

# MLPy Workshop 4: Solutions

Sara Wade, Erik Hormann

January 11, 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 will need for this workshop.

```
[1]: # 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

[2]: # 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.

```
[3]: 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_])
```

```
[4]: 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), linestyle='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.

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

```
[5]:   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)

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

```
[6]: #Part a

# For general info
df_insurance.info()
```

```
# For the summary statistics
df_insurance.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 6 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   charges     1338 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 62.8+ KB
```

```
[6]:
```

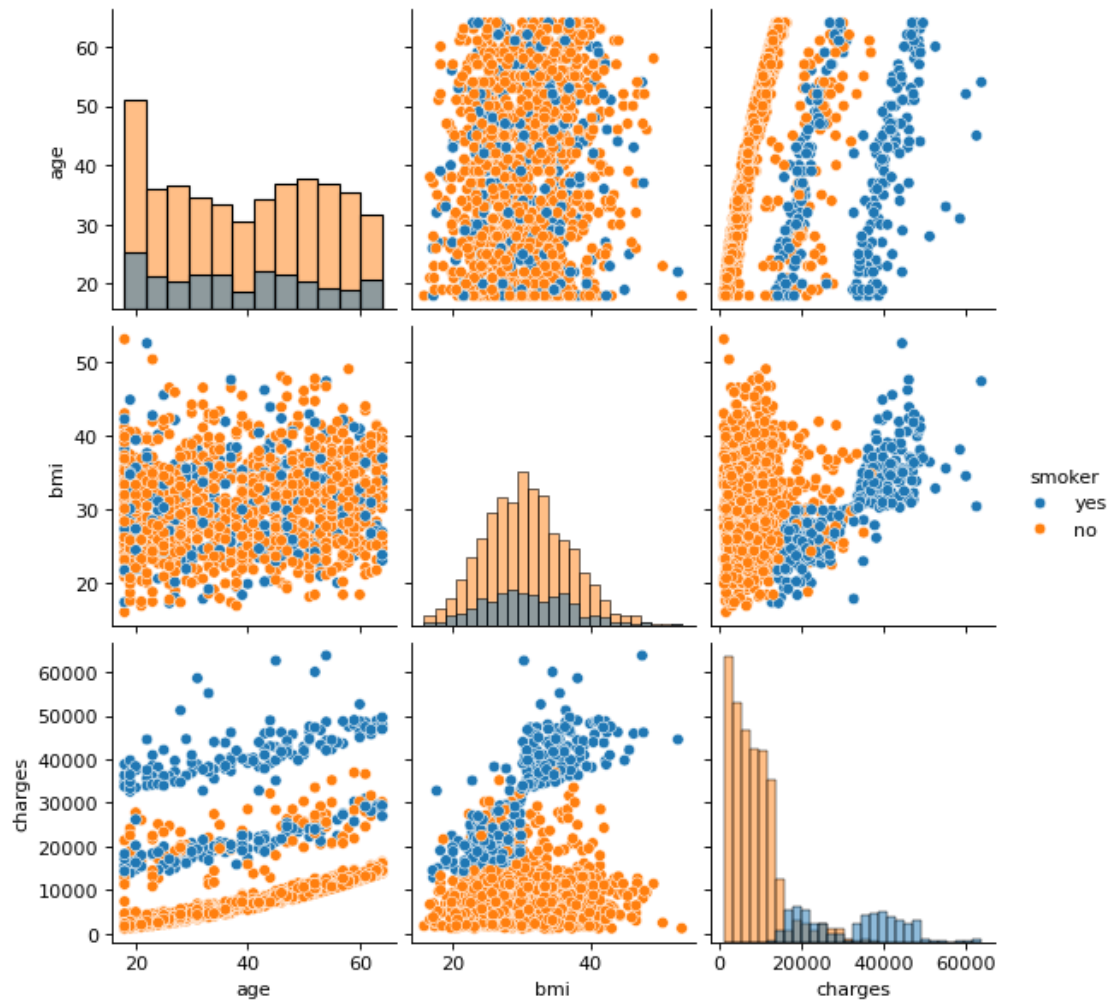
	age	bmi	children	charges
count	1338.000000	1338.000000	1338.000000	1338.000000
mean	39.207025	30.663397	1.094918	13270.422265
std	14.049960	6.098187	1.205493	12110.011237
min	18.000000	15.960000	0.000000	1121.873900
25%	27.000000	26.296250	0.000000	4740.287150
50%	39.000000	30.400000	1.000000	9382.033000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

The target variable **charges** is numeric, with both a large mean and scale; **bmi** is also a numeric variable, while **age** and **children** are both integers. There are two binary categorical variables that we will need to encode, which are **sex** and **smoker**.

```
[7]: # Part b

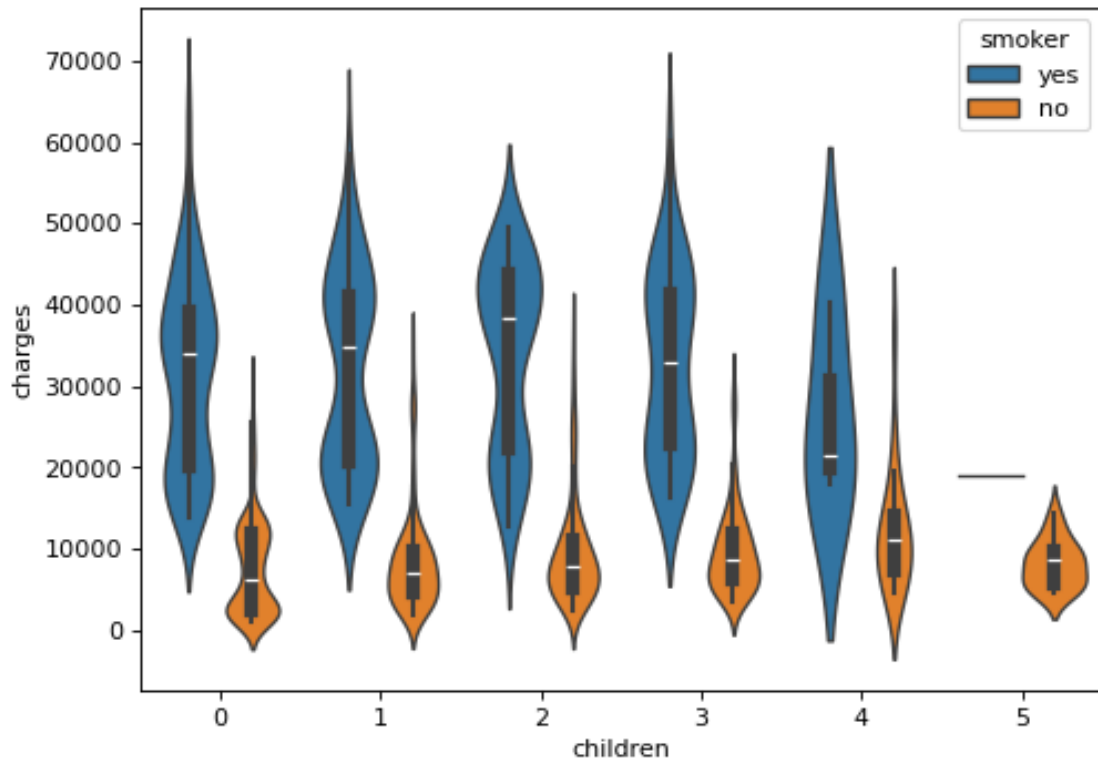
# # Plotting the data set
# sns.pairplot(data = df_insurance, hue = "smoker", diag_kind='hist')
# plt.show()

# Many points are plotted on top of each other for children, let's remove it
# from the pairplot
sns.pairplot(data = df_insurance, hue = "smoker", diag_kind='hist',
             vars=['age', 'bmi', 'charges'])
plt.show()
```



```
[8]: # Violin plot for discrete variables
sns.violinplot(data=df_insurance, x='children', y='charges', hue='smoker')
plt.show()

# # Or a strip plot
# sns.stripplot(data=df_insurance, x='children', y='charges', hue='smoker')
# plt.show()
```



From the pairs plot, we observe that:

- **bmi** and **charges** as well as **age** and **charges** pairs seem to exhibit possible linear relationships.
- There is a distinct difference in the **charges** between smokers and non smokers, with overall smaller charges for non-smokers.
- Our response variable **charges** seems to be skewed.
- There does not appear to be an evident relationship between **children** and **charges**.

### 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](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.

```
[9]: # 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?

```
[10]: # 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)
```

- In lots of real world ML applications you train on historical data, and make predictions on future unseen data. If this was the case we may want our data sorted by timestamp before creating the splits, since it matches the way you'll apply the model in the real world.
- However, in this case where we do not have a timestamp, we probably want to shuffle our data first because the ordering may effect our assessment of generalisation performance.
- The ordering of the data depends on where it came from and how it was exported and it's not uncommon that real world data is sorted in some manner.
- Therefore if there is an order according to one of the attributes, we could have a very poor test set.



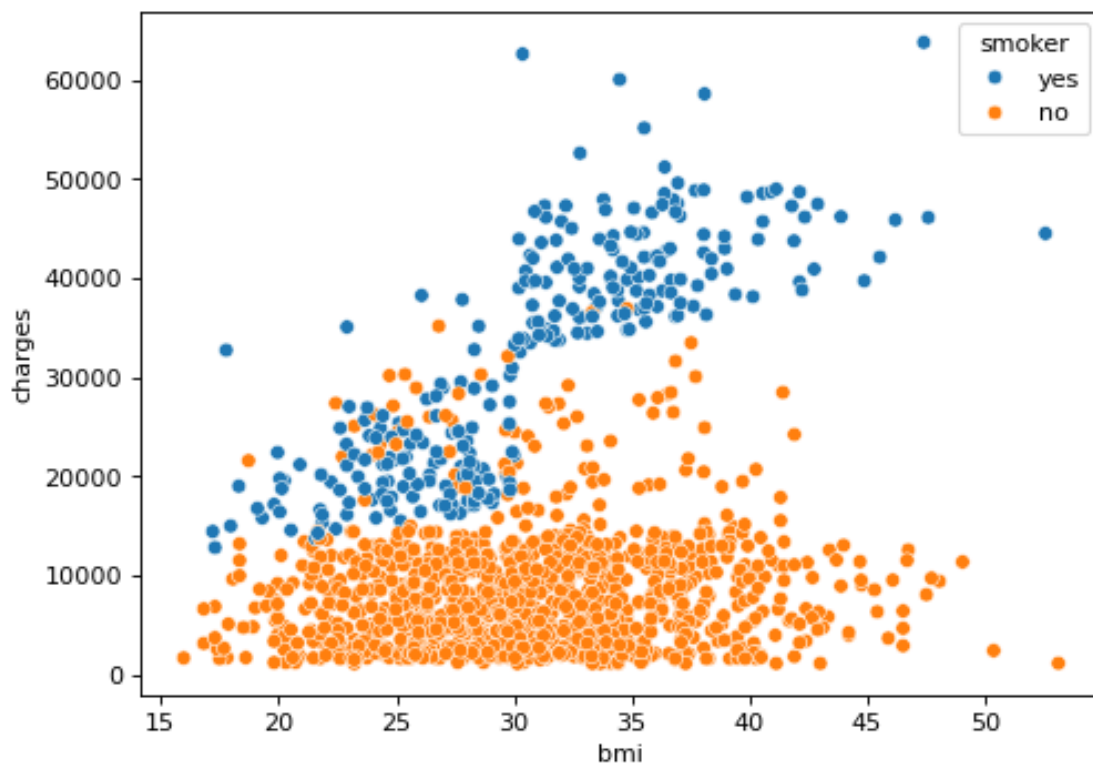
## 4 Least Squares Estimation

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.

```
[11]: # Scatter of charges v bmi, colored by smoker
sns.scatterplot(x="bmi", y="charges", hue="smoker", data = df_insurance)
plt.show()
```



- There does appear to be a relationship between `bmi` and `charges`. A linear relationship may be a good baseline model to start with.
- There is an apparent difference in the smokers and non-smokers, and scatterplot suggest that different slopes may also be needed within each class.

### 4.0.2 Exercise 4 (CORE)

Now, let's compute the least square estimates.

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

```
[12]: # Part a

# Writing the design matrix
X = np.c_[
    np.ones(len(X_train)),
    X_train.bmi,
    X_train.smoker == "yes"
]

# Converting response training object to numpy array
y = np.array(y_train)

X.shape, y.shape
```

```
[12]: ((936, 3), (936,))
```

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

w = solve(X.T @ X, X.T @ y)

print(w)
```

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

```
[14]: # Part c

print("The intercept for non-smokers is: %.3f" % w[0])

print("The intercept for smokers is: %.3f" % (w[0]+w[2]))
```

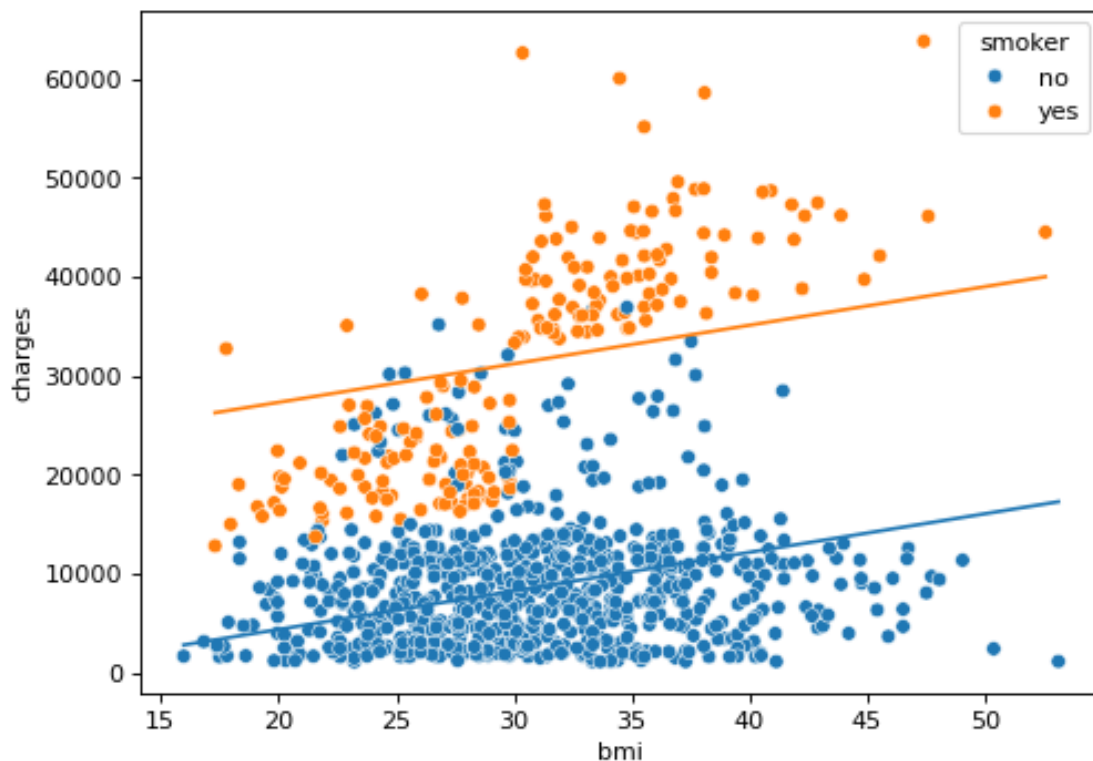
The intercept for non-smokers is: -3385.172  
The intercept for smokers is: 19541.309

#### 4.0.3 Exercise 5 (CORE)

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

```
[15]: # Part a: Compute fitted values
yhat = X @ w

# Part b: Plot fitted values
ax = sns.scatterplot(x = X_train.bmi, y = y, 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()
```



There is an evident increase in predicted medical charges for smokers and an increase with higher bmi. This is a sensible baseline model, but it could be improved, for example by including an

interaction between smoker and bmi (visually, it appears the slope may be steep for smokers) and possibly also nonlinear terms.

## 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 fitted value for observation  $n$ .

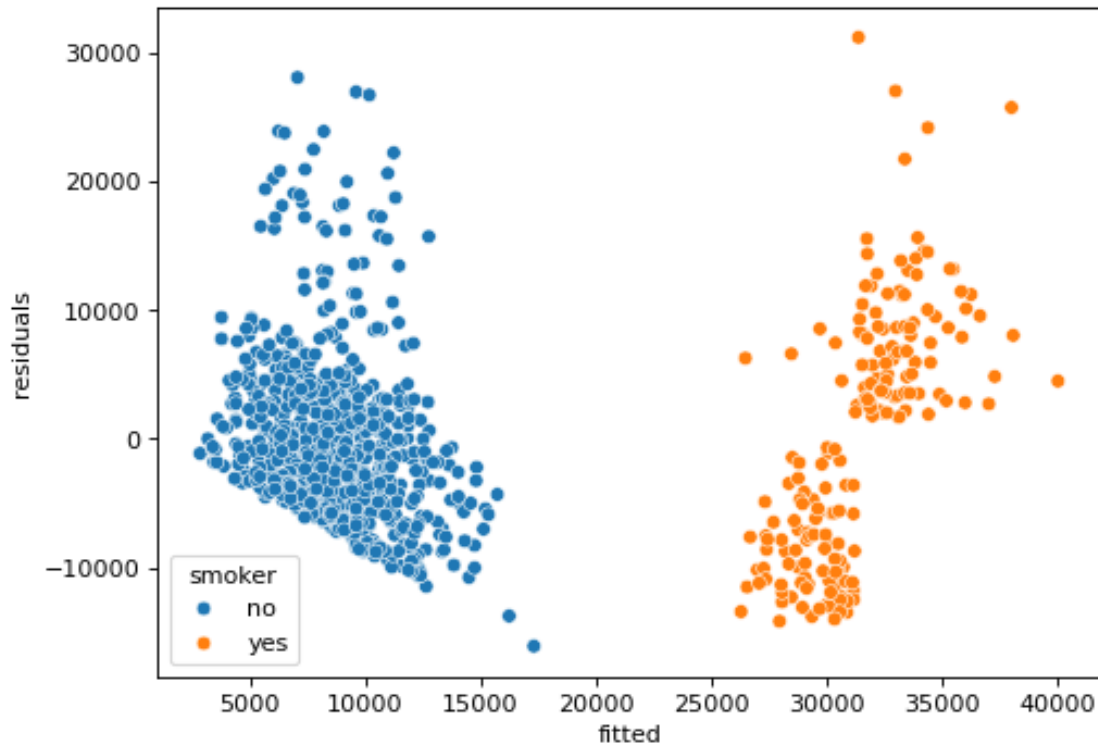
Studying the properties of the residuals is important for assessing the quality of the fitted regression model. This scatterplot (fitted vs residuals) gives us more intuition about the model performance. Briefly,

- 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  $y$  and its mean).

```
[16]: # Part a
resid = y - yhat
ax = sns.scatterplot(x = yhat, y = resid, hue = X_train.smoker)
ax.set(xlabel = 'fitted', ylabel='residuals')
plt.show()
```



Within each group, the points do not seem to be randomly scattered around zero and there appears to be a pattern. In particular for non-smokers, the residuals tend to be greater than zero for smaller fitted values (underestimation) and the residuals tend to be smaller than zero for larger fitted values (overestimation). The converse appears to be true for smokers. Since the model assumes a common slope for both smokers and non-smokers, the behavior across the two groups is averaged. Allowing different slopes for smokers and non-smokers may help address this.

```
[17]: # Part b
R_2 = 1 - sum(resid**2)/sum((y - np.mean(y))**2)
print(R_2)
```

```
0.6350967450118411
```

The  $R^2$  value is fairly good. The model explains about 63.5% of the variability of the response variable around its mean.

## 4.2 Rank deficiency

### 4.2.1 Exercise 7 (CORE)

Now let's 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.

```
[18]: # 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)
```

```
[19]: np.linalg.matrix_rank(X)
```

```
[19]: 3
```

We get an error when trying to compute this least squares estimate. This is because our design matrix is rank deficient, as the columns are linearly dependent: the first column (of ones) is equal to the sum of the third column indicating smokers and the fourth column indicating non-smokers. In this case, the rank of the design matrix is 3 not 4. To address this, we should drop one of the features in one-hot encoding (i.e. non-smoker), which is referred to as the reference category, as the intercept represents the intercept for the reference category.

---

## 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.

```
[20]: 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.

```
[21]: 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_`.

```

[22]: 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]

```

```

[22]: 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.

```

[23]: # 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))

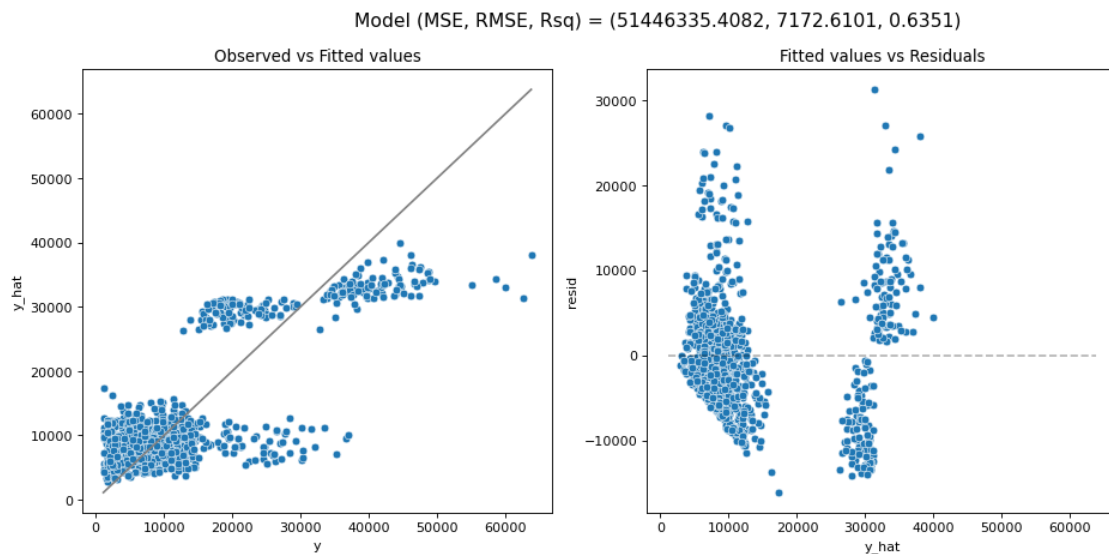
      # Another way for R2 calculation

```

```
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

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



```
[24]: (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

- includes `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?

```
[25]: 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 including 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

```

```

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

```

```

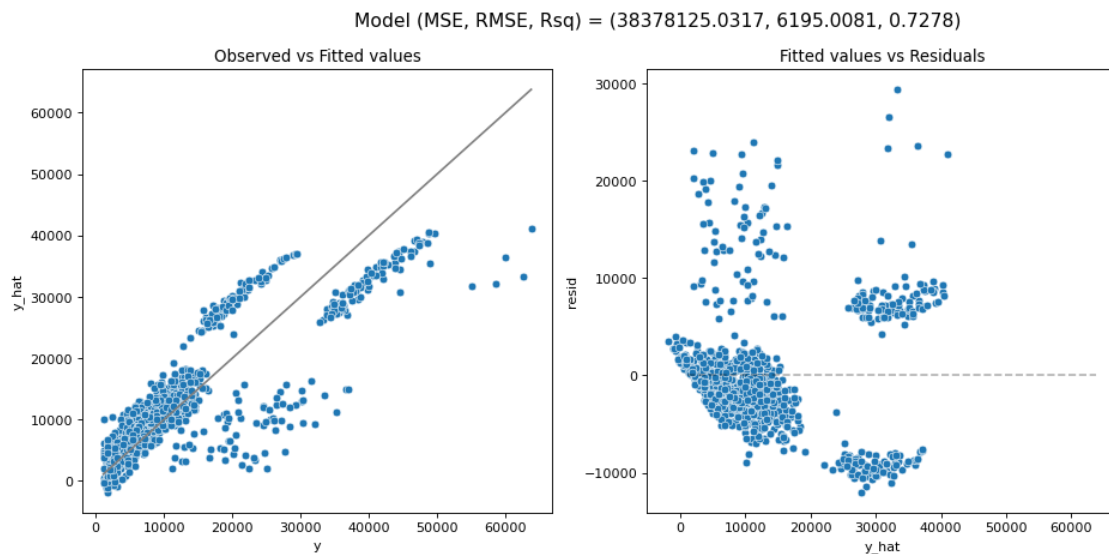
[26]: # Fit the model
lr2_fit = reg_pipe_1.fit(X_train, y_train)

```

```

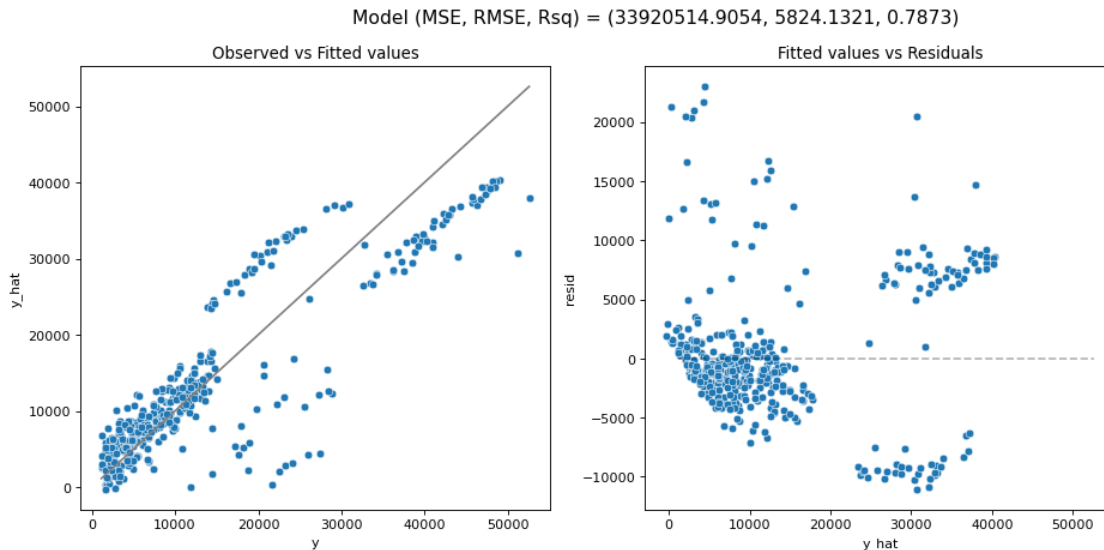
[27]: # Using the model_fit function on training data
model_fit(lr2_fit, X_train, y_train, plot = True)

```



[27]: (38378125.031708114, 6195.008073578929, 0.7277881381968744)

```
[28]: # Using the model_fit function on test data
model_fit(lr2_fit, X_test, y_test, plot = True)
```



[28]: (33920514.90536016, 5824.132116063316, 0.7872914737498631)

Including age helps to improve the model, providing better performance metrics. As we have included an additional feature `age`, the training MSE/RMSE is lower (higher R2) as expected, but the test MSE/RMSE is also lower (higher R2).

We do still notice some patterns in the residuals. For example, the evident under and over estimation for smokers with low and high bmi.

---

## 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.

```
[29]: 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:

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

```
[30]: 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`.

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

```
[31]: 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:

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

```
[32]: 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`

```
[33]: # 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())
])

```

```

[34]: #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.

```

[35]: print(reg_pipe_2['model'].coef_)

```

```

[-17265.67627046    102.34290762   1318.82587829]

```

```

[36]: 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

```

```

5459.528970943511

```

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 transformers. 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.

```

[37]: # 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):

- What is the intercept and slope for non-smokers and what is the intercept and slope for smokers?
- 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).
- How does this model compare to the previous ones?

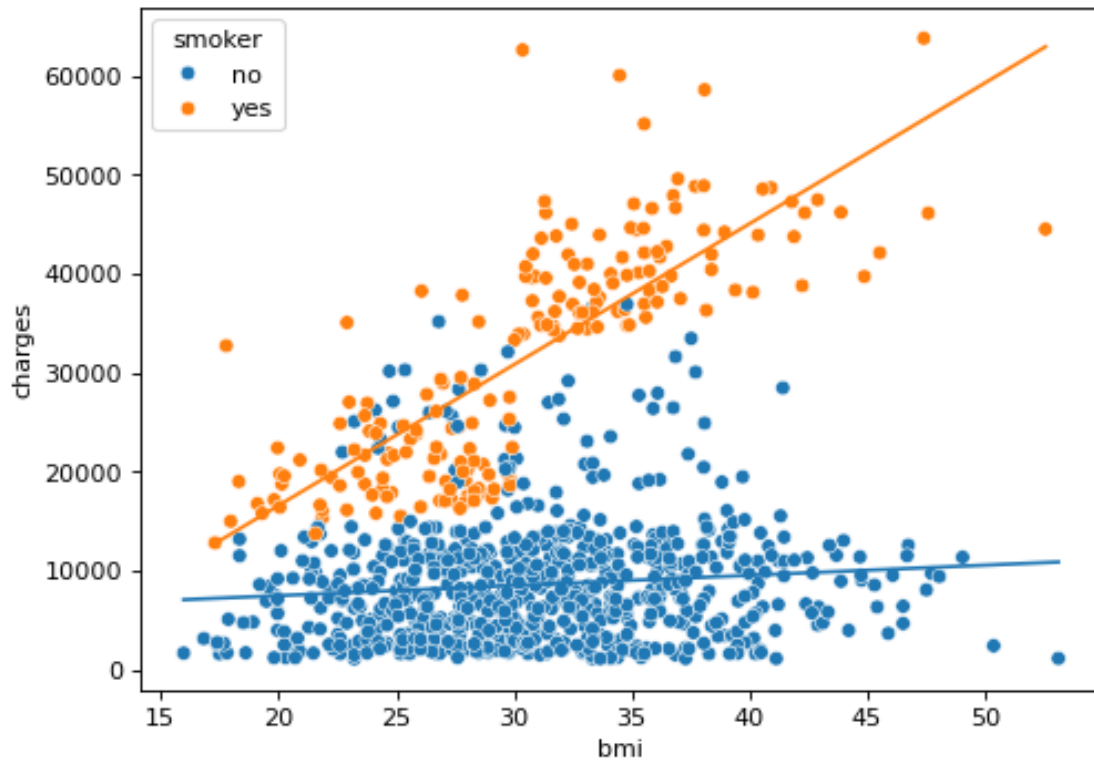
```
[38]: #Part a
w = get_coefs(lr3_fit)

print("The intercept for non-smokers is: %.3f" % w[0])
print("The slope for non-smokers is: %.3f" % w[2])
print("The intercept for smokers is: %.3f" % (w[0]+w[1]))
print("The slope for smokers is: %.3f" % (w[2]+w[3]))
```

```
The intercept for non-smokers is: 5459.529
The slope for non-smokers is: 102.343
The intercept for smokers is: -11806.147
The slope for smokers is: 1421.169
```

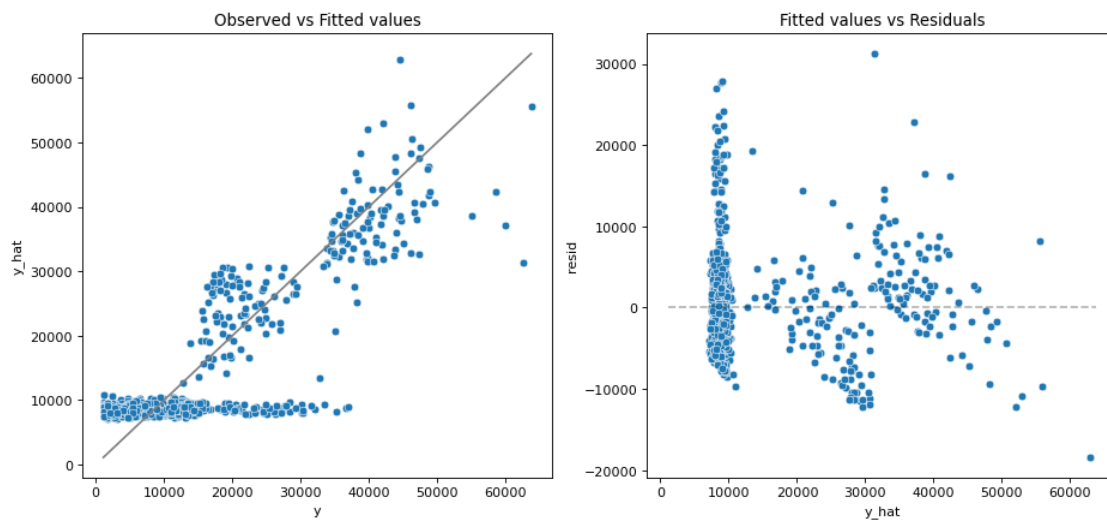
```
[39]: #Part b
# Compute fitted values
yhat = lr3_fit.predict(X_train)

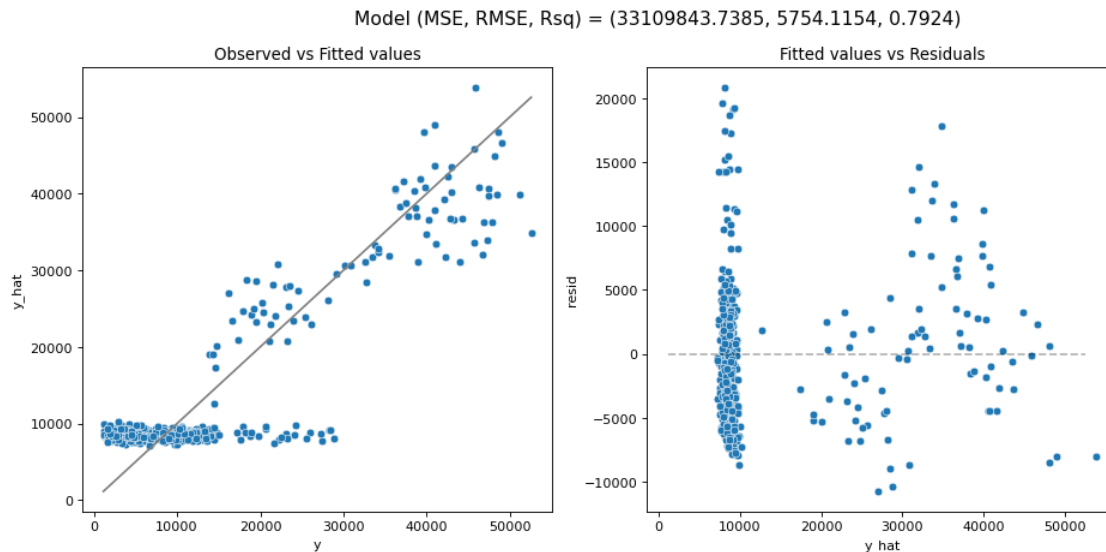
# Part b: Plot fitted values
ax = sns.scatterplot(x = X_train.bmi, y = y_train, hue = X_train.smoker)
sns.lineplot(x = X_train.bmi, y = yhat, hue = X_train.smoker, ax=ax, legend =
↪False)
ax.set(ylabel='charges')
plt.show()
```



```
[40]: # Part c
model_fit(lr3_fit, X_train, y_train, plot = True)
model_fit(lr3_fit, X_test, y_test, plot = True)
```

Model (MSE, RMSE, Rsq) = (39963574.31, 6321.675, 0.7165)





[40]: (33109843.738532413, 5754.115374106815, 0.7923750248000319)

Including the interaction term, results in improved performance metrics and visually seems to fit the data better. We do still observe some patterns in the residuals, such as higher variance in the residuals for non-smokers, but there is less structure than previously.

## 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?

```
[41]: # 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
]
```

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

```
[[ 1.    0.    0.33  0.    0.    0.11]
 [ 1.    0.    2.57  0.    0.    6.59]
 [ 1.    0.   -0.7   0.   -0.    0.49]
 [ 1.    0.   -0.53  0.   -0.    0.28]
 [ 1.    0.   -0.15  0.   -0.    0.02]
 [ 1.    1.    0.28  1.    0.28  0.08]
 [ 1.    0.    0.51  0.    0.    0.26]
 [ 1.    0.    0.85  0.    0.    0.73]
 [ 1.    0.    1.3   0.    0.    1.68]
 [ 1.    0.    0.84  0.    0.    0.71]
 [ 1.    0.   -0.99  0.   -0.    0.97]
 [ 1.    1.    1.11  1.    1.11  1.24]
 [ 1.    0.    0.65  0.    0.    0.42]
 [ 1.    0.    1.21  0.    0.    1.46]
 [ 1.    0.    0.04  0.    0.    0.  ]
 [ 1.    0.    0.26  0.    0.    0.07]
 [ 1.    0.    0.77  0.    0.    0.59]
 [ 1.    0.   -1.81  0.   -0.    3.27]
 [ 1.    0.    1.51  0.    0.    2.29]]
```

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

```
['1' 'bmi' 'smoker' 'bmi^2' 'bmi smoker' 'smoker^2']
```

There are six features after applying polynomial basis function expansion: - First feature is the bias. - Second feature is **bmi**. - Third feature is **smoker**. - Fourth feature is the square of **bmi**. - Fifth feature is the interaction **bmi** times **smoker**. - Sixth feature is the square of **smoker**.

Since **smoker** is binary, the third and sixth columns/features are the same. We do **NOT** want to apply basis expansion to the binary features, as this creates a rank deficient matrix and redundant features.

### 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.

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

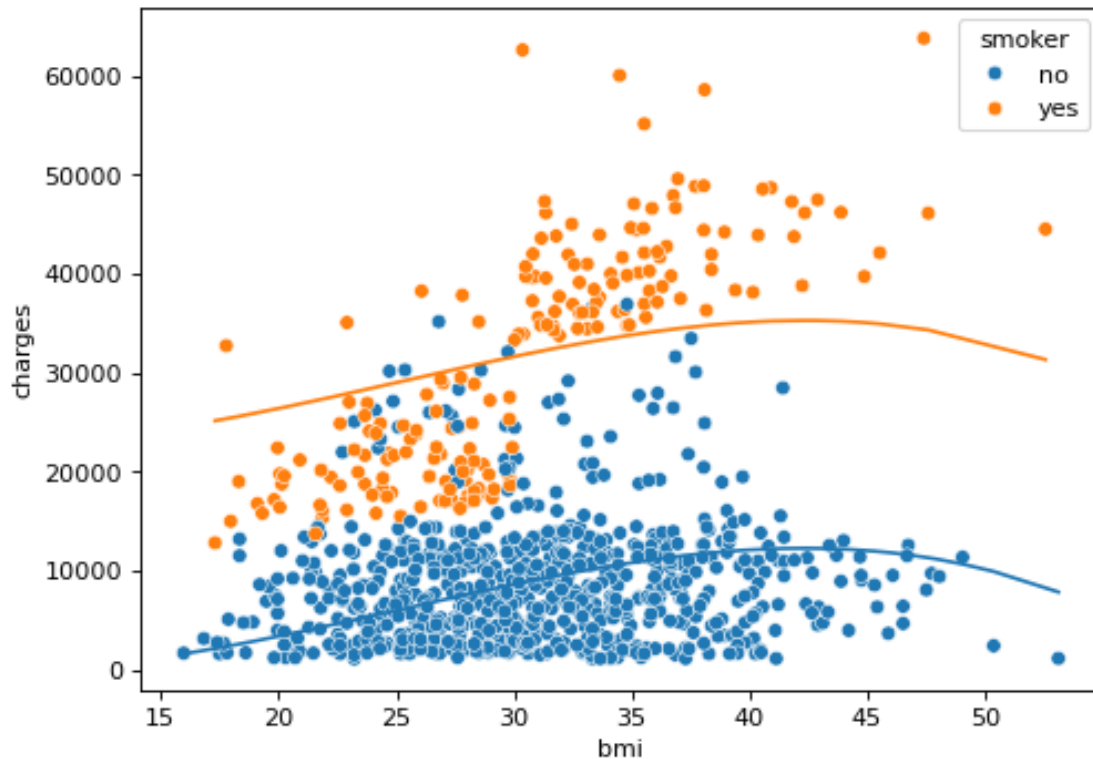
pf3 = PolynomialFeatures(degree=3,include_bias=False)

# Overall ML pipeline
polyreg_pipe = Pipeline([
    ("pre_processing", ColumnTransformer([
        ("cat_pre", cat_pre, [4]), # Applied to smoker
        ("poly", pf3, [2])])), # Applied to bmi
    ("model", LinearRegression())
])

#Train the model
polyreg_fit = polyreg_pipe.fit(X_train,y_train)

[45]: # Compute fitted values
yhat = polyreg_fit.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()
```



```
[46]: print(get_coefs(polyreg_fit))
```

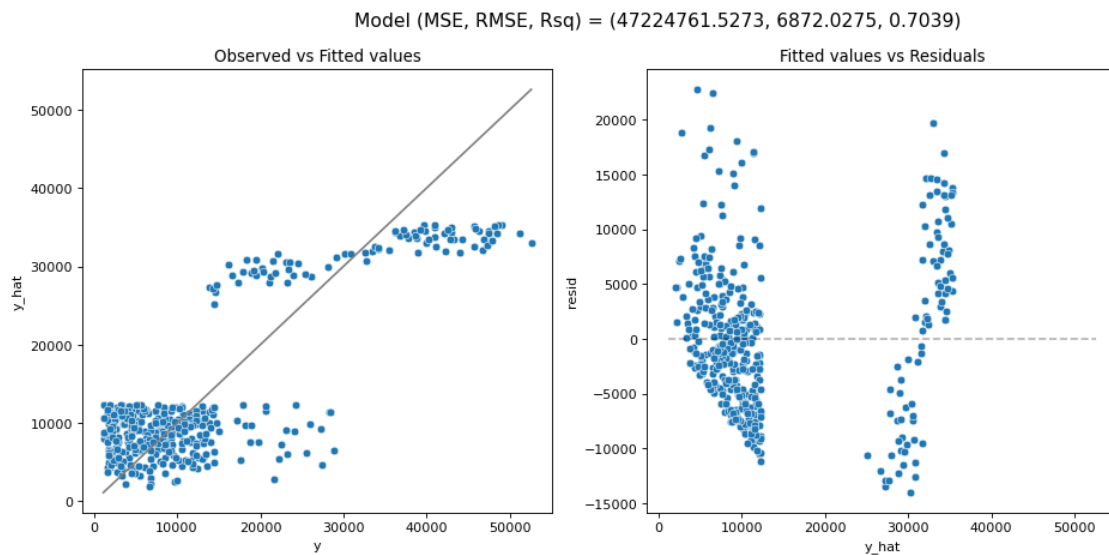
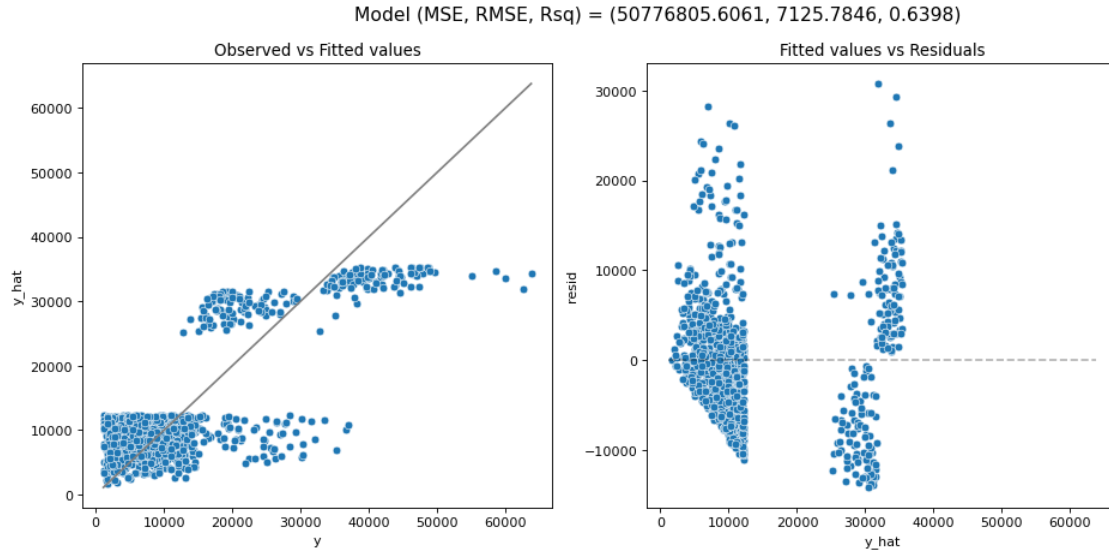
```
[ 2.31876389e+03  2.29965652e+04 -6.25545968e+02  4.62153114e+01
 -6.11273874e-01]
```

To see the names of the features we can again use the method `get_feature_names_out()`.

```
[47]: # Extract the names of the features
names_fe = polyreg_fit["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 = polyreg_fit['model'].coef_
print(np.c_[names_fe,w])
```

```
[['smoker_yes' '22996.565166214958']
 ['bmi' '-625.5459682691634']
 ['bmi^2' '46.21531137525285']
 ['bmi^3' '-0.6112738741172276']]
```

```
[48]: model_fit(polyreg_fit, X_train, y_train, plot = True)
model_fit(polyreg_fit, X_test, y_test, plot = True)
```



[48]: (47224761.52728285, 6872.027468461025, 0.7038632976236113)

## 6.4 Choosing the Order of the Polynomial

How can we choose the order of the polynomial?

In lecture, we discussed how choosing the degree to be too large can cause over fitting. 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?

```
[49]: 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)

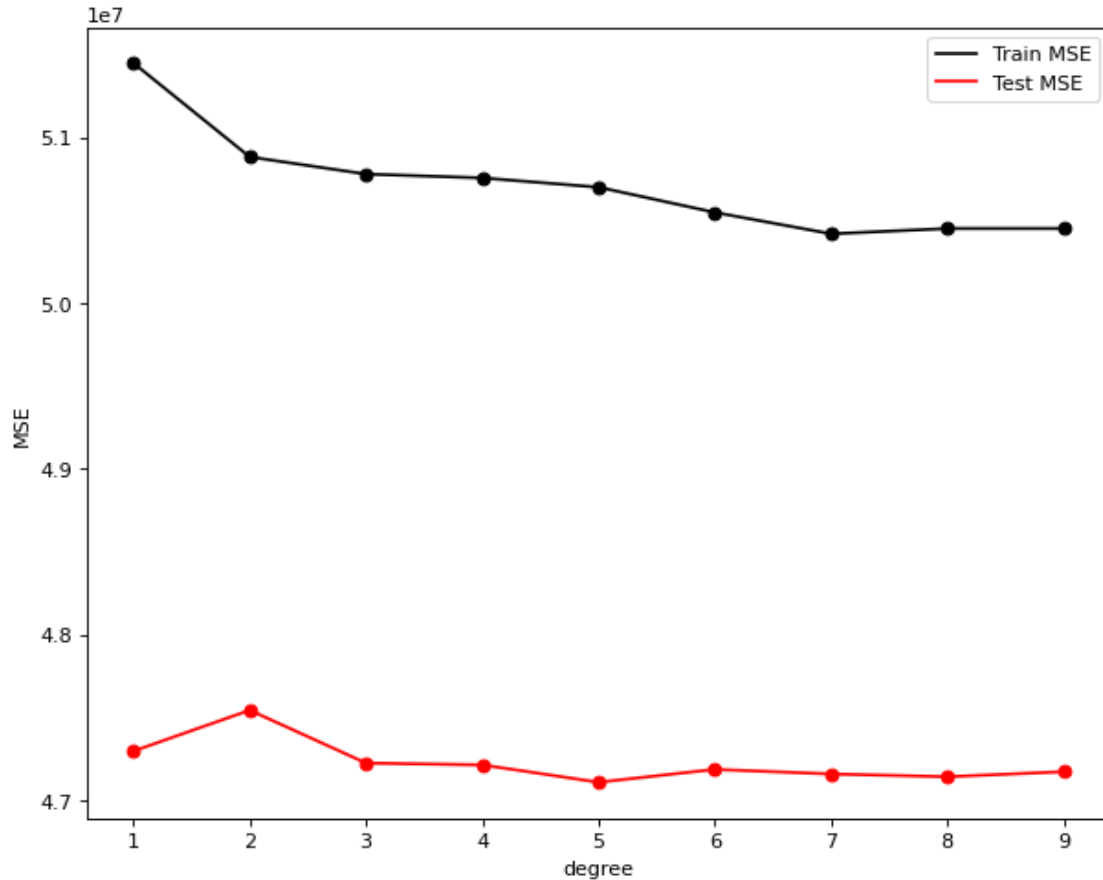
    # Overall ML pipeline
    p = Pipeline([
        ("pre_processing", ColumnTransformer([
            ("cat_pre", cat_pre, [4]), # Applied to smoker
            ("poly", pf, [2])])),
        ("model", LinearRegression())])

    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))
```

```
[50]: 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()
```



- When the degree is increased starting from **1** (linear model) , there is a tendency for the **MSE** on the training data to decrease slightly as we increase the degree of the polynomial.
- The test MSE is relatively flat, suggesting that a linear model may be sufficient. Keep in mind that a large degree polynomial has less explainability with respect to the increased model complexity!

However, we should use CV here to chose the degree and not the test data.

#### 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 hyperparameter 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.

```
[51]: 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)

[52]: #polyreg_pipe.get_params()
```

The above code goes through the process of fitting all  $5 \times 9$  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](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 preferred 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,

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

```
best index: 2
best param: {'pre_processing__poly__degree': 3}
```

best score: -51205617.181880936

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.

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

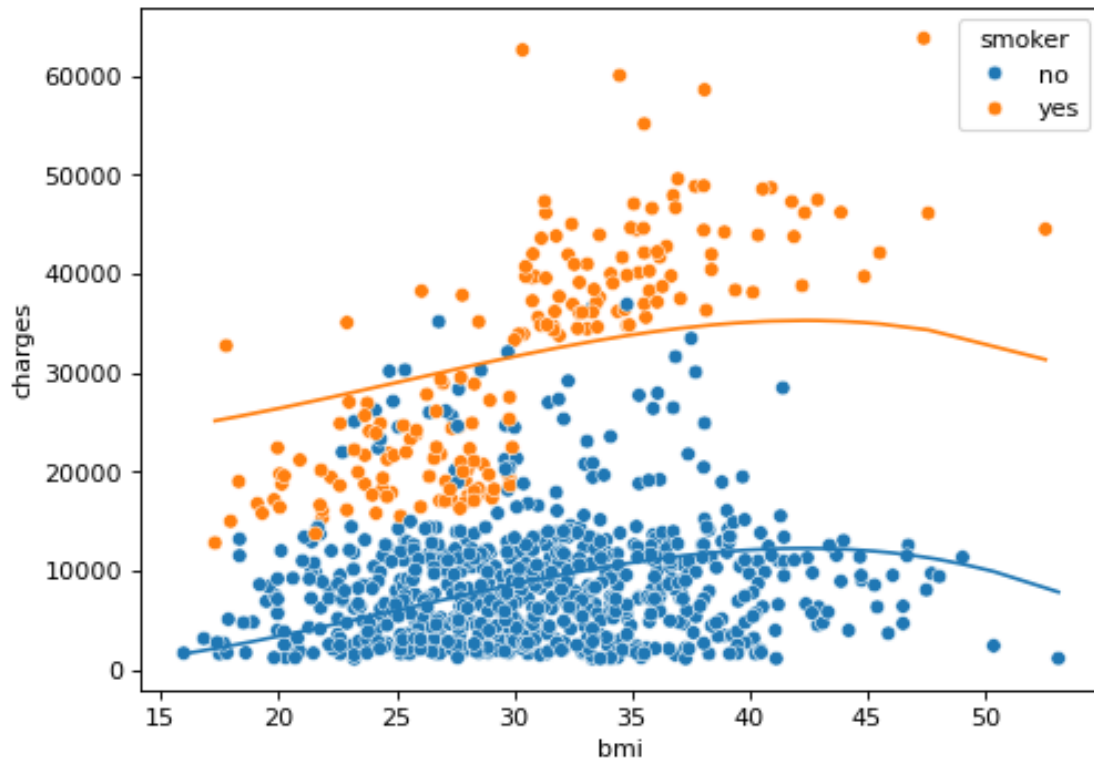
```
[54]: array([ 2.29965652e+04, -6.25545968e+02,  4.62153114e+01, -6.11273874e-01])
```

```
[55]: # 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])
```

```
[['smoker_yes' '22996.565166214958']
 ['bmi' '-625.5459682691634']
 ['bmi^2' '46.21531137525285']
 ['bmi^3' '-0.6112738741172276']]
```

```
[56]: # 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.

```
[57]: cv_results = pd.DataFrame(grid_search.cv_results_)
      cv_results
```

```
[57]:   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0      0.004707    0.000348      0.002842      0.000145
1      0.004565    0.000119      0.002770      0.000094
2      0.004552    0.000086      0.002696      0.000020
3      0.004488    0.000130      0.002640      0.000060
4      0.004437    0.000097      0.002595      0.000009
5      0.004411    0.000122      0.002601      0.000009
6      0.004545    0.000089      0.002753      0.000128
7      0.004452    0.000073      0.002661      0.000042
8      0.005391    0.001548      0.002641      0.000025

   param_pre_processing__poly__degree  params  \
0                                1  {'pre_processing__poly__degree': 1}
1                                2  {'pre_processing__poly__degree': 2}
2                                3  {'pre_processing__poly__degree': 3}
3                                4  {'pre_processing__poly__degree': 4}
```



```

4          5 {'pre_processing__poly__degree': 5}
5          6 {'pre_processing__poly__degree': 6}
6          7 {'pre_processing__poly__degree': 7}
7          8 {'pre_processing__poly__degree': 8}
8          9 {'pre_processing__poly__degree': 9}

split0_test_score split1_test_score split2_test_score split3_test_score \
0 -4.953120e+07 -5.421318e+07 -4.522664e+07 -5.802686e+07
1 -4.863959e+07 -5.332238e+07 -4.693936e+07 -5.682679e+07
2 -4.849697e+07 -5.323485e+07 -4.696317e+07 -5.677668e+07
3 -4.846416e+07 -5.318416e+07 -4.801395e+07 -5.687521e+07
4 -4.831336e+07 -5.309080e+07 -5.040526e+07 -5.682702e+07
5 -4.808782e+07 -5.313367e+07 -5.008988e+07 -5.815677e+07
6 -4.817315e+07 -5.327181e+07 -5.015637e+07 -5.828472e+07
7 -4.812417e+07 -5.331883e+07 -5.073123e+07 -5.825537e+07
8 -4.805491e+07 -5.328753e+07 -5.137225e+07 -5.831935e+07

... mean_test_score std_test_score rank_test_score split0_train_score \
0 ... -5.166557e+07 4.314726e+06 4 -5.197729e+07
1 ... -5.128167e+07 3.496021e+06 2 -5.150724e+07
2 ... -5.120562e+07 3.490302e+06 1 -5.141538e+07
3 ... -5.140654e+07 3.288286e+06 3 -5.139295e+07
4 ... -5.184219e+07 2.916572e+06 5 -5.138029e+07
5 ... -5.195126e+07 3.494663e+06 6 -5.124288e+07
6 ... -5.203509e+07 3.523605e+06 7 -5.106325e+07
7 ... -5.215016e+07 3.470058e+06 8 -5.111436e+07
8 ... -5.226487e+07 3.467621e+06 9 -5.112708e+07

split1_train_score split2_train_score split3_train_score \
0 -5.076974e+07 -5.300751e+07 -4.984967e+07
1 -5.029181e+07 -5.197860e+07 -4.944856e+07
2 -5.018797e+07 -5.185925e+07 -4.932849e+07
3 -5.017128e+07 -5.166660e+07 -4.932628e+07
4 -5.012581e+07 -5.131375e+07 -4.932590e+07
5 -4.993415e+07 -5.130996e+07 -4.889950e+07
6 -4.974740e+07 -5.130986e+07 -4.870801e+07
7 -4.977748e+07 -5.129224e+07 -4.875317e+07
8 -4.978745e+07 -5.124731e+07 -4.874294e+07

split4_train_score mean_train_score std_train_score
0 -5.150686e+07 -5.142221e+07 1.070237e+06
1 -5.095547e+07 -5.083633e+07 8.934741e+05
2 -5.086011e+07 -5.073024e+07 8.966502e+05
3 -5.085550e+07 -5.068252e+07 8.489775e+05
4 -5.077096e+07 -5.058334e+07 7.738649e+05
5 -5.065533e+07 -5.040837e+07 9.024238e+05
6 -5.050278e+07 -5.026626e+07 9.463187e+05

```

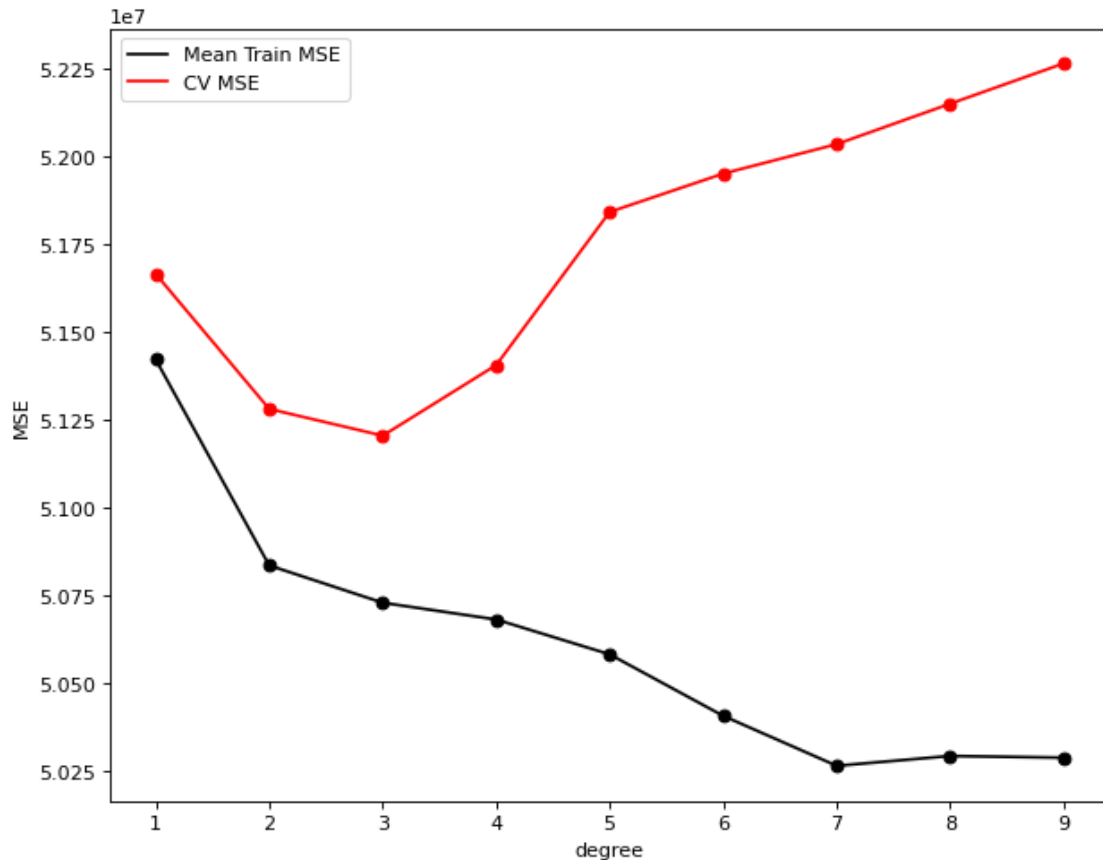
7	-5.053417e+07	-5.029428e+07	9.346777e+05
8	-5.054229e+07	-5.028941e+07	9.302402e+05

[9 rows x 21 columns]

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.

```
[58]: 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?

The curve is relatively flat around degrees 2 and 3, thus we may prefer the simpler model with degree 2.

### 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?

There are many examples that one could try and answers will vary. Here is another example, allowing for different basis coefficients and degree of polynomials for smokers and non-smokers.

```
[59]: pf_d = PolynomialFeatures(include_bias=False)
      pf = PolynomialFeatures(interaction_only=True, include_bias=False)

      # Overall ML pipeline
      polyreg2_pipe = Pipeline([
          ("pre_processing", ColumnTransformer([
```

```

        ("poly", 'passthrough', [2]), # Applied to bmi
        ("cat_pre", OneHotEncoder(), [4]))), # Applied to smoker (keep both
↳ columns)
        ("interact", pf), # add interaction (this creates all interactions)
        ("fe", ColumnTransformer([
            ("cat_pass", 'passthrough', [1,2]), # Applied to the two columns for
↳ smoker
            ("poly1", pf_d, [3]), # Applied to the interaction between bmi and
↳ smoker
            ("poly2", pf_d, [4]))]), # Applied to the interaction between bmi and
↳ nonsmoker
        ("model", LinearRegression(fit_intercept=False))])

parameters = {
    'fe__poly1__degree': np.arange(1,5,1), 'fe__poly2__degree': np.arange(1,5,1)
}

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

grid_search2 = GridSearchCV(polyreg2_pipe, parameters, cv = kf, scoring =
↳ 'neg_mean_squared_error', return_train_score=True).fit(X_train, y_train)

```

```

[60]: print("best param: ", grid_search2.best_params_)
      print("best score: ", grid_search2.best_score_)
      print("coefficients", grid_search2.best_estimator_['model'].coef_)

```

```

best param: {'fe__poly1__degree': 2, 'fe__poly2__degree': 1}
best score: -39976414.67705377
coefficients [ -6862.50780585 -11806.14729951    907.61388444   -12.6492562
              1421.16878591]

```

```

[61]: # Extract the names of the features from the first step
names_fe1 = grid_search2.best_estimator_["pre_processing"].
↳ get_feature_names_out()
# Strip the name of the column transformer
names_fe1 = [names_fe1[i].partition('__')[2] for i in range(len(names_fe1))]
print(names_fe1)
# Extract the names of the features from the second step
names_fe2 = grid_search2.best_estimator_["interact"].
↳ get_feature_names_out(names_fe1)
print(names_fe2)
# Extract the names of the features from the third step
names_fe3 = grid_search2.best_estimator_["fe"].get_feature_names_out(names_fe2)
# Strip the name of the column transformer
names_fe3 = [names_fe3[i].partition('__')[2] for i in range(len(names_fe3))]
print(names_fe3)
w = grid_search2.best_estimator_['model'].coef_

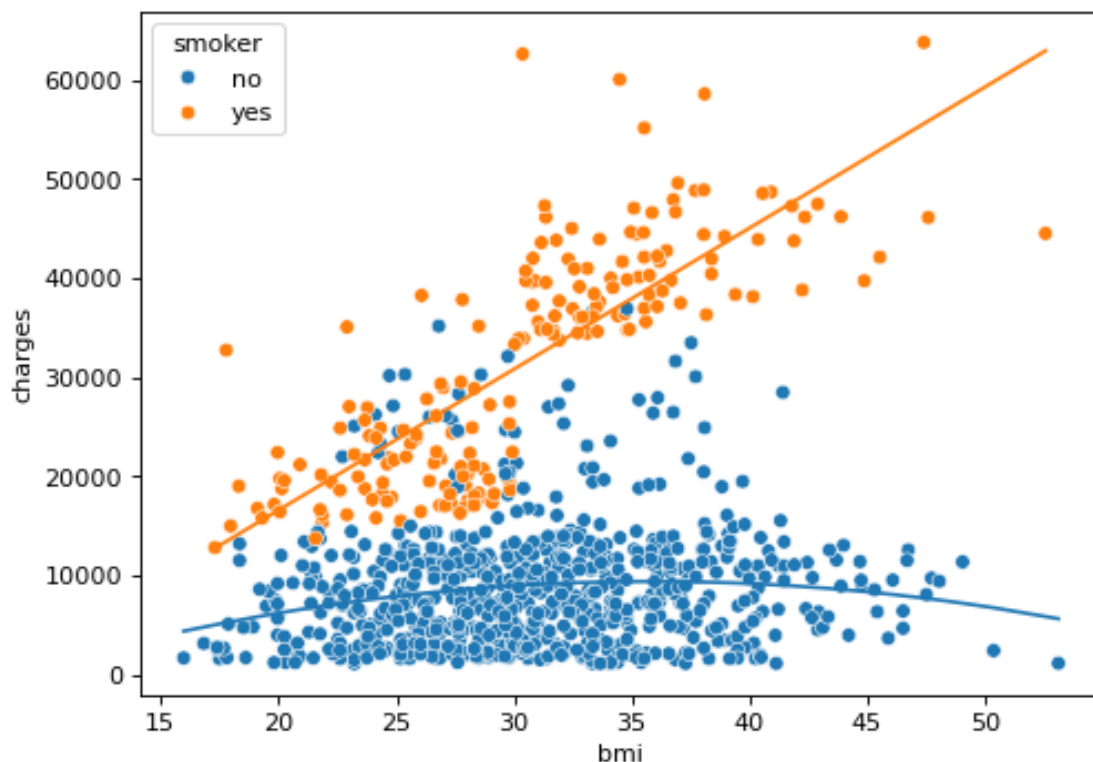
```

```
print(np.c_[names_fe3,w])
```

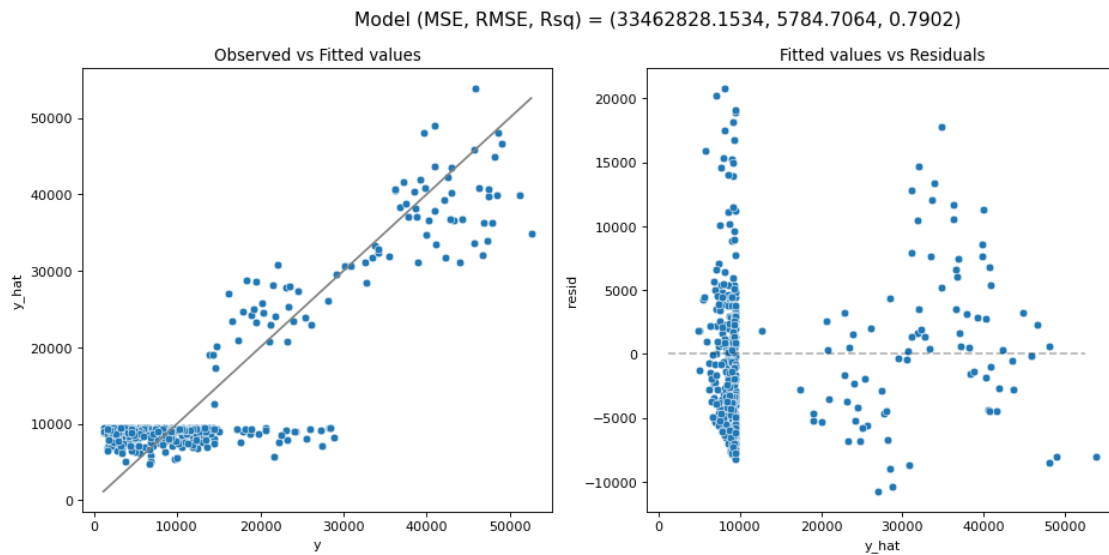
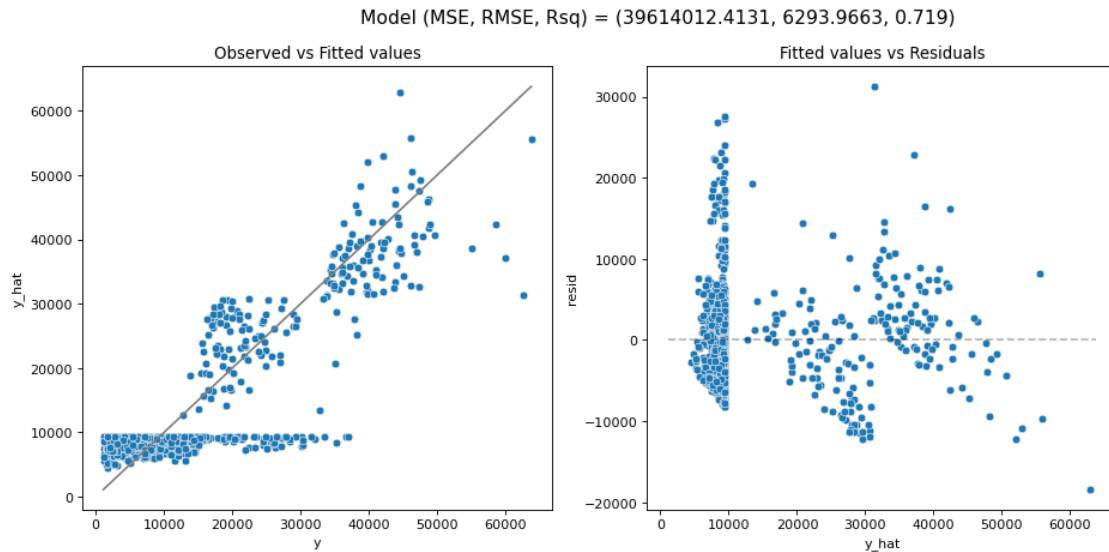
```
['bmi', 'smoker_no', 'smoker_yes']
['bmi' 'smoker_no' 'smoker_yes' 'bmi smoker_no' 'bmi smoker_yes'
 'smoker_no smoker_yes']
['smoker_no', 'smoker_yes', 'bmi smoker_no', 'bmi smoker_no^2', 'bmi
smoker_yes']
[['smoker_no' '-6862.507805847576']
 ['smoker_yes' '-11806.147299512555']
 ['bmi smoker_no' '907.6138844447283']
 ['bmi smoker_no^2' '-12.64925619795281']
 ['bmi smoker_yes' '1421.1687859134872']]
```

```
[62]: # Compute fitted values
yhat = grid_search2.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()
```



```
[63]: model_fit(grid_search2.best_estimator_, X_train, y_train, plot = True)
      model_fit(grid_search2.best_estimator_, X_test, y_test, plot = True)
```



[63]: (33462828.153391182, 5784.706401658703, 0.7901615326144515)

## 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#](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_key.ipynb
```

```
[ ]:
```