MLPy Workshop 4

Student 1, Student 2

February 6, 2025

1 Week 4 - Regression and Model Evaluation

1.1 Aims

By the end of this notebook you will be able to

- fit a linear regression
- understand the basics of polynomial regression
- understand how to evaluate and compare models and select tuning parameters with training, validation, and testing.

1. Problem Definition and Setup
2. Exploratory Data Analysis
3. Least Squares Estimation
4. Regression using scikit-Learn
5. Polynomial Regression

During workshops, you will complete the worksheets together in teams of 2-3, using **pair programming**. From this week onwards, the worksheets will no longer contain cues to switch roles between driver and navigator; this should occur approximately every 15 minutes and should be more natural after the first weeks. When completing worksheets:

- You will have tasks tagged by (CORE) and (EXTRA).
- Your primary aim is to complete the (CORE) components during the WS session, afterwards you can try to complete the (EXTRA) tasks for your self-learning process.

Instructions for submitting your workshops can be found at the end of worksheet. As a reminder, you must submit a pdf of your notebook on Learn by 16:00 PM on the Friday of the week the workshop was given.

---

2 Problem Definition and Setup

2.1 Packages

First, let's load the packages you wil need for this workshop.

[30]:

```
# Display plots inline
%matplotlib inline

# Data libraries
import pandas as pd
import numpy as np

# Plotting libraries
import matplotlib.pyplot as plt
import seaborn as sns

# sklearn modules and some other will be added later
import sklearn
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV, KFold
```

[31]:

```
# Plotting defaults for all Figures below
plt.rcParams['figure.figsize'] = (7,5)
plt.rcParams['figure.dpi'] = 80
```

2.2 User Defined Helper Functions

Below are two helper functions we will be using in this workshop. You can create your own if you think it is useful or simply use already available functions within `sklearn`.

- `get_coefs()`: Simple function that extracts both the intercept and coefficients from the model in the pipeline and then concatenates them.
- `model_fit()`: Returns the mean squared error, root mean squared error and R^2 value of a fitted model based on provided X and y values with plotting as add-on.

Feel free to also modify the functions based on your needs.

[32]:

```
def get_coefs(m):
    """Returns the model coefficients from a Scikit-learn model object as an␣
 ↪array,
    includes the intercept if available.
    """

    # If pipeline, use the last step as the model
    if (isinstance(m, sklearn.pipeline.Pipeline)):
        m = m.steps[-1][1]


    if m.intercept_ is None:
        return m.coef_

    return np.concatenate([[m.intercept_], m.coef_])
```

[33]:

```
def model_fit(m, X, y, plot = False):
    """Returns the mean squared error, root mean squared error and R^2 value of␣
↪a fitted model based
    on provided X and y values.

    Args:
        m: sklearn model object
        X: model matrix to use for prediction
        y: outcome vector to use to calculating rmse and residuals
        plot: boolean value, should fit plots be shown
    """

    y_hat = m.predict(X)
    MSE = mean_squared_error(y, y_hat)
    RMSE = np.sqrt(mean_squared_error(y, y_hat))
    Rsqr = r2_score(y, y_hat)

    Metrics = (round(MSE, 4), round(RMSE, 4), round(Rsqr, 4))

    res = pd.DataFrame(
        data = {'y': y, 'y_hat': y_hat, 'resid': y - y_hat}
    )

    if plot:
        plt.figure(figsize=(12, 6))

        plt.subplot(121)
        sns.lineplot(x='y', y='y_hat', color="grey", data =  pd.
↪DataFrame(data={'y': [min(y),max(y)], 'y_hat': [min(y),max(y)]}))
        sns.scatterplot(x='y', y='y_hat', data=res).set_title("Observed vs␣
↪Fitted values")

        plt.subplot(122)
        sns.scatterplot(x='y_hat', y='resid', data=res).set_title("Fitted␣
↪values vs Residuals")
        plt.hlines(y=0, xmin=np.min(y), xmax=np.max(y), linestyles='dashed',␣
↪alpha=0.3, colors="black")

        plt.subplots_adjust(left=0.0)

        plt.suptitle("Model (MSE, RMSE, Rsq) = " + str(Metrics), fontsize=14)
        plt.show()

    return MSE, RMSE, Rsqr
```

2.3 Data

To begin, we will examine `insurance.csv` data set on the medical costs which comes from the Medical Cost Personal dataset. Our goal is to model the yearly medical charges of an individual using some combination of the other features in the data. The included columns are as follows:

- `charges` - yearly medical charges in USD
- `age` - the individuals age
- `sex` - the individuals sex, either `"male"` or `"female"`
- `bmi` - the body mass index of the individual
- `children` - the number of dependent children the individual has
- `smoker` - a factor with levels `"yes"`, the individual is a smoker and `"no"`, the individual is not a smoker

We read the data into python using pandas.

[34]:
```
df_insurance = pd.read_csv("insurance.csv")
df_insurance.head()
```

[34]:
```
   age     sex     bmi  children smoker      charges
0   19  female  27.900         0    yes  16884.92400
1   18    male  33.770         1     no   1725.55230
2   28    male  33.000         3     no   4449.46200
3   33    male  22.705         0     no  21984.47061
4   32    male  28.880         0     no   3866.85520
```

3 Exploratory Data Analysis

Before modelling, we will start with EDA to gain an understanding of the data, through descriptive statistics and visualizations.

3.0.1 Exercise 1 (CORE)

a) Examine the data structure and look at the descriptive statistics. What are the types of variables in the data set?

b) Create a pairs plot of the data (make sure to include the `smoker` column), describe any relationships you observe in the data. To better visualize the relationship between `children` and `charges`, create a violin plot (since `children` only takes a small number of integer values, many points are overlaid in the scatterplot and making visualization difficult).

Hint

- .describe() can be used to create summary descriptive statistics on a pandas dataframe.
- You can use a sns.pairplot and sns.violinplot with the hue argument

[35]:
```
#Part a
print(df_insurance.describe())
print()
print(df_insurance.dtypes)
```

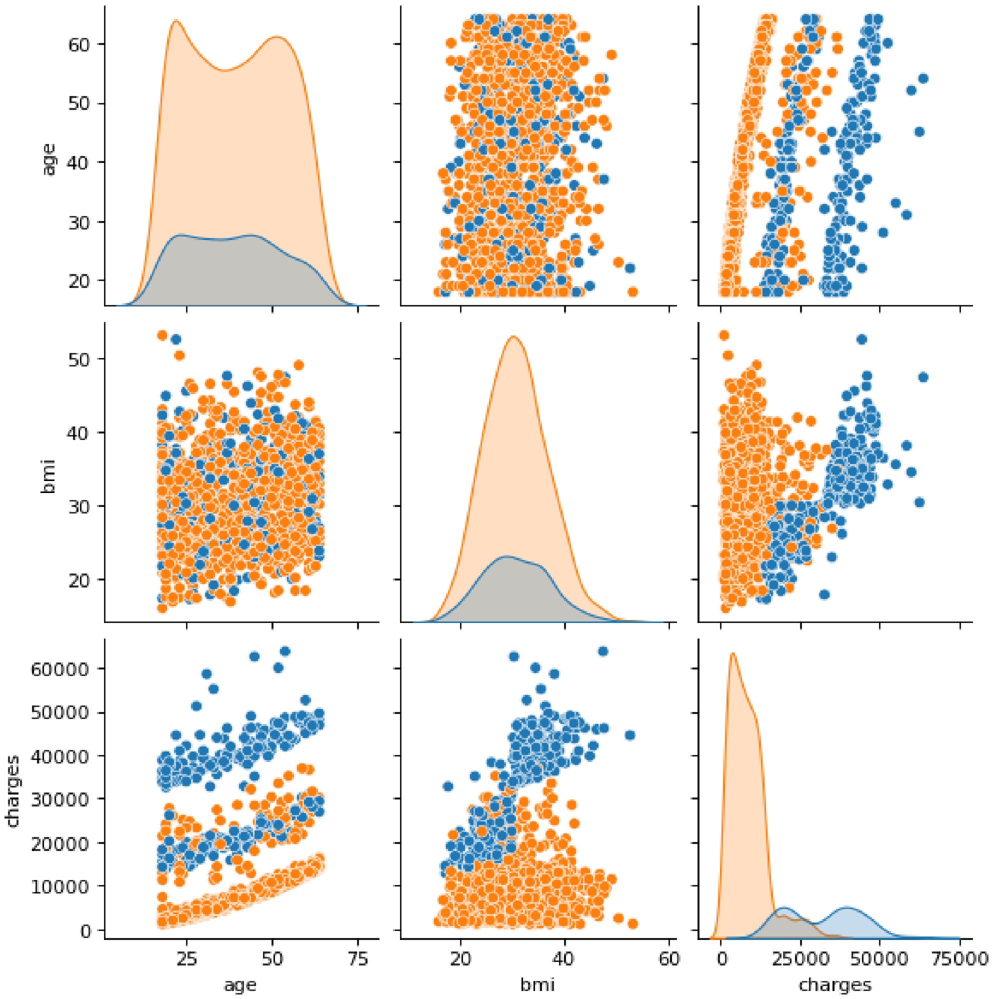
```
               age          bmi     children       charges
count  1338.000000  1338.000000  1338.000000   1338.000000
mean     39.207025    30.663397     1.094918  13270.422265
std      14.049960     6.098187     1.205493  12110.011237
min      18.000000    15.960000     0.000000   1121.873900
25%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

```
age           int64
sex          object
bmi         float64
children      int64
smoker       object
charges     float64
dtype: object
```

Age, BMI, Children, and Charges are all numerical values. Sex, and Smoker take `male, female` and `yes, no` respectively.

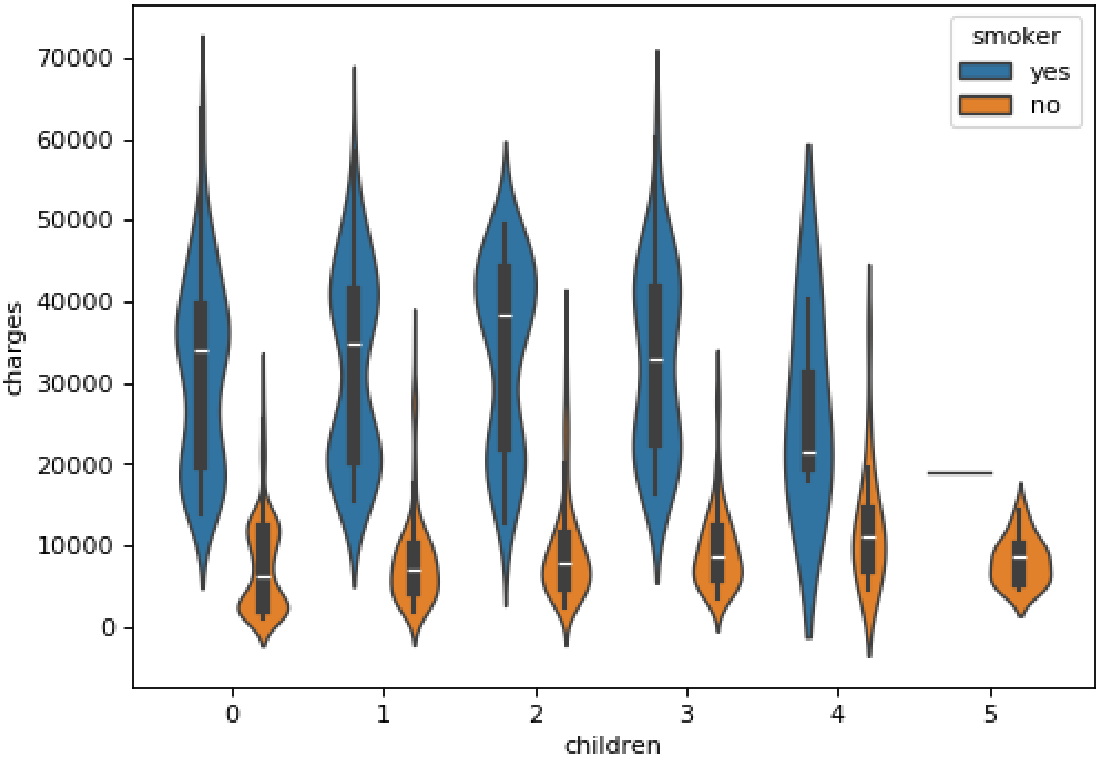
[36]:

```
# Part b
sns.pairplot(df_insurance.drop(columns=['children']), hue='smoker')
plt.show()
```



[37]:
```
sns.violinplot(df_insurance, x=df_insurance['children'],␣
↪y=df_insurance['charges'], hue='smoker')
plt.plot()
```

[37]:
```
[]
```



3.1 Creating a Train-Test Set

Before modelling, we will first split the data into the train and test sets. This ensures that that we do not violate one of the golden rules of machine learning: never use the test set for training. As EDA can help guide the choice and form of model, we also may want to split the data before EDA, to avoid peeking at the test data too much during this phase. However, in practice, we may need to investigate the entire data during EDA to get a better idea on how to handle issues such as missingness, categorical data (and rare categories), incorrect data, etc.

There are lots of ways of creating a test set. We will use a helpful function from `sklearn.model_selection` called `train_test_split`. You can have a look at the documentation: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html. Note that `train_test_split` defaults to randomly sampling the data to split it into training and validation/test sets, that is, the default value is `shuffle=True`

For reproducibility, we first fix the value in the numpy random seed about the state of randomness. This ensures that, every step including randomness, will produce the same output if we re-run the code or if someone else wants to reproduce our results (e.g. produce the same train-test split).

In `sklearn`, the suggestion to control randomness across multiple consecutive executions is as follows:

- In order to obtain reproducible (i.e. constant) results across multiple program executions, we need to remove all uses of `random_state=None`, which is the default.

- Declare your own `rng` variable (random number generator) at the top of the program, and pass it down to any object that accepts a `random_state` parameter. You can check some details from here; https://numpy.org/doc/1.16/reference/generated/numpy.random.RandomState.html

Thus, our first step before splitting the data is to define our `rng` variable.

[38]:
```
# To make this notebook's output identical at every run
rng = np.random.seed(0)
# might be good for our course
# np.random.seed(11205)
```

3.1.1 Exercise 2 (CORE)

Run the following code to use `train_test_split()` to split the data randomly into training (70%) and test (30%) sets. The `training set` will contain our training data, called `X_train` and `y_train`. The `test set` will contain our testing data, called `X_test` and `y_test`.

Can you think on a scenario where not shuffling would be a good idea? What about when we would want to shuffle our data?

[39]:
```
# As a good practice first split the data
from sklearn.model_selection import train_test_split

X = df_insurance.drop('charges', axis = 1) # Set of features
y = df_insurance['charges']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
 ↪random_state = rng)
```

Not shuffling would be a good idea when future datapoints are dependent on past data points. One such example would be language model training. The next 'token' in a sentence has a probability dependent on the previous 'tokens' that have come before it. As such, shuffling would not be an effective method to train.

4 Least Squares Estimation

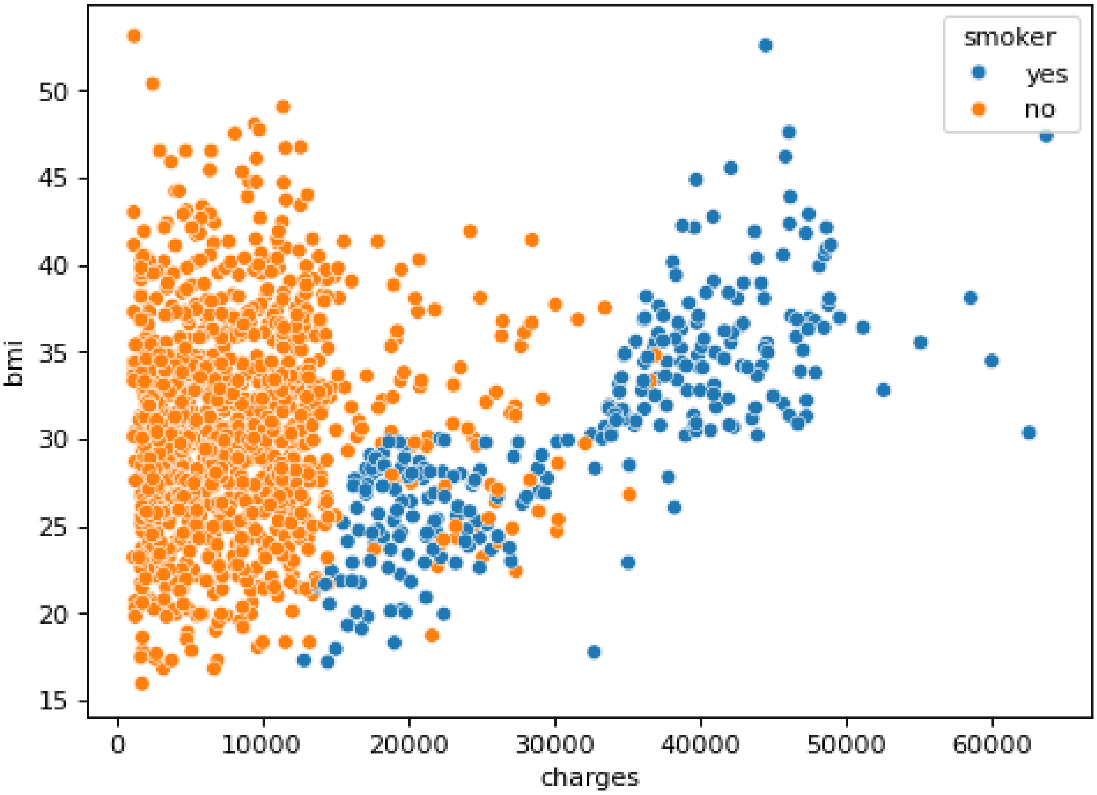
Consider a linear regression model for `charges` using `bmi` and `smoker` as features in our model. Without sklearn functionalities, let's compute and visualize the least squares estimates.

4.0.1 Exercise 3 (CORE)

Create a scatter plot using `sns.scatterplot` of `charges` vs `bmi`, colored by `smoker`. Describe any apparent relationship between `charges` and `bmi` and comment on the difference between smokers and non-smokers.

[40]:
```
# Scatter of charges v bmi, colored by smoker
sns.scatterplot(df_insurance, x='charges', y='bmi', hue='smoker')
```

[40]:
```
<Axes: xlabel='charges', ylabel='bmi'>
```



Generally speaking, smoker have higher charges overall with many having higher BMIs. There is an upward trend which would indicate correlation between the variables.

4.0.2 Exercise 4 (CORE)

Now, let's compute the least square estimates.

a) First construct the design matrix as an `np.array`. Recall that we need to include a column of ones to allow a non-zero intercept. You will also need to convert the response `y_train` to an `np.array`.

b) Compute the least squares estimates $\hat{w}$, using the expression from lectures and the `solve` function from `numpy.linalg`.

c) What is the intercept for non-smokers and what is the intercept for smokers?

Hint

Your design matrix should have three columns, with the last column indicating if the individual is a smoker:

$$x_{n,3} = \begin{cases} 1 & \text{if individual } n \text{ is a smoker} \\ 0 & \text{if individual } n \text{ is not a smoker} \end{cases}$$

You can create this feature in different ways, for example simply using `X_train.smoker == "yes"` (or using `pd.get_dummies` or `OneHotEncoder`).

[41]:
```
# Part a
y_train = y_train.to_numpy()
```

[42]:
```
# create the design matrix
de_mat = np.zeros(shape=(y_train.shape[0], 3))
de_mat[:,0] = 1
de_mat[:, 1] = X_train['bmi']
de_mat[:, 2] = X_train['smoker'].apply(lambda x: 1 if x == 'yes' else 0)
de_mat
```

[42]:
```
array([[ 1.    , 28.215,  0.   ],
       [ 1.    , 32.8  ,  0.   ],
       [ 1.    , 46.75 ,  0.   ],
       …,
       [ 1.    , 25.08 ,  0.   ],
       [ 1.    , 35.53 ,  0.   ],
       [ 1.    , 18.5  ,  0.   ]])
```

[43]:
```
# Part b
from numpy.linalg import solve

# use solve to get the coefficients
betas = solve(de_mat.T @ de_mat, de_mat.T @ y_train)
```

4.0.3 Exercise 5 (CORE)

a) Compute the fitted values from this model by calculating $\hat{\mathbf{y}} = X\hat{w}$.

b) Redraw your scatter plot of `charges`vs `bmi`, colored by `smoker` from Exercise 3, and overlay a line plot of the fitted values (fitted regression line) using `sns.lineplot`. Comment on the results and any potential feature engineering steps that could help to improve the model.

[44]:
```
# part a: compute fitted values
y_hat = de_mat @ betas
```

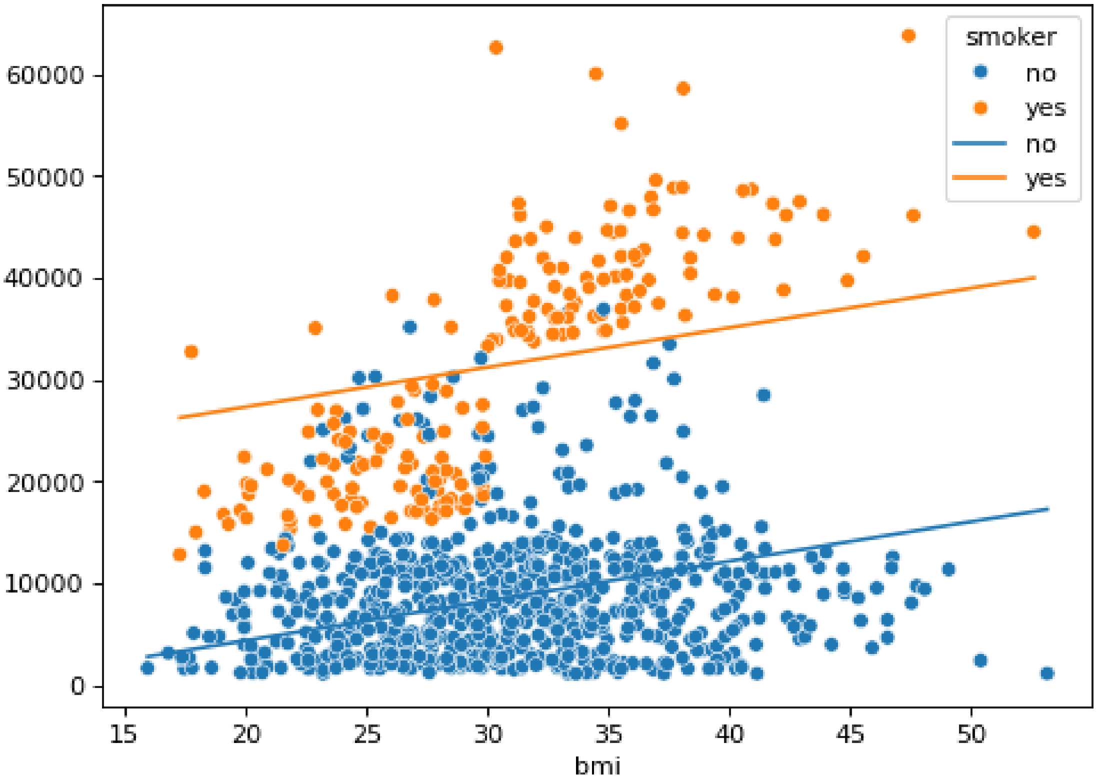
[45]:
```
X_train
```

[45]:
```
      age     sex     bmi  children smoker
1163   18  female  28.215         0     no
196    39  female  32.800         0     no
438    52  female  46.750         5     no
183    44  female  26.410         0     no
1298   33    male  27.455         2     no
…     …       …       …        …
763    27    male  26.030         0     no
835    42    male  35.970         2     no
1216   40    male  25.080         0     no
559    19    male  35.530         0     no
```

```
684    33  female  18.500         1      no

[936 rows x 5 columns]
```

[46]:
```
# part b: redraw scatter plot with regression fitted line.
sns.scatterplot(X_train, x='bmi', y=y_train, hue='smoker')
sns.lineplot(X_train, x='bmi', y=y_hat, hue='smoker')
```

[46]:
```
<Axes: xlabel='bmi'>
```



Some steps to improve the model would be to use more features for regression in the data set.

4.1 Residuals

A useful tool for evaluating a model is to examine the residuals of that model. For any standard regression model, the residual for observation n is defined as $y_n - \hat{y}_n$ where $\hat{y}_n$ is the model's fiited value for observation n.

Studying the properties of the residuals is important for assessing the quality of the fitted regression model. This scatterplot (fitted vs residuals) gives us more intuition about the model performance. Briefly,

- If the normal linear model assumption is true then the residuals should be randomly scattered around zero with no discernible clustering or pattern with respect to the fitted values.

- Furthermore, this plot can be useful to check the constant variance (homoscedastic) assumption to see whether the range of the scatter of points is consistent over the range of fitted values.

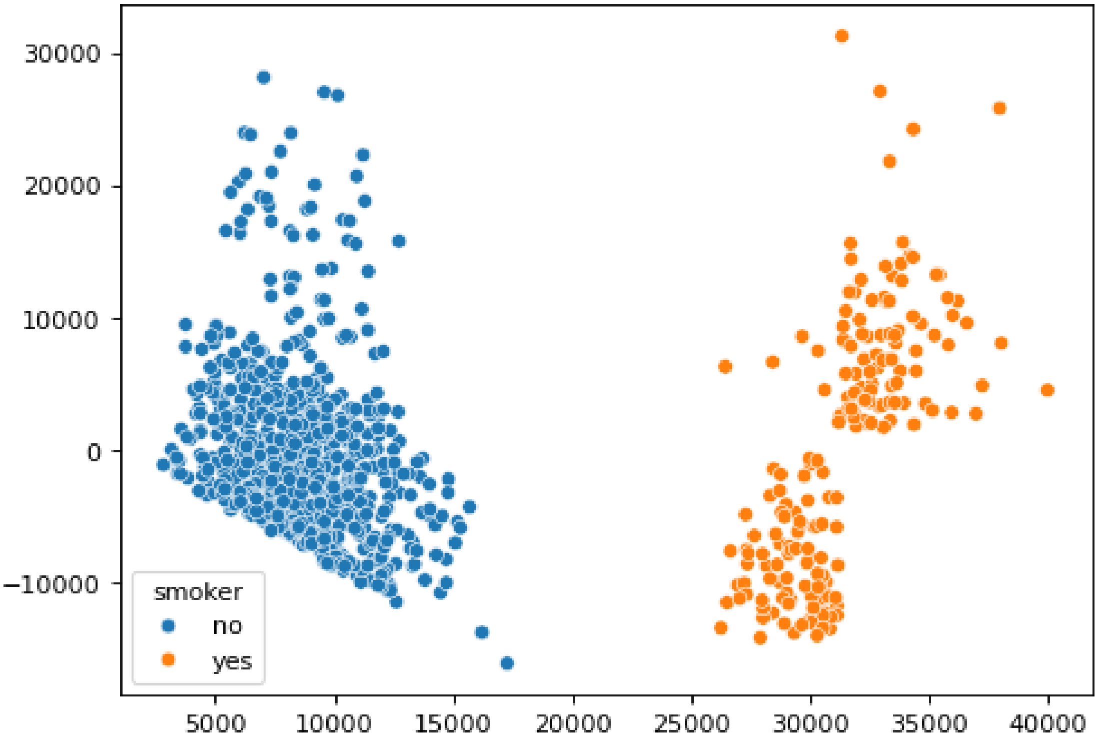
4.1.1 Exercise 6 (CORE)

a) Calculate the residuals and create a residual plot (scatter plot of fitted vs residuals) for this model and color by smoker. Comment on quality of the model based on this plot.

b) Compute the R^2 value for this model and comment on its value (recall from lectures that R^2 is 1 minus the sum of the squared residuals divided by the sum of squared differences between $\mathbf{y}$ and its mean).

[47]:
```
# Part a
res = y_train - y_hat

sns.scatterplot(X_train, x=y_hat, y=res, hue='smoker')
```

[47]:
```
<Axes: >
```



for non-smokers we see reasonable clustering, especialy in the beginning and we see a gradual spread, ideally the data points are all spread out evenly.

As for smokers, the spread it much more inline with what we expect but there is also observable clustering.

[48]:

```
# Part b
R_2 = 1 - np.sum(res ** 2) / np.sum((y_train - np.mean(y_train)) ** 2)
print(R_2)
```

```
0.6350967450118401
```

The R^2 value is 0.6350

4.2 Rank deficiency

4.2.1 Exercise 7 (CORE)

Now lets consider the model where we naively include both dummies variables for smokers and non-smokers as well as an intercept column in our model matrix. What happens when you try to compute the least squares estimate in this case? What is the rank of the design matrix? You can use `numpy.linalg.matrix_rank` to compute the rank.

[49]:

```
# Writing the design matrix
X = np.c_[
    np.ones(len(X_train)),
    X_train.bmi,
    X_train.smoker == "yes",
    X_train.smoker == "no"
]

w = solve(X.T @ X, X.T @ y_train)
```

[50]:

```
# Compute the rank
np.linalg.matrix_rank(X)
```

[50]:

```
3
```

The matrix has rank 3. I.e. there is a column vector that is a linear combination of other column vectors in our matrix

---

5 Regression using scikit-Learn

Linear regression is available in **scikit-learn** (**sklearn**) through `LinearRegression` from the `linear_model` submodule. You can browse through the documentation and examples here. Let's start by importing it.

[51]:

```
from sklearn.linear_model import LinearRegression
```

In general sklearn's models are implemented by first creating a model object, and then using that object to fit your data. As such, we will now create a linear regression model object `lr` and use it to fit our data. Once this object is created we use the `fit` method to obtain a model object fitted to our data.

Note that by default an intercept is included in the model. So, we do NOT need to add a column of ones to our design matrix.

[52]:

```
lr = LinearRegression()

X_train_ = np.c_[
    X_train.bmi,
    X_train.smoker == "yes"
]

lr_fit = lr.fit(
    X = X_train_,
    y = y_train
)
```

This model object then has various useful methods and attributes, including `intercept_` and `coef_` which contain our estimates for w.

Note that if `fit_intercept=False` and a column of ones is included in the design matrix, then both the intercept and coefficient will be stored in `coef_`.

[53]:

```
w_0 = lr_fit.intercept_  # Intercept term of the fitted model
w_1 = lr_fit.coef_

w = np.concatenate([[w_0], w_1])
print(w)

# Or use or helper function to combine the intercept and coefficient into a␣
 ↪single array
get_coefs(lr_fit)
```

```
[-3385.172326     389.05002587 22926.48168453]
```

[53]:

```
array([-3385.172326  ,   389.05002587, 22926.48168453])
```

The model fit objects also provide additional useful methods for evaluating the model R^2 (`score`) and calculating predictions (`predict`). Let's use the latter to compute the fitted values and predictions, as well as some metrics to evaluate the performance on the test data.

[54]:

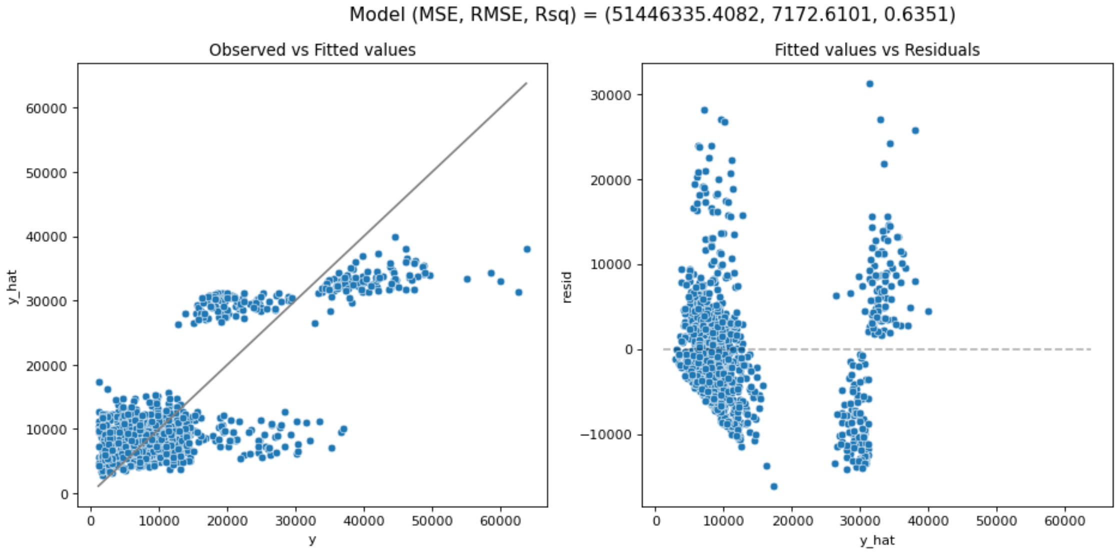
```
# Fitted values
y_fit = lr_fit.predict(X_train_)

# Predicted values
X_test_ = np.c_[
    X_test.bmi,
    X_test.smoker == "yes"
]
```

```
y_pred = lr_fit.predict(X_test_)

# The mean squared error of the training set
print("Training Mean squared error: %.3f" % mean_squared_error(y_train, y_fit))
# The R squared of the training set
print("Training R squared: %.3f" % r2_score(y_train, y_fit))

# The mean squared error of the test set
print("Test Mean squared error: %.3f" % mean_squared_error(y_test, y_pred))
# The R squared of the test set
print("Test R squared: %.3f" % r2_score(y_test, y_pred))

# Another way for R2 calculation
print(lr.score(X_train_, y_train))
```

```
Training Mean squared error: 51446335.408
Training R squared: 0.635
Test Mean squared error: 47297000.691
Test R squared: 0.703
0.6350967450118401
```

[55]:

```
# If we use the pre-defined function
model_fit(lr_fit, X_train_, y_train, plot = True)
```



[55]:

```
(51446335.408195145, 7172.610083379352, 0.6350967450118401)
```

5.0.1 Exercise 8 (CORE)

Next, let's create a pipeline for a regression model, using sklearn functionalities, that

- includies bmi and age as numerical values and smoker condition as a categorical value
- applies encoding for the smoker variable within the pipeline

Note that using the option `OneHotEncoder(drop=np.array(['Reference Category']))`, we can specify the which category to drop (the reference category) by replacing `'Reference Category'` with the desired category.

Fit the model on the training data and calculate performance metrics similar to above (R^2, and MSE / RMSE). How does this model compare to previous one?

[56]:

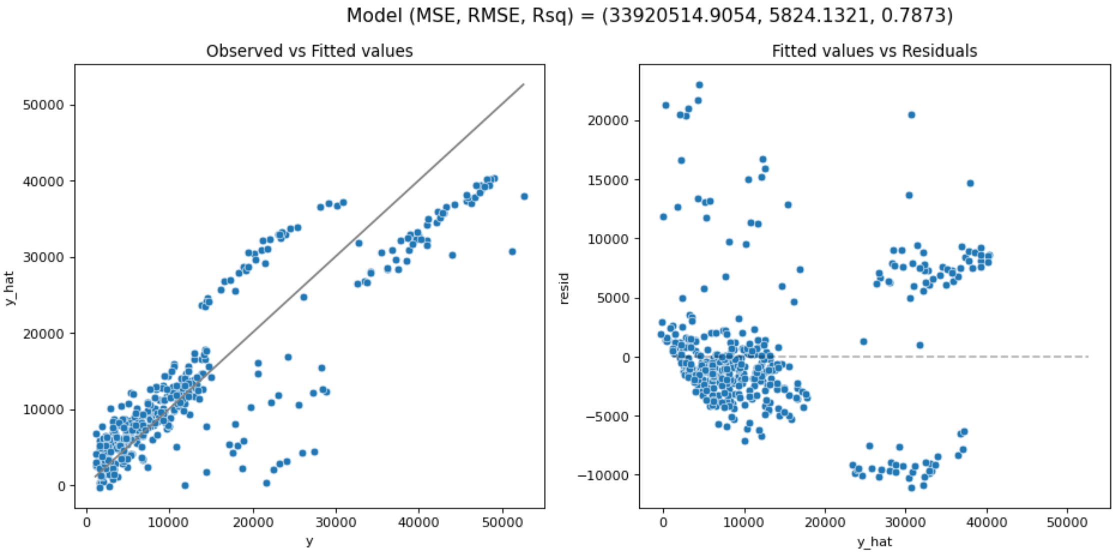
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

cat_pre = OneHotEncoder(drop=np.array(['no']))

# Overall ML pipeline inlcuding all
reg_pipe_1 = Pipeline([
    ("pre_processing", ColumnTransformer([
        ("cat_pre", cat_pre, [4]), # Applied to smoker
        ("num_pre", 'passthrough', [0, 2])])), # Applied to bmi and age
    ("model", LinearRegression())
])


reg_pipe_1
```

[56]:

```
Pipeline(steps=[('pre_processing',
                 ColumnTransformer(transformers=[('cat_pre',
OneHotEncoder(drop=array(['no'], dtype='<U2')),
                                                  [4]),
                                                 ('num_pre', 'passthrough',
                                                  [0, 2])])),
                ('model', LinearRegression())])
```

[59]:

```
# Fit the pipeline with the training data
reg_pipe_1.fit(X_train, y_train)

# Fit the model and calculate performance metrics
model_fit(reg_pipe_1, X_test, y_test, plot = True)
```



```
[59]: (33920514.90536028, 5824.132116063326, 0.7872914737498624)
```

```
[60]: y_pred = reg_pipe_1.predict(X_test)

      # The mean squared error of the training set
      print("Training Mean squared error: %.3f" % mean_squared_error(y_train, y_fit))
      # The R squared of the training set
      print("Training R squared: %.3f" % r2_score(y_train, y_fit))

      # The mean squared error of the test set
      print("Test Mean squared error: %.3f" % mean_squared_error(y_test, y_pred))
      # The R squared of the test set
      print("Test R squared: %.3f" % r2_score(y_test, y_pred))

      # Another way for R2 calculation
      print(reg_pipe_1.score(X_test, y_test))
```

```
Training Mean squared error: 51446335.408
Training R squared: 0.635
Test Mean squared error: 33920514.905
Test R squared: 0.787
0.7872914737498624
```

Our R^2 value has increased from 0.63 to 0.79

6 Polynomial Regression

6.1 Polynomial features in sklearn

sklearn has a built in function called `PolynomialFeatures` which can be used to simplify the process of including polynomial features in a model. You can browse the documentation here. This function is included in the *preprocessing* module of sklearn, as with other python functions we can import it as follows.

[61]:
```
from sklearn.preprocessing import PolynomialFeatures
```

Construction and use of this is similar to what we have already seen with other transformers; we construct a PolynomialFeatures object in which we set basic options (e.g. the degree of the polynomial) and then apply the transformation to our data by calling `fit_transform`. This will generate a new model matrix which includes the polynomial features up to the degree we have specified.

Run the following code for a simple illustration:

[62]:
```
x = np.array([1, 2, 3, 4])
PolynomialFeatures(degree = 2).fit_transform(x.reshape(-1,1))
```

[62]:
```
array([[ 1.,  1.,  1.],
       [ 1.,  2.,  4.],
       [ 1.,  3.,  9.],
       [ 1.,  4., 16.]])
```

Note that when we use this transformation, we get **all of the polynomial transformations of x from 0 to degree**.

In this case, the **0 degree column** is equivalent to **the intercept column**. If we do not want to include this we can construct `PolynomialFeatures` with the option `include_bias=False`.

[63]:
```
PolynomialFeatures(degree = 2, include_bias=False).fit_transform(x.
 ↪reshape(-1,1))
```

[63]:
```
array([[ 1.,  1.],
       [ 2.,  4.],
       [ 3.,  9.],
       [ 4., 16.]])
```

We can also use `PolynomialFeatures` to add only interaction terms, through the option `interaction_only=True`. As an illustration run the following code:

[64]:
```
x = np.array([[1, 2, 3, 4],[0,0,1,1]]).T
PolynomialFeatures(interaction_only=True,include_bias=False).fit_transform(x)
```

[64]:
```
array([[1., 0., 0.],
       [2., 0., 0.],
       [3., 1., 3.],
       [4., 1., 4.]])
```

6.2 Interactions

Now, let's create a pipeline that:

- includes `bmi` as a numerical variable and smoker condition as a categorical variable
- applies encoding for the `smoker` variable
- uses `PolynomialFeatures` to include an **interaction** between `smoker` and `bmi`

[65]:

```
# Create Pipeline for model that includes interactions
cat_pre = OneHotEncoder(drop=np.array(['no']))

pf = PolynomialFeatures(interaction_only=True,include_bias=False)

# Overall ML pipeline
reg_pipe_2 = Pipeline([
    ("pre_processing", ColumnTransformer([
        ("cat_pre", cat_pre, [4]), # Applied to smoker
        ("num_pre", 'passthrough', [2])])), # Applied to bmi
    ("interact", pf),
    ("model", LinearRegression())
])
```

[66]:

```
#Train the model
lr3_fit = reg_pipe_2.fit(X_train,y_train)
```

Note that:

- We have set `include_bias=False` as the intercept is included in linear regression by default
- The returned object is a `Pipeline` object so it will not provide direct access to step properties, such as the coefficients for the regression model.
- If we want access to the attributes or methods of a particular step we need to first access that step using either its name or position.

[67]:

```
print(reg_pipe_2['model'].coef_)
```

```
[-17265.67627046    102.34290762   1318.82587829]
```

[68]:

```
print(reg_pipe_2.steps[2][1].intercept_) # second subset is necessary here␣
 ↪because
                                       # each step is a tuple of a name and the
                                       # model / transform object
```

```
5459.528970943371
```

Alternatively, you can use the `get_coefs` helper function supplied.

We can also extract the **names of the features** using the method `get_feature_names_out()` of the transfomers. Here we need to first extract the names from the first feature engineering step and then pass them to the second step of the pipeline.

[69]:

```
# Extract the names of the features
# From the first step in feature engineering
names_fe1 = reg_pipe_2['pre_processing'].get_feature_names_out()
print(names_fe1)
# From the second step in feature engineering
names_fe2 = reg_pipe_2['interact'].get_feature_names_out(names_fe1)
print(names_fe2)

# or to strip the name of the column transformer
names_fe1 = [names_fe1[i].partition('__')[2] for i in range(len(names_fe1))]
print(names_fe1)
names_fe2 = reg_pipe_2['interact'].get_feature_names_out(names_fe1)
print(names_fe2)
```

```
['cat_pre__smoker_yes' 'num_pre__bmi']
['cat_pre__smoker_yes' 'num_pre__bmi' 'cat_pre__smoker_yes num_pre__bmi']
['smoker_yes', 'bmi']
['smoker_yes' 'bmi' 'smoker_yes bmi']
```

6.2.1 Exercise 9 (CORE)

For the model trained above (`lr3_fit`):

a) What is the intercept and slope for non-smokers and what is the intercept and slope for smokers?

b) Compute the fitted values by calling `predict`. Draw the scatter plot of bmi against charges, colored by smoker (from Exercise 3), and overlay a line plot of the fitted values (fitted regression lines).

c) How does this model compare to the previous ones?

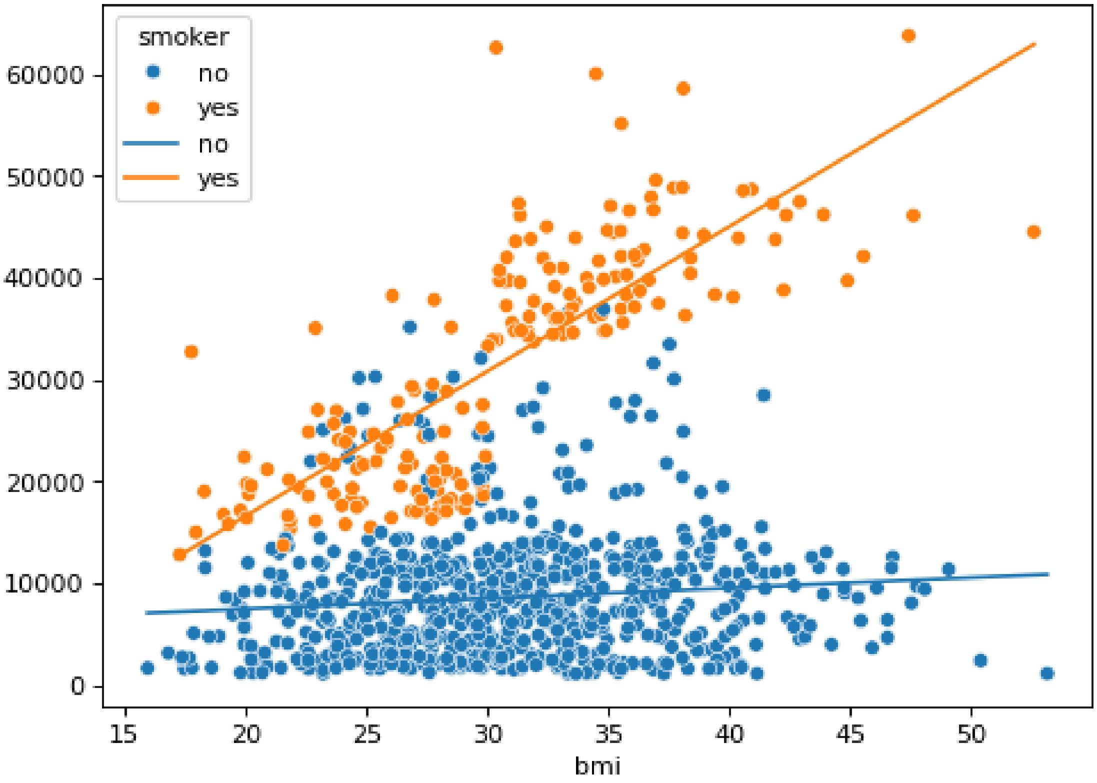
[72]:

```
#Part a
# what is th intercept and slope for non-smokers and smokers
# The intercept and slope for non-smokers
print("The intercept and slope for non-smokers")
print(reg_pipe_2['model'].intercept_)
print(reg_pipe_2['model'].coef_)
# The intercept and slope for smokers
print("The intercept and slope for smokers")
print(reg_pipe_2['model'].intercept_ + reg_pipe_2['model'].coef_[0])
print(reg_pipe_2['model'].coef_)
```

```
The intercept and slope for non-smokers
5459.528970943371
[-17265.67627046    102.34290762   1318.82587829]
The intercept and slope for smokers
-11806.147299512158
[-17265.67627046    102.34290762   1318.82587829]
```

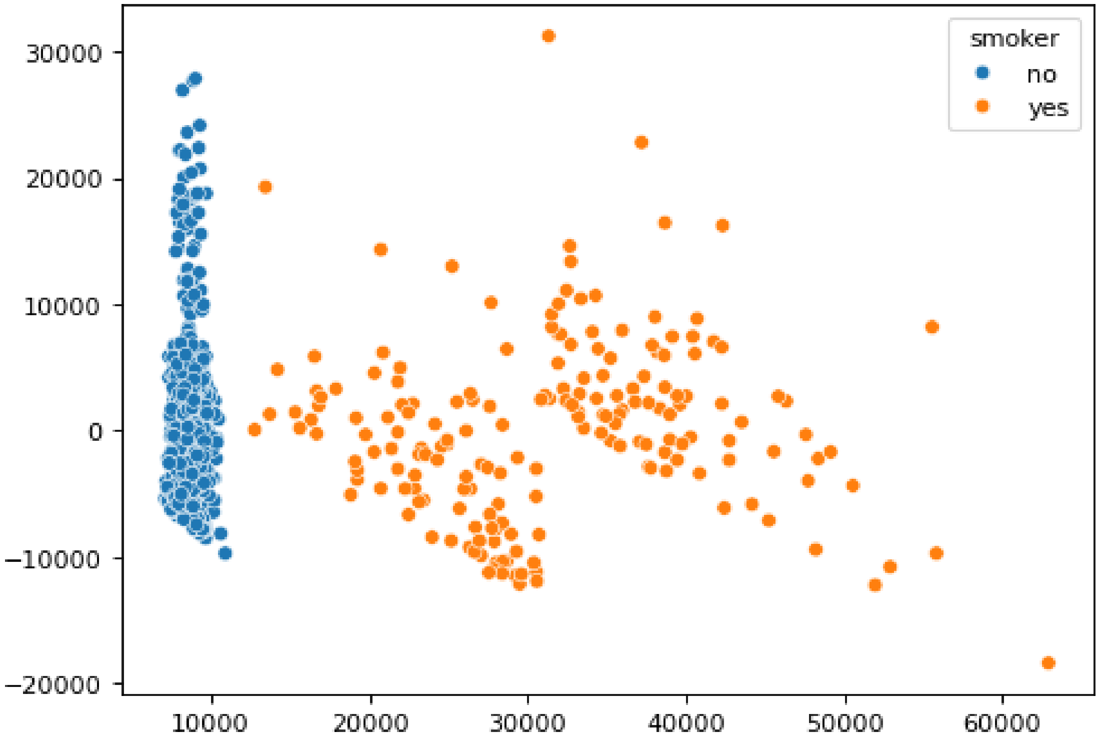
```
[ ]: #Part b
     # compute fitted values by calling predict() and draw scatter plot of bmi␣
      ↪against charges colored by smoker. overlay a line plot of the iftted values
     y_fit = reg_pipe_2.predict(X_train)
     sns.scatterplot(X_train, x='bmi', y=y_train, hue='smoker')
     sns.lineplot(X_train, x='bmi', y=y_fit, hue='smoker')
```

```
[ ]: <Axes: xlabel='bmi'>
```



```
[ ]: # Part c
     # compute residuals and draw scatter plot of fitted values against residuals␣
      ↪colored by smoker
     res = y_train - y_fit
     sns.scatterplot(X_train, x=y_fit, y=res, hue='smoker')
```

```
[ ]: <Axes: >
```



I feel like this residual plot is much more in-line with what we expect to see. A lot of random scattering about 0 and the variance in the data points is largely constant meeting the homodasticity assumption.

6.3 Nonlinearity

Now, let's explore including nonlinearity into the model through a polynomial basis function expansions of `bmi`.

6.3.1 Exercise 10 (CORE)

First, suppose you naively apply a polynomial basis function expansion to the feature matrix containing `bmi` and `smoker`:

- Run the code below to first create the feature matrix (we standardize `bmi` to simply help with visualization).
- Next, run the code to create the transformed feature matrix using `PolynomialFeatures` assuming `degree=2`. Print out the first 20 rows of this matrix.
- What are the number of features and what does each column represent? Why should we **NOT** use this naive polynomial basis function expansion?

[100]:
```
# Create a matrix containing bmi and smoker (note we standardize bmi here,␣
↪simply to help avoid
# printing very large numbers in the exercise)

from sklearn.preprocessing import StandardScaler

bmi_ss = StandardScaler().fit_transform(np.asarray(X_train.bmi).reshape(-1,1))

X_ = np.c_[
    X_train.smoker == "yes",
    bmi_ss
]
```

[ ]:

```
pf = PolynomialFeatures(degree=2)
X_mat = pf.fit_transform(X_)
print(np.round(X_mat[1:20,],2))
```

[ ]:

```
print(pf.get_feature_names_out(['bmi','smoker']))
```

6.3.2 Exercise 11 (CORE)

Now, let's create a model that allows nonlinearity of `bmi` through polynomial basis function expansion of degree 3 (with no interactions for ease of exposition).

Create a new pipeline to construct this model. Train this model and then plot the fitted regression line and compute the performance metrics.

[83]:

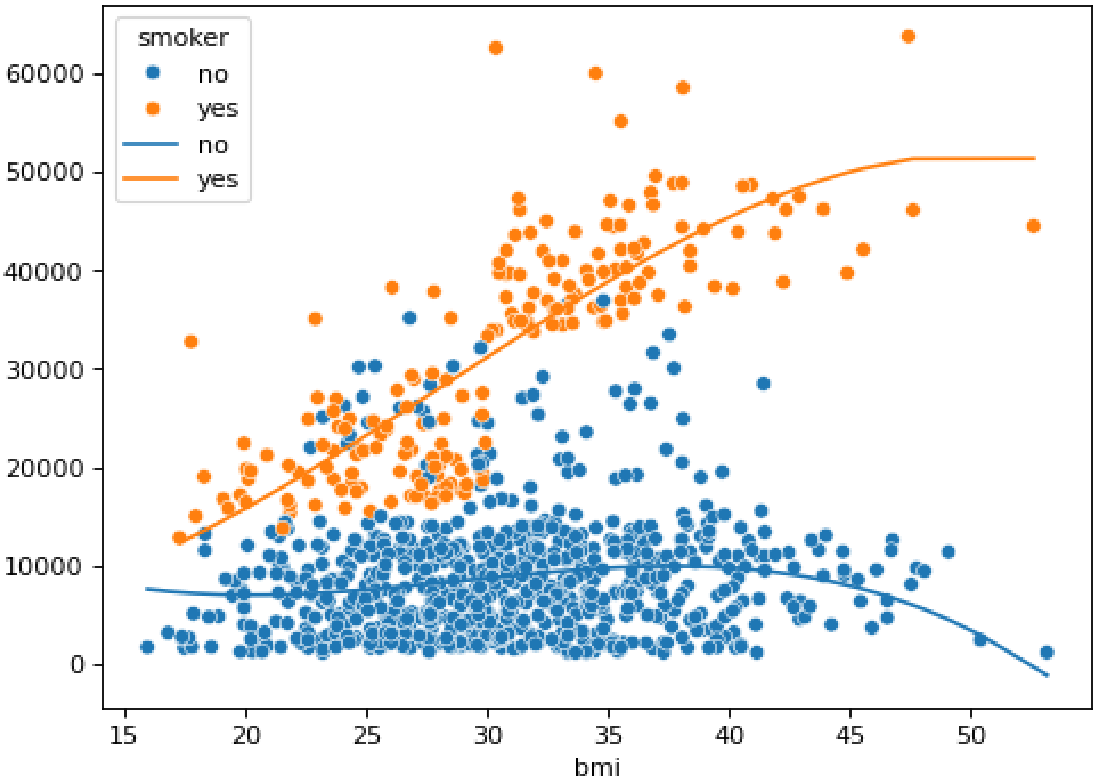
```
# Create Pipeline for model that includes polynomial expansions of bmi
cat_pre = OneHotEncoder(drop=np.array(['no']))

pf = PolynomialFeatures(degree=3, include_bias=False)

# Overall ML pipeline
reg_pipe_3 = Pipeline([
    ("pre_processing", ColumnTransformer([
        ("cat_pre", cat_pre, [4]), # Applied to smoker
        ("num_pre", 'passthrough', [2])])), # Applied to bmi
    ("poly", pf),
    ("model", LinearRegression())
])
```

[84]:

```
# Train and compute and plot fitted values
reg_pipe_3.fit(X_train, y_train)
y_fit = reg_pipe_3.predict(X_train)
sns.scatterplot(X_train, x='bmi', y=y_train, hue='smoker')
sns.lineplot(X_train, x='bmi', y=y_fit, hue='smoker')
```

[84]:

```
<Axes: xlabel='bmi'>
```



6.4 Choosing the Order of the Polynomial

How can we choose the order of the polynomial?

In lecture, we discussed how chosing the degree to be too large can cause over fittting. When we over fit a polynomial regression model, the MSE for the training data will appear to be low which might indicate that the model is a good fit. But, as a result of over fitting, the MSE for the predictions of the unseen test data may begin to increase. However, we can **NOT** use the test to determine the order of the polynomial, so in the following, we explore using cross-validation to do so.

6.4.1 Exercise 12 (EXTRA)

First, let's compute and plot the **training and test MSE** over a range of degree values. What do you notice about the fit as we increase the polynomial degree? Which degree seems better regarding the changes on training and testing MSE values?

[108]:

```
degree = range(1,10)
test_mse = np.array([])
train_mse = np.array([])

for d in degree:
    cat_pre = OneHotEncoder(drop=np.array(['no']))
    pf = PolynomialFeatures(degree=d,include_bias=False)

    # Overall ML pipeline
    p = Pipeline([
        ("pre_processing", ColumnTransformer([
            ("cat_pre", cat_pre, [4]), # Applied to smoker
            ("poly", pf, [2])])),
        ("model", LinearRegression())])

    m = p.fit(X_train,y_train)

    yhat = m.predict(X_train)
    ypred = m.predict(X_test)

    test_mse = np.append(test_mse, mean_squared_error(y_test, ypred))
    train_mse = np.append(train_mse, mean_squared_error(y_train, yhat))
```

[ ]:

```
fig, ax = plt.subplots(figsize=(9,7), ncols=1, nrows=1)
plt.scatter(degree,train_mse, color='k')
plt.plot(degree,train_mse, color='k', label='Train MSE')
plt.scatter(degree,test_mse, color='r')
plt.plot(degree,test_mse, color='r', label='Test MSE')
ax.legend()
ax.set_xlabel('degree')
ax.set_ylabel('MSE')
plt.show()
```

6.4.2 Tunning with GridSearchCV

If we wish to test over a specific set of parameter values using cross validation we can use the `GridSearchCV` function from the `model_selection` submodule. In this setting, the hyperparamer is actually the degree of the polynomial that we are investigating.

This argument is a dictionary containing parameters names as keys and lists of parameter settings to try as values. Since we are using a pipeline, our parameter name will be the name of the pipeline step, `pre_processing`, followed by `__`, (then, the name of next step if applicable, e.g. `poly__` since we are using `ColumnTransformer`), and then the parameter name, `degree`. So for our pipeline the parameter is named `pre_processing_poly__degree`. If you want to list the names of all available parameters you can call the `get_params()` method on the model object, e.g. `polyreg_pipe.get_params()` here.

[110]:

```
cat_pre = OneHotEncoder(drop=np.array(['no']))

pf = PolynomialFeatures(include_bias=False)

# Overall ML pipeline
polyreg_pipe = Pipeline([
    ("pre_processing", ColumnTransformer([
        ("cat_pre", cat_pre, [4]), # Applied to smoker
        ("poly", pf, [2])])), # Applied to bmi
    ("model", LinearRegression())])

parameters = {
    'pre_processing__poly__degree': np.arange(1,10,1)
}

kf = KFold(n_splits = 5, shuffle = True, random_state=rng)

grid_search = GridSearchCV(polyreg_pipe, parameters, cv = kf, scoring = 'neg_mean_squared_error', return_train_score=True).fit(X_train, y_train)
```

[112]:

```
#polyreg_pipe.get_params()
```

The above code goes through the process of fitting all 5 × 9 models as well as storing and ranking the results for the requested scoring metric(s). Note that here we have used `neg_mean_squared_error` as our scoring metric which returns the **negative of the mean squared error**. For more on metrics of regression models, please see: https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metrics

- As the name implies this returns the negative of the usual fit metric, this is because sklearn expects to always optimize for the maximum of a score and the model with the largest negative MSE will therefore be the "best".
- In this workshop we have used MSE as a metric for testing our models. This metric is entirely equivalent to the root mean squared error for purposes of ranking / ordering models (as the square root is a monotonic transformation).
- Sometimes the RMSE is prefered as it is more interpretable, because it has the same units as y.

Once all of the submodels are fit, we can determine the optimal hyperparameter value by accessing the object's `best_*` attributes,

[ ]:

```
print("best index: ", grid_search.best_index_)
print("best param: ", grid_search.best_params_)
print("best score: ", grid_search.best_score_)
```

The best estimator is stored in the `.best_estimator` attribute. By default, after this model is found, it is retrained on all training data points.

[ ]:

```
grid_search.best_estimator_['model'].coef_
```

[ ]:

```
# Extract the names of the features
names_fe = grid_search.best_estimator_["pre_processing"].get_feature_names_out()
# Strip the name of the column transformer
names_fe = [names_fe[i].partition('__')[2] for i in range(len(names_fe))]
w = grid_search.best_estimator_['model'].coef_
```

```
print(np.c_[names_fe,w])
```

[ ]:

```
# Compute fitted values
yhat =grid_search.predict(X_train)

# Plot fitted values
ax = sns.scatterplot(x = X_train.bmi, y = y_train, hue = X_train.smoker)
sns.lineplot(x = X_train.bmi, y = yhat, hue = X_train.smoker, ax=ax, legend =␣
  ↪False)
ax.set(ylabel='charges')
plt.show()
```

The cross-validated scores are stored in the attribute `cv_results_`. This contains a number of results related to the grid search and cross-validation. We can convert it into a pandas data frame to view the results.

[ ]:

```
cv_results = pd.DataFrame(grid_search.cv_results_)
cv_results
```

It is also recommend to plot the CV scores. Although the grid search may report a best value for the parameter corresponding to the maximum CV score (e.g. min CV MSE), if the curve is relatively flat around the minimum, we may prefer the simpler model.

Note in this case, I have also used the option `return_train_score=True` in `GridSearchCV()`, in order to save also the training scores. As expected training MSE decreases when increasing the degree of the polynomial, but the CV MSE has more of a U-shape.

[ ]:

```
degree = np.arange(1,10,1)
fig, ax = plt.subplots(figsize=(9,7), ncols=1, nrows=1)
plt.scatter(degree,-grid_search.cv_results_['mean_train_score'], color='k')
plt.plot(degree,-grid_search.cv_results_['mean_train_score'], color='k',␣
  ↪label='Mean Train MSE')
plt.scatter(degree,-grid_search.cv_results_['mean_test_score'], color='r')
plt.plot(degree,-grid_search.cv_results_['mean_test_score'], color='r',␣
  ↪label='CV MSE')
ax.legend()
ax.set_xlabel('degree')
ax.set_ylabel('MSE')
plt.show()
```

6.4.3 Exercise 13 (CORE)

Based on the plot above, would you use the best estimator or choose a different degree? Why?

6.4.4 Exercise 14 (EXTRA)

Try an alternative model of your choice. What have you chosen and why? Are there any parameters to tune?

[130]:

```
 
```

6.4.5 Further resources

- About common pitfalls and interpreting coefficients:

https://scikit-learn.org/stable/auto_examples/inspection/plot_linear_model_coefficient_interpretation.html#

7 Competing the Worksheet

At this point you have hopefully been able to complete all the CORE exercises and attempted the EXTRA ones. Now is a good time to check the reproducibility of this document by restarting the notebook's kernel and rerunning all cells in order.

Before generating the PDF, please go to Edit -> Edit Notebook Metadata and change 'Student 1' and 'Student 2' in the **name** attribute to include your name.

Once that is done and you are happy with everything, you can then run the following cell to generate your PDF. Once generated, please submit this PDF on Learn page by 16:00 PM on the Friday of the week the workshop was given.

[ ]:

```
!jupyter nbconvert --to pdf mlp_week04.ipynb
```