Data Science and Business Analytics

Practice Project III

Prediction of Medical Insurance Charges

By

Hayford Osumanu

December 2022

Objective of the Project

In this project, we are going to extract some important insights from a dataset that contains details about the background of a person who is purchasing medical insurance along with what amount of premium is charged to those individuals as well using Machine Learning in Python.

Importing necessary libraries

In [1]:
```
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd

import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import BaggingRegressor,RandomForestRegressor, GradientBoostingRe
from xgboost import XGBRegressor
from sklearn import metrics
from sklearn.model_selection import GridSearchCV, train_test_split
```

Reading the dataset

In [4]:
```
#Loading dataset
data=pd.read_csv("insurance.csv")
```

Overview of the dataset

View the first 5 rows of the dataset

In [5]:
```
data.head()
```

Out[5]:

	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

Check data types and number of non-null values for each column

In [6]:
```
data.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
```

In [7]:
```
data.isna().sum()
```

Out[7]:
```
age         0
sex         0
bmi         0
children    0
smoker      0
region      0
charges     0
dtype: int64
```

- There are no missing values in the data.

Summary of the dataset

In [8]:
```
# Summary of continuous columns
data.describe().T
```

Out[8]:

	count	mean	std	min	25%	50%	75%	max
age	1338.0	39.207025	14.049960	18.0000	27.00000	39.000	51.000000	64.00000
bmi	1338.0	30.663397	6.098187	15.9600	26.29625	30.400	34.693750	53.13000
children	1338.0	1.094918	1.205493	0.0000	0.00000	1.000	2.000000	5.00000
charges	1338.0	13270.422265	12110.011237	1121.8739	4740.28715	9382.033	16639.912515	63770.42801

Number of unique values in each column

In [9]:
```
data.nunique()
```

Out[9]:
```
age          47
sex           2
bmi         548
children      6
```

```
smoker          2
region          4
charges      1337
dtype: int64
```

Number of observations in each category

In [10]:

```
# Making a list of all catrgorical variables
cat_col = list(data.select_dtypes("object").columns)

# Printing number of count of each unique value in each column
for column in cat_col:
    print(data[column].value_counts())
    #print("cat_col")
```

```
male      676
female    662
Name: sex, dtype: int64
no     1064
yes     274
Name: smoker, dtype: int64
southeast    364
southwest    325
northwest    325
northeast    324
Name: region, dtype: int64
```

In [11]:

```
df = data.copy()
```

Exploratory Data Analysis (EDA) Summary

Statistical Summary of the Dataset

In [12]:

```
# let's view the statistical summary of minimum numerical columns in the data
data.describe(include=np.number).T.style.highlight_min(color="green", axis=0)
```

Out[12]:

	count	mean	std	min	25%	50%	75%
age	1338.000000	39.207025	14.049960	18.000000	27.000000	39.000000	51.000000
bmi	1338.000000	30.663397	6.098187	15.960000	26.296250	30.400000	34.693750
children	1338.000000	1.094918	1.205493	0.000000	0.000000	1.000000	2.000000
charges	1338.000000	13270.422265	12110.011237	1121.873900	4740.287150	9382.033000	16639.912515

In [13]:

```
# let's view the statistical summary of maximum numerical columns in the data
data.describe(include=np.number).T.style.highlight_max(color="indigo", axis=0)
```

Out[13]:

	count	mean	std	min	25%	50%	75%
age	1338.000000	39.207025	14.049960	18.000000	27.000000	39.000000	51.000000

	count	mean	std	min	25%	50%	75%
bmi	1338.000000	30.663397	6.098187	15.960000	26.296250	30.400000	34.693750
children	1338.000000	1.094918	1.205493	0.000000	0.000000	1.000000	2.000000
charges	1338.000000	13270.422265	12110.011237	1121.873900	4740.287150	9382.033000	16639.912515

In [14]:
```
# Extracting the Quantiles of the dataset
data.quantile([0.25, 0.5, 0.6, 0.75, 0.9, 0.95, 0.99]).T.style.highlight_max(
    color="purple", axis=0
)
```

Out[14]:

	0.25	0.5	0.6	0.75	0.9	0.95	0.9
age	27.000000	39.000000	44.000000	51.000000	59.000000	62.000000	64.00000
bmi	26.296250	30.400000	32.032000	34.693750	38.619500	41.106000	46.40790
children	0.000000	1.000000	1.000000	2.000000	3.000000	3.000000	5.00000
charges	4740.287150	9382.033000	11399.857160	16639.912515	34831.719700	41181.827787	48537.48072

In [ ]:
```
# Extracting the Quantiles of the dataset
data.quantile([0.25, 0.5, 0.6, 0.75, 0.9, 0.95, 0.99]).T.style.highlight_min(
    color="red", axis=0
)
```

Out[ ]:

	0.25	0.5	0.6	0.75	0.9	0.95	0.9
age	27.000000	39.000000	44.000000	51.000000	59.000000	62.000000	64.00000
bmi	26.296250	30.400000	32.032000	34.693750	38.619500	41.106000	46.40790
children	0.000000	1.000000	1.000000	2.000000	3.000000	3.000000	5.00000
charges	4740.287150	9382.033000	11399.857160	16639.912515	34831.719700	41181.827787	48537.48072

The below functions need to be defined to carry out the EDA.

In [ ]:
```
def histogram_boxplot(data, feature, figsize=(15, 10), kde=False, bins=None):
    """
    Boxplot and histogram combined

    data: dataframe
    feature: dataframe column
    figsize: size of figure (default (15,10))
    kde: whether to show the density curve (default False)
    bins: number of bins for histogram (default None)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(
        nrows=2,  # Number of rows of the subplot grid= 2
        sharex=True,  # x-axis will be shared among all subplots
```

```
        gridspec_kw={"height_ratios": (0.25, 0.75)},
        figsize=figsize,
    )  # creating the 2 subplots
    sns.boxplot(
        data=data, x=feature, ax=ax_box2, showmeans=True, color="violet"
    )  # boxplot will be created and a triangle will indicate the mean value of the col
    sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2, bins=bins
    ) if bins else sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2
    )  # For histogram
    ax_hist2.axvline(
        data[feature].mean(), color="green", linestyle="--"
    )  # Add mean to the histogram
    ax_hist2.axvline(
        data[feature].median(), color="black", linestyle="-"
    )  # Add median to the histogram
```

In [ ]:

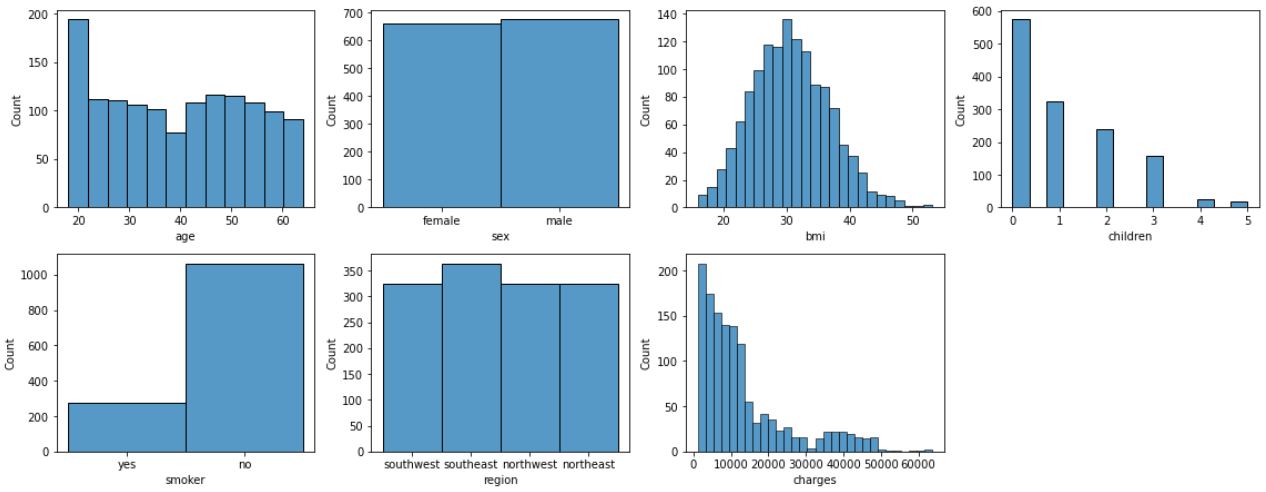
```
# function to create labeled barplots


def labeled_barplot(data, feature, perc=False, n=None):
    """
    Barplot with percentage at the top

    data: dataframe
    feature: dataframe column
    perc: whether to display percentages instead of count (default is False)
    n: displays the top n category levels (default is None, i.e., display all levels)
    """

    total = len(data[feature])  # length of the column
    count = data[feature].nunique()
    if n is None:
        plt.figure(figsize=(count + 2, 6))
    else:
        plt.figure(figsize=(n + 2, 6))

    plt.xticks(rotation=90, fontsize=15)
    ax = sns.countplot(
        data=data,
        x=feature,
        palette="Paired",
        order=data[feature].value_counts().index[:n],
    )

    for p in ax.patches:
        if perc == True:
            label = "{:.1f}%".format(
                100 * p.get_height() / total
            )  # percentage of each class of the category
        else:
            label = p.get_height()  # count of each level of the category

        x = p.get_x() + p.get_width() / 2  # width of the plot
        y = p.get_height()  # height of the plot

        ax.annotate(
```

```
            label,
            (x, y),
            ha="center",
            va="center",
            size=12,
            xytext=(0, 5),
            textcoords="offset points",
        )  # annotate the percentage

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

Univariate analysis

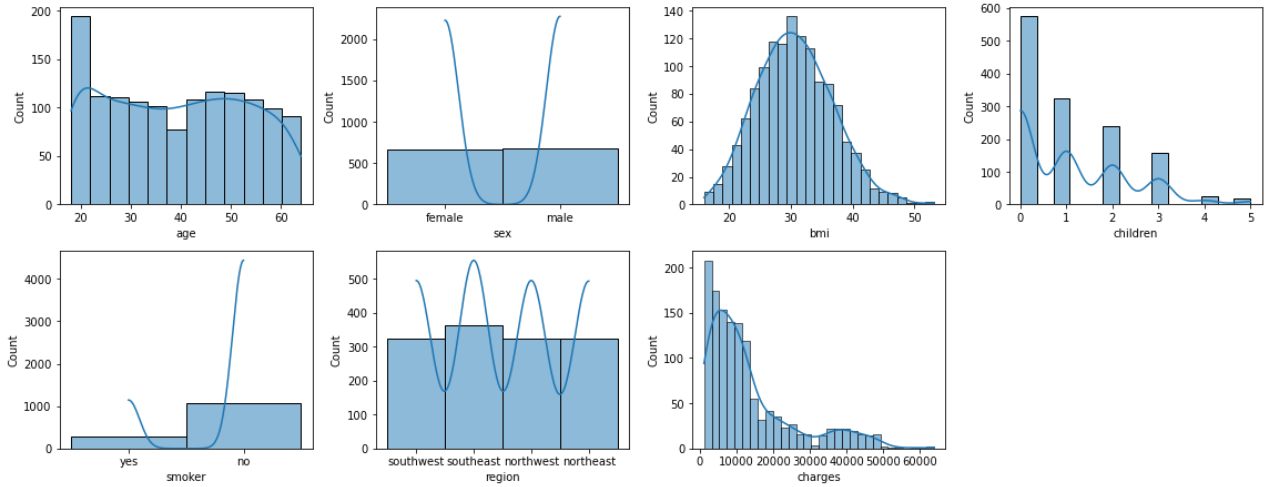
In [ ]:

```
# Checking the histogram plot of numerical variables of the entire dataset
cols = 4
rows = 5
num_cols = data.select_dtypes(exclude='category').columns
fig = plt.figure( figsize=(cols*4, rows*3))
for i, col in enumerate(num_cols):

    ax=fig.add_subplot(rows,cols,i+1)

    sns.histplot(x =data[col], ax = ax)

fig.tight_layout()
plt.show()
```



In [ ]:

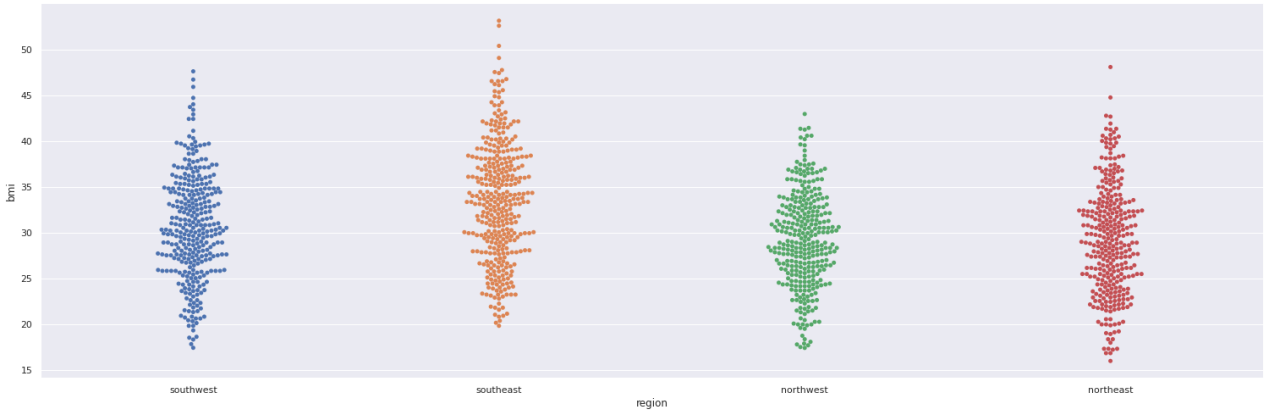
```
# Checking the histogram plot of numerical variables of the entire dataset
cols = 4
rows = 5
num_cols = data.select_dtypes(exclude='category').columns
fig = plt.figure( figsize=(cols*4, rows*3))
for i, col in enumerate(num_cols):

    ax=fig.add_subplot(rows,cols,i+1)

    sns.histplot(x =data[col], kde = True, ax = ax)

fig.tight_layout()
plt.show()
```



Bivariate analysis

In [15]:

```
sns.set(rc={'figure.figsize':(21,7)})
sns.catplot(x="region", y="bmi", kind="swarm", data=data, height=7, aspect=3);
```



Part III: EDA - Multivariate Data Analysis

Boxplot Comparison Analysis

In [19]:

```
### function to plot distributions wrt target


def distribution_plot_wrt_target(data, predictor, target):

    fig, axs = plt.subplots(2, 2, figsize=(25, 9))

    target_uniq = data[target].unique()

    axs[0, 0].set_title("Distribution of target for target=" + str(target_uniq[0]))
    sns.histplot(
        data=data[data[target] == target_uniq[0]],
        x=predictor,
        kde=True,
        ax=axs[0, 0],
        color="teal",
```

```
        stat="density",
    )

    axs[0, 1].set_title("Distribution of target for target=" + str(target_uniq[1]))
    sns.histplot(
        data=data[data[target] == target_uniq[1]],
        x=predictor,
        kde=True,
        ax=axs[0, 1],
        color="orange",
        stat="density",
    )

    axs[1, 0].set_title("Boxplot w.r.t target")
    sns.boxplot(data=data, x=target, y=predictor, ax=axs[1, 0], palette="gist_rainbow")

    axs[1, 1].set_title("Boxplot (without outliers) w.r.t target")
    sns.boxplot(
        data=data,
        x=target,
        y=predictor,
        ax=axs[1, 1],
        showfliers=False,
        palette="gist_rainbow",
    )

    plt.tight_layout()
    plt.show()
```

Comparison of the Numerical Columns

In [ ]:
```
# copying the data to another variable to avoid any changes to original data
df = data.copy()
```

In [ ]:
```
# Extracting the columns of the entire datasets
df.columns
```

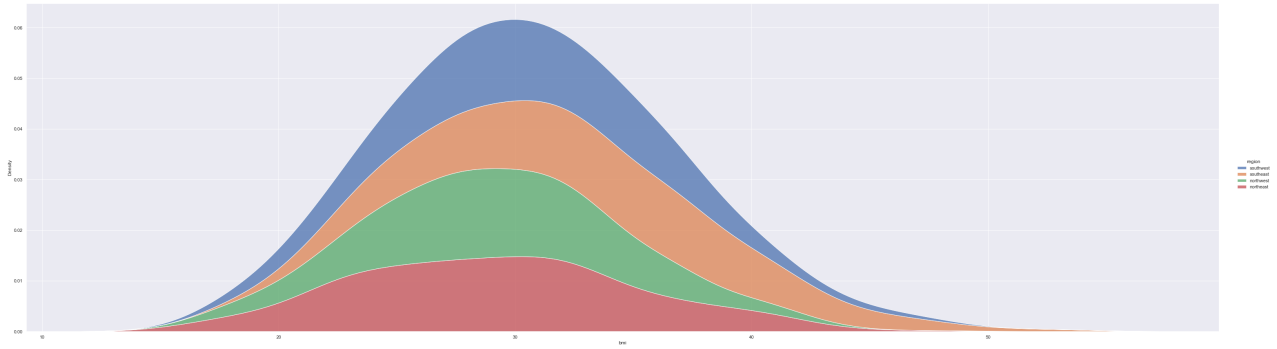
Out[ ]:
```
Index(['age', 'sex', 'bmi', 'children', 'smoker', 'region', 'charges'], dtype='object')
```

In [ ]:
```
# Extracting the numerical col of the datasets
col_var = ['age','bmi', 'children', 'region', 'charges'
]
```

BMI in Relation to Region

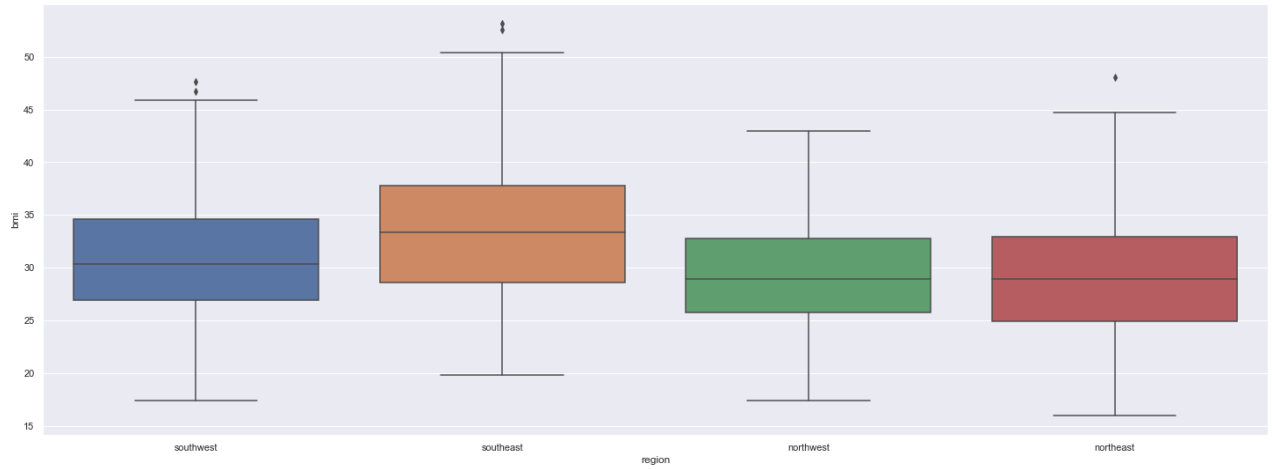
In [ ]:
```
# Ploting a displot of bmi vs region
sns.displot(
    data=df,
    x="bmi",
```

```
    hue="region",
    multiple="stack",
    kind="kde",
    height=12,
    aspect=3.5,
);
```



In [ ]:

```
# Boxplot of  BMI vs Region
plt.figure(figsize=(25, 9))
sns.boxplot(data=df, x="region", y="bmi");
```



In [ ]:

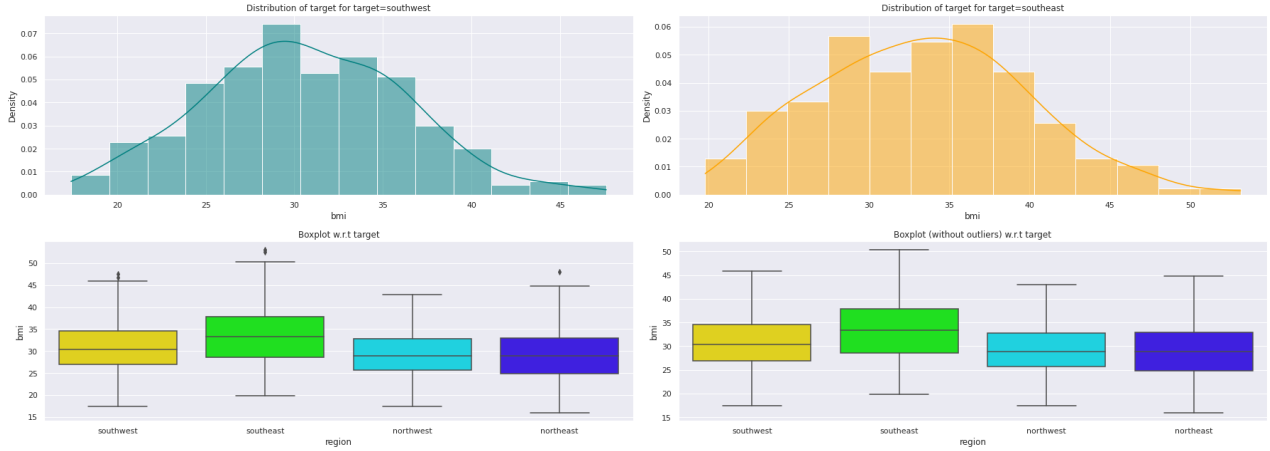
```
# comparing
data.groupby(['region'])['bmi'].describe()
```

Out[ ]:

region	count	mean	std	min	25%	50%	75%	max
northeast	324.0	29.173503	5.937513	15.960	24.86625	28.88	32.89375	48.07
northwest	325.0	29.199785	5.136765	17.385	25.74500	28.88	32.77500	42.94
southeast	364.0	33.355989	6.477648	19.800	28.57250	33.33	37.81250	53.13
southwest	325.0	30.596615	5.691836	17.400	26.90000	30.30	34.60000	47.60

In [20]:

```
distribution_plot_wrt_target(data, 'bmi', 'region')
```



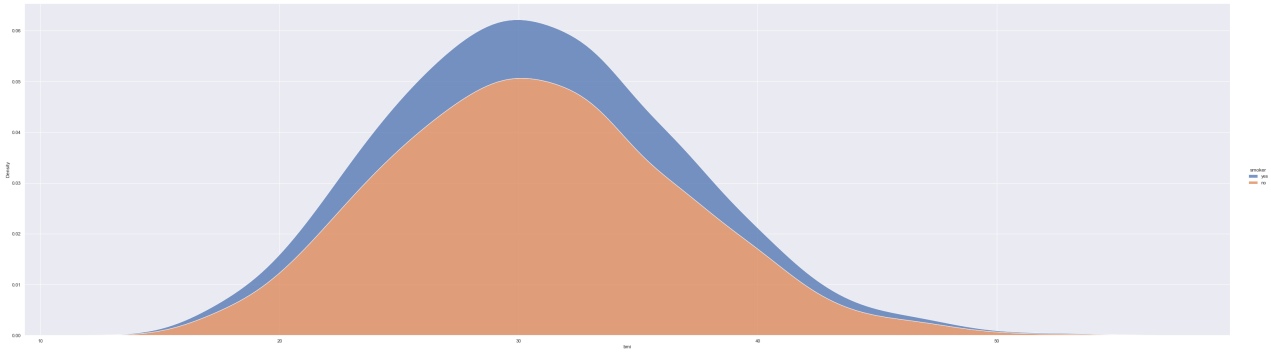
BMI in Relation to Smoker

In [ ]:

```
# Ploting a displot of BMI vs Smoker
sns.displot(
    data=df,
    x="bmi",
    hue="smoker",
    multiple="stack",
    kind="kde",
    height=12,
    aspect=3.5,
)
```

Out[ ]:

```
<seaborn.axisgrid.FacetGrid at 0x17cbbb81790>
```

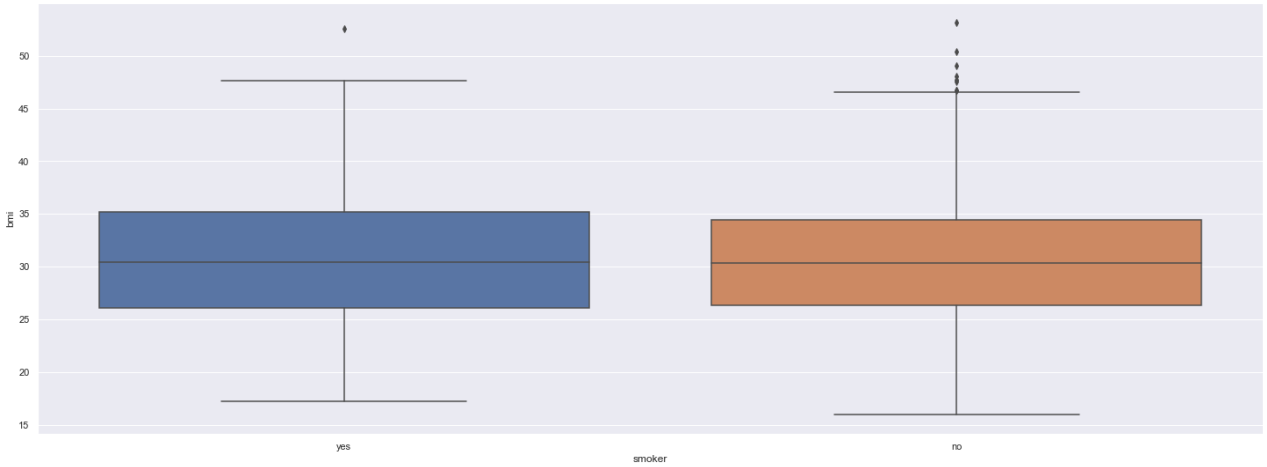


In [ ]:

```
# Boxplot of BMI vs Smoker
plt.figure(figsize=(25, 9))
sns.boxplot(data=df, x="smoker", y="bmi")
```

Out[ ]:

```
<AxesSubplot:xlabel='smoker', ylabel='bmi'>
```

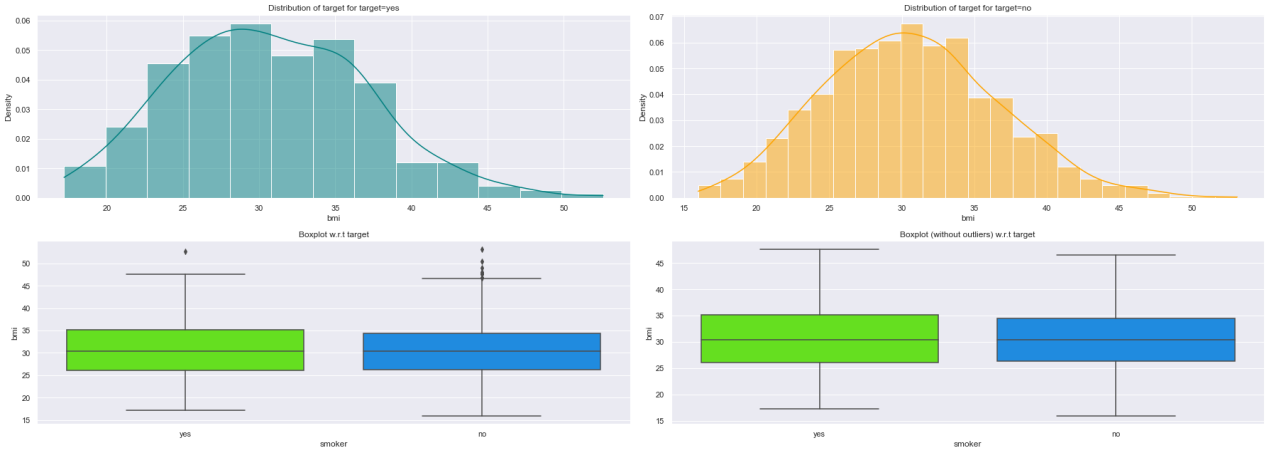


In [ ]:
```
# comparing BMI vs Smoker
data.groupby(['smoker'])['bmi'].describe()
```

Out[ ]:

smoker	count	mean	std	min	25%	50%	75%	max
no	1064.0	30.651795	6.043111	15.960	26.31500	30.3525	34.43	53.13
yes	274.0	30.708449	6.318644	17.195	26.08375	30.4475	35.20	52.58

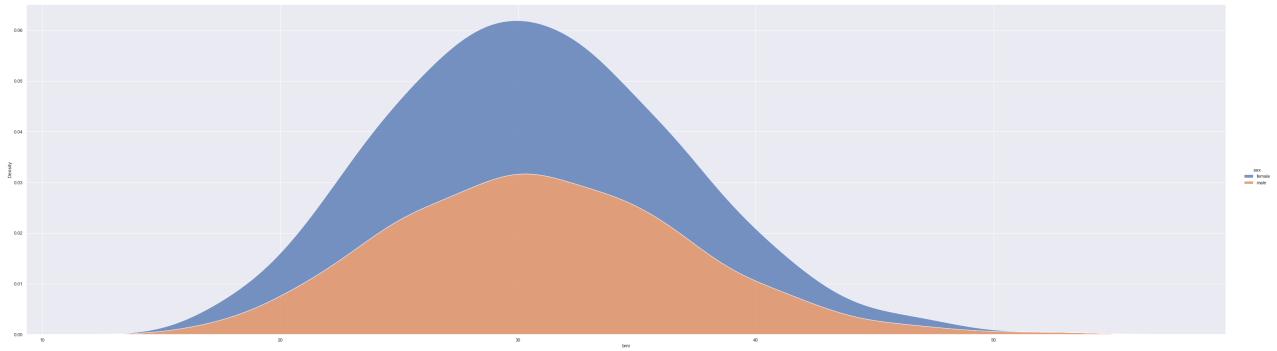
In [ ]:
```
distribution_plot_wrt_target(data, 'bmi', 'smoker')
```



BMI in Relation to Sex

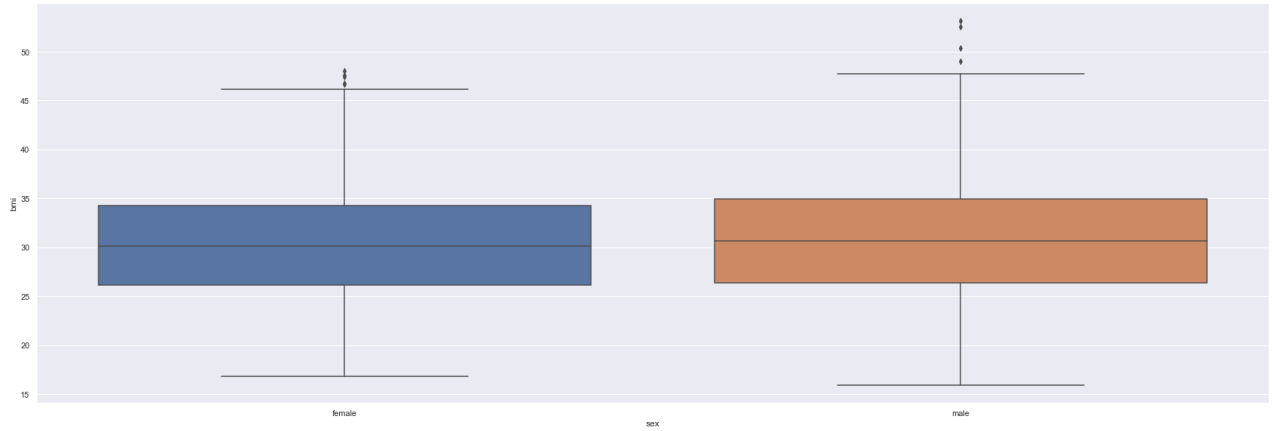
In [ ]:
```
# Ploting a displot of BMI vs Sex
sns.displot(
    data=df,
    x="bmi",
    hue="sex",
    multiple="stack",
    kind="kde",
    height=12,
```

```
    aspect=3.5,
);
```



In [ ]:
```
# Boxplot of BMI vs Sex
plt.figure(figsize=(30, 10))
sns.boxplot(data=df, x="sex", y="bmi")
```

Out[ ]:
```
<AxesSubplot:xlabel='sex', ylabel='bmi'>
```

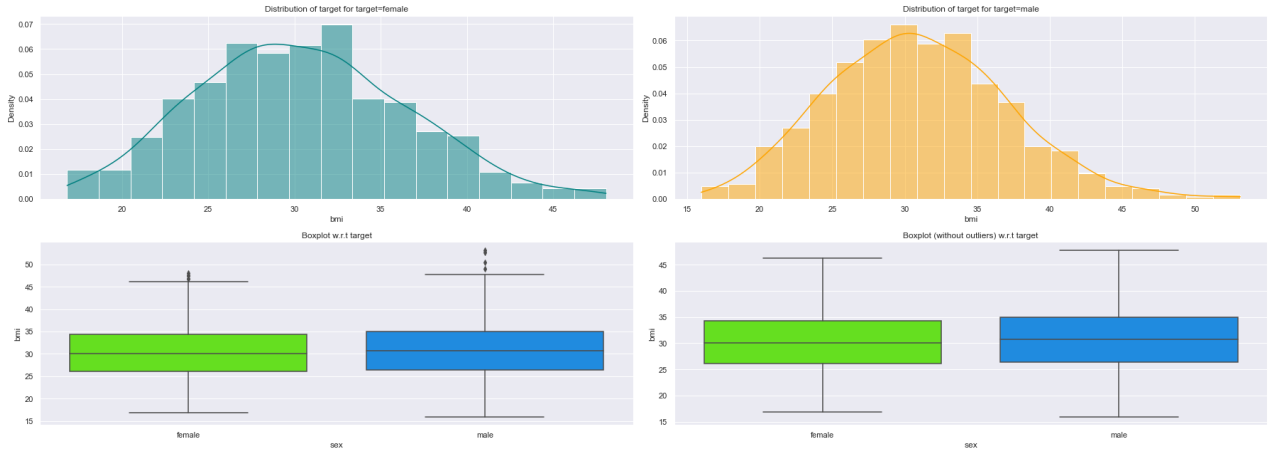


In [ ]:
```
# comparing BMI vs Sex
data.groupby(['sex'])['bmi'].describe()
```

Out[ ]:

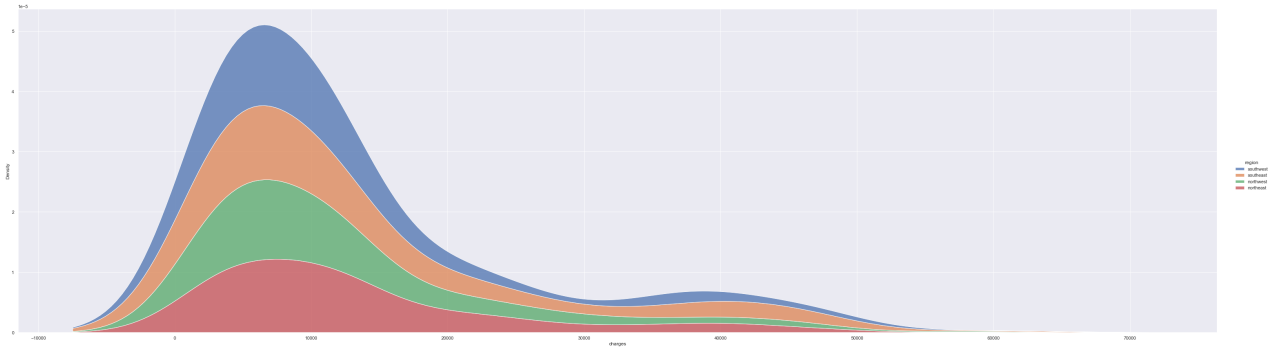
sex	count	mean	std	min	25%	50%	75%	max
female	662.0	30.377749	6.046023	16.815	26.125	30.1075	34.31375	48.07
male	676.0	30.943129	6.140435	15.960	26.410	30.6875	34.99250	53.13

In [ ]:
```
distribution_plot_wrt_target(data, 'bmi', 'sex')
```



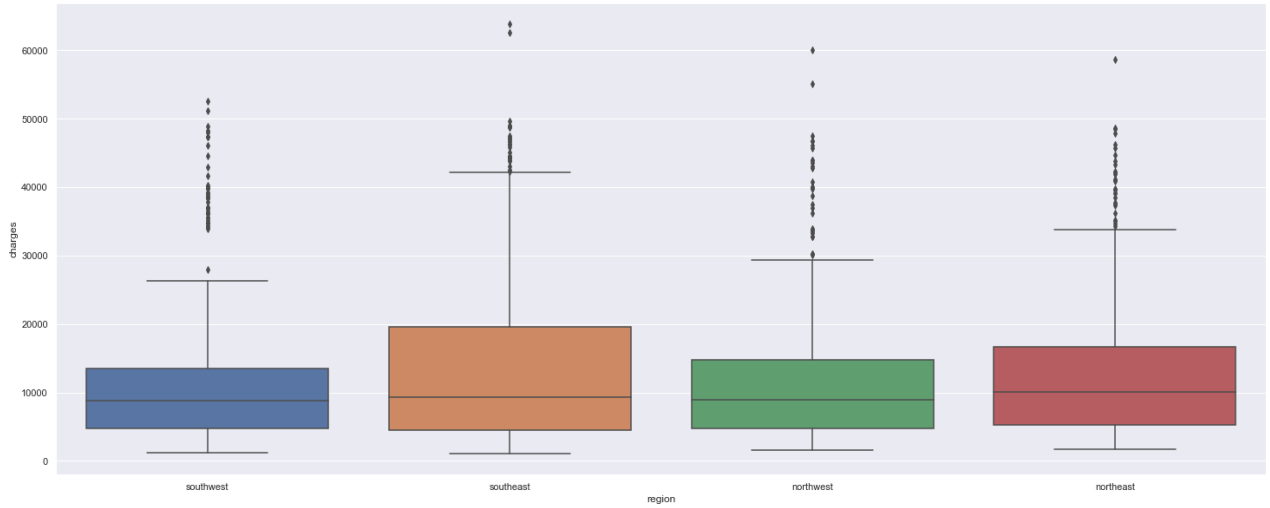
Charges in Relation to Region

In [ ]:
```
# Ploting a displot of  Charges vs Region
sns.displot(
    data=df,
    x="charges",
    hue="region",
    multiple="stack",
    kind="kde",
    height=12,
    aspect=3.5,
);
```



In [ ]:
```
#  Boxplot of  Charges vs Region
plt.figure(figsize=(25, 10))
sns.boxplot(data=df, x="region", y="charges")
```

Out[ ]:
```
<AxesSubplot:xlabel='region', ylabel='charges'>
```

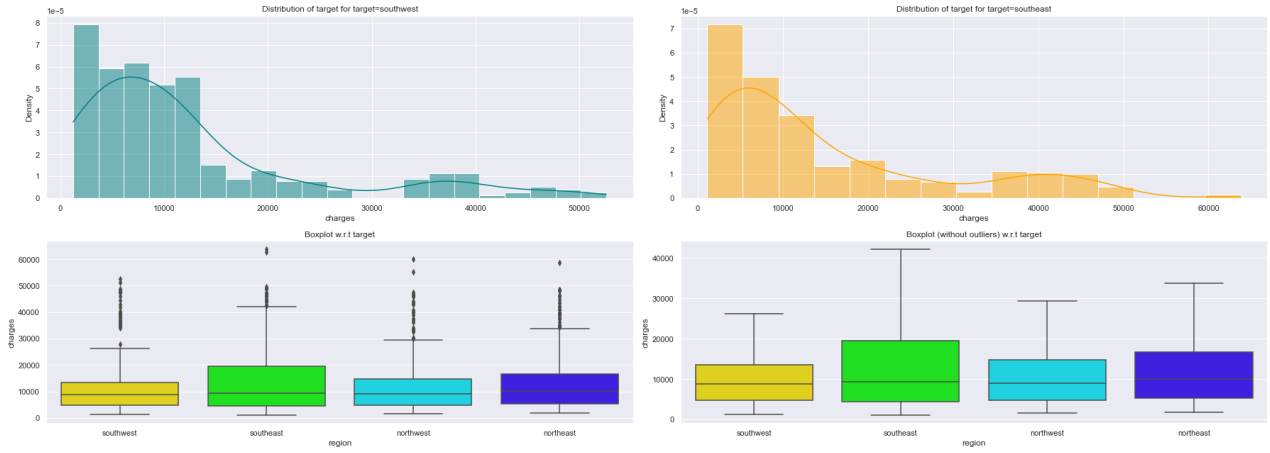


In [ ]:
```
# comparing Charges vs Region
data.groupby(['region'])['charges'].describe()
```

Out[ ]:

region	count	mean	std	min	25%	50%	75%	
northeast	324.0	13406.384516	11255.803066	1694.7964	5194.322288	10057.652025	16687.3641	58571.
northwest	325.0	12417.575374	11072.276928	1621.3402	4719.736550	8965.795750	14711.7438	60021.
southeast	364.0	14735.411438	13971.098589	1121.8739	4440.886200	9294.131950	19526.2869	63770.
southwest	325.0	12346.937377	11557.179101	1241.5650	4751.070000	8798.593000	13462.5200	52590.

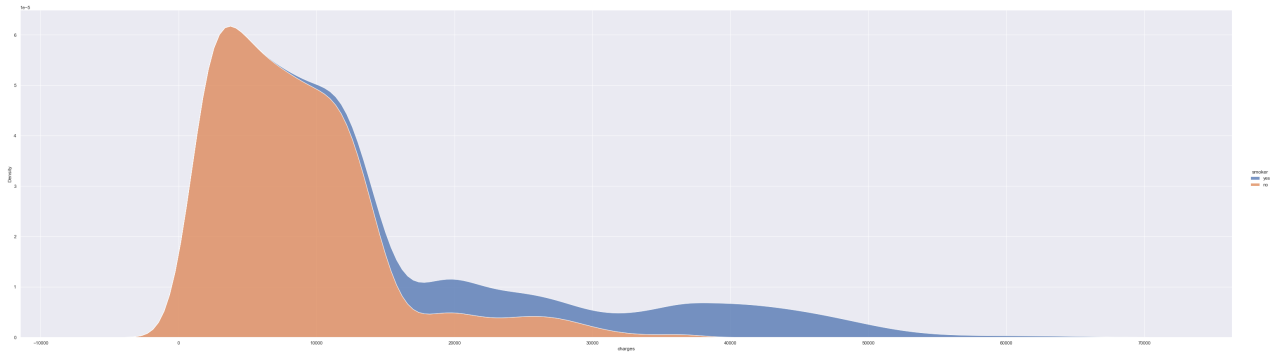
In [ ]:
```
distribution_plot_wrt_target(data, 'charges', 'region') ## Complete the code to find di
```



Charges in Relation to Smoker

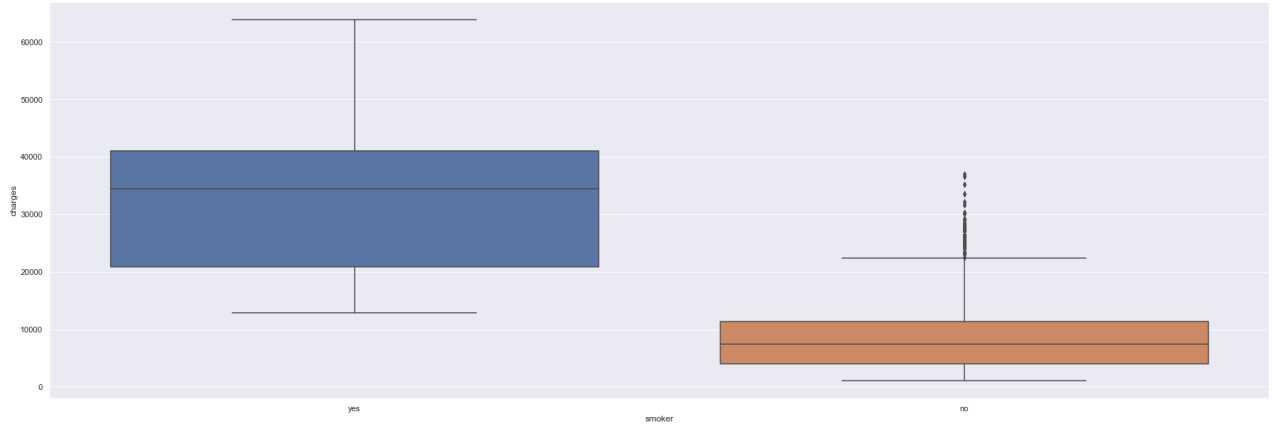
In [ ]:
```
# Ploting a displot of Charges vs Smoker
sns.displot(
    data=df,
```

```
    x="charges",
    hue="smoker",
    multiple="stack",
    kind="kde",
    height=12,
    aspect=3.5,
);
```



In [ ]:
```
# Boxplot of Charges vs Smoker
plt.figure(figsize=(30, 10))
sns.boxplot(data=df, x="smoker", y="charges")
```

Out[ ]:
```
<AxesSubplot:xlabel='smoker', ylabel='charges'>
```

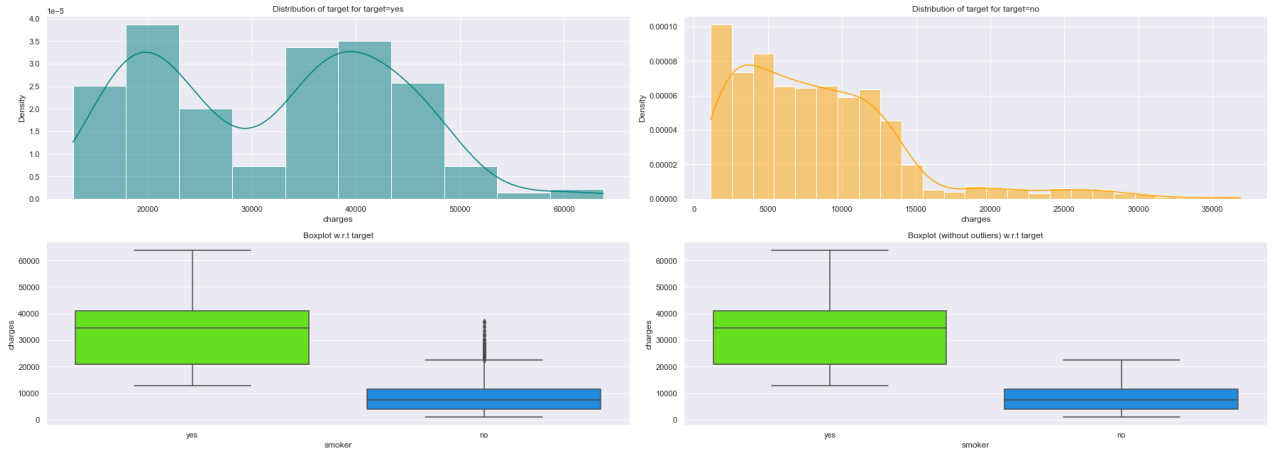


In [ ]:
```
# comparing Charges vs Smoker
data.groupby(['smoker'])['charges'].describe()
```

Out[ ]:

smoker	count	mean	std	min	25%	50%	75%	
no	1064.0	8434.268298	5993.781819	1121.8739	3986.438700	7345.40530	11362.887050	3691(
yes	274.0	32050.231832	11541.547176	12829.4551	20826.244213	34456.34845	41019.207275	6377(

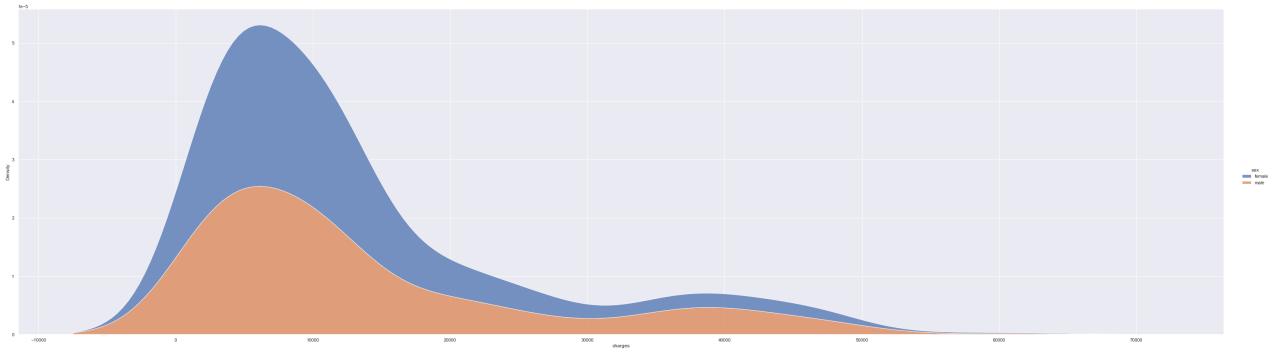
In [ ]:
```
# Charges vs Smoker
distribution_plot_wrt_target(data, 'charges', 'smoker')
```



Charges in Relation to Sex

In [ ]:

```
# Ploting a displot of Charges vs Sex
sns.displot(
    data=df,
    x="charges",
    hue="sex",
    multiple="stack",
    kind="kde",
    height=12,
    aspect=3.5,
);
```

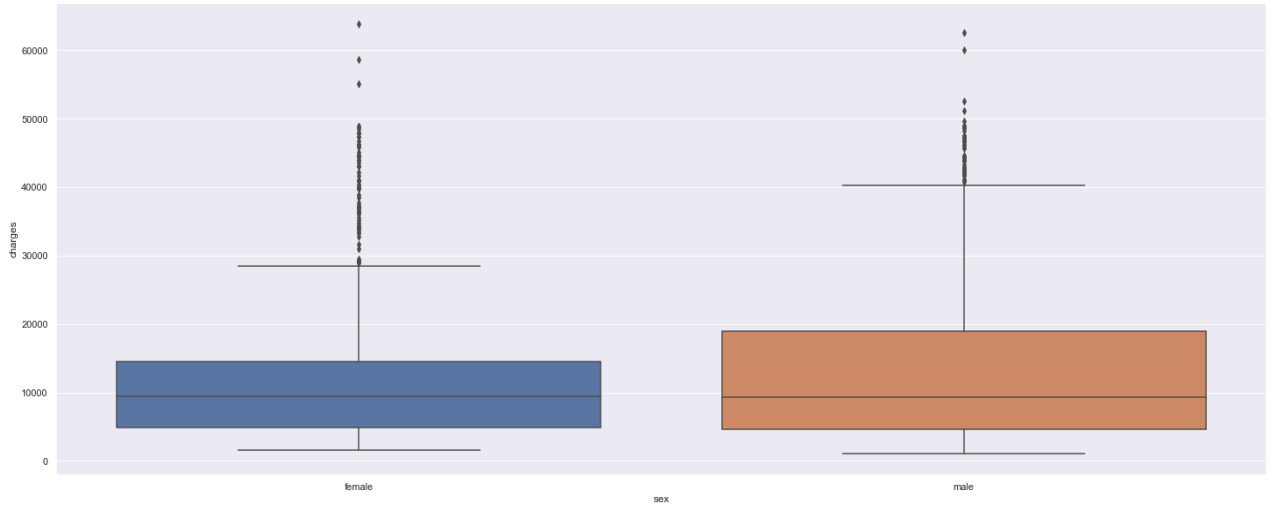


In [ ]:

```
# Boxplot of Charges vs Sex
plt.figure(figsize=(25, 10))
sns.boxplot(data=df, x="sex", y="charges")
```

Out[ ]:

```
<AxesSubplot:xlabel='sex', ylabel='charges'>
```

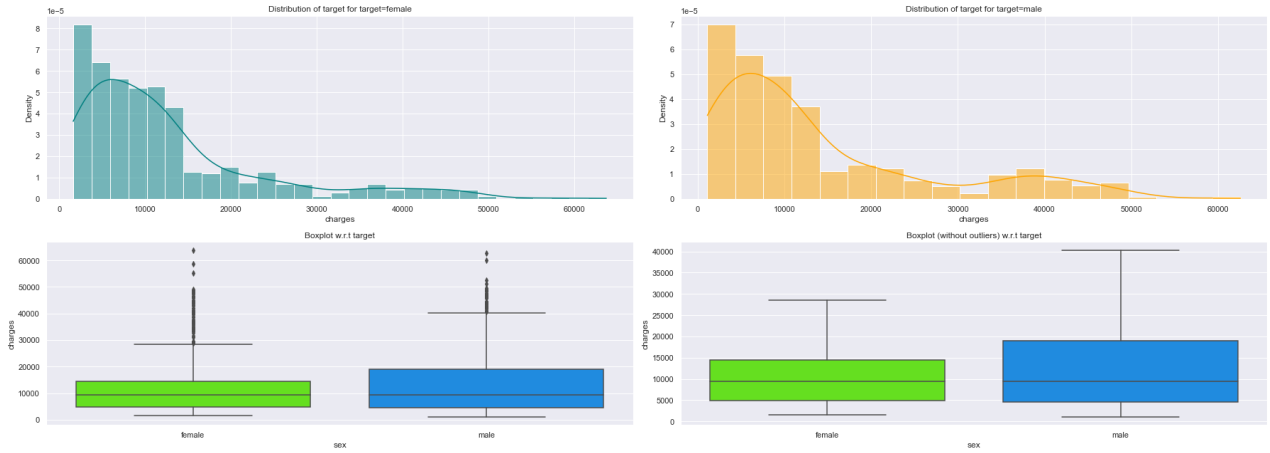


In [ ]:
```
# comparing Charges vs Sex
data.groupby(['sex'])['charges'].describe()
```

Out[ ]:

sex	count	mean	std	min	25%	50%	75%	max
female	662.0	12569.578844	11128.703801	1607.5101	4885.1587	9412.96250	14454.691825	63770.42801
male	676.0	13956.751178	12971.025915	1121.8739	4619.1340	9369.61575	18989.590250	62592.87309

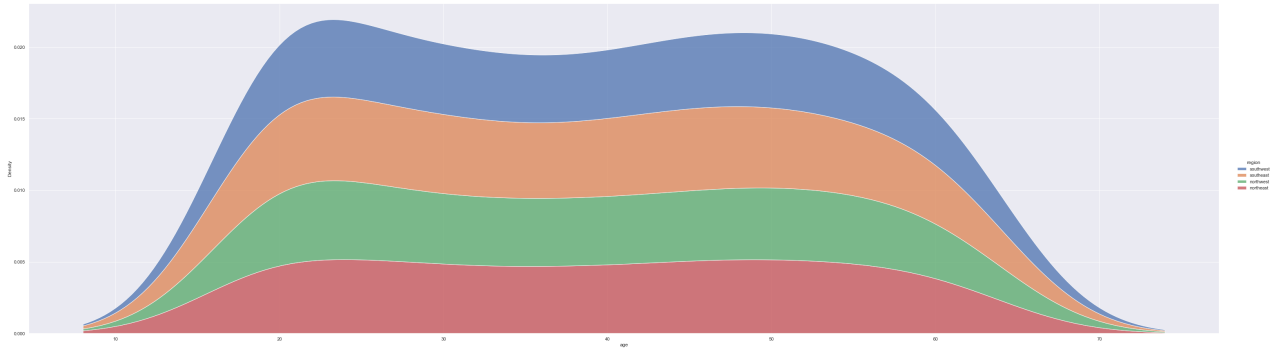
In [ ]:
```
# Charges vs Sex
distribution_plot_wrt_target(data, 'charges', 'sex')
```



Age in Relation to Region

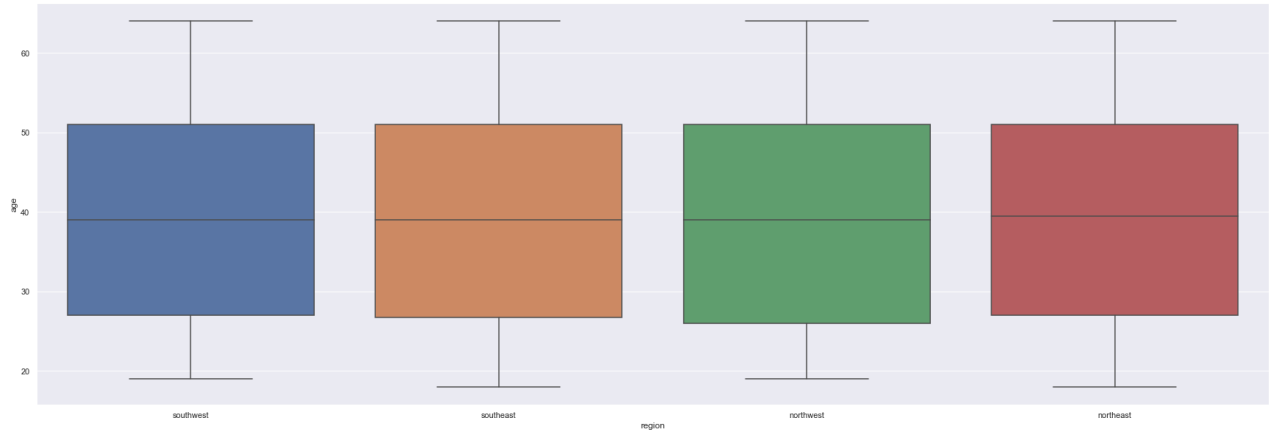
In [ ]:
```
# Ploting a displot of Age vs Region
sns.displot(
    data=df,
    x="age",
    hue="region",
    multiple="stack",
```

```
    kind="kde",
    height=12,
    aspect=3.5,
);
```



In [ ]:

```
# Boxplot of Age vs Region
plt.figure(figsize=(30, 10))
sns.boxplot(data=df, x="region", y="age");
```



In [ ]:

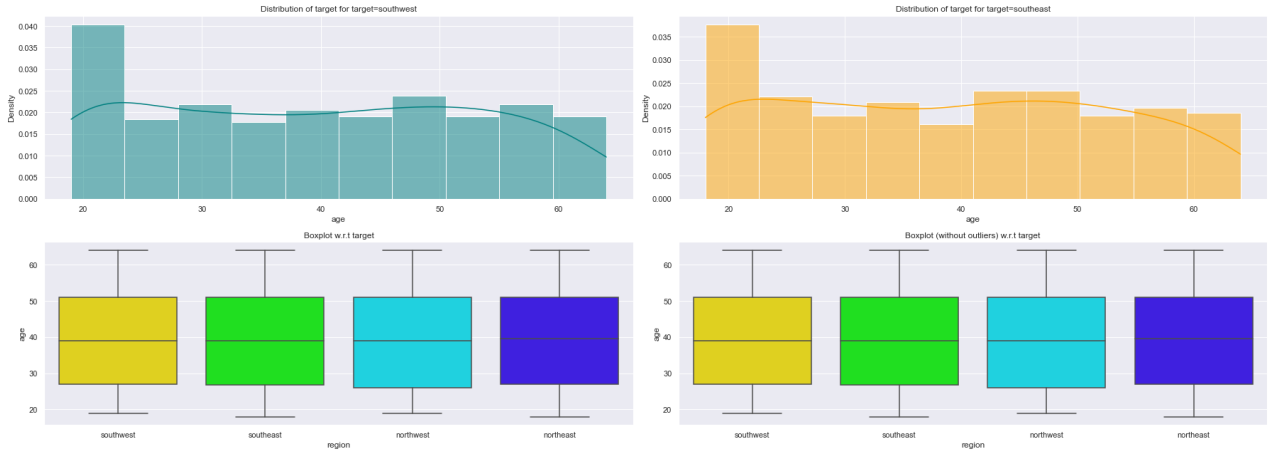
```
# comparing Age vs Region
data.groupby(['region'])['age'].describe()
```

Out[ ]:

region	count	mean	std	min	25%	50%	75%	max
northeast	324.0	39.268519	14.069007	18.0	27.00	39.5	51.0	64.0
northwest	325.0	39.196923	14.051646	19.0	26.00	39.0	51.0	64.0
southeast	364.0	38.939560	14.164585	18.0	26.75	39.0	51.0	64.0
southwest	325.0	39.455385	13.959886	19.0	27.00	39.0	51.0	64.0

In [ ]:

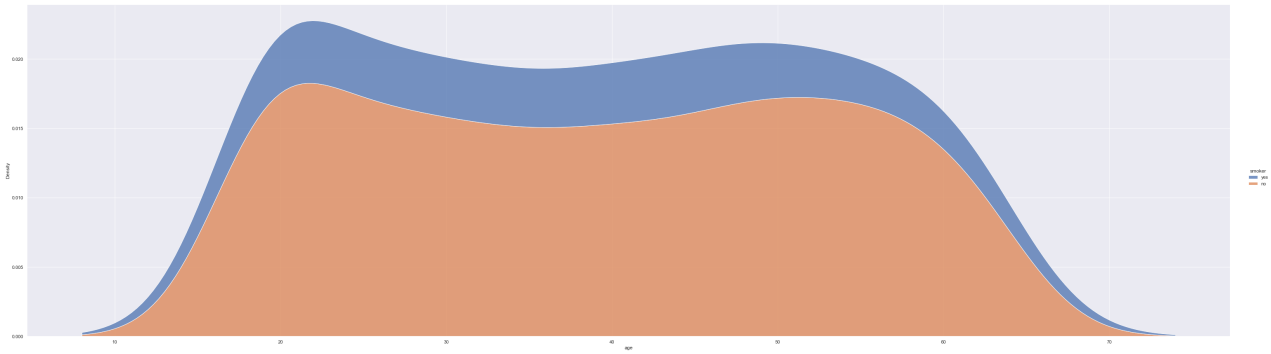
```
# Age vs Region
distribution_plot_wrt_target(data, 'age', 'region')
```



Age in Relation to Smoker

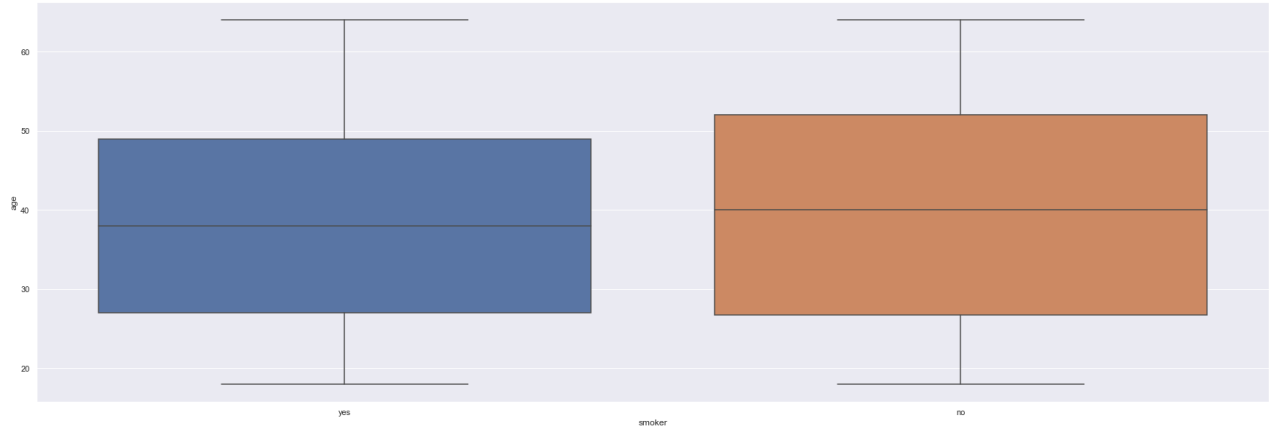
In [ ]:

```
# Ploting a displot of Age vs smoker
sns.displot(
    data=df,
    x="age",
    hue="smoker",
    multiple="stack",
    kind="kde",
    height=12,
    aspect=3.5,
);
```



In [ ]:

```
# Boxplot of Age vs smoker
plt.figure(figsize=(30, 10))
sns.boxplot(data=df, x="smoker", y="age");
```

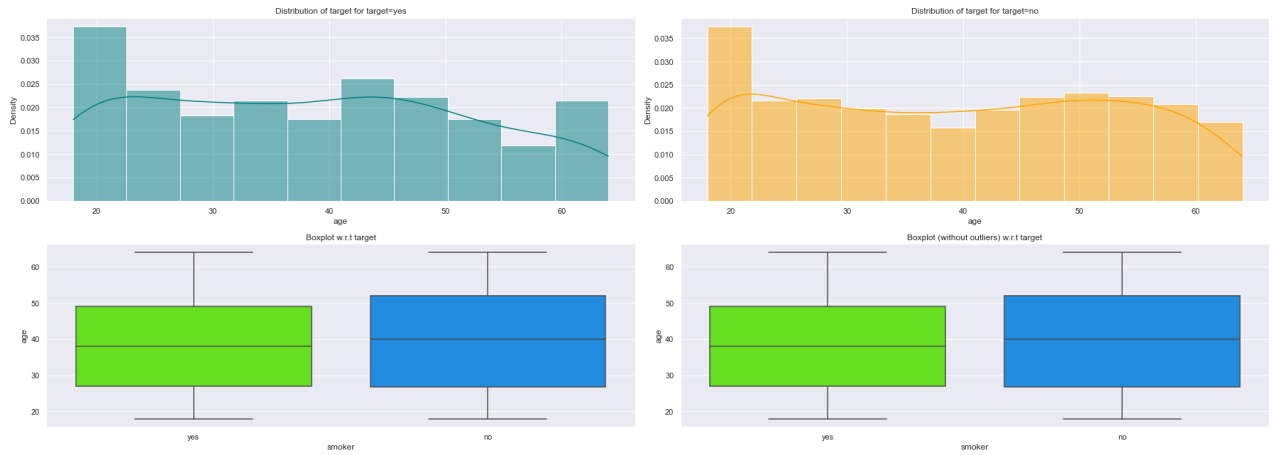


In [ ]:
```
# comparing Age vs smoker
data.groupby(['smoker'])['age'].describe()
```

Out[ ]:

smoker	count	mean	std	min	25%	50%	75%	max
no	1064.0	39.385338	14.083410	18.0	26.75	40.0	52.0	64.0
yes	274.0	38.514599	13.923186	18.0	27.00	38.0	49.0	64.0

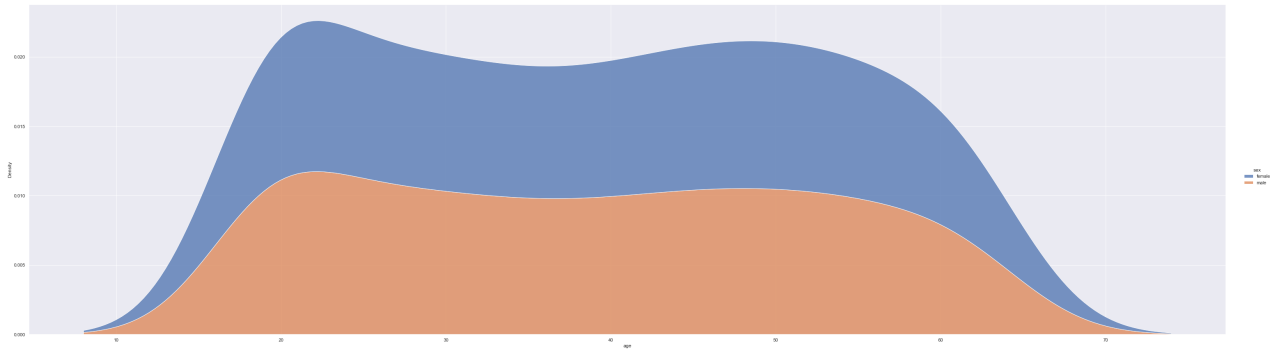
In [ ]:
```
# Age vs smoker
distribution_plot_wrt_target(data, 'age', 'smoker')
```



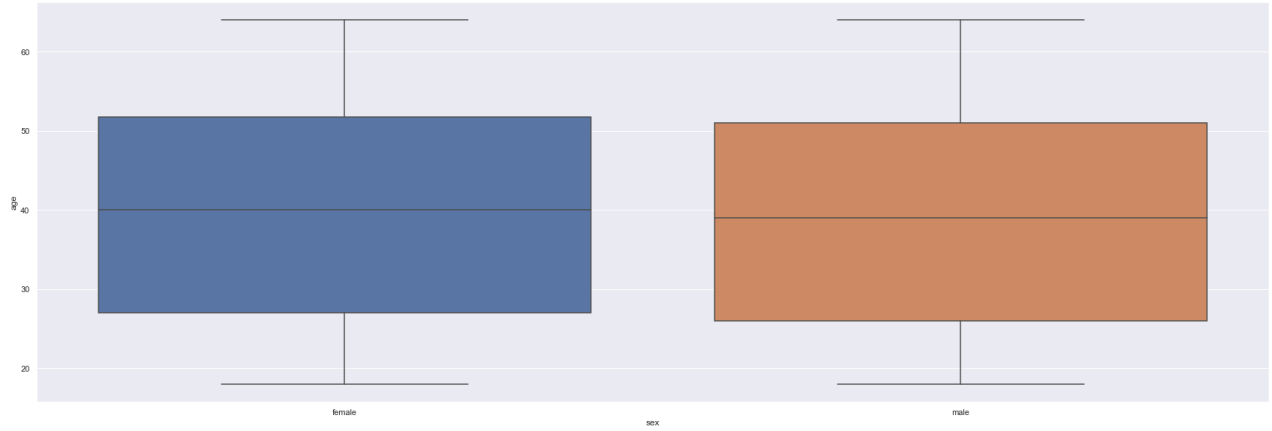
Age in Relation to Sex

In [ ]:
```
# Ploting a displot of Age vs Sex
sns.displot(
    data=df,
    x="age",
    hue="sex",
    multiple="stack",
    kind="kde",
    height=12,
```

```
    aspect=3.5,
);
```



In [ ]:
```
# Boxplot of Age vs sex
plt.figure(figsize=(30, 10))
sns.boxplot(data=df, x="sex", y="age");
```

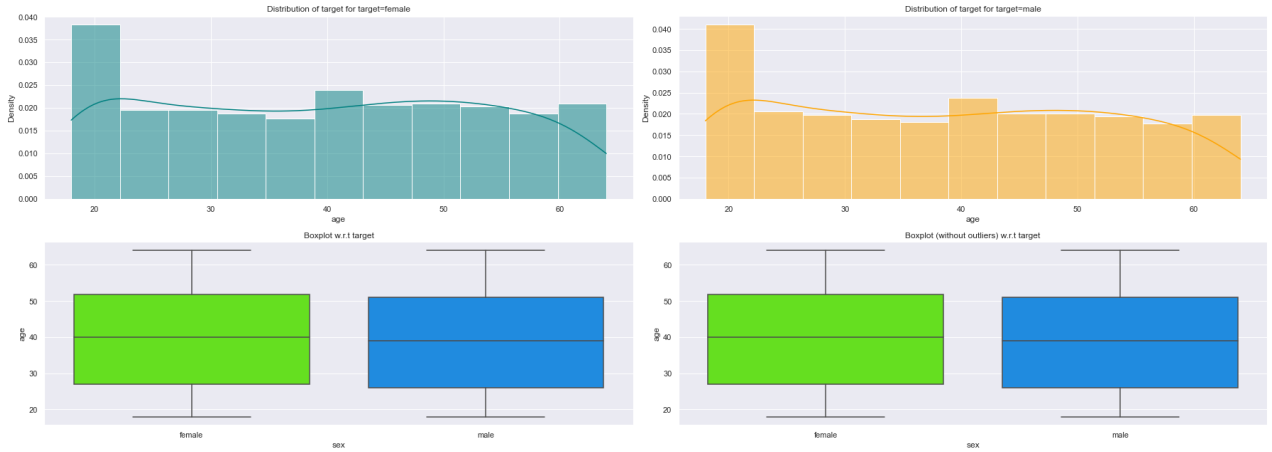


In [ ]:
```
# comparing Age vs sex
data.groupby(['sex'])['age'].describe()
```

Out[ ]:

sex	count	mean	std	min	25%	50%	75%	max
female	662.0	39.503021	14.054223	18.0	27.0	40.0	51.75	64.0
male	676.0	38.917160	14.050141	18.0	26.0	39.0	51.00	64.0

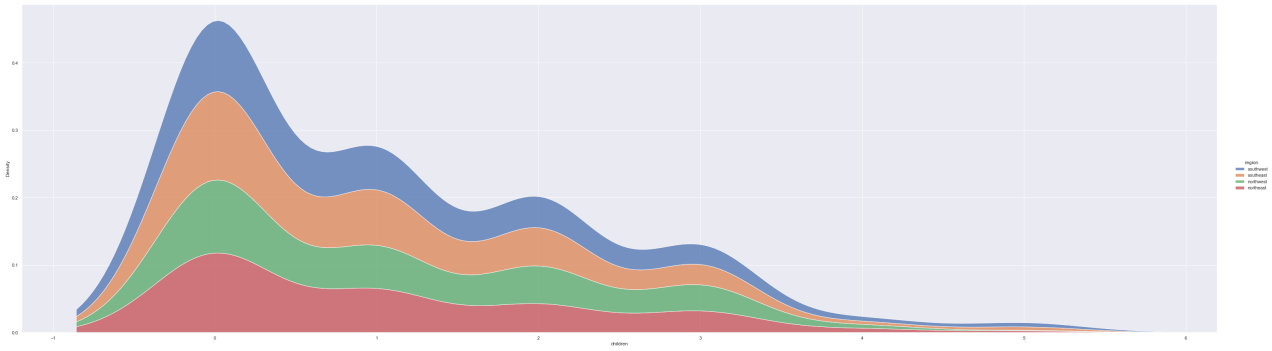
In [ ]:
```
# Age vs sex
distribution_plot_wrt_target(data, 'age', 'sex')
```



Number of Children in Relation to Region

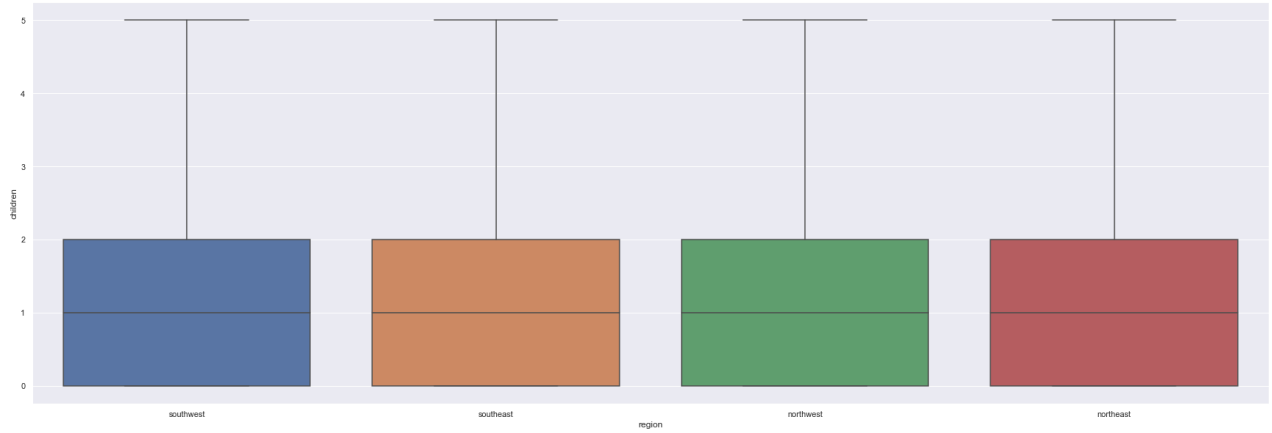
In [ ]:

```
# Ploting a displot of Children vs Region
sns.displot(
    data=df,
    x="children",
    hue="region",
    multiple="stack",
    kind="kde",
    height=12,
    aspect=3.5,
);
```



In [ ]:

```
# Boxplot of Children vs Region
plt.figure(figsize=(30, 10))
sns.boxplot(data=df, x="region", y="children");
```

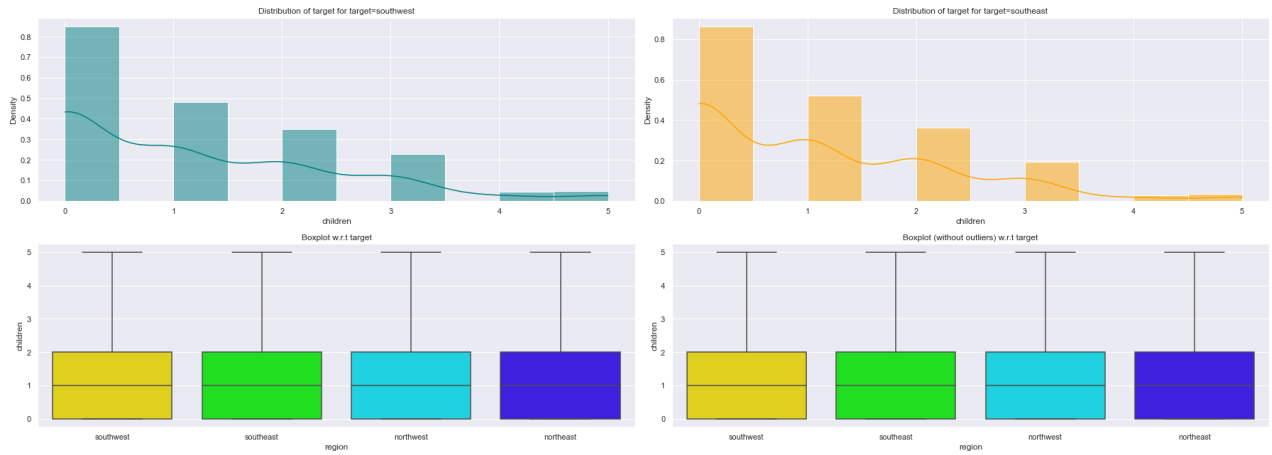


In [ ]:
```
# comparing Children vs Region
data.groupby(['region'])['children'].describe()
```

Out[ ]:

region	count	mean	std	min	25%	50%	75%	max
northeast	324.0	1.046296	1.198949	0.0	0.0	1.0	2.0	5.0
northwest	325.0	1.147692	1.171828	0.0	0.0	1.0	2.0	5.0
southeast	364.0	1.049451	1.177276	0.0	0.0	1.0	2.0	5.0
southwest	325.0	1.141538	1.275952	0.0	0.0	1.0	2.0	5.0

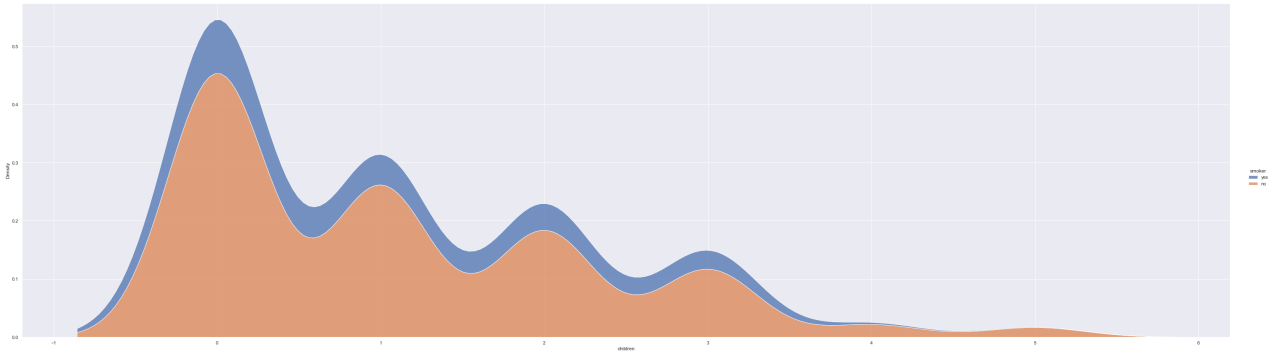
In [ ]:
```
# Children vs Region
distribution_plot_wrt_target(data, 'children', 'region')
```



Number of Children in Relation to smoker

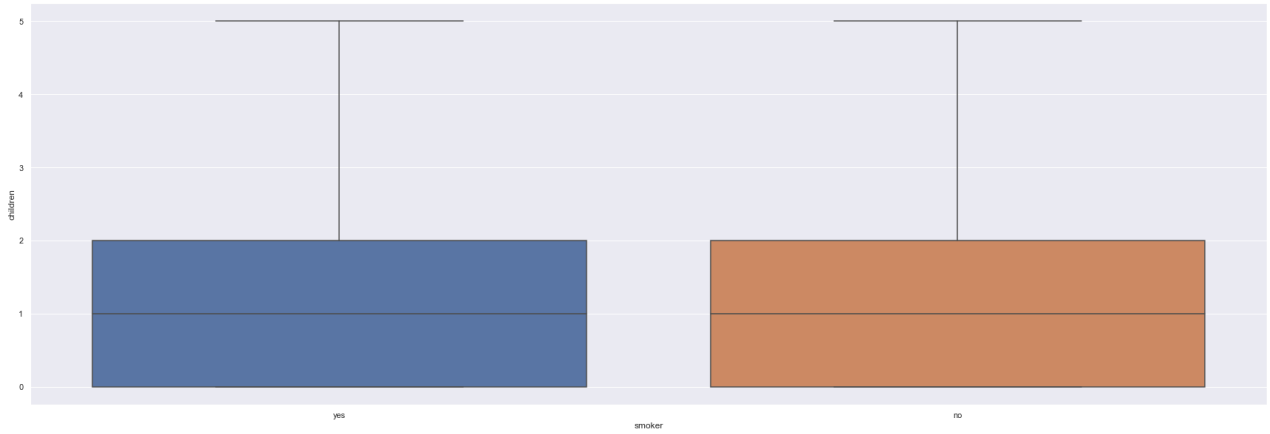
In [ ]:
```
# Ploting a displot of Children vs Smoker
sns.displot(
    data=df,
    x="children",
    hue="smoker",
    multiple="stack",
```

```
    kind="kde",
    height=12,
    aspect=3.5,
);
```



In [ ]:

```
# Boxplot of Children vs Smoker
plt.figure(figsize=(30, 10))
sns.boxplot(data=df, x="smoker", y="children");
```



In [ ]:

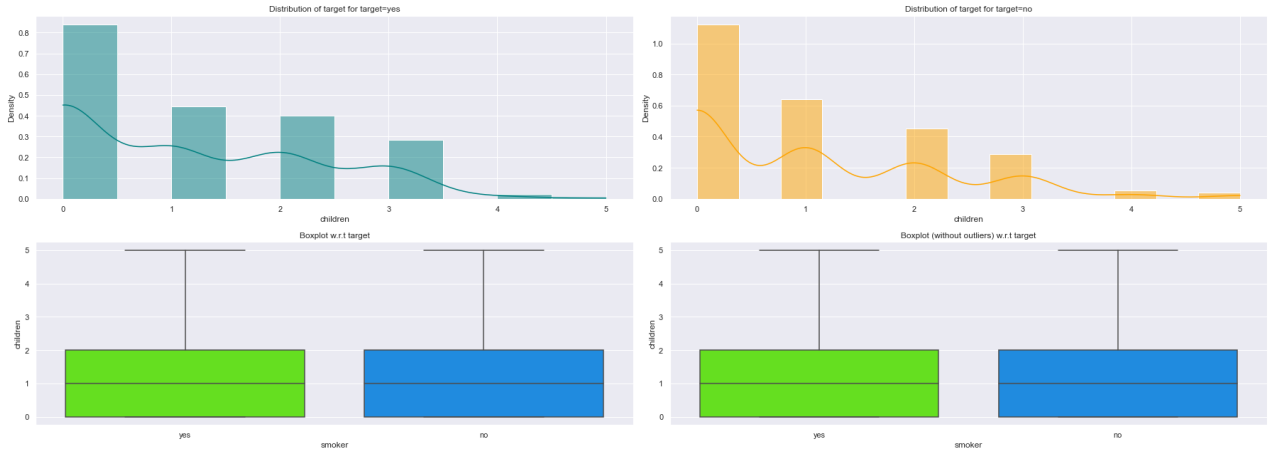
```
# comparing Children vs Smoker
data.groupby(['smoker'])['children'].describe()
```

Out[ ]:

smoker	count	mean	std	min	25%	50%	75%	max
no	1064.0	1.090226	1.218136	0.0	0.0	1.0	2.0	5.0
yes	274.0	1.113139	1.157066	0.0	0.0	1.0	2.0	5.0

In [ ]:

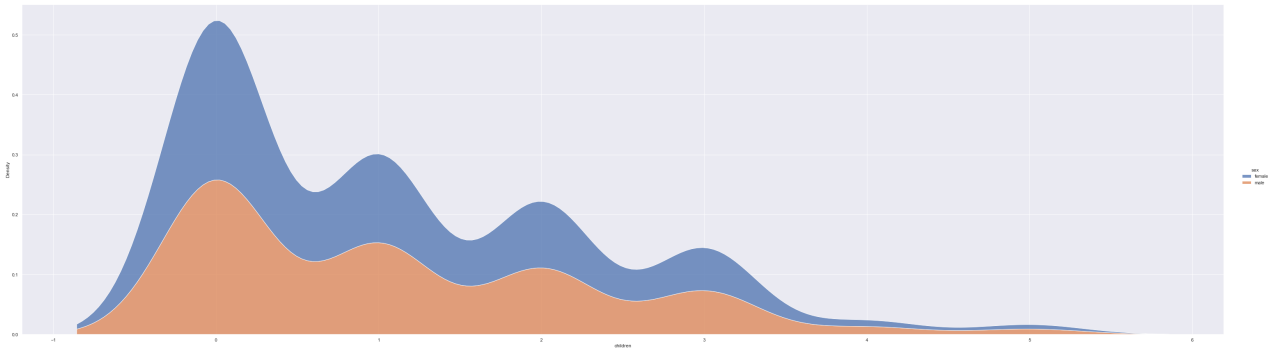
```
# Children vs Smoker
distribution_plot_wrt_target(data, 'children', 'smoker')
```



Number of Children in Relation to Sex

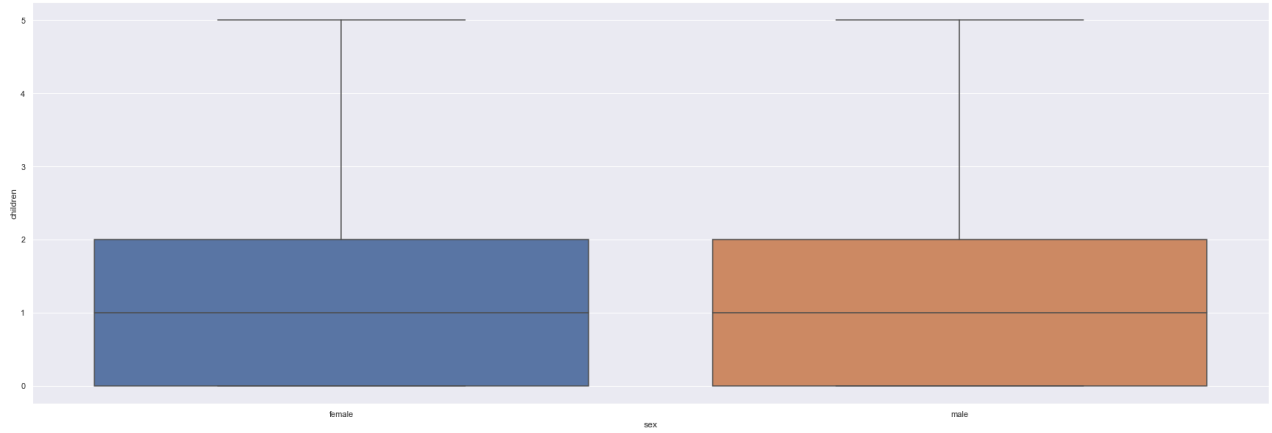
In [ ]:

```
# Ploting a displot of Children vs Sex
sns.displot(
    data=df,
    x="children",
    hue="sex",
    multiple="stack",
    kind="kde",
    height=12,
    aspect=3.5,
);
```



In [ ]:

```
# Boxplot of Children vs Sex
plt.figure(figsize=(30, 10))
sns.boxplot(data=df, x="sex", y="children");
```

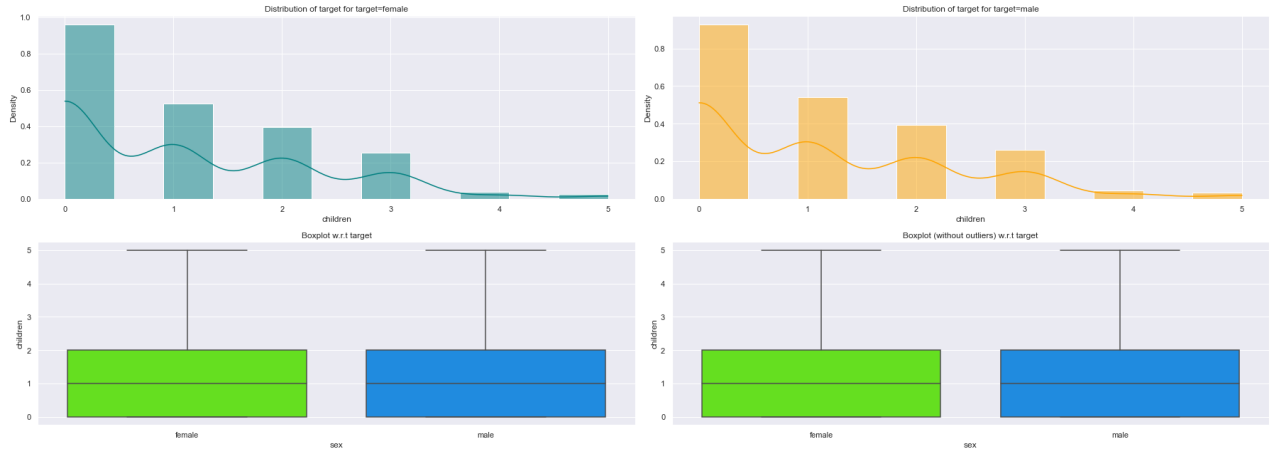


In [ ]:
```
# comparing Children vs Sex
data.groupby(['sex'])['children'].describe()
```

Out[ ]:

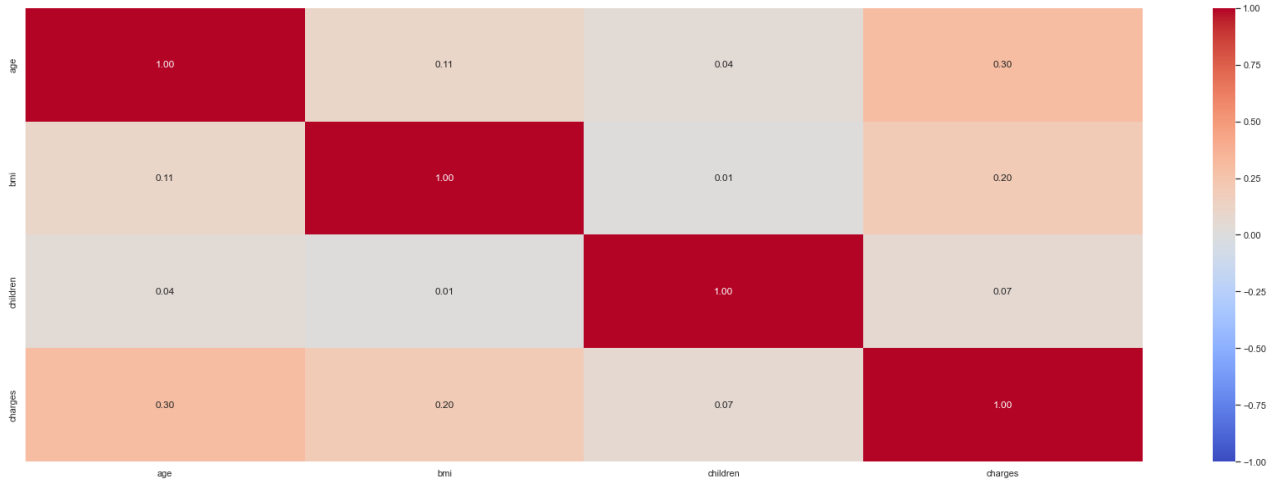
sex	count	mean	std	min	25%	50%	75%	max
female	662.0	1.074018	1.192115	0.0	0.0	1.0	2.0	5.0
male	676.0	1.115385	1.218986	0.0	0.0	1.0	2.0	5.0

In [ ]:
```
# Children vs Sex
distribution_plot_wrt_target(data, 'children', 'sex')
```



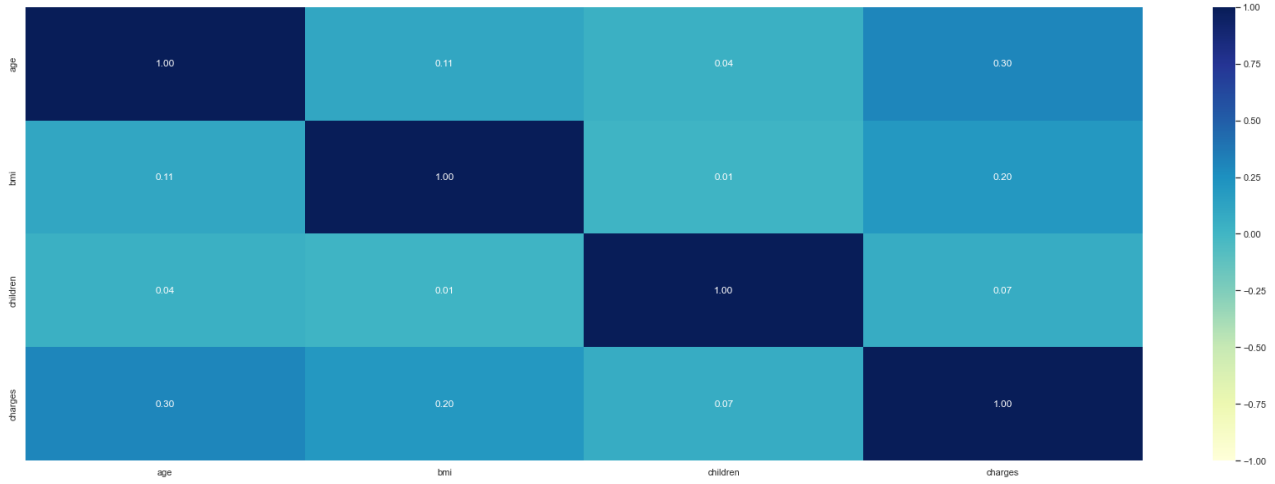
Correlation and Pairplot Analysis

In [ ]:
```
# Displaying the correlation between numerical variables of the dataset
plt.figure(figsize=(30, 10))
sns.heatmap(df.corr(), annot=True, vmin=-1, vmax=1, fmt=".2f", cmap="coolwarm")
plt.show()
```



In [ ]:

```
# creates heatmap showing correlation of numeric columns in data
plt.figure(figsize=(30, 10))
sns.heatmap(df.corr(), vmin=-1, vmax=1, cmap="YlGnBu", annot=True, fmt=".2f");
```

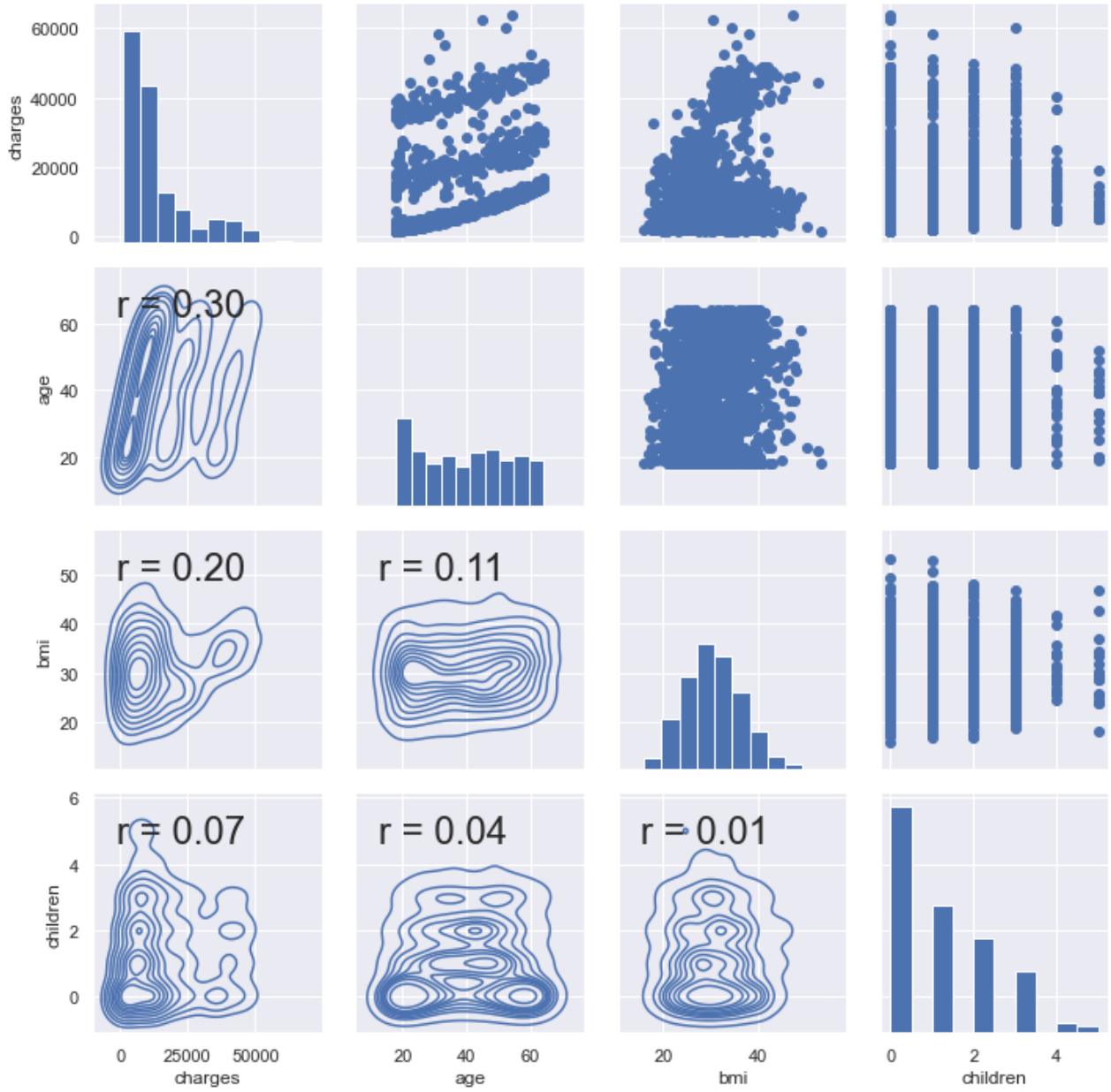


In [ ]:

```
import scipy

# Function to calculate correlation coefficient between two variables
def corrfunc(x, y, **kwgs):
    r = np.corrcoef(x, y)[0][1]
    ax = plt.gca()
    ax.annotate("r = {:.2f}".format(r),
                xy=(.1, .8), xycoords=ax.transAxes,
                size = 24)

# Create a PairGrid
g = sns.PairGrid(data = df,
                 vars = ['charges', 'age', 'bmi', 'children'])

# Map a scatterplot to the upper triangle
g.map_upper(plt.scatter)

# Map a histogram to the diagonal
g.map_diag(plt.hist)
```

```
# Map a kde plot to the lower triangle
g.map_lower(sns.kdeplot)

# Map the correlation coefficient to the lower diagonal
g.map_lower(corrfunc)
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x17cbc309ca0>
```



Data Preprocessing

Data Preparataion for model building

In [ ]:
```
# Separating features and the target column
X = data.drop('charges', axis=1)
y = data['charges']
```

```
X = pd.get_dummies(X,  columns=X.select_dtypes(include=["object", "category"]).columns.
    drop_first=True,
)
X.head()  ## Complete the code to create dummies
```

In [ ]:
```
# Splitting the data into train and test sets in 70:30 ratio
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=
```

In [ ]:
```
X_train.shape, X_test.shape
```

Out[ ]:
```
((936, 8), (402, 8))
```

In [ ]:
```
y.head()
```

Out[ ]:
```
0    16884.92400
1     1725.55230
2     4449.46200
3    21984.47061
4     3866.85520
Name: charges, dtype: float64
```

Bagging - Model Building and Hyperparameter Tuning

In [ ]:
```
# function to compute adjusted R-squared
def adj_r2_score(predictors, targets, predictions):
    r2 = r2_score(targets, predictions)
    n = predictors.shape[0]
    k = predictors.shape[1]
    return 1 - ((1 - r2) * (n - 1) / (n - k - 1))


# function to compute MAPE
def mape_score(targets, predictions):
    return np.mean(np.abs(targets - predictions) / targets) * 100


# function to compute different metrics to check performance of a regression model
def model_performance_regression(model, predictors, target):
    """
    Function to compute different metrics to check regression model performance

    model: regressor
    predictors: independent variables
    target: dependent variable
    """

    # predicting using the independent variables
    pred = model.predict(predictors)

    r2 = r2_score(target, pred)  # to compute R-squared
    adjr2 = adj_r2_score(predictors, target, pred)  # to compute adjusted R-squared
    rmse = np.sqrt(mean_squared_error(target, pred))  # to compute RMSE
```

```
    mae = mean_absolute_error(target, pred)  # to compute MAE
    mape = mape_score(target, pred)  # to compute MAPE

    # creating a dataframe of metrics
    df_perf = pd.DataFrame(
        {
            "RMSE": rmse,
            "MAE": mae,
            "R-squared": r2,
            "Adj. R-squared": adjr2,
            "MAPE": mape,
        },
        index=[0],
    )

    return df_perf
```

In [ ]:

```
##  Function to calculate r2_score and RMSE on train and test data
def get_model_score(model, flag=True):
    '''
    model : classifier to predict values of X

    '''
    # defining an empty list to store train and test results
    score_list=[]

    pred_train = model.predict(X_train)
    pred_test = model.predict(X_test)

    train_r2=metrics.r2_score(y_train,pred_train)
    test_r2=metrics.r2_score(y_test,pred_test)
    train_rmse=np.sqrt(metrics.mean_squared_error(y_train,pred_train))
    test_rmse=np.sqrt(metrics.mean_squared_error(y_test,pred_test))

    #Adding all scores in the list
    score_list.extend((train_r2,test_r2,train_rmse,test_rmse))

    # If the flag is set to True then only the following print statements will be dispa
    if flag==True:
        print("R-sqaure on training set : ",metrics.r2_score(y_train,pred_train))
        print("R-square on test set : ",metrics.r2_score(y_test,pred_test))
        print("RMSE on training set : ",np.sqrt(metrics.mean_squared_error(y_train,pred
        print("RMSE on test set : ",np.sqrt(metrics.mean_squared_error(y_test,pred_test

    # returning the list with train and test scores
    return score_list
```

Decision Tree Model

In [ ]:

```
dtree=DecisionTreeRegressor(random_state=1)
dtree.fit(X_train,y_train)
```

Out[ ]:

```
▾ DecisionTreeRegressor

DecisionTreeRegressor(random_state=1)
```

In [ ]:
```
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
```

In [ ]:
```
dtree_model_train_perf=model_performance_regression(dtree, X_train,y_train)
print("Training performance \n",dtree_model_train_perf)
```

```
Training performance
    RMSE  MAE  R-squared  Adj. R-squared  MAPE
0   0.0  0.0        1.0             1.0   0.0
```

In [ ]:
```
dtree_model_test_perf=model_performance_regression(dtree, X_test,y_test)
print("Testing performance \n",dtree_model_test_perf)
```

```
Testing performance
           RMSE        MAE  R-squared  Adj. R-squared       MAPE
0  6552.935061  3126.1233   0.696994        0.690826  36.211025
```

- The Decision tree model with default parameters is overfitting the train data.
- Let's see if we can reduce overfitting and improve performance on test data by tuning hyperparameters.

Hyperparameter Tuning

In [ ]:
```
# Choose the type of classifier.
dtree_tuned = DecisionTreeRegressor(random_state=1)

# Grid of parameters to choose from
parameters = {'max_depth': list(np.arange(2,20)) + [None],
              'min_samples_leaf': [1, 3, 5, 7, 10],
              'max_leaf_nodes' : [2, 3, 5, 10, 15] + [None],
              'min_impurity_decrease': [0.001, 0.01, 0.1, 0.0]
             }

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(dtree_tuned, parameters, scoring=scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
dtree_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
dtree_tuned.fit(X_train, y_train)
```

Out[ ]:
```
DecisionTreeRegressor(max_depth=4, max_leaf_nodes=15,
                      min_impurity_decrease=0.001, min_samples_leaf=10,
                      random_state=1)
```

In [ ]:
```
dtree_tuned_model_train_perf = model_performance_regression(dtree_tuned, X_train,y_trai
```

```
print("Training performance \n",dtree_tuned_model_train_perf)
```

```
Training performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  4424.516926  2563.840942   0.868275        0.867138  31.244158
```

In [ ]:

```
dtree_tuned_model_test_perf = model_performance_regression(dtree_tuned, X_test,y_test)
print("Testing performance \n",dtree_tuned_model_test_perf)
```

```
Testing performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  4575.947376  2574.906191   0.852245        0.849238  30.527671
```

Plotting the feature importance of each variable

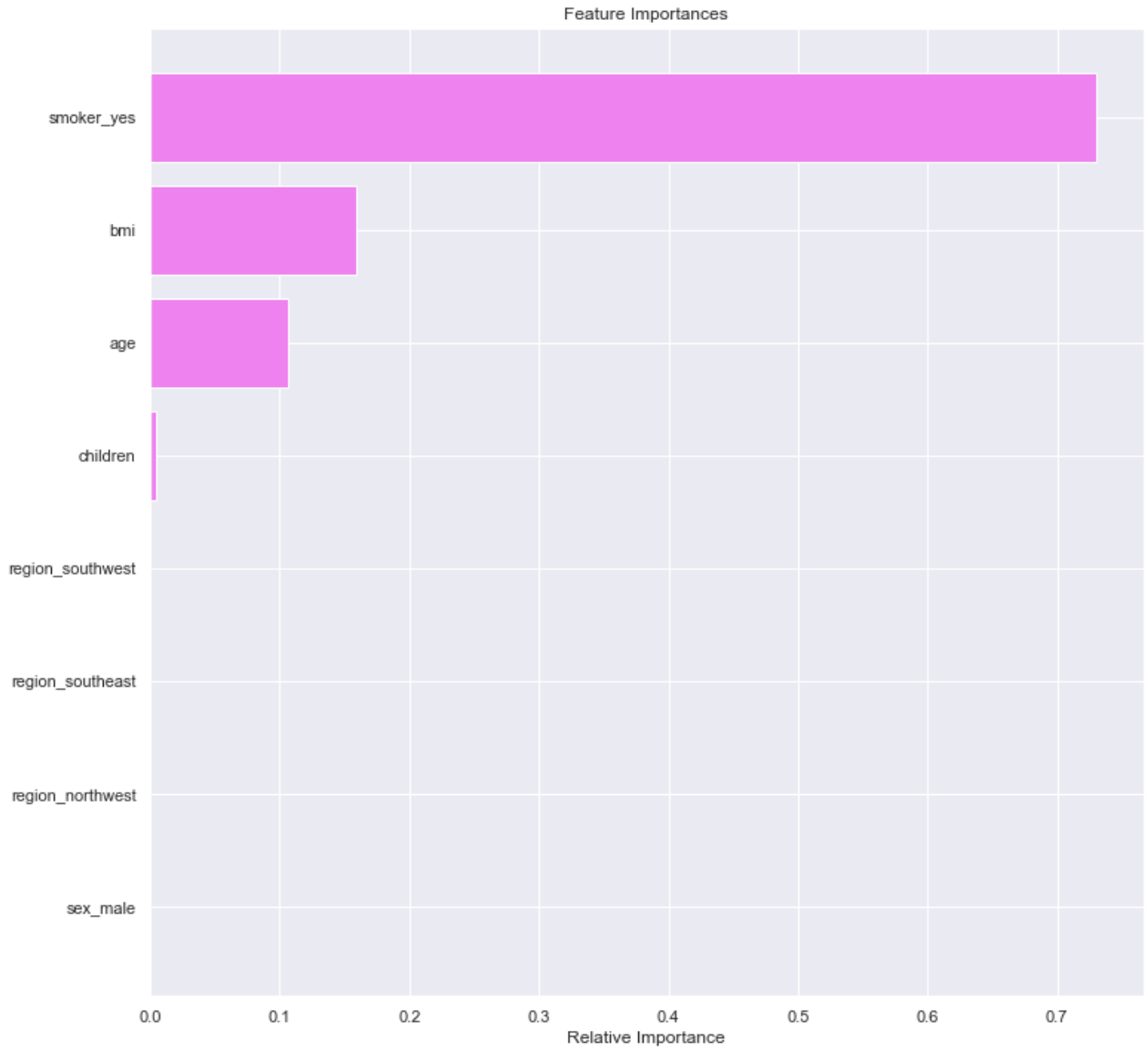
In [ ]:

```
# importance of features in the tree building ( The importance of a feature is computed
#(normalized) total reduction of the criterion brought by that feature. It is also know

print(pd.DataFrame(dtree_tuned.feature_importances_, columns = ["Imp"], index = X_train
```

```
                     Imp
smoker_yes      0.729986
bmi             0.158917
age             0.106619
children        0.004478
sex_male        0.000000
region_northwest  0.000000
region_southeast  0.000000
region_southwest  0.000000
```

In [ ]:

```
feature_names = X_train.columns
importances = dtree_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



- hr is the most important feature, in addition to temp and yr, for tuned decision tree model

Random Forest Model

In [ ]:
```
rf_estimator=RandomForestRegressor(random_state=1)
rf_estimator.fit(X_train,y_train)
```

Out[ ]:
```
RandomForestRegressor
RandomForestRegressor(random_state=1)
```

In [ ]:
```
rf_estimator_model_train_perf = model_performance_regression(rf_estimator, X_train,y_tr
print("Training performance \n",rf_estimator_model_train_perf)
```

```
Training performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  1883.056551  1051.740727    0.97614        0.975935  13.031236
```

In [ ]:
```
rf_estimator_model_test_perf = model_performance_regression(rf_estimator, X_test,y_test
```

```
print("Testing performance \n",rf_estimator_model_test_perf)
```

```
Testing performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  4928.605414  2801.110497   0.828594        0.825104  33.403338
```

- Random forest is giving a good r2 score of 94% on the test data but it is slightly overfitting the train data.
- Let's try to reduce this overfitting by hyperparameter tuning.

Hyperparameter Tuning

In [ ]:
```
# Choose the type of classifier.
rf_tuned = RandomForestRegressor(random_state=1)

# Grid of parameters to choose from
parameters = {
                'max_depth':[4, 6, 8, 10, None],
                'max_features': ['sqrt','log2',None],
                'n_estimators': [80, 90, 100, 110, 120]
}

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(rf_tuned, parameters, scoring=scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
rf_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
rf_tuned.fit(X_train, y_train)
```

Out[ ]:
```
RandomForestRegressor(max_depth=4, max_features=None, n_estimators=120,
                      random_state=1)
```

In [ ]:
```
rf_tuned_model_train_perf = model_performance_regression(rf_tuned, X_train, y_train)
print("Training performance \n",rf_tuned_model_train_perf)
```

```
Training performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  4267.891894  2428.453898   0.877436        0.876378  30.931194
```

In [ ]:
```
rf_tuned_model_test_perf = model_performance_regression(rf_tuned, X_test, y_test)
print("Testing performance \n",rf_tuned_model_test_perf)
```

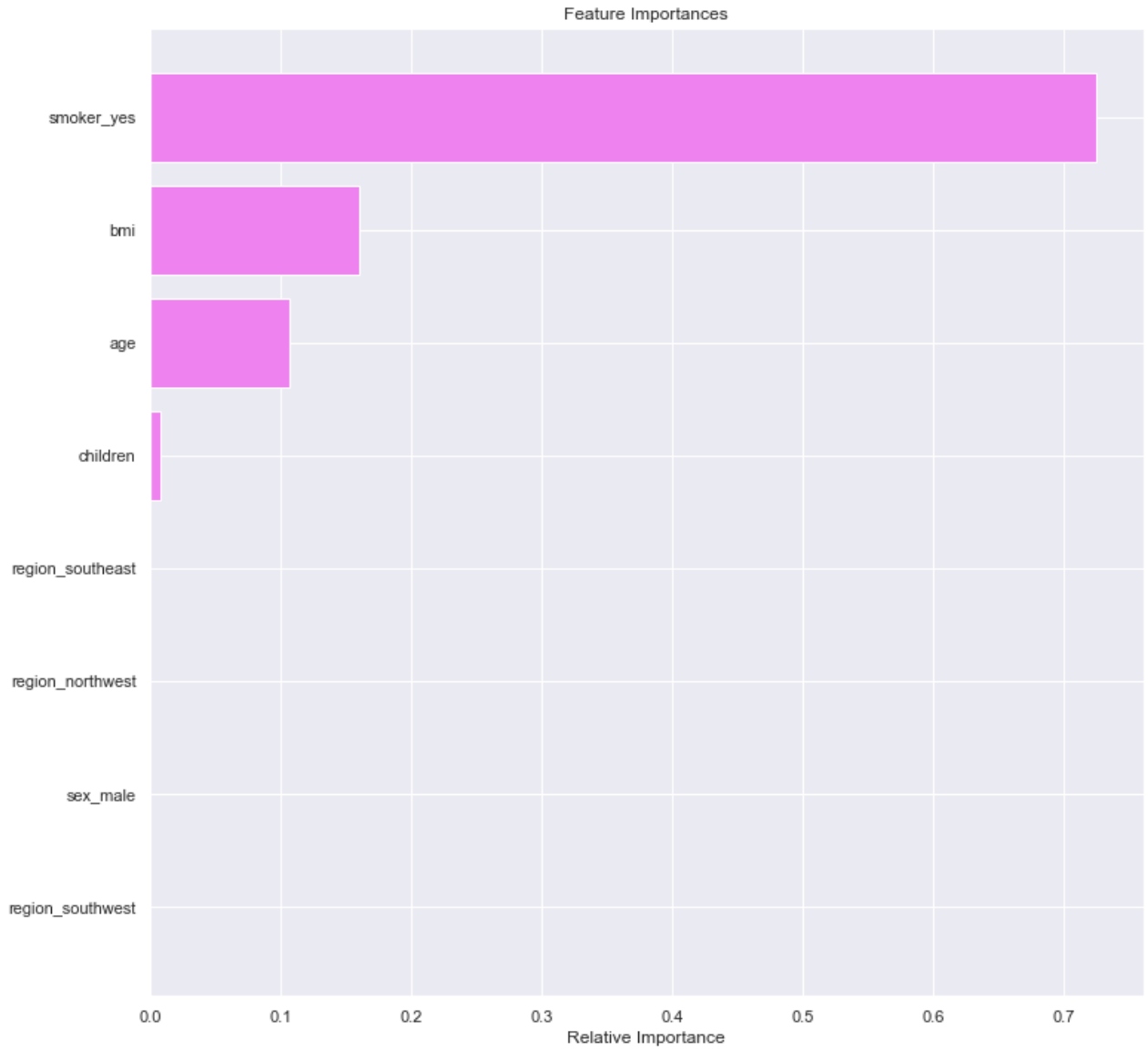
```
Testing performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  4512.976949  2492.230058   0.856284        0.853358  30.755716
```

- No significant change in the result. The result is almost the same before or after the hyperparameter tuning.

In [ ]:
```
# importance of features in the tree building ( The importance of a feature is computed
#(normalized) total reduction of the criterion brought by that feature. It is also know

print(pd.DataFrame(rf_tuned.feature_importances_, columns = ["Imp"], index = X_train.co
```

```
                     Imp
smoker_yes      0.724633
bmi             0.160317
age             0.107047
children        0.007396
region_southeast  0.000195
region_northwest  0.000184
sex_male        0.000164
region_southwest  0.000063
```

In [ ]:
```
feature_names = X_train.columns
importances = rf_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



Boosting - Model Building and Hyperparameter Tuning

AdaBoost Regressor

In [ ]:
```
ab_regressor=AdaBoostRegressor(random_state=1)
ab_regressor.fit(X_train,y_train)
```

Out[ ]:
```
AdaBoostRegressor

AdaBoostRegressor(random_state=1)
```

In [ ]:
```
ab_regressor_model_train_perf = model_performance_regression(ab_regressor, X_train,y_tr
print("Training performance \n",ab_regressor_model_train_perf)
```

```
Training performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  4823.327249  3667.453141   0.843458        0.842107  61.908061
```

In [ ]:

```
ab_regressor_model_test_perf = model_performance_regression(ab_regressor, X_test,y_test
print("Testing performance \n",ab_regressor_model_test_perf)
```

```
Testing performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  5041.805227  3813.516982   0.820629        0.816978  63.698328
```

Hyperparameter Tuning

In [ ]:
```
# Choose the type of classifier.
ab_tuned = AdaBoostRegressor(random_state=1)

# Grid of parameters to choose from
parameters = {'n_estimators': np.arange(10,100,10),
              'learning_rate': [1, 0.1, 0.5, 0.01],
              }

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(ab_tuned, parameters, scoring=scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
ab_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
ab_tuned.fit(X_train, y_train)
```

Out[ ]:
```
AdaBoostRegressor(learning_rate=0.01, n_estimators=40, random_state=1)
```

In [ ]:
```
ab_tuned_model_train_perf = model_performance_regression(ab_tuned, X_train,y_train)
print("Training performance \n",ab_tuned_model_train_perf)
```

```
Training performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  4530.377367  2805.837375   0.861896        0.860705  38.454285
```

In [ ]:
```
ab_tuned_model_test_perf = model_performance_regression(ab_tuned, X_test,y_test)
print("Testing performance \n",ab_tuned_model_test_perf)
```

```
Testing performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  4634.50077  2768.702279    0.84844        0.845355  38.517113
```

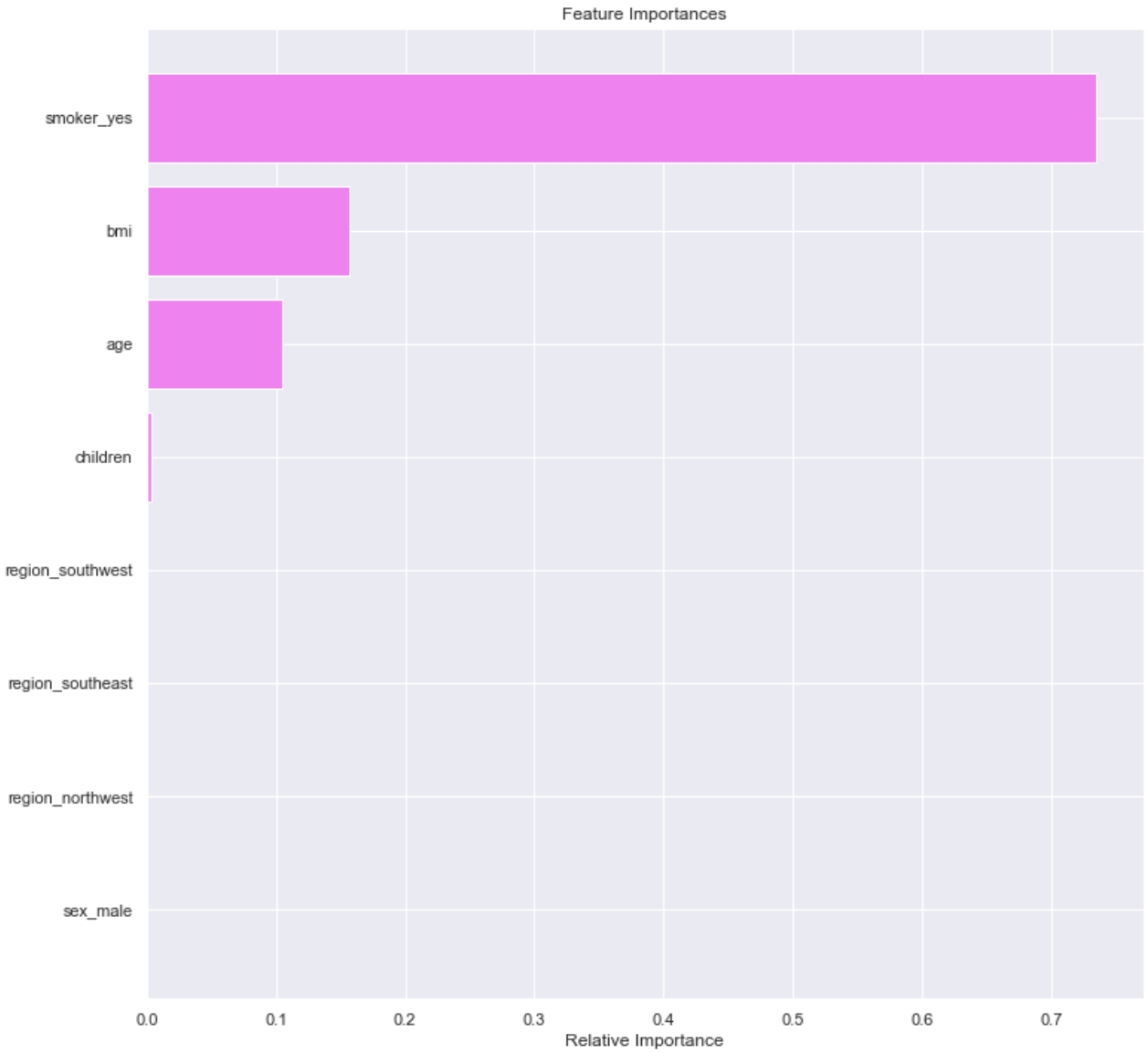
In [ ]:
```
# importance of features in the tree building

print(pd.DataFrame(ab_tuned.feature_importances_, columns = ["Imp"], index = X_train.co
```

```
                  Imp
smoker_yes   0.734671
bmi          0.156457
```

```
age               0.105257
children          0.003615
sex_male          0.000000
region_northwest  0.000000
region_southeast  0.000000
region_southwest  0.000000
```

In [ ]:

```
feature_names = X_train.columns
importances = ab_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



Gradient Boosting Regressor

In [ ]:

```
gb_estimator=GradientBoostingRegressor(random_state=1)
```

```
gb_estimator.fit(X_train,y_train)
```

Out[ ]:
```
GradientBoostingRegressor(random_state=1)
```

In [ ]:
```
gb_estimator_model_train_perf = model_performance_regression(gb_estimator, X_train,y_tr
print("Training performance \n",gb_estimator_model_train_perf)
```

```
Training performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  3734.365029  2054.952053   0.906164        0.905354  25.576065
```

In [ ]:
```
gb_estimator_model_test_perf = model_performance_regression(gb_estimator, X_test, y_tes
print("Testing performance \n",gb_estimator_model_test_perf)
```

```
Testing performance
          RMSE          MAE  R-squared  Adj. R-squared      MAPE
0  4496.06822  2481.420389   0.857359        0.854455  29.40346
```

Hyperparameter Tuning

In [ ]:
```
# Choose the type of classifier.
gb_tuned = GradientBoostingRegressor(random_state=1)

# Grid of parameters to choose from
parameters = {'n_estimators': np.arange(50,200,25),
              'subsample':[0.7,0.8,0.9,1],
              'max_features':[0.7,0.8,0.9,1],
              'max_depth':[3,5,7,10]
              }

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(gb_tuned, parameters, scoring=scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
gb_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
gb_tuned.fit(X_train, y_train)
```

Out[ ]:
```
GradientBoostingRegressor(max_features=0.8, n_estimators=50, random_state=1,
                          subsample=1)
```

In [ ]:
```
gb_tuned_model_train_perf = model_performance_regression(gb_tuned, X_train,y_train)
print("Training performance \n",gb_tuned_model_train_perf)
```

```
Training performance
```

```
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  4111.338582  2309.011179   0.886263        0.885281  30.148772
```

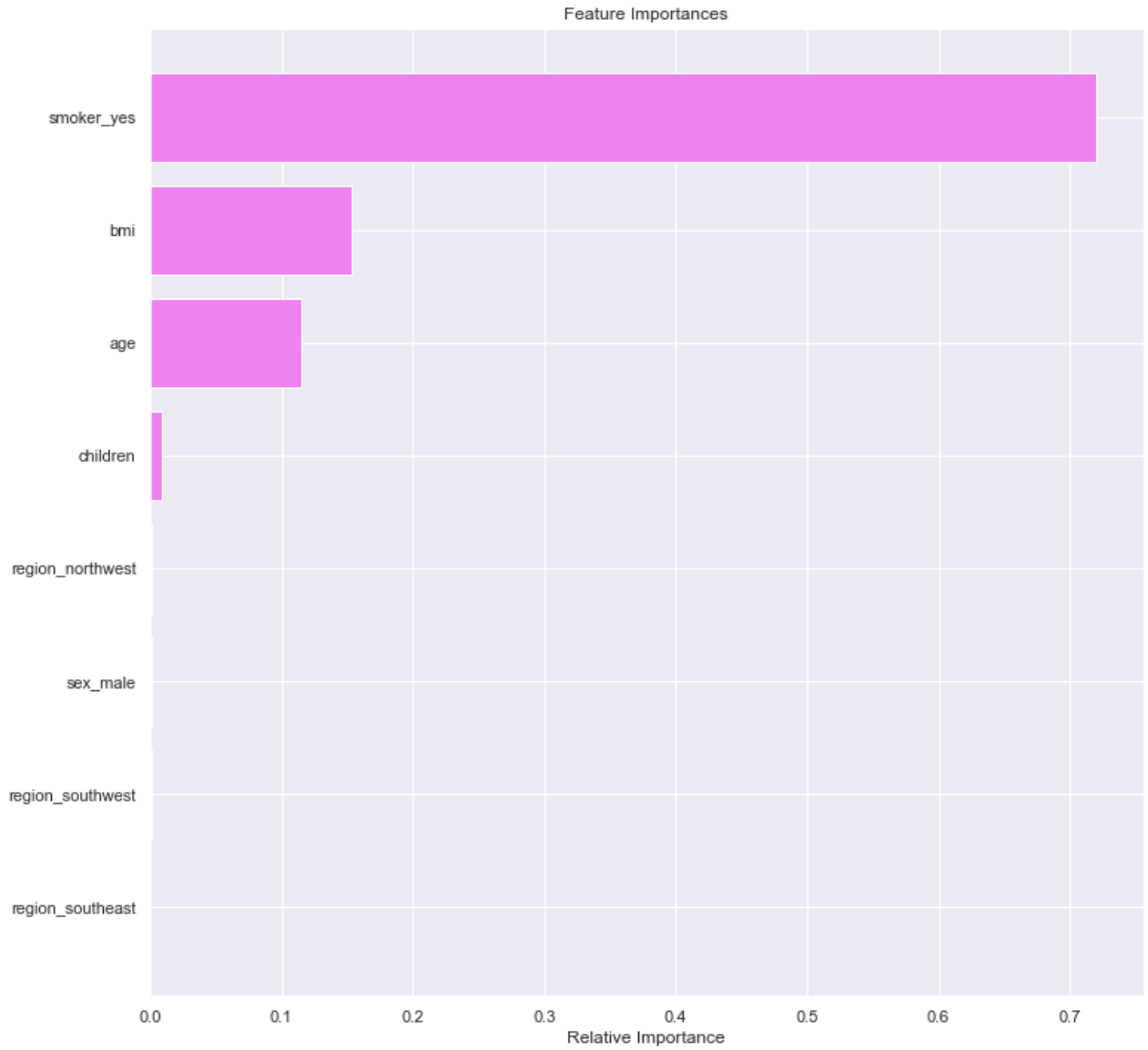
In [ ]:
```
gb_tuned_model_test_perf = model_performance_regression(gb_tuned, X_test, y_test)
print("Testing performance \n",gb_tuned_model_test_perf)
```

```
Testing performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  4441.722267  2463.022505   0.860786        0.857952  30.820256
```

In [ ]:
```
# importance of features in the tree building ( The importance of a feature is computed
#(normalized) total reduction of the criterion brought by that feature. It is also know

print(pd.DataFrame(gb_tuned.feature_importances_, columns = ["Imp"], index = X_train.co
```

```
                    Imp
smoker_yes        0.719894
bmi               0.153856
age               0.115253
children          0.008707
region_northwest  0.000835
sex_male          0.000607
region_southwest  0.000538
region_southeast  0.000310
```

In [ ]:
```
feature_names = X_train.columns
importances = gb_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



XGBoost Regressor

In [ ]:
```
xgb_estimator=XGBRegressor(random_state=1, verbosity = 0)
xgb_estimator.fit(X_train,y_train)
```

Out[ ]:

XGBRegressor

```
XGBRegressor(base_score=0.5, booster='gbtree', callbacks=None,
             colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
             early_stopping_rounds=None, enable_categorical=False,
             eval_metric=None, gamma=0, gpu_id=-1, grow_policy='depthwise',
             importance_type=None, interaction_constraints='',
             learning_rate=0.300000012, max_bin=256, max_cat_to_onehot=4,
             max_delta_step=0, max_depth=6, max_leaves=0, min_child_weight=
1,
             missing=nan, monotone_constraints='()', n_estimators=100, n_job
s=0,
             num_parallel_tree=1, predictor='auto', random_state=1, reg_alph
```

In [ ]:
```
xgb_estimator_model_train_perf = model_performance_regression(xgb_estimator, X_train, y
print("Training performance \n",xgb_estimator_model_train_perf)
```

```
Training performance
         RMSE         MAE  R-squared  Adj. R-squared      MAPE
0  704.30054  416.407284   0.996662        0.996633  5.838824
```

In [ ]:
```
xgb_estimator_model_test_perf = model_performance_regression(xgb_estimator, X_test,y_te
print("Testing performance \n",xgb_estimator_model_test_perf)
```

```
Testing performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  5095.029528  2905.879545   0.816822        0.813094  37.670149
```

Hyperparameter Tuning

In [ ]:
```
# Choose the type of classifier.
xgb_tuned = XGBRegressor(random_state=1, verbosity = 0)

# Grid of parameters to choose from
parameters = {'n_estimators': [75,100,125,150],
              'subsample':[0.7, 0.8, 0.9, 1],
              'gamma':[0, 1, 3, 5],
              'colsample_bytree':[0.7, 0.8, 0.9, 1],
              'colsample_bylevel':[0.7, 0.8, 0.9, 1]
              }

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(xgb_tuned, parameters, scoring=scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
xgb_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
xgb_tuned.fit(X_train, y_train)
```

Out[ ]:
```
XGBRegressor(base_score=0.5, booster='gbtree', callbacks=None,
             colsample_bylevel=0.7, colsample_bynode=1, colsample_bytree=0.
9,
             early_stopping_rounds=None, enable_categorical=False,
             eval_metric=None, gamma=0, gpu_id=-1, grow_policy='depthwise',
             importance_type=None, interaction_constraints='',
             learning_rate=0.300000012, max_bin=256, max_cat_to_onehot=4,
             max_delta_step=0, max_depth=6, max_leaves=0, min_child_weight=
1,
             missing=nan, monotone_constraints='()', n_estimators=75, n_jobs
=0,
```

In [ ]:
```
xgb_tuned_model_train_perf = model_performance_regression(xgb_tuned, X_train, y_train)
```

```
print("Training performance \n",xgb_tuned_model_train_perf)
```

```
Training performance
           RMSE         MAE  R-squared  Adj. R-squared       MAPE
0  1216.787536  731.513653   0.990038        0.989952  10.684835
```

In [ ]:

```
xgb_tuned_model_test_perf = model_performance_regression(xgb_tuned, X_test, y_test)
print("Testing performance \n",xgb_tuned_model_test_perf)
```

```
Testing performance
          RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  5174.31062  3075.994602   0.811077        0.807232  40.450388
```

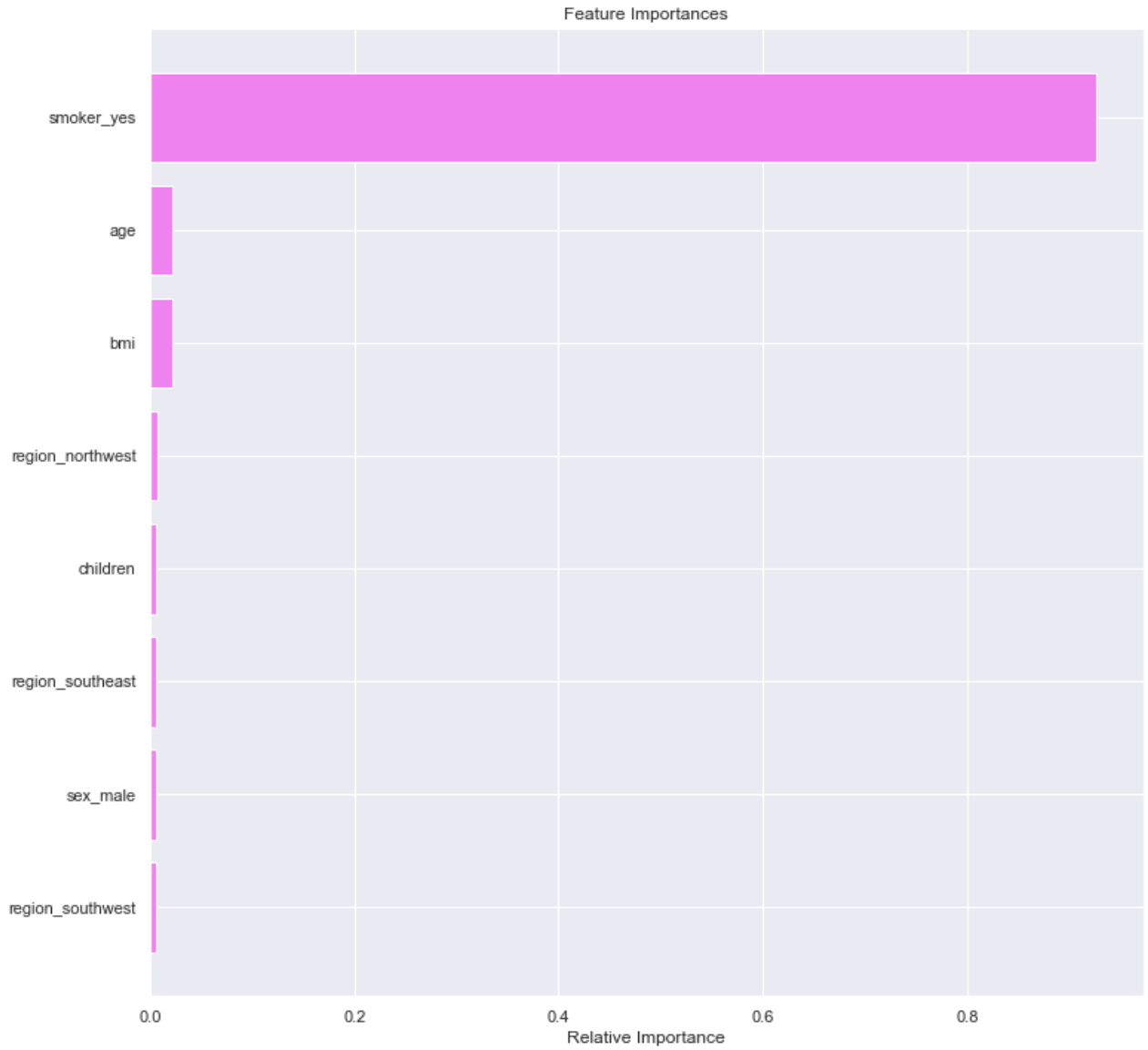
In [ ]:

```
# importance of features in the tree building ( The importance of a feature is computed
#(normalized) total reduction of the criterion brought by that feature. It is also know

print(pd.DataFrame(xgb_tuned.feature_importances_, columns = ["Imp"], index = X_train.c
```

```
                     Imp
smoker_yes      0.926460
age             0.022242
bmi             0.021028
region_northwest  0.006797
children        0.006268
region_southeast  0.006264
sex_male        0.005919
region_southwest  0.005022
```

In [ ]:

```
feature_names = X_train.columns
importances = xgb_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



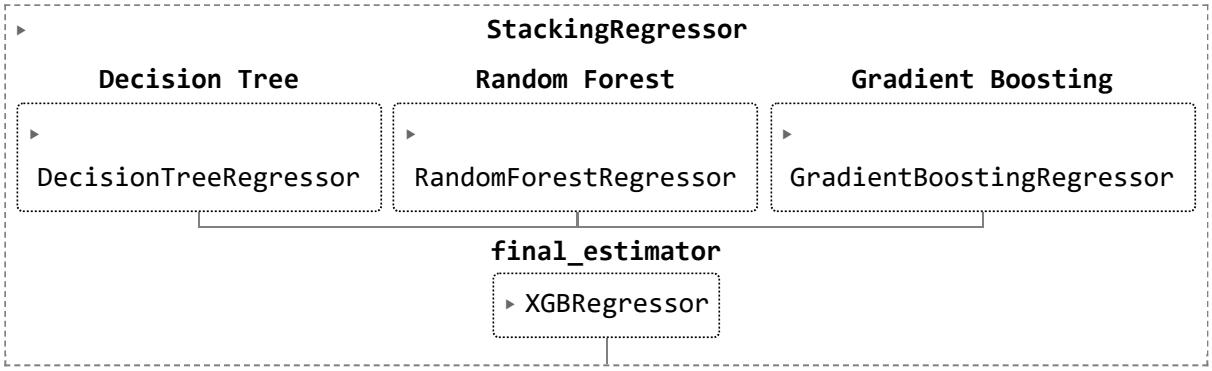
Stacking Model

Now, let's build a stacking model with the tuned models - decision tree, random forest, and gradient boosting, then use XGBoost to get the final prediction.

In [ ]:
```
estimators=[('Decision Tree', dtree_tuned),('Random Forest', rf_tuned),
            ('Gradient Boosting', gb_tuned)]
final_estimator=XGBRegressor(random_state=1)
```

In [ ]:
```
stacking_estimator=StackingRegressor(estimators=estimators, final_estimator=final_estim
stacking_estimator.fit(X_train,y_train)
```

Out[ ]:



In [ ]:
```
stacking_estimator_model_train_perf = model_performance_regression(stacking_estimator,
print("Training performance \n",stacking_estimator_model_train_perf)
```

```
Training performance
           RMSE         MAE  R-squared  Adj. R-squared       MAPE
0  4932.568761  2714.85011   0.836287        0.834874  31.640557
```

In [ ]:
```
stacking_estimator_model_test_perf = model_performance_regression(stacking_estimator, X
print("Testing performance \n",stacking_estimator_model_test_perf)
```

```
Testing performance
           RMSE          MAE  R-squared  Adj. R-squared       MAPE
0  5199.363611  2744.985882   0.809243         0.80536  28.680015
```

Comparing all models

In [ ]:
```
# training performance comparison

models_train_comp_df = pd.concat(
    [dtree_model_train_perf.T, dtree_tuned_model_train_perf.T, rf_estimator_model_train
    ab_regressor_model_train_perf.T,ab_tuned_model_train_perf.T,gb_estimator_model_trai
    xgb_estimator_model_train_perf.T,xgb_tuned_model_train_perf.T,stacking_estimator_mo
    axis=1,
)

models_train_comp_df.columns = [
    "Decision Tree",
    "Decision Tree Tuned",
    "Random Forest Estimator",
    "Random Forest Tuned",
    "Adaboost Regressor",
    "Adaboost Tuned",
    "Gradient Boost Estimator",
    "Gradient Boost Tuned",
    "XGB",
    "XGB Tuned",
    "Stacking Classifier"
]

print("Training performance comparison:")
models_train_comp_df
```

```
Training performance comparison:
```

Out[ ]:

	Decision Tree	Decision Tree Tuned	Random Forest Estimator	Random Forest Tuned	Adaboost Regressor	Adaboost Tuned	Gradient Boost Estimator	Gr
RMSE	0.0	4424.516926	1883.056551	4267.891894	4823.327249	4530.377367	3734.365029	4111.
MAE	0.0	2563.840942	1051.740727	2428.453898	3667.453141	2805.837375	2054.952053	2309.
R-squared	1.0	0.868275	0.976140	0.877436	0.843458	0.861896	0.906164	0.
Adj. R-squared	1.0	0.867138	0.975935	0.876378	0.842107	0.860705	0.905354	0.
MAPE	0.0	31.244158	13.031236	30.931194	61.908061	38.454285	25.576065	30.

In [ ]:

```
# Testing performance comparison

models_test_comp_df = pd.concat(
    [dtree_model_test_perf.T, dtree_tuned_model_test_perf.T, rf_estimator_model_test_pe
    ab_regressor_model_test_perf.T,ab_tuned_model_test_perf.T,gb_estimator_model_test_p
    xgb_estimator_model_test_perf.T,xgb_tuned_model_test_perf.T,stacking_estimator_mode
    axis=1,
)

models_test_comp_df.columns = [
    "Decision Tree",
    "Decision Tree Tuned",
    "Random Forest Estimator",
    "Random Forest Tuned",
    "Adaboost Regressor",
    "Adaboost Tuned",
    "Gradient Boost Estimator",
    "Gradient Boost Tuned",
    "XGB",
    "XGB Tuned",
    "Stacking Classifier"
]

print("Testing performance comparison:")
models_test_comp_df
```

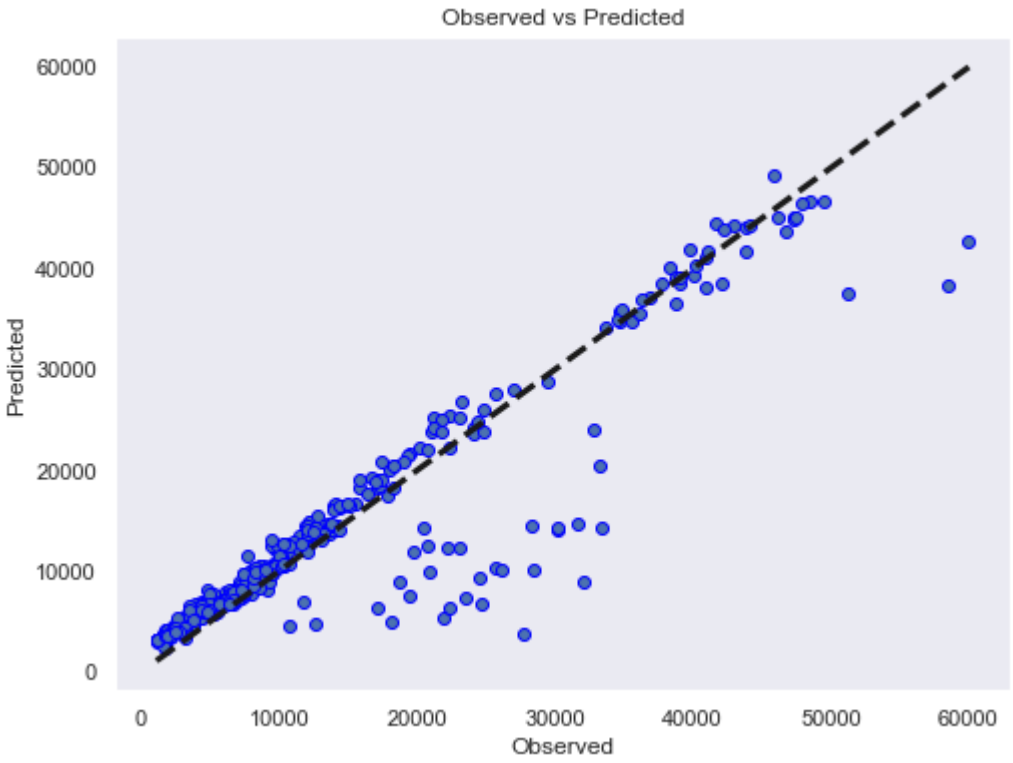
```
Testing performance comparison:
```

Out[ ]:

	Decision Tree	Decision Tree Tuned	Random Forest Estimator	Random Forest Tuned	Adaboost Regressor	Adaboost Tuned	Gradient Boost Estimator	
RMSE	6552.935061	4575.947376	4928.605414	4512.976949	5041.805227	4634.500770	4496.068220	44
MAE	3126.123300	2574.906191	2801.110497	2492.230058	3813.516982	2768.702279	2481.420389	24
R-squared	0.696994	0.852245	0.828594	0.856284	0.820629	0.848440	0.857359	
Adj. R-squared	0.690826	0.849238	0.825104	0.853358	0.816978	0.845355	0.854455	
MAPE	36.211025	30.527671	33.403338	30.755716	63.698328	38.517113	29.403460	

- The tuned gradient boosting model is the best model here. It has the highest r2 score of approx 95.5% and the lowest RMSE of approx 39 on the test data.
- Gradient boosting, XGBoost, and stacking regressor are the top 3 models. They are all giving a similar performance.

In [ ]:
```
# So plot observed and predicted values of the test data for the best model i.e. tuned
fig, ax = plt.subplots(figsize=(8, 6))
y_pred=gb_tuned.predict(X_test)
ax.scatter(y_test, y_pred, edgecolors=(0, 0, 1))
ax.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=3)
ax.set_xlabel('Observed')
ax.set_ylabel('Predicted')
ax.set_title("Observed vs Predicted")
plt.grid()
plt.show()
```



- We can see that points are dense on the line where predicted is equal to the observed.
- This implies that most of the predicted values are close to the true values with some exceptions as seen in the plot.

In [ ]:
```

```