Fourth Practice ML

In this practice we will learn how to clean and prepare datasets for Machine Learning (This process is called **Data Cleansing** (https://en.wikipedia.org/wiki/Data_cleansing)).
We will also learn **One-Hot encoding** and dividing the data into parts (**train-validation-test**).

Plotly

Link: https://plotly.com/ (https://plotly.com/)
Plotly's Python graphing library makes interactive, publication-quality graphs.
Plotly supports over 40 unique chart types covering a wide range of statistical, financial, geographic, scientific, and 3-dimensional use-cases.
The packege is free and open source and you can view the source, report issues or contribute on GitHub (https://github.com/plotly).

Downloads, Imports and Definitions

First, we want to check Plotly version on the system.

In [2]:

```
# show plotly version
!pip show plotly
```

```
Name: plotly
Version: 4.4.1
Summary: An open-source, interactive graphing library for Python
Home-page: https://plot.ly/python/
Author: Chris P
Author-email: chris@plot.ly
License: MIT
Location: /usr/local/lib/python3.7/dist-packages
Requires: retrying, six
Required-by: cufflinks
```

This version (4.4.1) is old, we want to update it so we will have the new features of Plotly (like the new `sunburst` graph).

In [3]:

```
# update plotly version
!pip install --upgrade plotly
```

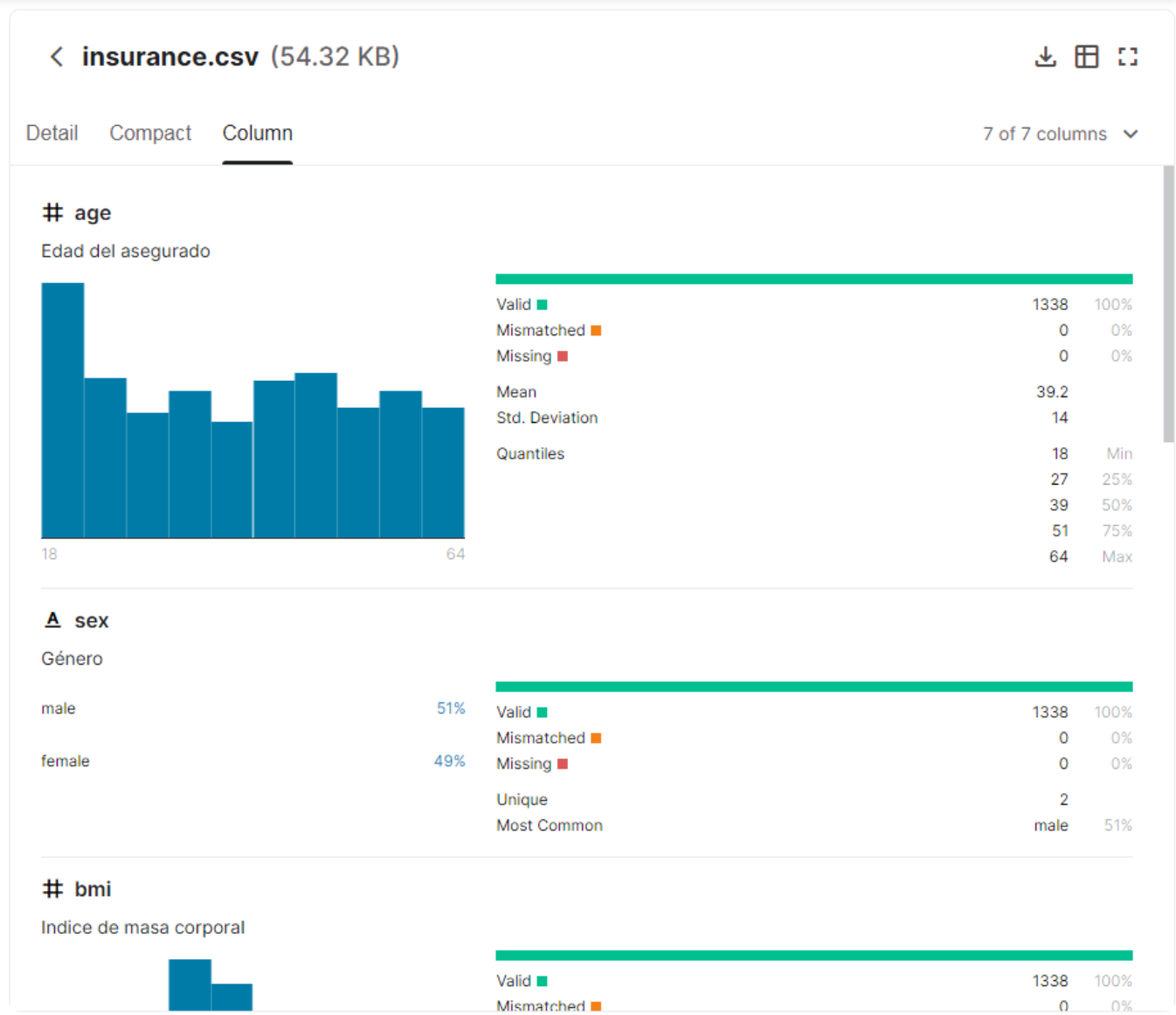
```
Requirement already satisfied: plotly in /usr/local/lib/python3.7/dist-pac
kages (4.4.1)
Collecting plotly
  Downloading plotly-5.4.0-py2.py3-none-any.whl (25.3 MB)
     |████████████████████████████████| 25.3 MB 65 kB/s
Collecting tenacity>=6.2.0
  Downloading tenacity-8.0.1-py3-none-any.whl (24 kB)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packag
es (from plotly) (1.15.0)
Installing collected packages: tenacity, plotly
  Attempting uninstall: plotly
    Found existing installation: plotly 4.4.1
    Uninstalling plotly-4.4.1:
      Successfully uninstalled plotly-4.4.1
Successfully installed plotly-5.4.0 tenacity-8.0.1
```

In [4]:

```
# import numpy, matplotlib, etc.
import numpy as np
import pandas as pd

# sklearn imports
from sklearn import metrics
from sklearn import pipeline
from sklearn import linear_model
from sklearn import preprocessing
from sklearn import model_selection
```

Data Cleansing

We will use the insurance (https://github.com/stedy/Machine-Learning-with-R-datasets/blob/master/insurance.csv) dataset loaded from Github.
with this dataset we want to predict the individual medical cost billed by health insurance.
We can read more about the dataset in Kaggle (https://www.kaggle.com/mirichoi0218/insurance).
We can also see the a summary of the dataseet's columns in the bottom of the Kaggle page:



We could grab the dataset from Kaggle servers, but it is simpler to download it from Github (Kaggle requires an account in it's site).
Let's download the dataset from Github with Linux command `wget` .

In [5]:

```
# download insurance.csv file from Github
!wget https://github.com/stedy/Machine-Learning-with-R-datasets/raw/master/insurance.csv
```

```
--2021-11-19 22:20:03--  https://github.com/stedy/Machine-Learning-with-R-datasets/raw/master/insurance.csv
Resolving github.com (github.com)... 192.30.255.113
Connecting to github.com (github.com)|192.30.255.113|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/master/insurance.csv [following]
--2021-11-19 22:20:03--  https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/master/insurance.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 54288 (53K) [text/plain]
Saving to: ‘insurance.csv’

insurance.csv       100%[===================>]  53.02K  --.-KB/s    in 0.006s

2021-11-19 22:20:03 (8.03 MB/s) - ‘insurance.csv’ saved [54288/54288]
```

In [6]:

```
# load the insurance csv file
insurance_df = pd.read_csv('insurance.csv')
insurance_df
```

Out[6]:

	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
...	...	...	...	...	...	...	...
1333	50	male	30.970	3	no	northwest	10600.54830
1334	18	female	31.920	0	no	northeast	2205.98080
1335	18	female	36.850	0	no	southeast	1629.83350
1336	21	female	25.800	0	no	southwest	2007.94500
1337	61	female	29.070	0	yes	northwest	29141.36030

1338 rows × 7 columns

We have 7 columns; 6 features and 1 label.
The Features:

1. **age**: age of primary beneficiary (int).
2. **sex**: insurance contractor gender (string; female, male).
3. **bmi**: Body mass index, providing an understanding of body, weights that are relatively high or low relative to height, objective index of body weight (kg / m ^ 2) using the ratio of height to weight, ideally 18.5 to 24.9 (float).
4. **children**: Number of children covered by health insurance / Number of dependents (int).
5. **smoker**: Smoking? (string; yes, no).
6. **region**: the beneficiary's residential area in the US (string; northeast, southeast, southwest, northwest).

The Traget:

- **charges**: Individual medical costs billed by health insurance (float).

Some of our features are **categorical** (`sex` , `smoker` and `region`).
Categorical features are features that has no intrinsic order between their values (smoker, non-smoker).

Some of our features are **ordinal** (`children`).
Ordinal features are features similar to categorical features, but there is an order between the values (1 child, 2 children, etc.).
In ordinal features, there is no meaning for the values in between (there is no 1.5 child).

Some of our features are **numerical** (`bmi` , `age` and the target `charges`).
Numerical features are like ordinal features, but there is meaning to the values in between (`bmi` is a scale, there is meaning to each fraction).

Let's start with cleansing the dataset.
The first thing to do, is to check for empty values.
Empty values can be `''` in string columns, or `NaN` values.

In [7]:

```
# detect np.NaN values in the df
np.where(insurance_df.isnull())
```

Out[7]:

```
(array([], dtype=int64), array([], dtype=int64))
```

There is no empty values.
Let's insert one empty line to the data.

In [8]:

```
# add an empty line to the df
insurance_df_cp = insurance_df.copy()
insurance_df_cp.loc[len(insurance_df)] = [np.NaN, "", np.NaN, None, None, "", np.NaN]
insurance_df_cp.loc[len(insurance_df_cp)] = [np.NaN, "", np.NaN, np.NaN, None, None, np
.NaN]
insurance_df_cp
```

Out[8]:

	age	sex	bmi	children	smoker	region	charges
0	19.0	female	27.900	0	yes	southwest	16884.92400
1	18.0	male	33.770	1	no	southeast	1725.55230
2	28.0	male	33.000	3	no	southeast	4449.46200
3	33.0	male	22.705	0	no	northwest	21984.47061
4	32.0	male	28.880	0	no	northwest	3866.85520
...	...	...	...	...	...	...	...
1335	18.0	female	36.850	0	no	southeast	1629.83350
1336	21.0	female	25.800	0	no	southwest	2007.94500
1337	61.0	female	29.070	0	yes	northwest	29141.36030
1338	NaN		NaN	None	None		NaN
1339	NaN		NaN	NaN	None	None	NaN

1340 rows × 7 columns

The data in real life will have empty values.
When we get a new dataset, we need to fill the empty values, or remove the rows/columns that have them.
In this practice we will fill the values (we don't want to lose valuble data).
Let's check the types of the columns.

In [9]:

```
# print the type of the columns
insurance_df_cp.dtypes
```

Out[9]:

```
age         float64
sex          object
bmi         float64
children     object
smoker       object
region       object
charges     float64
dtype: object
```

When a column is of type `float64`, we know that it is a floating point number.
So, in this column, the only empty value possible is `np.NaN`.
When a column is of type `object`, we know that it is a string or a floating point number (with None values).
So, in this column, the empty values possible are `np.NaN`, `""` and `None`.
The type hierarchy is: **`int64 < float64 < object`**.
When there is at least one `float64` element in the columm, the column type will be `float64`.
When there is at least one `object` element in the columm, the column type will be `object`.
Let's translate all the empty values to `np.NaN` values (Pandas works best with these values).

In [10]:

```
# replace all empty values to np.NaN values
insurance_df_cp.replace('', np.NaN, inplace=True)
insurance_df_cp.fillna(np.NaN, inplace=True)
insurance_df_cp
```

Out[10]:

	age	sex	bmi	children	smoker	region	charges
0	19.0	female	27.900	0.0	yes	southwest	16884.92400
1	18.0	male	33.770	1.0	no	southeast	1725.55230
2	28.0	male	33.000	3.0	no	southeast	4449.46200
3	33.0	male	22.705	0.0	no	northwest	21984.47061
4	32.0	male	28.880	0.0	no	northwest	3866.85520
...	...	...	...	...	...	...	...
1335	18.0	female	36.850	0.0	no	southeast	1629.83350
1336	21.0	female	25.800	0.0	no	southwest	2007.94500
1337	61.0	female	29.070	0.0	yes	northwest	29141.36030
1338	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1339	NaN	NaN	NaN	NaN	NaN	NaN	NaN

1340 rows × 7 columns

Let's see the empty values (rows, cols).

In [11]:

```
# detect np.NaN or None values in the copy of df
print(f'There are {len(np.where(insurance_df_cp.isnull())[0])} empty values in the data frame:')
print(np.where(insurance_df_cp.isnull()))
```

```
There are 14 empty values in the dataframe:
(array([1338, 1338, 1338, 1338, 1338, 1338, 1338, 1339, 1339, 1339, 1339,
       1339, 1339, 1339]), array([0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5,
6]))
```

We can count how many empty values we have in each column.

In [12]:

```
# count empty values in each column
def count_empty_values_in_each_column(df: pd.DataFrame):
  print('empty values')
  print('------------\n')

  for col in df.columns:
    print(f"{col}: {df[col].isna().sum()}")

count_empty_values_in_each_column(insurance_df_cp)
```

```
empty values
------------

age: 2
sex: 2
bmi: 2
children: 2
smoker: 2
region: 2
charges: 2
```

count empty values in each column using the `eval` function of python

Warning: Though `eval` is a very useful function for performing commands from strings, it also might make your programs open to exploits in terms of security, so keep that in mind!

In [13]:

```
def count_empty_values_in_each_column(df: pd.DataFrame):
  print('empty values')
  print('------------\n')
  command = "df[col].isna().sum()"

  for col in df.columns:
    print(f"{col}: {eval(command)}")

count_empty_values_in_each_column(insurance_df_cp)
```

```
empty values
------------

age: 2
sex: 2
bmi: 2
children: 2
smoker: 2
region: 2
charges: 2
```

In [14]:

```
# count empty values in each column
def count_empty_values_in_each_column(df):
    print('empty values:')
    code = "len(np.where(df[column].isnull())[0])"
    for column in df.columns:
        print(f'`{column}`: {eval(code)}')

count_empty_values_in_each_column(insurance_df_cp)
```

```
empty values:
`age`: 2
`sex`: 2
`bmi`: 2
`children`: 2
`smoker`: 2
`region`: 2
`charges`: 2
```

There isn't a correct way of completing these values.
There are few options for this:

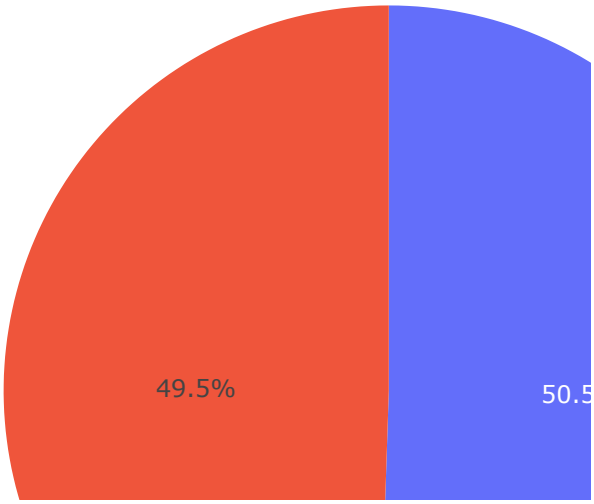
1. Enter a **constant** value.
   For continuous values, the constant value can be calculated from the rest of the values in the column (min, max, mean, median, etc.) or derived from expert knowledge.
   For categorical values, the constant value can be one of the values in the column or a different value not present in the column.
2. Enter **random** values.
   Continuous values can be randomly picked from the values of the column, or be randomly generated from the range of the optional values in the column.
   Categorical variables can be randomly picked from the values in the column. We can use normal distribution or column distribution.
3. Enter **prediction** of the values.
   For continuous values, we can use regression methods to predict the missing values.
   For categorical values, we can use classification mathods to predict the missing values.

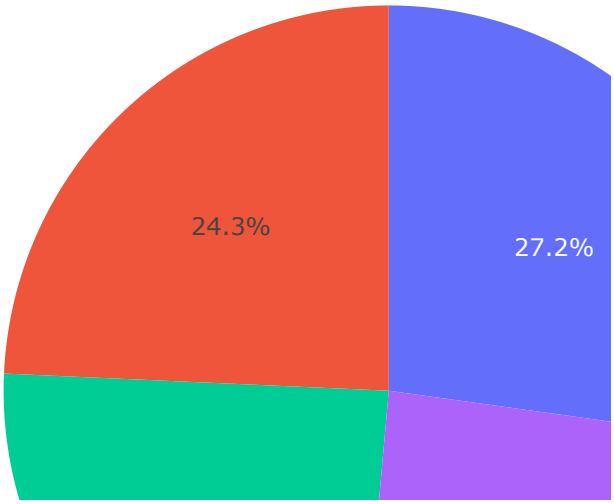
Ordinal features are something between categorical and numerical features, so each one of the methods can work for them.

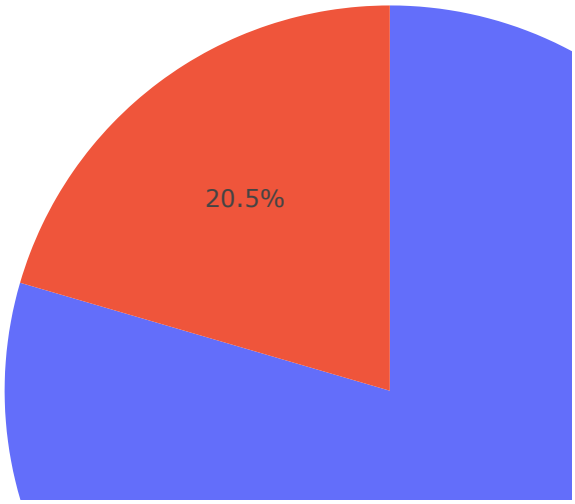
To complete the empty values in each column, we need to get some data on the column.
Let's start with the categorical columns.
We can show the distribution of each column with pie charts (https://plotly.com/python/pie-charts/).

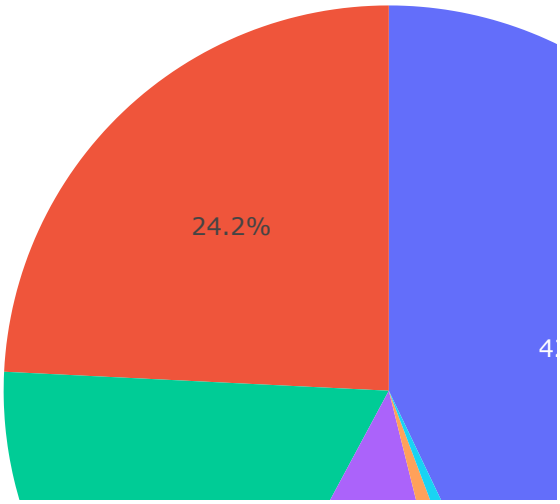
In [15]:

```
# import px and create pie charts for each categorical feature
import plotly.express as px
def create_pie_chart_of_count(df, column_name):
    df_not_null = df[~df[column_name].isnull()]
    fig = px.pie(df_not_null.groupby([column_name]).size().reset_index(name='count'), n
ames=column_name, values='count')
    fig.show()

create_pie_chart_of_count(insurance_df, 'sex')
create_pie_chart_of_count(insurance_df, 'region')
create_pie_chart_of_count(insurance_df, 'smoker')
create_pie_chart_of_count(insurance_df, 'children')
```


49.5%
50.5


24.3%
27.2%

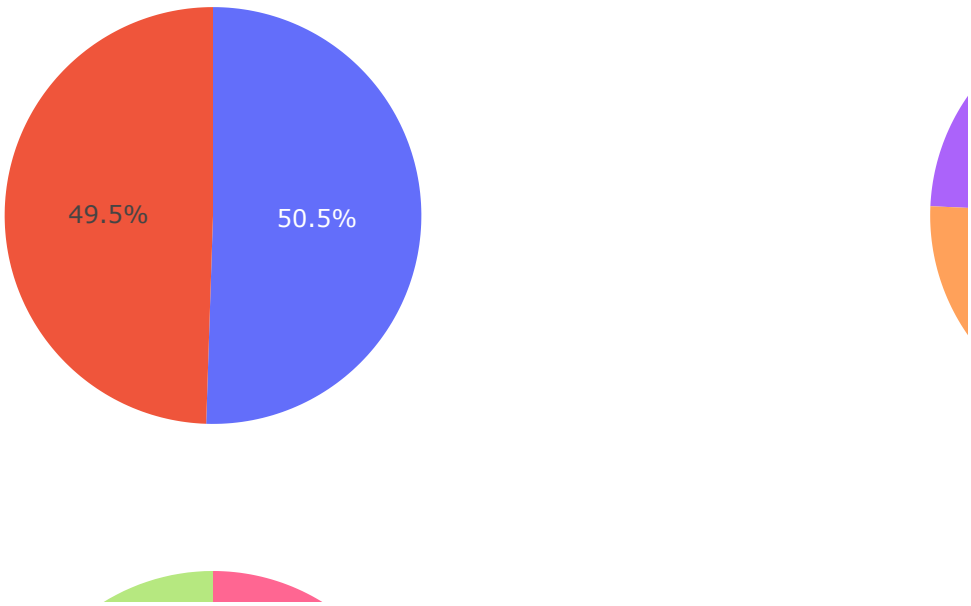

20.5%



We can show all the plots as subplots (https://plotly.com/python-api-reference/generated/plotly.subplots.make_subplots.html) with graph objects (https://plotly.com/python/creating-and-updating-figures/#figures-as-graph-objects) and pie charts (https://plotly.github.io/plotly.py-docs/generated/plotly.graph_objects.Pie.html).

In [16]:

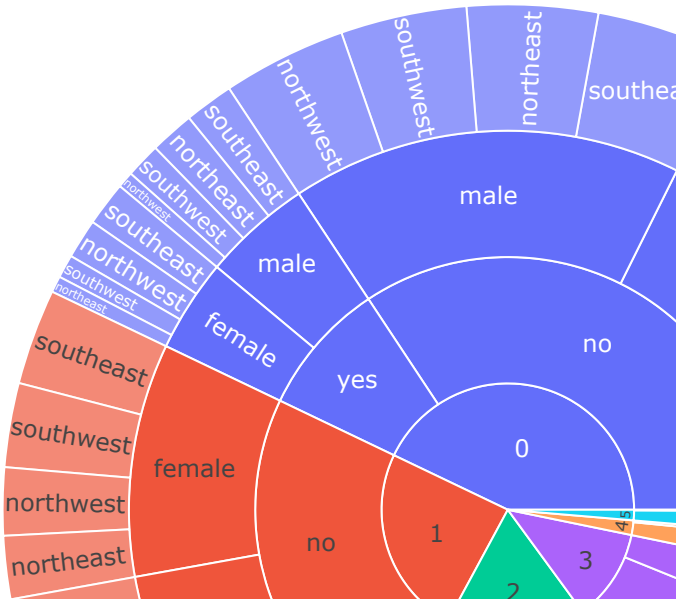
```
# import go and make_subplots and create pie charts subplots of the categorical features
import plotly.graph_objects as go
from plotly.subplots import make_subplots
def create_pie_chart_subplot_of_count(df, columns_names):
    rows = int(np.ceil(np.sqrt(len(columns_names))))
    cols = int(np.ceil(len(columns_names)/rows))
    fig = make_subplots(rows=rows, cols=cols, specs=[[{"type": "domain"} for i in range(cols)] for j in range(rows)])
    for i, column_name in enumerate(columns_names):
        df_not_null = df[~df[column_name].isnull()]
        fig.add_trace(go.Pie(labels=df_not_null.groupby([column_name]).size().reset_index(name='count')[column_name],
                             values=df_not_null.groupby([column_name]).size().reset_index(name='count')['count'],
                             name=column_name),
                      (i)//cols+1, (i)%cols+1)
    fig.update_layout(margin=dict(t=10, l=10, r=10, b=10))
    fig.show()

create_pie_chart_subplot_of_count(insurance_df, ['sex', 'region', 'smoker', 'children'])
```

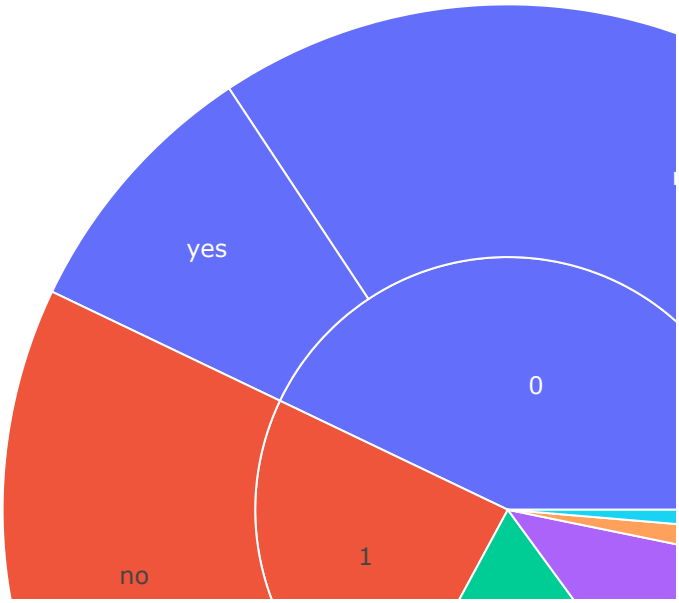


We can show the inner distribution with sunburst charts (https://plotly.com/python/sunburst-charts/).
We can show it even for the non-categorical features, we can limit the depth of the chart and put the non-categorical features at the end of the chain.

In [17]:

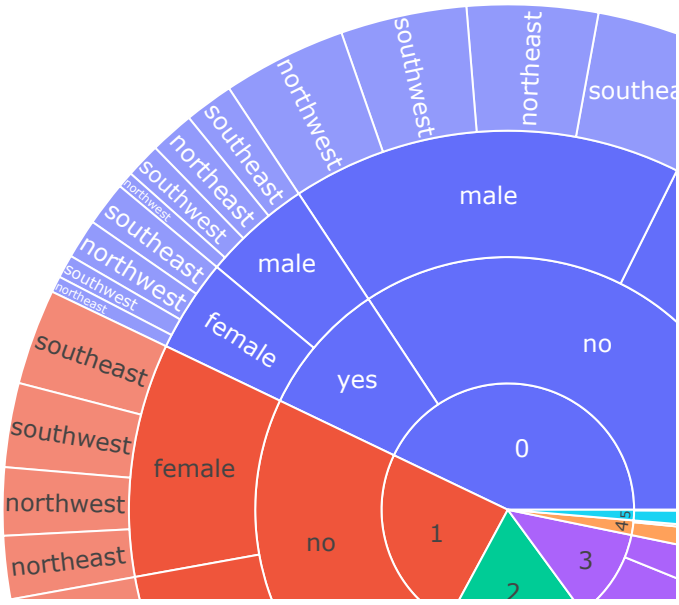
```
# create sunburst charts of the features
insurance_df_cp2 = insurance_df.copy()
fig = px.sunburst(insurance_df_cp2 , path=['children', 'smoker', 'sex', 'region'])
fig.update_layout(margin=dict(t=10, l=10, r=10, b=10))
fig.show()
fig = px.sunburst(insurance_df_cp2 , path=['children', 'smoker', 'sex', 'region', 'age'
, 'bmi'], maxdepth=2)
fig.update_layout(margin=dict(t=10, l=10, r=10, b=10))
fig.show()
```

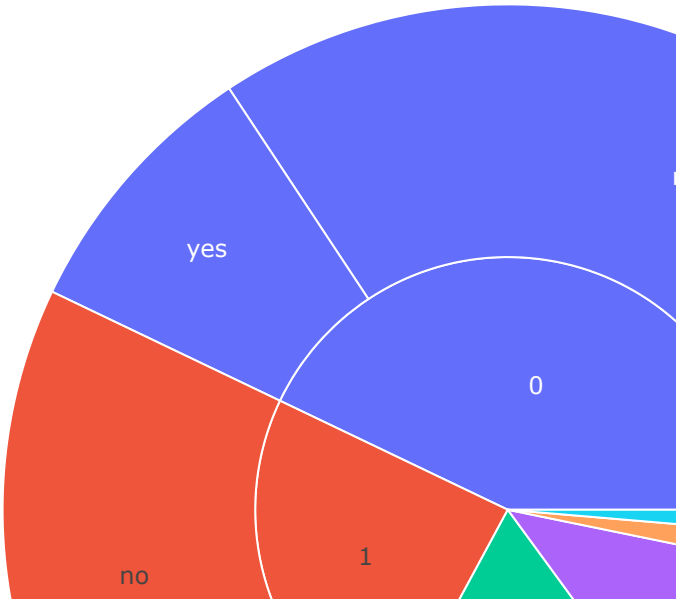

southeast
southwest
northwest
northeast
female
male
yes
no
0
1
2
3
4
5


yes
0
no
1

In [18]:

```
# create sunburst charts of the features
insurance_df_cp2 = insurance_df.copy()
insurance_df_cp2.insert(len(insurance_df_cp2.columns), "count", 1, True)
fig = px.sunburst(insurance_df_cp2 , path=['children', 'smoker', 'sex', 'region'], valu
es='count')
fig.update_layout(margin=dict(t=10, l=10, r=10, b=10))
fig.show()
fig = px.sunburst(insurance_df_cp2 , path=['children', 'smoker', 'sex', 'region', 'age'
, 'bmi'], values='count', maxdepth=2)
fig.update_layout(margin=dict(t=10, l=10, r=10, b=10))
fig.show()
```


southeast
southwest
northwest
northeast
female
male
yes
no
0
1
2
3
4
5



In general, we can randomly pick one of the values for categorical features, and pick the mean or median for numerical features (but it won't always be the best way).

In [19]:

```
# fill empty values in the dataframe
def fill_na_median(df, column_name):
    df_not_null = df[~df[column_name].isnull()]
    df[column_name].fillna(df_not_null[column_name].median(), inplace=True)

def fill_na_mean(df, column_name):
    df_not_null = df[~df[column_name].isnull()]
    df[column_name].fillna(df_not_null[column_name].mean(), inplace=True)

def fill_na_random_pick_column_distribution(df, column_name):
    df_not_null = df[~df[column_name].isnull()]
    df_null = df[df[column_name].isnull()]
    options = np.random.choice(df_not_null[column_name])
    df[column_name] = df[column_name].apply(lambda x: np.random.choice(df_not_null[colu
mn_name]) if pd.isnull(x) else x)

fill_na_median(insurance_df_cp, 'age')
fill_na_mean(insurance_df_cp, 'bmi')
fill_na_mean(insurance_df_cp, 'charges')
fill_na_random_pick_column_distribution(insurance_df_cp, 'region')
fill_na_random_pick_column_distribution(insurance_df_cp, 'children')
fill_na_random_pick_column_distribution(insurance_df_cp, 'smoker')
fill_na_random_pick_column_distribution(insurance_df_cp, 'sex')
insurance_df_cp
```

Out[19]:

	age	sex	bmi	children	smoker	region	charges
0	19.0	female	27.900000	0.0	yes	southwest	16884.924000
1	18.0	male	33.770000	1.0	no	southeast	1725.552300
2	28.0	male	33.000000	3.0	no	southeast	4449.462000
3	33.0	male	22.705000	0.0	no	northwest	21984.470610
4	32.0	male	28.880000	0.0	no	northwest	3866.855200
...	...	...	...	...	...	...	...
1335	18.0	female	36.850000	0.0	no	southeast	1629.833500
1336	21.0	female	25.800000	0.0	no	southwest	2007.945000
1337	61.0	female	29.070000	0.0	yes	northwest	29141.360300
1338	39.0	male	30.663397	0.0	no	southwest	13270.422265
1339	39.0	female	30.663397	3.0	no	southwest	13270.422265

1340 rows × 7 columns

In [20]:

```
# check for empty values
count_empty_values_in_each_column(insurance_df_cp)
```

```
empty values:
`age`: 0
`sex`: 0
`bmi`: 0
`children`: 0
`smoker`: 0
`region`: 0
`charges`: 0
```

We can see that there are no more empty values.

Some machine learning algorithms (like logistic and linear regression) can not work with categorical features and may only work with numerical or ordinal featurs.
The next step in preparing the dataset for model learning is converting the categorical features into numerical features.
There are few ways of doing that:

1. **Label Encoding:**
   Tranform the categorical data into ordinal data. Translate each category to an integer number.
   This should be done when there is an order to the values or when there are too many values to handle.
   Example:

old_region_column	new_region_column
southwest	0
northwest	1
southeast	2
northeast	3

1. **One-Hot Encoding:**
   Transform the the categorical data into few binary columns. Translate each category into a column with 0 and 1 values (1 if the original categorical value is present in the row, and 0 if not).
   This should be done when there is no order to the values and where there aren't as many different values in the column.
   This should be used with regularized regressions (may suffer from bias issues without the added column).
   Example:

old_region_column	new_southwest_column	new_northwest_column	new_southeast_column	new_northeast_co
southwest	1	0	0	0
northwest	0	1	0	0
southeast	0	0	1	0
northeast	0	0	0	1

1. **Dummy Encoding:**
   Same as one-hot encoding, but without one of the columns (the category that was represented by that missing column will be represented by 0 in all the other columns).
   This should be used with unregularized regressions or with neural networks (may suffer from variance issues with the added column).

old_region_column	new_southwest_column	new_northwest_column	new_southeast_column
southwest	1	0	0
northwest	0	1	0
southeast	0	0	1
northeast	0	0	0

We will use Scikit-learn OneHotEncoder (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html).
We will use it as Dummy Encoder.

In [21]:

```
# dummy encode the categorical variables in the df
from sklearn.preprocessing import OneHotEncoder
insurance_df_cat = insurance_df_cp[['sex', 'smoker', 'region']]
enc = OneHotEncoder(drop='first', sparse=False)
insurance_df_cat_enc = pd.DataFrame(enc.fit_transform(insurance_df_cat))
insurance_df_cp_enc = insurance_df_cp.drop(['sex', 'smoker', 'region'], axis=1).join(in
surance_df_cat_enc)
insurance_df_cp_enc
```

Out[21]:

	age	bmi	children	charges	0	1	2	3	4
0	19.0	27.900000	0.0	16884.924000	0.0	1.0	0.0	0.0	1.0
1	18.0	33.770000	1.0	1725.552300	1.0	0.0	0.0	1.0	0.0
2	28.0	33.000000	3.0	4449.462000	1.0	0.0	0.0	1.0	0.0
3	33.0	22.705000	0.0	21984.470610	1.0	0.0	1.0	0.0	0.0
4	32.0	28.880000	0.0	3866.855200	1.0	0.0	1.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...
1335	18.0	36.850000	0.0	1629.833500	0.0	0.0	0.0	1.0	0.0
1336	21.0	25.800000	0.0	2007.945000	0.0	0.0	0.0	0.0	1.0
1337	61.0	29.070000	0.0	29141.360300	0.0	1.0	1.0	0.0	0.0
1338	39.0	30.663397	0.0	13270.422265	1.0	0.0	0.0	0.0	1.0
1339	39.0	30.663397	3.0	13270.422265	0.0	0.0	0.0	0.0	1.0

1340 rows × 9 columns

We can also use the `get_feature_names` function of the encoder in order to get the names of the columns, and by that have a better understanding of the encoded dataset.

In [22]:

```
from sklearn.preprocessing import OneHotEncoder
insurance_df_cat = insurance_df_cp[['sex', 'smoker', 'region']]
enc = OneHotEncoder(sparse=False, drop='first')
insurance_df_cat_enc = pd.DataFrame(enc.fit_transform(insurance_df_cat), columns=enc.ge
t_feature_names_out())
insurance_df_cp_enc = insurance_df_cp.drop(['sex', 'smoker', 'region'], axis=1).join(in
surance_df_cat_enc)
# insurance_df_cp_enc = insurance_df_cp.join(insurance_df_cat_enc)
insurance_df_cp_enc
```

Out[22]:

	age	bmi	children	charges	sex_male	smoker_yes	region_northwest	regi
0	19.0	27.900000	0.0	16884.924000	0.0	1.0	0.0	
1	18.0	33.770000	1.0	1725.552300	1.0	0.0	0.0	
2	28.0	33.000000	3.0	4449.462000	1.0	0.0	0.0	
3	33.0	22.705000	0.0	21984.470610	1.0	0.0	1.0	
4	32.0	28.880000	0.0	3866.855200	1.0	0.0	1.0	
...	...	...	...	...	...	...	...	
1335	18.0	36.850000	0.0	1629.833500	0.0	0.0	0.0	
1336	21.0	25.800000	0.0	2007.945000	0.0	0.0	0.0	
1337	61.0	29.070000	0.0	29141.360300	0.0	1.0	1.0	
1338	39.0	30.663397	0.0	13270.422265	1.0	0.0	0.0	
1339	39.0	30.663397	3.0	13270.422265	0.0	0.0	0.0	

1340 rows × 9 columns

We can see that `sex` column has been converted to 1 binary column, the `smoker` column has been converted to 1 binary column, and the `region` column has been converted to 3 binary columns.
We can create a method to do this task.

In [ ]:

```
# dummy encode the categorical variables in the df with method
def dummy_encode(df, columns_names):
    df_cat = df[columns_names]
    enc = OneHotEncoder(drop='first', sparse=False)
    df_cat_enc = pd.DataFrame(enc.fit_transform(df_cat))
    df_enc = df.drop(columns_names, axis=1).join(df_cat_enc)
    return df_enc

insurance_df_cp_enc2 = dummy_encode(insurance_df_cp, ['sex', 'smoker', 'region'])
insurance_df_cp_enc2
```

Out[ ]:

	age	bmi	children	charges	0	1	2	3	4
0	19.0	27.900000	0.0	16884.924000	0.0	1.0	0.0	0.0	1.0
1	18.0	33.770000	1.0	1725.552300	1.0	0.0	0.0	1.0	0.0
2	28.0	33.000000	3.0	4449.462000	1.0	0.0	0.0	1.0	0.0
3	33.0	22.705000	0.0	21984.470610	1.0	0.0	1.0	0.0	0.0
4	32.0	28.880000	0.0	3866.855200	1.0	0.0	1.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...
1335	18.0	36.850000	0.0	1629.833500	0.0	0.0	0.0	1.0	0.0
1336	21.0	25.800000	0.0	2007.945000	0.0	0.0	0.0	0.0	1.0
1337	61.0	29.070000	0.0	29141.360300	0.0	1.0	1.0	0.0	0.0
1338	39.0	30.663397	0.0	13270.422265	0.0	0.0	1.0	0.0	0.0
1339	39.0	30.663397	3.0	13270.422265	0.0	0.0	0.0	0.0	1.0

1340 rows × 9 columns

We can also use Pandas get_dummies (https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html) method in one line, and even attach names to the new columns.

In [ ]:

```
# dummy encode the categorical variables in the df with get_dummies
insurance_df_dum = pd.get_dummies(insurance_df_cp, columns=['sex', 'smoker', 'region'],
prefix=["sex_type_is", "smoker_type_is", "region_type_is"], drop_first=True)
insurance_df_dum
```

Out[ ]:

	age	bmi	children	charges	sex_type_is_male	smoker_type_is_yes	region_
0	19.0	27.900000	0.0	16884.924000	0	1	
1	18.0	33.770000	1.0	1725.552300	1	0	
2	28.0	33.000000	3.0	4449.462000	1	0	
3	33.0	22.705000	0.0	21984.470610	1	0	
4	32.0	28.880000	0.0	3866.855200	1	0	
...	...	...	...	...	...	...	
1335	18.0	36.850000	0.0	1629.833500	0	0	
1336	21.0	25.800000	0.0	2007.945000	0	0	
1337	61.0	29.070000	0.0	29141.360300	0	1	
1338	39.0	30.663397	0.0	13270.422265	0	0	
1339	39.0	30.663397	3.0	13270.422265	0	0	

1340 rows × 9 columns

The difference between the `get_dummies` approach and the `OneHotEncoder` approach is that `OneHotEncoder` can transform few datasets with the same encoding (if we have for example, train and test), while `get_dummies` only converts one dataframe at a time (the result may have different encodings for the same column in different datasets).

Data Slicing

In real life scenerios, we don't have the test data.
We can not check the performence of the model on the same dataset that the model was trained on.
This will result in wrong estimation for the model generalization capabilities.
In order to check our prediction and fine-tune the model parameters, we need to slice the dataset into 2 groups:

1. train
2. validation

We will train on the train data and check the performance on the validation data.
We will slice the dataset with Scikit-learn train_test_split (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html).
First, let's split the data to features `X` and target `t`.

In [ ]:

```
# divide the data to features and target
t = insurance_df_cp_enc['charges'].copy()
X = insurance_df_cp_enc.drop(['charges'], axis=1)
print('t')
display(t)
print()
print('X')
display(X)
```

```
t

0       16884.924000
1        1725.552300
2        4449.462000
3       21984.470610
4        3866.855200
            ...
1335     1629.833500
1336     2007.945000
1337    29141.360300
1338    13270.422265
1339    13270.422265
Name: charges, Length: 1340, dtype: float64

X
```

	age	bmi	children	x0_male	x1_yes	x2_northwest	x2_southeast	x2_southwest
0	19.0	27.900000	0.0	0.0	1.0	0.0	0.0	1.0
1	18.0	33.770000	1.0	1.0	0.0	0.0	1.0	0.0
2	28.0	33.000000	3.0	1.0	0.0	0.0	1.0	0.0
3	33.0	22.705000	0.0	1.0	0.0	1.0	0.0	0.0
4	32.0	28.880000	0.0	1.0	0.0	1.0	0.0	0.0
...	...	...	...	...	...	...	...	...
1335	18.0	36.850000	0.0	0.0	0.0	0.0	1.0	0.0
1336	21.0	25.800000	0.0	0.0	0.0	0.0	0.0	1.0
1337	61.0	29.070000	0.0	0.0	1.0	1.0	0.0	0.0
1338	39.0	30.663397	0.0	0.0	0.0	1.0	0.0	0.0
1339	39.0	30.663397	3.0	0.0	0.0	0.0	0.0	1.0

1340 rows × 8 columns

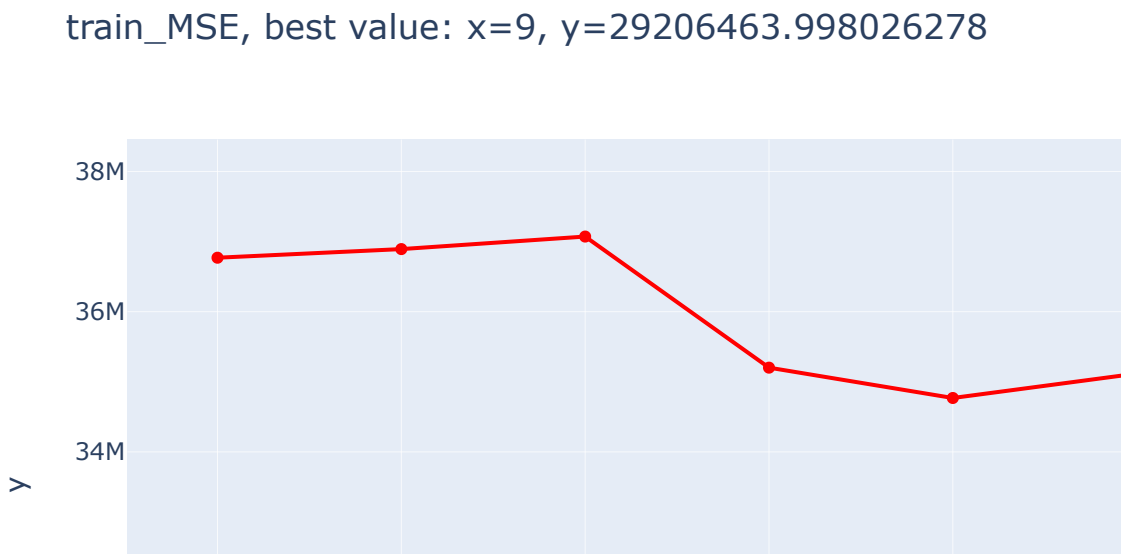
Now, we can split the data to train and validation.
We can choose number of values for the `test_size` argument.
Let's check few of them with NE and MSE.
We can plot the data with Plotly [scatter (https://plotly.com/python/line-and-scatter/)](https://plotly.com/python/line-and-scatter/).

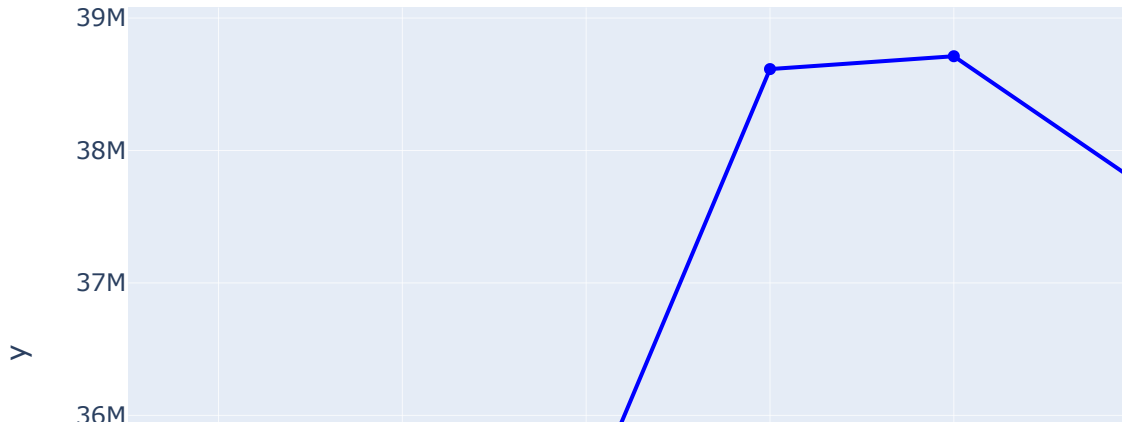
In [ ]:

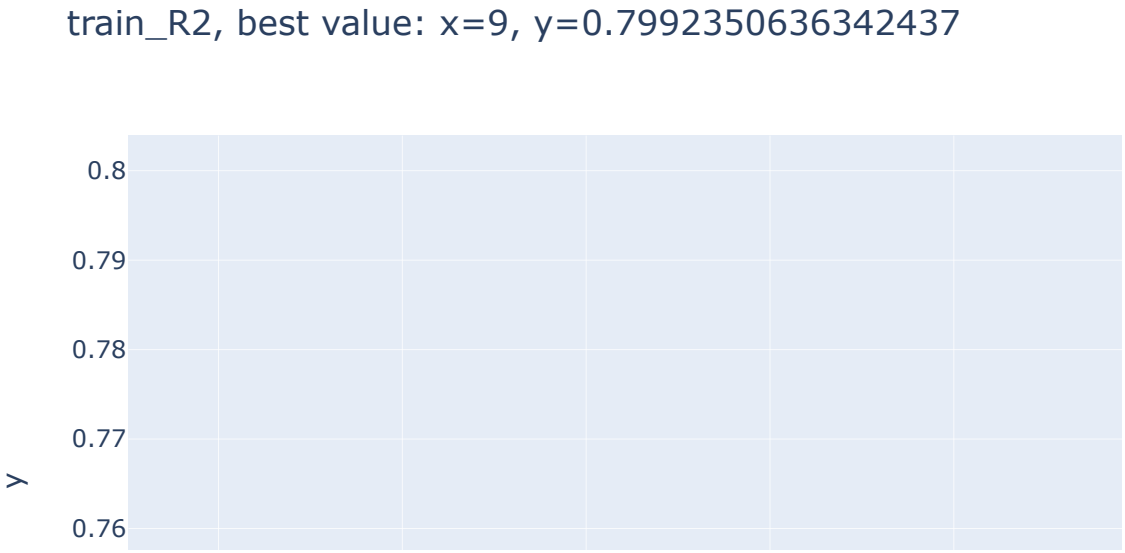
```
# print 4 graphs: mse of train/test and r2 of train/test
def print_graphs_r2_mse(graph_points):
    for k, v in graph_points.items():
        best_value = max(v.values()) if 'R2' in k else min(v.values())
        best_index = np.argmax(list(v.values())) if 'R2' in k else np.argmin(list(v.val
ues()))
        color = 'red' if 'train' in k else 'blue'
        fig = px.scatter(x=v.keys(), y=v.values(), title=f'{k}, best value: x={best_ind
ex + 1}, y={best_value}', color_discrete_sequence=[color])
        fig.data[0].update(mode='markers+lines')
        fig.show()
```

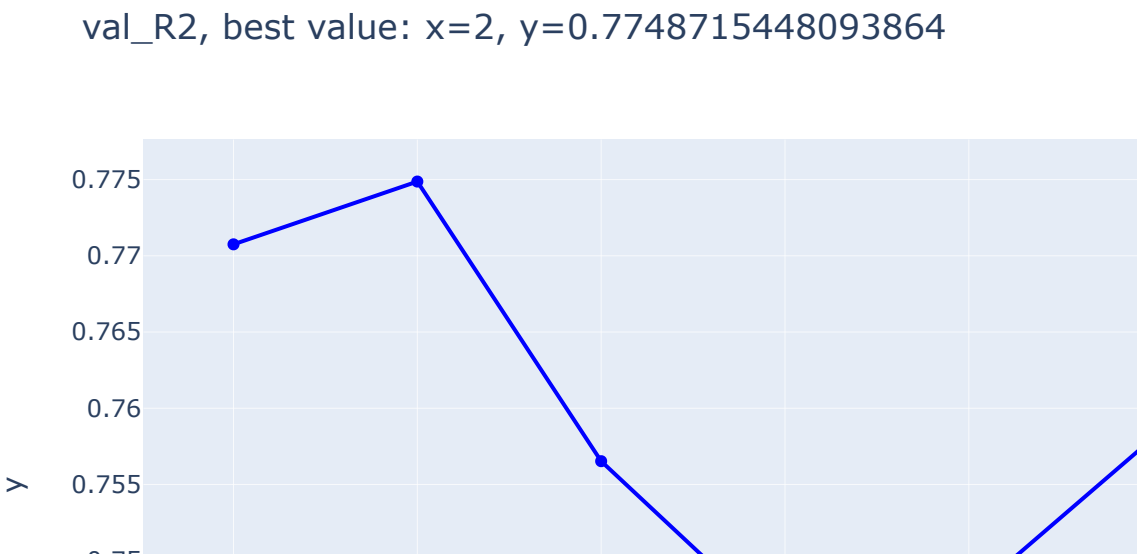
In [ ]:

```
from plotly import express as px

# plot the score by split and the loss by split
def plot_score_and_loss_by_split(X, t):
    graph_points = {
                    'train_MSE':{},
                    'val_MSE': {},
                    'train_R2': {},
                    'val_R2': {}
                    }
    for size in range(10, 100, 10):
        X_train, X_val, t_train, t_val = model_selection.train_test_split(X, t, test_si
ze=size/100, random_state=42)
        NE_reg = linear_model.LinearRegression().fit(X_train, t_train)
        y_train = NE_reg.predict(X_train)
        y_val = NE_reg.predict(X_val)
        graph_points['train_MSE'][size/100] = metrics.mean_squared_error(t_train, y_tra
in)
        graph_points['val_MSE'][size/100] = metrics.mean_squared_error(t_val, y_val)
        graph_points['train_R2'][size/100] = NE_reg.score(X_train, t_train)
        graph_points['val_R2'][size/100] = NE_reg.score(X_val, t_val)
    print_graphs_r2_mse(graph_points)

plot_score_and_loss_by_split(X, t)
```


train_MSE, best value: x=9, y=29206463.998026278
38M
36M
34M
y


val_MSE, best value: x=1, y=34183267.79009147
39M
38M
37M
36M
y


train_R2, best value: x=9, y=0.7992350636342437
y
0.8
0.79
0.78
0.77
0.76



In [ ]:

```
def fit_and_eval_ne(X_train, X_val, t_train, t_val):
  NE_reg = linear_model.LinearRegression().fit(X_train, t_train)
  y_train = NE_reg.predict(X_train)
  y_val = NE_reg.predict(X_val)
  print("MSE Train:", metrics.mean_squared_error(t_train, y_train))
  print("MSE Valid:", metrics.mean_squared_error(t_val, y_val))
  print("R^2 Train:", NE_reg.score(X_train, t_train))
  print("R^2 Valid:", NE_reg.score(X_val, t_val))
```

We can see that when the validation data size is small, its loss is small.
One explanation to this is that for a small group of samples, it is easier to match a linear hypothesis.
We can see that from 0.1 to 0.3, the validation loss is smaller than the train loss, and from 0.4 to 0.9 the validation loss is smaller than the test loss.
So, let's give the validation group 35% of the dataset, it is about the right point where the validation loss is equal to the train loss.

In [ ]:

```
# split the data to train and validation
X_train, X_val, t_train, t_val = model_selection.train_test_split(X, t, test_size=0.35,
random_state=1)
print('X_train')
display(X_train)
print()
print('t_train')
display(t_train)
print()
print('X_val')
display(X_val)
print()
print('t_val')
display(t_val)
```

```
X_train
```

	age	bmi	children	x0_male	x1_yes	x2_northwest	x2_southeast	x2_southwest
275	47.0	26.600	2.0	0.0	0.0	0.0	0.0	0.0
719	58.0	33.440	0.0	0.0	0.0	1.0	0.0	0.0
345	34.0	29.260	3.0	0.0	0.0	0.0	1.0	0.0
979	36.0	29.920	0.0	0.0	0.0	0.0	1.0	0.0
966	51.0	24.795	2.0	1.0	1.0	1.0	0.0	0.0
...	...	...	...	...	...	...	...	...
715	60.0	28.900	0.0	1.0	0.0	0.0	0.0	1.0
905	26.0	29.355	2.0	0.0	0.0	0.0	0.0	0.0
1096	51.0	34.960	2.0	0.0	1.0	0.0	0.0	0.0
235	40.0	22.220	2.0	0.0	1.0	0.0	1.0	0.0
1061	57.0	27.940	1.0	1.0	0.0	0.0	1.0	0.0

871 rows × 8 columns

```
t_train
```

```
275      9715.84100
719     12231.61360
345      6184.29940
979      4889.03680
966     23967.38305
           ...
715     12146.97100
905      4564.19145
1096    44641.19740
235     19444.26580
1061    11554.22360
Name: charges, Length: 871, dtype: float64
```

```
X_val
```

	age	bmi	children	x0_male	x1_yes	x2_northwest	x2_southeast	x2_southwest
559	19.0	35.530	0.0	1.0	0.0	1.0	0.0	0.0
1089	56.0	22.100	0.0	1.0	0.0	0.0	0.0	1.0
1021	22.0	31.020	3.0	0.0	1.0	0.0	1.0	0.0
460	49.0	36.630	3.0	0.0	0.0	0.0	1.0	0.0
802	21.0	22.300	1.0	1.0	0.0	0.0	0.0	1.0
...	...	...	...	...	...	...	...	...
860	37.0	47.600	2.0	0.0	1.0	0.0	0.0	1.0
207	35.0	27.740	2.0	1.0	1.0	0.0	0.0	0.0
237	31.0	38.390	2.0	1.0	0.0	0.0	1.0	0.0
1032	30.0	27.930	0.0	0.0	0.0	0.0	0.0	0.0
1071	63.0	31.445	0.0	1.0	0.0	0.0	0.0	0.0

469 rows × 8 columns

```
t_val
```

```
559      1646.42970
1089    10577.08700
1021    35595.58980
460     10381.47870
802      2103.08000
           ...
860     46113.51100
207     20984.09360
237      4463.20510
1032     4137.52270
1071    13974.45555
Name: charges, Length: 469, dtype: float64
```

We can see that the data has been splitted randomly to train and validation (X and y are splitted on the same values).

Bias and Variance

Let's try to train the NE model on the data and print the MSE and R2 graphs of the train and test.
Let's use Scikit-learn PolynomialFeatures (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html#sklearn.preprocessing.Po to raise the degree of the model.
This function is adding more featurs, polynomial features of the data.
Example:
If we have the features `[a, b]` and we want to raise it to the 2nd degree, we get `[1, a, b, a^2, ab, b^2]`.
We can choose not to include the intercept with the `include_bias` option, and get `[a, b, a^2, ab, b^2]`.
A linear model that will train on these features is like a polinomial model that will train on the original features.
Let's see how it is done on 2 features, `age` and `bmi`.

In [ ]:

```
# add 2nd degree features to `age` and `bmi` features
print('original features')
display(X_train[['age', 'bmi']])
pol =  preprocessing.PolynomialFeatures(2, include_bias=False)
print()
print('altered features')
pd.DataFrame(pol.fit_transform(X_train[['age', 'bmi']]), columns=['age', 'bmi', 'age^2'
, 'age*bmi', 'bmi^2'])
```

`original features`

	age	bmi
275	47.0	26.600
719	58.0	33.440
345	34.0	29.260
979	36.0	29.920
966	51.0	24.795
...	...	...
715	60.0	28.900
905	26.0	29.355
1096	51.0	34.960
235	40.0	22.220
1061	57.0	27.940

871 rows × 2 columns

`altered features`

Out[ ]:

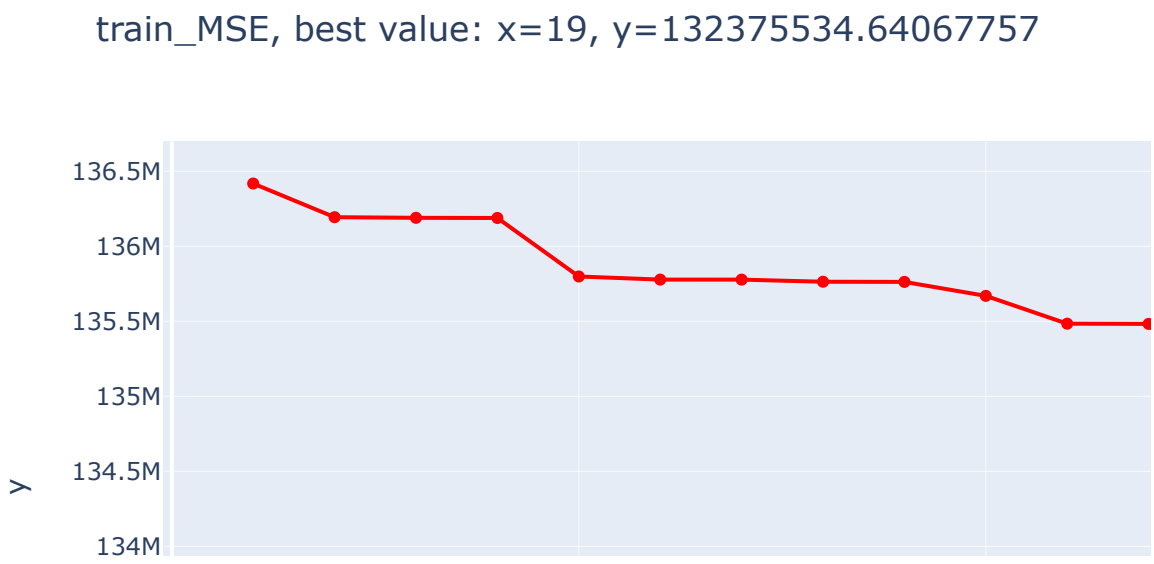
	age	bmi	age^2	age*bmi	bmi^2
0	47.0	26.600	2209.0	1250.200	707.560000
1	58.0	33.440	3364.0	1939.520	1118.233600
2	34.0	29.260	1156.0	994.840	856.147600
3	36.0	29.920	1296.0	1077.120	895.206400
4	51.0	24.795	2601.0	1264.545	614.792025
...	...	...	...	...	...
866	60.0	28.900	3600.0	1734.000	835.210000
867	26.0	29.355	676.0	763.230	861.716025
868	51.0	34.960	2601.0	1782.960	1222.201600
869	40.0	22.220	1600.0	888.800	493.728400
870	57.0	27.940	3249.0	1592.580	780.643600

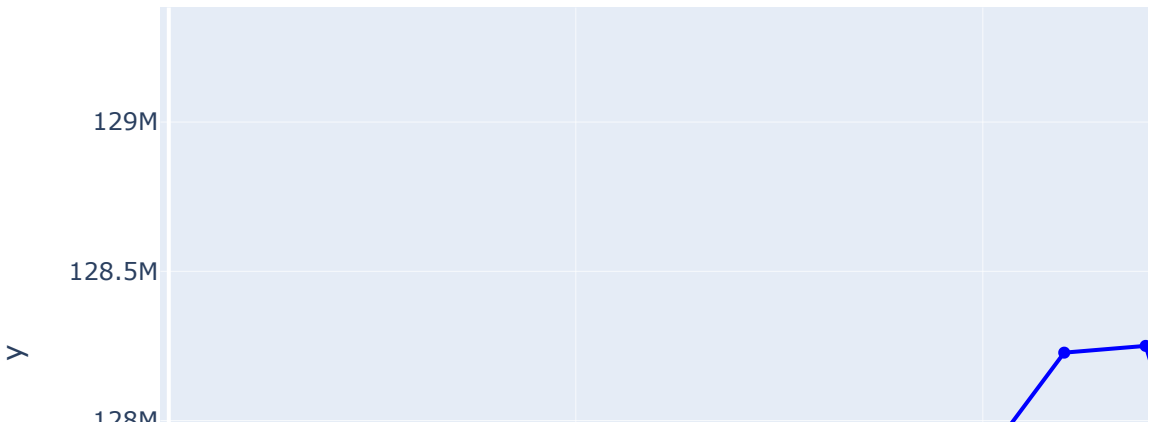
871 rows × 5 columns

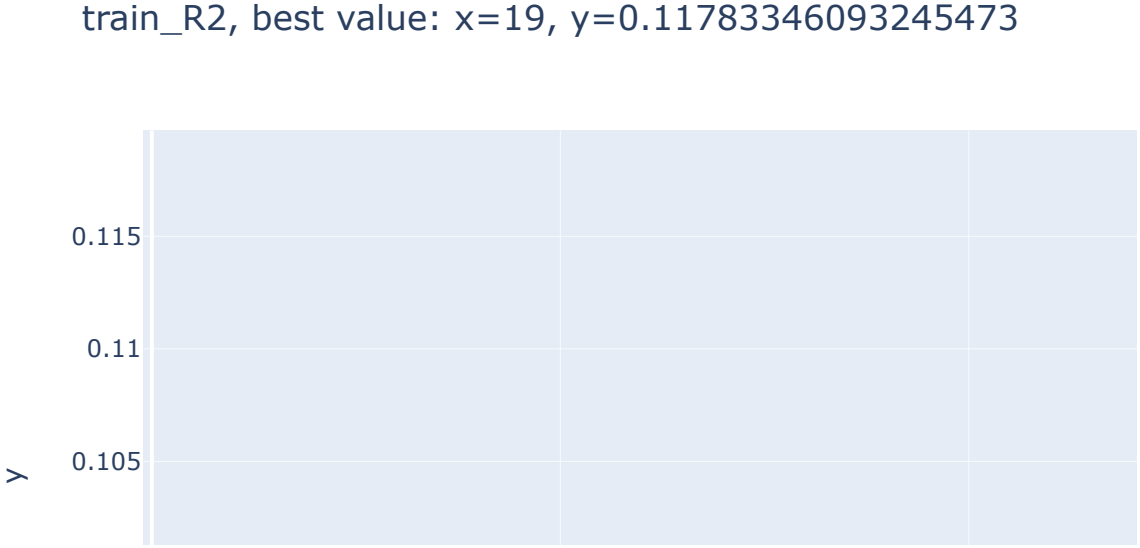
We can see that we got the `age^2` , the `age*bmi` and the `bmi^2` added columns.
Let's train NE model with few degrees and see which degree is best on the `age` feature.

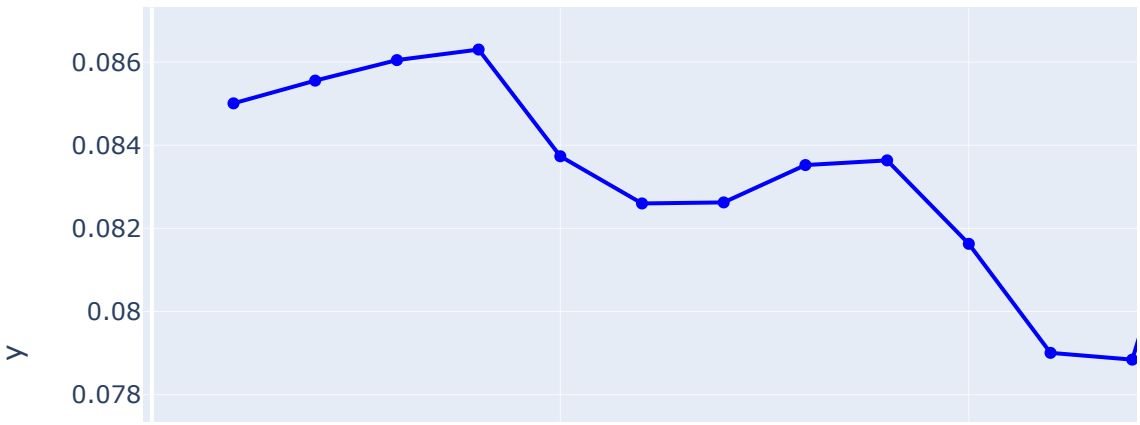
In [ ]:

```
# plot the score by degree and the loss by degree
def plot_score_and_loss_by_degree(X_train, t_train, X_val, t_val):
    graph_points = {
                    'train_MSE':{},
                    'val_MSE': {},
                    'train_R2': {},
                    'val_R2': {}
                    }

    st_scalar = preprocessing.StandardScaler().fit(X_train)
    X_train = st_scalar.transform(X_train)
    X_val = st_scalar.transform(X_val)
    max_degree_of_features = 20
    for degree in range(1, max_degree_of_features):
        NE_reg = pipeline.make_pipeline(preprocessing.PolynomialFeatures(degree, includ
e_bias=False), linear_model.LinearRegression())
        NE_reg.fit(X_train, t_train)
        y_train = NE_reg.predict(X_train)
        y_val = NE_reg.predict(X_val)
        graph_points['train_MSE'][degree] = metrics.mean_squared_error(y_train, t_train
)
        graph_points['val_MSE'][degree] = metrics.mean_squared_error(y_val, t_val)
        graph_points['train_R2'][degree] = NE_reg.score(X_train, t_train)
        graph_points['val_R2'][degree] = NE_reg.score(X_val, t_val)
    print_graphs_r2_mse(graph_points)

plot_score_and_loss_by_degree(X_train[['age']], t_train, X_val[['age']], t_val)
```

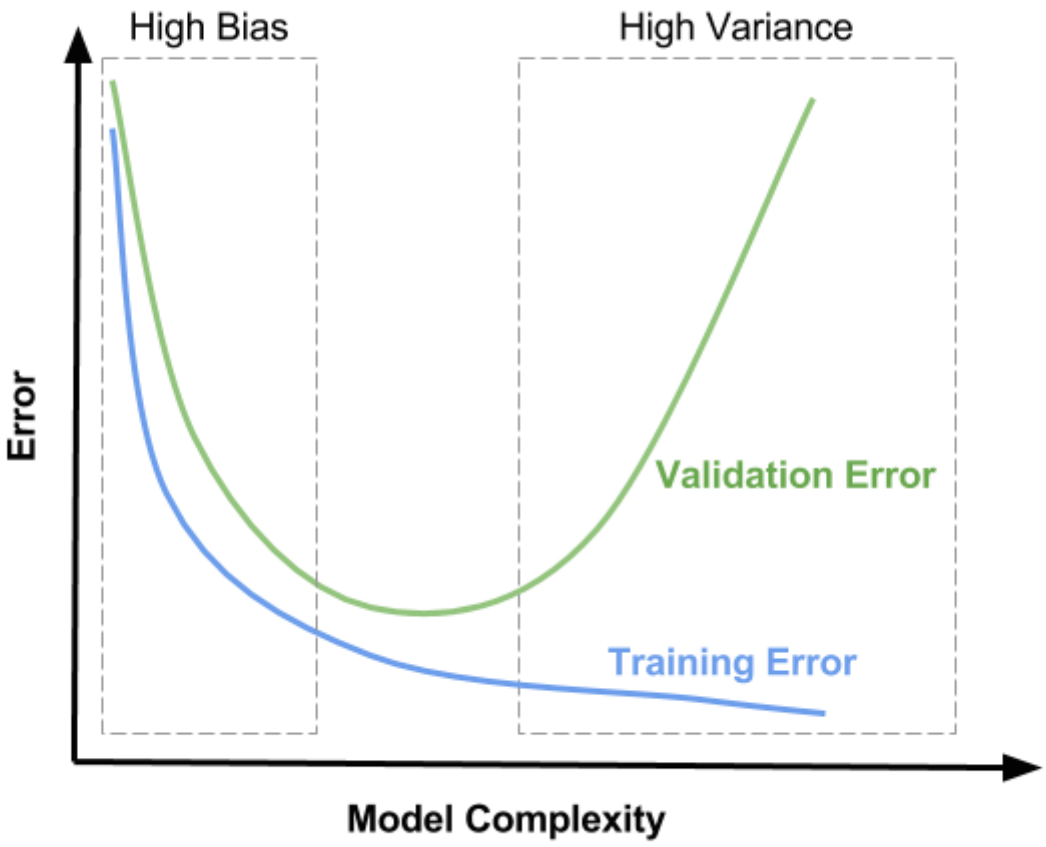

train_MSE, best value: x=19, y=132375534.64067757
136.5M
136M
135.5M
135M
134.5M
134M
y


val_MSE, best value: x=4, y=127211511.84870693
129M
128.5M
y


train_R2, best value: x=19, y=0.11783346093245473
0.115
0.11
0.105
y


val_R2, best value: x=4, y=0.08630288114179341
y
0.086
0.084
0.082
0.08
0.078

We can see that in the MSE loss of the train is smaller than the MSE loss of the validation (it means that the model is performing better on the train).
This is what will happen most of the time (but not all of the time).
The train loss is going down as the complexity of the model gets higher.
The validation loss is going down at first, until it reaches it's peek in degree 3, and then it starts going up as the complexity of the model gets higher.
The case where the train and validation/test losses are going down together is called **High Bias**.
It means that the model is not robust enough and we need to make it more complex to help it learn better on the data.
The case where the train loss is going down and the validation/test loss is going up is called **High Variance**.
It means that the model is fitted too much to the train data, and we need to make the model less complex (lower the degree).



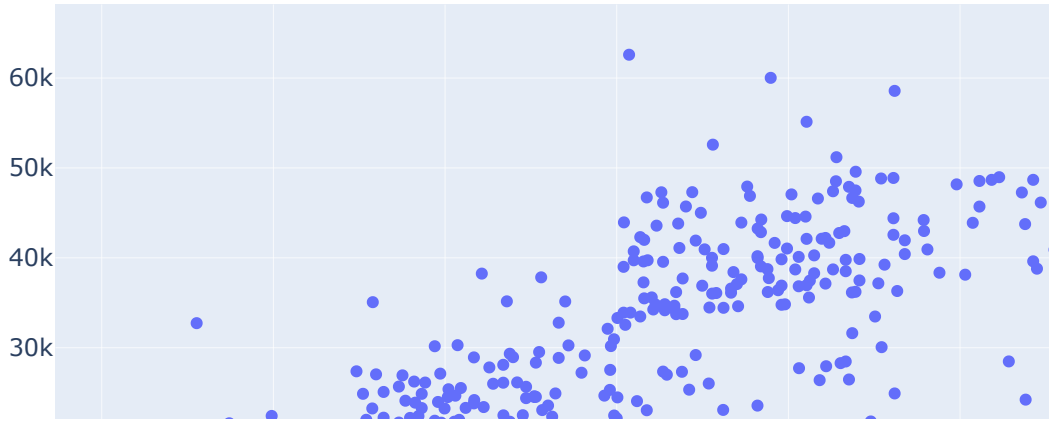
We can see the same phenomena (in the opposit direction) in the R2 score graphs. On **High Bias** the R2 score graph of the train and validation/test will go up together.
On **High Variance** the R2 score graph of the train will go up, and the R2 graph of the validation/test will go down.
Another thing we can see in the line graphs of the `age` feature, is that the R2 score is very low.
This feature alone is not enough to predict the `charges` target on its own.

Regression Graph

If we want to see the regression hypothesis in a graph, we need to choose 1 feature for a 2D graph or 2 featurs for a 3D graph.
Let's plot 2D graph with Plotly Scatter (https://plotly.com/python/line-and-scatter/) samples with the `bmi` feature on the `x` axis.
The target `charges` will be on the `y` axis.
Let's add the regression line (with NumPy linspace (https://numpy.org/doc/stable/reference/generated/numpy.linspace.html)).
We can add slider for different degrees with Plotly Slider (https://plotly.com/python/sliders/).

In [ ]:

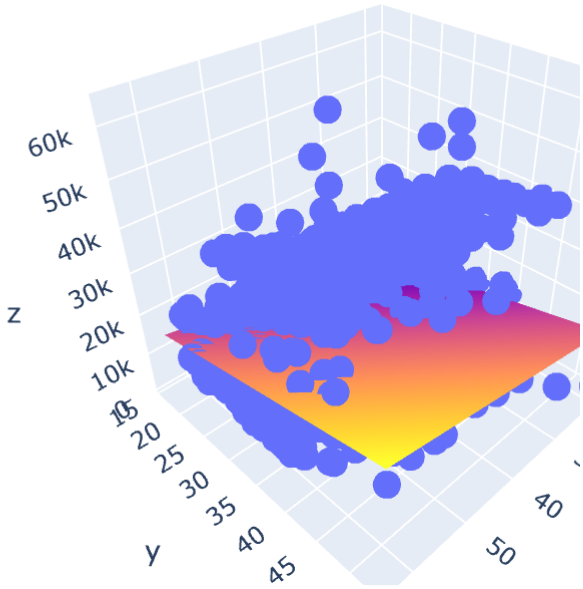
```
from plotly import graph_objects as go

# plot the samples by age and bmi, with the regression surface
def plot_samples_with_regression_line(df):
    linspace_size = 1000
    margin = 0

    X_part = df[['bmi']]
    t = df['charges']

    x_min, x_max = X_part.bmi.min() - margin, X_part.bmi.max() + margin
    xrange = pd.DataFrame(np.linspace(x_min, x_max, linspace_size), columns=['bmi'])

    graph_points = {
                'MSE':{},
                'R2': {},
                }

    fig = go.Figure()
    for degree in np.arange(1, 50):
        NE_reg = pipeline.make_pipeline(preprocessing.PolynomialFeatures(degree, includ
e_bias=False), linear_model.LinearRegression()).fit(X_part, t)
        pred = NE_reg.predict(xrange)
        fig.add_traces(go.Scatter(x=X_part['bmi'], y=t, mode='markers', visible=False,
name="original data points")) # Scatter of the original data points
        fig.add_traces(go.Scatter(x=xrange['bmi'], y=pred, mode='lines', name="line deg
ree = " + str(degree), visible=False)) # Drawing the line of the hypothesis itself
    fig.data[0].visible = True
    fig.data[1].visible = True

    steps = []
    for i in range(len(fig.data)//2):
        step = dict(
            method="update",
            args=[{"visible": [False] * len(fig.data)},
                  {"title": f"Slider switched to degree: {str(i+1)}"}],
            label=i+1
        )
        step["args"][0]["visible"][i*2] = True
        step["args"][0]["visible"][i*2+1] = True
        steps.append(step)

    sliders = [dict(
        active=0,
        currentvalue={"prefix": "Degree: "},
        steps=steps
    )]

    fig.update_layout(
        sliders=sliders
    )

    fig.show()

plot_samples_with_regression_line(insurance_df_cp)
```



Let's plot 3D graph with Plotly Scatter3d (https://plotly.com/python/3d-scatter-plots/) samples with the `age` feature on the `x` axis, and the `bmi` feature on the `y` axis.
The target `charges` will be on the `z` axis.
Let's add the regression surface (with NumPy meshgrid (https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html)).
We can add slider for different degrees with Plotly Slider (https://plotly.com/python/sliders/).

In [ ]:

```
# plot the samples by age and bmi, with the regression surface
def plot_samples_with_regression_surface(df):
    mesh_size = 1
    margin = 0

    X_part = df[['age', 'bmi']]
    t = df['charges']

    x_min, x_max = X_part.age.min() - margin, X_part.age.max() + margin
    y_min, y_max = X_part.bmi.min() - margin, X_part.bmi.max() + margin
    xrange = np.arange(x_min, x_max, mesh_size)
    yrange = np.arange(y_min, y_max, mesh_size)
    xx, yy = np.meshgrid(xrange, yrange)

    graph_points = {
                'MSE':{},
                'R2': {},
                }

    fig = go.Figure()
    for degree in np.arange(1, 20):
        NE_reg = pipeline.make_pipeline(preprocessing.PolynomialFeatures(degree, includ
e_bias=False), linear_model.LinearRegression()).fit(X_part, t)
        pred = NE_reg.predict(np.c_[xx.ravel(), yy.ravel()])
        pred = pred.reshape(xx.shape)
        fig.add_trace(go.Scatter3d(x=X_part['age'], y=X_part['bmi'], z=t, mode='marker
s', visible=False, name="original data points"))
        fig.add_traces(go.Surface(x=xrange, y=yrange, z=pred, name="surface degree = "
+ str(degree), visible=False))
    fig.data[0].visible = True
    fig.data[1].visible = True

    steps = []
    for i in range(len(fig.data)//2):
        step = dict(
            method="update",
            args=[{"visible": [False] * len(fig.data)},
                  {"title": f"Slider switched to degree: {str(i+1)}"}],
            label=i+1
        )
        step["args"][0]["visible"][i*2] = True
        step["args"][0]["visible"][i*2+1] = True
        steps.append(step)

    sliders = [dict(
        active=0,
        currentvalue={"prefix": "Degree: "},
        steps=steps
    )]

    fig.update_layout(
        sliders=sliders
    )

    fig.show()

plot_samples_with_regression_surface(insurance_df_cp)
```


60k
50k
40k
30k
20k
10k
z
15
20
25
30
35
40
45
y
40
50

More Information

Explanation on the difference between Matplotlib, Seaborn and Plotly:
Matplotlib vs. Seaborn vs. Plotly (https://towardsdatascience.com/matplotlib-vs-seaborn-vs-plotly-f2b79f5bddb)

Post on how to clean datasets using Pandas:
How To Clean Machine Learning Datasets Using Pandas (https://www.activestate.com/blog/how-to-clean-machine-learning-datasets-using-pandas/)

Explanation on the difference between scatterplot and dotplot:
Difference between scatter-plot and a dotplot (https://math.stackexchange.com/a/691754)

Tutorial on how to use bar charts with Plotly Express:
Step by step bar-charts using Plotly Express (https://towardsdatascience.com/step-by-step-bar-charts-using-plotly-express-bb13a1264a8b)

Explanation on the differences between Categorical, Ordinal and Numerical variables:
What is the Difference Between Categorical Ordinal and Numerical Variables? (https://stats.idre.ucla.edu/other/mult-pkg/whatstat/what-is-the-difference-between-categorical-ordinal-and-numerical-variables/)

Explanation on why it is important to define correctly categorical and ordinal features:
Categorical and ordinal feature data representation in regression analysis? (https://datascience.stackexchange.com/a/9211)

A package for regression tasks on ordinal target:
mord: Ordinal Regression in Python (https://pythonhosted.org/mord/)

Explanation on how to predict empty values:
Predict Missing Values in the Dataset (https://towardsdatascience.com/predict-missing-values-in-the-dataset-897912a54b7b)

Explanation on the differences between label encoding, one-hot encoding and dummy encoding:
One-Hot Encoding vs. Label Encoding using Scikit-Learn (https://www.analyticsvidhya.com/blog/2020/03/one-hot-encoding-vs-label-encoding-using-scikit-learn/)

Wikipedia on multicollinearity:
Multicollinearity (https://en.wikipedia.org/wiki/Multicollinearity)

Tutorial on how to use label encoding and one-hot encoding:
Categorical encoding using Label-Encoding and One-Hot-Encoder (https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd)

A post on the Bias-Variance Decomposition:
Bias-Variance Decomposition (http://rasbt.github.io/mlxtend/user_guide/evaluate/bias_variance_decomp/)

Examples of plots in Plotly that are best for ML Regression:
ML Regression in Python (https://plotly.com/python/ml-regression/)

Documentation of Plotly sliders:
Python Figure Reference: layout.sliders (https://plotly.com/python/reference/layout/sliders/)

How to change default control values in Plotly sliders:
Python: Change Custom Control Values in Plotly (https://stackoverflow.com/a/58976725)

An explentation on some rare train/test scenerios:
How is it possible to obtain better results on the test set than on the training set?
(https://www.researchgate.net/post/How_is_it_possible_to_obtain_better_results_on_the_test_set_than_on_th