MLPy Workshop 4: Solutions

Sara Wade, Ozan Evkaya

February 8, 2024

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.

```
[92]: # 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
```

```
[93]: # Plotting defaults for all Figures below
plt.rcParams['figure.figsize'] = (8,5)
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.

```
[94]: 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_])
```

```
[95]: 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("Actual vs Fitted␣
       ↪plot")

              plt.subplot(122)
              sns.scatterplot(x='y_hat', y='resid', data=res).set_title("Fitted vs␣
       ↪Residual plot")
              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, Rsqr) = " + 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.

[96]:
```
df_insurance = pd.read_csv("insurance.csv")
df_insurance.head()
```

[96]:
```
   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?

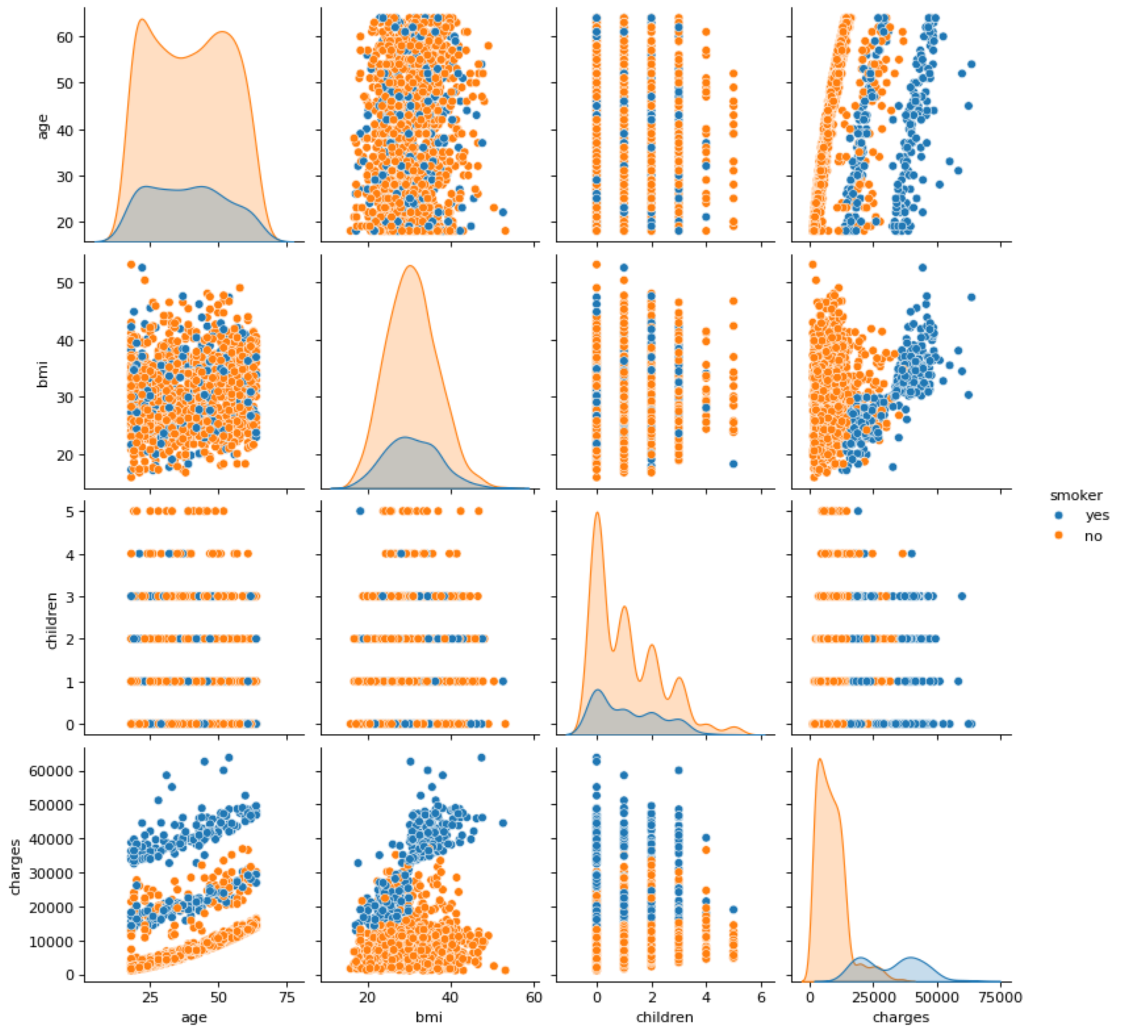
b) Create a pairs plot of the data (make sure to include the `smoker` column), describe any relationships you observe in the data. To better visualize the relationship between `children` and `charges`, create a violin plot (since `children` only takes a small number of integer values, many points are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

Hint

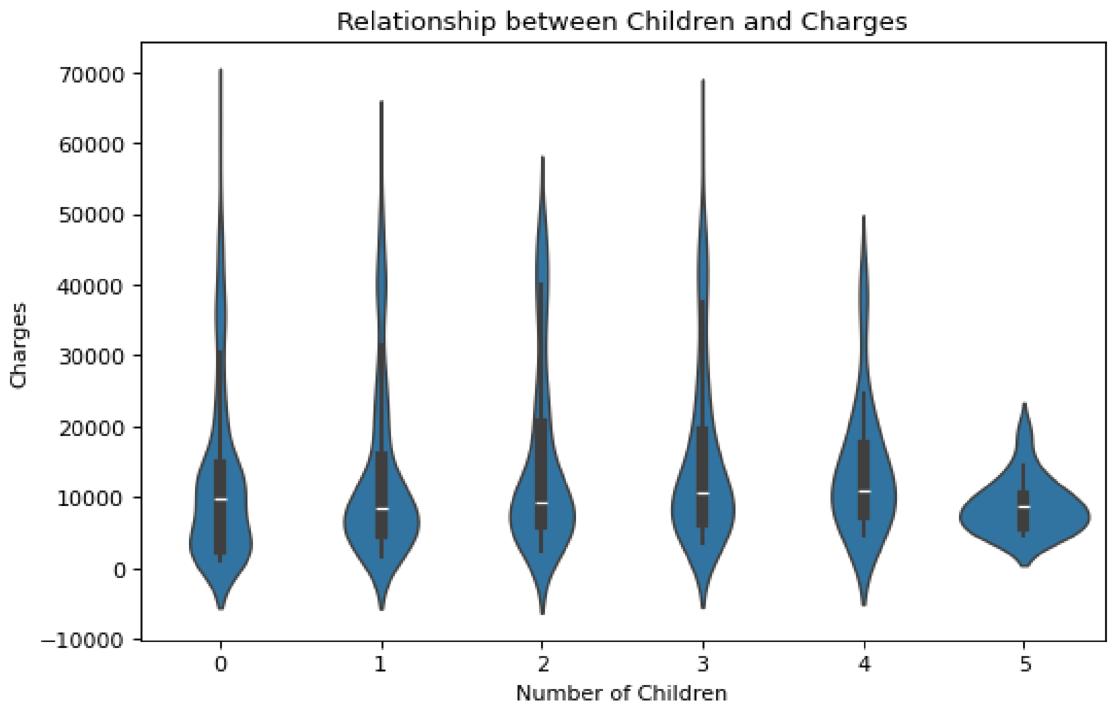
- .describe() can be used to create summary descriptive statistics on a pandas dataframe.
- You can use a sns.pairplot and sns.violinplot with the hue argument

[97]:
```
# Part a
sns.pairplot(df_insurance, hue='smoker')
```

[97]:
```
<seaborn.axisgrid.PairGrid at 0x16bb32930>
```



```
[98]: # Part b
      sns.violinplot(x='children', y='charges', data=df_insurance)
      plt.title('Relationship between Children and Charges')
      plt.xlabel('Number of Children')
      plt.ylabel('Charges')
      plt.show()
```



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.

[99]:
```
# 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)

Use `train_test_split()` to split the data randomly into training (70%) and test (30%) sets. The `training set` will contain our test data, which you should call `X_train` and `y_train`. The `test set` will contain our testing data, which you should call `X_test` and `y_test`. Don't forget to pass in the `rng` variable to `random_state`.

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

[100]:
```
from sklearn.model_selection import train_test_split

# First split off features and targets
X = df_insurance.drop('charges', axis = 1) # Set of features
y = df_insurance['charges']

# Split the data into train-test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=rng)

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[100]:
```
((1070, 5), (268, 5), (1070,), (268,))
```

Situations when shuffling might be beneficial

- IID Data: Trivial
- Data with class imbalances e.g. transactions datasets for fraud analysis
- When we want to cross-validate our model

Data when shuffling will not be beneficial

- Time-series data: Trivial
- Data with other types of sequential relationships.

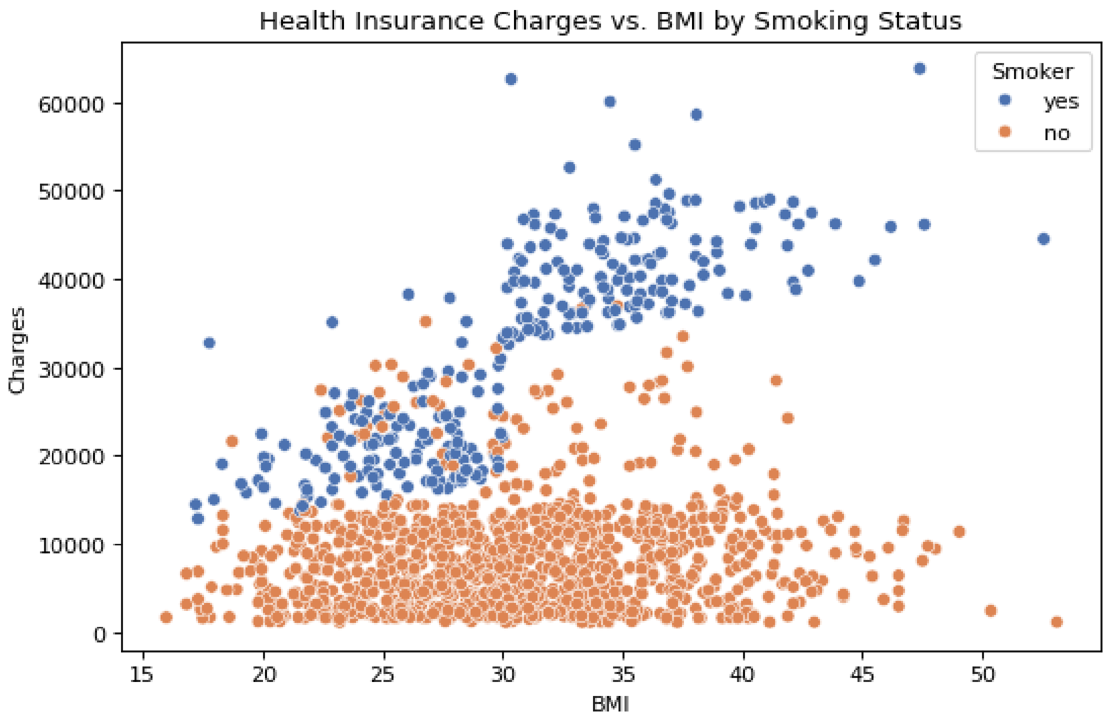
4 Least Squares Estimation

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 the `df_insurance` data frame. Describe any apparent relationship between `charges` and `bmi` (including the `smoker` attribute) and comment on the difference between smokers and non-smokers.

[101]:
```python
sns.scatterplot(data=df_insurance, x='bmi', y='charges', hue='smoker',␣
 ↪palette='deep')

plt.title('Health Insurance Charges vs. BMI by Smoking Status')
plt.xlabel('BMI')
plt.ylabel('Charges')
plt.legend(title='Smoker')
plt.show()
```



There is positive correlation between the features `bmi`, `smoker` and the target `charges` however the correllation is stronger for the `smoker` feature.

The variability in charges are high for smokers and non-smokers but there seems to be a stronger linear relationship between BMI and charges for smokers thus creating a stronger indicator of higher medical expenses.

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`).

[102]:
```
# Part a
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
X_train_encoded_smoker = le.fit_transform(X_train['smoker'])

# Construct the design matrix
X_design = np.column_stack((np.ones(X_train.shape[0]), X_train['bmi'],␣
 ↪X_train_encoded_smoker))

y_train_array = y_train.to_numpy()
```

[103]:
```
# Part b
from numpy.linalg import solve
w_hat = solve(X_design.T @ X_design, X_design.T @ y_train_array)
```

[104]:
```
w_hat
```

[104]:
```
array([-3434.74482161,   386.94616325, 23187.3567059 ])
```

[105]:
```
# Part c
intercept_non_smoker = w_hat[0]
intercept_smoker = w_hat[0] + w_hat[2]

print("Intercept for Non-Smokers:", intercept_non_smoker)
print("Intercept for Smokers:", intercept_smoker)
```

```
Intercept for Non-Smokers: -3434.7448216114212
Intercept for Smokers: 19752.611884286005
```

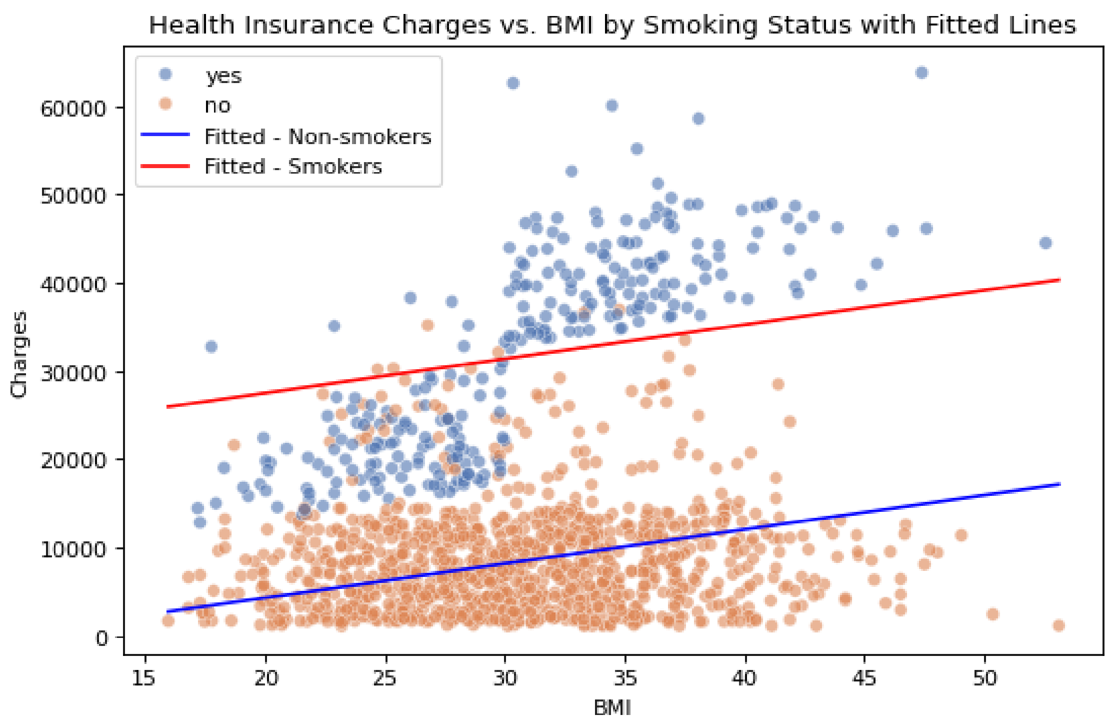
4.0.3 Exercise 5 (CORE)

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

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

[106]:
```
# Part a: Compute fitted values
y_hat = X_design @ w_hat
```

[107]:
```
# Part b: Plot fitted values
X_full = df_insurance['bmi']
smoker_encoded_full = le.transform(df_insurance['smoker'])
X_full_design = np.column_stack((np.ones(df_insurance.shape[0]),␣
 ↪df_insurance['bmi'], smoker_encoded_full))
y_full_hat = X_full_design.dot(w_hat)

# Scatter plot of charges vs. bmi colored by smoker status with the fitted␣
 ↪regression lines
sns.scatterplot(data=df_insurance, x='bmi', y='charges', hue='smoker',␣
 ↪palette='deep', alpha=0.6)

# Overlay the fitted regression lines

# For non-smokers
sns.lineplot(x=X_full, y=X_full_design[:,0]*w_hat[0] + X_full_design[:
 ↪,1]*w_hat[1] + 0*w_hat[2], color='blue', label='Fitted - Non-smokers')
# For smokers
sns.lineplot(x=X_full, y=X_full_design[:,0]*w_hat[0] + X_full_design[:
 ↪,1]*w_hat[1] + 1*w_hat[2], color='red', label='Fitted - Smokers')

plt.title('Health Insurance Charges vs. BMI by Smoking Status with Fitted␣
 ↪Lines')
plt.xlabel('BMI')
plt.ylabel('Charges')
plt.legend()
plt.show()
```



- The regression lines show clearly that there are two groups based on smoking. This is inline with what we gathered from the coeffs.
- There are linear relationships between charges and BMI for both smokers nd non-smokers like we found two exercises ago.
- The scatter-plot indicates that there are potential non-linear relationships andheteroscedasticity since the variance of the charges increase as the BMI increases.

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,

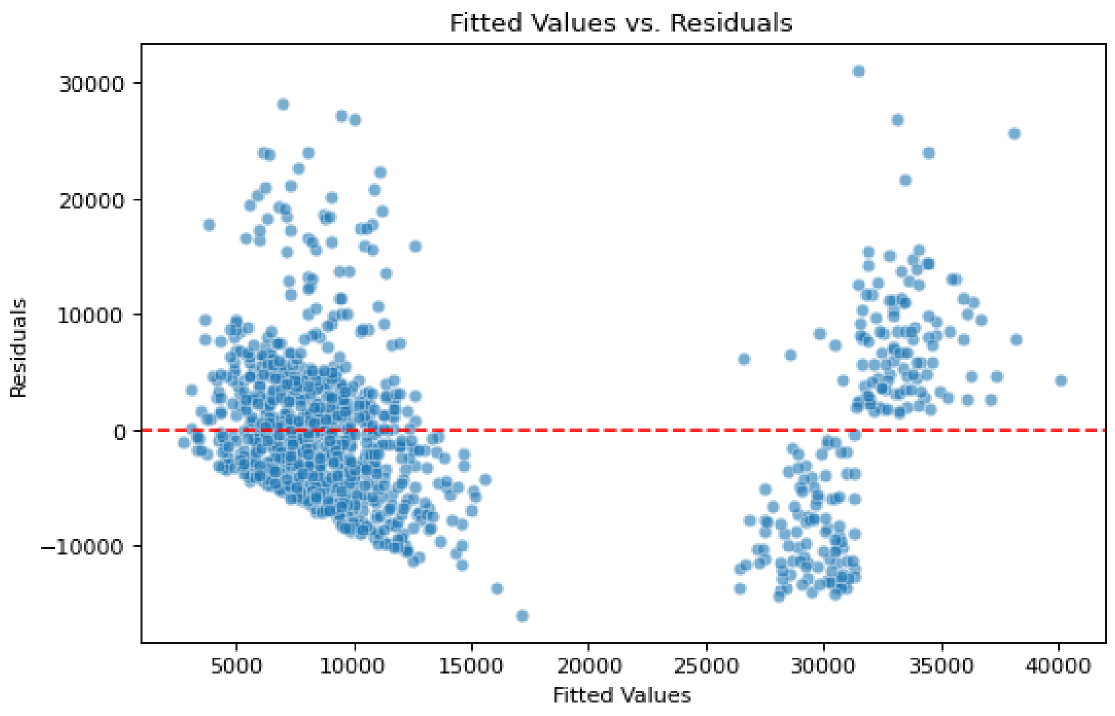
- If the normal linear model assumption is true then the residuals should be randomly scattered around zero with no discernible clustering or pattern with respect to the fitted values.
- Furthermore, this plot can be useful to check the constant variance (homoscedastic) assumption to see whether the 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).

[108]:
```python
# Calculate the residuals
residuals = y_train_array - y_hat

# Scatter plot of fitted values vs. residuals
sns.scatterplot(x=y_hat, y=residuals, alpha=0.6)
plt.axhline(y=0, color='red', linestyle='--')

plt.title('Fitted Values vs. Residuals')
plt.xlabel('Fitted Values')
plt.ylabel('Residuals')
plt.show()
```



The variance of the residuals does not appear to be constant across the range of fitted values. Specifically, for higher fitted values (which correspond to smokers), the residuals exhibit a broader spread, indicating heteroscedasticity. This violates the assumption of homoscedasticity (constant variance) for linear regression models.

[109]:
```python
# Part b
# Calculate R-squared value
mean_y = np.mean(y_train_array)
ss_residual = np.sum((y_train_array - y_hat) ** 2)
ss_total = np.sum((y_train_array - mean_y) ** 2)
r_squared = 1 - (ss_residual / ss_total)

# Print the R-squared value
print("R-squared value:", r_squared)
```

```
R-squared value: 0.6442384337089928
```

$R^2 = 0.654$ indicates a moderately strong fit with room for improvement.

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.

[110]:
```python
# Compute the least squares estimate when both dummy variable are included
smoker_dummies = pd.get_dummies(X_train['smoker'], drop_first=False)
X_design_naive = np.column_stack((np.ones(X_train.shape[0]), X_train['bmi'],␣
 ↪smoker_dummies))
```

[111]:
```python
# Compute the rank of the design matrix
from numpy.linalg import matrix_rank

rank_naive = matrix_rank(X_design_naive)

rank_naive, X_design_naive.shape[1]
```

[111]:
```
(3, 4)
```

This discrepancy between the rank of the matrix (3) and the number of columns (4) indicates that the matrix is rank deficient. In other words, there is a linear dependency among the columns of the design matrix. The LSE is not feasible in this situation due to the rank-deficiency.

---

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.

[112]:
```python
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.

```
[113]: 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_`.

```
[114]: 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)
```

```
[-3434.74482161   386.94616325 23187.3567059 ]
```

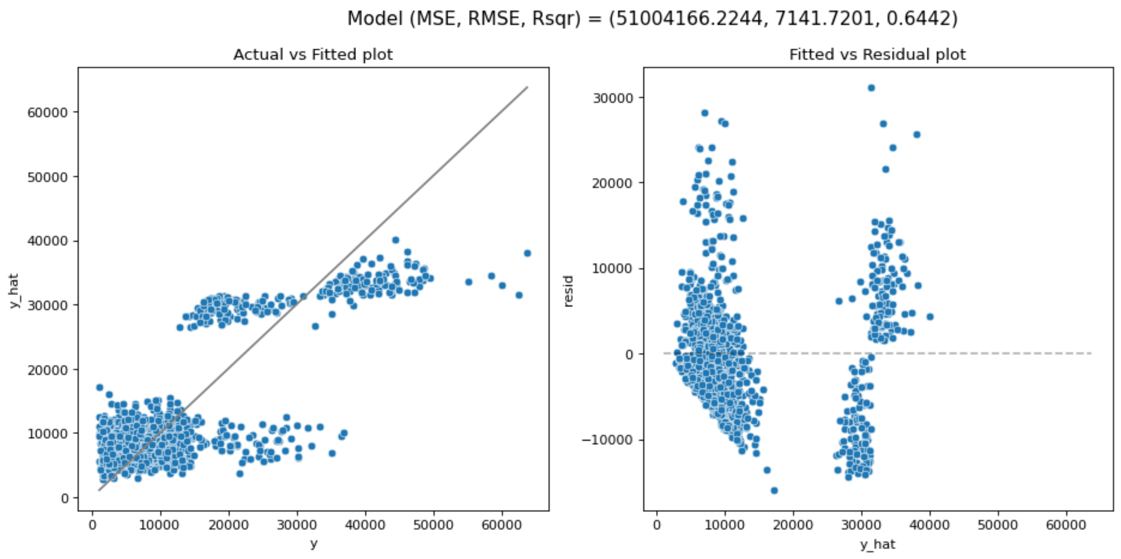
```
[114]: array([-3434.74482161,   386.94616325, 23187.3567059 ])
```

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.

```
[115]: # 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 coefficient of determination 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 coefficient of determination 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: 51004166.224
Training R squared: 0.644
Test Mean squared error: 46797181.737
Test R squared: 0.706
0.6442384337089928
```

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



```
[116]: (51004166.22441513, 7141.720116639627, 0.6442384337089928)
```

5.0.1 Exercise 8 (CORE)

Consider a pipeline for a regression model, using sklearn functionalities

- including `bmi` and `age` as numerical values and smoker condition as a categorical value
- Apply encoding for the `smoker` variable within the pipeline
- Fit a model including these variables and calculate performance metrics similar to above (R^2, and MSE / RMSE). How does this model compare to previous one?

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.

[117]:
```python
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Define column transformer for preprocessing
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), ['bmi', 'age']),
        ('cat', OneHotEncoder(drop=['no']), ['smoker'])
    ])

# Create the regression pipeline
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])
```

[118]:
```python
# Fit the model and calculate metrics
pipeline.fit(X_train[['bmi', 'age', 'smoker']], y_train)

# Predict on training and test sets
y_train_pred = pipeline.predict(X_train[['bmi', 'age', 'smoker']])
y_test_pred = pipeline.predict(X_test[['bmi', 'age', 'smoker']])

# Calculate performance metrics
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
train_rmse = np.sqrt(train_mse)
test_rmse = np.sqrt(test_mse)

train_r2, test_r2, train_rmse, test_rmse
```

[118]:
```
(0.7342532129942394, 0.7945500805653087, 6172.446341436985, 5717.80009607945)
```

Comparison to the Previous Model:

- **Improved Performance:** The inclusion of `age` as an additional predictor alongside `bmi` and the encoded `smoker` variable results in a model that performs better than the one considering only `bmi` and `smoker` status. This is evident from the higher (R^2) values for both the training and test sets, indicating a better fit to the data.
- **Decreased Prediction Error:** The RMSE values for both the training and test sets are lower than what was observed with the previous model, suggesting that predictions are, on average, closer to the actual insurance charges.
- The addition of `age` as a predictor and the proper handling of the `smoker` categorical variable within a `scikit-learn` pipeline has led to an improved regression model. This model not only explains a higher proportion of variance in the insurance charges but also reduces the average prediction error, making it more effective for predicting insurance charges based on `bmi`, `age`, and smoking status.

---

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.

[119]:
```python
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:

[120]:
```python
x = np.array([1, 2, 3, 4])
PolynomialFeatures(degree = 2).fit_transform(x.reshape(-1,1))
```

[120]:
```
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`.

[121]:
```
PolynomialFeatures(degree = 2, include_bias=False).fit_transform(x.
 ↪reshape(-1,1))
```

[121]:
```
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:

[122]:
```
x = np.array([[1, 2, 3, 4],[0,0,1,1]]).T
PolynomialFeatures(interaction_only=True,include_bias=False).fit_transform(x)
```

[122]:
```
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`

[123]:
```
# 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())
])
```

[124]:
```
#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.

[125]:
```
print(reg_pipe_2.named_steps['model'].coef_)
```

```
[-17992.39949983     88.34948331   1347.74608661]
```

[126]:
```
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
```

```
5759.032918562546
```

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.

[127]:
```
# Extract the names of the features
names_fe1 = reg_pipe_2['pre_processing'].get_feature_names_out()
#print(names_fe1)
print(reg_pipe_2['interact'].get_feature_names_out(names_fe1))
```

```
['cat_pre__smoker_yes' 'num_pre__bmi' 'cat_pre__smoker_yes num_pre__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?

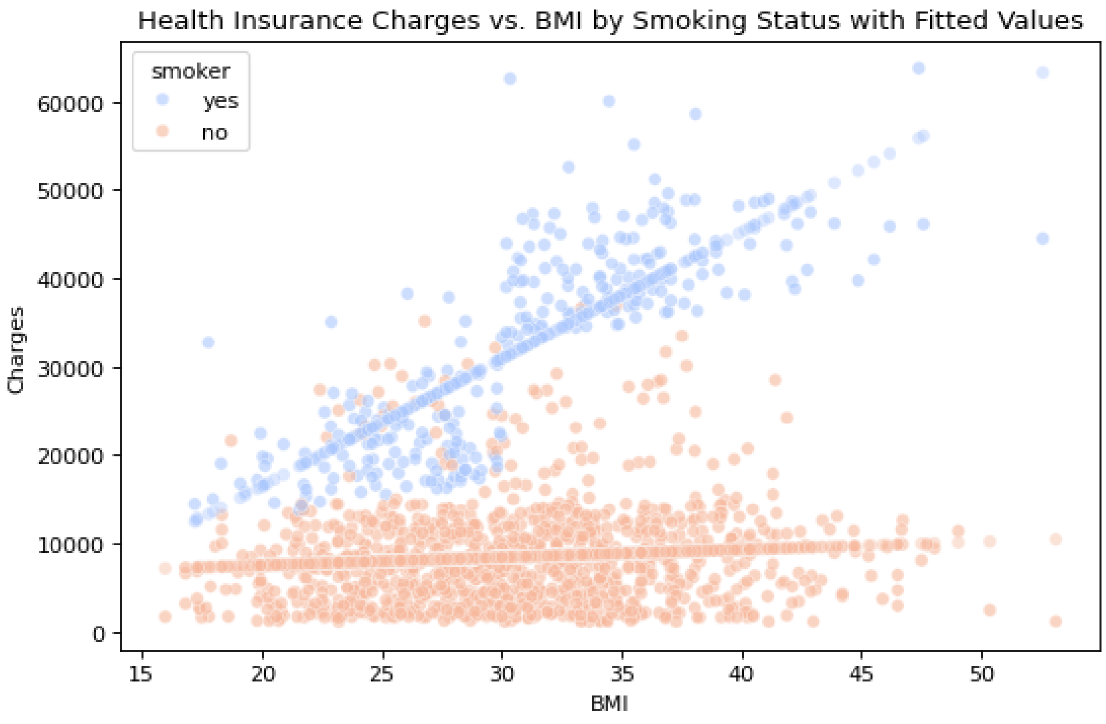
b) Compute the fitted values by calling `predict`. Draw the scatter plot of bmi against charges, colored by smoker (from Exercise 3), and overlay a line plot of the fitted values (fitted regression lines).

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

[128]:
```
#Part a
model_coefs = lr3_fit.named_steps['model'].coef_
model_intercept = lr3_fit.named_steps['model'].intercept_

model_intercept, model_coefs
```

[128]:
```
(5759.032918562546, array([-17992.39949983,     88.34948331,   1347.74608661]))
```

[129]:
```
#Part b
# Compute the fitted values for the entire dataset for visualization
y_fitted = lr3_fit.predict(df_insurance[['smoker', 'bmi']])

# Scatter plot of charges vs. bmi colored by smoker status with the fitted␣
↪regression lines
sns.scatterplot(data=df_insurance, x='bmi', y='charges', hue='smoker',␣
↪palette='coolwarm', alpha=0.6)

# Overlay the fitted values
# It's challenging to directly plot a line due to the interaction term; we plot␣
↪the fitted values instead
sns.scatterplot(x=df_insurance['bmi'], y=y_fitted, hue=df_insurance['smoker'],␣
↪palette='coolwarm', alpha=0.4, legend=False)

plt.title('Health Insurance Charges vs. BMI by Smoking Status with Fitted␣
↪Values')
plt.xlabel('BMI')
plt.ylabel('Charges')
plt.show()
```



Part c

- Improved Fit: By considering interactions, the model can capture more complex patterns in the data, likely leading to improved accuracy and a better understanding of the factors influencing insurance charges.

- Increased Complexity: While the model may perform better in terms of fit and predictive accuracy, it also introduces additional complexity, which can make interpretation more challenging.

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, 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?

Hint

- Try printing out the names of each feature with the method `get_feature_names_out(['bmi','smoker'])` on your fitted `PolynomialFeatures`object.

[130]:
```
# 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
]
```

[131]:
```
# Create transformed feature matrix
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X_)

# Print out the first 20 rows of the transformed feature matrix
X_poly[:20, :], poly.get_feature_names_out(['smoker', 'bmi'])
```

```
[131]: (array([[ 1.        ,  1.        ,  0.54530479,  1.        ,  0.54530479,
          0.29735732],
        [ 1.        ,  0.        ,  0.59867181,  0.        ,  0.        ,
          0.35840793],
        [ 1.        ,  1.        ,  0.96092064,  1.        ,  0.96092064,
          0.92336847],
        [ 1.        ,  0.        ,  0.72319484,  0.        ,  0.        ,
          0.52301078],
        [ 1.        ,  0.        ,  0.26957522,  0.        ,  0.        ,
          0.0726708 ],
        [ 1.        ,  0.        , -0.63685545,  0.        , -0.        ,
          0.40558486],
        [ 1.        ,  1.        ,  0.72319484,  1.        ,  0.72319484,
          0.52301078],
        [ 1.        ,  0.        ,  0.4078443 ,  0.        ,  0.        ,
          0.16633697],
        [ 1.        ,  0.        ,  0.10057967,  0.        ,  0.        ,
          0.01011627],
        [ 1.        ,  0.        , -1.03791665,  0.        , -0.        ,
          1.07727097],
        [ 1.        ,  0.        ,  0.26957522,  0.        ,  0.        ,
          0.0726708 ],
        [ 1.        ,  0.        , -0.32797363,  0.        , -0.        ,
          0.1075667 ],
        [ 1.        ,  0.        , -0.57540252,  0.        , -0.        ,
          0.33108806],
        [ 1.        ,  0.        ,  0.82265155,  0.        ,  0.        ,
          0.67675557],
        [ 1.        ,  0.        , -0.1759585 ,  0.        , -0.        ,
          0.03096139],
        [ 1.        ,  0.        , -0.09914234,  0.        , -0.        ,
          0.0098292 ],
        [ 1.        ,  0.        ,  1.02237356,  0.        ,  0.        ,
          1.0452477 ],
        [ 1.        ,  0.        ,  0.31889927,  0.        ,  0.        ,
          0.10169675],
        [ 1.        ,  0.        ,  0.5808828 ,  0.        ,  0.        ,
          0.33742483],
        [ 1.        ,  1.        , -0.88266715,  1.        , -0.88266715,
          0.7791013 ]]),
 array(['1', 'smoker', 'bmi', 'smoker^2', 'smoker bmi', 'bmi^2'],
       dtype=object))
```

Features(6)

- Intercept (Bias) Term: Represented by the constant 1.
- Smoker: A binary indicator for whether the individual is a smoker (1 for yes, 0 for no).
- BMI: The standardized BMI value.

- Smoker^2: The square of the smoker indicator, which will be identical to the smoker indicator since it's binary (1 for smokers, 0 for non-smokers).
- Smoker BMI: An interaction term between the smoker status and standardized BMI.
- BMI^2: The square of the standardized BMI.

Why not?

- Redundancy in features such as `smoker^2` = `smoker`, due to it being a categorical variable
- Harder to interpret
- Risk of overfitting

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.

[132]:
```
# Create Pipeline for model that includes polynomial expansion of bmi
poly_pipeline = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3, include_bias=False,␣
 ↪interaction_only=False)),
    ("scaler", StandardScaler()),  # Standardize the features after polynomial␣
 ↪transformation
    ("linear_regression", LinearRegression())
])

# Select bmi and transform it into a 2D array for the pipeline
X_train_bmi_poly = X_train[['bmi']].values
```

[133]:
```
# Train and compute and plot fitted values
poly_pipeline.fit(X_train_bmi_poly, y_train)

# Compute the fitted values for plotting
X_full_bmi_poly = df_insurance[['bmi']].values
y_fitted_poly = poly_pipeline.predict(X_full_bmi_poly)

# Compute performance metrics
y_pred_poly = poly_pipeline.predict(X_train_bmi_poly)
poly_r2 = r2_score(y_train, y_pred_poly)
poly_mse = mean_squared_error(y_train, y_pred_poly)
poly_rmse = np.sqrt(poly_mse)

poly_r2, poly_mse, poly_mse
```

[133]:
```
(0.03678011073620091, 138093127.52599996, 138093127.52599996)
```

Insanely bad fit!!

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?

```
[ ]:
```

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.

```
[134]: 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 for grid search
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)
```

[135]:
```
#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,

[136]:
```
print("best index: ", grid_search.best_index_)
print("best param: ", grid_search.best_params_)
print("best score: ", grid_search.best_score_)
```

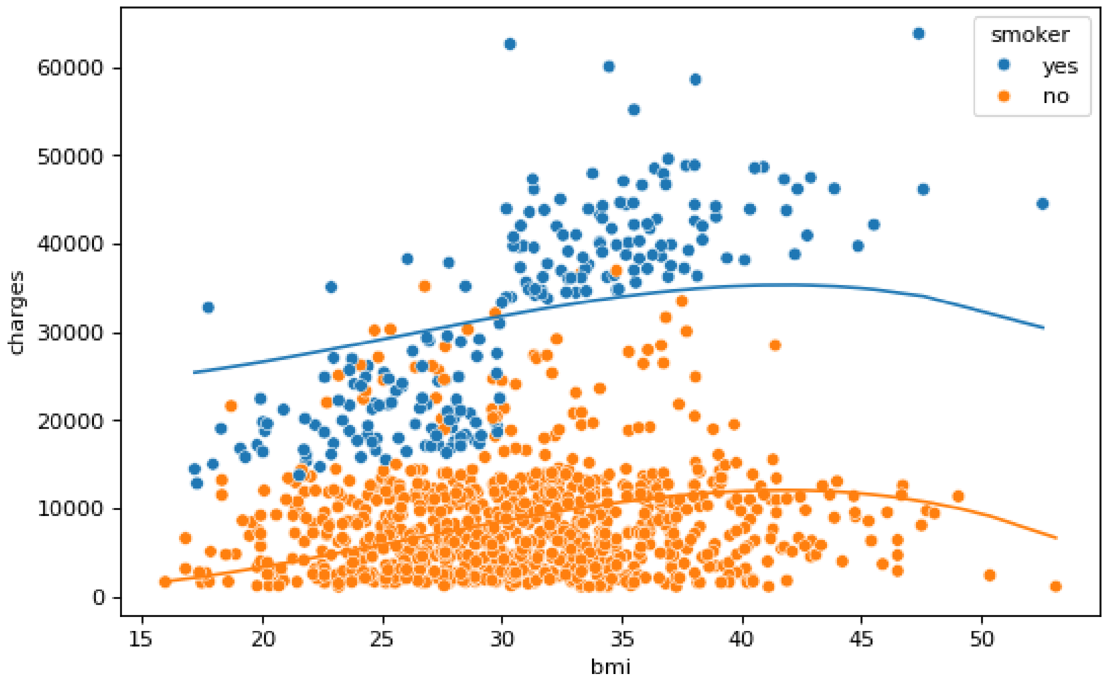
```
best index:  2
best param:  {'pre_processing__poly__degree': 3}
best score:  -50713368.324627556
```

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.

[137]:
```
grid_search.best_estimator_['model'].coef_
```

[137]:
```
array([ 2.32451692e+04, -9.00772163e+02,  5.54455069e+01, -7.12462949e-01])
```

[138]:
```
# 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.

[139]:
```
cv_results = pd.DataFrame(grid_search.cv_results_)
cv_results
```

[139]:
```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       0.001870      0.000344         0.000956        0.000067   
1       0.001815      0.000203         0.001050        0.000116   
2       0.001864      0.000189         0.001042        0.000089   
3       0.001737      0.000235         0.000934        0.000087   
4       0.001874      0.000363         0.001006        0.000119   
5       0.001716      0.000096         0.000941        0.000136   
6       0.001747      0.000129         0.000894        0.000032   
7       0.001865      0.000224         0.001034        0.000135   
8       0.001699      0.000127         0.000899        0.000047   

  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}   
```

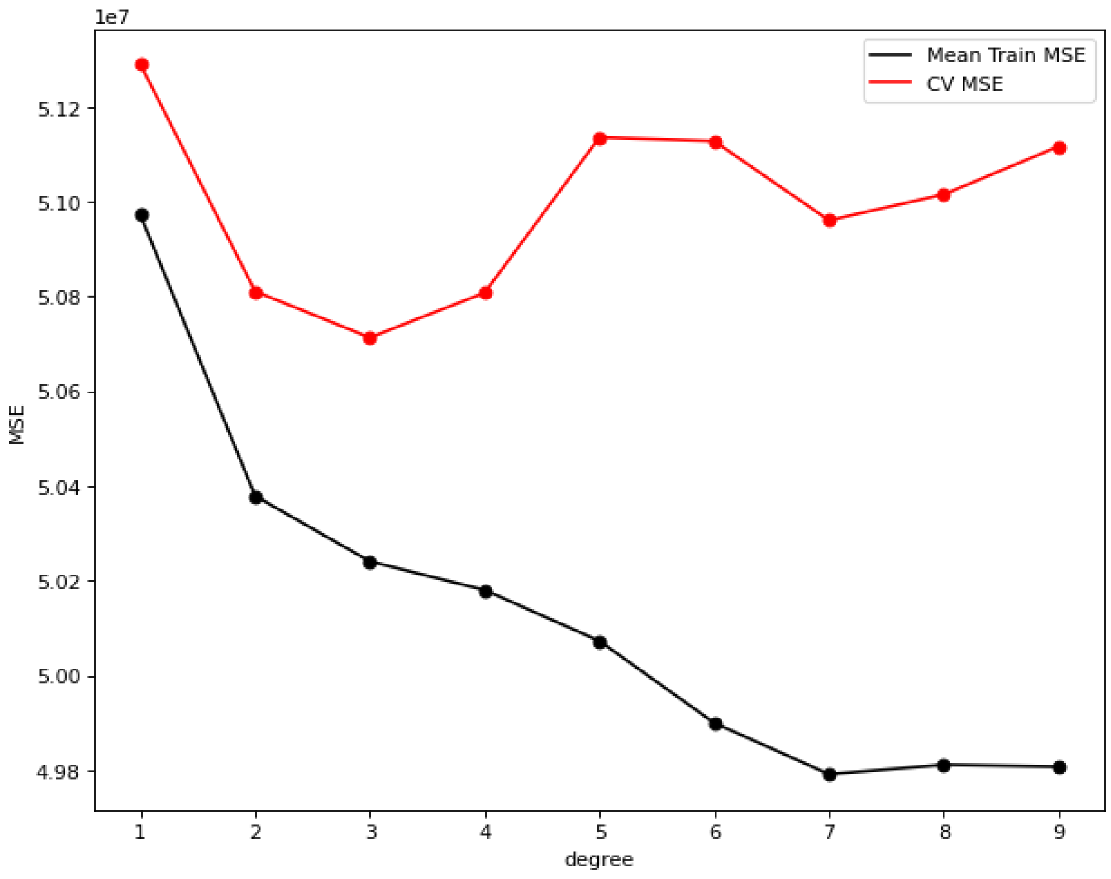
```
6                                  7  {'pre_processing__poly__degree': 7}
7                                  8  {'pre_processing__poly__degree': 8}
8                                  9  {'pre_processing__poly__degree': 9}

   split0_test_score  split1_test_score  split2_test_score  split3_test_score  \
0      -4.161891e+07      -5.049249e+07      -4.885489e+07      -6.113953e+07
1      -4.099657e+07      -4.984110e+07      -4.757841e+07      -6.217477e+07
2      -4.076527e+07      -4.991759e+07      -4.719010e+07      -6.205974e+07
3      -4.063586e+07      -4.977325e+07      -4.778834e+07      -6.213996e+07
4      -4.070776e+07      -4.969903e+07      -4.934777e+07      -6.231684e+07
5      -4.078963e+07      -5.009320e+07      -4.855567e+07      -6.196867e+07
6      -4.115669e+07      -5.011739e+07      -4.786815e+07      -6.169863e+07
7      -4.100641e+07      -5.036572e+07      -4.793709e+07      -6.173378e+07
8      -4.101049e+07      -5.054951e+07      -4.822266e+07      -6.171403e+07

   …  mean_test_score  std_test_score  rank_test_score  split0_train_score  \
0  …    -5.129065e+07    6.425836e+06                9       -5.338675e+07
1  …    -5.081082e+07    6.984028e+06                3       -5.282341e+07
2  …    -5.071337e+07    7.060914e+06                1       -5.271939e+07
3  …    -5.080784e+07    7.077410e+06                2       -5.270462e+07
4  …    -5.113588e+07    7.001897e+06                8       -5.259760e+07
5  …    -5.112809e+07    6.953168e+06                7       -5.237924e+07
6  …    -5.096073e+07    6.792780e+06                4       -5.216962e+07
7  …    -5.101541e+07    6.842129e+06                5       -5.222323e+07
8  …    -5.111674e+07    6.811154e+06                6       -5.222370e+07

   split1_train_score  split2_train_score  split3_train_score  \
0       -5.115498e+07       -5.162161e+07       -4.851776e+07
1       -5.058530e+07       -5.122538e+07       -4.757679e+07
2       -5.041279e+07       -5.115972e+07       -4.742651e+07
3       -5.040524e+07       -5.102245e+07       -4.735898e+07
4       -5.040446e+07       -5.077226e+07       -4.724254e+07
5       -5.012938e+07       -5.073188e+07       -4.722886e+07
6       -4.999847e+07       -5.066309e+07       -4.716211e+07
7       -4.999267e+07       -5.068604e+07       -4.718219e+07
8       -4.999103e+07       -5.067565e+07       -4.718823e+07

   split4_train_score  mean_train_score  std_train_score
0       -5.018101e+07     -5.097242e+07     1.608444e+06
1       -4.968302e+07     -5.037878e+07     1.736133e+06
2       -4.948393e+07     -5.024047e+07     1.761277e+06
3       -4.940866e+07     -5.017999e+07     1.771289e+06
4       -4.934540e+07     -5.007245e+07     1.761438e+06
5       -4.902782e+07     -4.989944e+07     1.719707e+06
6       -4.896236e+07     -4.979113e+07     1.676848e+06
7       -4.897068e+07     -4.981096e+07     1.687266e+06
8       -4.895663e+07     -4.980705e+07     1.685819e+06
```

```
[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.

[140]:
```
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 (EXTRA)

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

```
[ ]:
```

6.4.4 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.

```
[142]: !jupyter nbconvert --to pdf mlp_week04.ipynb
```

```
[NbConvertApp] Converting notebook mlp_week04.ipynb to pdf
[NbConvertApp] Support files will be in mlp_week04_files/
[NbConvertApp] Making directory ./mlp_week04_files
[NbConvertApp] Writing 99269 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 774735 bytes to mlp_week04.pdf
```

```
[ ]:
```