medical-insurance-pred-full

April 12, 2024

1 Personal Medical Insurance Cost with ML Regression Model

1.1 Libraries imports

Library imports for i/o

[1]:
```
# working with structured data
import pandas as pd

# support for arrays, matrices, and mathematical functions to operate on data␣
↪structures
import numpy as np

# manipulating file paths and directories
import os.path
```

Libraries for general purposes

[2]:
```
# embedding HTML content like visualizations or interactive elements within a␣
↪Jupyter notebook
from IPython.display import HTML

# generating random strings or for various string manipulation tasks
from string import ascii_letters

# generating random numbers within a specified range
from random import randint

import time
```

Filtering warnings

[3]:
```
import warnings
warnings.filterwarnings("ignore")
```

Matplotlib and Seaborn

Seaborn:

1. Seaborn is built on top of Matplotlib and provides a higher-level interface for creating attractive statistical graphics.
2. It simplifies the process of creating complex plots such as histograms, KDE plots, and regression plots by providing easy-to-use functions with sensible defaults.
3. Seaborn is particularly useful for exploring and visualizing relationships in complex datasets.
4. It is well-suited for statistical data analysis and exploratory data visualization.

Matplotlib:

1. Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.
2. It provides a MATLAB-like interface for creating plots and offers fine-grained control over plot customization.
3. Matplotlib is highly customizable and can create nearly any type of plot imaginable, although sometimes with more verbose syntax compared to other libraries.
4. It is a foundational library for data visualization in Python and is often used for creating publication-quality graphics and embedding plots in various applications.

[4]:
```python
# statistical data visualization
import seaborn as sns
# for Q-Q plots
import scipy.stats as stats

# creating static, animated, and interactive visualizations
import matplotlib.pyplot as plt
```

Plotly Packages

1. Plotly is another library for creating interactive visualizations, with a focus on producing publication-quality graphics.
2. It supports a wide range of chart types and offers numerous customization options for fine-tuning plots.
3. Plotly also provides an online platform (Chart Studio) for sharing, collaborating on, and hosting Plotly graphs.
4. It is well-suited for creating interactive visualizations that can be easily shared and embedded in web applications, reports, and presentations.

[5]:
```python
# Python graphing library that makes interactive, publication-quality graphs␣
↪online
from plotly import tools
import chart_studio.plotly as py
import plotly.figure_factory as ff
import plotly.graph_objs as go
from plotly.offline import init_notebook_mode, iplot
# work in offline mode with Jupyter notebooks
init_notebook_mode(connected=True)
```

Plot Library for flexible data visualization purposes

1. It provides a concise and powerful interface for creating a wide variety of plots, including interactive plots suitable for web applications and dashboards.
2. Bokeh emphasizes interactivity and can handle large datasets with ease.
3. It is well-suited for creating complex, interactive visualizations for web-based applications and dashboards.

[6]:
```python
# creating interactive and web-ready visualizations
from bokeh.io import output_notebook, show
from bokeh.plotting import figure
output_notebook()
from bokeh.layouts import gridplot
```

Statistical Libraries

[7]:
```python
from scipy import stats
```

Data-preprocessing libraries

[8]:
```python
# encoding categorical features
from sklearn.preprocessing import LabelEncoder

# generate polynomial and interaction features
from sklearn.preprocessing import PolynomialFeatures

# scale numerical features to a specified range
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
```

Model tuning libraries

[9]:
```python
# chaining together multiple processing steps into a single object
from sklearn.pipeline import Pipeline

# split the dataset into training and testing sets
from sklearn.model_selection import train_test_split

# regression evaluation metrics
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error

# cross-validation technics
from sklearn.model_selection import KFold
from sklearn.model_selection import GroupKFold
from sklearn.model_selection import ShuffleSplit
```

Regression models libraries

[10]:
```python
from sklearn.linear_model import LinearRegression, Ridge, Lasso

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
```

```python
from sklearn.tree import DecisionTreeRegressor, ExtraTreeRegressor

from sklearn.neighbors import KNeighborsRegressor

from sklearn.gaussian_process import GaussianProcessRegressor

import xgboost as xgb
```

Model hyperparameter tuning

[11]:
```python
from sklearn.model_selection import GridSearchCV
```

1.2 Read the Dataset for health insurance

[12]:
```python
# path to the original dataset
df_path ="D://programming//information-technologies-of-smart-systems//
↪calculation-and-graphic work//personal-medical-insurance-cost-prediction//
↪data//insurance.csv"
```

[13]:
```python
# is there such path?
print(os.path.exists(df_path))
```

```
True
```

[14]:
```python
# read the health insurance dataset
df = pd.read_csv(df_path)
```

1.3 Exploratory Data Analysis (EDA)

1.3.1 Data Shapes

[15]:
```python
print('columns count - ', len(df.columns), '\n')
print('columns: ', list(df.columns))
```

```
columns count -  7

columns:  ['age', 'sex', 'bmi', 'children', 'smoker', 'region', 'charges']
```

[21]:
```python
print('Samples count: ', df.shape[0])
```

```
Samples count:  1338
```

[19]:
```python
df.head(10)
```

[19]:

	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
5   31  female  25.740         0      no  southeast   3756.62160
6   46  female  33.440         1      no  southeast   8240.58960
7   37  female  27.740         3      no  northwest   7281.50560
8   37    male  29.830         2      no  northeast   6406.41070
9   60  female  25.840         0      no  northwest  28923.13692
```

Data Types

[22]:
```
df.info()
```

```
<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
```

Descriptive Statistics

[23]:
```
df.describe(include='O')     # for categorical variables
```

[23]:

	sex	smoker	region
count	1338	1338	1338
unique	2	2	4
top	male	no	southeast
freq	676	1064	364

[24]:
```
df.describe(exclude='O')     # for numerical variables
```

[24]:

	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

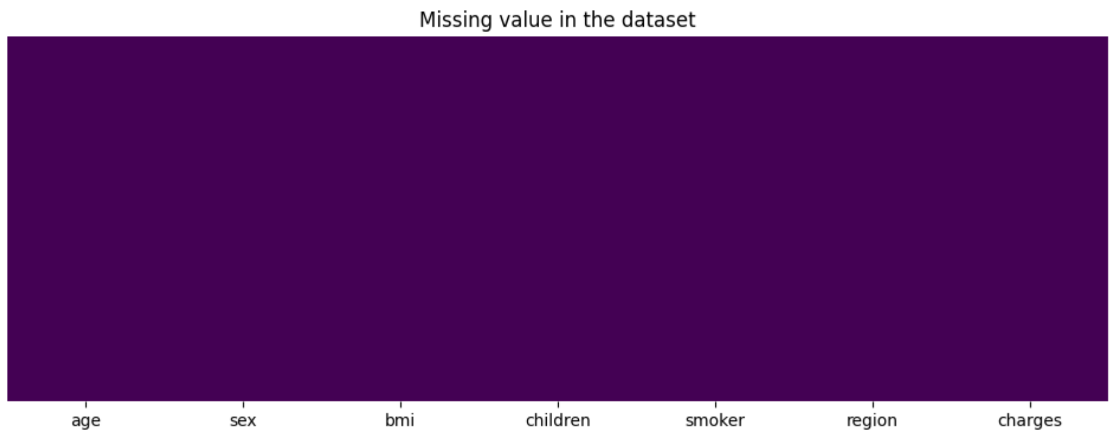
1.3.2 Check for missing value

[26]:
```
df.isnull().sum()
```

[26]:
```
age         0
sex         0
bmi         0
children    0
smoker      0
region      0
charges     0
dtype: int64
```

There is no missing value in the data :)

[25]:
```
# no missing data visualisation
plt.figure(figsize=(12,4))
sns.heatmap(df.isnull(),cbar=False,cmap='viridis',yticklabels=False)
plt.title('Missing value in the dataset')
```

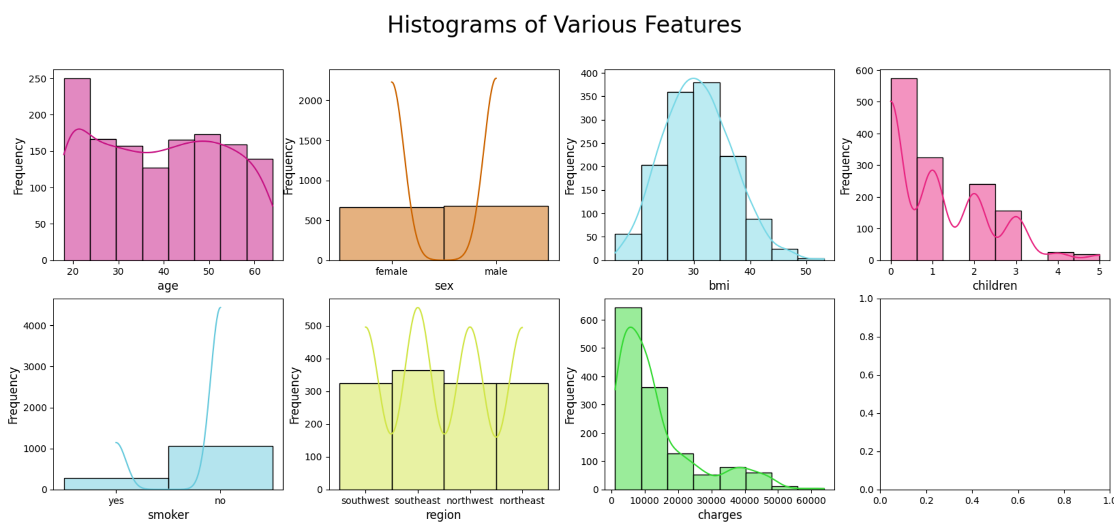


1.3.3 Plots for data exploring

General mixed plots

[40]:
```
selected_cols = [col for col in df.columns if col]

# generate random colors for each selected column
colors = ['mediumvioletred']
for i in range(len(selected_cols)):
    colors.append('#%06X' % randint(0, 0xFFFFFF))

# determine the number of subplots based on the selected columns
num_subplots = len(selected_cols)
```

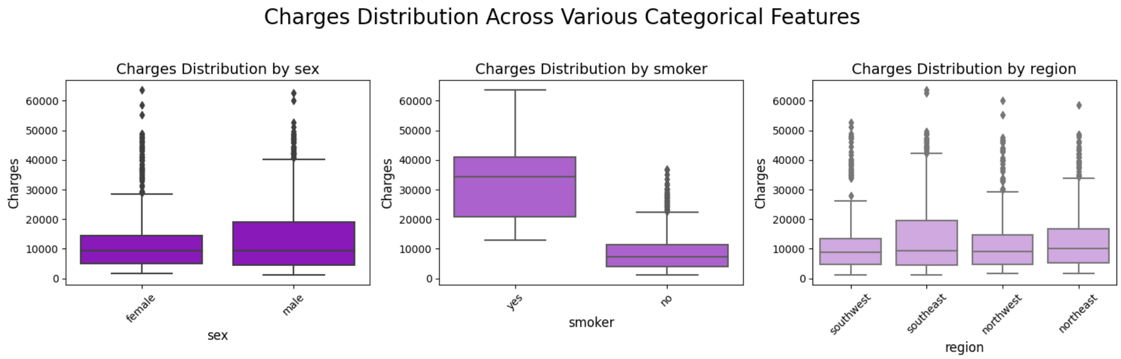
```
num_rows = (num_subplots - 1) // 5 + 1
num_cols = min(4, num_subplots)

# create the figure and axes for subplots
fig, axes = plt.subplots(num_rows, num_cols, figsize=(20, num_rows * 4), facecolor='white')
fig.suptitle("Histograms of Various Features", size=24)

for i in range(num_rows):
    for j in range(num_cols):
        idx = i * num_cols + j
        if idx < num_subplots:
            sns.histplot(df[selected_cols[idx]], ax=axes[i, j], color=colors[idx], kde=True, bins=8)
            axes[i, j].set_xlabel(selected_cols[idx], fontsize=12)  # Set x-axis label font size
            axes[i, j].set_ylabel("Frequency", fontsize=12)         # Set y-axis label font size

#plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



There’s something insanely beautiful about bmi distribution, isn’t there? The average BMI in patients is 30.

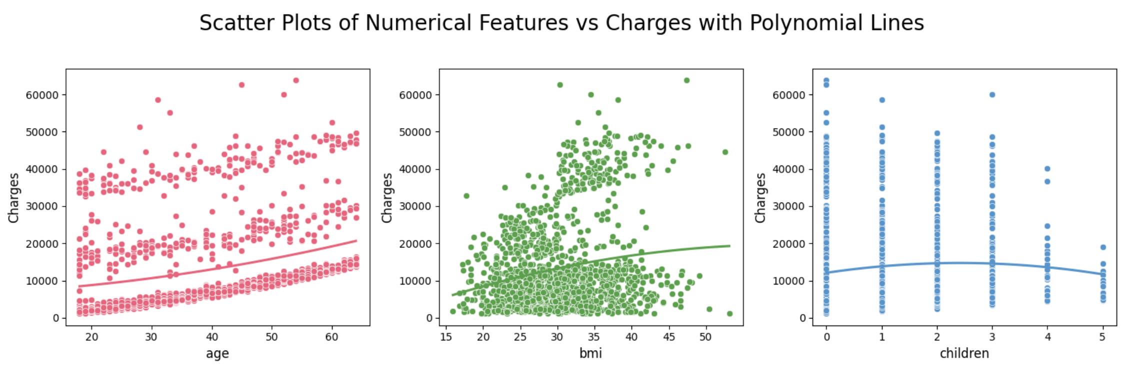
[50]:
```
# selecting categorical columns from the DataFrame
categorical_cols = df.select_dtypes(include=['object']).columns

# extracting column names into a list
```

```python
selected_cols = [col for col in categorical_cols]

# calculating the number of subplots needed
num_subplots = len(selected_cols)
num_rows = (num_subplots - 1) // 3 + 1
num_cols = min(3, num_subplots)

# creating the subplot grid and setting figure size
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, num_rows * 5),
    facecolor='white')

# adding a title to the figure
fig.suptitle("Charges Distribution Across Various Categorical Features",
    size=20)

# generating a palette of colors for visualization
colors = sns.light_palette('darkviolet', n_colors=len(selected_cols)+1,
    reverse=True)

# plotting boxplots for each selected categorical feature
for idx, col in enumerate(selected_cols):
    row = idx // num_cols
    col_idx = idx % num_cols
    ax = axes[idx] if num_rows == 1 else axes[row, col_idx]
    sns.boxplot(x=col, y='charges', data=df, ax=ax, palette=[colors[idx]])
    ax.set_title(f'Charges Distribution by {col}', fontsize=14)
    ax.set_xlabel(col, fontsize=12)
    ax.set_ylabel("Charges", fontsize=12)
    ax.tick_params(axis='x', rotation=45)

# removing excess empty subplots if there are any
for ax in axes.flat[num_subplots:]:
    ax.remove()

# adjusting layout for better visualization
plt.tight_layout(rect=[0, 0.03, 1, 0.95])

# displaying the plot
plt.show()
```



[54]:
```python
# selecting numerical columns from the DataFrame
numerical_cols = df.select_dtypes(include=['float64', 'int64', 'int32']).columns

# excluding the 'charges' column from the selected numerical features
selected_cols = [col for col in numerical_cols if col != 'charges']

# calculating the number of subplots needed
num_subplots = len(selected_cols)
num_rows = (num_subplots - 1) // 3 + 1
num_cols = min(3, num_subplots)

# creating the subplot grid and setting figure size
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, num_rows * 5),
    facecolor='white')

# adding a title to the figure
fig.suptitle("Scatter Plots of Numerical Features vs Charges with Polynomial
    Lines", size=20)

# generating a palette of colors for visualization
palette = sns.husl_palette(n_colors=len(selected_cols), s=0.7, l=0.6)

# plotting scatter plots with polynomial lines for each selected numerical
    feature
for idx, col in enumerate(selected_cols):
    if num_rows == 1 or num_cols == 1:
        ax = axes[idx]
    else:
        row = idx // num_cols
        col_idx = idx % num_cols
        ax = axes[row, col_idx]

    sns.scatterplot(x=col, y='charges', data=df, ax=ax, color=palette[idx])
```

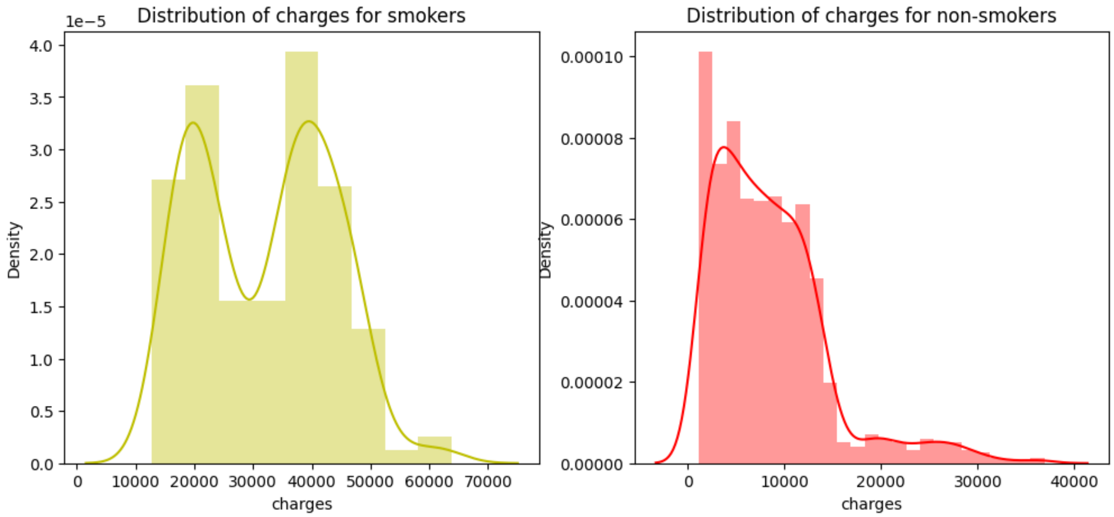
```
    sns.regplot(x=col, y='charges', data=df, ax=ax, scatter=False, order=2,␣
↪color=palette[idx], ci=None)
    ax.set_xlabel(col, fontsize=12)
    ax.set_ylabel("Charges", fontsize=12)

# removing excess empty subplots if there are any
for ax in axes.flat[num_subplots:]:
    ax.remove()

# adjusting layout for better visualization
plt.tight_layout(rect=[0, 0.03, 1, 0.95])

# displaying the plot
plt.show()
```



Distribution for smokers Smoking patients spend more on treatment.

[114]:
```
f= plt.figure(figsize=(12,5))

ax=f.add_subplot(121)
sns.distplot(df[(df.smoker == 1)]["charges"],color='y',ax=ax)
ax.set_title('Distribution of charges for smokers')

ax=f.add_subplot(122)
sns.distplot(df[(df.smoker == 0)]['charges'],color='r',ax=ax)
ax.set_title('Distribution of charges for non-smokers')
```

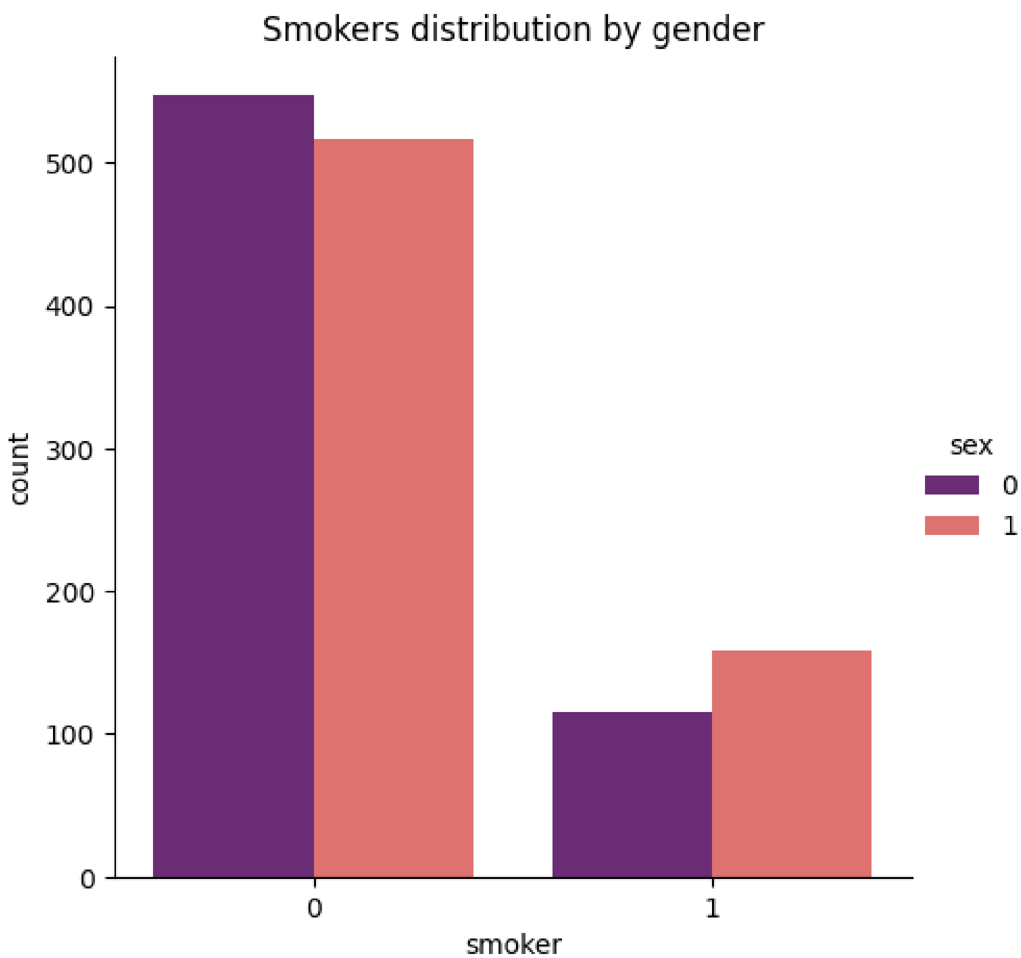
[114]:
```
Text(0.5, 1.0, 'Distribution of charges for non-smokers')
```



Exploring dependencies of gender distribution and smoking Please note that women are coded with the symbol "0" and men - "1". Thus non-smoking people and the truth more. Also we can notice that more male smokers than women smokers. It can be assumed that the total cost of treatment in men will be more than in women, given the impact of smoking.

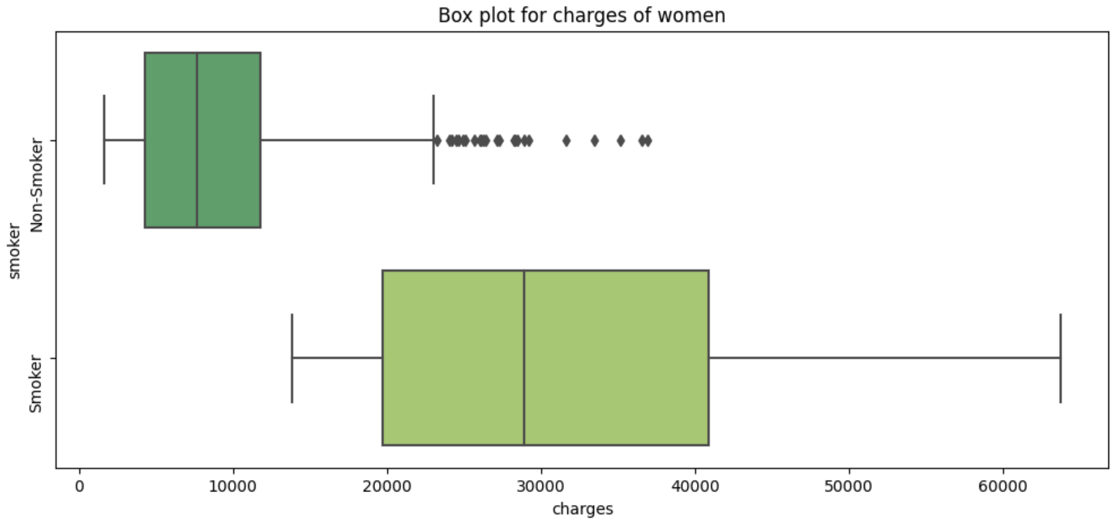
[112]:
```
sns.catplot(x="smoker", kind="count", hue = 'sex', palette="magma", data=df)
plt.title("Smokers distribution by gender")
# plt.xticks([0, 1], ['Non-Smoker', 'Smoker'])
```

[112]:
```
Text(0.5, 1.0, 'Smokers distribution by gender')
```



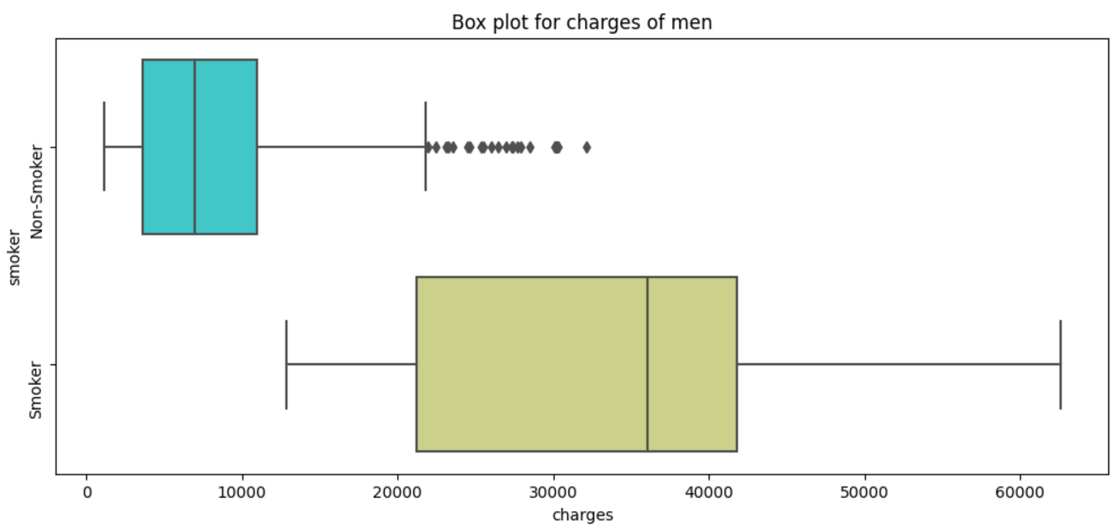
[229]:
```
plt.figure(figsize=(12,5))
plt.title("Box plot for charges of women")
sns.boxplot(y="smoker", x="charges", data =  df[(df.sex == 0)] , orient="h",␣
 ↪palette = 'summer')
# adding annotations to clarify y-axis labels
plt.yticks([0, 1], ['Non-Smoker', 'Smoker'], rotation = 90)
```

[229]:
```
([<matplotlib.axis.YTick at 0x1c0e3873b10>,
  <matplotlib.axis.YTick at 0x1c0e3870c10>],
 [Text(0, 0, 'Non-Smoker'), Text(0, 1, 'Smoker')])
```



[227]:
```python
plt.figure(figsize=(12,5))
plt.title("Box plot for charges of men")
sns.boxplot(y="smoker", x="charges", data =  df[(df.sex == 1)] , orient="h",␣
 ↪palette = 'rainbow')
# adding annotations to clarify y-axis labels
plt.yticks([0, 1], ['Non-Smoker', 'Smoker'], rotation = 90)
```

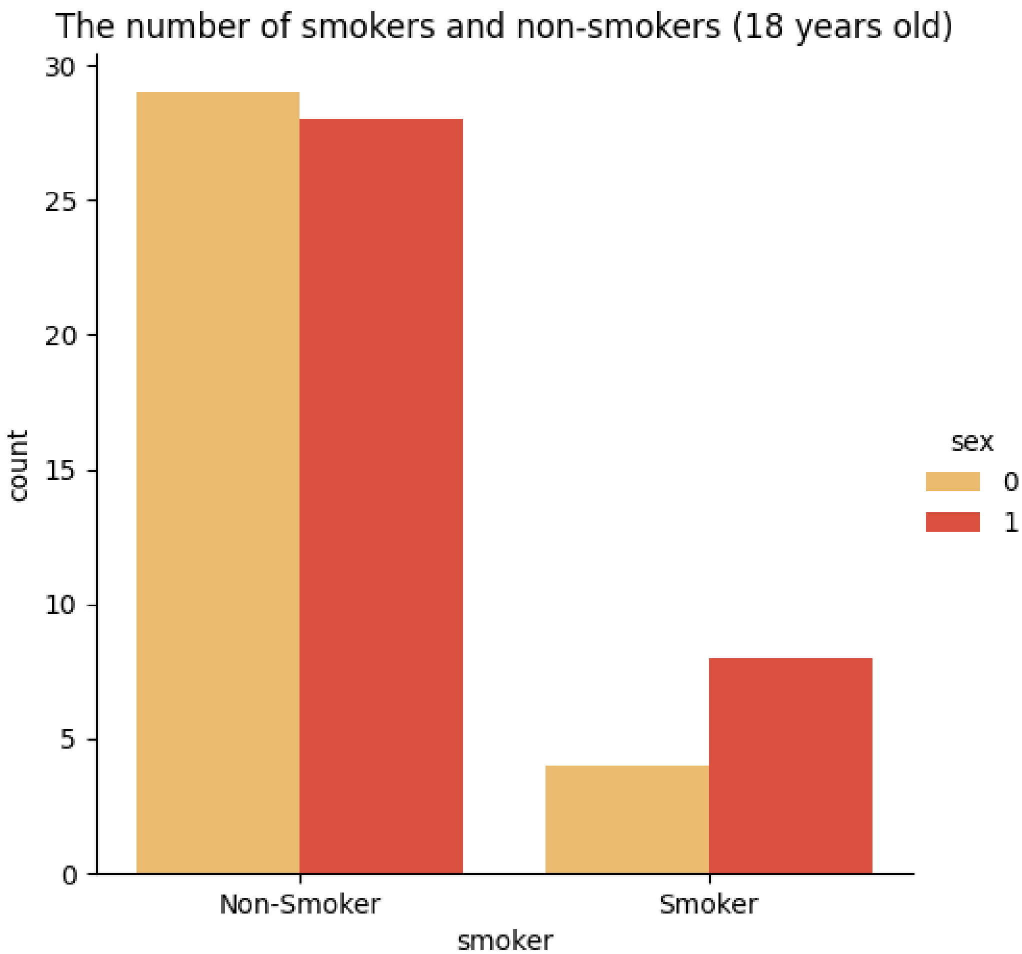
[227]:
```
([<matplotlib.axis.YTick at 0x1c0e38ed1d0>,
  <matplotlib.axis.YTick at 0x1c0e7e6ca50>],
 [Text(0, 0, 'Non-Smoker'), Text(0, 1, 'Smoker')])
```



Exploring dependencies between gender distribution (y.o. 18), smoking and charges

[222]:
```
sns.catplot(x="smoker", kind="count",hue = 'sex', palette="YlOrRd", data=df[(df.
↪age == 18)])
plt.title("The number of smokers and non-smokers (18 years old)")
plt.xticks([0, 1], ['Non-Smoker', 'Smoker'])
```

[222]:
```
([<matplotlib.axis.XTick at 0x1c0e3b2f5d0>,
  <matplotlib.axis.XTick at 0x1c0e3b3ac90>],
 [Text(0, 0, 'Non-Smoker'), Text(1, 0, 'Smoker')])
```

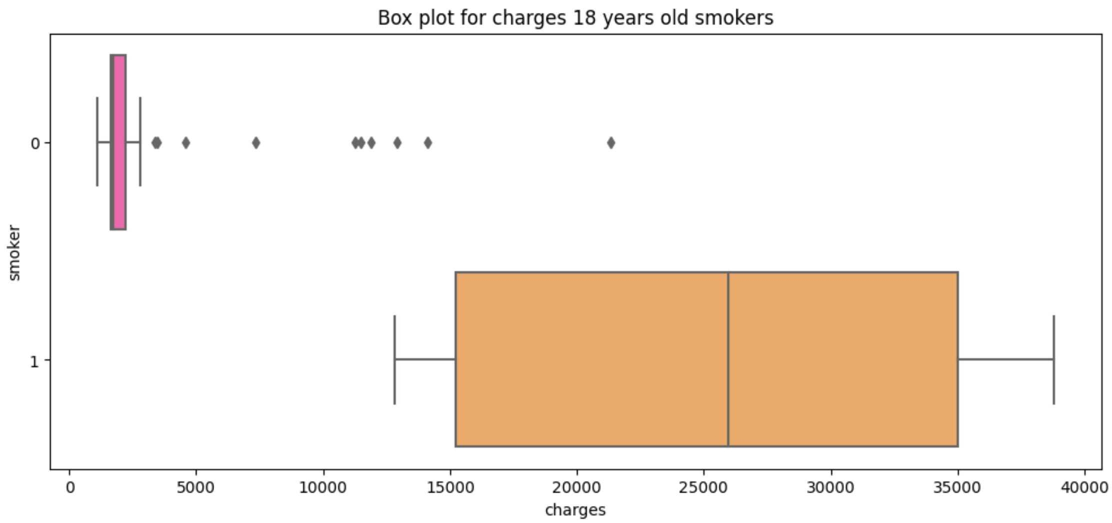


As we can see, even at the age of 18 smokers spend much more on treatment than non-smokers. Among non-smokers we are seeing some "tails". I can assume that this is due to serious diseases or accidents. Now let's see how the cost of treatment depends on the age of smokers and non-smokers patients.

[223]:
```
plt.figure(figsize=(12,5))
plt.title("Box plot for charges 18 years old smokers")
```

```
sns.boxplot(y="smoker", x="charges", data = df[(df.age == 18)] , orient="h",␣
    ↪palette = 'spring')
```

[223]: <Axes: title={'center': 'Box plot for charges 18 years old smokers'}, xlabel='charges', ylabel='smoker'>



[233]:
```
# non-smokers
p = figure(width=500, height=450)
p.circle(x=df[(df.smoker == 0)].age, y=df[(df.smoker == 0)].charges, size=7,␣
    ↪line_color="navy", fill_color="orange", fill_alpha=0.9)

show(p)
```

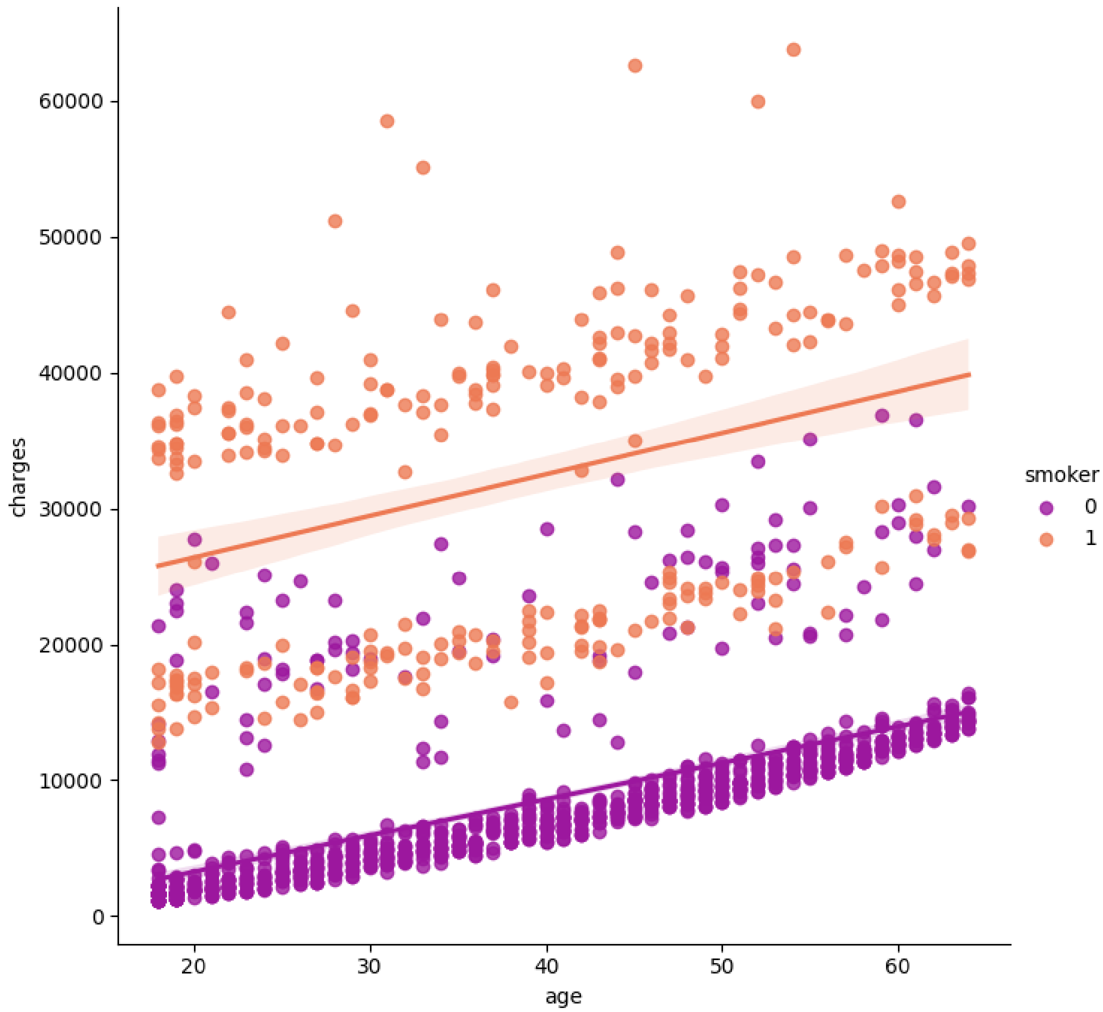
[232]:
```
#smokers
p = figure(width=500, height=450)
p.circle(x=df[(df.smoker == 1)].age, y=df[(df.smoker == 1)].charges, size=7,␣
    ↪line_color="green", fill_color="red", fill_alpha=0.9)

show(p)
```

In non-smokers, the cost of treatment increases with age. That makes sense. In smoking people, we do not see such dependence. I think that it is not only in smoking but also in the peculiarities of the dataset. Such a strong effect of Smoking on the cost of treatment would be more logical to judge having a set of data with a large number of records and signs. Let's pay attention to bmi. I am surprised that this figure but affects the cost of treatment in patients. Or are we on a diet for nothing?

[240]:
```
sns.lmplot(x="age", y="charges", hue="smoker", data=df, palette='plasma',␣
    ↪height=7)
```

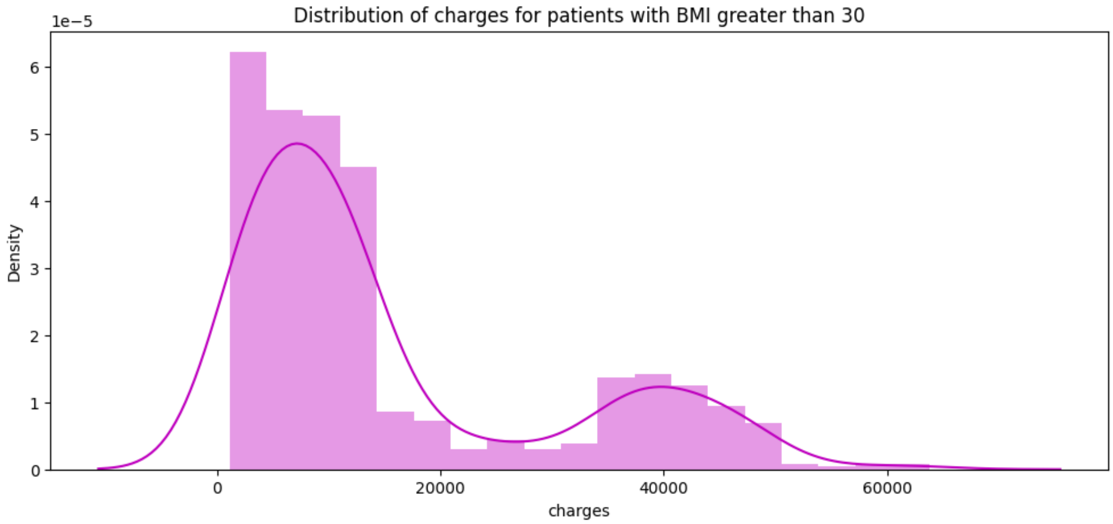
```
ax.set_title('Smokers and non-smokers')
```

[240]: Text(0.5, 1.0, 'Smokers and non-smokers')

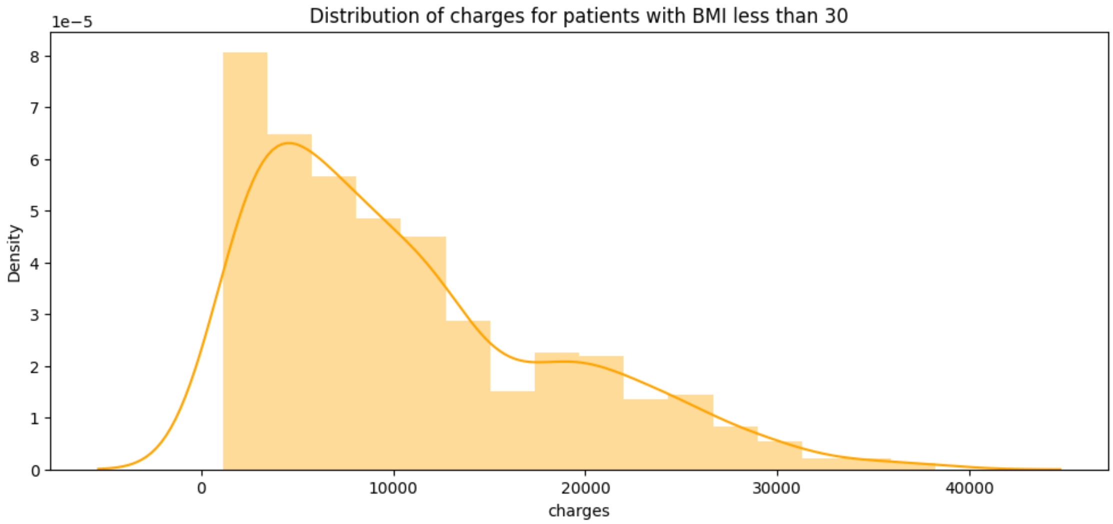


Plot distributing by bmi Let's start to explore distribution of bmi. First, let's look at the distribution of costs in patients with BMI greater than 30 and less than 30.

[245]:
```
plt.figure(figsize=(12,5))
plt.title("Distribution of charges for patients with BMI greater than 30")
ax = sns.distplot(df[(df.bmi >= 30)]['charges'], color = 'm')
```



[247]:
```
plt.figure(figsize=(12,5))
plt.title("Distribution of charges for patients with BMI less than 30")
ax = sns.distplot(df[(df.bmi < 30)]['charges'], color = 'orange')
```

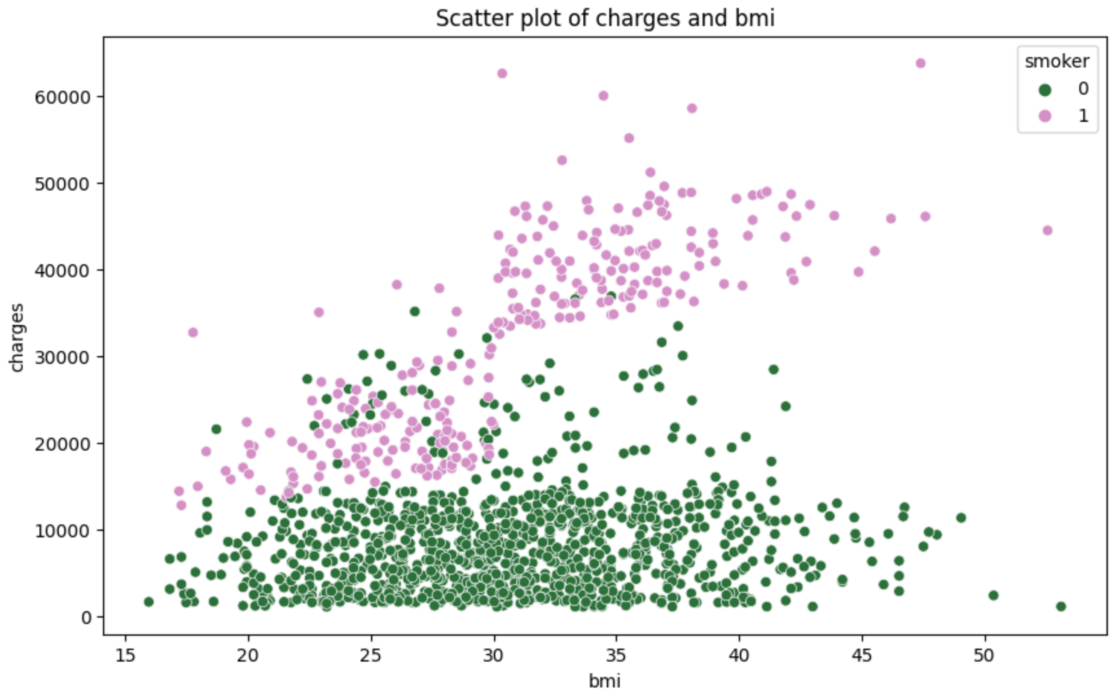


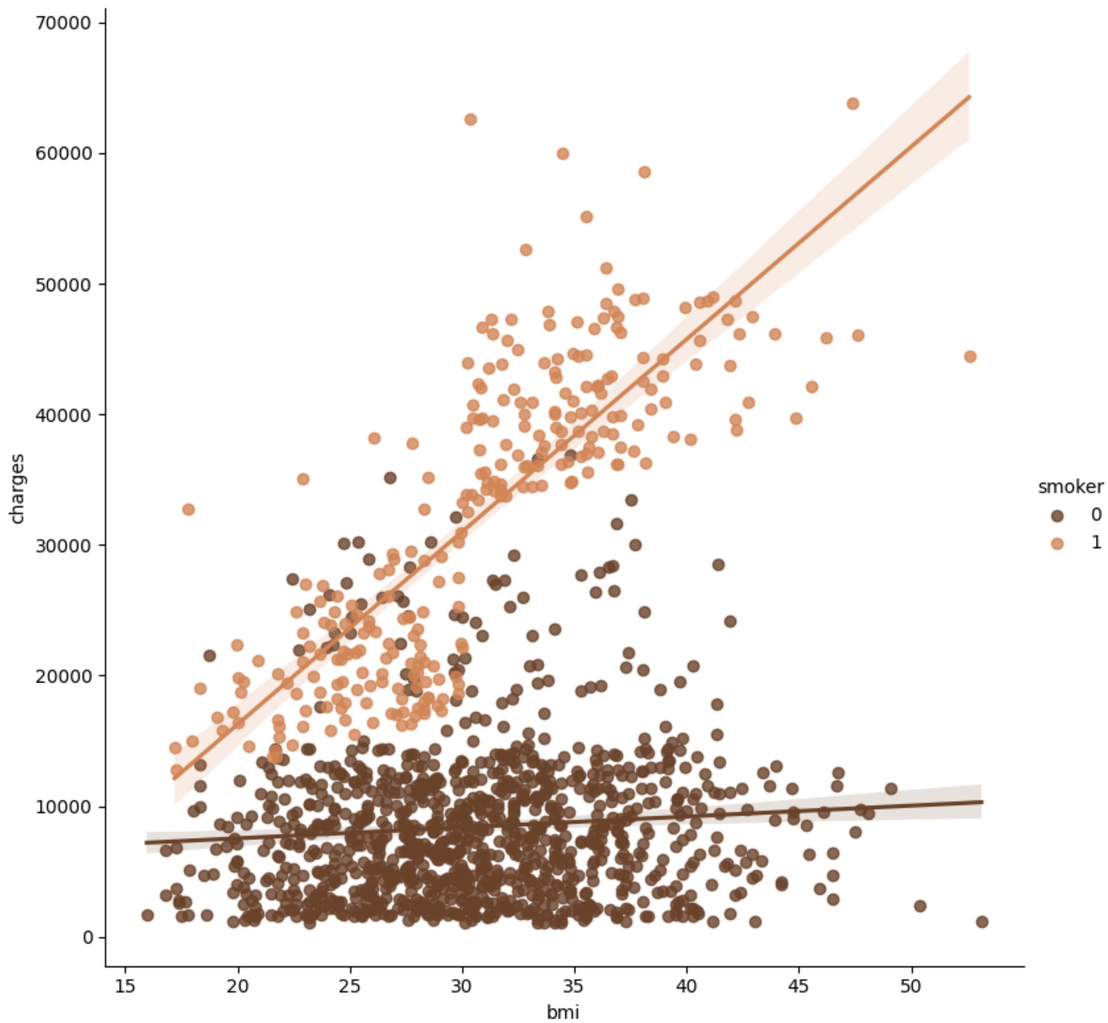
Patients with BMI above 30 spend more on treatment.

[250]:
```
plt.figure(figsize=(10,6))
ax = sns.
 ↪scatterplot(x='bmi',y='charges',data=df,palette='cubehelix',hue='smoker')
ax.set_title('Scatter plot of charges and bmi')
```

```
sns.lmplot(x="bmi", y="charges", hue="smoker", data=df, palette = 'copper',␣
↪height = 8)
```

[250]: <seaborn.axisgrid.FacetGrid at 0x1c0ea9a5210>





Age Analysis: Turning Age into Categorical Variables: 1. Young Adult: from 18 - 35 2. Senior Adult: from 36 - 55 3. Elder: 56 or older 4. Share of each Category: Young Adults (42.9%), Senior Adults (41%) and Elder (16.1%)

[251]:
```
df['age_cat'] = np.nan
lst = [df]

for col in lst:
    col.loc[(col['age'] >= 18) & (col['age'] <= 35), 'age_cat'] = 'Young Adult'
    col.loc[(col['age'] > 35) & (col['age'] <= 55), 'age_cat'] = 'Senior Adult'
    col.loc[col['age'] > 55, 'age_cat'] = 'Elder'


labels = df["age_cat"].unique().tolist()
amount = df["age_cat"].value_counts().tolist()
```

```
colors = ["#ff9999", "#b3d9ff", " #e6ffb3"]

trace = go.Pie(labels=labels, values=amount,
               hoverinfo='label+percent', textinfo='value',
               textfont=dict(size=20),
               marker=dict(colors=colors,
                           line=dict(color='#000000', width=2)))

data = [trace]
layout = go.Layout(title="Amount by Age Category")

fig = go.Figure(data=data, layout=layout)
iplot(fig, filename='basic_pie_chart')
```

Is there a Relationship between BMI and Age

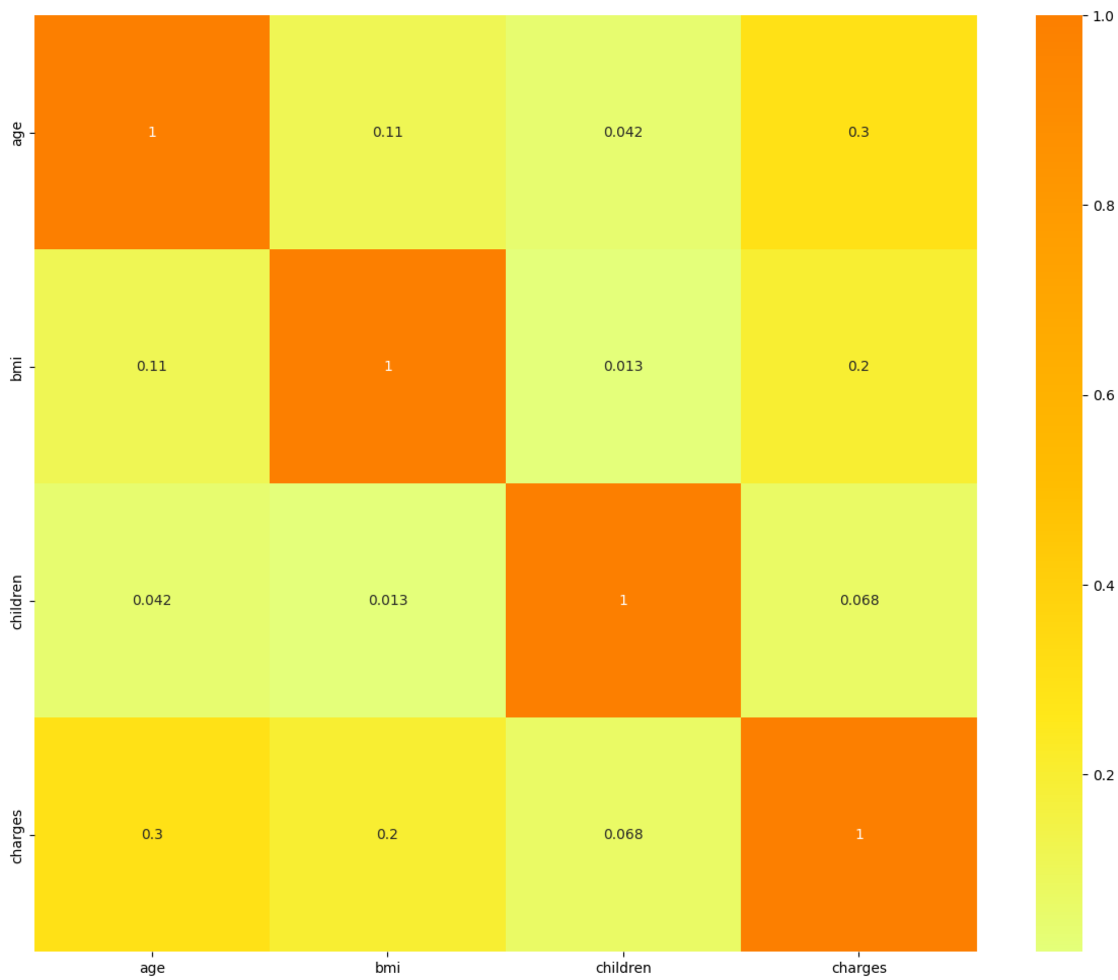
1. BMI frequency: Most of the BMI frequency is concentrated between 27 - 33.
2. Correlations Age and charges have a correlation of 0.29 while bmi and charges have a correlation of 0.19
3. Relationship betweem BMI and Age: The correlation for these two variables is 0.10 which is not that great. Therefore, we can disregard that age has a huge influence on BMI.

[252]:
```
# extracting BMI values for different age categories
young_adults = df["bmi"].loc[df["age_cat"] == "Young Adult"].values
senior_adult = df["bmi"].loc[df["age_cat"] == "Senior Adult"].values
elders = df["bmi"].loc[df["age_cat"] == "Elder"].values

# creating Box plots for each age category
trace0 = go.Box(
    y=young_adults,
    name='Young Adults',
    boxmean=True,
    marker=dict(
        color='rgb(214, 12, 140)',
    )
)
trace1 = go.Box(
    y=senior_adult,
    name='Senior Adults',
    boxmean=True,
    marker=dict(
        color='rgb(0, 128, 128)',
    )
)

trace2 = go.Box(
```

```
    y=elders,
    name='Elders',
    boxmean=True,
    marker=dict(
        color='rgb(247, 186, 166)',
    )
)

# combining data into a list
data = [trace0, trace1, trace2]

# creating layout for the plot
layout = go.Layout(
    title="Body Mass Index <br> by Age Category",
    xaxis=dict(title="Age Category", titlefont=dict(size=16)),
    yaxis=dict(title="Body Mass Index", titlefont=dict(size=16))
)

# creating the figure and plotting it
fig = go.Figure(data=data, layout=layout)
iplot(fig)
```

Correlation plot without categorical variables

[59]:
```
# features for matrix
col_for_corr = ['age', 'bmi', 'children', 'charges']

# correlation matrix calculation
corr = df[col_for_corr].corr()
sns.heatmap(corr, cmap = 'Wistia', annot= True)
```



1.4 Feature engineering

Feature engineering is the process of selecting and transforming variables (features) to improve model performance in machine learning tasks. It involves creating new features, selecting relevant ones, and encoding categorical variables, among other techniques.

Importance of Feature Engineering: - **Improved Model Performance**. Well-engineered features can lead to better model accuracy and generalization. - **Better Interpretability**. Properly engineered features can make models more interpretable and understandable. - **Reduced Overfitting**. Feature engineering can help in reducing overfitting by providing the model with more relevant information.

Techniques in Feature Engineering

1. **Imputation**. Handling missing values in the dataset using techniques like mean, median, or mode imputation.
2. **Feature Scaling**. Scaling features to a similar range, such as normalization or standardization, to prevent bias in models.

3. **Feature Encoding**. Converting categorical variables into numerical format, such as one-hot encoding or label encoding.
4. **Feature Transformation**. Transforming features using techniques like logarithm, square root, or Box-Cox transformation to make them more suitable for modeling.
5. **Feature Selection**. Selecting the most relevant features using techniques like correlation analysis, feature importance, or recursive feature elimination.
6. **Interaction Features**. Creating new features by combining existing ones, such as product or ratio features, to capture interaction effects.
7. **Dimensionality Reduction**. Reducing the number of features using techniques like Principal Component Analysis (PCA) or Singular Value Decomposition (SVD) to simplify the model.

1.4.1 Encoding Categorical Features with LabelEncoder()

Categorical variables are non-numeric variables that represent categories or groups. Label encoding is a technique used to convert categorical variables into numerical format, where each category is assigned a unique integer.

[15]:
```
# sex feature
le = LabelEncoder()
le.fit(df.sex.drop_duplicates())
df.sex = le.transform(df.sex)
```

[16]:
```
# get the mapping between original categories and encoded values
encoded_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
print(encoded_mapping)
```

```
{'female': 0, 'male': 1}
```

[167]:
```
df.head(5)
```

[167]:

	age	sex	bmi	children	smoker	region	charges
0	19	0	27.900	0	yes	southwest	16884.92400
1	18	1	33.770	1	no	southeast	1725.55230
2	28	1	33.000	3	no	southeast	4449.46200
3	33	1	22.705	0	no	northwest	21984.47061
4	32	1	28.880	0	no	northwest	3866.85520

[17]:
```
# smoker or not feature
le.fit(df.smoker.drop_duplicates())
df.smoker = le.transform(df.smoker)
```

[169]:
```
# get the mapping between original categories and encoded values
encoded_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
print(encoded_mapping)
```

```
{'no': 0, 'yes': 1}
```

[65]:
```
df.head(5)
```

[65]:

	age	sex	bmi	children	smoker	region	charges
0	19	0	27.900	0	1	southwest	16884.92400
1	18	1	33.770	1	0	southeast	1725.55230
2	28	1	33.000	3	0	southeast	4449.46200
3	33	1	22.705	0	0	northwest	21984.47061
4	32	1	28.880	0	0	northwest	3866.85520

A few words about coding “region”. In general, categorical variables with large variability are best encoded using OneHotEncoder and so on. But in this case, nothing will change, because there is no special order in which the regions would be listed

[18]:
```
# region feature
le.fit(df.region.drop_duplicates())
df.region = le.transform(df.region)
```

[171]:
```
# get the mapping between original categories and encoded values
encoded_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
print(encoded_mapping)
```

```
{'northeast': 0, 'northwest': 1, 'southeast': 2, 'southwest': 3}
```

[82]:
```
df.head(5)
```

[82]:

	age	sex	bmi	children	smoker	region	charges
0	19	0	27.900	0	1	3	16884.92400
1	18	1	33.770	1	0	2	1725.55230
2	28	1	33.000	3	0	2	4449.46200
3	33	1	22.705	0	0	1	21984.47061
4	32	1	28.880	0	0	1	3866.85520

[172]:
```
df.info()
```

```
<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   int32
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64
 4   smoker    1338 non-null   int32
 5   region    1338 non-null   int32
 6   charges   1338 non-null   float64
dtypes: float64(2), int32(3), int64(2)
memory usage: 57.6 KB
```

Save the new encoded dataset

[173]:
```
df_encoded = df.copy()
df_encoded.to_csv('.//data//explore//insurance_encoded.csv', index=False)
```

[11]:
```
dataset_encoded_path = "D://programming//
↪information-technologies-of-smart-systems//calculation-and-graphic work//
↪personal-medical-insurance-cost-prediction//data//explore//insurance_encoded.
↪csv"
```

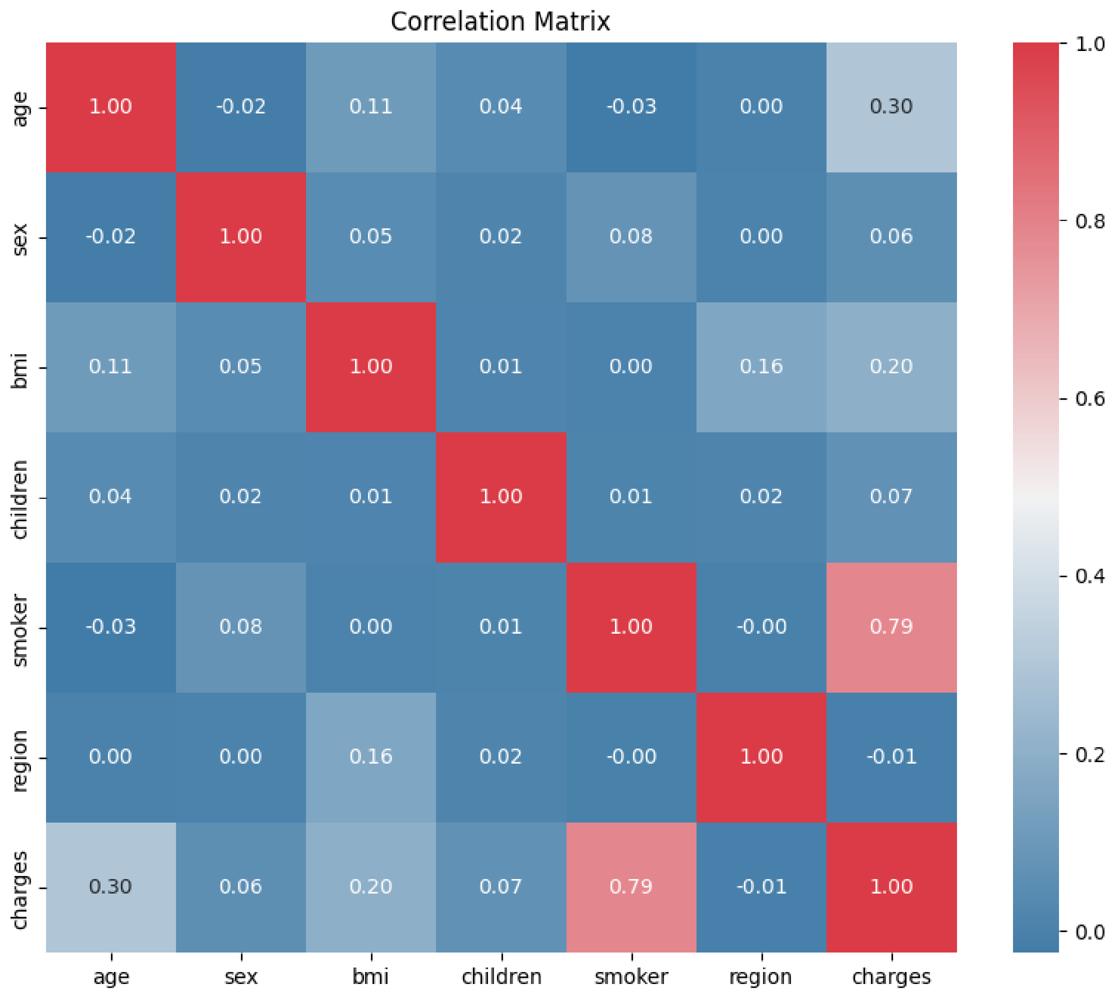
[12]:
```
# is there such path?
print(os.path.exists(dataset_encoded_path))
```

```
True
```

[13]:
```
df_encoded= pd.read_csv(dataset_encoded_path)
```

Correlation plot with encoded categorical variables

[88]:
```
# create a subplot with specified figure size
f, ax = plt.subplots(figsize=(10, 8))

corr = df.corr()

# Create a heatmap of the correlation matrix
# Set the mask to hide the upper triangle of the heatmap
# Set the color map using seaborn's diverging_palette
# Add annotations to display correlation values in each cell
# Format annotations to two decimal places
sns.heatmap(corr, mask=np.zeros_like(corr, dtype=bool), cmap=sns.
↪diverging_palette(240,10,as_cmap=True),
            square=True, ax=ax, annot=True, fmt=".2f")

ax.set_title('Correlation Matrix')
plt.show()
```



[19]:
```python
# corelation matrix in text output
corr = df.corr()
print("Correlation Matrix:")
print(corr)
```

```
Correlation Matrix:
               age       sex       bmi  children    smoker    region   charges
age       1.000000 -0.020856  0.109272  0.042469 -0.025019  0.002127  0.299008
sex      -0.020856  1.000000  0.046371  0.017163  0.076185  0.004588  0.057292
bmi       0.109272  0.046371  1.000000  0.012759  0.003750  0.157566  0.198341
children  0.042469  0.017163  0.012759  1.000000  0.007673  0.016569  0.067998
smoker   -0.025019  0.076185  0.003750  0.007673  1.000000 -0.002181  0.787251
region    0.002127  0.004588  0.157566  0.016569 -0.002181  1.000000 -0.006208
charges   0.299008  0.057292  0.198341  0.067998  0.787251 -0.006208  1.000000
```

1.4.2 Outlier Engineering

Outliers are data points that significantly differ from other observations in a dataset. Outlier engineering involves identifying and handling these outliers to prevent them from adversely affecting model performance.

Techniques for Outlier Engineering

1. **Identification**:
   - Statistical Methods: Use statistical techniques such as Z-score, interquartile range (IQR), or modified Z-score to identify outliers.
   - Visualization: Plotting techniques like box plots, scatter plots, or histograms can help visually identify outliers.
   - Domain Knowledge: Leverage domain knowledge to identify anomalies that may be outliers in the dataset.
2. **Handling Outliers**:
   - Removal. Exclude outliers from the dataset if they are deemed to be errors or anomalies.
   - Transformation. Apply transformations such as log transformation or Winsorization to mitigate the impact of outliers.
   - Imputation. Impute outliers with more representative values using techniques like mean, median, or nearest neighbors.
3. **Model-Specific Approaches**:
   - Robust Models. Use robust machine learning algorithms that are less sensitive to outliers, such as robust regression or tree-based models.
   - Weighted Models. Assign different weights to outliers or downsample them to reduce their influence on the model.
4. **Ensemble Methods**:
   - Ensemble techniques like bagging or boosting can help in reducing the impact of outliers by combining predictions from multiple models.

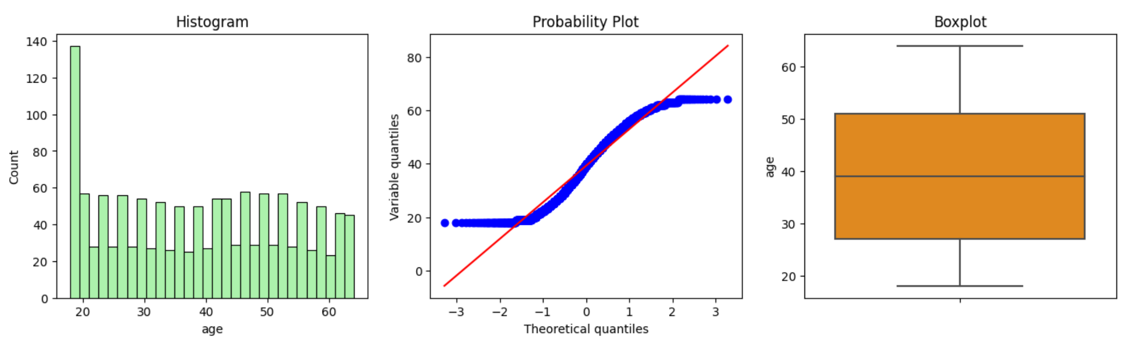
[211]:
```python
# function to create histogram, Q-Q plot and

def diagnostic_plots(df, variable):
    # function takes a dataframe (df) and
    # the variable of interest as arguments

    # define figure size
    plt.figure(figsize=(16, 4))

    # histogram
    plt.subplot(1, 3, 1)
    sns.histplot(df[variable], bins=30, color='lightgreen')
    plt.title('Histogram')

    # Q-Q plot
    plt.subplot(1, 3, 2)
    stats.probplot(df[variable], dist="norm", plot=plt)
    plt.ylabel('Variable quantiles')
```

```
    plt.gca().get_lines()[0].set_color('blue')

    # boxplot
    plt.subplot(1, 3, 3)
    sns.boxplot(y=df[variable], color='darkorange')
    plt.title('Boxplot')

    plt.show()
```

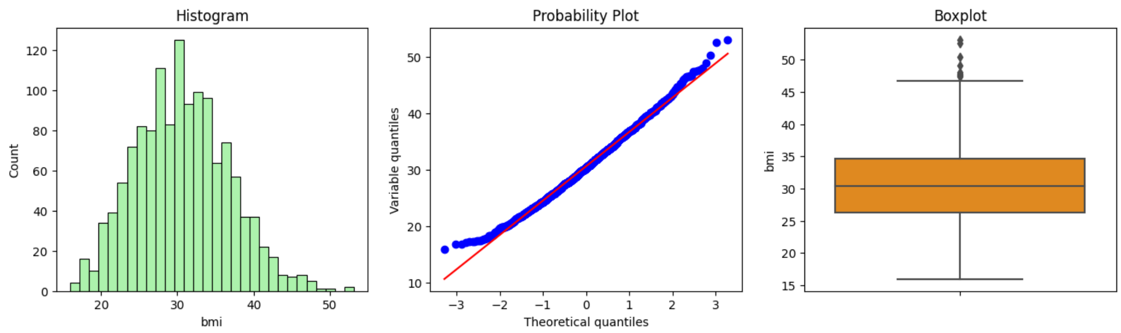
[118]:
```
# display unique values of each feature
df.nunique()
```

[118]:
```
age            47
sex             2
bmi           548
children        6
smoker          2
region          4
charges      1337
dtype: int64
```

[212]:
```
# let's find outliers in age
diagnostic_plots(df, 'age')
```



[213]:
```
# let's find outliers in bmi
diagnostic_plots(df, 'bmi')
```



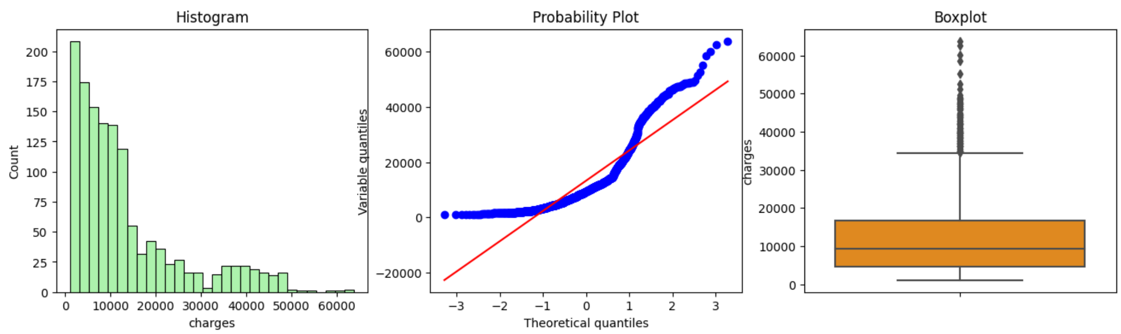
BMI (Body Mass Index) can definitely be greater than 50. BMI is a measure of body fat based on a person's weight and height, calculated by dividing weight in kilograms by height in meters squared. It's a useful tool for assessing whether someone is underweight, normal weight, overweight, or obese.

A BMI of 50 or above would indicate extreme obesity (Obese Class III). While less common, it's certainly possible for individuals to have BMIs in this range, especially in cases of severe obesity. However, it's important to note that BMI is just one indicator of health and doesn't account for factors like muscle mass, bone density, or distribution of fat, so it's not always a perfect measure of an individual's health status.

[137]:
```
bmi_gt_50 = df[df['bmi'] > 50][['bmi', 'sex', 'charges', 'smoker']]
print(bmi_gt_50)
```

```
        bmi  sex     charges  smoker
847   50.38    1   2438.0552       0
1047  52.58    1  44501.3982       1
1317  53.13    1   1163.4627       0
```

[214]:
```
# let's find outliers in charges
diagnostic_plots(df, 'charges')
```



Charges are greater than 35000 for smokers.

[151]:
```
charges_gt_50000 = df[(df['charges'] > 35000) & (df['smoker'] == 1)][['bmi',␣
↪'sex', 'charges', 'smoker']]
print(charges_gt_50000)
```

```
          bmi  sex      charges  smoker
14     42.130    1  39611.75770       1
19     35.300    1  36837.46700       1
23     31.920    0  37701.87680       1
29     36.300    1  38711.00000       1
30     35.600    1  35585.57600       1
…         …    …          …       …
1300   30.360    1  62592.87309       1
1301   30.875    1  46718.16325       1
1303   27.800    1  37829.72420       1
1313   34.700    0  36397.57600       1
1323   40.370    0  43896.37630       1

[130 rows x 4 columns]
```

[152]:
```
charges_gt_50000 = df[(df['charges'] > 35000)][['bmi', 'sex', 'charges',␣
↪'smoker']]
print(charges_gt_50000)
```

```
          bmi  sex      charges  smoker
14     42.130    1  39611.75770       1
19     35.300    1  36837.46700       1
23     31.920    0  37701.87680       1
29     36.300    1  38711.00000       1
30     35.600    1  35585.57600       1
…         …    …          …       …
1300   30.360    1  62592.87309       1
1301   30.875    1  46718.16325       1
1303   27.800    1  37829.72420       1
1313   34.700    0  36397.57600       1
1323   40.370    0  43896.37630       1

[133 rows x 4 columns]
```

1.4.3 Numeric Data Standardization

MinMaxScaler() MinMaxScaler transforms features by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one.

The transformation is given by: 1. Xstd = (X - Xmin(axis=0)) / (Xmax(axis=0) - Xmin(axis=0)) 2. Xscaled = Xstd * (max - min) + min

where min, max = feature_range.

This transformation is often used as an alternative to zero mean, unit variance scaling.

MinMaxScaler doesn't reduce the effect of outliers, but it linearly scales them down into a fixed range, where the largest occurring data point corresponds to the maximum value and the smallest one corresponds to the minimum value. For an example visualization, refer to Compare MinMaxScaler with other scalers.

[53]:
```
# standardization: with the MinMaxScaler() from sklearn

# set up the scaler
scaler = MinMaxScaler()

# fit the scaler to the train set, it will learn the parameters
scaler.fit(df_encoded[['age', 'bmi', 'charges']])

# transform train and test sets
MinMaxScaled = scaler.transform(df_encoded[['age', 'bmi', 'charges']])
```

[54]:
```
# the scaler stores the min of the features, learned from train set
scaler.data_min_
```

[54]:
```
array([  18.     ,   15.96   , 1121.8739])
```

[55]:
```
# the scaler stores the max of the features, learned from train set
scaler.data_max_
```

[55]:
```
array([6.4000000e+01, 5.3130000e+01, 6.3770428e+04])
```

[56]:
```
# access the range of each feature
scaler.data_range_
```

[56]:
```
array([4.60000000e+01, 3.71700000e+01, 6.26485541e+04])
```

[57]:
```
MinMaxScaled
```

[57]:
```
array([[0.02173913, 0.3212268 , 0.25161076],
       [0.        , 0.47914985, 0.00963595],
       [0.2173913 , 0.45843422, 0.05311516],
       …,
       [0.        , 0.56201238, 0.00810808],
       [0.06521739, 0.26472962, 0.01414352],
       [0.93478261, 0.35270379, 0.44724873]])
```

[58]:
```
df_encoded_MinMaxScaler = df_encoded.copy()
```

[59]:
```
# let's transform the returned NumPy arrays to dataframes for the rest of
# the demo
```

```
MinMaxScaled = pd.DataFrame(MinMaxScaled, columns=['age', 'bmi', 'charges'])
```

[61]:
```
# check contents of scaled columnss
MinMaxScaled.head(5)
```

[61]:

	age	bmi	charges
0	0.021739	0.321227	0.251611
1	0.000000	0.479150	0.009636
2	0.217391	0.458434	0.053115
3	0.326087	0.181464	0.333010
4	0.304348	0.347592	0.043816

[63]:
```
# replace the data in the df_encoded_MinMaxScaler variable using the data from␣
↪the MinMaxScaled variable
df_encoded_MinMaxScaler['age'] = MinMaxScaled['age']
df_encoded_MinMaxScaler['bmi'] = MinMaxScaled['bmi']
df_encoded_MinMaxScaler['charges'] = MinMaxScaled['charges']
```

[64]:
```
# check content of copy of dataset with scaled features
df_encoded_MinMaxScaler.head(5)
```

[64]:

	age	sex	bmi	children	smoker	region	charges
0	0.021739	0	0.321227	0	1	3	0.251611
1	0.000000	1	0.479150	1	0	2	0.009636
2	0.217391	1	0.458434	3	0	2	0.053115
3	0.326087	1	0.181464	0	0	1	0.333010
4	0.304348	1	0.347592	0	0	1	0.043816

[65]:
```
df_encoded_MinMaxScaler.to_csv('.//data//explore//
↪insurance_encoded_MinMaxScaler.csv', index=False)
```

[66]:
```
ds_encoded_MinMaxScaler_path = "D://programming//
↪information-technologies-of-smart-systems//calculation-and-graphic work//
↪personal-medical-insurance-cost-prediction//data//explore//
↪insurance_encoded_MinMaxScaler.csv"
```

[67]:
```
# is there such path?
if os.path.exists(ds_encoded_MinMaxScaler_path):
    print("File exists at the specified path.")
else:
    print("File does not exist at the specified path.")
```

```
File exists at the specified path.
```

[68]:
```
df_encoded_MinMaxScaler= pd.read_csv(ds_encoded_MinMaxScaler_path)
```

StandardScaler() Standardisation involves centering the variable at zero, and standardising the variance to 1. The procedure involves subtracting the mean of each observation and then dividing by the standard deviation:

z = (x - xmean) / std

The result of the above transformation is z, which is called the z-score, and represents how many standard deviations a given observation deviates from the mean. A z-score specifies the location of the observation within a distribution (in numbers of standard deviations respect to the mean of the distribution). The sign of the z-score (+ or -) indicates whether the observation is above (+) or below (-) the mean.

The shape of a standardised (or z-scored normalised) distribution will be identical to the original distribution of the variable. If the original distribution is normal, then the standardised distribution will be normal. But, if the original distribution is skewed, then the standardised distribution of the variable will also be skewed. In other words, standardising a variable does not normalize the distribution of the data and if this is the desired outcome, we should implement any of the techniques discussed in section 7 of the course.

[24]:
```
# standardization: with the StandardScaler() from sklearn

# set up the scaler
scaler = StandardScaler()

# fit the scaler to the train set, it will learn the parameters
scaler.fit(df_encoded[['age', 'bmi', 'charges']])

# transform train and test sets
StandardScaled = scaler.transform(df_encoded[['age', 'bmi', 'charges']])
```

[25]:
```
# the scaler stores the mean of the features, learned from train set
print("Mean of features:", scaler.mean_)

# the scaler stores the standard deviation deviation of the features,
# learned from train set
print("Standard deviation of features:", scaler.scale_)
```

```
Mean of features: [   39.20702541    30.66339686 13270.42226514]
Standard deviation of features: [1.40447090e+01 6.09590764e+00 1.21054850e+04]
```

[26]:
```
StandardScaled
```

[26]:
```
array([[-1.43876426, -0.45332   ,  0.2985838 ],
       [-1.50996545,  0.5096211 , -0.95368917],
       [-0.79795355,  0.38330685, -0.72867467],
       …,
       [-1.50996545,  1.0148781 , -0.96159623],
       [-1.29636188, -0.79781341, -0.93036151],
       [ 1.55168573, -0.26138796,  1.31105347]])
```

[44]:
```python
df_encoded_StandardScaler = df_encoded.copy()
```

[62]:
```python
df_encoded_StandardScaler.head(5)
```

[62]:

	age	sex	bmi	children	smoker	region	charges
0	-1.438764	0	-0.453320	0	1	3	0.298584
1	-1.509965	1	0.509621	1	0	2	-0.953689
2	-0.797954	1	0.383307	3	0	2	-0.728675
3	-0.441948	1	-1.305531	0	0	1	0.719843
4	-0.513149	1	-0.292556	0	0	1	-0.776802

[45]:
```python
# let's transform the returned NumPy arrays to dataframes
# for the rest of the demo

StandardScaled = pd.DataFrame(StandardScaled, columns=['age', 'bmi', 'charges'])
```

[46]:
```python
# check contents of scaled columns
StandardScaled.head(5)
```

[46]:

	age	bmi	charges
0	-1.438764	-0.453320	0.298584
1	-1.509965	0.509621	-0.953689
2	-0.797954	0.383307	-0.728675
3	-0.441948	-1.305531	0.719843
4	-0.513149	-0.292556	-0.776802

[47]:
```python
# replace the data in the df_encoded_StandardScaler variable using the data␣
↪from the StandardScaled variable
df_encoded_StandardScaler['age'] = StandardScaled['age']
df_encoded_StandardScaler['bmi'] = StandardScaled['bmi']
df_encoded_StandardScaler['charges'] = StandardScaled['charges']
```

[48]:
```python
# check content of copy of dataset with scaled features
df_encoded_StandardScaler.head(5)
```

[48]:

	age	sex	bmi	children	smoker	region	charges
0	-1.438764	0	-0.453320	0	1	3	0.298584
1	-1.509965	1	0.509621	1	0	2	-0.953689
2	-0.797954	1	0.383307	3	0	2	-0.728675
3	-0.441948	1	-1.305531	0	0	1	0.719843
4	-0.513149	1	-0.292556	0	0	1	-0.776802

[49]:
```python
df_encoded_StandardScaler.to_csv('.//data//explore//
↪insurance_encoded_StandardScaler.csv', index=False)
```

[50]:
```python

```

```
ds_encoded_StandardScaler_path = "D://programming//
↪information-technologies-of-smart-systems//calculation-and-graphic work//
↪personal-medical-insurance-cost-prediction//data//explore//
↪insurance_encoded_StandardScaler.csv"
```

[51]:
```
# is there such path?
if os.path.exists(ds_encoded_StandardScaler_path):
    print("File exists at the specified path.")
else:
    print("File does not exist at the specified path.")
```

```
File exists at the specified path.
```

For **Polynomial Regression**, feature scaling is often less critical compared to some other algorithms like SVMs or Neural Networks. However, it can still be beneficial in certain scenarios. Here's a brief overview of how feature scaling can affect Polynomial Regression:

StandardScaler: - StandardScaler scales features to have a mean of 0 and a standard deviation of 1. This can be useful if the features have different scales or units and you want to center them around zero. - Polynomial features can sometimes lead to multicollinearity, where features are highly correlated. Standardizing the features can mitigate multicollinearity to some extent.

MinMaxScaler: + MinMaxScaler scales features to a specified range, typically between 0 and 1. It preserves the relationships between the data points and can be useful if you want to bound the features within a specific range. + It can also help in cases where the polynomial features are bounded within a certain range and you want to maintain that range in the scaled data.

No Scaling: * In some cases, particularly if the features are already on similar scales or if the polynomial features are generated in a way that preserves the original scale, you may choose not to scale the features at all. * Additionally, if interpretability of the coefficients is important, you might prefer not to scale the features.

It's often a good idea to experiment with both scalers and evaluate their impact on the model's performance using cross-validation or other validation techniques.

1.5 Modelling

1.5.1 Regression

Regression in machine learning is a type of supervised learning task where the goal is to predict a continuous output variable based on one or more input features. In simpler terms, regression models try to find the relationship between independent variables (features) and dependent variables (target) and use this relationship to make predictions.

Key points about regression in machine learning: 1. **Continuous Output**: Unlike classification, where the output is a discrete label or category, regression predicts a continuous value. For example, predicting house prices, stock prices, temperature, or sales figures are all regression problems. 2. **Linear Regression**: One of the simplest and most commonly used regression techniques is linear regression, where the relationship between the input features and the target variable is modeled as a linear equation. The goal is to find the best-fitting line (or hyperplane in higher dimensions) that minimizes the difference between the actual and predicted values. 3. **Non-linear Regression**:

In many real-world scenarios, the relationship between the input features and the target variable may not be linear. In such cases, more complex regression techniques like polynomial regression, decision tree regression, random forest regression, support vector regression, or neural network regression can be used to capture non-linear patterns. 4. **Evaluation Metrics**: Regression models are evaluated using metrics that quantify the difference between the actual and predicted values. Common evaluation metrics for regression include Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), R-squared (coefficient of determination), and others. 5. **Overfitting and Underfitting**: Like other machine learning models, regression models can suffer from overfitting (capturing noise in the training data) or underfitting (failing to capture the underlying patterns). Techniques such as regularization, cross-validation, and feature selection can help mitigate these issues. 6. **Feature Engineering**: Feature engineering plays a crucial role in regression tasks. It involves selecting, transforming, or creating new features from the raw data to improve the model's predictive performance. Techniques like scaling, normalization, encoding categorical variables, handling missing values, and creating interaction terms can be applied to preprocess the data before training the regression model.

Overall, regression is a fundamental technique in machine learning that finds numerous applications in various domains such as finance, healthcare, marketing, and engineering for making predictions and understanding relationships between variables.

1.5.2 Baseline regression models with default hyperparameters

[72]:
```
# get list of all features
df_encoded.columns
```

[72]:
```
Index(['age', 'sex', 'bmi', 'children', 'smoker', 'region', 'charges'],
dtype='object')
```

[117]:
```
# Split the DataFrame into features (X) and target (y)
X = df_encoded[['age', 'sex', 'bmi', 'children', 'smoker', 'region']] # features

y = df_encoded['charges'] # target
```

[118]:
```
# Split the transformed data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[119]:
```
# initialize regression models with specific random states
lm = LinearRegression()
ridge = Ridge(random_state = 42)
lasso = Lasso(random_state = 42)
knn = KNeighborsRegressor()
rf = RandomForestRegressor(random_state = 42)
xgbt = xgb.XGBRegressor(random_state = 42)
dtree = DecisionTreeRegressor(random_state = 42)
gbr =  GradientBoostingRegressor(random_state = 42)
etr = ExtraTreeRegressor(random_state = 42)
```

```
# list of all regression models
algo = [xgbt, lm, ridge, lasso, knn, rf, dtree, gbr, etr]

result = []

for i in algo:
        start = time.process_time()
        # fit the model on the training data and calculate performance metrics
        ml_model = i.fit(X_train,y_train)

        result.append([str(i).split("(")[0] + str("_baseline"), ml_model.
 ↪score(X_train, y_train), ml_model.score(X_test, y_test),
                    np.sqrt(mean_squared_error(y_train, ml_model.
 ↪predict(X_train))),
                    np.sqrt(mean_squared_error(y_test, ml_model.predict(X_test))),
                    mean_absolute_error(y_train, ml_model.predict(X_train)),
                    mean_absolute_error(y_test, ml_model.predict(X_test)),
                    r2_score(y_train, ml_model.predict(X_train)),
                    r2_score(y_test, ml_model.predict(X_test))]),
        print(str(i).split("(")[0],"  \t", "{}".format(round(time.
 ↪process_time()-start,3)),"sec")

# create DataFrame from the result list and set the index as Algorithm
result = pd.DataFrame(result, columns = ["Algorithm", "Train_Score",␣
 ↪"Test_Score", "Train_Rmse",
                                          "Test_Rmse", "Train_Mae", "Test_Mae",␣
 ↪"Train_R2", "Test_R2"]).sort_values("Test_Rmse").set_index("Algorithm")
result
```

```
XGBRegressor     1.078 sec
LinearRegression         0.0 sec
Ridge            0.172 sec
Lasso            0.156 sec
KNeighborsRegressor      0.328 sec
RandomForestRegressor            0.688 sec
DecisionTreeRegressor            0.016 sec
GradientBoostingRegressor        0.109 sec
ExtraTreeRegressor       0.016 sec
```

[119]:

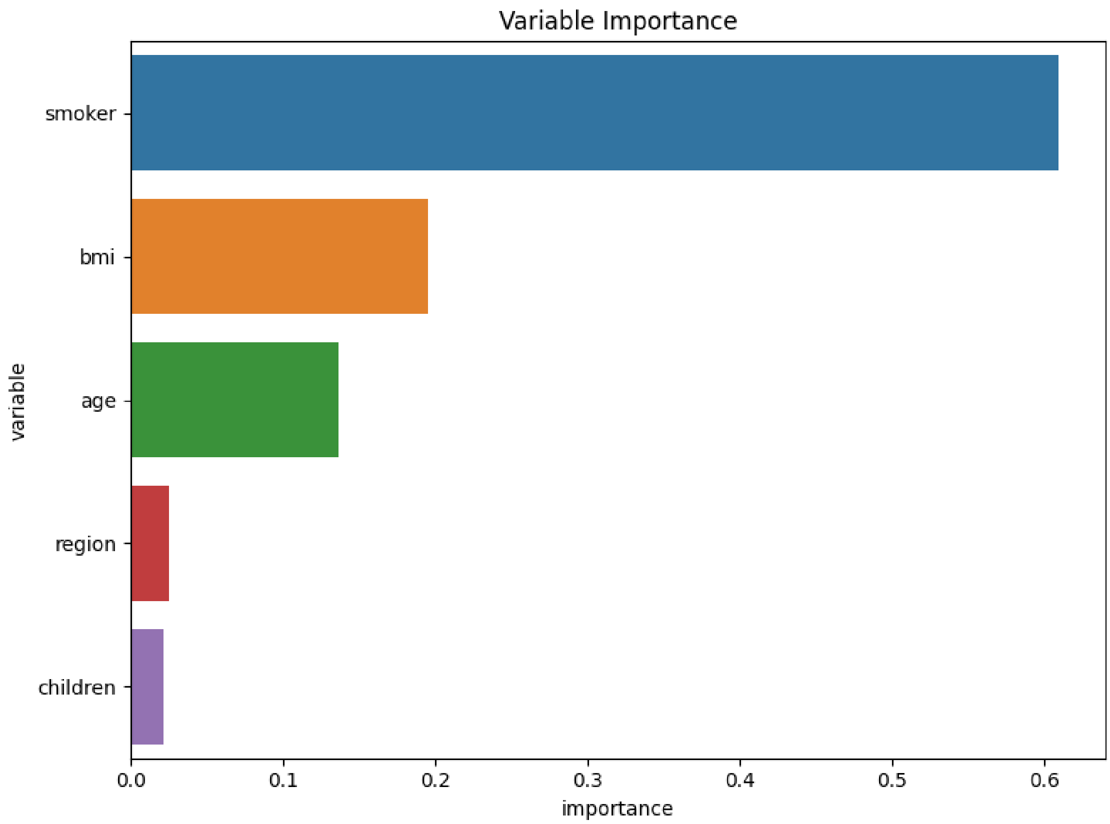
Algorithm	Train_Score	Test_Score	Train_Rmse	\
GradientBoostingRegressor_baseline	0.898046	0.877973	3836.065033	
RandomForestRegressor_baseline	0.974309	0.864261	1925.624128	
XGBRegressor_baseline	0.994139	0.850168	919.765561	
LinearRegression_baseline	0.741705	0.783346	6105.789320	
Lasso_baseline	0.741705	0.783320	6105.790345	
Ridge_baseline	0.741684	0.783085	6106.033325	

```
DecisionTreeRegressor_baseline          0.998308    0.684357    494.205984
ExtraTreeRegressor_baseline             0.998308    0.671399    494.205984
KNeighborsRegressor_baseline            0.393768    0.144504   9354.125544

                                       Test_Rmse     Train_Mae      Test_Mae  \
Algorithm
GradientBoostingRegressor_baseline   4352.538932  2101.361701  2447.951558
RandomForestRegressor_baseline       4590.573539  1053.588715  2533.674644
XGBRegressor_baseline                4822.991168   499.339156  2791.832518
LinearRegression_baseline            5799.587091  4208.762029  4186.508898
Lasso_baseline                       5799.943043  4209.135074  4187.244900
Ridge_baseline                       5803.084710  4218.431488  4198.141005
DecisionTreeRegressor_baseline       7000.231682    29.572515  3154.705669
ExtraTreeRegressor_baseline          7142.470320    29.572515  3310.218896
KNeighborsRegressor_baseline        11524.523708  6491.688549  7953.210498

                                    Train_R2   Test_R2
Algorithm
GradientBoostingRegressor_baseline  0.898046  0.877973
RandomForestRegressor_baseline      0.974309  0.864261
XGBRegressor_baseline               0.994139  0.850168
LinearRegression_baseline           0.741705  0.783346
Lasso_baseline                      0.741705  0.783320
Ridge_baseline                      0.741684  0.783085
DecisionTreeRegressor_baseline      0.998308  0.684357
ExtraTreeRegressor_baseline         0.998308  0.671399
KNeighborsRegressor_baseline        0.393768  0.144504
```

Feature importance

[ ]:
```
rankings = ml_model.feature_importances_.tolist()
```

[120]:
```
rankings = ml_model.feature_importances_.tolist()
importance = pd.DataFrame(sorted(zip(X_train.
 ↪columns,rankings),reverse=True),columns=["variable","importance"]).
 ↪sort_values("importance",ascending = False)

plt.figure(figsize=(8,6))
sns.barplot(x="importance",
            y="variable",
            data=importance[:5])
plt.title('Variable Importance')
plt.tight_layout()
```



1.5.3 Baseline regression models *with PolynomialFeatures* matrix of 2

[141]:
```
# Split the DataFrame into features (X) and target (y)
X = df_encoded[['age', 'sex', 'bmi', 'children', 'smoker', 'region']] # features

y = df_encoded['charges'] # target
```

[142]:
```
# creating an instance of PolynomialFeatures with degree 2
quad = PolynomialFeatures (degree = 2)
# transforming the input features X into quadratic polynomial features
# this generates new features that are combinations of the original features up
 ↪to degree 2
x_quad = quad.fit_transform(X)
```

[143]:
```
# Split the transformed data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(x_quad, y, test_size=0.2,
 ↪random_state=42)
```

[144]:
```
# initialize regression models with specific random states
lm = LinearRegression()
```

```
ridge = Ridge(random_state = 42)
lasso = Lasso(random_state = 42)
knn = KNeighborsRegressor()
rf = RandomForestRegressor(random_state = 42)
xgbt = xgb.XGBRegressor(random_state = 42)
dtree = DecisionTreeRegressor(random_state = 42)
gbr =  GradientBoostingRegressor(random_state = 42)
etr = ExtraTreeRegressor(random_state = 42)

# list of all regression models
algo = [xgbt, lm, ridge, lasso, knn, rf, dtree, gbr, etr]

result = []

for i in algo:
        start = time.process_time()
        # fit the model on the training data and calculate performance metrics
        ml_model = i.fit(X_train,y_train)

        result.append([str(i).split("(")[0] + str("_baseline"), ml_model.
 ↪score(X_train, y_train), ml_model.score(X_test, y_test),
                  np.sqrt(mean_squared_error(y_train, ml_model.
 ↪predict(X_train))),
                  np.sqrt(mean_squared_error(y_test, ml_model.predict(X_test))),
                  mean_absolute_error(y_train, ml_model.predict(X_train)),
                  mean_absolute_error(y_test, ml_model.predict(X_test)),
                  r2_score(y_train, ml_model.predict(X_train)),
                  r2_score(y_test, ml_model.predict(X_test))]),
        print(str(i).split("(")[0],"  \t", "{}".format(round(time.
 ↪process_time()-start,3)),"sec")

# create DataFrame from the result list and set the index as Algorithm
result = pd.DataFrame(result, columns = ["Algorithm", "Train_Score",␣
 ↪"Test_Score", "Train_Rmse",
                                         "Test_Rmse", "Train_Mae", "Test_Mae",␣
 ↪"Train_R2", "Test_R2"]).sort_values("Test_Rmse").set_index("Algorithm")
result
```

```
XGBRegressor     1.312 sec
LinearRegression         0.0 sec
Ridge            0.156 sec
Lasso            0.172 sec
KNeighborsRegressor      0.406 sec
RandomForestRegressor            1.422 sec
DecisionTreeRegressor            0.016 sec
GradientBoostingRegressor        0.312 sec
ExtraTreeRegressor       0.016 sec
```

[144]:

```
                                   Train_Score  Test_Score   Train_Rmse  \
Algorithm
GradientBoostingRegressor_baseline    0.913613    0.882020  3531.079729
RandomForestRegressor_baseline        0.974581    0.869866  1915.426618
Ridge_baseline                        0.840397    0.868124  4799.589844
Lasso_baseline                        0.840507    0.867811  4797.941643
LinearRegression_baseline             0.837402    0.864922  4844.420432
XGBRegressor_baseline                 0.996256    0.852146   735.134486
DecisionTreeRegressor_baseline        0.998308    0.801469   494.205984
ExtraTreeRegressor_baseline           0.998308    0.744335   494.205984
KNeighborsRegressor_baseline          0.346751    0.119860  9710.090728

                                      Test_Rmse    Train_Mae     Test_Mae  \
Algorithm
GradientBoostingRegressor_baseline   4279.741000  1957.139704  2409.996822
RandomForestRegressor_baseline       4494.788638  1053.655276  2413.199889
Ridge_baseline                       4524.765957  2921.514403  2732.520621
Lasso_baseline                       4530.143241  2925.518663  2726.472040
LinearRegression_baseline            4579.378770  3035.118503  2876.026334
XGBRegressor_baseline                4791.052846   347.089509  2635.593164
DecisionTreeRegressor_baseline       5551.726579    29.572515  2308.392866
ExtraTreeRegressor_baseline          6300.129213    29.572515  2829.846888
KNeighborsRegressor_baseline        11689.335127  6799.720735  8219.705903

                                   Train_R2   Test_R2
Algorithm
GradientBoostingRegressor_baseline  0.913613  0.882020
RandomForestRegressor_baseline      0.974581  0.869866
Ridge_baseline                      0.840397  0.868124
Lasso_baseline                      0.840507  0.867811
LinearRegression_baseline           0.837402  0.864922
XGBRegressor_baseline               0.996256  0.852146
DecisionTreeRegressor_baseline      0.998308  0.801469
ExtraTreeRegressor_baseline         0.998308  0.744335
KNeighborsRegressor_baseline        0.346751  0.119860
```

1.5.4 Baseline regression models with *MinMaxScaler* data without PolynomialFeatures

[90]:

```
# Split the DataFrame into features (X) and target (y)
X = df_encoded_MinMaxScaler[['age', 'sex', 'bmi', 'children', 'smoker',␣
↪'region']] # features

y = df_encoded_MinMaxScaler['charges'] # target
```

[91]:

```
# Split the transformed data into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[92]:
```
# initialize regression models with specific random states
lm = LinearRegression()
ridge = Ridge(random_state = 42)
lasso = Lasso(random_state = 42)
knn = KNeighborsRegressor()
rf = RandomForestRegressor(random_state = 42)
xgbt = xgb.XGBRegressor(random_state = 42)
dtree = DecisionTreeRegressor(random_state = 42)
gbr =  GradientBoostingRegressor(random_state = 42)
etr = ExtraTreeRegressor(random_state = 42)
#rnr = RadiusNeighborsRegressor(random_state = 42)
# svr = SVR(random_state = 42)
gpr = GaussianProcessRegressor(random_state = 42)

# list of all regression models
algo = [xgbt, lm, ridge, lasso, knn, rf, dtree, gbr, etr, gpr]

result = []

for i in algo:
        start = time.process_time()
        # fit the model on the training data and calculate performance metrics
        ml_model = i.fit(X_train,y_train)

        result.append([str(i).split("(")[0] + str("_baseline"), ml_model.
↪score(X_train, y_train), ml_model.score(X_test, y_test),
                  np.sqrt(mean_squared_error(y_train, ml_model.
↪predict(X_train))),
                  np.sqrt(mean_squared_error(y_test, ml_model.predict(X_test))),
                  mean_absolute_error(y_train, ml_model.predict(X_train)),
                  mean_absolute_error(y_test, ml_model.predict(X_test)),
                  r2_score(y_train, ml_model.predict(X_train)),
                  r2_score(y_test, ml_model.predict(X_test))]),
        print(str(i).split("(")[0],"  \t", "{}".format(round(time.
↪process_time()-start,3)),"sec")

# create DataFrame from the result list and set the index as Algorithm
result = pd.DataFrame(result, columns = ["Algorithm", "Train_Score",␣
↪"Test_Score", "Train_Rmse",
                                          "Test_Rmse", "Train_Mae", "Test_Mae",␣
↪"Train_R2", "Test_R2"]).sort_values("Test_Rmse").set_index("Algorithm")
result
```

```
XGBRegressor     0.875 sec
```

```
LinearRegression         0.172 sec
Ridge            0.141 sec
Lasso            0.172 sec
KNeighborsRegressor      0.516 sec
RandomForestRegressor            0.516 sec
DecisionTreeRegressor            0.016 sec
GradientBoostingRegressor        0.109 sec
ExtraTreeRegressor       0.016 sec
GaussianProcessRegressor         1.016 sec
```

[92]:

Algorithm	Train_Score	Test_Score	Train_Rmse	Test_Rmse	Train_Mae	Test_Mae	Train_R2	Test_R2
GradientBoostingRegressor_baseline	0.898046	0.877973	0.061232	0.069475	0.033542	0.039074	0.898046	0.877973
RandomForestRegressor_baseline	0.974440	0.863972	0.030658	0.073353	0.016765	0.040546	0.974440	0.863972
XGBRegressor_baseline	0.993995	0.852686	0.014861	0.076335	0.007866	0.043728	0.993995	0.852686
LinearRegression_baseline	0.741705	0.783346	0.097461	0.092573	0.067181	0.066825	0.741705	0.783346
Ridge_baseline	0.741643	0.783118	0.097473	0.092622	0.067076	0.066686	0.741643	0.783118
ExtraTreeRegressor_baseline	0.998308	0.771167	0.007889	0.095140	0.000472	0.042052	0.998308	0.771167
KNeighborsRegressor_baseline	0.803382	0.735285	0.085032	0.102328	0.051107	0.063950	0.803382	0.735285
DecisionTreeRegressor_baseline	0.998308	0.727382	0.007889	0.103844	0.000472	0.048087	0.998308	0.727382
Lasso_baseline	0.000000	-0.000919	0.191766	0.198977	0.143667	0.153129	0.000000	-0.000919
GaussianProcessRegressor_baseline	0.976524	-55.113747	0.029382	1.489833	0.013307	0.578576	0.976524	

```
GaussianProcessRegressor_baseline  -55.113747
```

1.5.5 Baseline regression models with *StandardScaler* data without PolynomialFeatures

[133]:
```
# Split the DataFrame into features (X) and target (y)
X = df_encoded_StandardScaler[['age', 'sex', 'bmi', 'children', 'smoker',␣
↪'region']] # features

y = df_encoded_StandardScaler['charges'] # target
```

[ ]:
```
# Split the transformed data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[135]:
```
# initialize regression models with specific random states
lm = LinearRegression()
ridge = Ridge(random_state = 42)
lasso = Lasso(random_state = 42)
knn = KNeighborsRegressor()
rf = RandomForestRegressor(random_state = 42)
xgbt = xgb.XGBRegressor(random_state = 42)
dtree = DecisionTreeRegressor(random_state = 42)
gbr =  GradientBoostingRegressor(random_state = 42)
etr = ExtraTreeRegressor(random_state = 42)
#rnr = RadiusNeighborsRegressor(random_state = 42)
# svr = SVR(random_state = 42)
gpr = GaussianProcessRegressor(random_state = 42)

# list of all regression models
algo = [xgbt, lm, ridge, lasso, knn, rf, dtree, gbr, etr, gpr]

result = []

for i in algo:
        start = time.process_time()
        # fit the model on the training data and calculate performance metrics
        ml_model = i.fit(X_train,y_train)

        result.append([str(i).split("(")[0] + str("_baseline"), ml_model.
↪score(X_train, y_train), ml_model.score(X_test, y_test),
                    np.sqrt(mean_squared_error(y_train, ml_model.
↪predict(X_train))),
                    np.sqrt(mean_squared_error(y_test, ml_model.predict(X_test))),
                    mean_absolute_error(y_train, ml_model.predict(X_train)),
                    mean_absolute_error(y_test, ml_model.predict(X_test)),
                    r2_score(y_train, ml_model.predict(X_train)),
```

```
                  r2_score(y_test, ml_model.predict(X_test))]),
        print(str(i).split("(")[0]," \t", "{}".format(round(time.
 ↪process_time()-start,3)),"sec")

# create DataFrame from the result list and set the index as Algorithm
result = pd.DataFrame(result, columns = ["Algorithm", "Train_Score",␣
 ↪"Test_Score", "Train_Rmse",
                                          "Test_Rmse", "Train_Mae", "Test_Mae",␣
 ↪"Train_R2", "Test_R2"]).sort_values("Test_Rmse").set_index("Algorithm")
result
```

```
XGBRegressor     1.078 sec
LinearRegression         0.0 sec
Ridge            0.172 sec
Lasso            0.172 sec
KNeighborsRegressor      0.438 sec
RandomForestRegressor            0.516 sec
DecisionTreeRegressor            0.031 sec
GradientBoostingRegressor        0.156 sec
ExtraTreeRegressor       0.016 sec
GaussianProcessRegressor         0.922 sec
```

[135]:

```
                                   Train_Score    Test_Score  Train_Rmse  \
Algorithm
GradientBoostingRegressor_baseline     0.898046      0.877550    0.316887
RandomForestRegressor_baseline         0.974276      0.865119    0.159173
XGBRegressor_baseline                  0.994077      0.850465    0.076379
LinearRegression_baseline              0.741705      0.783346    0.504382
Ridge_baseline                         0.741684      0.783085    0.504402
KNeighborsRegressor_baseline           0.795118      0.710149    0.449214
DecisionTreeRegressor_baseline         0.998308      0.709617    0.040825
ExtraTreeRegressor_baseline            0.998308      0.669776    0.040825
Lasso_baseline                         0.000000     -0.000919    0.992434
GaussianProcessRegressor_baseline      0.997850 -12424.387271    0.046013

                                    Test_Rmse  Train_Mae   Test_Mae  \
Algorithm
GradientBoostingRegressor_baseline   0.360173   0.173588   0.202335
RandomForestRegressor_baseline       0.378014   0.087103   0.208042
XGBRegressor_baseline                0.398018   0.041398   0.230354
LinearRegression_baseline            0.479088   0.347674   0.345836
Ridge_baseline                       0.479376   0.348439   0.346760
KNeighborsRegressor_baseline         0.554139   0.269205   0.322244
DecisionTreeRegressor_baseline       0.554648   0.002443   0.246004
ExtraTreeRegressor_baseline          0.591474   0.002443   0.274660
Lasso_baseline                       1.029749   0.743511   0.792479
GaussianProcessRegressor_baseline  114.732592   0.003875  32.749994
```

```
                                   Train_R2       Test_R2
Algorithm
GradientBoostingRegressor_baseline  0.898046      0.877550
RandomForestRegressor_baseline      0.974276      0.865119
XGBRegressor_baseline               0.994077      0.850465
LinearRegression_baseline           0.741705      0.783346
Ridge_baseline                      0.741684      0.783085
KNeighborsRegressor_baseline        0.795118      0.710149
DecisionTreeRegressor_baseline      0.998308      0.709617
ExtraTreeRegressor_baseline         0.998308      0.669776
Lasso_baseline                      0.000000     -0.000919
GaussianProcessRegressor_baseline   0.997850 -12424.387271
```

1.5.6 Baseline regression models without *'children'* feature and with PolynomialFeatures

[154]:
```
# deleting feature 'children'
X = df_encoded.drop(['charges','children'], axis = 1)
Y = df_encoded.charges  # target

# creating an instance of PolynomialFeatures with degree 2
quad = PolynomialFeatures (degree = 2)
# transforming the input features X into quadratic polynomial features
# this generates new features that are combinations of the original features up␣
↪to degree 2
x_quad = quad.fit_transform(X)

X_train,X_test,Y_train,Y_test = train_test_split(x_quad, Y, test_size=0.2,␣
↪random_state=42)

lm = LinearRegression()
ridge = Ridge(random_state = 42)
lasso = Lasso(random_state = 42)
knn = KNeighborsRegressor()
rf = RandomForestRegressor(random_state = 42)
xgbt = xgb.XGBRegressor(random_state = 42)
dtree = DecisionTreeRegressor(random_state = 42)
gbr =  GradientBoostingRegressor(random_state = 42)
etr = ExtraTreeRegressor(random_state = 42)

# list of all regression models
algo = [xgbt, lm, ridge, lasso, knn, rf, dtree, gbr, etr]

result = []

for i in algo:
```

```
        start = time.process_time()
        # fit the model on the training data and calculate performance metrics
        ml_model = i.fit(X_train,y_train)

        result.append([str(i).split("(")[0] + str("_baseline"), ml_model.
 ↪score(X_train, y_train), ml_model.score(X_test, y_test),
                    np.sqrt(mean_squared_error(y_train, ml_model.
 ↪predict(X_train))),
                    np.sqrt(mean_squared_error(y_test, ml_model.predict(X_test))),
                    mean_absolute_error(y_train, ml_model.predict(X_train)),
                    mean_absolute_error(y_test, ml_model.predict(X_test)),
                    r2_score(y_train, ml_model.predict(X_train)),
                    r2_score(y_test, ml_model.predict(X_test))]),
        print(str(i).split("(")[0],"  \t", "{}".format(round(time.
 ↪process_time()-start,3)),"sec")

# create DataFrame from the result list and set the index as Algorithm
result = pd.DataFrame(result, columns = ["Algorithm", "Train_Score",␣
 ↪"Test_Score", "Train_Rmse",
                                         "Test_Rmse", "Train_Mae", "Test_Mae",␣
 ↪"Train_R2", "Test_R2"]).sort_values("Test_Rmse").set_index("Algorithm")
result
```

```
RandomForestRegressor           0.781 sec
GradientBoostingRegressor       0.281 sec
LinearRegression        0.0 sec
```

[154]:

Algorithm	Train_Score	Test_Score	Train_Rmse	Test_Rmse	Train_Mae	Test_Mae	Train_R2	Test_R2
GradientBoostingRegressor_baseline	0.906376	0.873384	3676.018847	4433.625698	2083.829667	2519.561704	0.906376	0.873384
LinearRegression_baseline	0.835682	0.862207	4869.972761	4625.173407	2983.118678	2823.290299	0.835682	0.862207
RandomForestRegressor_baseline	0.973032	0.856412	1972.897358	4721.430221	1108.386813	2657.723463	0.973032	0.856412

1.5.7 Baseline regression models without *'region'* feature and with PolynomialFeatures

[169]:
```
# deleting feature 'region'
X = df_encoded.drop(['charges','region'], axis = 1)
Y = df_encoded.charges

# creating an instance of PolynomialFeatures with degree 2
quad = PolynomialFeatures (degree = 2)
# transforming the input features X into quadratic polynomial features
# this generates new features that are combinations of the original features up␣
↪to degree 2
x_quad = quad.fit_transform(X)

X_train,X_test,Y_train,Y_test = train_test_split(x_quad, Y, test_size=0.2,␣
↪random_state=42)

lm = LinearRegression()
ridge = Ridge(random_state = 42)
lasso = Lasso(random_state = 42)
knn = KNeighborsRegressor()
rf = RandomForestRegressor(random_state = 42)
xgbt = xgb.XGBRegressor(random_state = 42)
dtree = DecisionTreeRegressor(random_state = 42)
gbr =  GradientBoostingRegressor(random_state = 42)
etr = ExtraTreeRegressor(random_state = 42)

# list of all regression models
algo = [xgbt, lm, ridge, lasso, knn, rf, dtree, gbr, etr]

# list for results of previous best models
result = []

for i in algo:
        start = time.process_time()
        # fit the model on the training data and calculate performance metrics
        ml_model = i.fit(X_train,y_train)

        result.append([str(i).split("(")[0] + str("_baseline"), ml_model.
↪score(X_train, y_train), ml_model.score(X_test, y_test),
                    np.sqrt(mean_squared_error(y_train, ml_model.
↪predict(X_train))),
                    np.sqrt(mean_squared_error(y_test, ml_model.predict(X_test))),
                    mean_absolute_error(y_train, ml_model.predict(X_train)),
                    mean_absolute_error(y_test, ml_model.predict(X_test)),
                    r2_score(y_train, ml_model.predict(X_train)),
                    r2_score(y_test, ml_model.predict(X_test))]),
```

```
        print(str(i).split("(")[0],"  \t", "{}".format(round(time.
 ↪process_time()-start,3)),"sec")

# create DataFrame from the result list and set the index as Algorithm
result = pd.DataFrame(result, columns = ["Algorithm", "Train_Score",␣
 ↪"Test_Score", "Train_Rmse",
                                         "Test_Rmse", "Train_Mae", "Test_Mae",␣
 ↪"Train_R2", "Test_R2"]).sort_values("Test_Rmse").set_index("Algorithm")
result
```

```
XGBRegressor     1.016 sec
LinearRegression         0.0 sec
Ridge            0.125 sec
Lasso            0.172 sec
KNeighborsRegressor      0.391 sec
RandomForestRegressor            1.188 sec
DecisionTreeRegressor            0.016 sec
GradientBoostingRegressor        0.25 sec
ExtraTreeRegressor       0.016 sec
```

[169]:

```
                                   Train_Score  Test_Score   Train_Rmse  \
Algorithm
GradientBoostingRegressor_baseline     0.909564    0.884762  3612.890453
Ridge_baseline                         0.837345    0.867382  4845.256885
Lasso_baseline                         0.837453    0.866962  4843.651407
LinearRegression_baseline              0.837456    0.866944  4843.610528
RandomForestRegressor_baseline         0.972708    0.860197  1984.746067
XGBRegressor_baseline                  0.995600    0.840341   796.868123
DecisionTreeRegressor_baseline         0.998308    0.755805   494.205984
ExtraTreeRegressor_baseline            0.998308    0.729844   494.205984
KNeighborsRegressor_baseline           0.393287    0.157401  9357.834084

                                     Test_Rmse    Train_Mae     Test_Mae  \
Algorithm
GradientBoostingRegressor_baseline   4229.719923  1994.232121  2402.180667
Ridge_baseline                       4537.480504  2941.739688  2781.345595
Lasso_baseline                       4544.673041  2945.426050  2779.964683
LinearRegression_baseline            4544.969862  2946.198677  2783.356805
RandomForestRegressor_baseline       4658.787101  1082.612897  2455.683798
XGBRegressor_baseline                4978.639363   395.158384  2720.287917
DecisionTreeRegressor_baseline       6157.183038    29.572515  2694.995542
ExtraTreeRegressor_baseline          6476.221024    29.572515  3028.218157
KNeighborsRegressor_baseline        11437.329081  6543.612219  8062.165955

                                   Train_R2   Test_R2
Algorithm
GradientBoostingRegressor_baseline  0.909564  0.884762
```

```
Ridge_baseline                    0.837345  0.867382
Lasso_baseline                    0.837453  0.866962
LinearRegression_baseline         0.837456  0.866944
RandomForestRegressor_baseline    0.972708  0.860197
XGBRegressor_baseline             0.995600  0.840341
DecisionTreeRegressor_baseline    0.998308  0.755805
ExtraTreeRegressor_baseline       0.998308  0.729844
KNeighborsRegressor_baseline      0.393287  0.157401
```

1.5.8 Final model choice

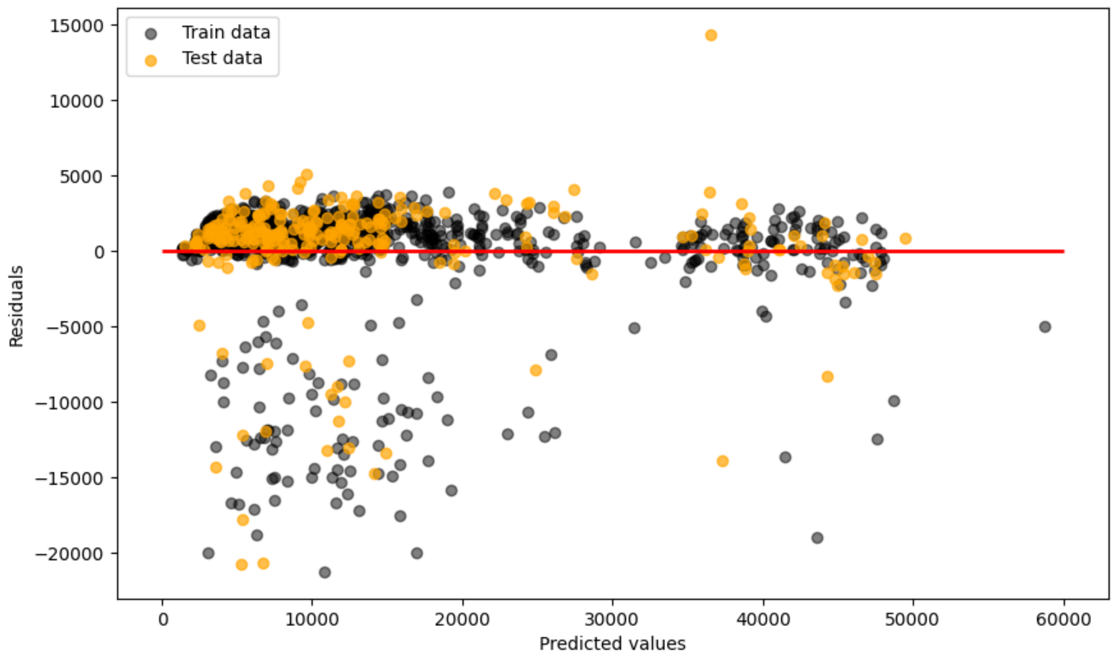
[23]:
```
X = df_encoded.drop(['charges','region'], axis = 1)
Y = df_encoded.charges

# quad = PolynomialFeatures (degree = 2)
# x_quad = quad.fit_transform(X)

X_train,X_test,Y_train,Y_test = train_test_split(X,Y, test_size=0.2,␣
↪random_state = 0)

plr = GradientBoostingRegressor(random_state = 42).fit(X_train,Y_train)

Y_train_pred = plr.predict(X_train)
Y_test_pred = plr.predict(X_test)

# Calculating metrics
score_train = plr.score(X_train, Y_train)
score_test = plr.score(X_test, Y_test)

mse_train = mean_squared_error(Y_train, Y_train_pred)
mse_test = mean_squared_error(Y_test, Y_test_pred)

mae_train = mean_absolute_error(Y_train, Y_train_pred)
mae_test = mean_absolute_error(Y_test, Y_test_pred)

r2_train = r2_score(Y_train, Y_train_pred)
r2_test = r2_score(Y_test, Y_test_pred)

# Function to format score_train in desired pattern
def format_score(score):
    return "{:.6f}".format(score)

# Displaying metrics in a grid
metrics_df = pd.DataFrame({
    'Metric': ['Score', 'MSE', 'MAE', 'R2'],
    'Train Data': [format_score(score_train), mse_train, mae_train, r2_train],
    'Test Data': [format_score(score_test), mse_test, mae_test, r2_test]
```

```
})

print(metrics_df)
```

```
  Metric        Train Data         Test Data
0  Score          0.895655          0.897123
1    MSE  14959484.462928  16370815.053277
2    MAE       2123.736738        2366.20557
3     R2          0.895655          0.897123
```

Scatter plot of predicted values versus residuals (the difference between predicted and actual values) for both the training and test data sets

[19]:
```
# Set the figure size
plt.figure(figsize=(10,6))

# Plot the residuals for the training data
plt.scatter(Y_train_pred, Y_train_pred - Y_train,
            c='black', marker='o', s=35, alpha=0.5,
            label='Train data')

# Plot the residuals for the test data
plt.scatter(Y_test_pred, Y_test_pred - Y_test,
            c='orange', marker='o', s=35, alpha=0.7,
            label='Test data')

# Label the x and y axes
plt.xlabel('Predicted values')
plt.ylabel('Residuals')

# Add a legend
plt.legend(loc='upper left')

# Add a horizontal line at y=0 for reference
plt.hlines(y=0, xmin=0, xmax=60000, lw=2, color='red')

# Show the plot
plt.show()
```



Feature importance for Gradient Boosting Regressor

[ ]:
```
rankings = plr.feature_importances_.tolist()
importance = pd.DataFrame(sorted(zip(X_train.
 ↪columns,rankings),reverse=True),columns=["variable", "importance"]).
 ↪sort_values("importance",ascending = False)

plt.figure(figsize=(8,6))
sns.barplot(x="importance",
            y="variable",
            data=importance[:5])
plt.title('Variable Importance')
plt.tight_layout()
```

1.5.9 Cross-validation technics

K-fold validation When evaluating different settings ("hyperparameters") for estimators, such as the CV setting that must be manually set for an SVM, there is still a risk of overfitting on the test set because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set can "leak" into the model and evaluation metrics no longer report on generalization performance. To solve this problem, yet another part of the dataset can be held out as a so-called "validation set": training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

A solution to this problem is a procedure called cross-validation (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, called k-fold CV, the training set is split into k smaller sets (other approaches are described below, but generally follow the same principles). The following procedure is followed for each of the k "folds":

[187]:
```
# Define feature matrix X and target variable Y
X = df_encoded.drop(['charges', 'region'], axis=1)
Y = df_encoded['charges']

# Transform features using polynomial features if necessary
quad = PolynomialFeatures(degree=2)
X_quad = quad.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, Y_train, Y_test = train_test_split(X_quad, Y, test_size=0.2,␣
 ↪random_state=0)

# Initialize the model
model = GradientBoostingRegressor(random_state=42)
# RandomForestRegressor(random_state = 42)
# LinearRegression()
# Ridge(random_state = 42)
# GradientBoostingRegressor(random_state=42)
# Lasso(random_state = 42)

# Perform k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
train_scores, test_scores, train_mse, test_mse, train_mae, test_mae, train_r2,␣
 ↪test_r2 = [], [], [], [], [], [], [], []

for train_index, test_index in kf.split(X_quad):
    X_train, X_test = X_quad[train_index], X_quad[test_index]
    Y_train, Y_test = Y.iloc[train_index], Y.iloc[test_index]

    model.fit(X_train, Y_train)

    Y_train_pred = model.predict(X_train)
    Y_test_pred = model.predict(X_test)

    train_scores.append(model.score(X_train, Y_train))
    test_scores.append(model.score(X_test, Y_test))

    train_mse.append(mean_squared_error(Y_train, Y_train_pred))
    test_mse.append(mean_squared_error(Y_test, Y_test_pred))
```

```
    train_mae.append(mean_absolute_error(Y_train, Y_train_pred))
    test_mae.append(mean_absolute_error(Y_test, Y_test_pred))

    train_r2.append(r2_score(Y_train, Y_train_pred))
    test_r2.append(r2_score(Y_test, Y_test_pred))

# Calculate average performance metrics across all folds
avg_train_score = np.mean(train_scores)
avg_test_score = np.mean(test_scores)

avg_train_mse = np.mean(train_mse)
avg_test_mse = np.mean(test_mse)

avg_train_mae = np.mean(train_mae)
avg_test_mae = np.mean(test_mae)

avg_train_r2 = np.mean(train_r2)
avg_test_r2 = np.mean(test_r2)

# Print the results
print('Average Score train data: %.5f, Average Score test data: %.5f' %
 ↪(avg_train_score, avg_test_score))
print('Average MSE train data: %.5f, Average MSE test data: %.5f' %
 ↪(avg_train_mse, avg_test_mse))
print('Average MAE train data: %.5f, Average MAE test data: %.5f' %
 ↪(avg_train_mae, avg_test_mae))
print('Average R2 train data: %.5f, Average R2 test data: %.5f' %
 ↪(avg_train_r2, avg_test_r2))
```

```
Average Score train data: 0.91739, Average Score test data: 0.85188
Average MSE train data: 12083975.90761, Average MSE test data: 21101318.40206
Average MAE train data: 1891.18876, Average MAE test data: 2524.18689
Average R2 train data: 0.91739, Average R2 test data: 0.85188
```

Random permutations cross-validation The ShuffleSplit iterator will generate a user defined number of independent train / test dataset splits. Samples are first shuffled and then split into a pair of train and test sets.

It is possible to control the randomness for reproducibility of the results by explicitly seeding the random_state pseudo random number generator.

[194]:
```
# Define your DataFrame and preprocessing steps
X = df_encoded.drop(['charges','region'], axis=1)
y = df_encoded['charges']

# Initialize ShuffleSplit with desired parameters
```

```python
shuffle_split = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)

# Initialize lists to store evaluation metrics
train_scores, test_scores, train_mse, test_mse, train_mae, test_mae, train_r2, test_r2 = [], [], [], [], [], [], [], []

# Iterate through ShuffleSplit splits
for train_index, test_index in shuffle_split.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Initialize and fit model
    plr = GradientBoostingRegressor(random_state=42).fit(X_train, y_train)

    # Predict on training and test sets
    y_train_pred = plr.predict(X_train)
    y_test_pred = plr.predict(X_test)

    # Calculate evaluation metrics
    train_score = plr.score(X_train, y_train)
    test_score = plr.score(X_test, y_test)
    train_mse_score = mean_squared_error(y_train, y_train_pred)
    test_mse_score = mean_squared_error(y_test, y_test_pred)
    train_mae_score = mean_absolute_error(y_train, y_train_pred)
    test_mae_score = mean_absolute_error(y_test, y_test_pred)
    train_r2_score = r2_score(y_train, y_train_pred)
    test_r2_score = r2_score(y_test, y_test_pred)

    # Append metrics to lists
    train_scores.append(train_score)
    test_scores.append(test_score)
    train_mse.append(train_mse_score)
    test_mse.append(test_mse_score)
    train_mae.append(train_mae_score)
    test_mae.append(test_mae_score)
    train_r2.append(train_r2_score)
    test_r2.append(test_r2_score)

# Calculate mean and standard deviation of evaluation metrics
mean_train_score = np.mean(train_scores)
mean_test_score = np.mean(test_scores)
mean_train_mse = np.mean(train_mse)
mean_test_mse = np.mean(test_mse)
mean_train_mae = np.mean(train_mae)
mean_test_mae = np.mean(test_mae)
mean_train_r2 = np.mean(train_r2)
mean_test_r2 = np.mean(test_r2)
```

```python
std_train_score = np.std(train_scores)
std_test_score = np.std(test_scores)
std_train_mse = np.std(train_mse)
std_test_mse = np.std(test_mse)
std_train_mae = np.std(train_mae)
std_test_mae = np.std(test_mae)
std_train_r2 = np.std(train_r2)
std_test_r2 = np.std(test_r2)

# Print the mean and standard deviation of evaluation metrics
print("Mean train score: {:.5f} +/- {:.5f}".format(mean_train_score,␣
 ↪std_train_score))
print("Mean test score: {:.5f} +/- {:.5f}".format(mean_test_score,␣
 ↪std_test_score))
print("Mean train MSE: {:.5f} +/- {:.5f}".format(mean_train_mse, std_train_mse))
print("Mean test MSE: {:.5f} +/- {:.5f}".format(mean_test_mse, std_test_mse))
print("Mean train MAE: {:.5f} +/- {:.5f}".format(mean_train_mae, std_train_mae))
print("Mean test MAE: {:.5f} +/- {:.5f}".format(mean_test_mae, std_test_mae))
print("Mean train R^2: {:.5f} +/- {:.5f}".format(mean_train_r2, std_train_r2))
print("Mean test R^2: {:.5f} +/- {:.5f}".format(mean_test_r2, std_test_r2))
```

```
Mean train score: 0.90242 +/- 0.01112
Mean test score: 0.85730 +/- 0.04462
Mean train MSE: 14224739.00574 +/- 1271403.97849
Mean test MSE: 20427228.90336 +/- 5363680.73304
Mean train MAE: 2042.63645 +/- 142.04313
Mean test MAE: 2452.22782 +/- 238.02025
Mean train R^2: 0.90242 +/- 0.01112
Mean test R^2: 0.85730 +/- 0.04462
```

1.6 Model tuning

https://en.wikipedia.org/wiki/Hyperparameter_optimization

1.6.1 Parameters vs Hyperparameters

Let's now define what are hyperparameters, but before doing that let's consider the difference between a parameter and a hyperparameter.

A parameter can be considered to be intrinsic or internal to the model and can be obtained after the model has learned from the data. Examples of parameters are regression coefficients in linear regression, support vectors in support vector machines and weights in neural networks.

A hyperparameter can be considered to be extrinsic or external to the model and can be set arbitrarily by the practitioner. Examples of hyperparameters include the k in k-nearest neighbors, number of trees and maximum number of features in random forest, learning rate and momentum in neural networks, the C and gamma parameters in support vector machines.

1.6.2 Hyperparameter tuning

As there are no universal best hyperparameters to use for any given problem, hyperparameters are typically set to default values. However, the optimal set of hyperparameters can be obtained from manual empirical (trial-and-error) hyperparameter search or in an automated fashion via the use of optimization algorithm to maximize the fitness function.

Two common hyperparameter tuning methods include grid search and random search. As the name implies, a grid search entails the creation of a grid of possible hyperparameter values whereby models are iteratively built for all of these hyperparameter combinations in a brute force manner. In a random search, not all hyperparameter combinations are used, but instead each iteration makes use of a random hyperparameter combination.

Building a Baseline Gradient Boosting Regressor Here, we will first start by building a baseline Gradient Boosting Regressor that will serve as a baseline for comparative purpose with the model using the optimal set of hyperparameters. For the baseline model, we will set a default hyperparameters of Gradient Boosting Regressor.

[24]:
```
X = df_encoded.drop(['charges','region'], axis = 1)
Y = df_encoded.charges

X_train,X_test,Y_train,Y_test = train_test_split(X,Y, test_size=0.2,␣
 ↪random_state = 0)

plr = GradientBoostingRegressor(random_state = 42).fit(X_train,Y_train)

Y_train_pred = plr.predict(X_train)
Y_test_pred = plr.predict(X_test)

# calculating metrics
score_train = plr.score(X_train, Y_train)
score_test = plr.score(X_test, Y_test)

mse_train = mean_squared_error(Y_train, Y_train_pred)
mse_test = mean_squared_error(Y_test, Y_test_pred)

mae_train = mean_absolute_error(Y_train, Y_train_pred)
mae_test = mean_absolute_error(Y_test, Y_test_pred)

r2_train = r2_score(Y_train, Y_train_pred)
r2_test = r2_score(Y_test, Y_test_pred)

# function to format score_train in desired pattern
def format_score(score):
    return "{:.6f}".format(score)

# displaying metrics in a grid
metrics_df = pd.DataFrame({
```

```
    'Metric': ['Score', 'MSE', 'MAE', 'R2'],
    'Train Data': [format_score(score_train), mse_train, mae_train, r2_train],
    'Test Data': [format_score(score_test), mse_test, mae_test, r2_test]
})

print(metrics_df)
```

```
  Metric       Train Data        Test Data
0  Score         0.895655         0.897123
1    MSE  14959484.462928  16370815.053277
2    MAE      2123.736738       2366.20557
3     R2         0.895655         0.897123
```

[25]:
```
# display baseline model params
plr.get_params()
```

[25]:
```
{'alpha': 0.9,
 'ccp_alpha': 0.0,
 'criterion': 'friedman_mse',
 'init': None,
 'learning_rate': 0.1,
 'loss': 'squared_error',
 'max_depth': 3,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_iter_no_change': None,
 'random_state': 42,
 'subsample': 1.0,
 'tol': 0.0001,
 'validation_fraction': 0.1,
 'verbose': 0,
 'warm_start': False}
```

Hyperparameter Tuning Now we will be performing the tuning of hyperparameters of the random forest model.

[37]:
```
# grid with all param to tune on model
param_grid = {
    'n_estimators': [10, 50, 100, 150, 200],
    'max_depth': [2, 3, 5, 7, 9, 11],
    'learning_rate': [0.01, 0.1, 0.3],
    'min_samples_split': [2, 5, 10, 13],
```

```
    'min_samples_leaf': [1, 2, 4, 5]
}
```

[38]:
```
# create new GridSearchCV object
grid_search = GridSearchCV(estimator=GradientBoostingRegressor(random_state=42),
                           param_grid=param_grid,
                           scoring='neg_mean_squared_error', #
 ↪
                           cv=5, #                    -
                           n_jobs=-1) #                    CPU
```

[39]:
```
# release Grid Search
grid_search.fit(X_train, Y_train)
```

[39]:
```
GridSearchCV(cv=5, estimator=GradientBoostingRegressor(random_state=42),
             n_jobs=-1,
             param_grid={'learning_rate': [0.01, 0.1, 0.3],
                         'max_depth': [2, 3, 5, 7, 9, 11],
                         'min_samples_leaf': [1, 2, 4, 5],
                         'min_samples_split': [2, 5, 10, 13],
                         'n_estimators': [10, 50, 100, 150, 200]},
             scoring='neg_mean_squared_error')
```

[53]:
```
# display best tuned hyperparameters
print("Best hyperparameters:", grid_search.best_params_)
```

```
Best hyperparameters: {'learning_rate': 0.1, 'max_depth': 3, 'min_samples_leaf':
4, 'min_samples_split': 13, 'n_estimators': 50}
```

[50]:
```
# define features for model
X = df_encoded.drop(['charges','region', 'sex'], axis = 1)
Y = df_encoded.charges

# split dataset
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,␣
 ↪random_state = 0)

plr = GradientBoostingRegressor(random_state = 42, learning_rate = 0.1,␣
 ↪max_depth = 3, min_samples_leaf = 4, min_samples_split = 13, n_estimators =␣
 ↪50).fit(X_train,Y_train)

# train model
Y_train_pred = plr.predict(X_train)
Y_test_pred = plr.predict(X_test)

# calculating metrics
score_train = plr.score(X_train, Y_train)
```

```python
score_test = plr.score(X_test, Y_test)

mse_train = mean_squared_error(Y_train, Y_train_pred)
mse_test = mean_squared_error(Y_test, Y_test_pred)

mae_train = mean_absolute_error(Y_train, Y_train_pred)
mae_test = mean_absolute_error(Y_test, Y_test_pred)

r2_train = r2_score(Y_train, Y_train_pred)
r2_test = r2_score(Y_test, Y_test_pred)

# function to format score_train in desired pattern
def format_score(score):
    return "{:.6f}".format(score)

# displaying metrics in a grid
metrics_df = pd.DataFrame({
    'Metric': ['Score', 'MSE', 'MAE', 'R2'],
    'Train Data': [format_score(score_train), mse_train, mae_train, r2_train],
    'Test Data': [format_score(score_test), mse_test, mae_test, r2_test]
})

print(metrics_df)
```

```
  Metric        Train Data         Test Data
0  Score          0.873020          0.899796
1    MSE   18204620.074767   15945425.744968
2    MAE       2353.121722         2375.0363
3     R2           0.87302          0.899796
```

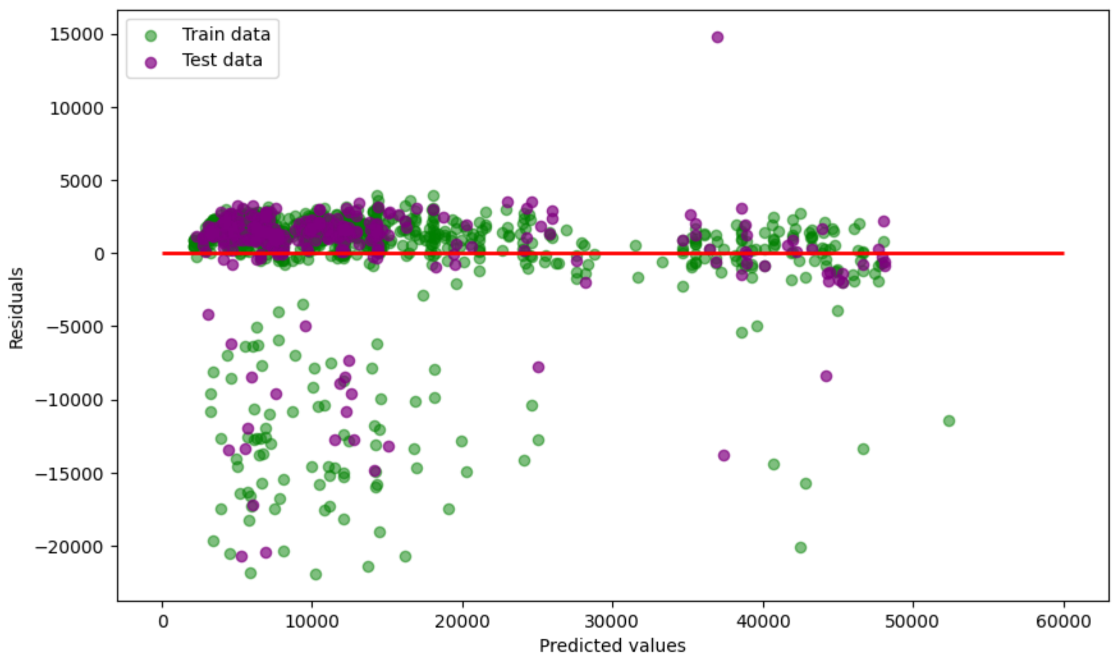
Result Analysis (R2) , (0.87 0.90), , .

: 1. (MSE) 15,955,774. , $15,955 . 2. (MAE) 2,374. , $2,374.

, , *Gradient Boosting Regressor* , , .

[51]:
```python
# Set the figure size
plt.figure(figsize=(10,6))

# Plot the residuals for the training data
plt.scatter(Y_train_pred, Y_train_pred - Y_train,
            c='green', marker='o', s=35, alpha=0.5,
```

```
            label='Train data')

# Plot the residuals for the test data
plt.scatter(Y_test_pred, Y_test_pred - Y_test,
            c='purple', marker='o', s=35, alpha=0.7,
            label='Test data')

# Label the x and y axes
plt.xlabel('Predicted values')
plt.ylabel('Residuals')

# Add a legend
plt.legend(loc='upper left')

# Add a horizontal line at y=0 for reference
plt.hlines(y=0, xmin=0, xmax=60000, lw=2, color='red')

# Show the plot
plt.show()
```



Feature importance for Final Gradient Boosting Regressor

[52]:
```
rankings = plr.feature_importances_.tolist()
importance = pd.DataFrame(sorted(zip(X_train.
 ↪columns,rankings),reverse=True),columns=["variable","importance"]).
 ↪sort_values("importance",ascending = False)
```

```
plt.figure(figsize=(8,6))
sns.barplot(x="importance",
            y="variable",
            data=importance[:5])
plt.title('Variable Importance')
plt.tight_layout()
```