Decision Trees and Ensembles

After diving into Linear Regression, in this part, we use more complex models such as Decision tree and Ensemble methods, likewise, we also explore data and remove outliers with another ensemble method that is the extension of isolation forest. This chapter is our approach with Decision tree and some ensemble learning methods based on Decision tree.

For this part, we implement the algorithm with Scikit-learn library (sklearn) for prediction model and H2O library (h2o) for outlier detection.

6.1. Explanatory data analysis

- After plotting explanatory variables to response variable, we observe that there are 4 explanatory variables that don't have any effect in separation, they are "alcohol", "bmi", "gdp", "thinness_5_9_year".



As you can see in the plot, while "life_expectancy" has high variances, the explanatory variables merely distribute around 0, although there are still some data points that don't follow pattern, probably they are outliers. So we decide to remove these 4 attributes.

6.2. Outlier detection

- Although model like random forest can reduce effect from outlier, the others badly suffered from it, such as Decision tree, hence, we decide to provide algorithm to remove outliers, get better result for our model, hence, we choose Extended Isolation Forest (EIF) for our purpose as this approach mainly focus on ensemble learning.

- Isolation Forest and its extension base on an assumption that the anomaly instances are usually rare and different from those of normal instances in a given data set, which makes them more susceptible to isolation in a number of binary tree structures than the normal instances. The resulting IF algorithm is a convenient solution to detect anomalies without assumptions on the data distribution and it is computationally efficient.

- However, this algorithm suffers from a bias due to the way trees are created. Indeed, by randomly choosing one dimension to split the data, parallel hyperplanes are used (with a normal vector collinear to the selected dimension), and data spread around stripes parallel to the axis and passing through the cluster have a lower anomaly score, as depicted in Fig. 1.

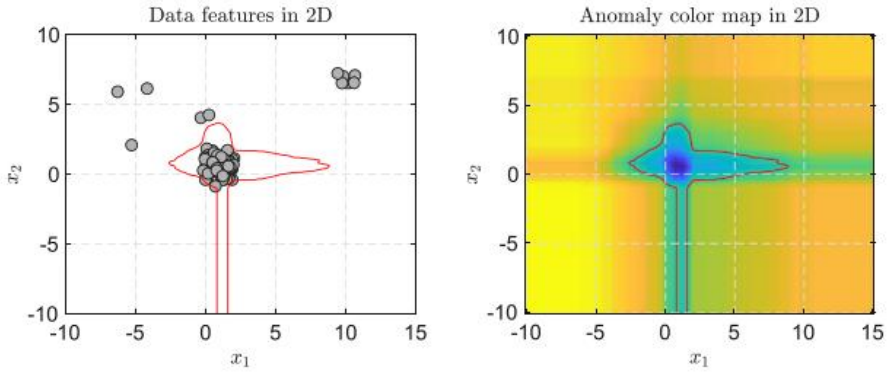


Fig. 1. Illustration of IF problems using artificial 2D data. Training data are depicted in the left figure as well as the curve $s(\boldsymbol{x}, n) = s_0$ (displayed in red). The right figure shows the heat map of the anomaly score (dark blue corresponds to values next to 0 and light yellow to value close to 1). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

- Therefore, we drive to another extension of isolation forest, which is proved to enhance results. In Extended Isolation Forest (EIF), the selection of the branch cuts still only requires two pieces of information, but they are: 1) a random slope for the branch cut, and 2) a random intercept for the branch cut which is chosen from the range of available values of the training data.

- For an N dimensional dataset, selecting a random slope for the branch cut is the same as choosing a normal vector, $\vec{n}$, uniformly over the unit N-Sphere. This can be accomplished by drawing a random number for each coordinate of $\vec{n}$ from the standard normal distribution $\mathcal{N}$ (0, 1) [8]. This results in a uniform selection of points on the N-sphere. For the intercept, $\vec{p}$, we simply draw from a uniform distribution over the range of values present at each branching point. Once these two pieces of information are determined, the branching criteria for the data splitting for a given point $\vec{x}$ is as follows:

$$(\vec{x} - \vec{p}) \cdot \vec{n} \geq 0$$

- Using this, during the training phase, the algorithm will create a number of a tree by picking a random slope and a random intercept from the training data. The data points are then sent down the left or the

right branch according to the rules of the algorithm. By creating many such trees, we can use the average depths of the branches to assign anomaly scores. So any new observed data point can traverse down each tree following such trained rules. The average depth of the branches this data point traverses will be translated to an anomaly score using equation (1).

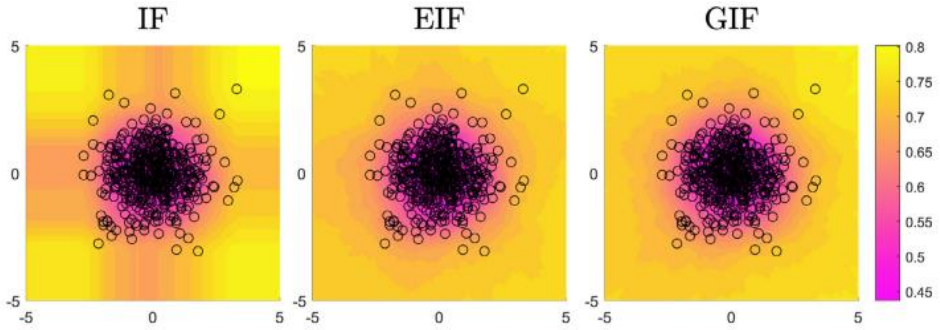
$$s(x,\, n) = 2^{-\frac{E(h(x))}{c(n)}} \quad (1)$$

where $E\big(h(x)\big)$ is the mean value of the depths a single data point, x, reaches in all trees. $c(n)$ is the normalizing factor defined as the average depth in an unsuccessful search in a Binary Search Tree (BST):

$$c(n) = \; 2H(n \; - \; 1) - \; \left(\frac{2(n-1)}{n}\right) \quad (2)$$

where H(i) is the harmonic number and can be estimated by $\ln(i) + \; 0.5772156649$ (Euler's constant) and n is the number of points used in the construction of trees.

- The important property of concentrating where the data is clustered, persists. The intercept points $\vec{p}$ tend to accumulate where the data is, since the selection of these points gets restricted to available data at each branching point as we move deeper into the tree. This results in more possible branching options where the data is clustered and fewer possible branching where there is less concentration of data. However, there are no regions that artificially receive more attention than the rest. The results of this are score maps that are free of artifacts previously observed. Below is an illustration.



6.3. Decision trees (regression)

We use DecisionTreeRegressor of sklearn which provides the tree with an optimized version of the CART (Classification and Regression Trees) algorithm. CART constructs binary trees meaning that the tree is spilt into 2 at every node except for leaves and this strategy is kept despite the characteristic of the features. The feature and threshold which yield the largest information gain at a node are chosen. An important thing of this implementation is that categorical variables are not supported, and we have to change them into numerical.

Note for parameters:

- Decision trees tend to overfit on data with a large number of features. Getting the right ratio of samples to number of features is important, since a tree with few samples in high dimensional space is very likely to overfit. So, we consider the **max_features** parameter: The number of features to consider when looking for the best split.

- Prevent overfitting with **min_sample_leaf** (the minimum number of samples required to be at a leaf node) and **max_depth** (the maximum depth of the tree)
- **criterion**: "squared_error" is the function to measure the quality of a split.
- **splitter**: "best" (choose the best split at each node).

Decision Trees are sensible with noise which makes them unstable and leads to high variance. We continue using tree-based ensemble methods to overcome this problem, with random forests, adaptive boosting and gradient boosting.

6.4. Random Forests (Regression)

First, we implement a bagging method: Random Regression Forests.

We use the RandomForestRegressor implementation of sklearn, with the base learners are the above decision trees (DecisionTreeRegressor).

Note for parameters:

- **boostrap**: True (bootstrap samples are used when building trees).
- We have to consider the number of trees in the forest: **n_estimators.**
- The same other parameters as decision trees.

6.5. AdaBoost (with regression decision trees)

In bagging methods, each leaner is built separately; therefore, can make different mistakes. That is the reason why the bagging methods can theoretically reduce the bias but still suffer from a large bias. We continue with boosting algorithms, which aim at reducing the bias by training the leaners sequentially.

The AdaBoostRegressor of sklearn is an implementation of AdaBoost.R2 algorithm with some changes, below are some important points:

- Error/Loss

We define:

$$D = \sup \left| y_i^{(p)}(x_i) - y_i \right|,$$

where $y_i^{(p)}(x_i)$ is the predicted value for the sample **i** of the **pth** tree of the model.

The error ϵ is now the average loss calculated by one of these loss functions: linear, square law or exponential.

$$L_i = \frac{|\, y_i^{(p)}(\boldsymbol{x}_i) - y_i \,|}{D} \qquad (linear)$$

$$L_i = \frac{|\, y_i^{(p)}(\boldsymbol{x}_i) - y_i \,|^2}{D^2} \qquad (square\ \ law)$$

$$L_i = 1 - exp\left[\frac{-|\, y_i^{(p)}(\boldsymbol{x}_i) - y_i \,|}{D}\right] \qquad (exponential)$$

$$\epsilon = \sum \quad L_i w_i$$

- Learning rate (η): This parameter is used to control the amount the weights are changed at each iteration

$$\beta = \frac{\epsilon}{1-\epsilon}$$

$$\alpha = \eta \ln\left(\frac{1-\epsilon}{e}\right) = -\eta ln\beta$$

$$w_i \rightarrow \frac{w_i \beta^{\eta(1-L_i)}}{Z}$$

where Z is the normalization factor.

- Final model: The final model is the chosen as the **weighted median** of all weak learners.

For a particular input x_i, each of the T trees makes a prediction h_t , t =1, ...,T. Obtain the cumulative prediction h_f using the T predictors:

$$h_f = \inf\left\{y \in Y: \sum_{t:\, h_t \le y} \log\left(\frac{1}{\beta_t}\right) \ge \frac{1}{2}\sum_t \log\left(\frac{1}{\beta_t}\right)\right\}$$

This is the **weighted median**. Equivalently, each machine h_t has a prediction $y_i^{(t)}$ on the **i'th** pattern and an associated β_t. For pattern **i** the predictions are relabeled such that for pattern **i** we have:

$$y_i^{(1)} < y_i^{(2)} < \cdots < y_i^{(T)}$$

(Retain the association of the β_t with its $y_i^{(t)}$). Then sum the $\log\left(\frac{1}{\beta_t}\right)$until we reach the smallest **t** so that the inequality is satisfied. The prediction from that machine **t** we take as the ensemble prediction. If the β_t were all equal, this would be the median.

Note for parameters:

- **base_estimator**: None, meaning that it is DecisionTreeRegressor(max_depth = 3)
- **n_estimator**: Considering the number of weak learners. (One thing interesting of Adaboost is that if we continue increasing the number of weak models, the final model seems not to get to overfitting.)
- **loss**: Choosing the loss function

- **learning_rate**: Choose “good” learning rate

6.6. Gradient Boosting (Regression)

Gradient Tree Boosting or Gradient Boosted Decision Trees (GBDT) is a generalization of boosting to arbitrary differentiable loss functions.

We use the GradientBoostingRegressor of sklearn. The strategy is explained clearly at: https://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting

Gradient boosting regression involves three elements:

- Loss function: must be differentiable.
- Weak learner: (short) decision trees
- Additive model:

$$\hat{y}_i = F_M(x_i) = \sum_{m=1}^{M} h_m(x_i)$$

- Trees are added one at a time, and existing trees in the model are not changed.

$$F_m(x) = F_{m-1}(x) + h_m(x),$$

- A gradient descent procedure is used to minimize the loss when adding trees.

$$h_m = \arg\min_h L_m = \arg\min_h \sum_{i=1}^{n} l(y_i, F_{m-1}(x_i) + h(x_i)),$$

- After calculating the loss, to perform the gradient descent procedure, we must add a tree to the model that reduces the loss (i.e., follow the gradient). We do this by parameterizing the tree, then modify the parameters of the tree and move in the right direction.

Using a first-order Taylor approximation, the value of l can be approximated as follows:

$$l(y_i, F_{m-1}(x_i) + h_m(x_i)) \approx l(y_i, F_{m-1}(x_i)) + h_m(x_i) \left[\frac{\partial l(y_i, F(x_i))}{\partial F(x_i)} \right]_{F=F_{m-1}}.$$

Note: Briefly, a first-order Taylor approximation says that $l(z) \approx l(a) + (z-a)\frac{\partial l(a)}{\partial a}$. Here, z corresponds to $F_{m-1}(x_i) + h_m(x_i)$, and a corresponds to $F_{m-1}(x_i)$

The quantity $\left[\frac{\partial l(y_i, F(x_i))}{\partial F(x_i)} \right]_{F=F_{m-1}}$ is the derivative of the loss with respect to its second parameter, evaluated at $F_{m-1}(x)$. It is easy to compute for any given $F_{m-1}(x_i)$ in a closed form since the loss is differentiable. We will denote it by g_i.

Removing the constant terms, we have:

$$h_m \approx \arg\min_h \sum_{i=1}^{n} h(x_i) g_i$$

This is minimized if $h(x_i)$ is fitted to predict a value that is proportional to the negative gradient $-g_i$. Therefore, at each iteration, **the estimator h_m is fitted to predict the negative gradients of the samples**. The gradients are updated at each iteration. This can be considered as some kind of gradient descent in a functional space.

(Images from https://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting)

- The output for the new tree is then added to the output of the existing sequence of trees in an effort to correct or improve the final output of the model.
- A fixed number of trees are added or training stops once loss reaches an acceptable level or no longer improves on an external validation dataset.
- Learning rate is used to scale the contribution of each weak learner by a constant factor ν

$$F_m(x) = F_{m-1}(x) + \nu h_m(x)$$

Note for parameters:

- **loss:** choosing appropriate loss function.
- **learning_rate:** Choose “good” learning rate
- **n_estimators:** Considering the number of weak learners.
- **max_depth**: choosing max_depth = 3
- The same other parameters as decision trees.

VII. Result

We rate our learning methods by calculating the r2_score (R-squared score), mae (mean absolute error), rmse (root mean square error) with Scikit-learn functions. The table below shows the highest score and correctness we received when using different methods:

	Before tunning			After tunning		
	r2_score	**mae**	**rmse**	**r2_score**	**mae**	**rmse**
Kmeans + Elastic Net	0.72289	3.73580	5.004032	0.88652	2.30974	3.22289
Kmeans + Elastic Net + KNN	0.72362	3.71434	4.99736	0.88942	2.28921	3.18144
Ridge Regression	0.91163	2.020412	1.695179	0.95117	1.373056	1.46153
Decision Tree	0.93293	1.45820	2.47761	0.93356	1.62018	2.46609
Random Forest	0.96000	1.21685	1.91345	0.96170	1.19757	1.87229
AdaBoost	0.89586	2.40390	3.08734	0.89732	2.36688	3.06569
Gradient Boosting	0.93926	1.41665	1.92984	0.95805	1.32736	1.95943

Overall, the two clustered linear regression implementations have the lowest accuracy, 0.88 for r2_score. AdaBoost is slightly better (0.89) and the Decision Tree model reaches a much better score (0.93). Ridge Regression, Random Forest and Gradient Boosting have the highest scores, around 0.95-0.96.

	Train set			Test set		
	r2_score	**mae**	**rmse**	**r2_score**	**mae**	**rmse**

Kmeans + Elastic Net	0.92470	1.92785	2.60837	0.88652	2.30974	3.22289
Kmeans + Elastic Net + KNN	0.92470	1.92785	2.60837	0.88942	2.28921	3.18144
Ridge Regression	0.96108	1.23162	1.86058	0.95117	1.37305	1.461533
Decision Tree	0.97488	0.94816	1.51422	0.93356	1.62018	2.46609
Random Forest	0.99490	0.42702	0.68167	0.96170	1.19757	1.87229
AdaBoost	0.90671	2.26713	2.91816	0.89732	2.36688	3.06569
Gradient Boosting	0.96007	1.40816	1.90917	0.95805	1.32736	1.95943

VIII. Conclusion

This report has presented our process to solve a regression problem. We first explored the dataset, proposed different strategies to fill missing data and also detect outlier. The two main learning strategies are using linear regression and tree-based methods, requiring different ways to develop algorithms and optimize the performance of the model. At the end, the results we received is quite good: the models all score 0.88-0.96 over 1.0 on the test set, measured by the r2_score function.

IX. References

1. Julien Lesouple, Cédric Baudoin, Marc Spigai, Jean-Yves Tourneret, Generalized isolation forest for anomaly detection, 2021.
2. Sahand Hariri, Matias Carrasco Kind, Robert J. Brunner, Extended Isolation Forest, 2020.
3. Harris Drucker, Improving Regressors Using Boosting Techniques, 1997.
4. Scikit-learn: Ensemble methods: https://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting