8. Regression

COMP3314
Machine Learning

Outline

- In this chapter, we will discuss the main concepts of regression models and cover the following topics
  - Exploring and visualizing datasets
  - Looking at different approaches to implement linear regression models
  - Training regression models that are robust to outliers
  - Evaluating regression models and diagnosing common problems
  - Fitting regression models to nonlinear data

Introducing Linear Regression

- The goal of linear regression is to model the relationship between one or multiple features and a continuous target variable
  - In contrast to classification—a different subcategory of supervised learning—regression analysis aims to predict outputs on a continuous scale rather than categorical class labels
- In the following subsections, you will be introduced to the most basic type of linear regression, simple linear regression, and understand how to relate it to the more general, multivariate case (linear regression with multiple features)

Simple Linear Regression

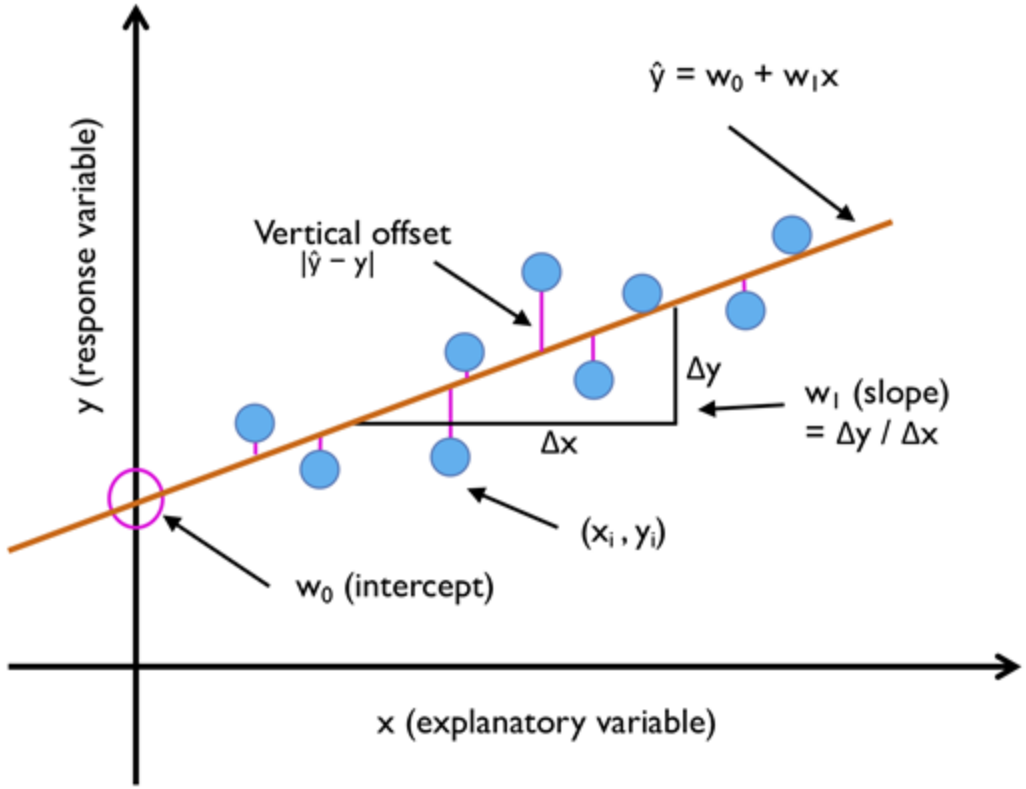
- The goal of simple (univariate) linear regression is to model the relationship between a single feature (explanatory variable, x) and a continuous-valued target (response variable, y)
- The equation of a linear model with one explanatory variable is defined as follows

$$y = w_0 + w_1 x$$

- Here, the weight, w_0, represents the y axis intercept and w_1 is the weight coefficient of the explanatory variable
- Our goal is to learn the weights of the linear equation to describe the relationship between the explanatory variable and the target variable, which can then be used to predict the responses of new explanatory variables that were not part of the training dataset

Simple Linear Regression

- Linear regression can be understood as finding the best-fitting straight line through the training examples
  - The orange best-fitting line is aka regression line
  - The pink vertical lines are aka residual errors
    - I.e. the errors of our prediction



Multiple Linear Regression

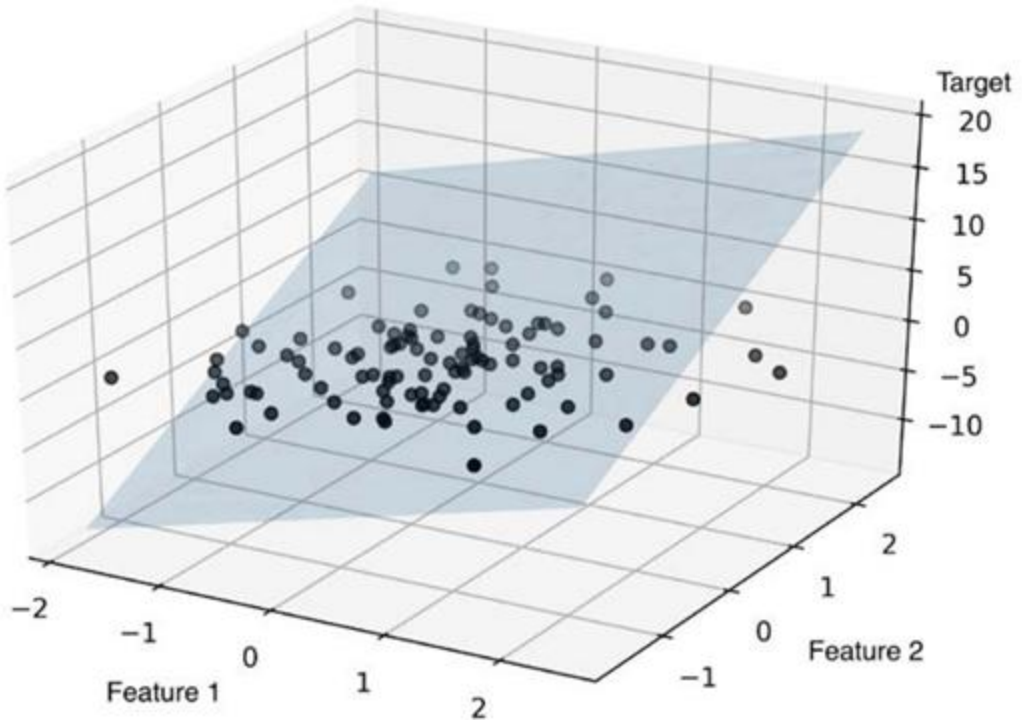
- Simple linear regression is a special case of multiple linear regression
  - We can also generalize the linear regression model to multiple explanatory variables; this process is called multiple linear regression

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{i=0}^{n} w_i x_i = w^T x$$

  - Here, w_0 is the y axis intercept with $x_0 = 1$

Multiple Linear Regression

- The following figure shows how the two-dimensional, fitted hyperplane of a multiple linear regression model with two features could look



Housing Dataset

- Before we implement the first linear regression model, let's discuss a new dataset
- The Housing dataset contains information about houses in the suburbs of Boston collected by D. Harrison and D.L. Rubinfeld in 1978
- The dataset is available here
- The 506 samples of the housing dataset have the following features
  - CRIM: Per capita crime rate by town
  - ZN: Proportion of residential land zoned for lots over 25,000 sq. ft.
  - INDUS: Proportion of non-retail business acres per town
  - CHAS: Charles River dummy variable
    - = 1 if tract bounds river and 0 otherwise
  - NOX: Nitric oxide concentration (parts per 10 million)
  - RM: Average number of rooms per dwelling
  - AGE: Proportion of owner-occupied units built prior to 1940

Housing Dataset

- DIS: Weighted distances to five Boston employment centers
- RAD: Index of accessibility to radial highways
- TAX: Full-value property tax rate per $10,000
- PTRATIO: Pupil-teacher ratio by town
- B: $1000(Bk - 0.63)^2$, where Bk is the proportion of [people of African American descent] by town
- LSTAT: Percentage of lower status of the population
- MEDV: Median value of owner-occupied homes in $1000s

- For the rest of this chapter, we will regard the house prices (MEDV) as our target variable
  - I.e., the variable that we want to predict using one or more of the 13 explanatory variables

Housing Dataset

- In the following subsection, we are going to focus on the following subset of features only
  - LSTAT: Percentage of lower status
  - INDUS: Proportion of non-retail business
  - NOX: Nitric oxide concentration
  - RM: Average number of rooms
  - MEDV: Median value

Code - Regression.ipynb

- Available here on CoLab

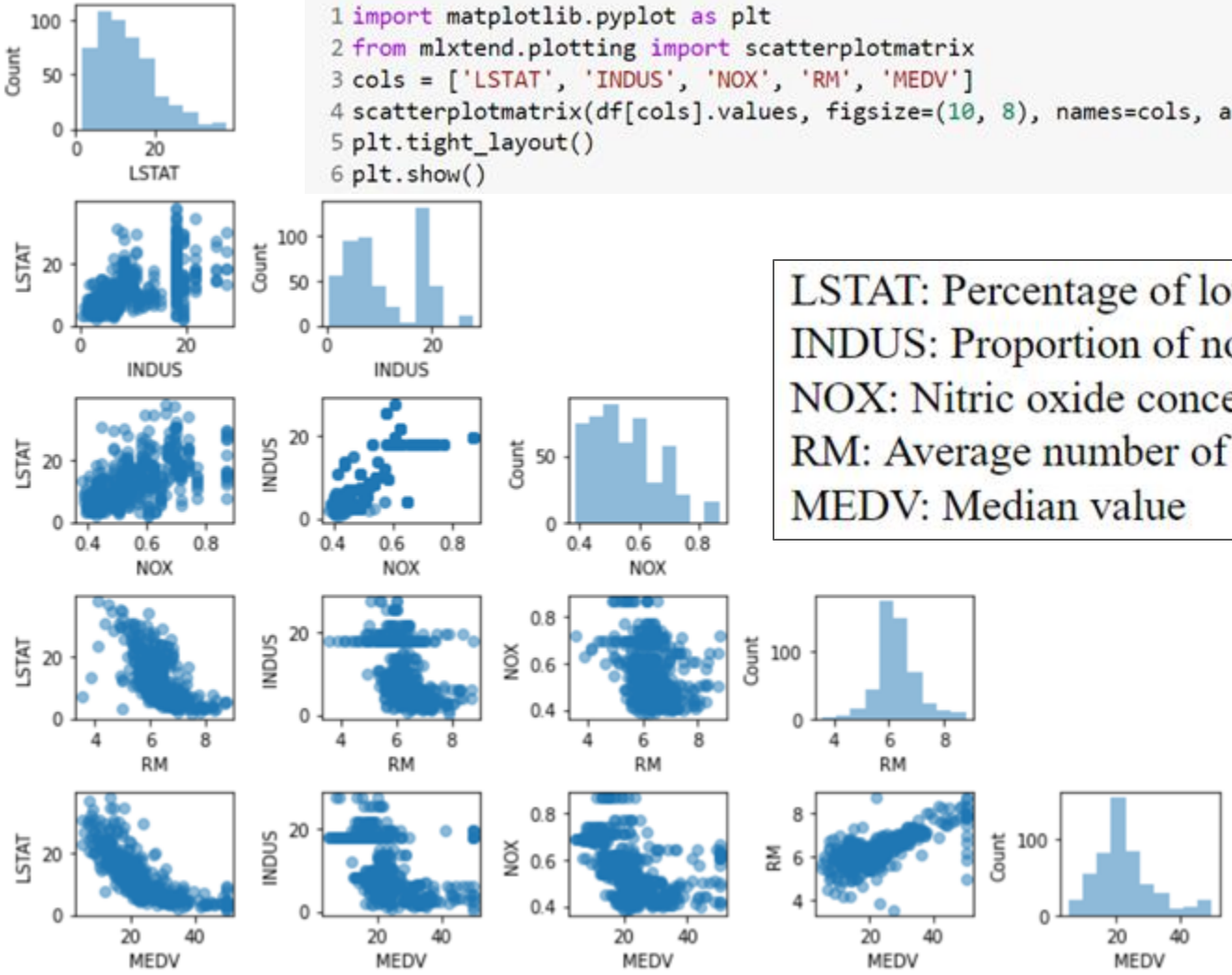
```
import pandas as pd
df = pd.read_csv('https://raw.githubusercontent.com/rasbt/'
                 'python-machine-learning-book-3rd-edition/'
                 'master/ch10/housing.data.txt',
                 header=None,
                 sep='\s+')
df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
              'NOX', 'RM', 'AGE', 'DIS', 'RAD',
              'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
df.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

Exploratory Data Analysis (EDA)

- An important first step prior to the training of a machine learning model
- We will now use some simple yet useful techniques from the graphical EDA toolbox that may help us to visually detect the presence of outliers, the distribution of the data, and the relationships between features
- First, we will create a scatterplot matrix that allows us to visualize the pair-wise correlations between the different features in this dataset in one place
- To plot the scatterplot matrix, we will use the scatterplotmatrix function from the MLxtend library
    - A Python library that contains various convenience functions for machine learning and data science applications in Python

```
1 import matplotlib.pyplot as plt
2 from mlxtend.plotting import scatterplotmatrix
3 cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
4 scatterplotmatrix(df[cols].values, figsize=(10, 8), names=cols, alpha=0.5)
5 plt.tight_layout()
6 plt.show()
```



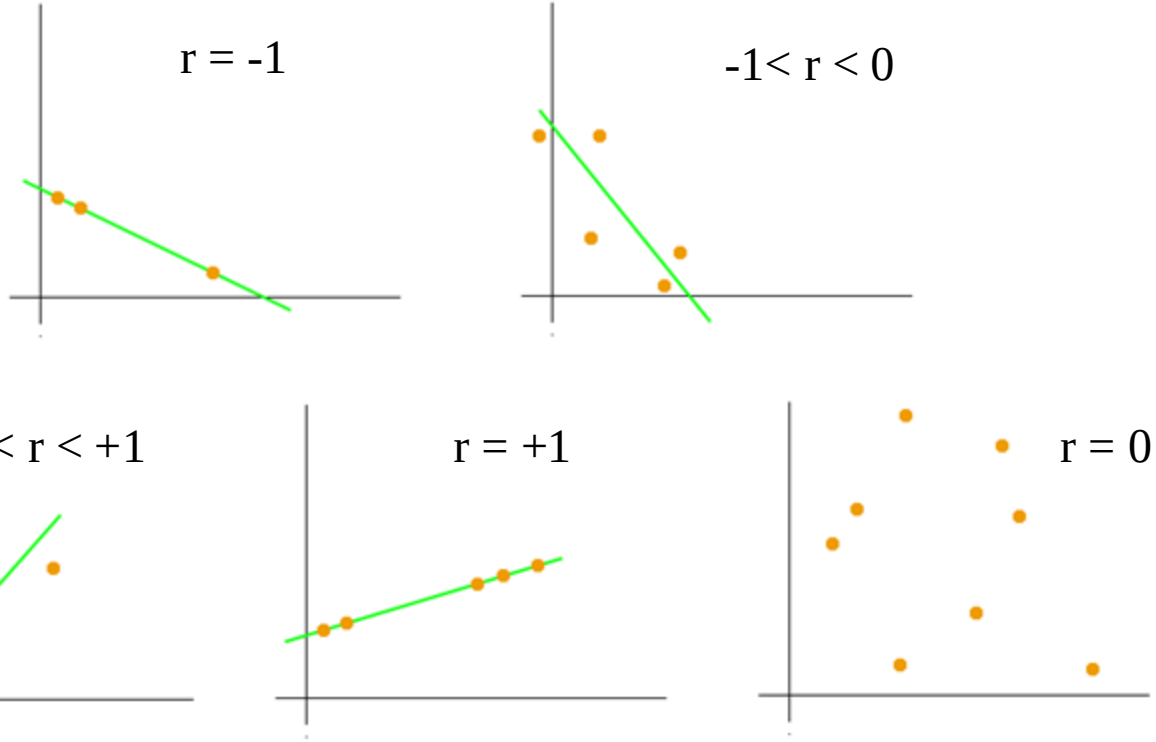
LSTAT: Percentage of lower status
INDUS: Proportion of non-retail business
NOX: Nitric oxide concentration
RM: Average number of rooms
MEDV: Median value

Correlation Matrix

- Let’s create a correlation matrix to quantify linear relationships between variables
- The correlation matrix is a square matrix that contains the Pearson product-moment correlation coefficient (often abbreviated as Pearson's r), which measures the linear dependence between pairs of features
- The correlation coefficients are in the range -1 to 1
  - Two features have
    - a perfect positive correlation if r = 1
    - no correlation if r = 0
    - a perfect negative correlation if r = -1

Example: Pearson's r

- Examples of scatter diagrams with different values of correlation coefficient (r)



Correlation Matrix

- Pearson's correlation coefficient can simply be calculated as the covariance between two features, x and y (numerator), divided by the product of their standard deviations (denominator)

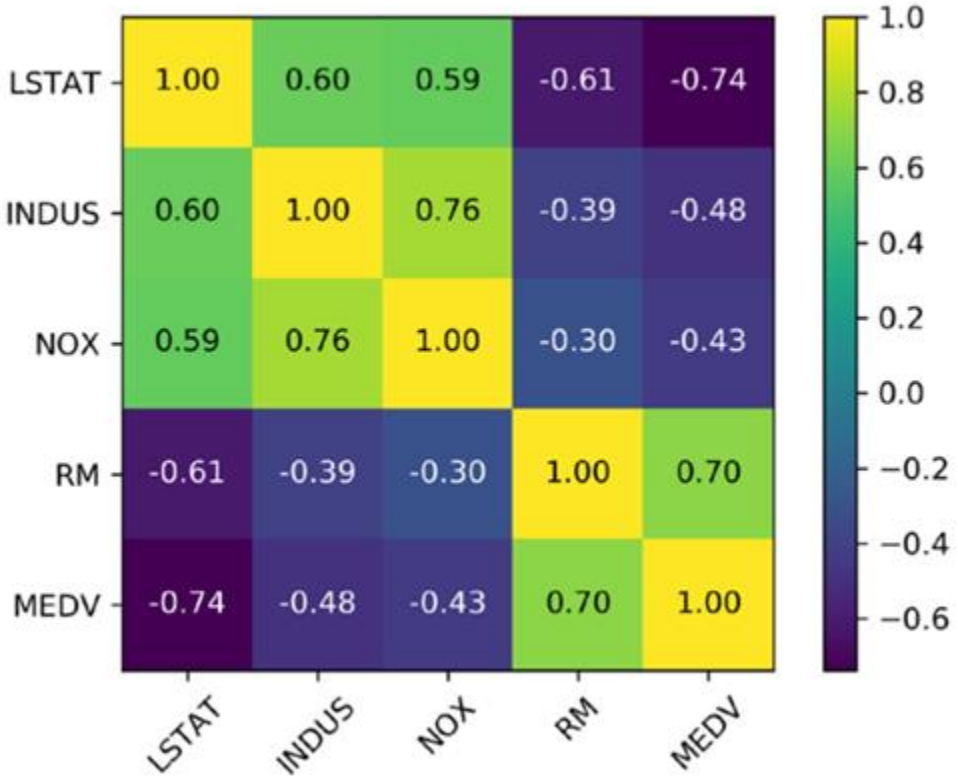
$$r = \frac{\sum_{i=1}^{n}\left[\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)\right]}{\sqrt{\sum_{i=1}^{n}\left(x^{(i)} - \mu_x\right)^2}\sqrt{\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

- Here, μ denotes the mean of the corresponding feature, σ_{xy} is the covariance between the features x and y, and σ_x and σ_y are the features' standard deviations

```
1 import numpy as np
2 cm = np.corrcoef(df[cols].values.T)
3 print(cm)
```

```
[[ 1.          0.60379972  0.59087892 -0.61380827 -0.73766273]
 [ 0.60379972  1.          0.76365145 -0.39167585 -0.48372516]
 [ 0.59087892  0.76365145  1.         -0.30218819 -0.42732077]
 [-0.61380827 -0.39167585 -0.30218819  1.          0.69535995]
 [-0.73766273 -0.48372516 -0.42732077  0.69535995  1.        ]]
```

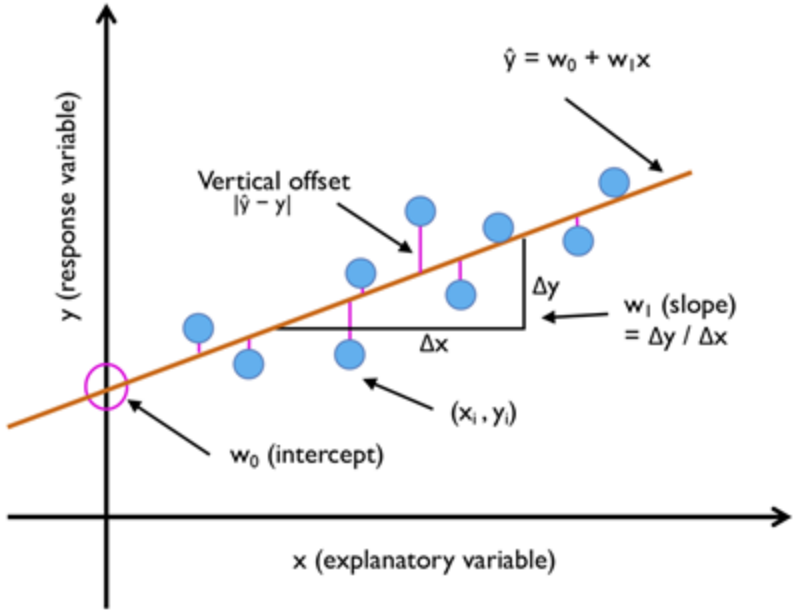
LSTAT: Percentage of lower status
INDUS: Proportion of non-retail business
NOX: Nitric oxide concentration
RM: Average number of rooms
MEDV: Median value



Ordinary Least Squares (OLS)

- Estimates the parameters of the linear regression line that minimizes the sum of the squared distances (residuals or errors) to the training examples
- Aka linear least squares

$$y = w_0 + w_1 x$$



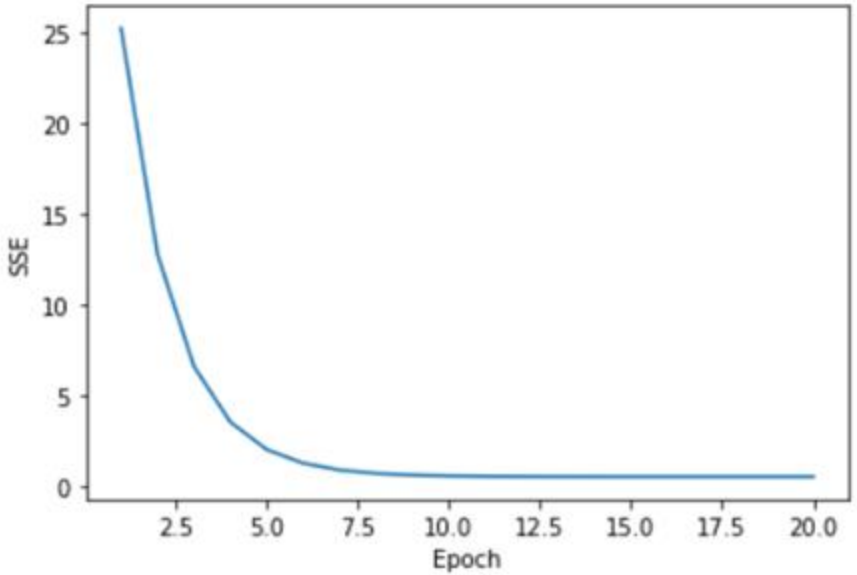
Ordinary Least Squares - Cost Function

- Recall our implementation of the Adaptive Linear Neuron (Adaline)
  - You will remember that the artificial neuron uses a linear activation function. Also, we defined a cost function, J(w), which we minimized to learn the weights via optimization algorithms, such as gradient descent (GD) and stochastic gradient descent (SGD)
  - This cost function in Adaline is the sum of squared errors (SSE), which is identical to the cost function that we use for OLS

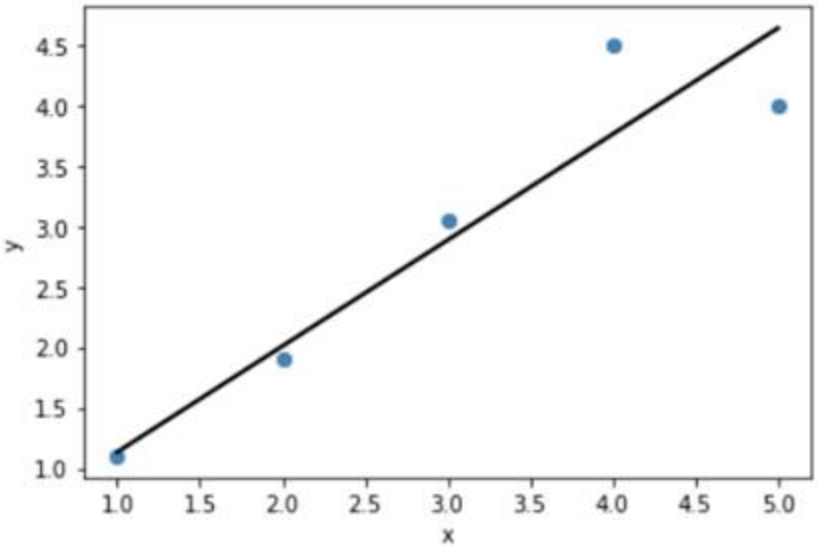
$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

```
class LinearRegressionGD(object):
    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter
    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self
    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]
    def predict(self, X):
        return self.net_input(X)
```

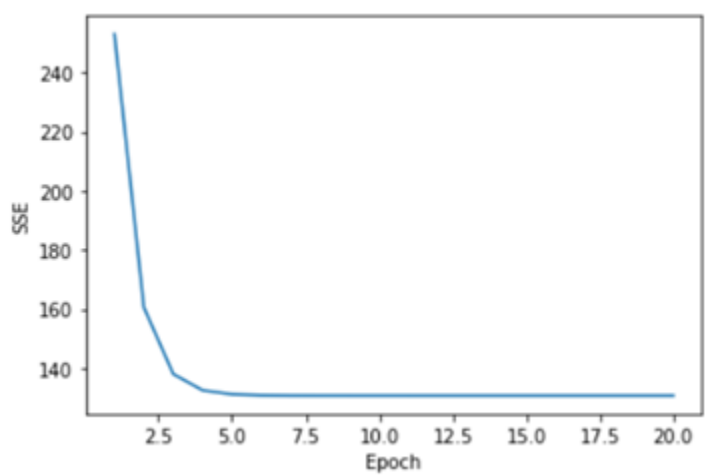
```
1 X = np.array([[1], [2], [3], [4], [5]])
2 y = np.array([1.1, 1.9, 3.05, 4.5, 4])
3 lr = LinearRegressionGD(eta=0.005).fit(X, y)
4 plt.plot(range(1, lr.n_iter+1), lr.cost_)
5 plt.ylabel('SSE')
6 plt.xlabel('Epoch')
7 plt.show()
```



```
def lin_regplot(X, y, model):
    plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
    plt.plot(X, model.predict(X), color='black', lw=2)
    return
lin_regplot(X, y, lr)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

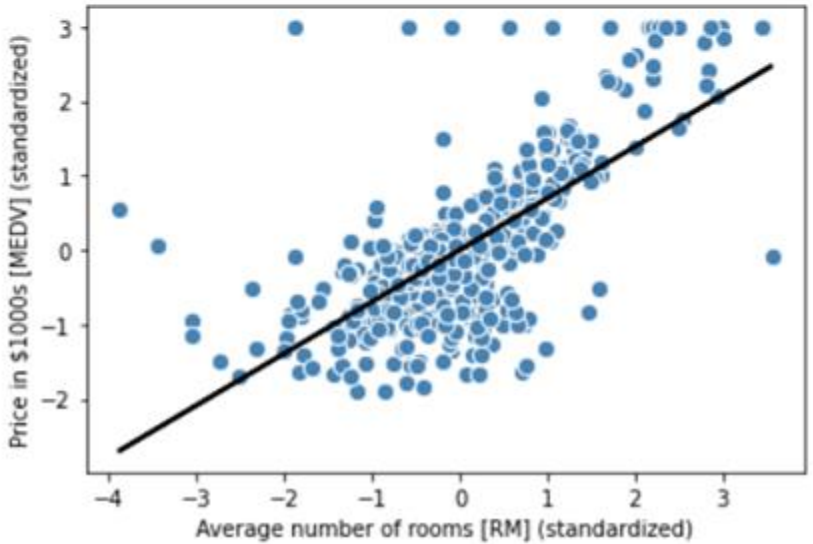


```
X = df[['RM']].values
y = df['MEDV'].values
from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X_std = sc_x.fit_transform(X)
y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()
lr = LinearRegressionGD()
lr.fit(X_std, y_std)
plt.plot(range(1, lr.n_iter+1), lr.cost_)
plt.ylabel('SSE')
plt.xlabel('Epoch')
plt.show()
```



```
def lin_regplot(X, y, model):
    plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
    plt.plot(X, model.predict(X), color='black', lw=2)
    return
```

```
lin_regplot(X_std, y_std, lr)
plt.xlabel('Average number of rooms [RM] (standardized)')
plt.ylabel('Price in $1000s [MEDV] (standardized)')
plt.show()
```



```
1 num_rooms_std = sc_x.transform(np.array([[5.0]]))
2 price_std = lr.predict(num_rooms_std)
3 print("Price in $1000s: %.3f" % sc_y.inverse_transform(price_std))
```

```
Price in $1000s: 10.840
```

```
1 print('Slope: %.3f' % lr.w_[1])
2 print('Intercept: %.3f' % lr.w_[0])
```

```
Slope: 0.695
Intercept: -0.000
```

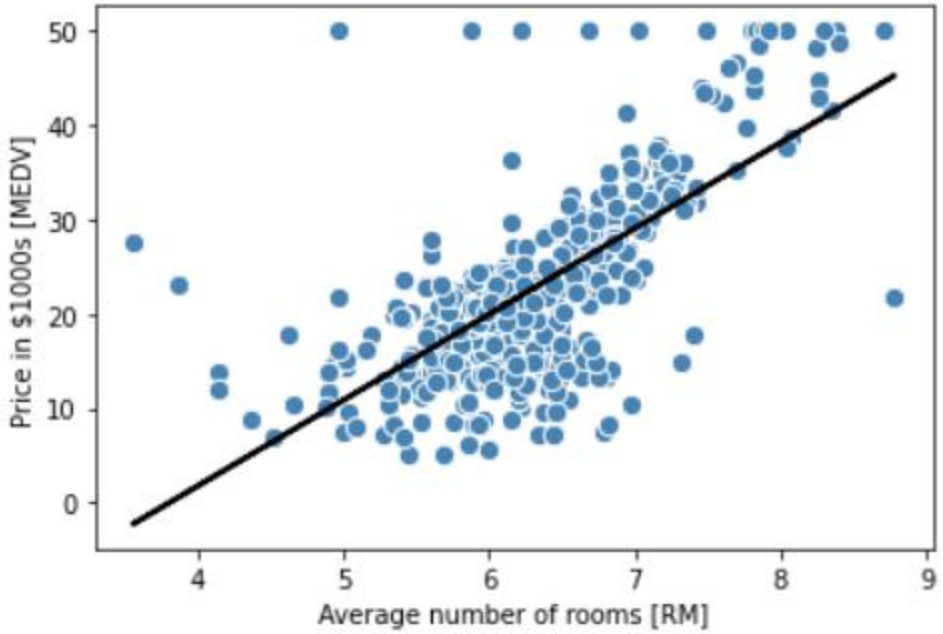
Regression via Scikit-Learn

- More efficient implementation for regression estimators make use of the least squares implementation in SciPy (scipy.linalg.lstsq), which in turn uses highly optimized code optimizations based on the Linear Algebra Package (LAPACK)
- The linear regression implementation in scikit-learn also works (better) with unstandardized variables, since it does not use (S)GD-based optimization, so we can skip the standardization step

```
1 from sklearn.linear_model import LinearRegression
2 slr = LinearRegression()
3 slr.fit(X, y)
4 y_pred = slr.predict(X)
5 print('Slope: %.3f' % slr.coef_[0])
6 print('Intercept: %.3f' % slr.intercept_)
```

```
Slope: 9.102
Intercept: -34.671
```

```
1 lin_regplot(X, y, slr)
2 plt.xlabel('Average number of rooms [RM]')
3 plt.ylabel('Price in $1000s [MEDV]')
4 plt.show()
```



Linear Regression - Analytical Solutions

- As an alternative to using machine learning libraries, there is also a closed-form solution for solving OLS involving a system of linear equations that can be found in most introductory statistics textbooks

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

```
# adding a column vector of "ones"
Xb = np.hstack((np.ones((X.shape[0], 1)), X))
w = np.zeros(X.shape[1])
z = np.linalg.inv(np.dot(Xb.T, Xb))
w = np.dot(z, np.dot(Xb.T, y))

print('Slope: %.3f' % w[1])
print('Intercept: %.3f' % w[0])
```

```
Slope: 9.102
Intercept: -34.671
```

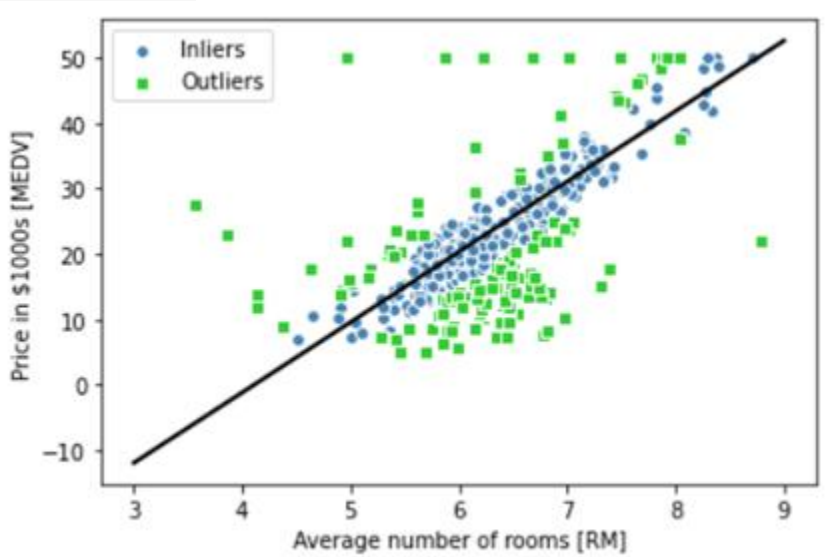
Robust Regression - RANSAC

- Linear regression models can be heavily impacted by the presence of outliers
  - In certain situations, a very small subset of our data can have a big effect on the estimated model coefficients
- There are many statistical tests that can be used to detect outliers
  - However, removing outliers always requires our own judgment as data scientists as well as our domain knowledge
- We will now look at a robust method of regression using the RANdom SAmple Consensus (RANSAC) algorithm, which automatically fits a regression model to a subset of the data, the so-called inliers

The RANSAC Algorithm

1. Select the number of samples to fit the model
   - This is often chosen as the minimal number of samples necessary to estimate the given estimator
2. Test all other data points against the fitted model and add those points that fall within a user-given tolerance to the inliers
3. Refit the model using all inliers
4. Estimate the error of the fitted model versus the inliers
5. Terminate the algorithm if the performance meets a certain user-defined threshold or if a fixed number of iterations were reached; go back to step 1 otherwise

```
from sklearn.linear_model import RANSACRegressor
ransac = RANSACRegressor(LinearRegression(), max_trials=100,
                         min_samples=2, loss='absolute_loss',
                         residual_threshold=5.0, random_state=0)
ransac.fit(X, y)
inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)
line_X = np.arange(3, 10, 1)
line_y_ransac = ransac.predict(line_X[:, np.newaxis])
plt.scatter(X[inlier_mask], y[inlier_mask],
            c='steelblue', edgecolor='white',
            marker='o', label='Inliers')
plt.scatter(X[outlier_mask], y[outlier_mask],
            c='limegreen', edgecolor='white',
            marker='s', label='Outliers')
plt.plot(line_X, line_y_ransac, color='black', lw=2)
plt.xlabel('Average number of rooms [RM]')
plt.ylabel('Price in $1000s [MEDV]')
plt.legend(loc='upper left')
plt.show()
```



Robust Regression - RANSAC

```
1 print('Slope: %.3f' % ransac.estimator_.coef_[0])
2 print('Intercept: %.3f' % ransac.estimator_.intercept_)
```

```
Slope: 10.735
Intercept: -44.089
```

- Using RANSAC, we reduced the potential effect of the outliers in this dataset
  - However, we don't know whether this approach will have a positive effect on the predictive performance for unseen data or not
- In the next section, we will look at different approaches for evaluating a regression model, which is a crucial part of building systems for predictive modeling

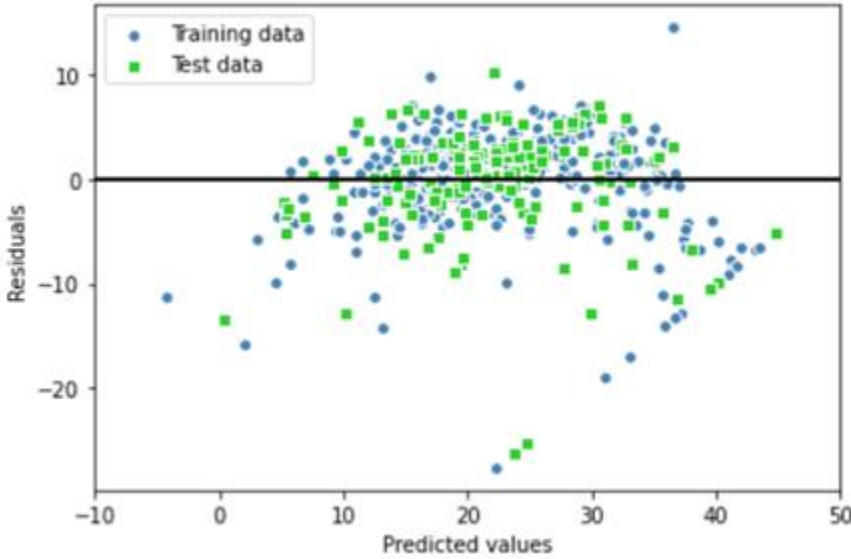
Performance Evaluation

- As we know, it is crucial to test the model on data that it hasn't seen during training to obtain a more unbiased estimate of its generalization performance
- We will now use all variables in the dataset and train a multiple regression model

```
from sklearn.model_selection import train_test_split
X = df.iloc[:, :-1].values
y = df['MEDV'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
slr = LinearRegression()
slr.fit(X_train, y_train)
y_train_pred = slr.predict(X_train)
y_test_pred = slr.predict(X_test)
```

Residual Plot

```
plt.scatter(y_train_pred,  y_train_pred - y_train,
            c='steelblue', marker='o', edgecolor='white',
            label='Training data')
plt.scatter(y_test_pred,  y_test_pred - y_test,
            c='limegreen', marker='s', edgecolor='white',
            label='Test data')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0, xmin=-10, xmax=50,
           color='black', lw=2)
plt.xlim([-10, 50])
plt.tight_layout()
plt.show()
```



Mean Squared Error (MSE)

- Another useful quantitative measure of a model's performance is the so-called mean squared error (MSE)
  - Simply the averaged value of the SSE cost that we minimized to fit the linear regression model
- The MSE is useful for comparing different regression models or for tuning their parameters via grid search and cross-validation, as it normalizes the SSE by the sample size

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

```
1 from sklearn.metrics import mean_squared_error
2 print('MSE train: %.3f, test: %.3f' % (
3         mean_squared_error(y_train, y_train_pred),
4         mean_squared_error(y_test, y_test_pred)))
5
```

```
MSE train: 19.958, test: 27.196
```

Coefficient of Determination

- It may sometimes be more useful to report the coefficient of determination
  - R^2 value
- The R^2 value can be understood as a standardized version of the MSE, for better interpretability of the model's performance
- The R^2 value is defined as

$$R^2 = 1 - \frac{SSE}{SST}$$

- Here, SSE is the sum of squared errors and SST is the total sum of squares

$$SST = \sum_{i=1}^{n} \left(y^{(i)} - \mu_y\right)^2$$

R2 Value

- Let's quickly show that R^2 is indeed just a rescaled version of the MSE

$$
\begin{aligned}
R^2 &= 1 - \frac{SSE}{SST} \\
&= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2} \\
&= 1 - \frac{MSE}{Var(y)}
\end{aligned}
$$

R2 Value

```
1 from sklearn.metrics import r2_score
2 print('R^2 train: %.3f, test: %.3f' % (
3         r2_score(y_train, y_train_pred),
4         r2_score(y_test, y_test_pred)))
```

```
R^2 train: 0.765, test: 0.673
```

Regularization for Regression

- As we know, regularization is one approach to tackling the problem of overfitting
  - Recall that it works by shrinking the parameter values of the model to induce a penalty against complexity
- The most popular approaches to regularized linear regression are the so-called
  - Ridge Regression
  - Least absolute shrinkage / Selection operator (LASSO)
  - Elastic Net

Regularization - Ridge Regression

- Ridge Regression is an L2 penalized model where we simply add the squared sum of the weights to our least-squares cost function

$$J(\boldsymbol{w})_{Ridge} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \|\boldsymbol{w}\|_2^2$$

$$L2: \lambda \|\boldsymbol{w}\|_2^2 = \lambda \sum_{j=1}^{m} {w_j}^2$$

Regularization - LASSO

- An alternative approach that can lead to sparse models is LASSO
- Depending on the regularization strength, certain weights can become zero, which also makes LASSO useful as a supervised feature selection technique

$$J(\boldsymbol{w})_{LASSO} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \|\boldsymbol{w}\|_1$$

$$L1: \lambda \|\boldsymbol{w}\|_1 = \lambda \sum_{j=1}^{m} |w_j|$$

Elastic Net

- A compromise between Ridge Regression and LASSO is elastic net, which has an L1 penalty to generate sparsity and an L2 penalty such that it can be used for selecting more than n features if m > n

$$J(\boldsymbol{w})_{ElasticNet} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_1 \sum_{j=1}^{m} {w_j}^2 + \lambda_2 \sum_{j=1}^{m} |w_j|$$

Regularization for Regression in Sk-learn

- Those regularized regression models are all available via scikit-learn

```
1 from sklearn.linear_model import Ridge
2 ridge = Ridge(alpha=1.0)
```

```
1 from sklearn.linear_model import Lasso
2 lasso = Lasso(alpha=1.0)
```

```
1 from sklearn.linear_model import ElasticNet
2 elanet = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

Polynomial Regression

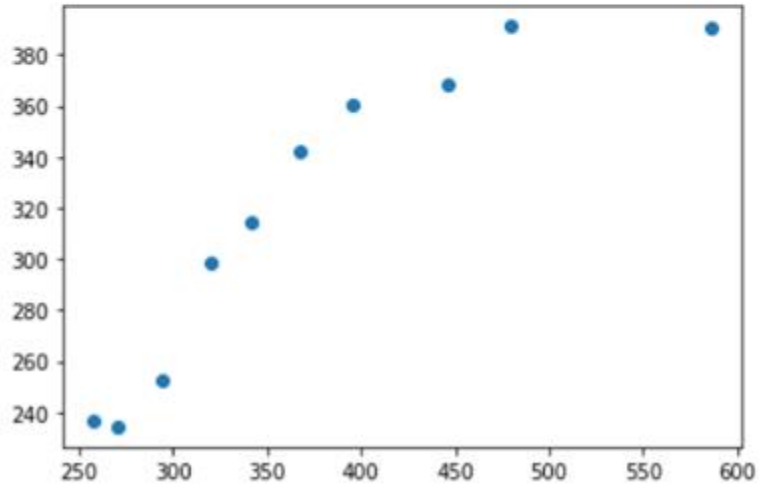
- In the previous sections, we assumed a linear relationship between explanatory and response variables
- One way to account for the violation of linearity assumption is to use a polynomial regression model by adding polynomial terms

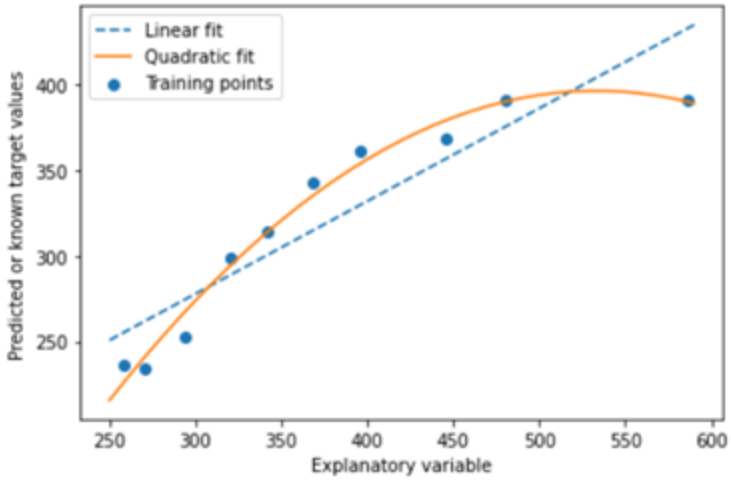
$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d$$

- Here, d denotes the degree of the polynomial
- In the following subsections, we will see how we can add such polynomial terms to an existing dataset conveniently and fit a polynomial regression model

```
X = np.array([258.0, 270.0, 294.0,
              320.0, 342.0, 368.0,
              396.0, 446.0, 480.0, 586.0])\
             [:, np.newaxis]

y = np.array([236.4, 234.4, 252.8,
              298.6, 314.2, 342.2,
              360.8, 368.0, 391.2,
              390.8])
plt.scatter(X, y)
plt.show()
```






```
from sklearn.preprocessing import PolynomialFeatures
lr = LinearRegression()
pr = LinearRegression()
quadratic = PolynomialFeatures(degree=2)
X_quad = quadratic.fit_transform(X)
lr.fit(X, y)
X_fit = np.arange(250, 600, 10)[:, np.newaxis]
y_lin_fit = lr.predict(X_fit)
pr.fit(X_quad, y)
y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
plt.scatter(X, y, label='Training points')
plt.plot(X_fit, y_lin_fit, label='Linear fit', linestyle='--')
plt.plot(X_fit, y_quad_fit, label='Quadratic fit')
plt.xlabel('Explanatory variable')
plt.ylabel('Predicted or known target values')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

```
y_lin_pred = lr.predict(X)
y_quad_pred = pr.predict(X_quad)
print('Training MSE linear: %.3f, quadratic: %.3f' % (
        mean_squared_error(y, y_lin_pred),
        mean_squared_error(y, y_quad_pred)))
print('Training R^2 linear: %.3f, quadratic: %.3f' % (
        r2_score(y, y_lin_pred),
        r2_score(y, y_quad_pred)))
```

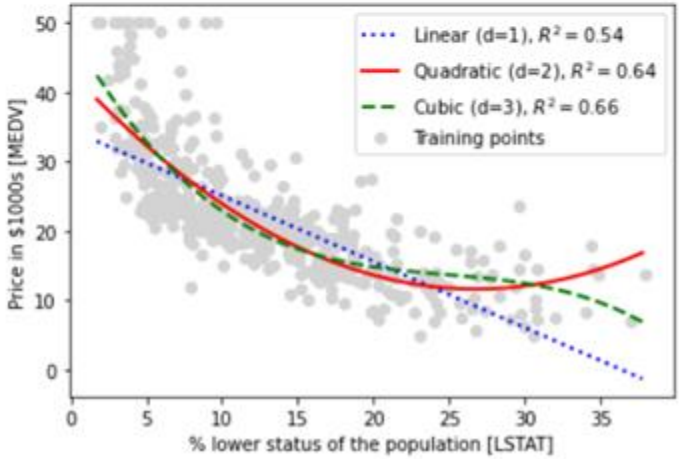
```
Training MSE linear: 569.780, quadratic: 61.330
Training R^2 linear: 0.832, quadratic: 0.982
```

Nonlinear Relationships in the Housing Dataset

- Let’s now model the relationship between house prices and LSTAT (percentage of lower status of the population) using second-degree (quadratic) and third-degree (cubic) polynomials
- We will then compare that to a linear fit

```
X = df[['LSTAT']].values
y = df['MEDV'].values
regr = LinearRegression()
# create quadratic features
quadratic = PolynomialFeatures(degree=2)
cubic = PolynomialFeatures(degree=3)
X_quad = quadratic.fit_transform(X)
X_cubic = cubic.fit_transform(X)
# fit features
X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]
regr = regr.fit(X, y)
y_lin_fit = regr.predict(X_fit)
linear_r2 = r2_score(y, regr.predict(X))
regr = regr.fit(X_quad, y)
y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
quadratic_r2 = r2_score(y, regr.predict(X_quad))
regr = regr.fit(X_cubic, y)
y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
cubic_r2 = r2_score(y, regr.predict(X_cubic))
```

```
plt.scatter(X, y, label='Training points', color='lightgray')
plt.plot(X_fit, y_lin_fit,
         label='Linear (d=1), $R^2=%.2f$' % linear_r2,
         color='blue', lw=2, linestyle=':')
plt.plot(X_fit, y_quad_fit,
         label='Quadratic (d=2), $R^2=%.2f$' % quadratic_r2,
         color='red', lw=2, linestyle='-')
plt.plot(X_fit, y_cubic_fit,
         label='Cubic (d=3), $R^2=%.2f$' % cubic_r2,
         color='green', lw=2, linestyle='--')
plt.xlabel('% lower status of the population [LSTAT]')
plt.ylabel('Price in $1000s [MEDV]')
plt.legend(loc='upper right')
plt.show()
```



LSTAT: Percentage of lower status
INDUS: Proportion of non-retail business
NOX: Nitric oxide concentration
RM: Average number of rooms
MEDV: Median value

Nonlinear Relationships in the Housing Dataset

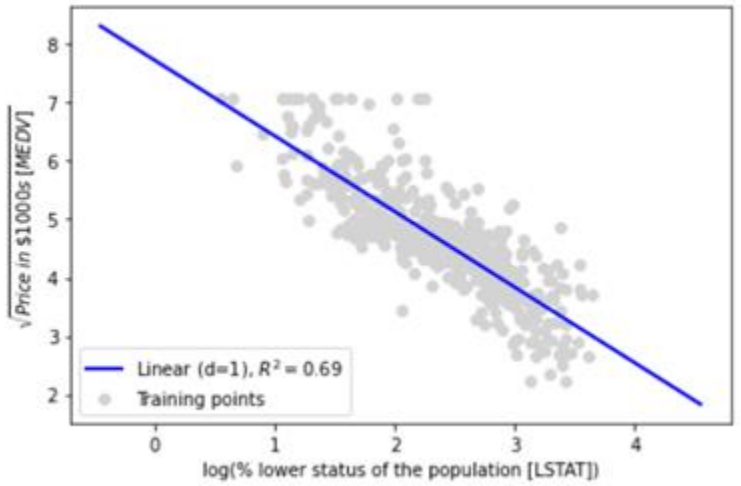
- Polynomial features are not always the best choice for modeling nonlinear relationships
- Looking at the MEDV - LSTAT scatterplot may lead to the hypothesis that a log-transformation of the LSTAT feature variable and the square root of MEDV may project the data onto a linear feature space suitable for a linear regression fit
- The relationship between the two variables looks quite similar to an exponential function

$$f(x) = e^{-x}$$

- Since the natural logarithm of an exponential function is a straight line, we assume that such a log-transformation can be usefully applied here

$$\log\big(f(x)\big) = -x$$

```
X = df[['LSTAT']].values
y = df['MEDV'].values
X_log = np.log(X)
y_sqrt = np.sqrt(y)
X_fit = np.arange(X_log.min()-1, X_log.max()+1, 1)[:, np.newaxis]
regr = regr.fit(X_log, y_sqrt)
y_lin_fit = regr.predict(X_fit)
linear_r2 = r2_score(y_sqrt, regr.predict(X_log))
plt.scatter(X_log, y_sqrt, label='Training points', color='lightgray')
plt.plot(X_fit, y_lin_fit,
         label='Linear (d=1), $R^2=%.2f$' % linear_r2,
         color='blue', lw=2)
plt.xlabel('log(% lower status of the population [LSTAT])')
plt.ylabel('$\sqrt{Price \; in \; \$1000s \; [MEDV]}$')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```



Random Forest Regression

- Random forest regression is conceptually quite different from the previous regression models in this chapter
  - The ensemble of decision trees can be understood as the sum of piecewise linear functions, in contrast to the global linear and polynomial regression models that we discussed previously
  - Via the decision tree algorithm, we subdivide the input space into smaller regions that become more manageable

Decision Tree Regression

- An advantage of the decision tree algorithm is that it does not require any transformation of the features if we are dealing with nonlinear data
  - Decision trees analyze one feature at a time, rather than taking weighted combinations into account
  - Likewise, normalizing or standardizing features is not required for decision trees
- Recall that we grow a decision tree by iteratively splitting its nodes until the leaves are pure or a stopping criterion is satisfied
- When we used decision trees for classification, we defined entropy as a measure of impurity to determine which feature split maximizes the information gain (IG), which can be defined as follows for a binary split

$$IG(D_p, x_i) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

Decision Tree Regression

- Recall that we want to find the feature split that maximizes the information gain
  - I.e., find the feature split that reduces the impurities in the child nodes most
- To use a decision tree for regression, we need an impurity metric that is suitable for continuous variables, so we define the impurity measure of a node, t, as the MSE

$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} \left(y^{(i)} - \hat{y}_t \right)^2$$

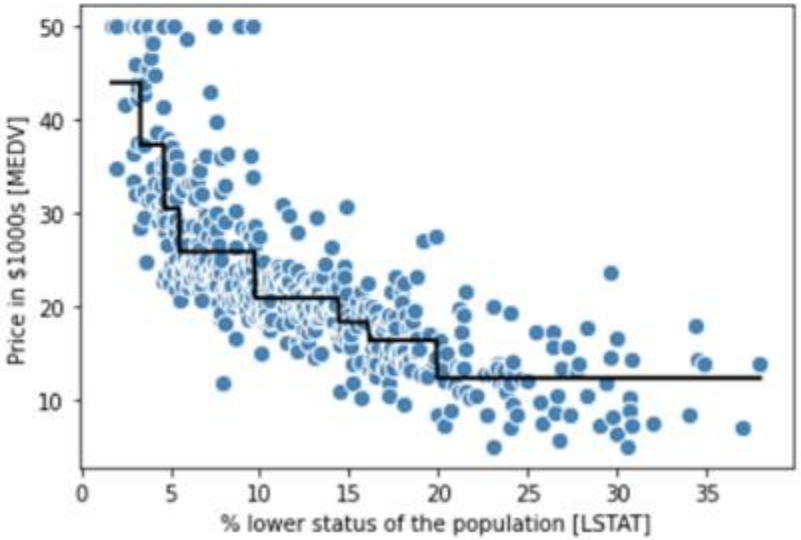
- N_t is the number of training examples at node t, D_t is the training subset at node t, $y^{(i)}$ is the true target value, and $\hat{y_t}$ is the predicted target value (sample mean)

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}$$

Decision Tree Regression

- To see what the line fit of a decision tree looks like, let's use the DecisionTreeRegressor to model the nonlinear relationship between the MEDV and LSTAT variables

```
from sklearn.tree import DecisionTreeRegressor
X = df[['LSTAT']].values
y = df['MEDV'].values
tree = DecisionTreeRegressor(max_depth=3)
tree.fit(X, y)
sort_idx = X.flatten().argsort()
lin_regplot(X[sort_idx], y[sort_idx], tree)
plt.xlabel('% lower status of the population [LSTAT]')
plt.ylabel('Price in $1000s [MEDV]')
plt.show()
```



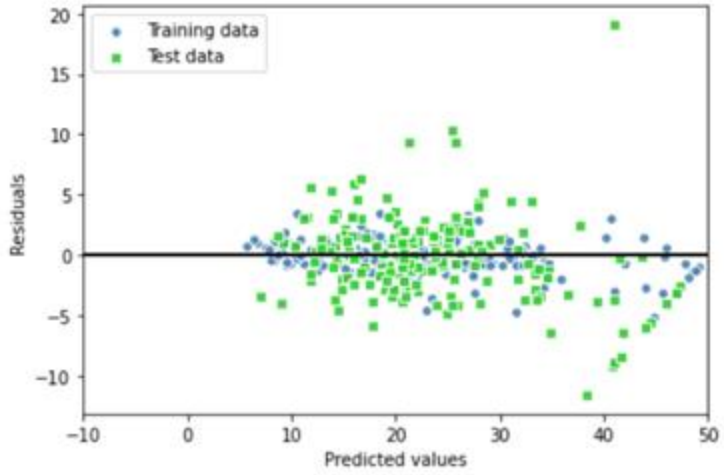
Random Forest Regression

- The basic random forest algorithm for regression is almost identical to the random forest algorithm for classification that we discussed earlier in the course
- The only difference is that we use the MSE criterion to grow the individual decision trees, and the predicted target variable is calculated as the average prediction over all decision trees
- Let's use all the features in the Housing dataset to fit a random forest regression model on 60 percent of the examples and evaluate its performance on the remaining 40 percent

```
X = df.iloc[:, :-1].values
y = df['MEDV'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=1)
```

```
from sklearn.ensemble import RandomForestRegressor
forest = RandomForestRegressor(n_estimators=1000, criterion='mse',
                               random_state=1, n_jobs=-1)
forest.fit(X_train, y_train)
y_train_pred = forest.predict(X_train)
y_test_pred = forest.predict(X_test)
print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
print('R^2 train: %.3f, test: %.3f' % (
        r2_score(y_train, y_train_pred),
        r2_score(y_test, y_test_pred)))
```

```
MSE train: 1.641, test: 11.056
R^2 train: 0.979, test: 0.878
```




```
plt.scatter(y_train_pred, y_train_pred - y_train,
            c='steelblue', edgecolor='white',
            marker='o', s=35,
            alpha=0.9, label='Training data')
plt.scatter(y_test_pred, y_test_pred - y_test,
            c='limegreen', edgecolor='white',
            marker='s', s=35,
            alpha=0.9, label='Test data')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='black')
plt.xlim([-10, 50])
plt.tight_layout()
plt.show()
```

References

- Most materials in this chapter are based on
  - Book
  - Code