

Medical Insurance Cost prediction

This exercise is about performing some of the steps described in the notebook for the California Housing Data on another dataset for Medical Insurance Cost prediction.

Get the Data

```
import pandas as pd
import numpy as np
import sklearn
from sklearn import *
import matplotlib.pyplot as plot

medical = pd.read_csv("https://bit.ly/44evDuW")
```

Take a Quick Look at the Data Structure

```
# display the first 5 rows of the dataset by calling the head()
function on medical
medical.head()
```

	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

Each row represents one patient. There are 7 attributes.

The `info()` method is useful to get a quick description of the data, in particular the total number of rows, each attribute's type, and the number of non-null values:

```
# get the number of rows, columns, and data types by using the info()
method
medical.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

# show the number of patients in each region by using the
# value_counts() method on the "region" column
medical.value_counts('region')

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

```

Let's look at the other fields. The describe() method shows a summary of the numerical attributes.

```

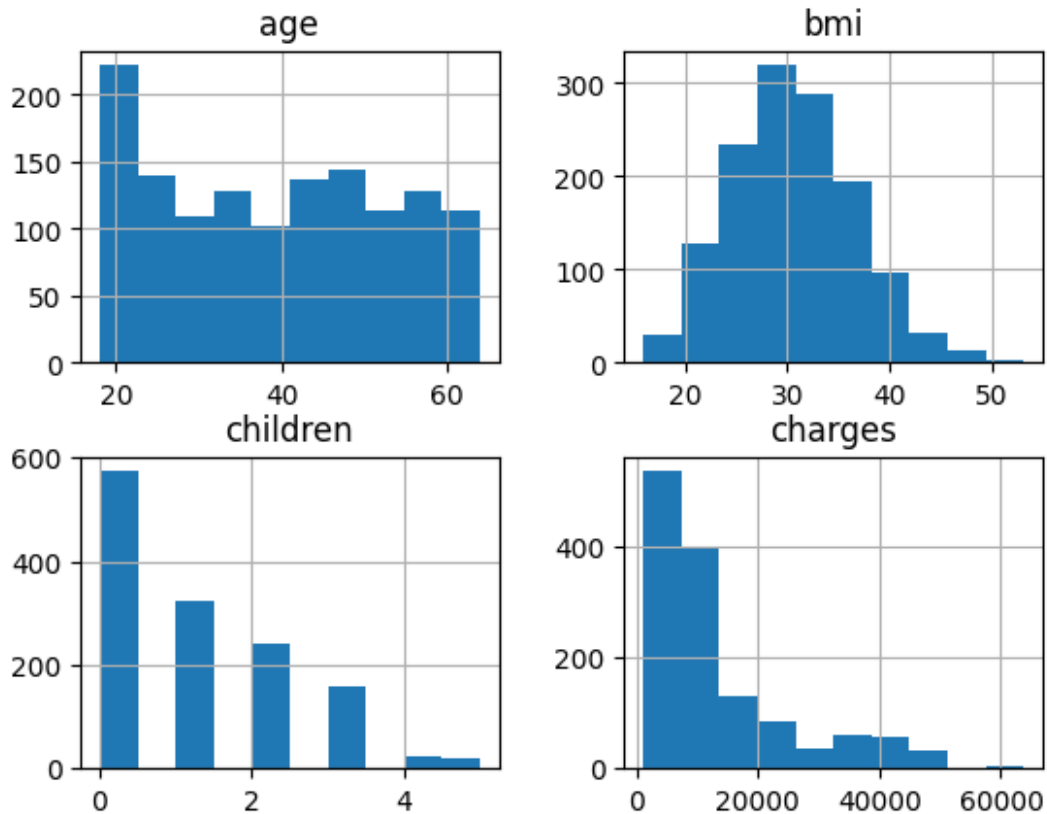
# show descriptive statistics for the dataset by calling the
# describe() method on medical
medical.describe()

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

# show histograms for the numerical columns by using the hist() method
# on medical
medical.hist()

array([[<Axes: title={'center': 'age'}>, <Axes: title={'center':
' bmi'}>],
      [<Axes: title={'center': 'children'}>,
       <Axes: title={'center': 'charges'}>]], dtype=object)

```



Briefly write here what you observe from these histograms.

Create a Test Set

```
# use train_test_split() to split the data into training and test sets
med_train, med_test =
sklearn.model_selection.train_test_split(medical, random_state=42)
```

Explore and Visualize the Data to Gain Insights

So far you have only taken a quick glance at the data to get a general understanding of the kind of data you are manipulating. Now the goal is to go into a little more depth.

First, make sure you have put the test set aside and you are only exploring the training set.

```
# make a copy of the train set and save it to a variable called
medical
medical = med_train.copy()
medical.head()
```

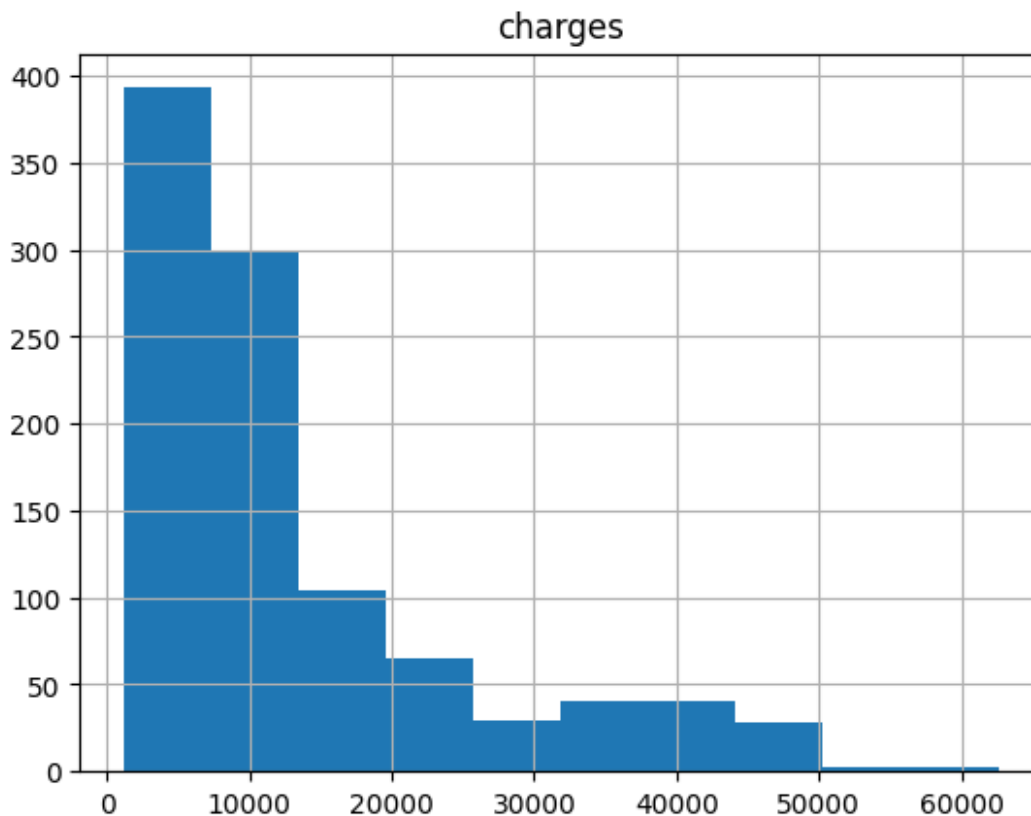
	age	sex	bmi	children	smoker	region	charges
693	24	male	23.655	0	no	northwest	2352.96845
1297	28	female	26.510	2	no	southeast	4340.44090

634	51	male	39.700	1	no	southwest	9391.34600
1022	47	male	36.080	1	yes	southeast	42211.13820
178	46	female	28.900	2	no	southwest	8823.27900

```
# build a histogram of the charges column
```

```
medical.hist('charges')
```

```
array([[<Axes: title={'center': 'charges'}>]], dtype=object)
```



This distribution is right-skewed. To make it closer to normal we can apply natural log

```
# apply a log transformation to the charges column using the  
np.log10() function
```

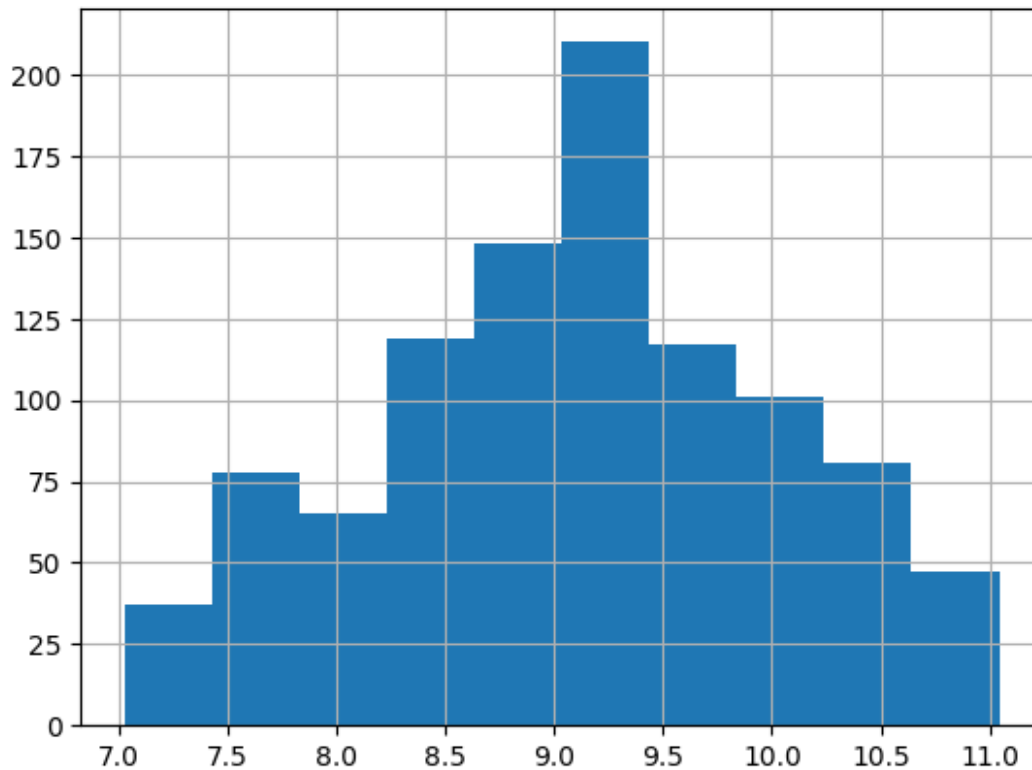
```
# build a histogram of the transformed column
```

```
log10_transformer = sklearn.preprocessing.FunctionTransformer(np.log,  
inverse_func=np.exp, feature_names_out = 'one-to-one')
```

```
log_charges = log10_transformer.transform(medical['charges'])
```

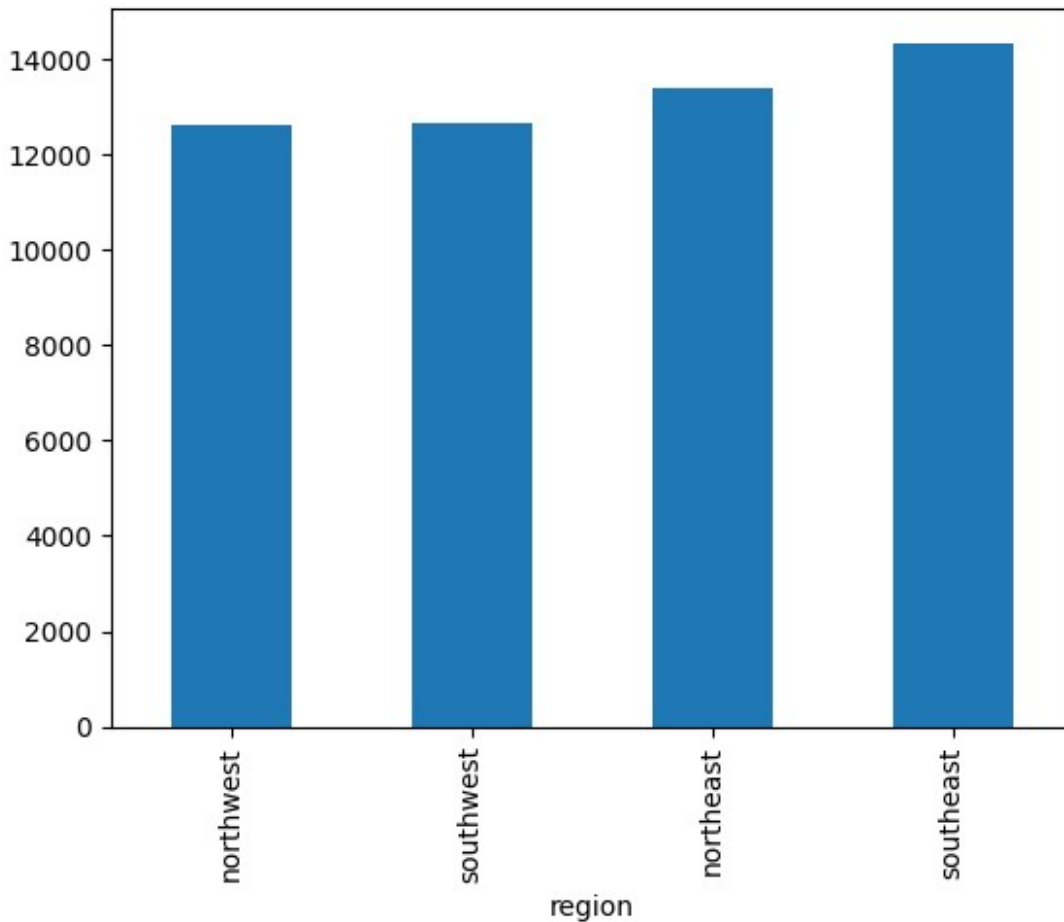
```
log_charges.hist()
```

```
<Axes: >
```



Now let's look at the mean charges by region

```
# compute the average insurance cost for each region  
# sort the charges_by_region Series from the lowest to highest cost  
# plot the sorted Series using the plot.bar() method  
  
charges_by_region = medical.groupby('region').mean(True)['charges']  
charges_by_region.sort_values().plot.bar()  
<Axes: xlabel='region'>
```



Overall the highest medical charges are in the Southeast and the lowest are in the Southwest. Taking into account certain factors (sex, smoking, having children) let's see how it changes by region.

Now, create three grouped barcharts for average charges by region grouped by sex, smoking, and number of children.

How to create grouped barcharts?

Creating grouped bar charts with Seaborn is a bit more intuitive compared to Matplotlib. You can use the `catplot` function with `kind='bar'` to create grouped bar charts. Here is an example on the tips datasets that comes with Seaborn. The tips dataset contains information about the total bill and tip amount for different meals, along with additional information such as the sex of the individual paying for the meal, whether they are a smoker, the day and time of the meal, and the size of the party.

We will create a grouped bar chart showing the average total bill for each day, grouped by whether the meal took place at lunch or dinner.

```
import seaborn as sns
import matplotlib.pyplot as plt
```

```
# Load the 'tips' dataset
tips = sns.load_dataset("tips")

# Create a grouped bar chart
sns.catplot(data=tips, x="day", y="total_bill", hue="time",
kind="bar")

plt.show()
```

In this plot, the height of the bars represents the average total bill for meals on each day, with separate bars for lunch and dinner.

The `catplot` function is a flexible function that can create a variety of different plot types. By setting `kind='bar'`, we specify that we want a bar chart. The `x` and `y` arguments specify the data for the x and y axes, and the `hue` argument specifies a third variable that is used to group the data.

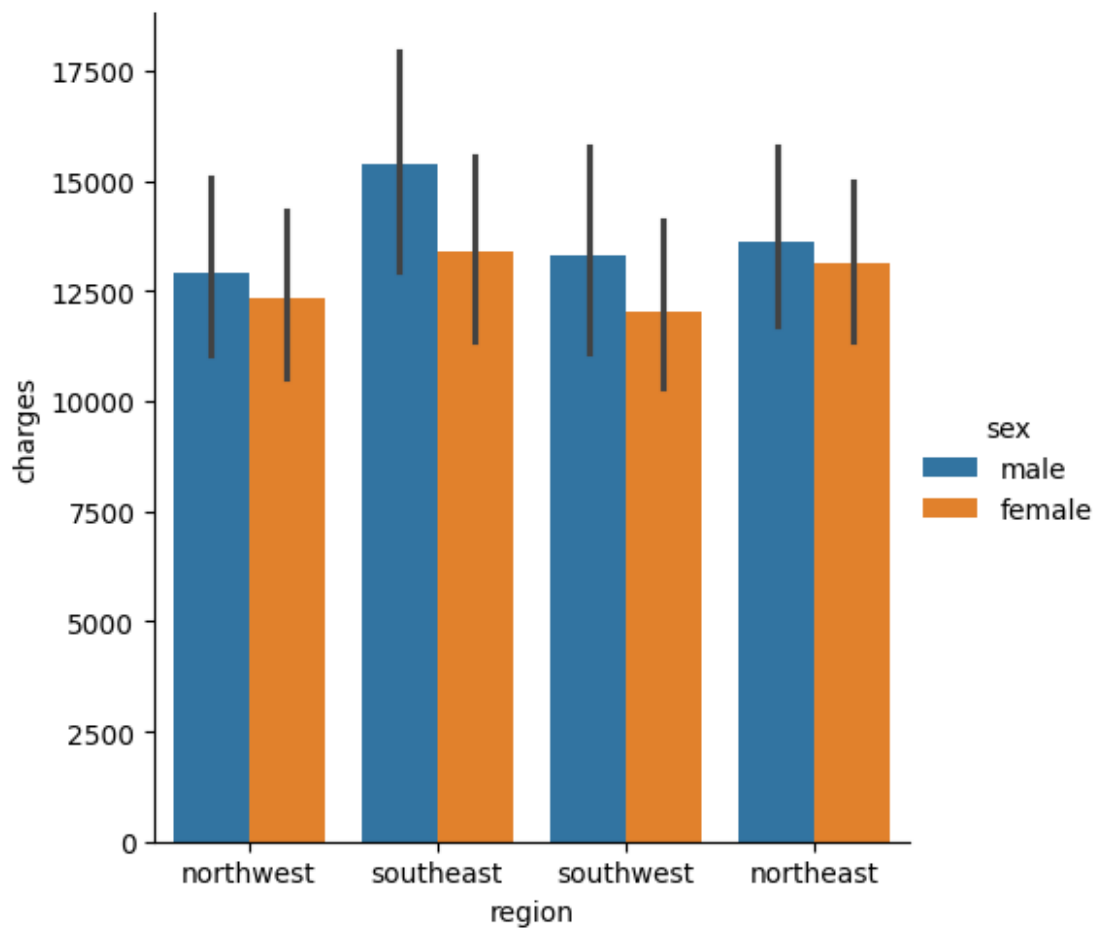
In the context of Seaborn and many other statistical visualization libraries, error bars commonly represent one standard deviation or standard error of the mean.

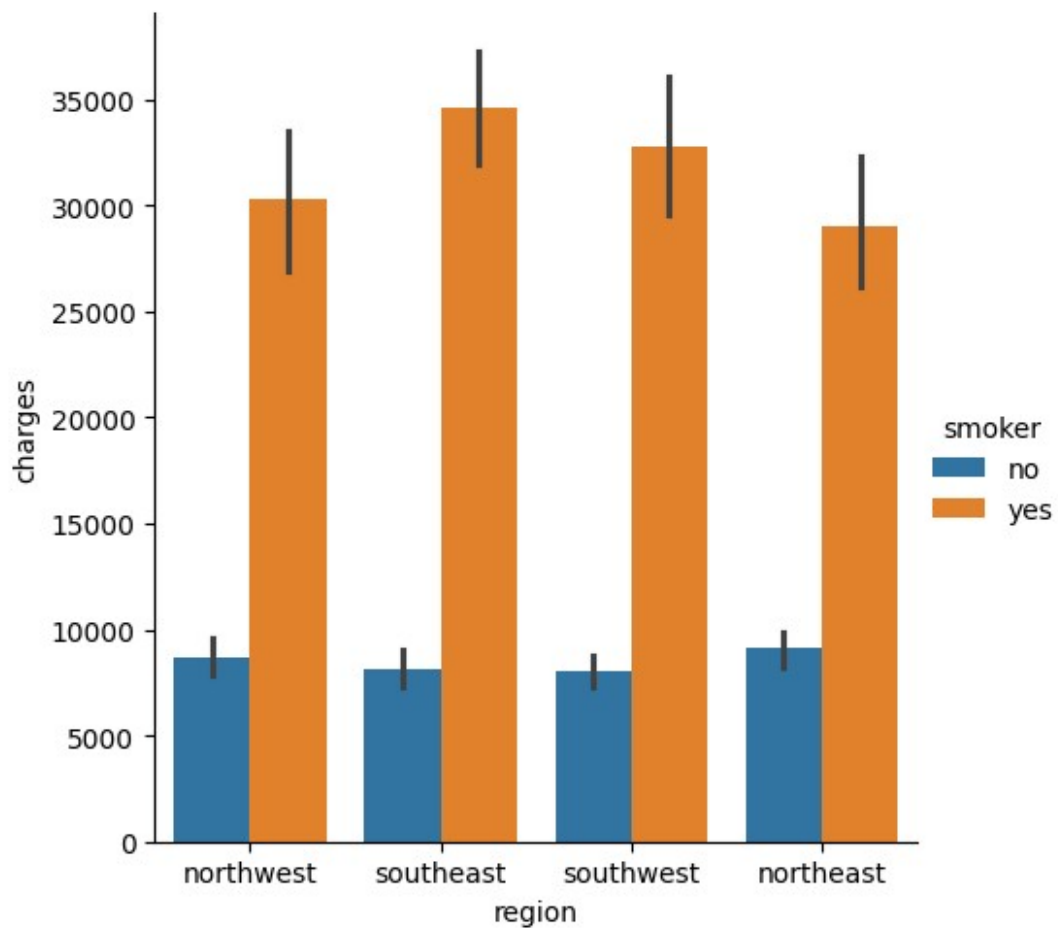
```
# plot grouped bar charts of region and insurance costs hue by sex,
smoker, and number of children (three separate charts)
# use the catplot() function to create the bar charts
# set the kind parameter to "bar" and the data parameter to medical
import seaborn as sns
import matplotlib.pyplot as plt

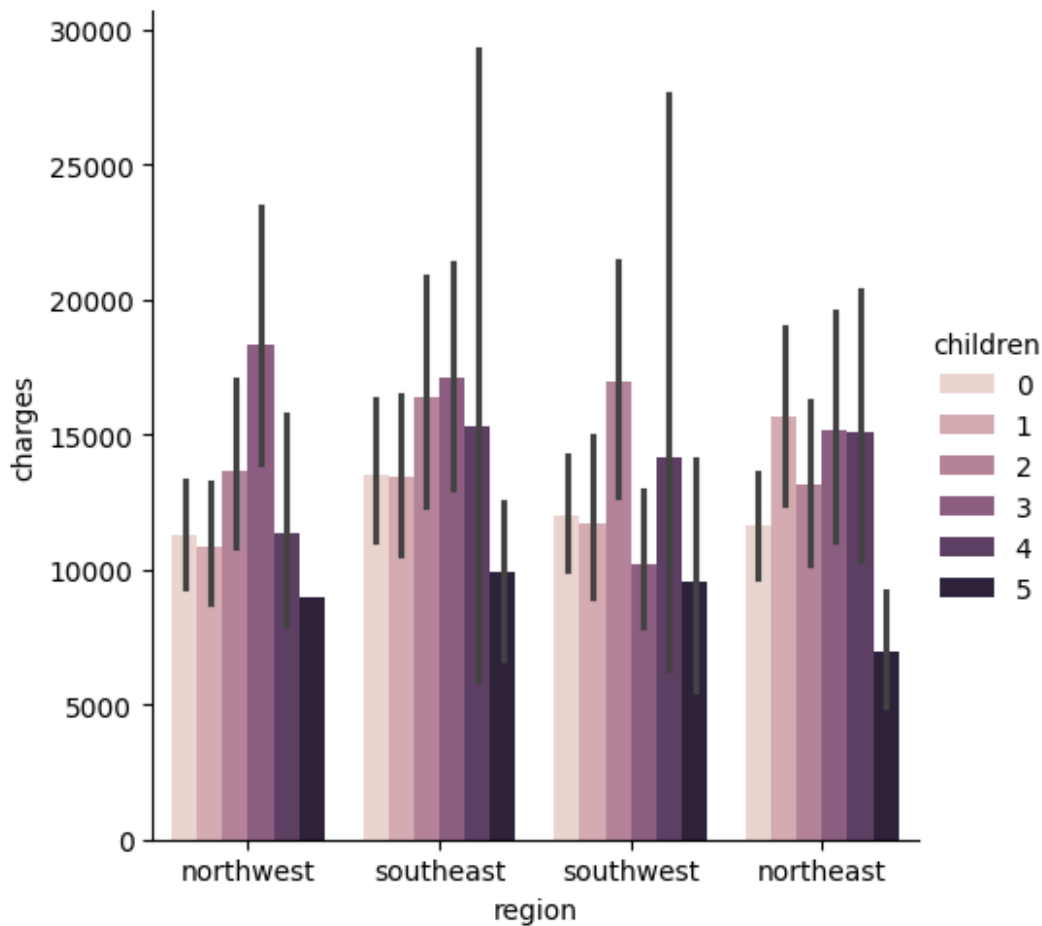
# Create a grouped bar chart
sns.catplot(data=medical, x='region', y='charges', hue='sex',
kind='bar')
plt.show()

sns.catplot(data=medical, x='region', y='charges', hue='smoker',
kind='bar')
plt.show()

sns.catplot(data=medical, x='region', y='charges', hue='children',
kind='bar')
plt.show()
```







What do you observe? Briefly write what you observe from the charts.

- Whether you smoke or not is a huge determining factor in how much your insurance costs
- Number of children doesn't seem to be very related with how much your insurance costs
- Women are more likely to have lower costing insurance than men

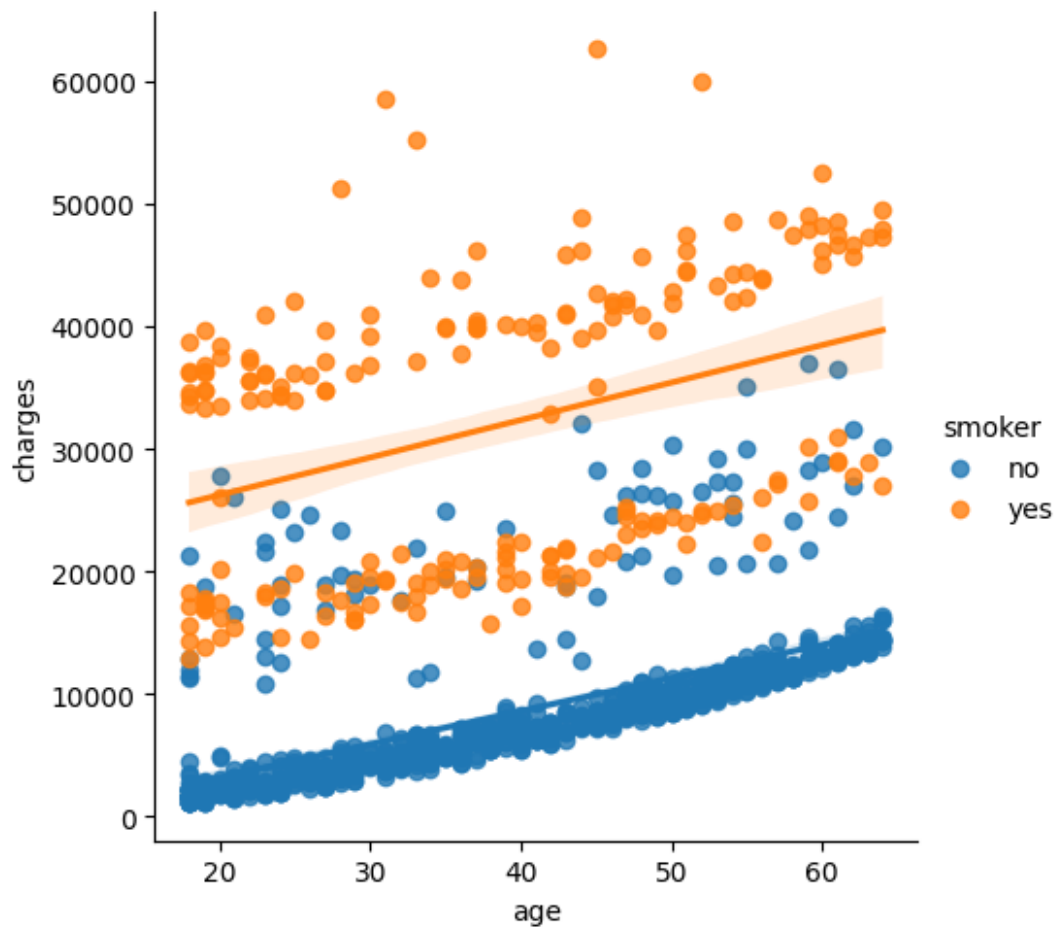
Now let's analyze the medical charges by age, bmi and children according to the smoking factor.

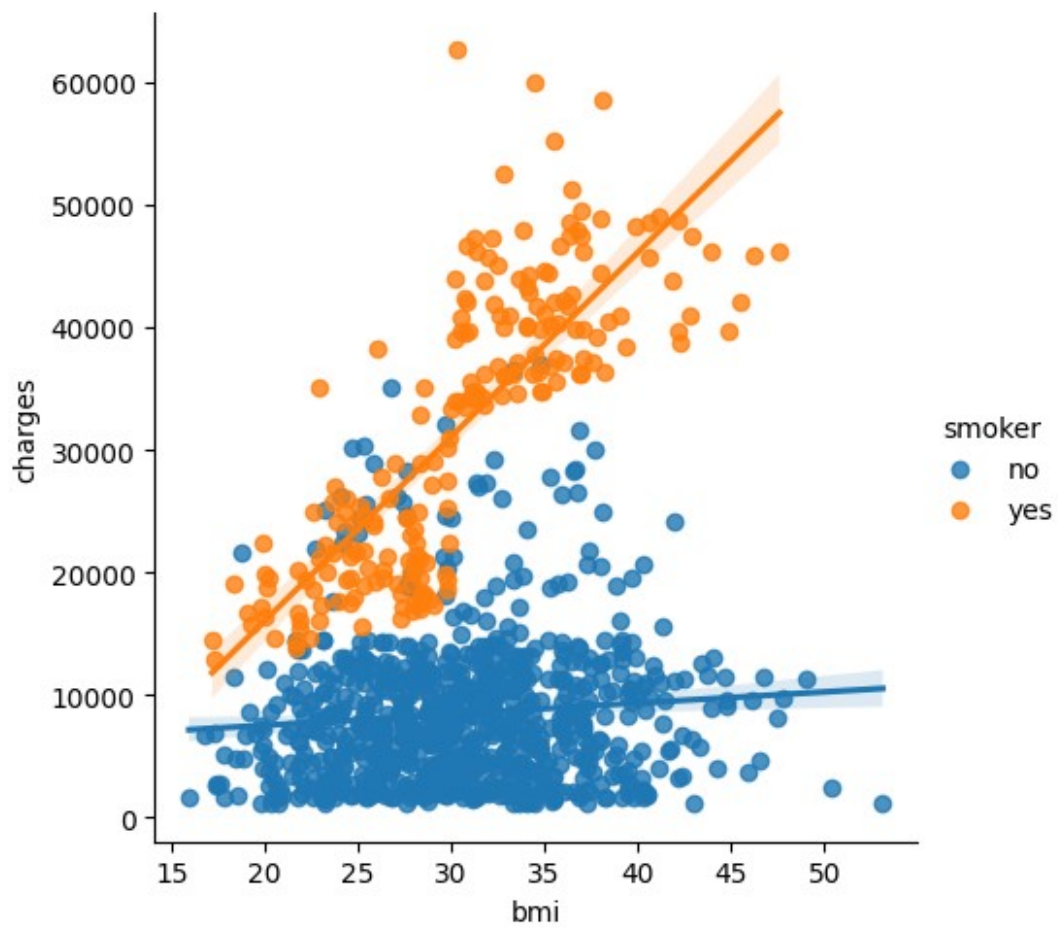
```
# using the lmpplot() function of seaborn, build a scatter plot of age  
and insurance costs, hue by smoker  
# build a second scatter plot of bmi and insurance costs, hue by  
smoker  
# build a third scatter plot of children and insurance costs, hue by  
smoker
```

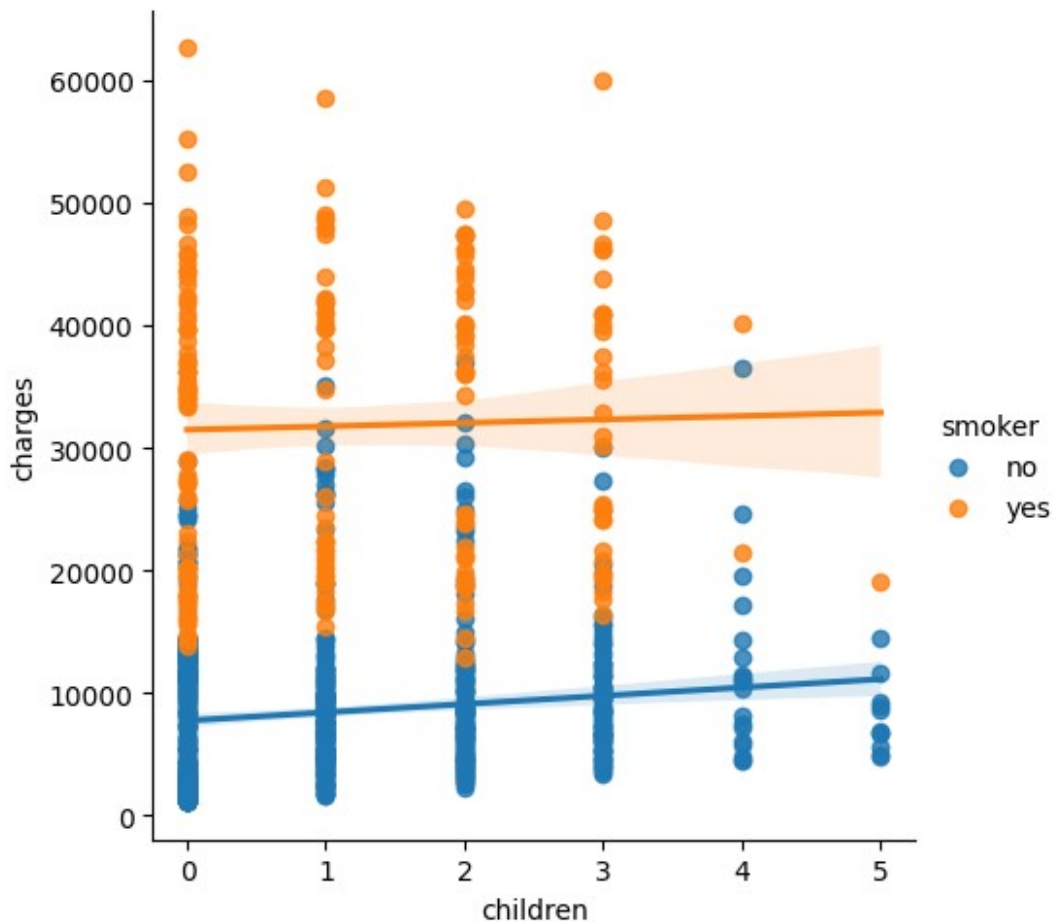
```
sns.lmpplot(data=medical, x='age', y='charges', hue='smoker')  
plt.show()
```

```
sns.lmpplot(data=medical, x='bmi', y='charges', hue='smoker')  
plt.show()
```

```
sns.lmplot(data=medical, x='children', y='charges', hue='smoker')  
plt.show()
```







Describe in a one-liner what you observe from the charts.

Smoking combined with high BMI is a bad combination for insurance costs, with a much greater slope on the fitted line. # of children again doesn't factor much. Age and smoking doesn't seem to have a ton of correlation, its just shifted up due to the presence of smoking.

Look for Correlations

```
# encode the categorical variables using the OneHotEncoder
cat_encoder = OneHotEncoder()
medical_encoded = cat_encoder.fit_transform(medical[cat_attribs])

# concatenate the encoded categorical variables with the numerical
variables
medical_prepared_encoded = np.concatenate((medical_encoded.toarray(),
medical[num_attribs]), axis=1)

# compute pairwise correlation of columns using the corr() method
correlation_matrix = pd.DataFrame(medical_prepared_encoded,
columns=cat_encoder.get_feature_names_out().tolist() +
num_attribs).corr()
correlation_matrix
```

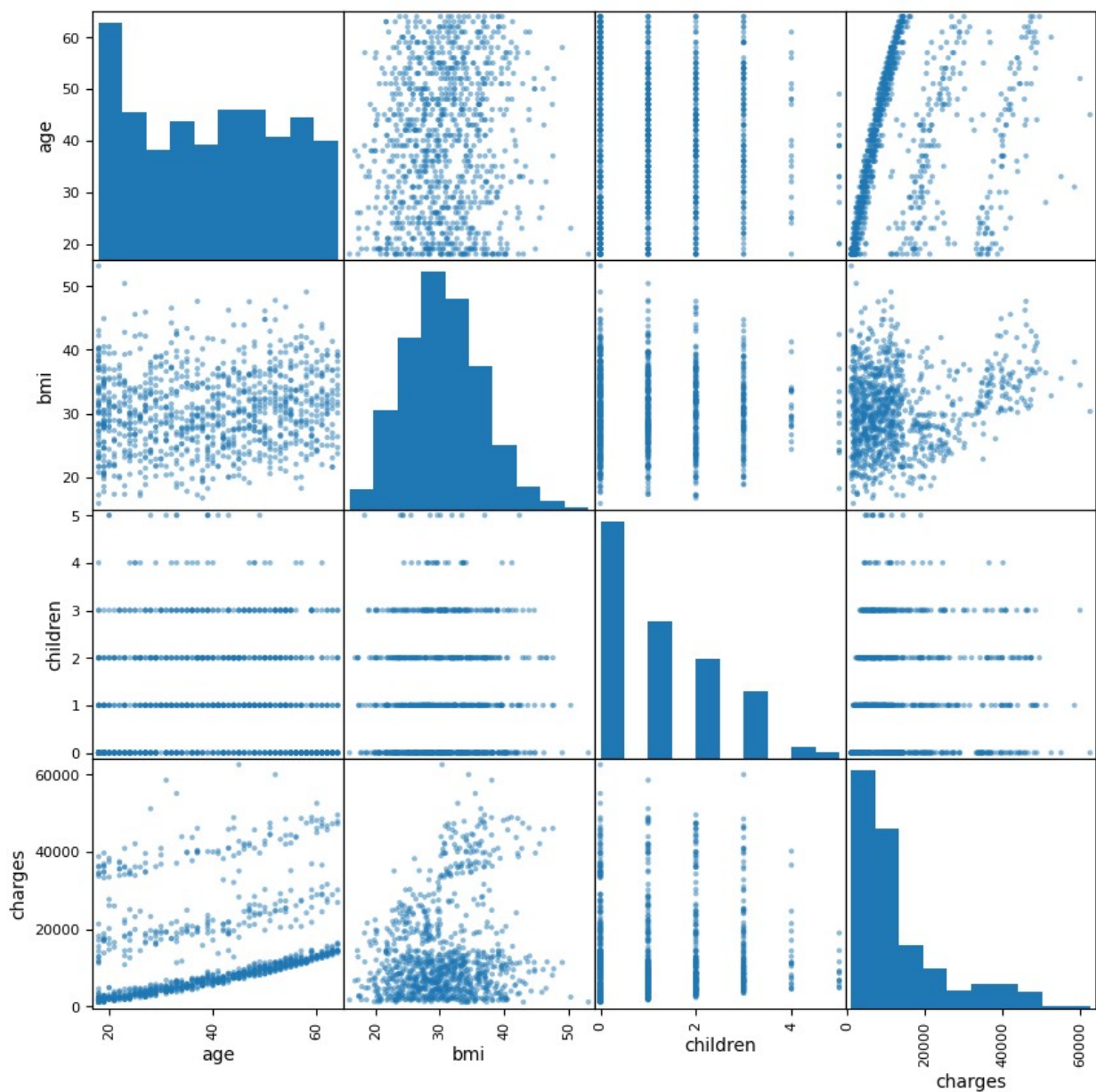
	sex_female	sex_male	smoker_no	smoker_yes	\
sex_female	1.000000	-1.000000	0.054671	-0.054671	
sex_male	-1.000000	1.000000	-0.054671	0.054671	
smoker_no	0.054671	-0.054671	1.000000	-1.000000	
smoker_yes	-0.054671	0.054671	-1.000000	1.000000	
region_northeast	-0.010928	0.010928	-0.013951	0.013951	
region_northwest	-0.031963	0.031963	0.031689	-0.031689	
region_southeast	0.025798	-0.025798	-0.044016	0.044016	
region_southwest	0.016700	-0.016700	0.027094	-0.027094	
age	0.012385	-0.012385	0.037916	-0.037916	
bmi	-0.001931	0.001931	0.001499	-0.001499	
children	-0.016610	0.016610	-0.025488	0.025488	
	region_northeast	region_northwest	region_southeast		
\					
sex_female	-0.010928	-0.031963	0.025798		
sex_male	0.010928	0.031963	-0.025798		
smoker_no	-0.013951	0.031689	-0.044016		
smoker_yes	0.013951	-0.031689	0.044016		
region_northeast	1.000000	-0.329342	-0.340872		
region_northwest	-0.329342	1.000000	-0.336343		
region_southeast	-0.340872	-0.336343	1.000000		
region_southwest	-0.330229	-0.325842	-0.337249		
age	-0.012805	0.016715	-0.023557		
bmi	-0.154166	-0.109244	0.238183		
children	0.009031	0.038969	-0.032380		
	region_southwest	age	bmi	children	
sex_female	0.016700	0.012385	-0.001931	-0.016610	
sex_male	-0.016700	-0.012385	0.001931	0.016610	
smoker_no	0.027094	0.037916	0.001499	-0.025488	
smoker_yes	-0.027094	-0.037916	-0.001499	0.025488	
region_northeast	-0.330229	-0.012805	-0.154166	0.009031	
region_northwest	-0.325842	0.016715	-0.109244	0.038969	
region_southeast	-0.337249	-0.023557	0.238183	-0.032380	
region_southwest	1.000000	0.020113	0.022136	-0.015102	
age	0.020113	1.000000	0.111325	0.055070	
bmi	0.022136	0.111325	1.000000	-0.003459	
children	-0.015102	0.055070	-0.003459	1.000000	

The correlation coefficient ranges from -1 to 1 . When it is close to 1 , it means that there is a strong positive correlation. Finally, coefficients close to 0 mean that there is no linear correlation.

Another way to check for correlation between attributes is to use the Pandas `scatter_matrix()` function, which plots every numerical attribute against every other numerical attribute.

```
# plot correlation matrix using scatter_matrix() function from  
pandas.plotting
```

```
pd.plotting.scatter_matrix(medical, figsize=(10, 10))  
plt.show()
```



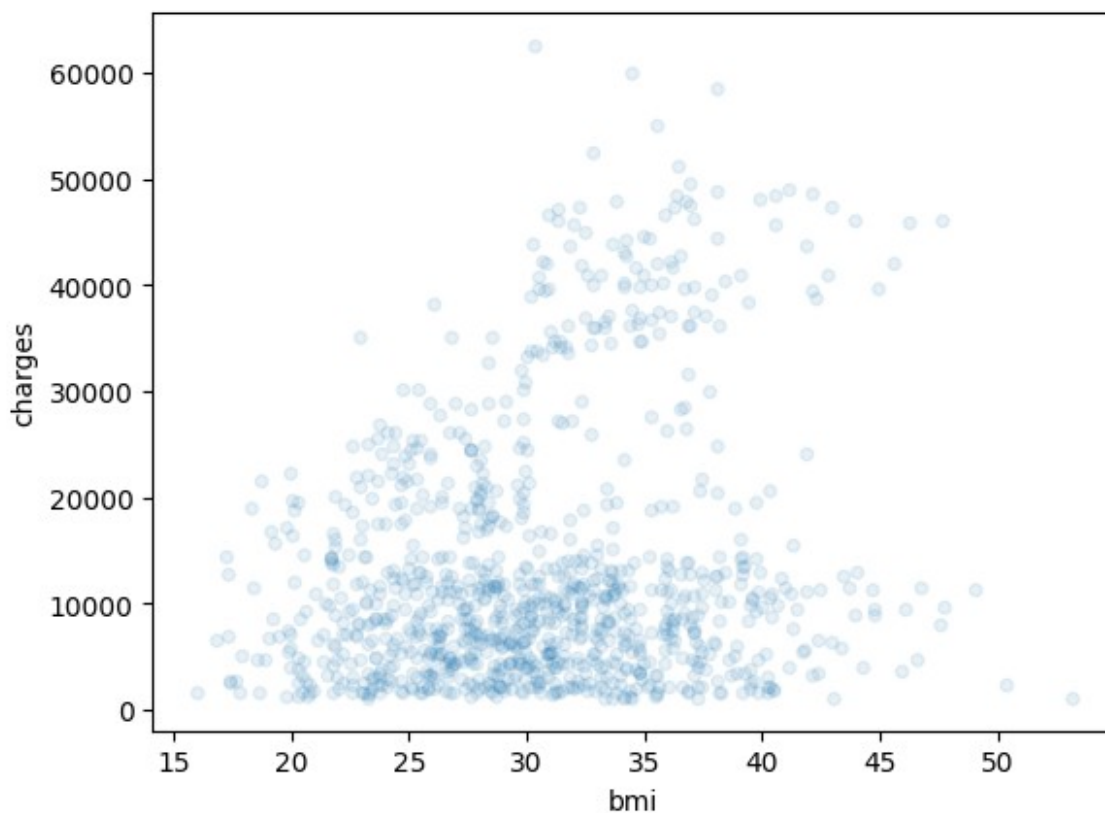
The main diagonal would be full of straight lines if Pandas plotted each variable against itself, which would not be very useful. So instead, the Pandas displays a histogram of each attribute.

Looking at the correlation scatterplots, it seems like the most promising attribute to predict the charge value is bmi, so let's zoom in on their scatterplot.

```
# plot a scatter plot of bmi vs. insurance costs using the  
medical.plot() method, use the alpha parameter to set the opacity of  
the points to 0.1
```

```
medical.plot(kind='scatter', x='bmi', y='charges', alpha=0.1)
```

```
<Axes: xlabel='bmi', ylabel='charges'>
```



Prepare the data for ML

```
# drop the charges column from the train_set and save the resulting  
dataset to a variable called `medical`  
# create a copy of the train_set labels and save it to a variable  
called `medical_labels`  
# replace None with the correct code
```



```
medical = med_train.drop('charges', axis=1)
medical_labels = med_train['charges'].copy()
```

Transformation Pipelines

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the Pipeline class to help with such sequences of transformations.

```
# uncomment the following code to create a pipeline for preprocessing the data

from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import FunctionTransformer
from sklearn.pipeline import make_pipeline

num_attribs = ["age", "bmi", "children"]
cat_attribs = ["sex", "smoker", "region"]

num_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    StandardScaler())

cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore"))

preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs)])

medical_prepared = preprocessing.fit_transform(medical)

print(medical_prepared.shape)
print(preprocessing.get_feature_names_out())

(1003, 11)
['num__age' 'num__bmi' 'num__children' 'cat__sex_female'
'cat__sex_male'
'cat__smoker_no' 'cat__smoker_yes' 'cat__region_northeast'
'cat__region_northwest' 'cat__region_southeast'
'cat__region_southwest']

import sklearn
print(sklearn.__version__)
```

Select and Train a Model

At last! You framed the problem, you got the data and explored it, you sampled a training set and a test set, and you wrote a preprocessing pipeline to automatically clean up and prepare your data for machine learning algorithms. You are now ready to select and train a machine learning model.

Train and Evaluate on the Training Set

The good news is that thanks to all these previous steps, things are now going to be easy! You decide to train a very basic linear regression model to get started:

```
from sklearn import linear_model
# create a pipeline for preprocessing the data and fitting a linear
# regression model

lin_reg = make_pipeline(preprocessing,
linear_model.LinearRegression())

# housing_labels is the column we want to predict
# uncomment the following line to fit the model

lin_reg.fit(medical, medical_labels)

Pipeline(steps=[('columntransformer',
ColumnTransformer(transformers=[('num',

Pipeline(steps=[('simpleimputer',

SimpleImputer(strategy='median')),

('standardscaler',

StandardScaler()))]),

[('age', 'bmi',

'children']),

('cat',

Pipeline(steps=[('simpleimputer',

SimpleImputer(strategy='most_frequent')),

('onehotencoder',

OneHotEncoder(handle_unknown='ignore'))]),

[('sex', 'smoker',
```

```
        'region']]))),  
        ('linearregression', LinearRegression()))]
```

Try out the model on the training set, look at the first five predictions and compare them to the labels:

```
# uncomment the following line to make predictions  
medical_predictions = lin_reg.predict(medical)  
  
print(medical_predictions[:5])  
print(medical_labels.head().values)  
  
[ 1821.07697573  4210.87812211 14204.47142399 35645.17999288  
 9617.33921854]  
[ 2352.96845  4340.4409   9391.346   42211.1382   8823.279 ]  
  
# uncomment the following lines to compute the RMSE  
  
from sklearn.metrics import mean_squared_error  
  
lin_rmse = mean_squared_error(medical_labels, medical_predictions,  
                              squared=False)  
lin_rmse  
  
e:\Program Files (x86)\Python\Lib\site-packages\sklearn\metrics\  
_regression.py:483: FutureWarning: 'squared' is deprecated in version  
1.4 and will be removed in 1.6. To calculate the root mean squared  
error, use the function'root_mean_squared_error'.  
    warnings.warn(  
  
6083.132596294014
```

Now try `DecisionTreeRegressor`, as this is a fairly powerful model capable of finding complex nonlinear relationships in the data (decision trees are covered later in the course):

```
# use DecisionTreeRegressor to train the model  
# use the make_pipeline() function to create a pipeline for  
# preprocessing and model training  
# use the preprocessing object you created earlier  
# make predictions on the training set and compute the RMSE  
  
from sklearn.tree import DecisionTreeRegressor  
  
tree_reg = make_pipeline(preprocessing,  
                          DecisionTreeRegressor(random_state=42))  
tree_reg.fit(medical, medical_labels)  
  
medical_predictions = tree_reg.predict(medical)
```

```
tree_rmse = mean_squared_error(medical_labels, medical_predictions,
squared=False)
tree_rmse
```

```
e:\Program Files (x86)\Python\Lib\site-packages\sklearn\metrics\
_regression.py:483: FutureWarning: 'squared' is deprecated in version
1.4 and will be removed in 1.6. To calculate the root mean squared
error, use the function'root_mean_squared_error'.
warnings.warn(
```

```
427.37357318969015
```

Better Evaluation Using Cross-Validation

The following code randomly splits the training set into 10 nonoverlapping subsets called folds, then it trains and evaluates the decision tree model 10 times, picking a different fold for evaluation every time and using the other 9 folds for training. The result is an array containing the 10 evaluation scores:

```
# uncomment the following lines to train the model and make
predictions

from sklearn.model_selection import cross_val_score

tree_rmses = -cross_val_score(tree_reg,
                              medical, medical_labels,
                              scoring="neg_root_mean_squared_error",
                              cv=10)

tree_rmses

array([6404.58583575, 5375.26580028, 6169.00946342, 5539.19412464,
       7407.79754914, 6067.56921586, 7524.60259553, 8282.02682165,
       6435.91872692, 5621.22279447])
```

Warning. Scikit-Learn's cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better), so the scoring function is actually the opposite of the RMSE. It's a negative value, so you need to switch the sign of the output to get the RMSE scores.

```
# uncomment the following line to compute the mean of the RMSEs

np.mean(tree_rmses)

6482.71929276435
```

Let's try one last model now: the RandomForestRegressor. As you will see later in the course, random forests work by training many decision trees on random subsets of the features, then averaging out their predictions.

```

# use RandomForestRegressor to train the model
# use the make_pipeline() function to create a pipeline for
preprocessing and model training
# use the preprocessing object you created earlier
# make predictions on the training set and compute the RMSEs using
cross-validation
# compute the mean of the RMSEs

from sklearn.ensemble import RandomForestRegressor

forest_reg = make_pipeline(preprocessing,
RandomForestRegressor(random_state=42))

forest_rmses = -cross_val_score(forest_reg,
                                medical, medical_labels,
                                scoring="neg_root_mean_squared_error",
                                cv=10)

forest_rmses

array([4516.74210672, 4864.50605645, 3695.62496178, 4493.95278316,
       5532.90528445, 4524.53379677, 5518.32768635, 5750.80171261,
       5267.20806735, 4688.8704783 ])

```

Fine-Tune Your Model

Let's assume that you now have a shortlist of promising models. You now need to fine-tune them.

Randomized Search for Good Hyperparameters

```

# uncomment the following lines to search for the best hyperparameters

from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint
from sklearn.pipeline import Pipeline

full_pipeline = Pipeline([("preprocessing", preprocessing),
                           ("random_forest",
RandomForestRegressor(random_state=42)),
                           ])

param_distributions = {'random_forest__max_features':
randint(low=2,high=20)}

rnd_search = RandomizedSearchCV(full_pipeline,
                                param_distributions=param_distributions,
                                n_iter=10,

```

```

        cv=3,
        scoring='neg_root_mean_squared_error',

        random_state=42)

rnd_search.fit(medical, medical_labels)
RandomizedSearchCV(cv=3,
                   estimator=Pipeline(steps=[('preprocessing',
ColumnTransformer(transformers=[('num',
Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='median')),
('standardscaler',
StandardScaler())])),
['age',
'bmi',
'children']),
('cat',
Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='most_frequent')),
('onehotencoder',
OneHotEncoder(handle_unknown='ignore'))])),
['sex',
'smoker',
'region'])])),
                                ('random_forest',
RandomForestRegressor(random_state=42))]),
        param_distributions={'random_forest__max_features':
<scipy.stats._distn_infrastructure.rv_discrete_frozen object at
0x000001F7F0193FB0>},
        random_state=42,
        scoring='neg_root_mean_squared_error')

```

```

# uncomment the following lines to print the best search scores

rn_res = pd.DataFrame(rnd_search.cv_results_)
rn_res.sort_values(by="mean_test_score", ascending=False,
inplace=True)
rn_res.head(5)["mean_test_score"]

7    -4802.519040
9    -4814.169971
0    -4891.998129
4    -4891.998129
3    -4897.026072
Name: mean_test_score, dtype: float64

# uncomment the following lines to print the feature importances

final_model = rnd_search.best_estimator_ # includes preprocessing
feature_importances =
final_model["random_forest"].feature_importances_
feature_importances

array([0.14371669, 0.18401593, 0.02196867, 0.00444203, 0.00480849,
       0.28081164, 0.34035216, 0.00572992, 0.00458347, 0.00550333,
       0.00406767])

# uncomment the following line to print the feature importances with
the feature names

sorted(zip(feature_importances,
final_model["preprocessing"].get_feature_names_out()),reverse=True)

[(0.34035216442476723, 'cat__smoker_yes'),
 (0.28081164480176857, 'cat__smoker_no'),
 (0.18401592877486048, 'num__bmi'),
 (0.14371668867549603, 'num__age'),
 (0.021968668681653616, 'num__children'),
 (0.005729916237902449, 'cat__region_northeast'),
 (0.005503332128132486, 'cat__region_southeast'),
 (0.0048084927716275944, 'cat__sex_male'),
 (0.004583467693537606, 'cat__region_northwest'),
 (0.0044420289175483895, 'cat__sex_female'),
 (0.004067666892705595, 'cat__region_southwest')]

# now that you have a final model, evaluate it on the test set (find
rmse)

final_predictions = final_model.predict(med_test)

final_rmse = mean_squared_error(med_test['charges'],
final_predictions, squared=False)

```

```
final_rmse
```

```
e:\Program Files (x86)\Python\Lib\site-packages\sklearn\metrics\  
_regression.py:483: FutureWarning: 'squared' is deprecated in version  
1.4 and will be removed in 1.6. To calculate the root mean squared  
error, use the function 'root_mean_squared_error'.  
  warnings.warn(
```

```
4618.046806938494
```