General approach to all regression problems

Applied to an Example Dataset From Cleaning to Model Evaluation

Saman Teymouri

August 2024

CONTENTS

ABSTRACT

This paper studies the Medical Cost Personal dataset. The objective of this study is to predict the continuous numeric target value which is the insurance cost of contractors with different conditions. The paper involves different phases of data analysis such as data exploratory analysis, data cleaning, model selection and training, and finally model interpretation and evaluation. This paper utilizes five algorithms including linear regression, ridge regression, decision tree regression, random forest regression, and XGBoost regression which creates the best results. The program written for this study is designed to be applicable, with slight modifications, for any regression problem with any dataset. It tries to minimize the redundancy in code, and every line of code has comprehensive comments. The links to the full version of the program are also available in the appendix section.

INTRODUCTION

In the digital world, the power of using data efficiently provides huge opportunities for companies. This leads industries to rely more on advanced technologies such as data analysis and machine learning for informed data-driven decision-making. These technologies help businesses with different types of analysis including descriptive, predictive, and prescriptive.

Predictive analytics utilizes a sort of statistical methods and machine learning techniques to analyze past and current data to predict the future as accurately as possible (Miller and Forte, 2017). Predictive analytics can be deployed in various industries to solve different business problems such as banking, healthcare, human resources (HR), marketing and sales, supply chain, etc (IBM, 2023). This means that each business that knows what to expect based on what has happened in the past, has a lot of advantages in different areas including security, risk reduction, operational efficiency, and improved decision-making (IBM, 2023).

Data analysis involves data collection and exploratory analysis, data cleaning and preparation, model selection and training, model interpretation and evaluation, and model optimization (Raja, 2023). Python provides a robust and practical platform for addressing all the above steps. Libraries such as NumPy (Van der Walt et al., 2011), Pandas (McKinney et al., 2011), and Matplotlib (Hunter, 2007) enable users to handle large datasets, extract descriptive information, and visualize them. Furthermore, the Scikit-learn library (Pedregosa et al., 2011) helps them build models for classification, regression, clustering, and deep learning.

In real-world scenarios, data analysis and machine learning are applied to different domains, including customer segmentation, anomaly detection, and sentiment analysis for marketing issues. Additionally, these techniques are used for predicting stock prices, disease diagnosis, and fraud detection (Johnston and Mathur, 2019). Since some of these fields are related to health or other critical issues, they need much accuracy. This amplifies the importance of predictive data analytics as the key to having a clear vision for the future.

The integration of Python in data analysis and machine learning has opened new paths in informed decision-making for solving complex real problems. Python's role in the future of AI and ML is undeniable. Based on its extensive use in the industry and its versatility, Python continues to shape the landscape of artificial intelligence and machine learning (Vaishalipal, 2024).

TASK 1: DESCRIPTIVE ANALYSIS AND DATA CLEANING

1.1 DESCRIPTIVE ANALYSIS

This study selects the Medical Cost Personal dataset (Choi, 2017) that contains insurance costs for contractors with different characteristics. This section includes loading the dataset and creating explanatory information.

At first, the program opens the dataset and extracts the first five rows of it to understand the structure better (Figure 3). load_data function does this (Figure 1).

```python
def load_data(file_path : str, logger : Logger = None) -> pd.DataFrame:
    # opening the file and show 5 first rows
    logger.log("#################### First 5 rows of original data ######################")
    # open csv file and load it into a dataframe
    data = pd.read_csv(file_path)
    # eliminate unnecessary columns
    data = data.drop(columns=unnecessary_cols)
    # return the first 5 rows of the dataset
    logger.log(data.head())
    return data
```

Figure 1: Loading the dataset

The main function is the start point of the program and calls all needed functions in a desired order. The main function and all needed libraries are shown below (Figure 2).

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import seaborn as sns
from sklearn.linear_model import LinearRegression, Ridge
from xgboost import XGBRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.tree import DecisionTreeRegressor, export_graphviz
from sklearn.ensemble import RandomForestRegressor
import graphviz
from sklearn.model_selection import train_test_split, GridSearchCV
from general import Logger
import io
from sklearn.preprocessing import minmax_scale, LabelEncoder
import warnings

# Target variable name is assigned here and used in the code globally
target_col = "charges"
# unnacessary columns should be included here to get eliminated before starting the process
unnecessary_cols = []

def main() -> int:
    # Folder path to save outputs
    output_folder = "output"
    logger = Logger()

    # Load dataset
    # SAMAN TEYMOURI PAMLP Report
    df = load_data("input/insurance.csv", logger=logger)

    # describe dataset characteristics
    describe_data(df, logger=logger)

    # Show data exploration
    plot_pair(df, "before_data_cleaning", output_folder=output_folder,logger=logger)

    # Plot before data cleaning
    plot_hist(df, "before_data_cleaning", output_folder, logger=logger)

    # Show line plot of features and target
    plot_line(df,output_folder=output_folder,logger=logger)
```

Figure 2: Part of the main function and all needed libraries

```
################### First 5 rows of original data ######################
   age     sex     bmi  children smoker     region      charges
0   19  female  27.900         0    yes  southwest  16884.92400
1   18    male  33.770         1     no  southeast   1725.55230
2   28    male  33.000         3     no  southeast   4449.46200
3   33    male  22.705         0     no  northwest  21984.47061
4   32    male  28.880         0     no  northwest   3866.85520
```

Figure 3: Output of load_data function

Afterward, it uses the info method of dataframe to get columns characteristics. It shows that the dataset has seven columns (four numeric and three categorical) (Figure 4). Choi (2017) explains that the dataset columns are:

- age: age of the primary beneficiary
- sex: insurance contractor gender (female, male)
- bmi: body mass index, providing an understanding of body
- children: number of children covered by health insurance/number of dependents
- smoker: smoking
- region: residential area in the US, northeast, southeast, southwest, northwest.
- charges: individual medical costs billed by health insurance

```
################### Describe original data specifications ###############
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       1338 non-null   int64
 1   sex       1338 non-null   object
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64
 4   smoker    1338 non-null   object
 5   region    1338 non-null   object
 6   charges   1338 non-null   float64
dtypes: float64(2), int64(2), object(3)
memory usage: 73.3+ KB
```

Figure 4: Info method output

It then uses the describe method of dataframe to extract measures of central tendency and dispersion for numeric variables (Figure 6).

```
                age          bmi     children       charges
count   1338.000000  1338.000000  1338.000000   1338.000000
mean      39.207025    30.663397     1.094918  13270.422265
std       14.049960     6.098187     1.205493  12110.011237
min       18.000000    15.960000     0.000000   1121.873900
25%       27.000000    26.296250     0.000000   4740.287150
50%       39.000000    30.400000     1.000000   9382.033000
75%       51.000000    34.693750     2.000000  16639.912515
max       64.000000    53.130000     5.000000  63770.428010
median    39.000000    30.400000     1.000000   9382.033000
mode      18.000000    32.300000     0.000000   1639.563100
```

Figure 5: Measures of central tendency and dispersion

The method not only gives the count, mean, standard deviation, and inter-quartile ranges of each column but also adds the median and mode of these columns to the dataframe built by describe method (Figure 5).

```python
def describe_data(data : pd.DataFrame, logger : Logger = None, suffix : str = "original"):
    # extract some descriptive analysis
    logger.log(f"\n#################### Describe {suffix} data specifications ##############")

    # show the rows and columns count plus columns data types
    info = io.StringIO()
    data.info(buf=info)
    logger.log(info.getvalue())

    # show statistic of the columns one by one
    describe = data.describe()

    # since median and mode are not included in describe method outputs, they are added manually to the output for numeric columns
    median = data.median(axis="index", numeric_only=True)
    numeric_mode = data.mode(numeric_only=True)
    for col in describe:
        describe.at["median",col] = median[col]
        describe.at["mode",col] = numeric_mode.loc[0,col]

    logger.log(describe)

    # for categorical columns only mode measure is available
    categorical_mode = data.mode().drop(columns=describe.columns)
    if not categorical_mode.empty:
        logger.log(f"\nModes of categorical variables are:\n{categorical_mode}")
        for col in categorical_mode:
            logger.log(f"\n{data.value_counts([col])}")

    logger.log("#########################################################################\n")
```

Figure 6: Describe function

It also identifies mode (Figure 7) and value count (Figure 8) for categorical variables.

```
Modes of categorical variables are:
    sex smoker     region
0  male     no  southeast
```

Figure 7: Mode of categorical variables

```
sex
male      676
female    662
Name: count, dtype: int64

smoker
no        1064
yes        274
Name: count, dtype: int64

region
southeast    364
northwest    325
southwest    325
northeast    324
Name: count, dtype: int64
```

Figure 8: Frequency of use of categorical variables

Based on the above information, it shows that the average age of contractors is 39 and most of them are about 18. Based on the equality of mean, mode, and median, BMI feature seems to have a normal distribution. The mean and mode of charges indicate that most of the charges are not high but there are a few very high charges. By investigating the categorical variables, some facts are revealed that the genders of contractors are equal, most of them are not smokers, and they are equally distributed in four regions.

For a better understanding of the relationship between columns, this paper provides some visualizations. The first one is pair plots that show the relationship of every two features and their relationship with the target variable (Figure 9).

```python
def plot_pair(data : pd.DataFrame, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    # show data exploration by plotting relationship between every two columns
    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)
    # data Exploration
    sns.pairplot(data)
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/data_exploration{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/data_exploration{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 9: Drawing pair plots

As it is shown in (Figure 10), before cleaning, the pair plots are available only for the four initial numeric variables but after that, they are available for all columns (Figure 11).

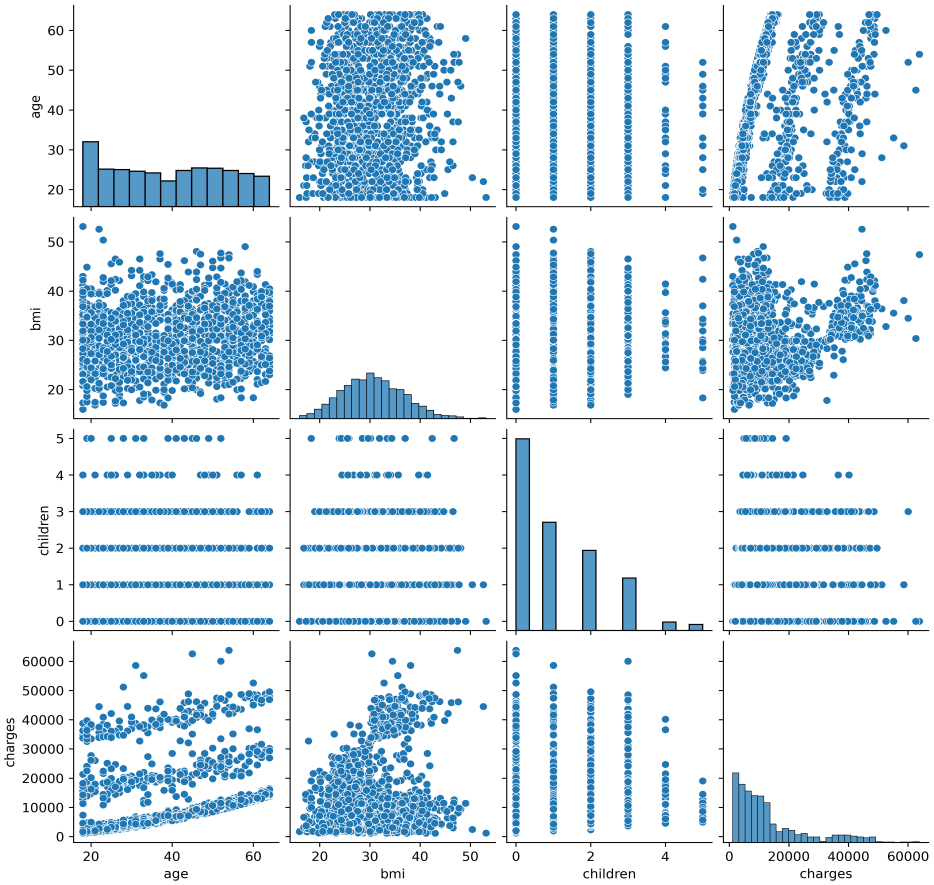

Figure 10: Pair plots before data cleaning

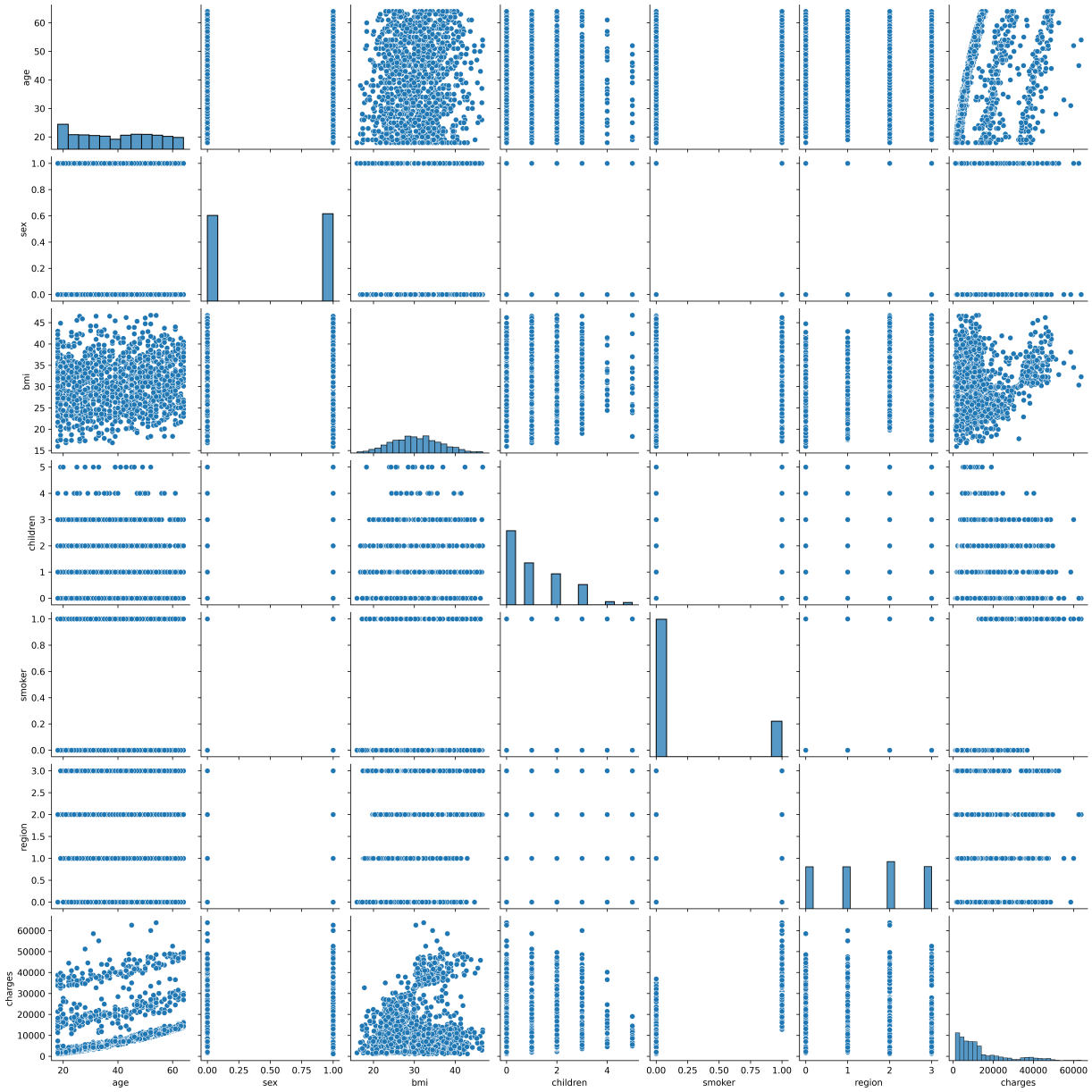

Figure 11: Pair plots after data cleaning

There is a lot of information to be extracted from these plots. For example, in different charge ranges, when the age of contractors increases the charges also increase linearly. Another example is regardless of age, most contractors have a BMI between 20 and 35 (Figure 10).

Afterward, the program creates histograms for features before (Figure 12) and after data cleaning (Figure 13).

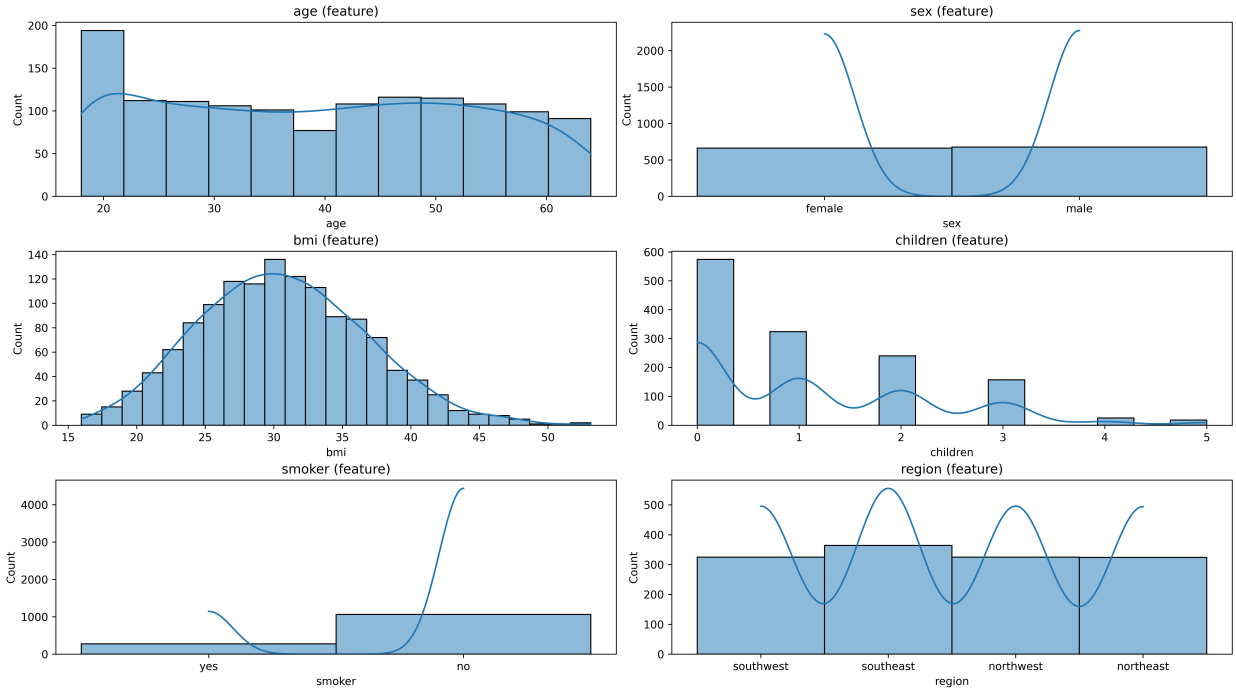

Figure 12: Histograms before data cleaning

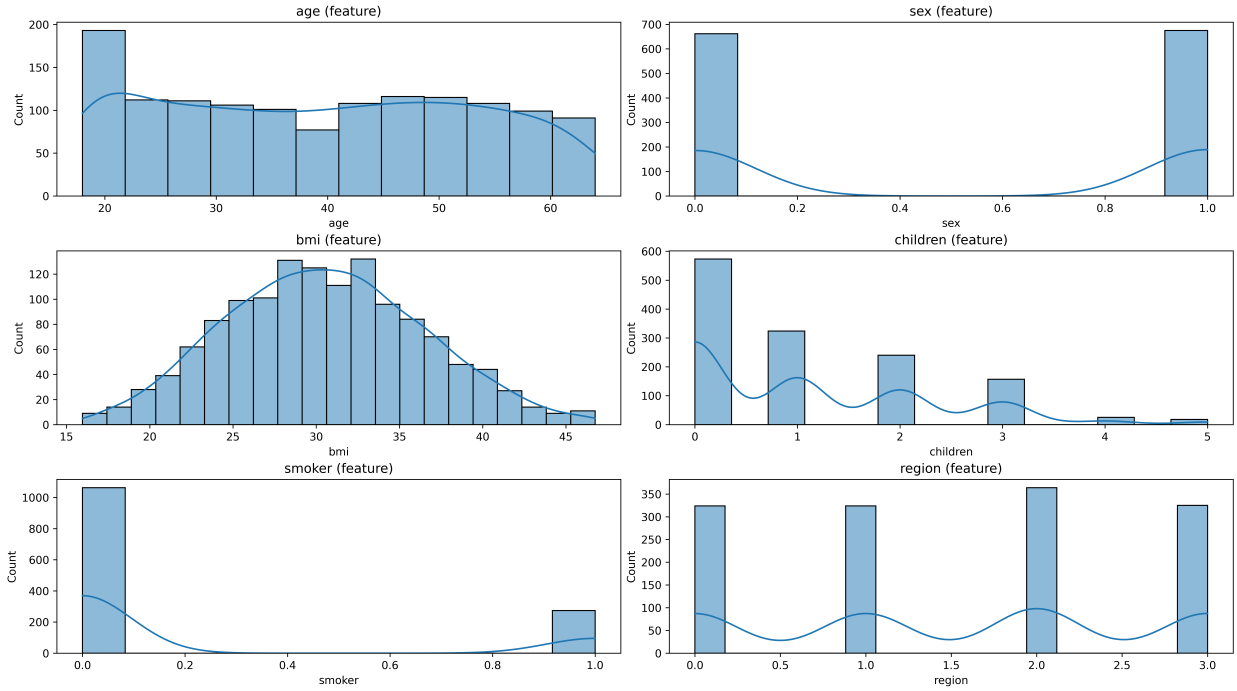

Figure 13: Histograms after data cleaning

Histograms are created by plot_hist function (Figure 14).

```python
def plot_hist(data : pd.DataFrame, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    # histogram of input and output of dataset (features and label)
    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)

    row_count = 3
    col_count = 2

    i = 1
    for col in data.drop(columns=target_col).columns:
        # Creating inputs subplots
        plt.subplot(row_count, col_count, i)
        sns.histplot(data[col], kde=True)
        plt.title(f"{col} (feature)")
        i += 1

    # setup the layout to fit in the figure
    plt.tight_layout(pad=1, h_pad=0.5, w_pad=0.5)
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/input_description{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/input_description{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 14: Creating histograms

Histograms depict some useful information such as the BMI histogram admits that it has a normal distribution. It also shows that a big portion of contractors are about 20, and the rest are equally distributed in different ages. These histograms also approved some facts about the value count of categorical variables that are mentioned in previous sections.

This study also calculates the correlation between columns of the dataset (Figure 15).

```python
def plot_correlation(data : pd.DataFrame, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    # Extract correlatoin between features themselves and also with label plus plot their heatmap
    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)
    correlations = data.corr()
    logger.log(f"correlations of features:\n{correlations}")
    sns.heatmap(correlations,annot=True)
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/correlations{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/correlations{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 15: Calculating correlation between dataset columns

The correlation report (Figure 16) shows no high correlation between features but the “smoker” and “age” columns have the highest correlation with the target variable “charges” respectively.

```
correlations of features:
               age       sex       bmi  children    smoker    region   charges
age       1.000000 -0.019814  0.114410  0.041536 -0.025587  0.001626  0.298308
sex      -0.019814  1.000000  0.040471  0.017848  0.076596  0.004936  0.058044
bmi       0.114410  0.040471  1.000000  0.016992 -0.002213  0.156201  0.192190
children  0.041536  0.017848  0.016992  1.000000  0.007331  0.016258  0.067389
smoker   -0.025587  0.076596 -0.002213  0.007331  1.000000 -0.002358  0.787234
region    0.001626  0.004936  0.156201  0.016258 -0.002358  1.000000 -0.006547
charges   0.298308  0.058044  0.192190  0.067389  0.787234 -0.006547  1.000000
```

Figure 16: Correlation between dataset columns

For a better understanding of correlations, there is the heat map of correlation below (Figure 17).

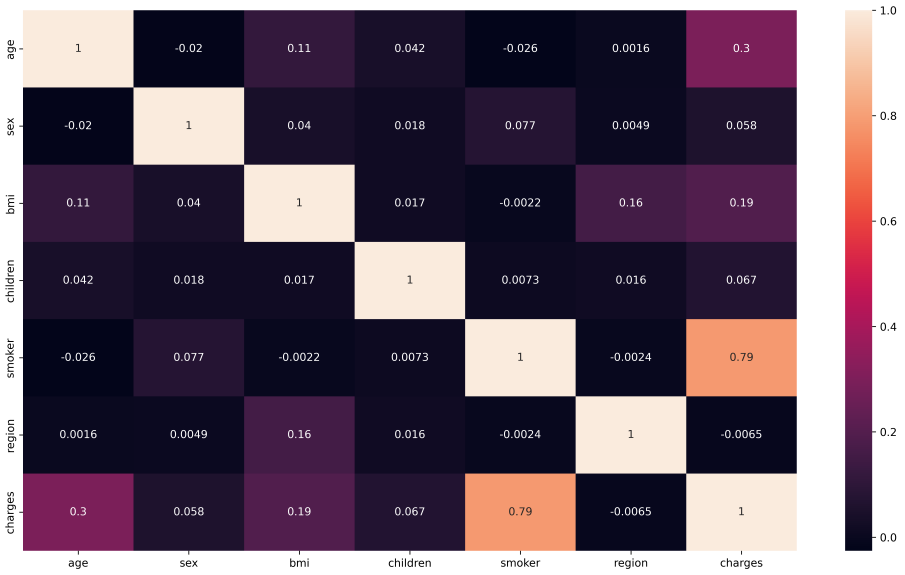

Figure 17: Heat map of correlation between columns

1.2 DATA CLEANING

This section describes data cleaning steps elaborately. At first, the program tries to find missing values (Figure 18). The results indicate that the dataset has no missing (na) values (Figure 19), so it is not necessary to handle anything about it.

```python
def handle_missing_values(data : pd.DataFrame, logger : Logger = None) -> pd.DataFrame:
    # examining data to discover missing values
    logger.log(f"Dataset has these missing values:\n{data.isna().sum()}\n")
    # Replace missing values via bfill method
    data = data.bfill()
    logger.log(f"Missing values are replaced by next valid value of that column using bfill method.\n")
    return data
```

Figure 18: Handling missing values

```
################### Data cleaning and dataset changes ###################
Dataset has these missing values:
age         0
sex         0
bmi         0
children    0
smoker      0
region      0
charges     0
dtype: int64

Missing values are replaced by next valid value of that column using bfill method.
```

Figure 19: Missing values of dataset

After managing missing values, this study checks the dataset to find duplicate samples (Figure 20).

```python
def handle_duplicate_values(data : pd.DataFrame, logger : Logger = None) -> pd.DataFrame:
    # exploriрng data to find duplicated rows
    logger.log(f"Duplicate rows are:\n{data.loc[data.duplicated()]}")
    logger.log(f"Duplicate rows count: {data.duplicated().sum()}")
    # eliminate duplicate rows from data
    data = data.drop_duplicates()
    # summary about data after removing duplicates
    logger.log(f"After removing duplicate rows the dataset has {data.shape[0]} rows.\n")
    return data
```

Figure 20: Handling duplicate values

The results show that there is only one duplicate row in the dataset (Figure 21). After removing that, the rows of the dataset are reduced to 1337.

```
Duplicate rows are:
     age   sex    bmi  children smoker     region    charges
581   19  male  30.59         0     no  northwest  1639.5631
Duplicate rows count: 1
After removing duplicate rows the dataset has 1337 rows.
```

Figure 21: Duplicate values of dataset

At this step, the program detects outliers. Magaga (2021) states that there are different ways to find the presence of outliers in the dataset such as the inter-quartile range method, skewness, and visualization. This paper uses both interquartile range (IQR) and skewness to find outliers (Figure 22). Visualizations by box plots also help in this process (Figure 25). The skewness value should be within the range of -1 to 1 for a normal distribution, any major changes from this value indicate the presence of an extreme value or outlier (Magaga, 2021). IQR method indicates that “bmi” column contains outliers, but skewness does not depict any out-of-range value (Figure 23). The program replaces them with the mode value of their column.

```python
def handle_outlier_values(data : pd.DataFrame, output_folder : str = "output", logger : Logger = None) -> pd.DataFrame:
    # Distinguish and remove the outliers based on IQR analysis
    describe = data.describe()
    try:
        describe = describe.drop(columns=target_col)
    except:
        pass
    # Box plot to show outliers
    plot_box(data=data, output_folder=output_folder, logger=logger)

    # Detecting outliers with IQR method
    logger.log("Outliers based on IQR method and skewness before handling them:\n")

    outliers_index = {}
    for col in describe.columns:
        # acquiring q1 and q3 to establish allowed area
        q1 = describe.loc["25%", col]
        q3 = describe.loc["75%", col]
        iqr = q3 - q1
        # every value outside of allowed area is distiguished as an outlier
        outliers = data[col].loc[(data[col] > q3 + 1.5*iqr) | (data[col] < q1 - 1.5*iqr)]
        logger.log(f"Outliers of Column {col} count: {outliers.count()}  ---  skewness: {data[col].skew()}")
        # outlier index of each column
        if not outliers.empty:
            outliers_index[col] = outliers.index

    # Replace outliers with mode of that columns
    for col in outliers_index.keys():
        data.loc[outliers_index[col],col] = data.mode(numeric_only=True).loc[0,col]

    # Detecting outliers with IQR method after handling them
    logger.log("\nOutliers based on IQR method and skewness after handling them:\n")

    outliers_index = {}
    for col in describe.columns:
        # acquiring q1 and q3 to establish allowed area
        q1 = describe.loc["25%", col]
        q3 = describe.loc["75%", col]
        iqr = q3 - q1
        # every value outside of allowed area is distiguished as an outlier
        outliers = data[col].loc[(data[col] > q3 + 1.5*iqr) | (data[col] < q1 - 1.5*iqr)]
        logger.log(f"Outliers of Column {col} count: {outliers.count()}  ---  skewness: {data[col].skew()}")
        # outlier index of each column
        if not outliers.empty:
            outliers_index[col] = outliers.index

    return data
```

Figure 22: Handling outliers

```
Outliers based on IQR method and skewness before handling them:

Outliers of Column age count: 0  ---  skewness: 0.054780773126998195
Outliers of Column bmi count: 9  ---  skewness: 0.28391419385321137
Outliers of Column children count: 0  ---  skewness: 0.9374206440474123
```

Figure 23: Outlier values of dataset

After replacing outliers with mode values, the results show that the skewness of “bmi” column reduces (Figure 24).

```
Outliers based on IQR method and skewness after handling them:

Outliers of Column age count: 0  ---  skewness: 0.054780773126998195
Outliers of Column bmi count: 0  ---  skewness: 0.15167131342427534
Outliers of Column children count: 0  ---  skewness: 0.9374206440474123
```

Figure 24: Skewness after handling outliers

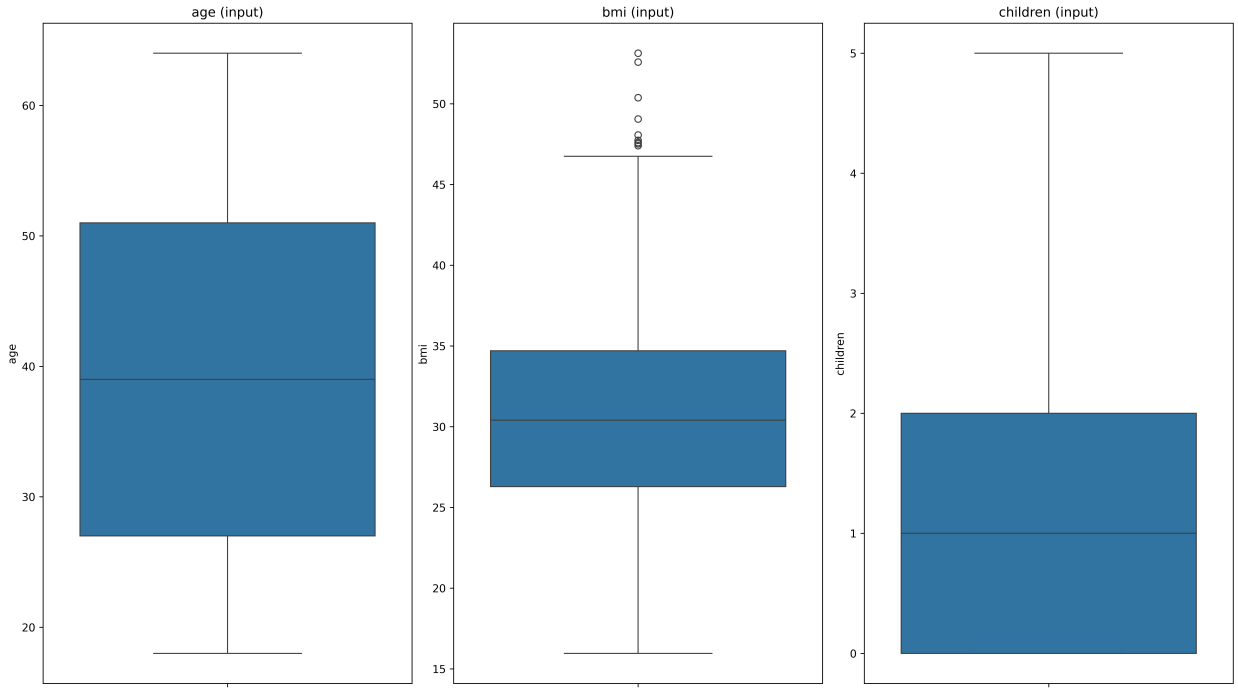

Figure 25: Box plots to investigate for outliers

After handling missing values, duplicates, and outliers, the program encodes categorical variables (Figure 26). For this, it uses the LabelEncoder class that assigns numbers from zero to categorical variables.

```
def encode_data(data : pd.DataFrame, logger : Logger) -> pd.DataFrame:
    # Encode categorical variables to numeric
    # Replace encoded variable using Label Encoding

    logger.log(f"\nBelow columns need encoding:")

    le = LabelEncoder()
    # for each columns in the dataset
    for col in data.columns:
        # if the column is categorical
        if data[col].dtype == "object":
            # encode the data into numeric
            data[col] = le.fit_transform(data[col])
            #unique values of encoded column
            logger.log(f"Values for {col} are {le.classes_}")

    return data
```

Figure 26: Encoding categorical variables

The three encoded variables and their different classes are shown below (Figure 27).

```
Below columns need encoding:
Values for sex are ['female' 'male']
Values for smoker are ['no' 'yes']
Values for region are ['northeast' 'northwest' 'southeast' 'southwest']
```

Figure 27: Categorical variables with their different classes

The last step in data cleaning is normalization and scaling. This paper uses minmax_scale to make a uniform view of features (Figure 28). It helps understand features better without any changes in the shape of data.

```python
def scale_data(data : pd.DataFrame, logger : Logger = None) -> pd.DataFrame:
    # Scale features values
    # Scale all columns except target column
    scaling_cols = list(data.drop(target_col,axis="columns").columns)

    # Extract columns which need Scaling
    need_scaling = data[scaling_cols]

    # Scale data
    ndarr = minmax_scale(need_scaling, axis=0) #axis=0 means column-wise

    scaled_data = pd.DataFrame()
    # Build dataframe based on the ndarray and original data
    scaled_data.index = data.index
    for col in data.columns:
        # all features values come from ndarr and the target values come from original data
        if col in scaling_cols:
            scaled_data[col] = ndarr[:,scaling_cols.index(col)]
        else:
            scaled_data[col] = data.loc[:,col]

    # summary about data after encoding and scaling
    logger.log(f"\nAfter encoding and Scaling dataset is:")
    logger.log(scaled_data)

    return scaled_data
```

Figure 28: Scaling the data

After scaling, the dataset is like below (Figure 29).

```
After encoding and Scaling dataset is:
           age  sex       bmi  children  smoker    region      charges
0     0.021739  0.0  0.387788       0.0     1.0  1.000000  16884.92400
1     0.000000  1.0  0.578435       0.2     0.0  0.666667   1725.55230
2     0.217391  1.0  0.553426       0.6     0.0  0.666667   4449.46200
3     0.326087  1.0  0.219065       0.0     0.0  0.333333  21984.47061
4     0.304348  1.0  0.419617       0.0     0.0  0.333333   3866.85520
...        ...  ...       ...       ...     ...       ...          ...
1333  0.695652  1.0  0.487496       0.6     0.0  0.333333  10600.54830
1334  0.000000  0.0  0.518350       0.0     0.0  0.000000   2205.98080
1335  0.000000  0.0  0.678467       0.0     0.0  0.666667   1629.83350
1336  0.065217  0.0  0.319584       0.0     0.0  1.000000   2007.94500
1337  0.934783  0.0  0.425788       0.0     1.0  0.333333  29141.36030

[1337 rows x 7 columns]
```

Figure 29: Dataset values after data cleaning finished

The output of the describe function after cleaning (except the scaling step) shows no categorical variable in the dataset (Figure 30).

```
                age          sex          bmi     children       smoker       region       charges
count   1337.000000  1337.000000  1337.000000  1337.000000  1337.000000  1337.000000   1337.000000
mean      39.222139     0.504862    30.549174     1.095737     0.204936     1.516081  13279.121487
std       14.044333     0.500163     5.903956     1.205571     0.403806     1.105208  12110.359656
min       18.000000     0.000000    15.960000     0.000000     0.000000     0.000000   1121.873900
25%       27.000000     0.000000    26.290000     0.000000     0.000000     1.000000   4746.344000
50%       39.000000     1.000000    30.400000     1.000000     0.000000     2.000000   9386.161300
75%       51.000000     1.000000    34.430000     2.000000     0.000000     2.000000  16657.717450
max       64.000000     1.000000    46.750000     5.000000     1.000000     3.000000  63770.428010
median    39.000000     1.000000    30.400000     1.000000     0.000000     2.000000   9386.161300
mode      18.000000     1.000000    32.300000     0.000000     0.000000     2.000000   1121.873900
```

Figure 30: Dataset description after cleaning (except scaling step)

At this stage, the dataset is completely cleaned and prepared. It is the time to select the appropriate models and train them based on the dataset in the next task.

TASK 2: MODEL SELECTION AND TRAINING

This paper addresses a regression problem (predicting a continuous numeric variable), so it selects five algorithms including linear regression, ridge regression, decision tree regression, random forest regression, and XGBoost regression.

The first model is linear regression (Figure 31). The linear regression model aims to find a linear function to predict the output based on the inputs. The impact of each input on the output is defined by some parameters called regression coefficients. When we train the model and discover the coefficients, we can predict the output by simply assigning every input to its place in the equation (Miller and Forte, 2017).

```python
def linear_regression(data : pd.DataFrame, train_test_sets : list, grid_best_params : dict = {}, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    logger.log(f"################ regression by linear regression {"(After Optimization)" if grid_best_params else "(Before Optimization)"} ################\n")
    # Build a regression model to predict the test set

    # grid best params is not empty when we are train model after the optimization
    if grid_best_params:
        reg = LinearRegression(copy_X=grid_best_params["copy_X"], fit_intercept=grid_best_params["fit_intercept"], n_jobs=grid_best_params["n_jobs"], positive=grid_best_param
    else:
        reg = LinearRegression()

    # Extract train and test datasets
    X_train, X_test, y_train, y_test = train_test_sets

    # train the model
    reg.fit(X_train, y_train)

    # Predict test dataset via model
    y_pred = reg.predict(X_test)

    # Evaluate the model
    evaluate_linear_regression(reg=reg, data=data, X_train=X_train, y_train=y_train, y_test=y_test, y_pred=y_pred, suffix="After Optimization" if grid_best_params else "Befor
```

Figure 31: Linear regression model

The first step is splitting the dataset into train and test sets. These are done using the extract_train_test function (Figure 32).

```python
def extract_train_test(data : pd.DataFrame, logger : Logger = None) -> list:
    # Seperate features and labels
    X = data.drop(columns=target_col)
    y = data.loc[:, target_col]

    # split train and test datasets
    train_test_sets = train_test_split(X, y, test_size=0.2, random_state=42)

    return train_test_sets
```

Figure 32: Splitting dataset into train and test sets

There are various methods of splitting datasets for machine learning models. The right approach for data splitting and the optimal split ratio both depend on several factors, including the use case, amount of data, quality of data, and the number of hyperparameters (Buhl, 2023).

Buhl (2023) highlighted that the most common approach for dividing a dataset is random sampling. As the name suggests, the method involves shuffling the dataset and randomly assigning samples to train or test sets according to predefined ratios.

This paper uses the random sampling method to split the dataset into train and test sets and allocates 0.2 for the test set size. The extract_train_test function is called once and used for all procedures.

The linear regression uses default parameters at first (“copy_X”: True, “fit_intercept”: True, “n_jobs”: None, “positive”: False). In task 3 of this paper, these parameters get optimized. There

are also some pieces of code for model evaluation and hyperparameter tuning which are covered in the further task.

The second model is ridge regression (Figure 33). Ridge regression is useful when the number of parameters is very high or independent variables are highly correlated. This algorithm tries to minimize the least square. In comparison with linear regression, it reduces the variance and helps the predicted results to be closer to the true values (Miller and Forte, 2017). Although there is no significant correlation between inputs in the dataset, this study applies ridge regression to examine its differences in outcomes with linear regression.

```
def ridge_regression(data : pd.DataFrame, train_test_sets : list, grid_best_params : dict = {}, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    logger.log(f"################ regression by ridge regression {"(After Optimization)" if grid_best_params else "(Before Optimization)"} ################\n")
    # Build a regression model to predict the test set

    # grid_best_params is not empty when we are train model after the optimization
    if grid_best_params:
        reg = Ridge(alpha = grid_best_params["alpha"], copy_X=grid_best_params["copy_X"], fit_intercept=grid_best_params["fit_intercept"], positive=grid_best_params["positive"]
    else:
        reg = Ridge()

    # Extract train and test datasets
    X_train, X_test, y_train, y_test = train_test_sets

    # train the model
    reg.fit(X_train, y_train)

    # Predict test dataset via model
    y_pred = reg.predict(X_test)

    # Evaluate the model
    evaluate_ridge_regression(reg=reg, data=data, X_train=X_train, y_train=y_train, y_test=y_test, y_pred=y_pred, suffix="After Optimization" if grid_best_params else "Before O
```

Figure 33: Ridge regression model

The ridge regression uses default parameters at first (“alpha”: 1, “copy_X”: True, “fit_intercept”: True, “n_jobs”: None, “positive”: False, “solver”: auto, “random_state”: None). It passes the same steps as linear regression such as splitting train and test sets, training the model, and predicting the test set for evaluation.

The Third model is decision tree (Figure 34). As the name suggests, decision tree is a learning algorithm that applies a sequential series of decisions based on input information to make the final prediction (Johnston and Mathur, 2019). Bonaccorso (2018) describes that building a decision tree starts from the root as the most important decision. Then, the algorithm chooses two children with the most information gain. This process continues until the impurity becomes minimum.

```python
def decisiontree_regression(data : pd.DataFrame, train_test_sets : list, grid_best_params : dict = {}, suffix : str = "", output_folder : str = "output", logger : Logger = None
    logger.log(f"################ regression by decision tree  {"(After Optimization)" if grid_best_params else "(Before Optimization)"} ################\n")
    # Build a regression model to predict the test set

    # grid_best_params is not empty when we are train model after the optimization
    if grid_best_params:
        reg = DecisionTreeRegressor(max_depth=grid_best_params["max_depth"], min_samples_leaf=grid_best_params["min_samples_leaf"], min_samples_split=grid_best_params["min_samp
    else:
        reg = DecisionTreeRegressor()

    # Extract train and test datasets
    X_train, X_test, y_train, y_test = train_test_sets

    # train the model
    reg.fit(X_train, y_train)

    # Predict test dataset via model
    y_pred = reg.predict(X_test)

    # Exporting decesion tree to png file
    export_tree(reg=reg, feature_names_in_=reg.feature_names_in_, suffix=suffix, output_folder=output_folder, logger=logger)

    # Evaluate the model
    evaluate_decisiontree_regression(reg=reg, data=data, X_train=X_train, y_train=y_train, y_test=y_test, y_pred=y_pred, suffix="After Optimization" if grid_best_params else "B
```

Figure 34: Regression by decision tree

Like the others, the first step in this model is to split the dataset to train and test sets. The next step is training or fitting the model. Afterward, the model is prepared to predict the test test.

The decision tree regressor uses default parameters at first ("max_depth": None, "min_samples_leaf": 1, "min_samples_split": 2, "random_state": None, "max_features" : None). In task 3 of this paper, these parameters get optimized.

The visualization of the created decision tree is provided below (Figure 35). Although regarding the resolution issues, the picture is not very clear, it may be useful to see the decision tree visually.

Figure 35: The whole decision tree

To have a better visual, part of the decision tree which contains the root node is shown below (Figure 36).

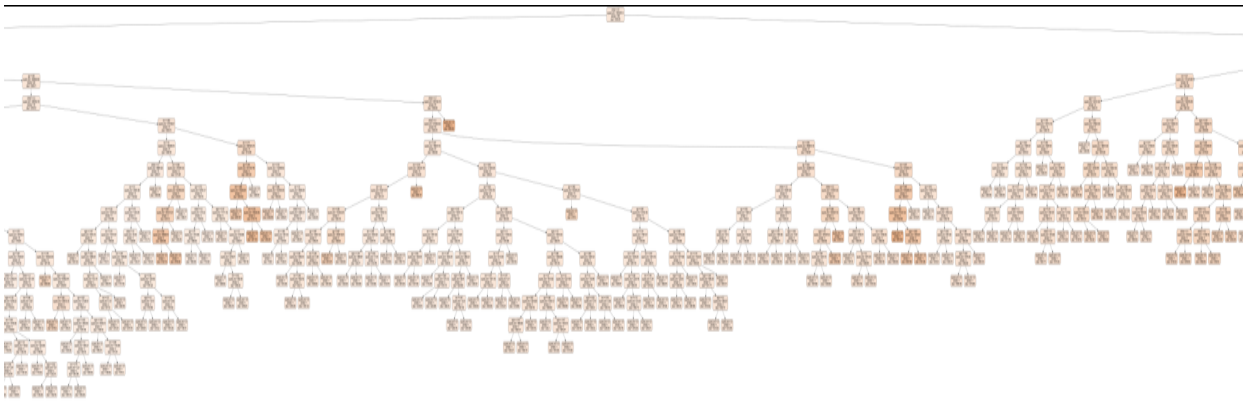

Figure 36: Part of the decision tree that contains root node

The visualization is performed by the export_tree function (Figure 37) using Graphviz utilities (Graphviz, 2021).

```python
def export_tree(reg : DecisionTreeRegressor, feature_names_in_ : np.ndarray, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    # export decision tree into dot file notaion
    dot_data = export_graphviz(reg, out_file =None ,feature_names=feature_names_in_, impurity=True, filled=True, rounded=True)
    # create png file based on dot data
    graph = graphviz.Source(dot_data, format="png")
    # render and save png file
    graph.render(filename=f"{output_folder}/decision_tree{('_' if suffix else '') + suffix}", view=False)
    logger.log(f"#################### {output_folder}/decision_tree{('_' if suffix else '') + suffix}.png file saved ##############################\n")
```

Figure 37: Decision tree visualization

The fourth algorithm used in this study is random forest (Figure 38). Johnston and Mathur (2019) explain that the common issue with decision trees is that the split on each node is performed using a greedy algorithm that minimizes the entropy of the leaf nodes. Random forest algorithm maintains randomness by using a subset of features in each decision tree. This ensures that the predictions from each tree in the forest have a lower probability of being correlated to the predictions from other trees (Johnston and Mathur, 2019).

```python
def randomforest_regression(data : pd.DataFrame, train_test_sets : list, grid_best_params : dict = {}, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    logger.log(f"################ regression by random forest {"(After Optimization)" if grid_best_params else "(Before Optimization)"} ################\n")
    # Build a regression model to predict the test set

    # grid best params is not empty when we are train model after the optimization
    if grid_best_params:
        reg = RandomForestRegressor(max_depth=grid_best_params["max_depth"], min_samples_leaf=grid_best_params["min_samples_leaf"], min_samples_split=grid_best_params["min_sample
    else:
        reg = RandomForestRegressor()

    # Extract train and test datasets
    X_train, X_test, y_train, y_test = train_test_sets

    # train the models
    reg.fit(X_train, y_train)

    # Predict test dataset via model
    y_pred = reg.predict(X_test)

    # Exporting decesion trees of the random forest to png file
    est_count = 1
    for estimator in reg.estimators_:
        export_tree(reg=estimator, feature_names_in_=reg.feature_names_in_, suffix=suffix + "_" + str(est_count), output_folder=output_folder, logger=logger)
        est_count += 1
        if est_count > 5:
            break

    # Evaluate the model
    evaluate_randomforest_regression(reg=reg, data=data, X_train=X_train, y_train=y_train, y_test=y_test, y_pred=y_pred, suffix="After Optimization" if grid_best_params else "Bef
```

Figure 38: Regression by random forest

The default parameters of the random forest regressor are the same as decision tree with one difference random forest does not accept “None” for the "max_features" parameter and the default value is “sqrt”.

Random forest regressor by default creates 100 decision trees for regression but this paper shows only the first five of them below (Figure 39).

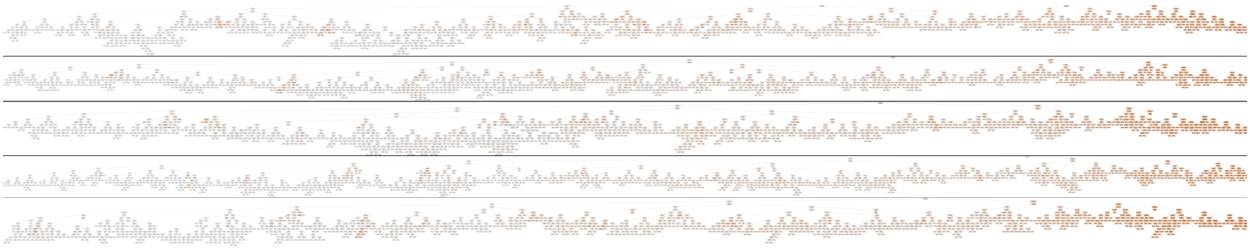

Figure 39: First five decision trees created by random forest algorithm

The last algorithm is XGBoost regression (Figure 40). Extreme Gradient Boosting (XGBoost) (Chen and Guestrin, 2016) is a library of functions designed and optimized specifically for boosting trees algorithms. XGBoost performs a sequential process of boosting by utilizing all cores of a machine. It applies internal cross-validation and uses these parts to train the model repeatedly. In addition, XGBoost has many customizable parameters that make the model so extendable and flexible. Other advantages offered by XGBoost include regularization, customizable parameters, deeper tree pruning, and built-in cross-validation (Miller and Forte, 2017).

```python
def xgboost_regression(data : pd.DataFrame, train_test_sets : list, grid_best_params : dict = {}, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    logger.log(f"################ regression by xgboost {"(After Optimization)" if grid_best_params else "(Before Optimization)"} ################\n")
    # Build a regression model to predict the test set

    # grid best params is not empty when we are train model after the optimization
    if grid_best_params:
        reg = XGBRegressor(max_depth=grid_best_params["max_depth"], learning_rate=grid_best_params["learning_rate"], subsample=grid_best_params["subsample"], min_child_weight=gr
    else:
        reg = XGBRegressor()

    # Extract train and test datasets
    X_train, X_test, y_train, y_test = train_test_sets

    # train the model
    reg.fit(X_train, y_train)

    # Predict test dataset via model
    y_pred = reg.predict(X_test)

    # Evaluate the model
    evaluate_xgboost_regression(reg=reg, data=data, X_train=X_train, y_train=y_train, y_test=y_test, y_pred=y_pred, suffix="After Optimization" if grid_best_params else "Before
```

Figure 40: XGBoost regression model

The XGBoost regression uses default parameters at first ("max_depth": 6, "learning_rate": 0.3, "subsample": 1, "min_child_weight": 1, "reg_alpha": 0, "reg_lambda": 0, "gamma": 0). It passes the same steps as other algorithms such as splitting train and test sets, training the model, and predicting the test set for evaluation.

TASK 3: MODEL INTERPRETATION AND EVALUATION

3.1 MODEL INTERPRETATION

This task is about extracting the feature importance of models and interpreting them. The first model is linear regression. The coefficients of the model are calculated (Figure 41). These values (Figure 42) show how every feature plays its role in the regression formula.

```
# set the figure resolution and dpi
fig = plt.figure(figsize=(16, 9), dpi=600)
# Extract model coefficients and plot their barh
# Creating a dataframe based on the  coefficients
coef = pd.DataFrame(reg.coef_,index=reg.feature_names_in_, columns=["coefficients"])
coef = coef.sort_values("coefficients",axis="index")
logger.log(f"Coefficients:\n{coef}")
plt.clf()
# Create a bar plot to compare feature importances visually
plt.barh(coef.index, coef.iloc[:,0])
plt.xlabel("coefficients")
# Save the file with proper dpi
plt.savefig(fname=f"{output_folder}/coefficients{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
logger.log(f"################### {output_folder}/coefficients{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 41: Calculating the coefficients of linear regression model

```
Coefficients:
          coefficients
region      -709.777289
sex          -44.035384
children    2624.688205
bmi        10049.367047
age        11381.701265
smoker     23047.195429
```

Figure 42: Coefficients of linear regression model before and after optimization

The results show that “smoker”, “age”, and “bmi” have the most impacts on the target variable. The results are also provided graphically in a bar plot below (Figure 43).

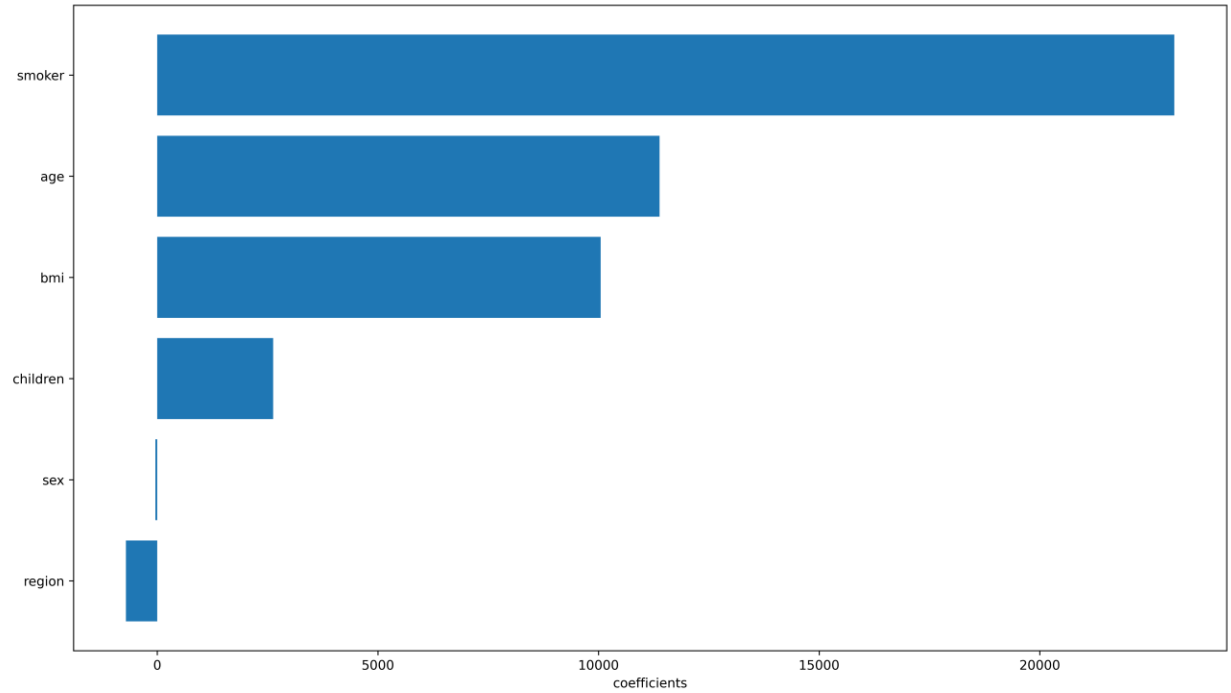

Figure 43: Bar plot of coefficients of linear regression model before and after optimization

Although this study applies optimization to the algorithms, the linear regression does not show any improvement and the results after optimization are identical to the before ones.

The second algorithm is ridge regression which has coefficients to evaluate the same as linear regression (Figure 44).

```
# set the figure resolution and dpi
fig = plt.figure(figsize=(16, 9), dpi=600)
# Extract model coefficients and plot their barh
# Creating a dataframe based on the  coefficients
coef = pd.DataFrame(reg.coef_,index=reg.feature_names_in_, columns=["coefficients"])
coef = coef.sort_values("coefficients",axis="index")
logger.log(f"Coefficients:\n{coef}")
plt.clf()
# Create a bar plot to compare feature importances visually
plt.barh(coef.index, coef.iloc[:,0])
plt.xlabel("coefficients")
# Save the file with proper dpi
plt.savefig(fname=f"{output_folder}/coefficients{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
logger.log(f"#################### {output_folder}/coefficients{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 44: Calculating the coefficients of ridge regression model

The results of coefficients are the same as linear regression and “smoker”, “age”, and “bmi” features have the most impact on the target variable respectively (Figure 45).

```
Coefficients:
          coefficients
region     -685.719609
sex         -33.501647
children   2595.848192
bmi        9803.997410
age       11282.375029
smoker    22907.315692
```

Figure 45: Coefficients of ridge regression model before optimization

The bar plot of coefficients is also available below (Figure 46).

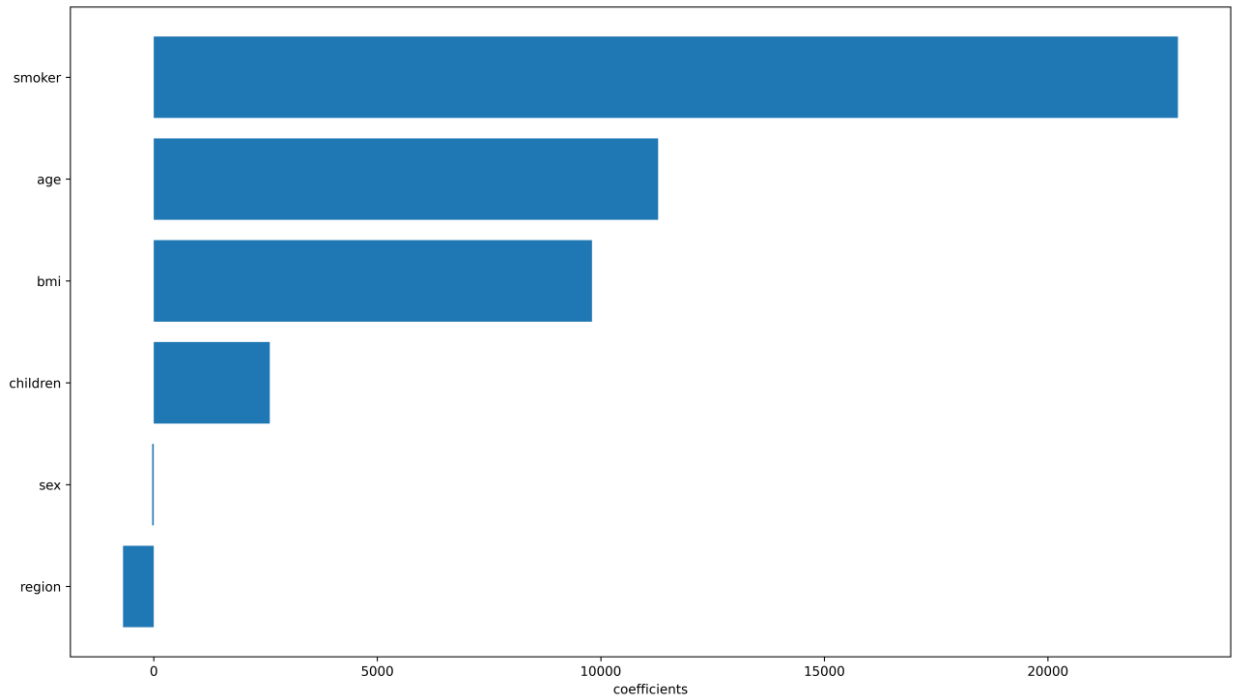

Figure 46: Bar plot of coefficients of ridge regression model before optimization

The coefficients are changed slightly after optimization but the order remains the same as before (Figure 47). Regarding to the very small changes, the bar plot after optimization would be excessive and unnecessary.

```
Coefficients:
          coefficients
region     -686.002048
sex         -33.755921
children   2597.235219
bmi        9804.465896
age       11281.410496
smoker    22907.674156
```

Figure 47: Coefficients of ridge regression model after optimization

The third model is decision tree and the feature importance is evaluated after training the model (Figure 48).

```
    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)
    # Extract model feature_importances and plot their barh
    # Creating a dataframe based on the feature importances
    feature_importances = pd.DataFrame(reg.feature_importances_,index=reg.feature_names_in_, columns=["feature_importances"])
    feature_importances = feature_importances.sort_values("feature_importances",axis="index")
    logger.log(f"Importance of features:\n{feature_importances}")
    plt.clf()
    # Create a bar plot to compare feature importances visually
    plt.barh(feature_importances.index, feature_importances.iloc[:,0])
    plt.xlabel("feature_importances")
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/feature_importances{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/feature_importances{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 48: Calculating the feature importance of decision tree model

The results of feature importance are shown below (Figure 49).

```
Importance of features:
          feature_importances
sex                  0.008668
region               0.020514
children             0.023610
age                  0.131150
bmi                  0.216779
smoker               0.599279
```

Figure 49: Feature importance of decision tree model before optimization

The results are sorted in ascending, and they show that the “smoker”, “bmi”, and “age” features are the most important ones.

```
Importance of features:
          feature_importances
sex                  0.003595
region               0.011928
children             0.020421
age                  0.148759
bmi                  0.164821
smoker               0.650475
```

Figure 50: Feature importance of decision tree model after optimization

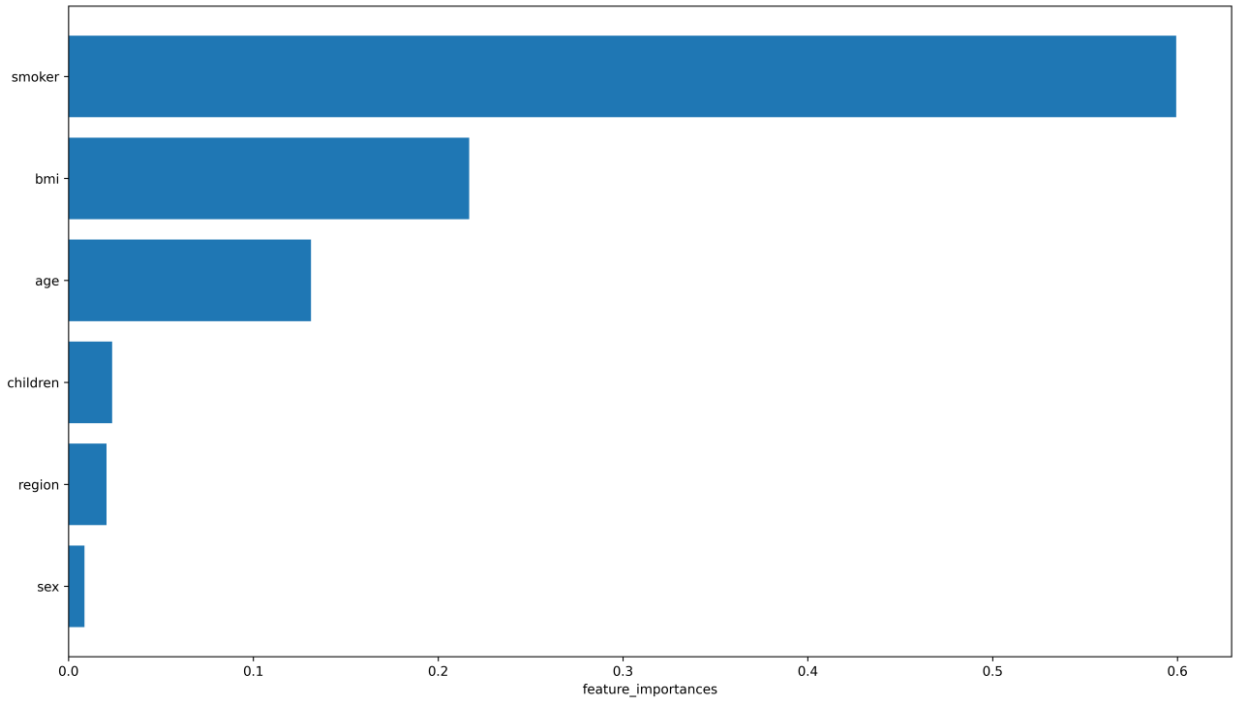

Figure 51: Bar plot of feature importance of decision tree model before optimization

As this study performs optimization on models, the results of feature importance after optimization are also available (Figure 50). The bar plot of these results before (Figure 51) and after (Figure 52) optimization are also provided. The comparison of results shows that the importance of the “smoker” feature increases and that can be the reason for better evaluation metrics after optimization.

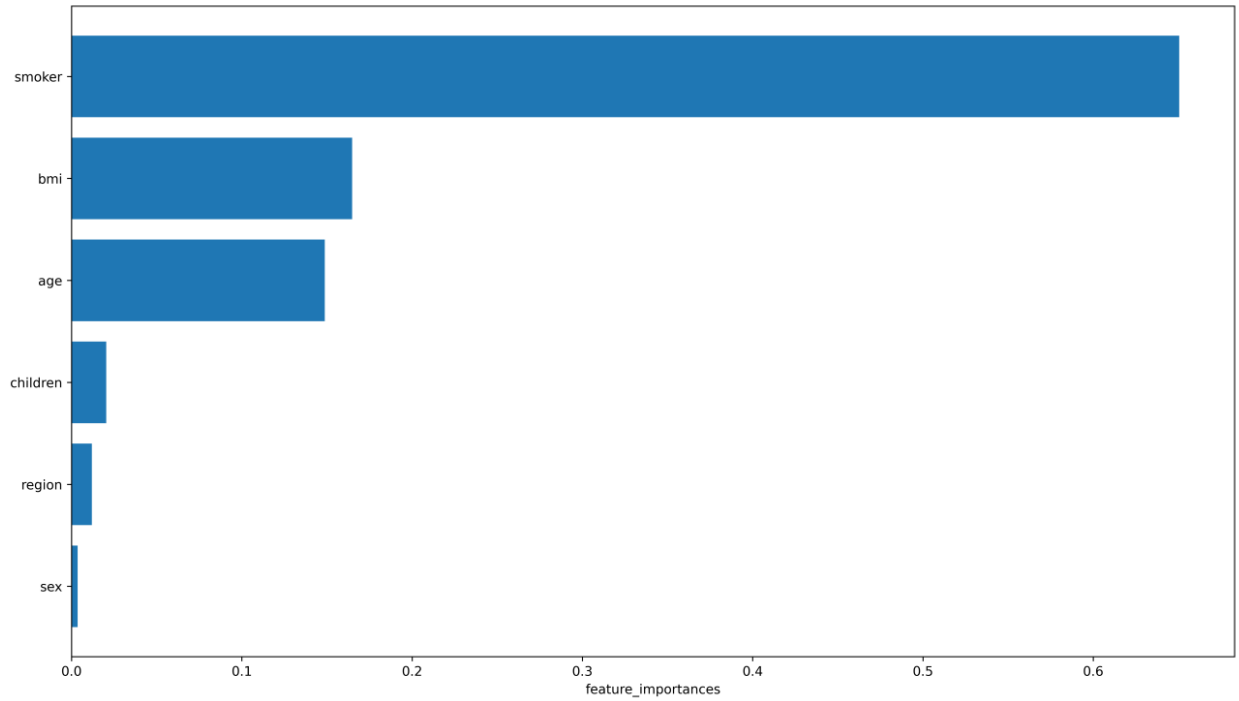

Figure 52: Bar plot of feature importance of decision tree model after optimization

The fourth algorithm is random forest which is very similar to the decision tree (Figure 53). As expected, the results of feature importance are close to the decision tree results (Figure 54). The "smoker" feature has the first place, and after that "bmi", and "age" come respectively.

```
# set the figure resolution and dpi
fig = plt.figure(figsize=(16, 9), dpi=600)
# Extract model feature_importances and plot their barh
# Creating a dataframe based on the feature importances
feature_importances = pd.DataFrame(reg.feature_importances_,index=reg.feature_names_in_, columns=["feature_importances"])
feature_importances = feature_importances.sort_values("feature_importances",axis="index")
logger.log(f"Importance of features:\n{feature_importances}")
plt.clf()
# Create a bar plot to compare feature importances visually
plt.barh(feature_importances.index, feature_importances.iloc[:,0])
plt.xlabel("feature_importances")
# Save the file with proper dpi
plt.savefig(fname=f"{output_folder}/feature_importances{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
logger.log(f"#################### {output_folder}/feature_importances{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 53: Calculating the feature importance of random forest model

```
Importance of features:
          feature_importances
sex                  0.008054
region               0.016681
children             0.022732
age                  0.139004
bmi                  0.214115
smoker               0.599414
```

Figure 54: Feature importance of random forest model before optimization

After optimization of random forest model, the order of feature importance does not change but the effect of the “smoker” feature increases (Figure 55). The bar plot of the feature importance of random forest model before (Figure 56) and after (Figure 57) optimization are shown below.

```
Importance of features:
         feature_importances
sex                 0.009112
region              0.020399
children            0.027699
age                 0.153674
bmi                 0.156577
smoker              0.632538
```

Figure 55: Feature importance of random forest model after optimization

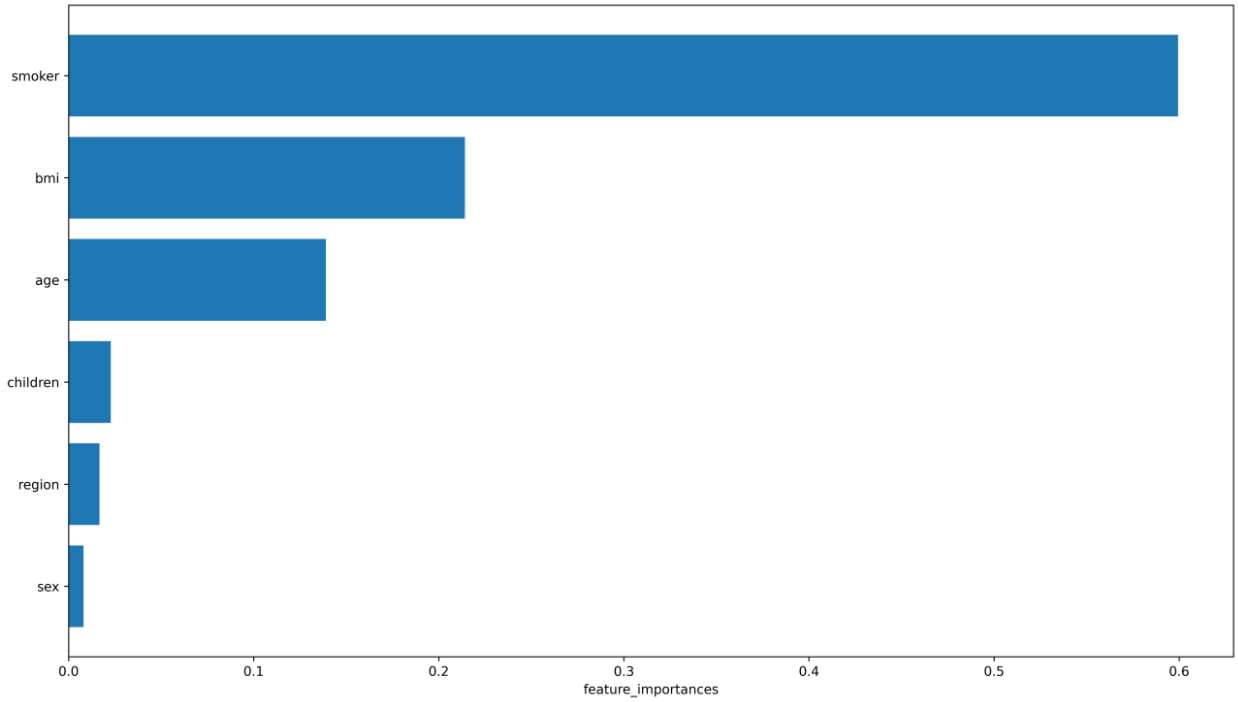

Figure 56: Bar plot of feature importance of random forest model before optimization

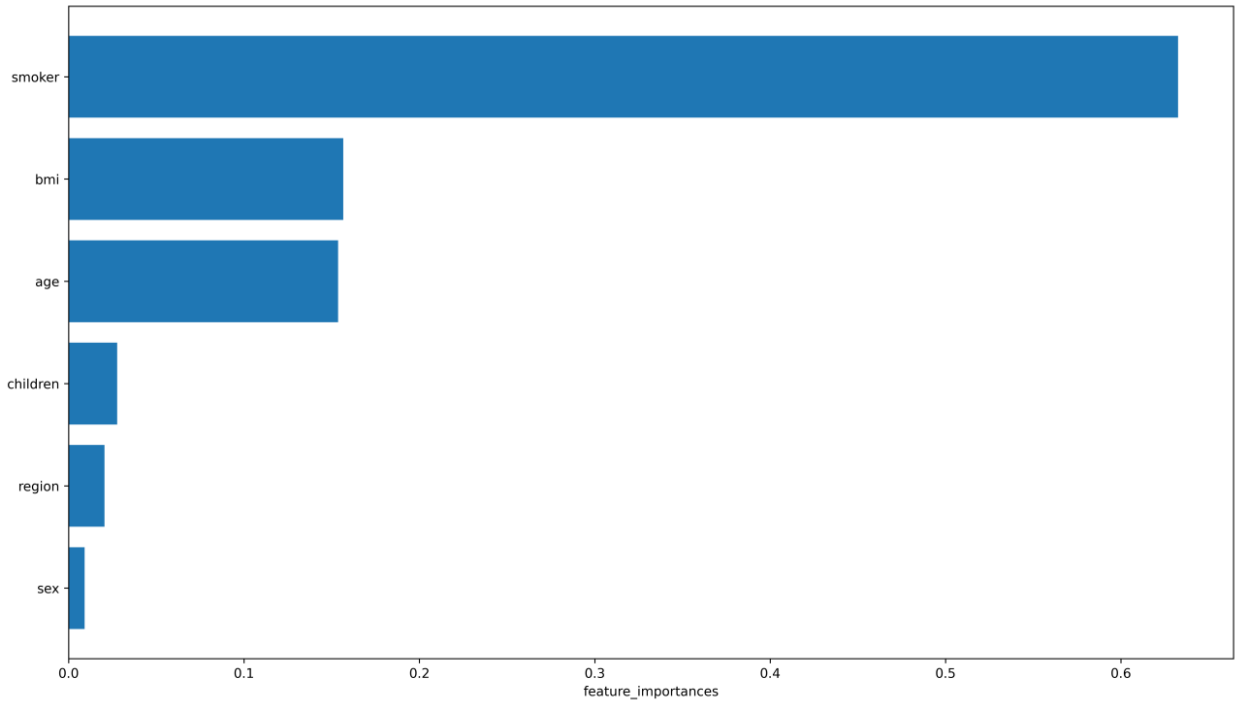

Figure 57: Bar plot of feature importance of random forest model after optimization

The last algorithm is XGBoost which unfortunately does not offer any coefficient or feature importance to interpret.

3.2 MODEL EVALUATION

This paper talks about the model evaluation and hyperparameter tuning in this section. For each algorithm, the results of evaluation metrics such as mean absolute error, mean squared error, and r2-score are stated in the first step. After hyperparameter tuning, the updated results are shown for comparison.

```python
def evaluate_linear_regression(reg : LinearRegression, data : pd.DataFrame, X_train : pd.DataFrame, y_train : pd.DataFrame, y_test : pd.DataFrame, y_pred : pd.DataFrame, suffix
    # Evaluating model
    logger.log(f"################ Evaluating the model{"(" + suffix + ")" if suffix else ''} ################")

    # Extract mean_absolute_error
    mea = mean_absolute_error(y_test, y_pred)
    logger.log(f"mean absolute error: {mea}\n")

    # Extract mean_squared_error
    mse = mean_squared_error(y_test, y_pred)
    logger.log(f"mean squared error: {mse}\n")

    # Extract r2_score
    r2score = r2_score(y_test, y_pred)
    logger.log(f"r2_score: {r2score}\n")

    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)
    # Extract model coefficients and plot their barh
    # Creating a dataframe based on the  coefficients
    coef = pd.DataFrame(reg.coef_,index=reg.feature_names_in_, columns=["coefficients"])
    coef = coef.sort_values("coefficients",axis="index")
    logger.log(f"Coefficients:\n{coef}")
    plt.clf()
    # Create a bar plot to compare feature importances visually
    plt.barh(coef.index, coef.iloc[:,0])
    plt.xlabel("coefficients")
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/coefficients{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/coefficients{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 58: Linear regression evaluation

The first algorithm is linear regression (Figure 58). The results show r2_score=0.8 for this algorithm and it does not get better after optimization (Figure 59).

```
################ Evaluating the model(Before Optimization) #################
mean absolute error: 4227.749967449583

mean squared error: 36628649.94480347

r2_score: 0.8006669867688775
```

Figure 59: Linear regression evaluation results before and after optimization

This study performs the optimization process via hyperparameter tuning using GridSearchCV. The hyperparameter_tuning_regression function is responsible for this task for all algorithms (Figure 60).

```python
def hyperparameter_tuning_regression(data : pd.DataFrame, train_test_sets : list, estimator, param : dict, output_folder : str = "output", logger : Logger = None) -> dict:
    logger.log("############ Hyperparameter tuning to optimized model parameters ############\n")
    # grid search to find best classifier parameters (Hyperparameter tuning)

    # Extract train and test sets
    X_train, X_test, y_train, y_test = train_test_sets
    # Some parameters do not match with each other. This code avoids unnecessary warnings
    warnings.filterwarnings("ignore")

    # Create and train gridsearch
    grid = GridSearchCV(estimator, param_grid=param, refit=True, n_jobs=-1, cv=10, scoring="r2")
    grid.fit(X_train, y_train)

    # Find the best paramters
    grid_best_params = grid.best_params_
    grid_best_score = grid.best_score_

    logger.log(f"Best Parameters: {grid_best_params}")
    logger.log(f"Best Score: {grid_best_score}")

    return grid_best_params
```

Figure 60: General function for hyperparameter tuning all algorithms

There are the input parameters for tuning linear regression (Figure 61).

```
# Optimizing the parameters of the linear regression model using hyperparameter tuning
grid_best_params = hyperparameter_tuning_regression(data=df, train_test_sets=train_test_sets, output_folder=output_folder + "/linearregression_model/before_optimization", est
    {"copy_X": [True,False],
     "fit_intercept": [True,False],
     "n_jobs": [1,5,10,15,None],
     "positive": [False]
}, logger=logger)
```

Figure 61: Parameters for tuning linear regression

The results of hyperparameter tuning of linear regression are shown below (Figure 62).

```
############ Hyperparameter tuning to optimized model parameters ############

Best Parameters: {'copy_X': True, 'fit_intercept': True, 'n_jobs': 1, 'positive': False}
Best Score: 0.7236807457496799
```

Figure 62: The results of hyperparameter tuning of linear regression

The second algorithm to evaluate is ridge regression (Figure 63).

```
def evaluate_ridge_regression(reg : ridge_regression, data : pd.DataFrame, X_train : pd.DataFrame, y_train : pd.DataFrame, y_test : pd.DataFrame, y_pred : pd.DataFrame, suffix :
    # Evaluating model
    logger.log(f"################ Evaluating the model{"(" + suffix + ")" if suffix else ''} #################")

    # Extract mean_absolute_error
    mea = mean_absolute_error(y_test, y_pred)
    logger.log(f"mean absolute error: {mea}\n")

    # Extract mean squared error
    mse = mean_squared_error(y_test, y_pred)
    logger.log(f"mean squared error: {mse}\n")

    # Extract r2_score
    r2score = r2_score(y_test, y_pred)
    logger.log(f"r2_score: {r2score}\n")

    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)
    # Extract model coefficients and plot their barh
    # Creating a dataframe based on the  coefficients
    coef = pd.DataFrame(reg.coef_,index=reg.feature_names_in_, columns=["coefficients"])
    coef = coef.sort_values("coefficients",axis="index")
    logger.log(f"Coefficients:\n{coef}")
    plt.clf()
    # Create a bar plot to compare feature importances visually
    plt.barh(coef.index, coef.iloc[:,0])
    plt.xlabel("coefficients")
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/coefficients{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/coefficients{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 63: Ridge regression evaluation

Based on the results, the r2_score is 0.8 for this algorithm (Figure 64).

```
################ Evaluating the model(Before Optimization) #################
mean absolute error: 4233.810829412121

mean squared error: 36886783.74394853

r2_score: 0.7992622233370306
```

Figure 64: Ridge regression evaluation results before optimization

After optimization, the results show no significant improvement (Figure 65).

```
################ Evaluating the model(After Optimization) #################
mean absolute error: 4233.817330540952

mean squared error: 36886703.04181155

r2_score: 0.7992626625178402
```

Figure 65: Ridge regression evaluation results after optimization

There are the input parameters for tuning ridge regression (Figure 66).

```
# Optimizing the parameters of the ridge regression model using hyperparameter tuning
grid_best_params = hyperparameter_tuning_regression(data=df, train_test_sets=train_test_sets, output_folder=output_folder + "/ridgeregression_model/before_optimization", es
    "alpha": [0.0001, 0.001, 0.01, 1, 10, 100 , 1000],
    "copy_X": [True,False],
    "fit_intercept": [True,False],
    "positive": [False],
    "solver": ["svd", "cholesky", "lsqr", "sparse_cg", "sag", "saga"],
    "random_state": [0, 1, 10, 20, 42, None]
}, logger=logger)
```

Figure 66: Parameters for tuning ridge regression

The results of hyperparameter tuning of ridge regression are shown below (Figure 67).

```
############ Hyperparameter tuning to optimized model parameters ############

Best Parameters: {'alpha': 1, 'copy_X': False, 'fit_intercept': True, 'positive': False, 'random_state': None, 'solver': 'saga'}
Best Score: 0.7237844836013961
```

Figure 67: The results of hyperparameter tuning of ridge regression

The third algorithm is decision tree (Figure 68) which gives 0.75 for r2_score before optimization (Figure 69).

```
def evaluate_decisiontree_regression(reg : DecisionTreeRegressor, data : pd.DataFrame, X_train : pd.DataFrame, y_train : pd.DataFrame, y_test : pd.DataFrame, y_pred : pd.DataFra
    # Evaluating model
    logger.log(f"################ Evaluating the model{"(" + suffix + ")" if suffix else ''} ################")

    # Extract mean_absolute_error
    mea = mean_absolute_error(y_test, y_pred)
    logger.log(f"mean absolute error: {mea}\n")

    # Extract mean_squared_error
    mse = mean_squared_error(y_test, y_pred)
    logger.log(f"mean squared error: {mse}\n")

    # Extract r2_score
    r2score = r2_score(y_test, y_pred)
    logger.log(f"r2_score: {r2score}\n")

    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)
    # Extract model feature_importances and plot their barh
    # Creating a dataframe based on the feature importances
    feature_importances = pd.DataFrame(reg.feature_importances_,index=reg.feature_names_in_, columns=["feature_importances"])
    feature_importances = feature_importances.sort_values("feature_importances",axis="index")
    logger.log(f"Importance of features:\n{feature_importances}")
    plt.clf()
    # Create a bar plot to compare feature importances visually
    plt.barh(feature_importances.index, feature_importances.iloc[:,0])
    plt.xlabel("feature_importances")
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/feature_importances{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/feature_importances{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 68: Decision tree regression evaluation

```
################ Evaluating the model(Before Optimization) #################
mean absolute error: 3103.0550115634333

mean squared error: 44499115.733098105

r2_score: 0.7578359333864197
```

Figure 69: Decision tree regression evaluation results before optimization

Different from the previous algorithms, optimization does work for decision tree and improves the r2_score from 0.75 to 0.84 (Figure 70).

```
################ Evaluating the model(After Optimization) #################
mean absolute error: 2994.971058297143

mean squared error: 30011400.984071765

r2_score: 0.8366780375892284
```

Figure 70: Decision tree regression evaluation results after optimization

There are the input parameters for tuning decision tree regression (Figure 71).

```
# Optimizing the parameters of the decision tree model using hyperparameter tuning
grid_best_params = hyperparameter_tuning_regression(data=df, train_test_sets=train_test_sets, estimator=DecisionTreeRegressor(), param = {
    "max_depth": [10,50,100,200,None],
    "min_samples_leaf": [1, 2],
    "min_samples_split": [2, 3, 4],
    "random_state": [0, 1, 10, 20, 42, None],
    "max_features" : ["sqrt", "log2", None]
}, logger=logger)
```

Figure 71: Parameters for tuning decision tree regression

The results of hyperparameter tuning of decision tree regression are shown below (Figure 72).

```
############ Hyperparameter tuning to optimized model parameters ############

Best Parameters: {'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 2, 'min_samples_split': 2, 'random_state': 42}
Best Score: 0.7512383798552673
```

Figure 72: The results of hyperparameter tuning of decision tree

The fourth algorithm is random forest (Figure 73). Random forest gives 0.87 for r2_score before optimization which is a great result (Figure 74).

```
def evaluate_randomforest_regression(reg : RandomForestRegressor, data : pd.DataFrame, X_train : pd.DataFrame, y_train : pd.DataFrame, y_test : pd.DataFrame, y_pred : pd.DataFra
    # Evaluating model
    logger.log(f"################ Evaluating the model{"(" + suffix + ")" if suffix else ''} ################")

    # Extract mean_absolute_error
    mea = mean_absolute_error(y_test, y_pred)
    logger.log(f"mean absolute error: {mea}\n")

    # Extract mean_squared_error
    mse = mean_squared_error(y_test, y_pred)
    logger.log(f"mean squared error: {mse}\n")

    # Extract r2_score
    r2score = r2_score(y_test, y_pred)
    logger.log(f"r2_score: {r2score}\n")

    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)
    # Extract model feature_importances and plot their barh
    # Creating a dataframe based on the feature importances
    feature_importances = pd.DataFrame(reg.feature_importances_,index=reg.feature_names_in_, columns=["feature_importances"])
    feature_importances = feature_importances.sort_values("feature_importances",axis="index")
    logger.log(f"Importance of features:\n{feature_importances}")
    plt.clf()
    # Create a bar plot to compare feature importances visually
    plt.barh(feature_importances.index, feature_importances.iloc[:,0])
    plt.xlabel("feature_importances")
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/feature_importances{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"################### {output_folder}/feature_importances{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 73: Random forest regression evaluation

```
################ Evaluating the model(Before Optimization) #################
mean absolute error: 2619.29533091306

mean squared error: 22828147.729278393

r2_score: 0.8757692822361901
```

Figure 74: Random forest regression evaluation results before optimization

For random forest same as decision tree, optimization has some positive effects on the results and improves the r2_score from 0.87 to 0.88 (Figure 75).

```
################ Evaluating the model(After Optimization) #################
mean absolute error: 2812.225318103302

mean squared error: 21869348.168447934

r2_score: 0.8809870668346674
```

Figure 75: Random forest regression evaluation results after optimization

There are the input parameters for tuning random forest regression (Figure 76).

```
# Optimizing the parameters of the random forest model using hyperparameter tuning
grid_best_params = hyperparameter_tuning_regression(data=df, train_test_sets=train_test_sets, estimator=RandomForestRegressor(), param = {
    "max_depth": [10,50,100,200,None],
    "min_samples_leaf": [1, 2],
    "min_samples_split": [2, 3, 4],
    "random_state": [0, 1, 10, 20, 42, None],
    "max_features" : ["sqrt", "log2"]
}, logger=logger)
```

Figure 76: Parameters for tuning random forest regression

The results of hyperparameter tuning of random forest regression are shown below (Figure 77).

```
############ Hyperparameter tuning to optimized model parameters ############

Best Parameters: {'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 4, 'random_state': None}
Best Score: 0.8329087900304927
```

Figure 77: The results of hyperparameter tuning of random forest

The evaluation of XGBoost algorithm is the last one (Figure 78). XGBoost regression gives 0.86 for r2_score before optimization (Figure 79).

```
def evaluate_xgboost_regression(reg : LinearRegression, data : pd.DataFrame, X_train : pd.DataFrame, y_train : pd.DataFrame, y_test : pd.DataFrame, y_pred : pd.DataFrame, suffix
    # Evaluating model
    logger.log(f"################ Evaluating the model{"(" + suffix + ")" if suffix else ''} ################")

    # Extract mean_absolute_error
    mea = mean_absolute_error(y_test, y_pred)
    logger.log(f"mean absolute error: {mea}\n")

    # Extract mean_squared_error
    mse = mean_squared_error(y_test, y_pred)
    logger.log(f"mean squared error: {mse}\n")

    # Extract r2_score
    r2score = r2_score(y_test, y_pred)
    logger.log(f"r2_score: {r2score}\n")
```

Figure 78: XGBoost regression evaluation

```
################ Evaluating the model(Before Optimization) #################
mean absolute error: 2870.6682332604655

mean squared error: 24969317.67894786

r2_score: 0.8641170412021715
```

Figure 79: XGBoost regression evaluation results before optimization

After optimization, the results get better, and XGBoost creates r2_score=0.90 which is an excellent outcome (Figure 80).

```
################ Evaluating the model(After Optimization) #################
mean absolute error: 2613.1634572248718

mean squared error: 18921059.855536677

r2_score: 0.8970316438030326
```

Figure 80: XGBoost regression evaluation results after optimization

There are the input parameters for tuning of XGBoost regression (Figure 81).

```
# Optimizing the parameters of the xgboost regression model using hyperparameter tuning
grid_best_params = hyperparameter_tuning_regression(data=df, train_test_sets=train_test_sets, output_folder=output_folder + "/xgboost_model/before_optimization", estimator=X
    {"max_depth": [3, 6, 9],
     "learning_rate": [0.03, 0.3, 0.9],
     "subsample": [0.5, 1],
     "min_child_weight": [1, 10, 15],
     "reg_alpha": [0, 0.5, 1],
     "reg_lambda": [0, 0.5, 1],
     "gamma": [0, 1, 10],
}, logger=logger)
```

Figure 81: Parameters for tuning XGBoost regression

The results of hyperparameter tuning of XGBoost regression are shown below (Figure 82).

```
############ Hyperparameter tuning to optimized model parameters ############

Best Parameters: {'gamma': 0, 'learning_rate': 0.03, 'max_depth': 3, 'min_child_weight': 10, 'reg_alpha': 0, 'reg_lambda': 0, 'subsample': 0.5}
Best Score: 0.8467373842411163
```

Figure 82: The results of hyperparameter tuning of XGBoost

At the end of this study, it could be interesting to know how much each operation takes to complete for this dataset (Figure 83).

```
Elapsed time for loading and describing the data: 15 seconds
Elapsed time for cleaning the data: 20 seconds
Elapsed time for regression via linear regression before optimization: 1 seconds
Elapsed time for hyperparameter tuning of linear regression: 2 seconds
Elapsed time for regression via linear regression after optimization: 1 seconds
Elapsed time for regression via ridge regression before optimization: 1 seconds
Elapsed time for hyperparameter tuning of ridge regression: 8 seconds
Elapsed time for regression via ridge regression after optimization: 1 seconds
Elapsed time for regression via decision tree before optimization: 1 seconds
Elapsed time for hyperparameter tuning of decision tree: 5 seconds
Elapsed time for regression via decision tree after optimization: 1 seconds
Elapsed time for regression via random forest before optimization: 2 seconds
Elapsed time for hyperparameter tuning of random forest: 99 seconds
Elapsed time for regression via random forest after optimization: 1 seconds
Elapsed time for regression via xgboost regression before optimization: 1 seconds
Elapsed time for hyperparameter tuning of xgboost regression: 130 seconds
Elapsed time for regression via xgboost regression after optimization: 0 seconds
```

Figure 83: Time elapsed for each operation

At last, this paper provides some charts to compare algorithms based on some of their evaluation metrics after optimization.

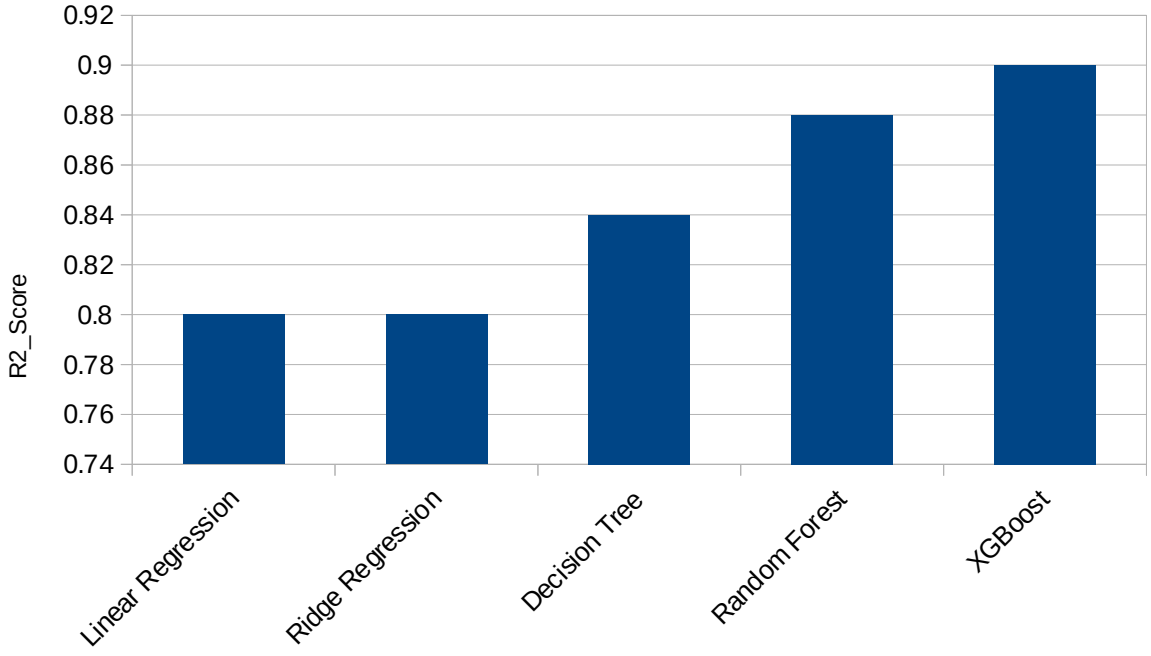

Figure 84: Comparing algorithm based on r2_score

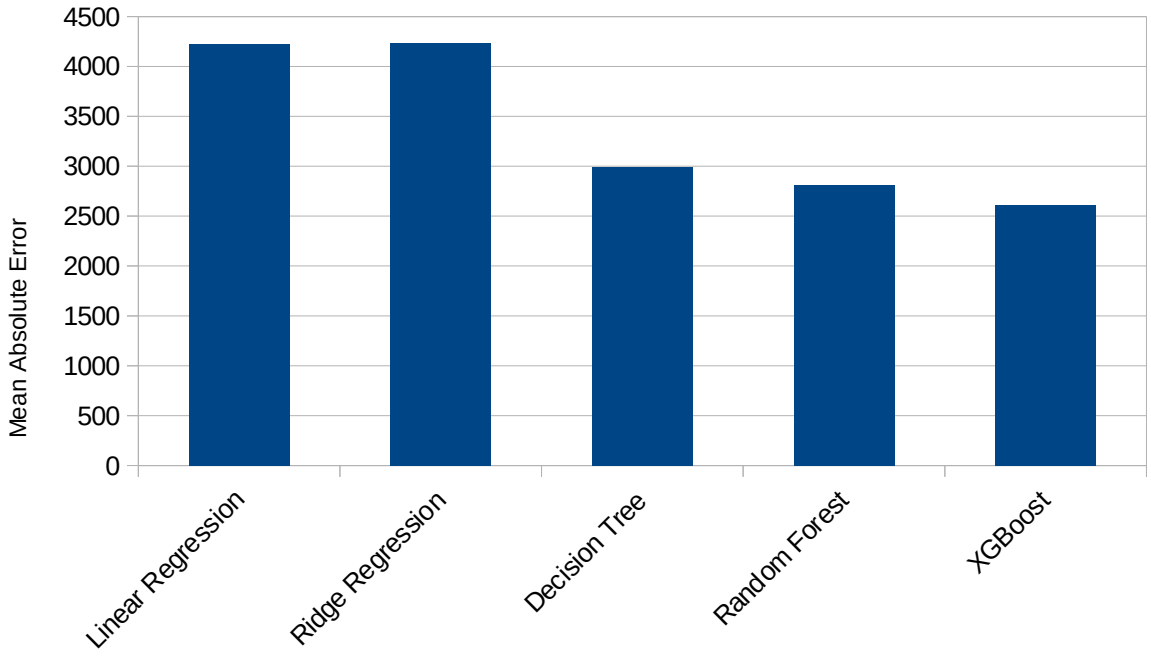

Figure 85: Comparing algorithm based on mean absolute error

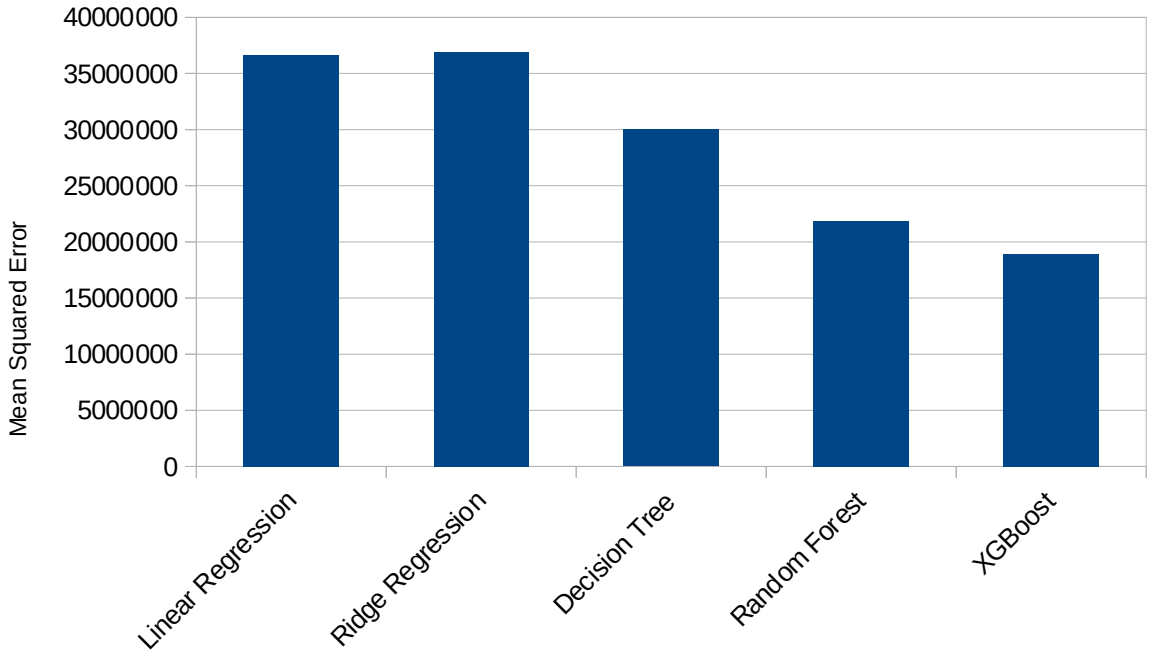

Figure 86: Comparing algorithm based on mean squared error

The above comparisons define that XGBoost regression model has the best performance among all algorithms.

CONCLUSION

This study has different steps each of which reveals some facts about the dataset and models' performance. The data exploratory phase shows that the average "age" of contractors is 39 and most of them are about 20, "bmi" has a normal distribution, and most of the "charges" are under 15,000 and a few of them are much more than this. It also indicates that the "gender" of contractors are equal, most of them are not "smoker", and are equally distributed in four regions. This paper applies five algorithms for this regression problem. The comparison reveals that XGBoost shows the best performance with results of r2_score=0.90, mean absolute error=2613, and mean squared error=18921059 after optimization. Models interpretation results show that "smoker", "bmi", and "age" features have the most impact on model predictions and the other features are not very important. These facts are also approved by the correlation between features and target variables.

BIBLIOGRAPHY

Magaga, A.M. (2021) Identifying, Cleaning and replacing outliers | Titanic Dataset. Available at: https://medium.com/analytics-vidhya/identifying-cleaning-and-replacing-outliers-titanic-dataset-20182a062893 (Accessed: 05 September 2024).

Raja, A.Z. (2023) The Machine Learning Process: From Data Collection to Model Deployment. Available at: https://alizahidraja.medium.com/the-machine-learning-process-from-data-collection-to-model-deployment-afbf1bdf8729 (Accessed: 19 September 2024).

Bonaccorso, G., 2018. Mastering machine learning algorithms: expert techniques to implement popular machine learning algorithms and fine-tune your models. Packt Publishing Ltd.

Chen, T. and Guestrin, C., 2016, August. XGBoost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining (pp. 785-794).

Graphviz (2021) Graph Visualization Software. Available at: https://graphviz.org/ (Accessed: 05 September 2024).

Hunter, J.D. (2007). Matplotlib: A 2D graphics environment. Computing in Science & Engineering, 9(3), 90-95.

IBM (2023) What is predictive analytics?. Available at: https://www.ibm.com/topics/predictive-analytics?utm_content=SRCWW&p1=Search&p4=43700075186401382&p5=p&p9=58700008277893012&gad_source=1&gclid=Cj0KCQjw3bm3BhDJARIsAKnHoVUBboycdCuti5n2cxM5Su3LKEwMQEmU1A9GSl7DAd31bWo6IGELOzIaAriNEALw_wcB&gclsrc=aw.ds (Accessed: 22 September 2024).

Johnston, B. and Mathur, I., 2019. Applied supervised learning with Python: use scikit-learn to build predictive models from real-world datasets and prepare yourself for the future of machine learning. Packt Publishing Ltd.

McKinney, W., et al. (2011). pandas: a foundational Python library for data analysis and statistics.

Miller, J.D. and Forte, R.M., 2017. Mastering Predictive Analytics with R. Packt Publishing Ltd.

Choi, M. (2017) Medical Cost Personal Datasets. Available at: https://www.kaggle.com/datasets/mirichoi0218/insurance (Accessed: 2 September 2024).

Buhl, N. (2023) Training, Validation, Test Split for Machine Learning Datasets. Available at: https://encord.com/blog/train-val-test-split/#:~:text=The%20train%2Dtest%20split%20is,model's%20performance%20and%20generalization%20capabilities.(Accessed: 11 September 2024).

Pedregosa, F., et al. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825-2830.

Vaishalipal (2024) Python's Role in Shaping the Future of AI and Machine Learning. Available at: https://www.datasciencesociety.net/pythons-role-in-shaping-the-future-of-ai-and-machine-learning/#:~:text=Conclusion,artificial%20intelligence%20and%20machine%20learning. (Accessed: 19 September 2024).

Van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2), 22-30.

TABLE OF FIGURES