Linear Regression

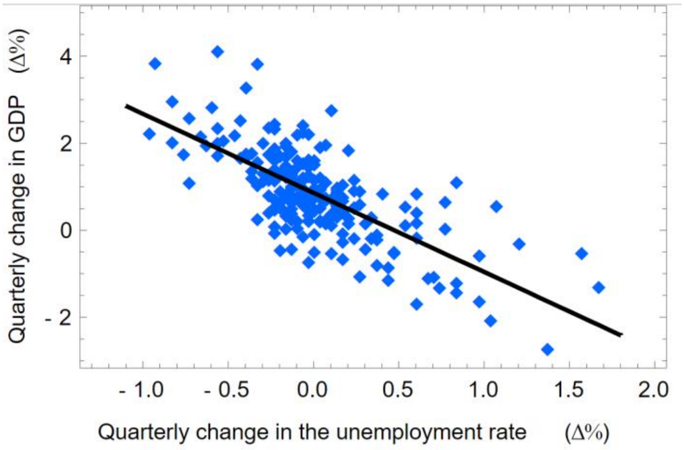
Roi Yehoshua

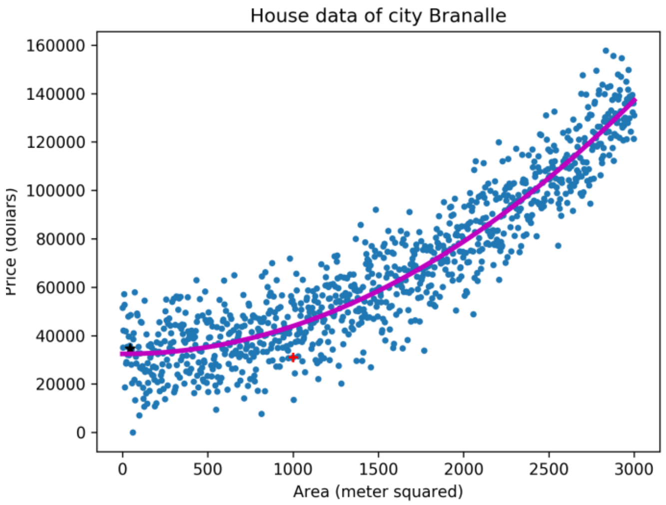
Agenda

- Linear regression
- Ordinary least squares (OLS)
- The normal equations
- Regression evaluation metrics
- Feature engineering
- Gradient descent
- Probabilistic interpretation of least squares
- Polynomial regression
- Basis functions
- Regularization
- The bias-variance tradeoff

Regression Task Definition

- **Given**: A **training set** of n labeled examples $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$
  - Each $\mathbf{x}_i$ is a d-dimensional vector of feature values, $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{id})^T$
    - Also known as the **explanatory** or **independent** variables
  - $y_i \in$ R is a continuous target value generated by an unknown function $y = f(\mathbf{x})$
    - Also known as the **response** or **dependent** variable
- **Goal**: Learn a function h (**hypothesis**) that maps a feature set $\mathbf{x}$ into a target y





Linear Regression

- In linear regression, we assume *y* is a linear function of **x**:

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + \ldots + w_d x_d$$

  - $\mathbf{w} = (w_0, \ldots, w_d)^T$ is a vector of **parameters** (also called **weights**)
  - w_0 is known as the **intercept** or **bias**
- To simplify the notation, we introduce a constant feature $x_0 = 1$
  - This allows us to write the prediction function as a dot product of **x** and **w**:

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T \mathbf{x} = \sum_{j=0}^{d} w_j x_j$$

- Our goal is to find **w** that will make $h(\mathbf{x})$ as close as possible to the true target *y*
  - at least on the training data (more on this later)

Ordinary Least Squares (OLS)

- A **loss function** measures how far the model's prediction is from the true label
- The most common loss function in regression tasks is the **squared loss**:

$$L_{\text{squared}}(y, h_{\mathbf{w}}(\mathbf{x})) = (y - h_{\mathbf{w}}(\mathbf{x}))^2$$

- Advantages:
  - Differentiable everywhere and convex (every local minimum is a global minimum)
  - Has as a nice probabilistic interpretation
- The **training error** (or **cost**) is the average squared loss over all training samples:

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

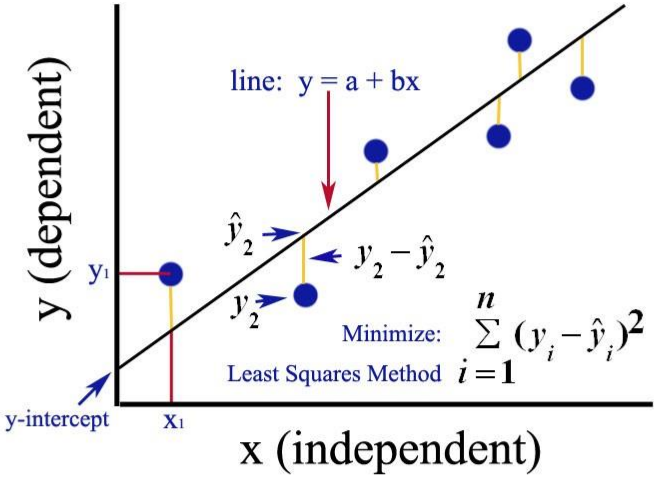
  - This function is known as **MSE (Mean Squared Error)**
- The method that aims to minimize the MSE is called **ordinary least squares (OLS)**

Simple Linear Regression

- A linear regression with a single feature x
- We are given n points: $(x_1, y_1), \ldots, (x_n, y_n)$
- The prediction function is a straight lines of the form:

$$h(x) = w_0 + w_1 x$$

  - w_0 is the intercept of the line and w_1 is its slope
- Our goal is to find the line that best fits the training data



- The cost function in this case is:

$$J(w_0, w_1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - h(x_i))^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - (w_0 + w_1 x_i))^2$$

- We find the minimum of J by computing its partial derivatives w.r.t. w_0 and w_1 and setting them to 0

Simple Linear Regression

- The partial derivative of the cost with respect to w_0 is:

$$\begin{aligned}
\frac{\partial}{\partial w_0} J(w_0, w_1) &= \frac{\partial}{\partial w_0} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - (w_0 + w_1 x_i)\right)^2 && \text{(definition of } J\text{)} \\
&= \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial w_0} \left(y_i - (w_0 + w_1 x_i)\right)^2 && \text{(sum of derivatives)} \\
&= \frac{1}{n} \sum_{i=1}^{n} \left[2 \left(y_i - (w_0 + w_1 x_i)\right) \frac{\partial}{\partial w_0} \left(y_i - (w_0 + w_1 x_i)\right) \right] && \text{(chain rule)} \\
&= \frac{1}{n} \sum_{i=1}^{n} 2(w_0 + w_1 x_i - y_i). && \text{(rearranging terms)}
\end{aligned}$$

- Setting the derivative to 0 yields:

$$\boxed{w_0 = \frac{\sum_{i=1}^{n} y_i - w_1 \sum_{i=1}^{n} x_i}{n}}$$

Simple Linear Regression

- The partial derivative of the cost with respect to w_1 is:

$$\begin{aligned}
\frac{\partial}{\partial w_1} J(w_0, w_1) &= \frac{\partial}{\partial w_1} \frac{1}{n} \sum_{i=1}^{n} (y_i - (w_0 + w_1 x_i))^2 && \text{(definition of } J\text{)} \\
&= \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial w_1} (y_i - (w_0 + w_1 x_i))^2 && \text{(sum of derivatives)} \\
&= \frac{1}{n} \sum_{i=1}^{n} \left[2\,(y_i - (w_0 + w_1 x_i)) \frac{\partial}{\partial w_1} (y_i - (w_0 + w_1 x_i)) \right] && \text{(chain rule)} \\
&= \frac{1}{n} \sum_{i=1}^{n} 2\,(y_i - (w_0 + w_1 x_i))\, x_i && \text{(partial derivative)}
\end{aligned}$$

- Setting the derivative to 0 yields the equation:

$$\sum_{i=1}^{n} x_i y_i - w_0 \sum_{i=1}^{n} x_i - w_1 \sum_{i=1}^{n} x_i^2 = 0$$

Simple Linear Regression

- Substituting the formula for w_0:

$$\sum_{i=1}^{n} x_i y_i - \left(\frac{\sum\limits_{i=1}^{n} y_i - w_1 \sum\limits_{i=1}^{n} x_i}{n} \right) \sum_{i=1}^{n} x_i - w_1 \sum_{i=1}^{n} x_i^2 = 0$$

$$\sum_{i=1}^{n} x_i y_i - \frac{\sum\limits_{i=1}^{n} x_i \sum\limits_{i=1}^{n} y_i}{n} + \frac{w_1 \left(\sum_{i=1}^{n} x_i\right)^2}{n} - w_1 \sum_{i=1}^{n} x_i^2 = 0$$

$$w_1 \left[\frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n} - \sum_{i=1}^{n} x_i^2 \right] = \frac{\sum\limits_{i=1}^{n} x_i \sum\limits_{i=1}^{n} y_i}{n} - \sum_{i=1}^{n} x_i y_i$$

$$w_1 = \frac{n \sum\limits_{i=1}^{n} x_i y_i - \sum\limits_{i=1}^{n} x_i \sum\limits_{i=1}^{n} y_i}{n \sum\limits_{i=1}^{n} x_i^2 - \left(\sum\limits_{i=1}^{n} x_i \right)^2}$$

Example

- Find the regression line that best matches the following data:

x	1	2	3	4	5
y	2.3	3.9	6.3	7.8	9.1

Example

- Computing common sums:

x	1	2	3	4	5	Σ = 15
y	2.3	3.9	6.3	7.8	9.1	Σ = 29.4
x²	1	4	9	16	25	Σ = 55
xy	2.3	7.8	18.9	31.2	45.5	Σ = 105.7

- Using the formulas for the coefficients:

$$w_1 = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} = \frac{5 \cdot 105.7 - 15 \cdot 29.4}{5 \cdot 55 - 15^2} = 1.75$$

$$w_0 = \frac{\sum_{i=1}^{n} y_i - w_1 \sum_{i=1}^{n} x_i}{n} = \frac{29.4 - 1.75 \cdot 15}{5} = 0.63$$

- Thus, the equation of the regression line is:

$$y = 1.75x + 0.63$$

Implementation in Python

- A function to compute the coefficients of the regression line:

```python
import numpy as np
import matplotlib.pyplot as plt
```

```python
def find_coefficients(x, y):
    n = len(x)  # Number of data points

    # Compute the slope
    w1 = (n * np.dot(x, y) - np.sum(x) * np.sum(y)) / 
         (n * np.sum(x**2) - np.sum(x)**2)

    # Compute the intercept
    w0 = (np.sum(y) - w1 * np.sum(x)) / n
    return w0, w1
```

Implementation in Python

- Defining the data points:

```
x = np.array([1, 2, 3, 4, 5])
y = np.array([2.3, 3.9, 6.3, 7.8, 9.1])
```

- Computing the coefficients:

```
w0, w1 = find_coefficients(x, y)
print(f'w0 = {w0:.3f}')
print(f'w1 = {w1:.3f}')
```

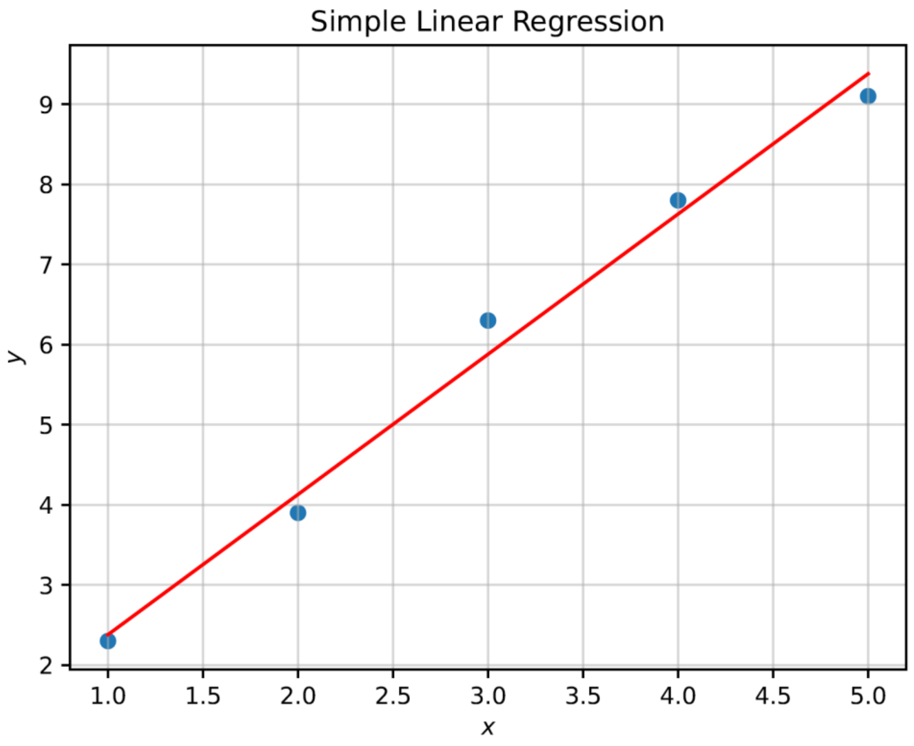
```
w0 = 0.63
w1 = 1.75
```

Implementation in Python

- To plot the regression line, we can take the min and max of the *x*-values and compute their corresponding *y* values:

```
def plot_regression_line(x, y, w0, w1):
    p_x = np.array([x.min(), x.max()])
    p_y = w0 + w1 * p_x
    plt.plot(p_x, p_y, 'r')
```

```
# Plot the data points along with the regression line
plt.scatter(x, y)
plot_regression_line(x, y, w0, w1)

plt.title('Simple Linear Regression')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.grid(alpha=0.5)
```



Multiple Linear Regression

- In the general case, we can have any number of input variables:

$$h(\mathbf{x}) = w_0 + w_1 x_1 + \ldots + w_d x_d = \mathbf{w}^T \mathbf{x}$$

- The dataset is now represented by a **feature matrix** *X* of size *n* × (*d* + 1)
  - *n* is the number of training samples, *d* is the number of features
  - We append a column of ones to the matrix to represent the bias terms

$$X = \begin{pmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1d} \\ 1 & x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix}$$

- The target labels are represented as a vector of size *n*

$$\mathbf{y} = (y_1, y_2, \ldots, y_n)^T$$

Multiple Linear Regression

- We can now write the cost function (MSE) in matrix form:

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2 = \frac{1}{n}(X\mathbf{w} - \mathbf{y})^T(X\mathbf{w} - \mathbf{y})$$

  - Proof:

$$X\mathbf{w} - \mathbf{y} = \begin{pmatrix} \mathbf{w}^T\mathbf{x}_1 \\ \vdots \\ \mathbf{w}^T\mathbf{x}_n \end{pmatrix} - \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} h_{\mathbf{w}}(\mathbf{x}_1) - y_1 \\ \vdots \\ h_{\mathbf{w}}(\mathbf{x}_n) - y_n \end{pmatrix}$$

$$(X\mathbf{w} - \mathbf{y})^T(X\mathbf{w} - \mathbf{y}) = \sum_{i=1}^{n}(h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2$$

Multiple Linear Regression

- To minimize *J*(**w**), we compute its gradient with respect to **w**

$$
\begin{aligned}
\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} &= \frac{1}{n}\cdot\frac{\partial\left((X\mathbf{w}-\mathbf{y})^T(X\mathbf{w}-\mathbf{y})\right)}{\partial \mathbf{w}} && \text{(definition of } J\text{)}\\
&= \frac{1}{n}\cdot\frac{\partial\left((X\mathbf{w})^T X\mathbf{w}-(X\mathbf{w})^T\mathbf{y}-\mathbf{y}^T(X\mathbf{w})+\mathbf{y}^T\mathbf{y}\right)}{\partial \mathbf{w}} && \text{(expanding brackets)}\\
&= \frac{1}{n}\cdot\frac{\partial\left(\mathbf{w}^T X^T X\mathbf{w}-\mathbf{y}^T(X\mathbf{w})-\mathbf{y}^T(X\mathbf{w})+\mathbf{y}^T\mathbf{y}\right)}{\partial \mathbf{w}} && ((AB)^T = B^T A^T,\ \mathbf{x}^T\mathbf{y}=\mathbf{y}^T\mathbf{x})\\
&= \frac{1}{n}\cdot\frac{\partial\left(\mathbf{w}^T X^T X\mathbf{w}-2\mathbf{y}^T(X\mathbf{w})\right)}{\partial \mathbf{w}} && (\mathbf{y}^T\mathbf{y}\text{ is not a function of }\mathbf{w})\\
&= \frac{1}{n}\cdot\frac{\partial\left(\mathbf{w}^T (X^T X)\mathbf{w}-2(X^T\mathbf{y})^T\mathbf{w}\right)}{\partial \mathbf{w}} && \text{(matrix multiplication associativity)}\\
&= \frac{1}{n}\left(\frac{\partial\left(\mathbf{w}^T(X^T X)\mathbf{w}\right)}{\partial \mathbf{w}} - 2\frac{\partial\left(\left(X^T\mathbf{y}\right)^T\mathbf{w}\right)}{\partial \mathbf{w}}\right) && \text{(derivatives of sum of functions)}\\
&= \frac{1}{n}\left(2X^T X\mathbf{w} - 2\frac{\partial\left(\left(X^T\mathbf{y}\right)^T\mathbf{w}\right)}{\partial \mathbf{w}}\right) && \left(\text{for a symmetric } A,\ \frac{\partial \mathbf{x}^T A\mathbf{x}}{\partial \mathbf{x}} = 2A\mathbf{x}\right)\\
&= \frac{1}{n}\left(2X^T X\mathbf{w} - 2X^T\mathbf{y}\right) && \left(\text{for any vector } \mathbf{u},\ \frac{\partial \mathbf{u}^T\mathbf{x}}{\partial \mathbf{x}} = \mathbf{u}\right)
\end{aligned}
$$

Multiple Linear Regression

- We now set the gradient to zero:

$$\frac{1}{n}\left(2X^TX\mathbf{w} - 2X^T\mathbf{y}\right) = 0$$

$$X^TX\mathbf{w} = X^T\mathbf{y}$$

  - These are known as the **normal equations**
- If the matrix X^TX is invertible (i.e., X has full rank), the optimal weight vector is:

$$\boxed{\mathbf{w}^* = (X^TX)^{-1}X^T\mathbf{y}}$$

- Otherwise, we use the Moore-Penrose pseudoinverse of X^TX

$$\mathbf{w}^* = (X^TX)^{+}X^T\mathbf{y}$$

  - The pseudoinverse is a generalization of matrix inverse that provides a least-squares solution to linear systems, even when the matrix is not invertible or not square

Implementation in Python

- A function for computing the solution:

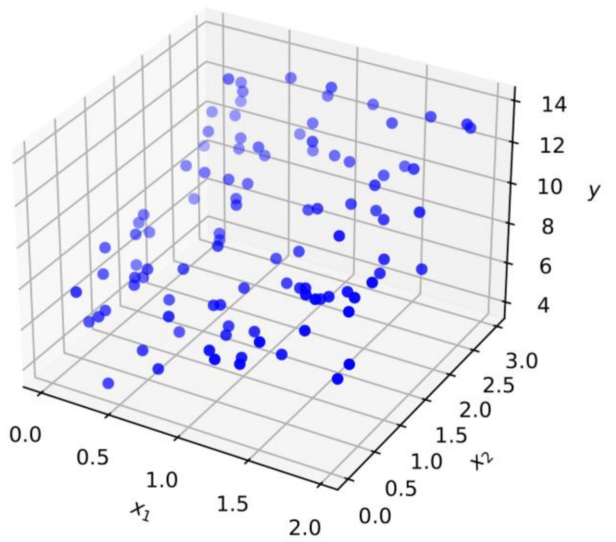
```
def closed_form_solution(X, y):
    w = np.linalg.pinv(X.T @ X) @ X.T @ y
    return w
```

- The function np.linalg.pinv compute the pseudo-inverse matrix
  - If the matrix is invertible, it returns the inverse

Implementation in Python

- To test the function, we generate a synthetic dataset with two features

```
def generate_data(n=100):
    # Generate synthetic data with two features and a linearly correlated label
    np.random.seed(42)

    x1 = 2 * np.random.rand(n)
    x2 = 3 * np.random.rand(n)
    y = 5 + 1 * x1 + 2 * x2 + np.random.randn(n)
    X = np.c_[x1, x2]  # Concatenate the two features
    return X, y

X, y = generate_data()
```



Implementation in Python

- Computing the optimal coefficients:

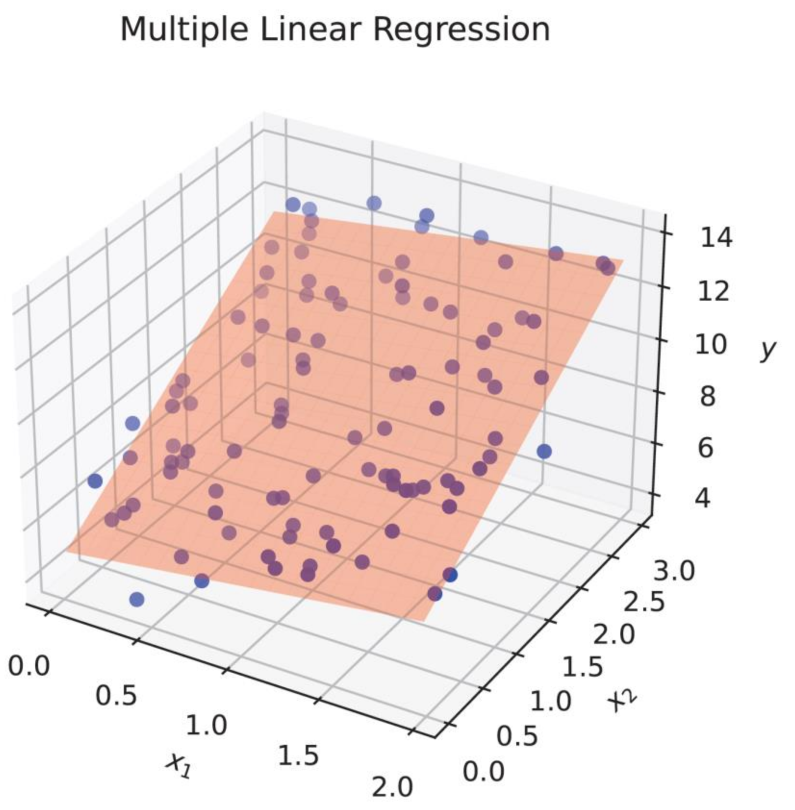
```
# Add a column of ones to X for the intercept term
n = len(X)
X_b = np.c_[np.ones((n, 1)), X]

# Compute the optimal coefficients
w = closed_form_solution(X_b, y)
print('Optimal coefficients:', np.round(w, 4))
```

```
Optimal coefficients: [4.9106 0.8291 2.2398]
```

Implementation in Python

- Plotting the data points with the regression plane

```
from mpl_toolkits.mplot3d import Axes3D  # For 3D plotting

# Set up the figure for 3D plotting
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot the data points
x1, x2 = X[:, 0], X[:, 1]
ax.scatter(x1, x2, y, color='blue')

# Create a mesh grid for the regression plane
x1_surf, x2_surf = np.meshgrid(np.linspace(x1.min(), x1.max(), 20),
                               np.linspace(x2.min(), x2.max(), 20))

# Calculate corresponding y values for the mesh grid
y_surf = w[0] + w[1] * x1_surf + w[2] * x2_surf

# Plot the regression plane
ax.plot_surface(x1_surf, x2_surf, y_surf, color='red', alpha=0.3)

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$y$')
```



The LinearRegression Class

- Scikit-Learn's LinearRegression class implements the closed-form solution for OLS

```
class sklearn.linear_model.LinearRegression(*, fit_intercept=True, copy_X=True,
tol=1e-06, n_jobs=None, positive=False)
```
[source]

  - **fit_intercept**: whether to calculate the intercept for the model
- Learned parameters:

Attribute	Description
coef_	Estimated coefficients
intercept_	The bias term

The LinearRegression Class

- Example for using the class on the same dataset:
  - No need to append a column of ones to the feature matrix this time

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X, y)
```

```
▾ LinearRegression
LinearRegression()
```

```
print('Intercept:', np.round(model.intercept_, 4))
print('Coefficients:', np.round(model.coef_, 4))
```

```
Intercept: 4.9106
Coefficients: [0.8291 2.2398]
```

Evaluation Metrics

- Common performance measures for regression models:
  - RMSE (Root Mean Squared Error)
  - MAE (Mean Absolute Error)
  - R^2 score

RMSE (Root Mean Squared Error)

- The square root of the mean of squared errors (MSE):

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

- Pros
  - Expressed in the same units as the target variable (easy interpretation)
  - Assigns greater weight to larger deviations (due to the squaring)
- Cons
  - Unsuitable for comparing models across different datasets
  - Sensitive to outliers

RMSE (Root Mean Squared Error)

- To compute it, use the function **root_mean_squared_error** from sklearn.metrics

```
from sklearn.metrics import root_mean_squared_error as RMSE

y_pred = model.predict(X)
rmse = RMSE(y, y_pred)
print(f'RMSE: {rmse:.4f}')
```

```
RMSE: 0.9725
```

MAE (Mean Absolute Error)

- The average of the absolute errors:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

- Pros
  - Simple and intuitive
  - Less sensitive to outliers than RMSE
- Cons
  - Unsuitable for comparing models across different datasets (like RMSE)
  - Treats all errors equally

MAE (Mean Absolute Error)

- To compute it, use the function **mean_absolute_error** from sklearn.metrics

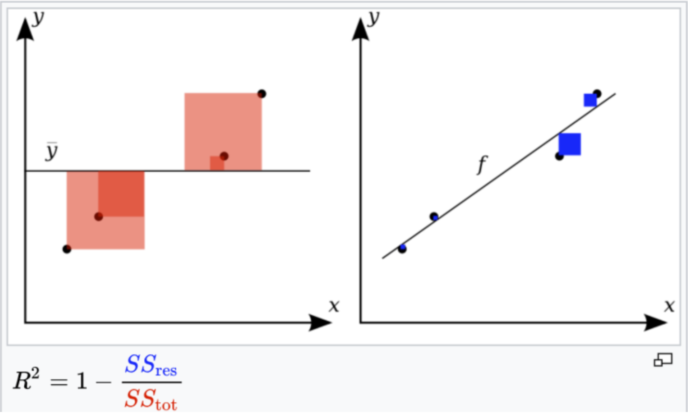
```
from sklearn.metrics import mean_absolute_error

mae = mean_absolute_error(y, y_pred)
print(f'MAE: {mae:.4f}')
```

```
MAE: 0.7748
```

R^2 Score

- Measures the ratio of the model's squared errors to the squared errors of a baseline model that always predicts the mean target value $\bar{y}$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$



  - R^2 = 1: a perfect model
  - R^2 < 0: the model performs worse than the baseline
- Pros
  - Scale independent, can be used to compare model across different datasets
  - Has a probabilistic interpretation (the proportion of variance in the target variable that is explained by the independent variables)
- Cons
  - Doesn't reflect the true magnitude of the prediction errors

R^2 Score

- To compute it, use the **r2_score()** function from sklearn.metrics:

```
from sklearn.metrics import r2_score

score = r2_score(y, y_pred)
print(f'R² score: {score:.4f}')
```

```
R² score: 0.8094
```

- Or directly using the **score()** method of the regressor:

```
score = model.score(X, y)
print(f'R² score: {score:.4f}')
```

```
R² score: 0.8094
```

Predicting House Prices

- We now build a regression model to predict housing prices in California
- The California housing dataset was derived from the 1990 US census
  - Each row represents a block group (district) with 600-3,000 residents
  - There are 8 input features including the median income, median house age, location
  - The target is the median house value in the district (measured in $100,000s)

Data Set Characteristics:

Number of Instances:	20640
Number of Attributes:	8 numeric, predictive attributes and the target
Attribute Information:	• MedInc median income in block group • HouseAge median house age in block group • AveRooms average number of rooms per household • AveBedrms average number of bedrooms per household • Population block group population • AveOccup average number of household members • Latitude block group latitude • Longitude block group longitude
Missing Attribute Values:	None

Loading the Data

- We first fetch the dataset using the sklearn.datasets module:

```
from sklearn.datasets import fetch_california_housing

X, y = fetch_california_housing(as_frame=True, return_X_y=True)
```

```
X.head()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25

Train-Test Split

- Splitting the dataset into training and test sets:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
X_train.shape, X_test.shape
```

```
((16512, 8), (4128, 8))
```

Model Training

- We now fit a regression model to the training set:

```
from sklearn.linear_model import LinearRegression

reg = LinearRegression()
reg.fit(X_train, y_train)
```

```
▾ LinearRegression
LinearRegression()
```

- The learned coefficients are:

```
print('Intercept:', np.round(reg.intercept_, 4))
print('Coefficients:', np.round(reg.coef_, 4))
```

```
Intercept: -37.0233
Coefficients: [ 0.4487  0.0097 -0.1233  0.7831 -0.      -0.0035 -0.4198 -0.4337]
```

Model Evaluation

- Computing the R^2 score on both the training and test sets:

```
train_r2 = reg.score(X_train, y_train)
print(f'R² score (train): {train_r2:.4f}')
test_r2 = reg.score(X_test, y_test)
print(f'R² score (test): {test_r2:.4f}')
```

```
R² score (train): 0.6126
R² score (test): 0.5758
```

- Computing RMSE:

```
from sklearn.metrics import root_mean_squared_error as RMSE

train_rmse = RMSE(y_train, reg.predict(X_train))
print(f'Train RMSE: {train_rmse:.4f}')
test_rmse = RMSE(y_test, reg.predict(X_test))
print(f'Test RMSE: {test_rmse:.4f}')
```

```
Train RMSE: 0.7197
Test RMSE: 0.7456
```

Feature Engineering

- Design new features based on the existing ones
  - Can help the model better capture important relationships in the data
  - Often requires domain knowledge
- For example, in the California housing dataset we have two features:
  - **AveRooms**: average number of rooms
  - **AveOccup**: average household size
- The ratio between these two features may correlate better with the target

```
X['RoomsPerIndividual'] = X['AveRooms'] / X['AveOccup']
X.head(3)
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	RoomsPerIndividual
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	2.732919
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	2.956685
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	2.957661

Feature Engineering

- The correlations of the features with the target variable:

```
correlations = X.corrwith(y).sort_values(ascending=False)
print(correlations)
```

```
MedInc                 0.688075
RoomsPerIndividual     0.209482
AveRooms               0.151948
HouseAge               0.105623
AveOccup              -0.023737
Population            -0.024650
Longitude             -0.045967
AveBedrms             -0.046701
Latitude              -0.144160
dtype: float64
```

  - The new feature has stronger correlation than the two individual features

Feature Engineering

- Let's train a regression model with the new feature:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

reg.fit(X_train, y_train)
print(f'R² score (train): {reg.score(X_train, y_train):.4f}')
print(f'R² score (test): {reg.score(X_test, y_test):.4f}')
```

```
R² score (train): 0.6489
R² score (test): 0.6395
```

- The test R^2 score improved from 0.5758 to 0.6395!

Feature Engineering

- Building a custom transformer that adds the new feature to the dataset:

```
from sklearn.base import BaseEstimator, TransformerMixin

class RoomsPerIndividualAdder(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X = X.copy()
        X['RoomsPerIndividual'] = X['AveRooms'] / X['AveOccup']
        return X
```

```
X, y = fetch_california_housing(as_frame=True, return_X_y=True)

transformer = RoomsPerIndividualAdder()
X_transformed = transformer.transform(X)
```

Discretization

- Discretization transforms a continuous feature into a discrete one
- Divides its range of values into a set of *n* **intervals** (also called **bins**)

$$[a, b] \quad \Rightarrow \quad [a, x_1], (x_1, x_2], \ldots, (x_n, b]$$

- Two main approaches:
  - **Equal-width binning**: all bins have the same width
  - **Equal-frequency binning**: all bins have the same number of data points
- Example: 1, 3, 4, 7, 11, 12, 15, 17, 24, 31
  - Equal-width binning into 5 bins
    - [1, 7], (7, 13], (13, 19], (19, 25], (25, 31]
  - Equal-frequency binning into 5 bins
    - [1, 3.5], (3.5, 9], (9, 13.5], (13.5, 20.5], (20.5, 31]

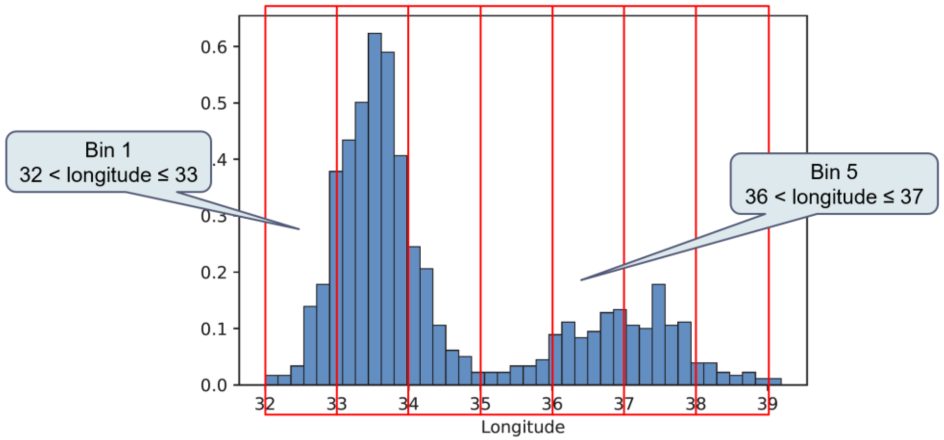
Discretization

- Can help the model learn independent mappings between the bins and the target
  - As each bin becomes an independent feature in the dataset
- For example, if *x* is the house location and *y* is its price, a simple linear model is:

$$h(x) = w_0 + w_1 x$$

  - e.g., if the house moves 10 radians to the right, the price automatically increases by 10
- By discretizing *x* into *k* bins, the linear model becomes:

$$h(x) = w_0 + w_1 x_1 + \ldots + w_k x_k$$



Discretization

- The class KBinsDiscretizer can be used for discretization

```
class sklearn.preprocessing.KBinsDiscretizer(n_bins=5, *, encode='onehot',
strategy='quantile', quantile_method='warn', dtype=None, subsample=200000,
random_state=None)
```
[source]

  - **n_bins**: the number of bins to create
  - **encode**: method to encode the discretized result ('onehot', 'onehot-dense', or 'ordinal')
  - **strategy**: strategy used to define the widths of the bins
    - 'uniform': equal-width binning
    - 'quantile': equal-frequency binning
    - 'kmeans': using k-means clustering

Discretization

- Example:

```
from sklearn.preprocessing import KBinsDiscretizer

discretizer = KBinsDiscretizer(n_bins=3, strategy='uniform', encode='onehot-dense')
X = [[-0.3, 2.0], [-0.2, 2.5], [0, 2.8], [0.3, 3.0]]
discretizer.fit_transform(X)
```

```
array([[1., 0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 1.],
       [0., 0., 1., 0., 0., 1.]])
```

- The bin edges can be inspected using the **bin_edges_** attribute:

```
discretizer.bin_edges_
```

```
array([array([-0.3, -0.1,  0.1,  0.3]),
       array([2.        , 2.33333333, 2.66666667, 3.        ])],
      dtype=object)
```

Discretization

- Let's discretize the Longitude and Latitude columns of the California housing data

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

discretizer = ColumnTransformer([
    ('disc', KBinsDiscretizer(n_bins=10), ['Longitude', 'Latitude'])
], remainder='passthrough')

model = Pipeline([
    ('adder', RoomsPerIndividualAdder()),
    ('discretizer', discretizer),
    ('reg', LinearRegression())
])
```

Ensures that all other features are included in the output of the column transformer

Discretization

- Model evaluation:

```
model.fit(X_train, y_train)
print(f'R² score (train): {model.score(X_train, y_train):.4f}')
print(f'R² score (test): {model.score(X_test, y_test):.4f}')
```

```
R² score (train): 0.6701
R² score (test): 0.6679
```

- The test R^2 score increased from 0.6395 to 0.6679!

Error Analysis

- Examine specific instances where predictions deviate significantly from the targets

```
# Compute the residuals on the test samples
y_test_pred = model.predict(X_test)
residuals = np.abs(y_test - y_test_pred)

# Add the residuals to the DataFrame
housing_df = pd.concat([X, y], axis=1)
housing_df.loc[X_test.index, 'Residual'] = residuals

# Sort the samples in a descending order of the residuals
housing_df.sort_values('Residual', ascending=False).head(5)
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedHouseVal	Residual
6688	0.4999	28.0	7.677419	1.870968	142.0	4.580645	34.15	-118.08	5.00001	4.679357
459	1.1696	52.0	2.436000	0.944000	1349.0	5.396000	37.87	-122.25	5.00001	4.171453
10574	1.9659	6.0	4.795455	1.159091	125.0	2.840909	33.72	-117.70	5.00001	4.004750
17306	2.7275	17.0	5.574286	1.051429	681.0	1.945714	34.38	-119.55	5.00001	3.436260
12069	4.2386	6.0	7.723077	1.169231	228.0	3.507692	33.83	-117.55	5.00001	3.391628

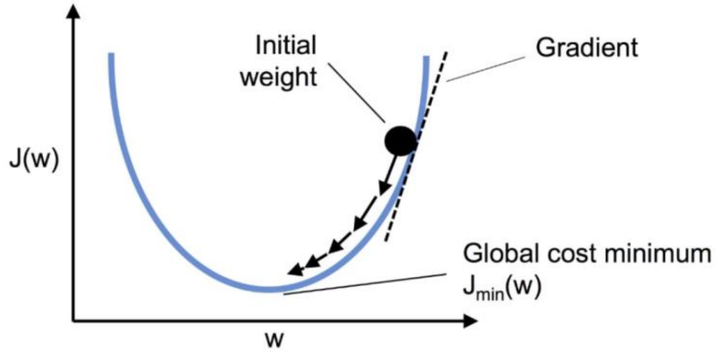
Issues with the Closed Form Solution

- High computational cost, especially in high-dimensional datasets
  - Requires inverting the matrix X^TX whose shape is $d \times d$ (d is the number of features)
  - Has a time complexity of $O(d^3)$ operations
- Requires loading the entire dataset in memory
- If X^TX is nearly singular or ill-conditioned, it can cause numerical stability issues
- Doesn't support incremental (online) learning
  - Any change to the dataset requires re-computing X^TX
- Alternative approach?

Gradient Descent!

Gradient Descent

- An iterative technique for minimizing a differentiable function
- Updates model parameters **w** in direction of the negative gradient of the cost *J*(**w**)
- The update rule is: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w})$
  - α is a **learning rate** that controls the step sizes



- Variants of gradient descent:
  - **Batch gradient descent**: The gradient is computed over the entire training set
  - **Stochastic gradient descent (SGD)**: The gradient is computed on a single training example
  - **Mini-batch gradient descent**: The gradient is computed on a small number of examples

Gradient Descent for Linear Regression

- The partial derivative of the MSE $J(\mathbf{w})$ with respect to each weight w_j is:

$$\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j}\frac{1}{n}\sum_{i=1}^{n}(y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2 && \text{(definition of } J\text{)} \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial w_j}(y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2 && \text{(linearity of the derivative)} \\
&= \frac{1}{n}\sum_{i=1}^{n}2(y_i - h_{\mathbf{w}}(\mathbf{x}_i))\frac{\partial}{\partial w_j}(y_i - h_{\mathbf{w}}(\mathbf{x}_i)) && \text{(chain rule)} \\
&= -\frac{2}{n}\sum_{i=1}^{n}(y_i - h_{\mathbf{w}}(\mathbf{x}_i))x_{ij} && \text{(since } \partial h_{\mathbf{w}}(\mathbf{x}_i)/\partial w_j = x_{ij}\text{)}
\end{aligned}$$

- Thus, the batch gradient descent update rule is:

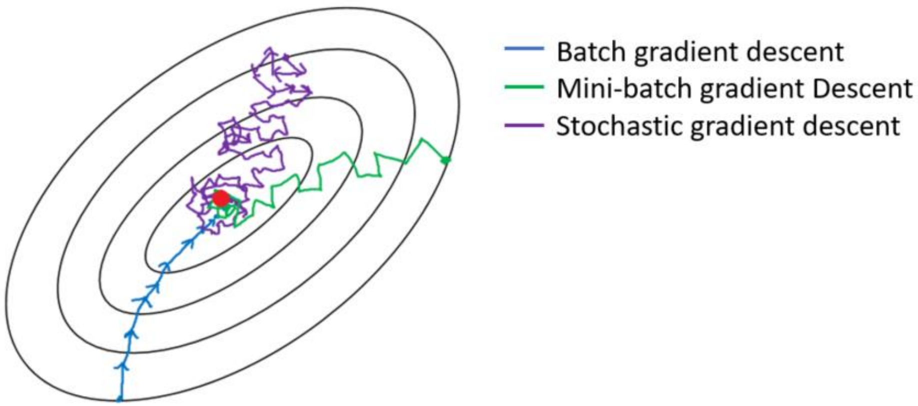
$$\boxed{\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{n}\sum_{i=1}^{n}(h_{\mathbf{w}}(\mathbf{x}_i) - y_i)\mathbf{x}_i}$$

Stochastic Gradient Descent (SGD)

- Updates the model parameters using a single training example at a time
- The weight update rule is:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha(h_{\mathbf{w}}(\mathbf{x}_i) - y_i)\mathbf{x}_i$$

- Advantages of SGD
  - Converges faster than batch gradient descent
  - Doesn't require the entire dataset to reside in memory
  - Can escape local minima more easily
  - Supports online learning
- Disadvantages
  - Optimization trajectory is more noisy
  - Typically, doesn't reach the exact minimum



The SGDRegressor Class

- Implements SGD for fitting linear regression models

```
class sklearn.linear_model.SGDRegressor(loss='squared_error', *, penalty='l2',
alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001,
shuffle=True, verbose=0, epsilon=0.1, random_state=None, learning_rate='invscaling',
eta0=0.01, power_t=0.25, early_stopping=False, validation_fraction=0.1,
n_iter_no_change=5, warm_start=False, average=False)
```
[source]

Parameter	Description
loss	The loss function to optimize.
penalty	The type of regularization to use (more on this later)
max_iter	Maximum number of passes over the training data (epochs)
tol	Training stops when (loss > previous_loss - tol)
shuffle	Whether or not the training data should be shuffled after each epoch
learning_rate	The learning rate schedule: can be 'constant', 'optimal', 'invscaling', or 'adaptive'
eta0	Initial learning rate

The SGDRegressor Class

- Fitting an SGDRegressor estimator to the California housing dataset:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split

X, y = fetch_california_housing(as_frame=True, return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
from sklearn.linear_model import SGDRegressor

model = SGDRegressor(random_state=42)
model.fit(X_train, y_train)

print(f'R² score (train): {model.score(X_train, y_train):.4f}')
print(f'R² score (test): {model.score(X_test, y_test):.4f}')
```

```
R² score (train): -64066325035282430173045063680.0000
R² score (test): -64065584159543795873711390720.0000
```

Why are the results so bad?

The SGDRegressor Class

- Need scaling!!

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

model = Pipeline([
    ('scaler', StandardScaler()),
    ('reg', SGDRegressor(random_state=42))
])

model.fit(X_train, y_train)
print(f'R² score (train): {model.score(X_train, y_train):.4f}')
print(f'R² score (test): {model.score(X_test, y_test):.4f}')
```

```
R² score (train): 0.6047
R² score (test): 0.5798
```

- Results very similar to the closed-form solution

```
R² score (train): 0.6126
R² score (test): 0.5758
```

Probabilistic Interpretation of OLS Regression

- Given a regression problem, why ordinary least squares regression is a good choice?
- The OLS estimator is the **MLE (maximum likelihood estimator)** for the regression model parameters **w** under the following assumptions:
  - The target variable is a linear function of the input variables (+ noise)

$$y_i = \mathbf{w}^T \mathbf{x}_i + \epsilon_i$$

    - ε_i is called **irreducible error** (represents random noise and unmodeled effects)
  - The errors are i.i.d. (independent and identically distributed)
  - The errors are normally distributed with zero mean and a constant variance

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

Proof

- From the normality of the errors: $y_i \sim \mathcal{N}(\mathbf{w}^T\mathbf{x}_i, \sigma^2)$

$$p(y_i|\mathbf{x}_i, \mathbf{w}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - \mathbf{w}^T\mathbf{x}_i)^2}{2\sigma^2}\right)$$

- The likelihood of the model parameters is (using the i.i.d. assumption):

$$\mathcal{L}(\mathbf{w}|X, \mathbf{y}) = \prod_{i=1}^{n} p(y_i|\mathbf{x}_i, \mathbf{w}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - \mathbf{w}^T\mathbf{x}_i)^2}{2\sigma^2}\right)$$

- Taking logarithm:

$$\log \mathcal{L}(\mathbf{w}|X, \mathbf{y}) = \sum_{i=1}^{n} \log\left(\frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(y_i - \mathbf{w}^T\mathbf{x}_i)^2}{2\sigma^2}\right]\right)$$

$$= -n\log(\sqrt{2\pi}\sigma) - \frac{1}{2\sigma^2} \boxed{\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i)^2}$$

- Conclusion: **Minimizing the sum of squared errors = maximizing the likelihood**

Assumptions of Linear Regression

- **Linearity:** The conditional expectation of the target is a linear function of the inputs

$$\mathbb{E}[y_i|\mathbf{x}_i] = \mathbf{w}^T\mathbf{x}_i$$

- **Exogeneity**: The error has zero mean given the input

$$\mathbb{E}[\epsilon_i|\mathbf{x}] = 0, \text{ for all } i$$

- **Independence of errors**: The errors are uncorrelated across observations

$$\mathbb{E}[\epsilon_i\epsilon_j] = 0, \quad \text{for } i \neq j$$

- **Homoscedasticity**: The errors have constant variance across all levels of the inputs

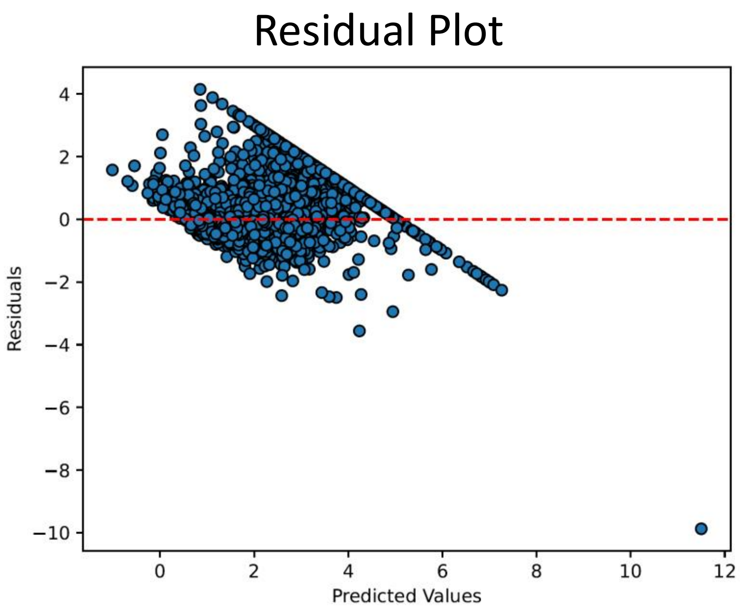
$$\text{Var}(\epsilon_i) = \sigma, \text{ for all } i$$

- **Normality of errors (optional)**: The errors are normally distributed

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

Assumptions of Linear Regression

- Violations of these assumptions may lead to:
  - Poor predictive performance
  - Biased coefficient estimates
  - Misleading evaluation metrics
- Plots that can help diagnose violations of the assumptions
  - **Residual plots** can reveal heteroscedasticity
  - **Residual histograms** or **QQ plots** help assess normality
  - **Autocorrelation plots** test independence



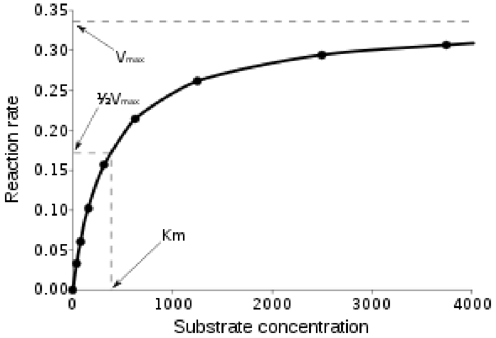
Linear Regression Summary

Algorithm	Large training set	Many features	Out-of-core support	Hyper-parameters	Scaling required	Scikit-Learn
Normal equations	Fast	Slow	No	0	No	LinearRegression
Batch GD	Slow	Fast	No	2	Yes	N/A
Stochastic GD	Fast	Fast	Yes	≥ 2	Yes	SGDRegressor
Mini-batch GD	Fast	Fast	Yes	≥ 2	Yes	N/A

Nonlinear Regression

- In many problems, the relationship between the features and the target is nonlinear
- In general, we assume that target is given by $y = f(\mathbf{x}; \mathbf{w})$
  - where *y* is a function of the inputs **x** and **w** is a vector of learnable parameters
- In some cases, the form of *f* is **known** and we only need to find the coefficients **w**
  - e.g., the Michaelis–Menten model describes the rate of enzyme reactions

$$y = \frac{w_1 x}{w_0 + x}$$



  - Each enzyme has its own coefficients w_0 and w_1 which need to be estimated from data
- In many other cases, the function *f* is **not known** a priori

Transformable Nonlinear Regression

- Some nonlinear functions can be transformed into linear functions
- For example, a power function $y = ax^b$ can be linearized by taking logs of both sides

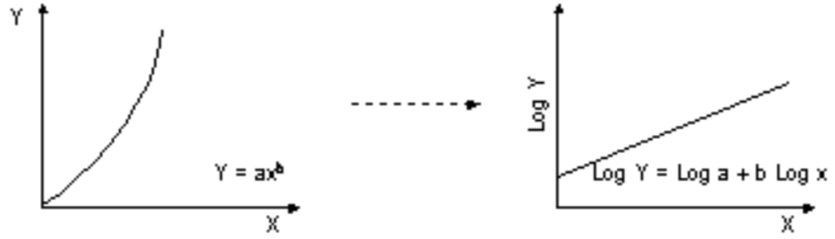
$$\log y = \log a + b \log x$$

- We can formulate this as a linear equation by defining new variables:

$$Y = \log y,\ \ X = \log x,\ \ \alpha = \log a,\ \ \beta = b$$

  - such that the equation in *X* and *Y* becomes linear:

$$Y = \alpha + \beta X$$



- We can now use linear regression to find the coefficients α, β
- Then, recover the parameters of the original function: $a = e^{\alpha},\ b = \beta$

Transformable Nonlinear Regression

- Other nonlinear transformations:

	Model	Model transformation	Parameters transformation
Power	$y = ax^b$	$Y = \log y, X = \log x$	$\alpha = \log a, \beta = b$
Exponential 1	$y = ae^{bx}$	$Y = \log y, X = x$	$\alpha = \log a, \beta = b$
Exponential 2	$y = ab^x$	$Y = \log y, X = x$	$\alpha = \log a, \beta = \log b$
Logarithmic	$y = \log(ax^b)$	$Y = y, X = \log x$	$\alpha = \log a, \beta = b$
Reciprocal 1	$y = \frac{1}{a+bx}$	$Y = \frac{1}{y}, X = x$	$\alpha = \log a, \beta = b$
Reciprocal 2	$y = a + \frac{b}{1+x}$	$Y = y, X = \frac{1}{1+x}$	$\alpha = a, \beta = b$
Square root	$y = a + b\sqrt{x}$	$Y = y, X = \sqrt{x}$	$\alpha = a, \beta = b$

Basis Functions

- <u>Idea</u>: Express the prediction function as a linear combination of simpler functions

$$h(x) = w_0 + w_1\phi_1(x) + w_2\phi_2(x) + \ldots + w_k\phi_k(x)$$

- The functions ϕ_j are called **basis functions**
  - e.g., when the basis functions are powers of *x*, we get **polynomial regression**:

$$h(x) = w_0 + w_1 x + w_2 x^2 + \ldots + w_k x^k$$

- By introducing new variables, we can transform this into a linear regression problem

$$h(x) = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_k x_k$$

  - where $x_1 = \phi_1(x), x_2 = \phi_2(x), \ldots, x_k = \phi_k(x)$
- We can now apply standard linear regression techniques to estimate **w** from the data
  - e.g., closed-form solution, gradient descent

Polynomial Regression

- A widely used technique for modeling nonlinear relationships
  - The **Weierstrass approximation theorem** guarantees that any continuous function on a closed interval can be approximated by a polynomial arbitrarily well
- For a single input feature *x*, the polynomial regression equation is:

$$h(x) = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d$$

- The **feature matrix** contains the powers of *x* up to degree *d* on its columns:

$$X = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^d \end{pmatrix}$$

- We can find the coefficients $w_0, \ldots, w_d$ using the closed-form solution:

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

Example: Quadratic Regression

- We throw a ball into the air and measure its height at different points in time:

t_i (seconds)	h_i (meters)
0	0
1	4.16
2	7.15
3	9.32
4	10.41
5	10.5

- From physics, the height can be modeled as a quadratic function of the time:

$$h_i = w_1 t_i + w_2 t_i^2 + \epsilon$$

- We want to estimate w_1 and w_2 from the given data

Example: Quadratic Regression

- We first define the feature matrix:

```
t = np.array([0, 1, 2, 3, 4, 5])  # time in seconds
h = np.array([0, 4.16, 7.15, 9.32, 10.41, 10.5])  # height in meters

X = np.c_[np.ones(len(t)), t, t**2]
print(X)
```

```
[[ 1.   0.   0.]
 [ 1.   1.   1.]
 [ 1.   2.   4.]
 [ 1.   3.   9.]
 [ 1.   4. 16.]
 [ 1.   5. 25.]]
```

Example: Quadratic Regression

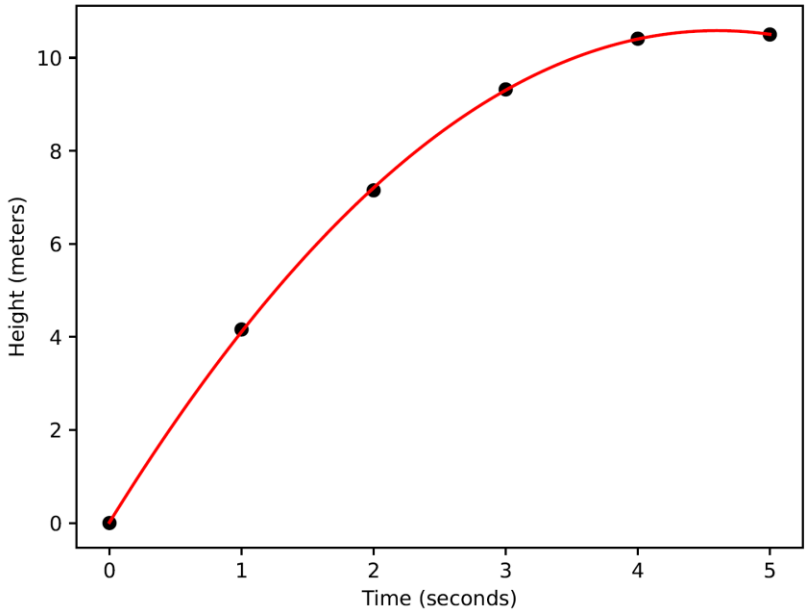
- We can now apply the closed-form solution:

```
w = np.linalg.inv(X.T @ X) @ X.T @ h
print(w)
```

```
[ 0.01535714  4.59325    -0.49910714]
```

- Plotting the function:

```
import matplotlib.pyplot as plt

plt.scatter(t, h, c='k')
t_test = np.linspace(0, 5, 100)
h_test = w[0] + w[1] * t_test + w[2] * t_test**2
plt.plot(t_test, h_test, c='r')
plt.xlabel('Time (seconds)')
plt.ylabel('Height (meters)')
```



Feature Crosses

- A **feature cross** is formed by multiplying two or more features
  - e.g., $x_1 x_2$ is a feature cross between x_1 and x_2
- Feature crosses can improve the model's ability to capture complex relationships
- In polynomial regression, we typically include all feature crosses up to degree d
- For example, for 3 variables x_1, x_2, x_3 and d = 2, the feature matrix becomes:

$$X = \begin{pmatrix} 1 & x_{11} & x_{12} & x_{13} & x_{11}x_{12} & x_{11}x_{13} & x_{12}x_{13} & x_{11}^2 & x_{12}^2 & x_{13}^2 \\ 1 & x_{21} & x_{22} & x_{23} & x_{21}x_{22} & x_{21}x_{23} & x_{22}x_{23} & x_{21}^2 & x_{22}^2 & x_{23}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & x_{n3} & x_{n1}x_{n2} & x_{n1}x_{n3} & x_{n2}x_{n3} & x_{n1}^2 & x_{n2}^2 & x_{n3}^2 \end{pmatrix}$$

  - Be careful: the number of features grows exponentially with the degree d

The PolynomialFeatures Class

- PolynomialFeatures is a transformer that generates the polynomial features including all interaction terms from a given design matrix *X*

```
class sklearn.preprocessing.PolynomialFeatures(degree=2, *, interaction_only=False,
include_bias=True, order='C')
```

[source]

```
from sklearn.preprocessing import PolynomialFeatures

X = np.array([[0, 1], [2, 3], [4, 5]])
poly_features = PolynomialFeatures(degree=2)
X_new = poly_features.fit_transform(X)
X_new
```

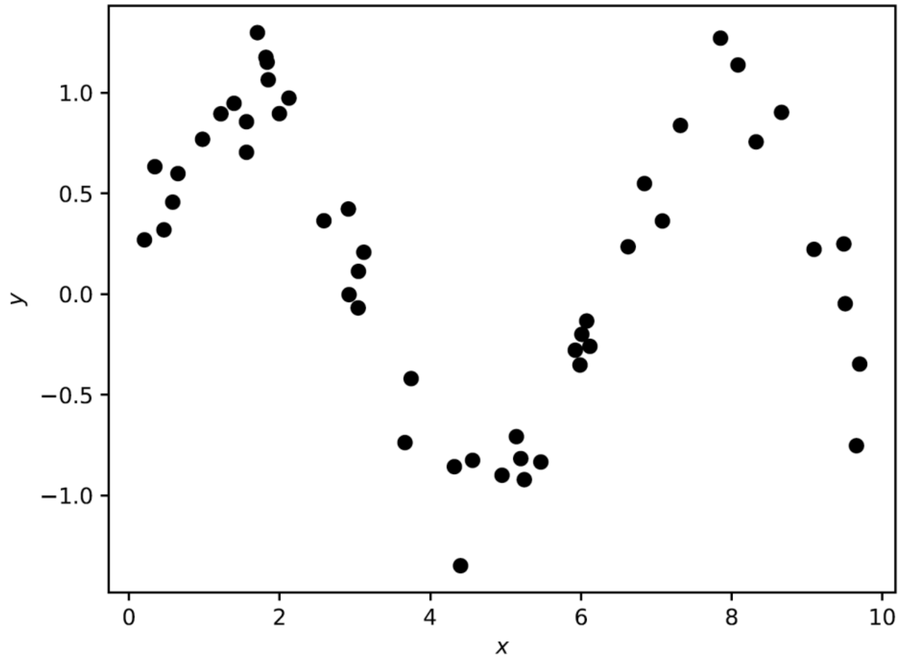
```
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

Polynomial Regression in Scikit-Learn

- Assume that we have 50 data points sampled from sin*x* in the interval [0, 10]
  - plus some random Gaussian noise with mean 0 and std 0.2

```
np.random.seed(42)

def make_data(n_samples=50, std=0.2):
    x = np.random.rand(n_samples) * 10
    err = np.random.normal(size=n_samples) * std
    y = np.sin(x) + err
    return x, y
```

```
x, y = make_data()
plt.scatter(x, y, color='k')
plt.xlabel('$x$')
plt.ylabel('$y$')
```



Polynomial Regression in Scikit-Learn

- Let's try to fit polynomials of various degrees to this data
- We define a pipeline that combines PolynomialFeatures with LinearRegression

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

def PolynomialRegression(degree=2):
    return Pipeline([('poly', PolynomialFeatures(degree, include_bias=False)),
                     ('reg', LinearRegression())])
```

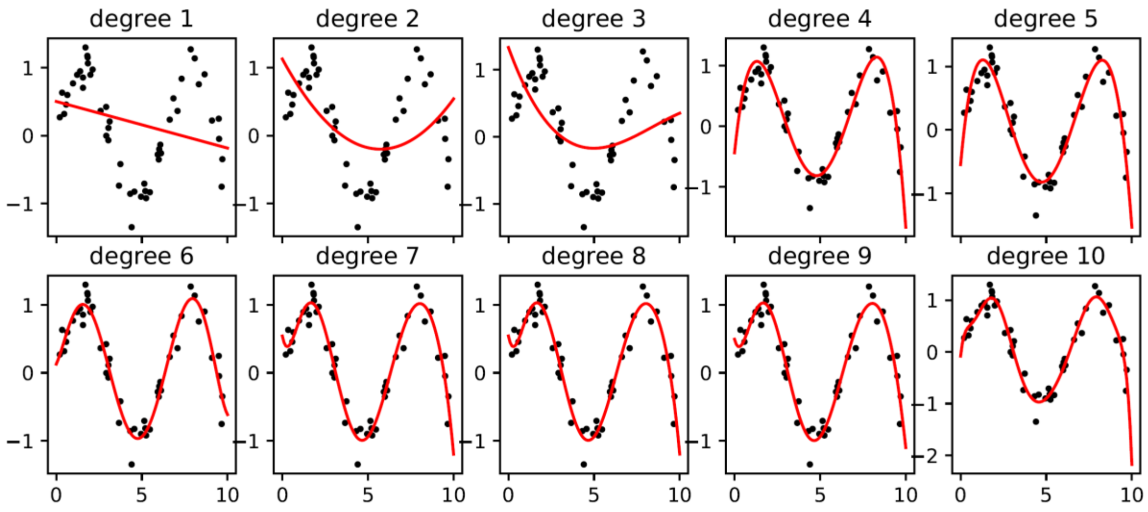
  - include_bias=False prevents the intercept from being added twice
- We also need to reshape our input feature to a column vector:

```
X = x.reshape(-1, 1)
```

Polynomial Regression in Scikit-Learn

- We now fit polynomials of degrees between 1 and 10 to the dataset:

```
fig, axes = plt.subplots(2, 5, figsize=(10, 4), sharex=True)

# Generate evenly spaced values for the test set
X_test = np.linspace(0, 10, 100).reshape(-1, 1)

# Fit polynomials of degree 1 through 10
for ax, degree in zip(axes.flat, range(1, 11)):
    ax.scatter(X, y, color='k', s=5)

    model = PolynomialRegression(degree)
    model.fit(X, y)
    y_test = model.predict(X_test)

    # Plot the predicted polynomial
    ax.plot(X_test, y_test, color='r')
    ax.set_title(f'degree {degree}')
```

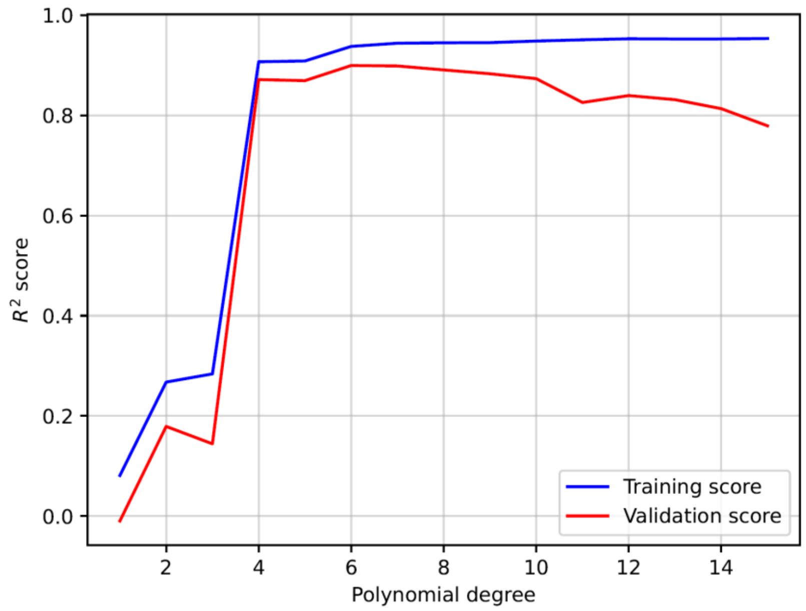


Which polynomial degree should we pick?

Validation Curve

- Shows the training and validation scores across different hyperparameter values
  - Can help identify the best hyperparameter value
- Can be generated using the function [validation_curve]
  - Takes an estimator, X, y, name of hyperparameter, and range of values to test
  - Returns two matrices with training scores and validation scores

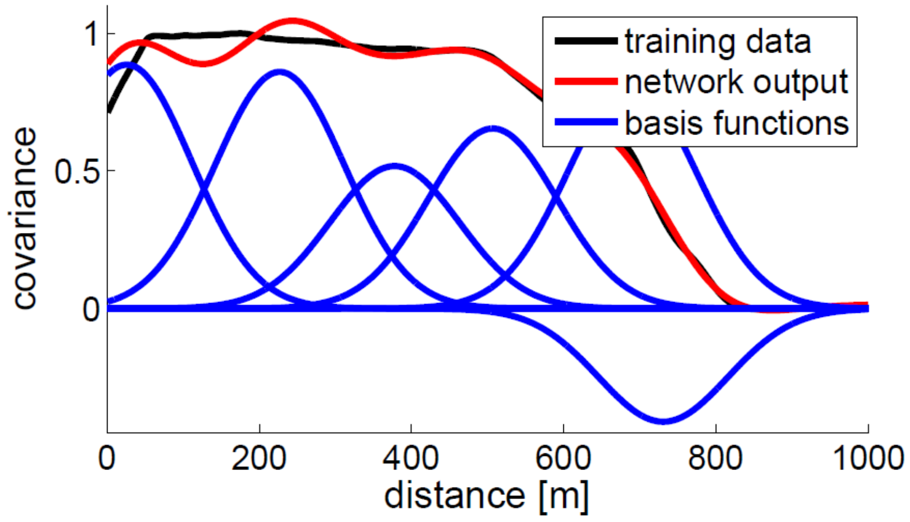
```
from sklearn.model_selection import validation_curve

degrees = np.arange(1, 16)
train_scores, val_scores = validation_curve(
    PolynomialRegression(), X, y, param_name='poly__degree', param_range=degrees
)
plt.plot(degrees, np.mean(train_scores, axis=1), 'b', label='Training score')
plt.plot(degrees, np.mean(val_scores, axis=1), 'r', label='Validation score')
plt.legend()
plt.xlabel('Polynomial degree')
plt.ylabel('$R^2$ score')
plt.grid(alpha=0.5)
```



Other Basis Functions

- Other basis functions may be more effective than polynomials in some cases
- Other common basis functions
  - **Splines**: Piecewise polynomial functions defined by a series of control points
  - **Radial basis functions**: Gaussians centered at different points with different widths

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$



Regularization

- A general technique to mitigate overfitting by penalizing complex models
- Add a component to the cost function that reflects the model's complexity:

$$\mathrm{Cost}(h) = \mathrm{TrainingError}(h) + \lambda \cdot \mathrm{Complexity}(h)$$

  - **Regularization coefficient** λ controls the tradeoff between fitting the data and complexity
- In linear models, complexity is typically measured by the norm of the vector **w**
  - Larger weights make the model more sensitive to the changes in the input
- Two common types of regularization
  - **L1 regularization** uses the L1 norm of **w**

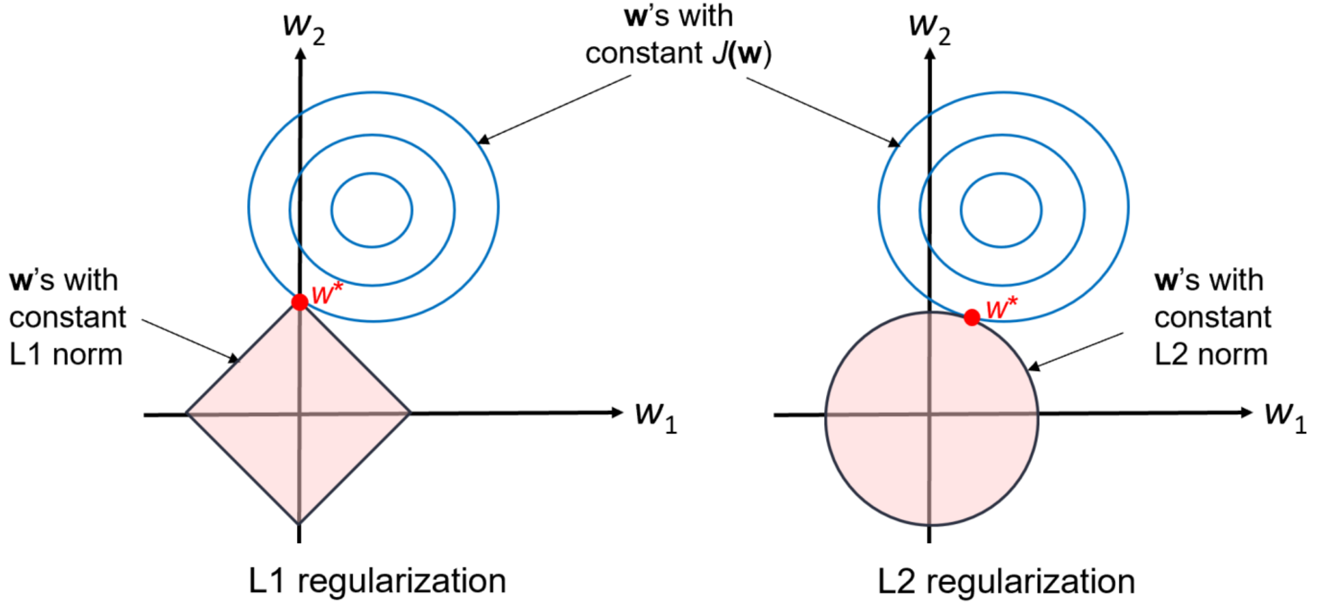
$$\|\mathbf{w}\|_1 = |w_0| + |w_1| + \ldots + |w_d|$$

  - **L2 regularization** (weight decay) uses the L2 norm of **w**

$$\|\mathbf{w}\|_2^2 = w_0^2 + w_1^2 + \ldots + w_d^2$$

Regularization

- L1 regularization shrinks some of the coefficients exactly to 0, encouraging sparsity
  - The derivative of the L1 penalty with respect to w_i is constant driving it to zero
- L2 regularization penalizes large weights more, leading to more uniform shrinkage
  - The derivative of the L2 penalty is $2w_i$, gradually decreasing as w_i gets closer to zero



Ridge Regression

- Adds an L2 regularization term to the least squares cost function

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2 + \lambda\sum_{j=1}^{d} w_j^2$$

  - The bias w_0 is usually not regularized
- The optimal coefficients can be found using a closed-form solution:

$$\mathbf{w}^* = \left(X^T X + \lambda I\right)^{-1} X^T \mathbf{y}$$

  - or using gradient descent
- In both methods, it is important to standardize the features
  - since regularization is sensitive to the scale of the inputs

Ridge Regression in Scikit-Learn

- In Scikit-Learn, the Ridge class can be used for ridge regression

```
class sklearn.linear_model.Ridge(alpha=1.0, *, fit_intercept=True, copy_X=True,
max_iter=None, tol=0.0001, solver='auto', positive=False,
random_state=None)
```
[source]

  - Automatically chooses the best solver based on the data characteristics
  - The parameter **alpha** specifies the regularization strength (λ)
- For an SGD solution, use SGDRegressor with **penalty='l2'** (the default)
  - Also has a parameter **alpha** that specifies the regularization strength

Ridge Regression in Scikit-Learn

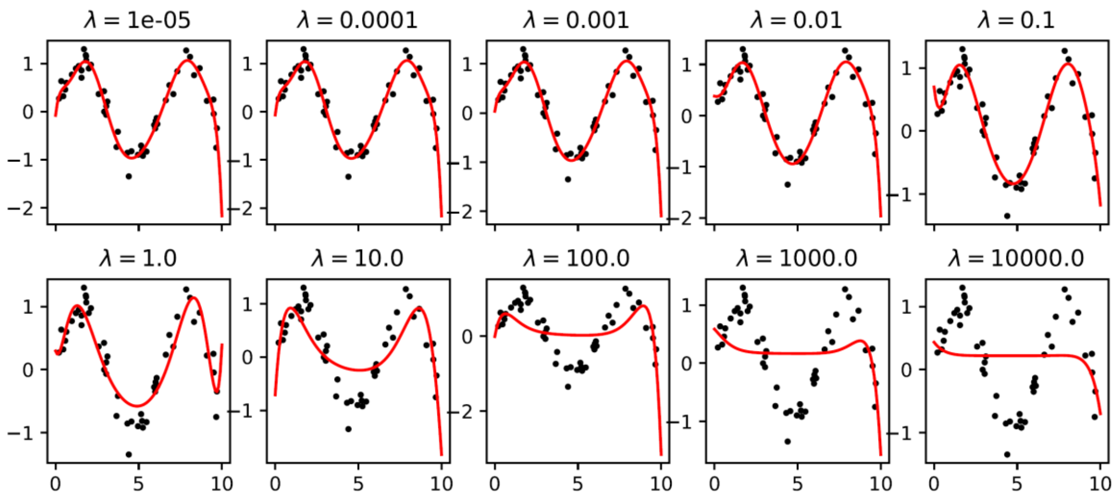
- To demonstrate the class, we will use the same sine data generated previously
- We first define a pipeline that combines PolynomialFeatures with Ridge regression:

```
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import Ridge
from sklearn.pipeline import Pipeline

def RidgePolynomialRegression(degree, alpha=1):
    return Pipeline([('scaler', StandardScaler()),
                     ('poly', PolynomialFeatures(degree)),
                     ('ridge', Ridge(alpha))])
```

Ridge Regression in Scikit-Learn

- We now a polynomial of degree 10 using increasing regularization coefficients

```
fig, axes = plt.subplots(2, 5, figsize=(10, 4), sharex=True)
plt.subplots_adjust(hspace=0.3)

X_test = np.linspace(0, 10, 100).reshape(-1, 1)

alpha = 0.00001
for ax in axes.flat:
    ax.scatter(X, y, color='k', s=5)

    model = RidgePolynomialRegression(degree=10, alpha=alpha)
    model.fit(X, y)
    y_test = model.predict(X_test)

    ax.plot(X_test, y_test, color='r')
    ax.set_title(f'$\lambda = ${alpha}')
    alpha *= 10
```



Ridge Regression in Scikit-Learn

- Examining the polynomial coefficients during training:

```
[-0.965 -0.274  3.545 -1.509 -1.056  3.215 -1.405 -1.849  1.     0.336 -0.188]
[-0.964 -0.282  3.544 -1.467 -1.067  3.15  -1.381 -1.812  0.985  0.328 -0.184]
[-0.96  -0.341  3.531 -1.132 -1.133  2.632 -1.215 -1.514  0.869  0.271 -0.159]
[-0.925 -0.516  3.279 -0.125 -0.903  1.062 -1.028 -0.602  0.615  0.093 -0.093]
[-0.813 -0.518  2.522  0.226 -0.148  0.148 -0.955  0.069  0.325 -0.055 -0.016]
[-0.552 -0.394  1.274 -0.021  0.376  0.097 -0.217  0.199 -0.201 -0.082  0.069]
[-0.227 -0.194  0.427 -0.039  0.294  0.017  0.093  0.008 -0.133  0.016  0.01 ]
[ 0.034 -0.05   0.083 -0.025  0.079 -0.014  0.06  -0.005  0.021  0.015 -0.022]
[ 0.166 -0.006  0.012 -0.002  0.015  0.001  0.017  0.003  0.015 -0.003 -0.007]
[ 0.218 -0.001  0.001 -0.     0.002 -0.     0.002 -0.     0.002 -0.002 -0.001]
```

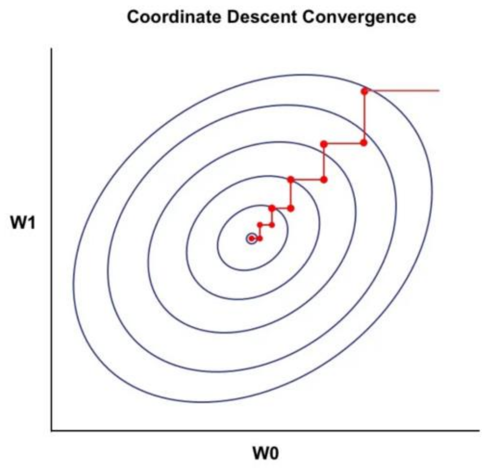
- The coefficients shrink over time except for the bias (w_0)

Lasso Regression

- Adds an L1 regularization term to the least squares cost function

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2 + \lambda\sum_{j=1}^{d}|w_j|$$

- It has no closed-form solution due to the non-differentiability at zero
- Solved using iterative optimization methods such as **coordinate descent**
  - Minimizing the cost function one coefficient at a time, while keeping all others fixed



Lasso Regression in Scikit-Learn

- The Lasso class implements a lasso regression using coordinate descent:

```
class sklearn.linear_model.Lasso(alpha=1.0, *, fit_intercept=True, precompute=False,
copy_X=True, max_iter=1000, tol=0.0001, warm_start=False, positive=False,
random_state=None, selection='cyclic')
```

[source]

  - The **alpha** parameter specifies the regularization strength (λ)
- For an SGD solution, use SGDRegressor with **penalty='l1'**

Lasso Regression in Scikit-Learn

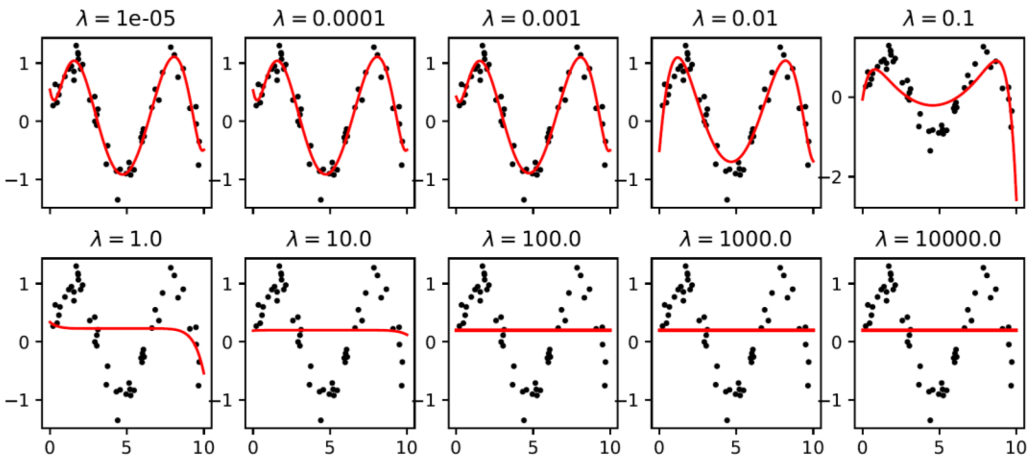
- To demonstrate the class, we will use the same sine data generated previously
- We first define a pipeline that combines PolynomialFeatures with Lasso regression:

```
from sklearn.linear_model import Lasso

def LassoPolynomialRegression(degree, alpha=1):
    return Pipeline([('scaler', StandardScaler()),
                     ('poly', PolynomialFeatures(degree)),
                     ('lasso', Lasso(alpha))])
```

Lasso Regression in Scikit-Learn

- We now fit a polynomial of degree 10 using increasing regularization coefficients

```
fig, axes = plt.subplots(2, 5, figsize=(10, 4), sharex=True)
plt.subplots_adjust(hspace=0.3)

X_test = np.linspace(0, 10, 100).reshape(-1, 1)

alpha = 0.00001
for ax in axes.flat:
    ax.scatter(X, y, color='k', s=5)

    model = LassoPolynomialRegression(degree=10, alpha=alpha)
    model.fit(X, y)
    y_test = model.predict(X_test)

    ax.plot(X_test, y_test, color='r')
    ax.set_title(f'$\lambda = ${alpha}')
    alpha *= 10
```



Lasso Regression in Scikit-Learn

- Examining the polynomial coefficients during training:

```
[-0.889 -0.579  3.158  0.297 -1.36   0.233 -0.033 -0.038  0.035 -0.027  0.013]
[-0.887 -0.577  3.146  0.295 -1.348  0.23  -0.035 -0.037  0.034 -0.027  0.013]
[-0.868 -0.552  3.029  0.282 -1.237  0.203 -0.051 -0.028  0.028 -0.025  0.013]
[-0.687 -0.283  1.92   0.058 -0.149  0.035 -0.267  0.043 -0.    -0.009  0.01 ]
[-0.205 -0.046  0.654 -0.     0.    -0.     0.    -0.    -0.     0.01 -0.011]
[ 0.228 -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.002 -0.   ]
[ 0.2 -0.   0.  -0.   0.  -0.  -0.  -0.  -0.  -0.  -0. ]
[ 0.196 -0.     0.    -0.     0.    -0.    -0.    -0.    -0.    -0. -0.   ]
[ 0.196 -0.     0.    -0.     0.    -0.    -0.    -0.    -0.    -0. -0.   ]
[ 0.196 -0.     0.    -0.     0.    -0.    -0.    -0.    -0.    -0. -0.   ]
```

- The coefficients shrink to exactly zero, resulting in a sparse solution
- Thus, lasso be used as a method for **feature selection**

Elastic Net

- A hybrid approach that combines both L1 and L2 penalties:

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2 + r\lambda\sum_{j=1}^{d}|w_j| + \frac{1-r}{2}\lambda\sum_{j=1}^{d}w_j^2$$

  - $0 \leq r \leq 1$ is a **mixing parameter** that controls the balance between the two penalties
  - When r = 0 it reduces to ridge regression; when r = 1 to lasso regression
- Also solved using **coordinate descent**
- The ElasticNet class implements the elastic net model

```
class sklearn.linear_model.ElasticNet(alpha=1.0, *, l1_ratio=0.5,
fit_intercept=True, precompute=False, max_iter=1000, copy_X=True, tol=0.0001,
warm_start=False, positive=False, random_state=None, selection='cyclic')      [source]
```

  - **alpha** is the regularization parameter (λ)
  - **l1_ratio** is the ratio of the L1 penalty (*r*)

Elastic Net in Scikit-Learn

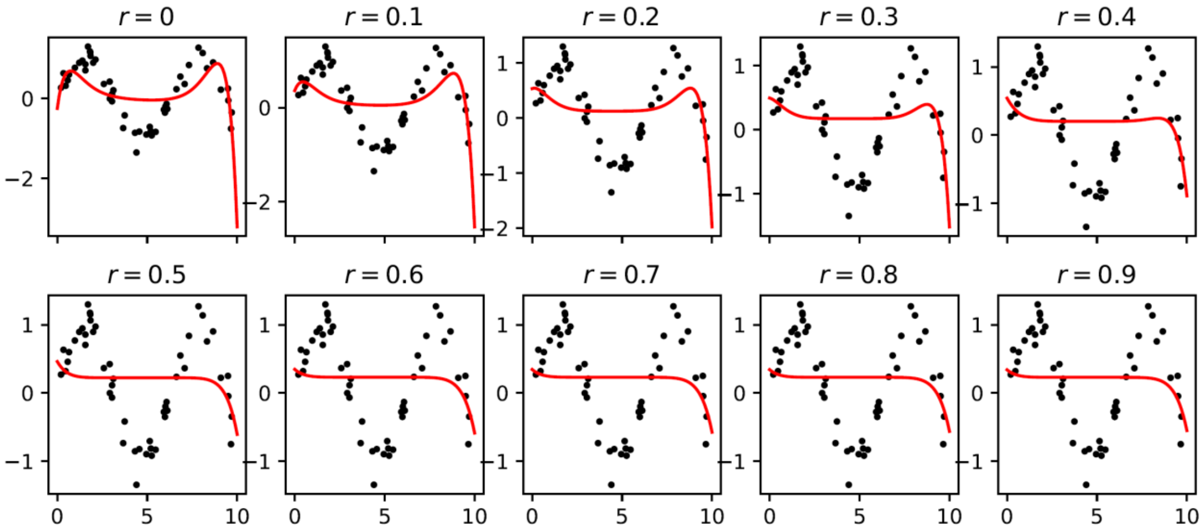
- To demonstrate the class, we will use the same sine data generated previously
- We first define a pipeline that combines PolynomialFeatures with ElasticNet:

```
from sklearn.linear_model import ElasticNet

def ElasticNetPolynomialRegression(degree, alpha=1, l1_ratio=0.5):
    return Pipeline([('scaler', StandardScaler()),
                     ('poly', PolynomialFeatures(degree)),
                     ('elastic_net', ElasticNet(alpha=alpha, l1_ratio=l1_ratio))])
```

Elastic Net in Scikit-Learn

- We now fit a polynomial of degree 10 using increasing mixing ratios

```
fig, axes = plt.subplots(2, 5, figsize=(10, 4), sharex=True)
plt.subplots_adjust(hspace=0.3)

l1_ratio = 0

for ax in axes.flat:
    ax.scatter(X, y, color='k', s=5)

    model = ElasticNetPolynomialRegression(degree=10, alpha=1,
                                           l1_ratio=l1_ratio)
    model.fit(X, y)
    y_test = model.predict(X_test)

    ax.plot(X_test, y_test, color='r')
    ax.set_title(f'$r = ${round(l1_ratio, 1)}')
    l1_ratio += 0.1
```



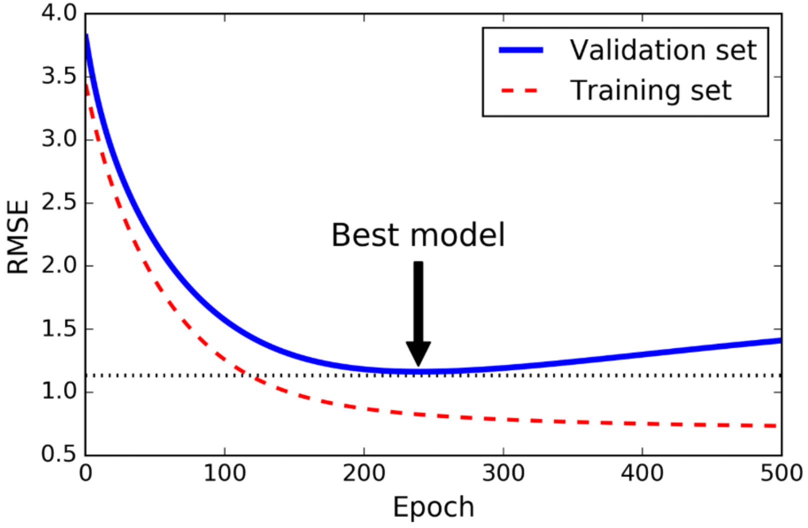
- As *r* increases, the L1 penalty has more influence, leading to increased sparsity

Which Regression to Choose?

Model	Advantages	Disadvantages
Ridge	- Retains all features in the model. - Has a closed-form solution. - Distributes the regularization across all coefficients. - Handles multicollinearity well by shrinking the coefficients of correlated predictors uniformly.	- Does not perform feature selection, resulting in non-sparse models. - May overfit in high-dimensional settings due to inclusion of all features.
Lasso	- Performs feature selection by shrinking some coefficients to zero, leading to simpler models. - Enhances model interpretability by removing irrelevant features.	- In the presence of multicollinearity, lasso may behave unpredictably by arbitrarily selecting one feature from the correlated predictors and shrinking the others to zero. - Does not have a closed-form solution.
Elastic Net	- Combines the strengths of ridge and lasso. - Performs feature selection while maintaining group-wise shrinkage. - Offers more stable feature selection and better generalization in high-dimensional settings.	- More computationally intensive than lasso or ridge alone. - Requires tuning both λ and the mixing ratio r.

Early Stopping

- Another regularization technique to mitigate overfitting
- Can be used with iterative optimization algorithms such as gradient descent
- Allocate part of the training set to validation (typically 10%)
- During training, monitor the performance on the validation set
- Once the validation performance stops improving, the training is terminated



Early Stopping in Scikit-Learn

- The SGD classes in Scikit-Learn support early stopping

```
class sklearn.linear_model.SGDRegressor(loss='squared_error', *, penalty='l2',
alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001,
shuffle=True, verbose=0, epsilon=0.1, random_state=None, learning_rate='invscaling',
eta0=0.01, power_t=0.25, early_stopping=False, validation_fraction=0.1,
n_iter_no_change=5, warm_start=False, average=False)
```

[source]

  - **early_stopping**: Set to True to enable early stopping
  - **validation_fraction**: The proportion of the training data to reserve for validation
  - **n_iter_no_change**: Number of consecutive iterations (epochs) with no improvement on the validation set before training is halted
  - **tol**: The minimum change in the validation loss required to qualify as an improvement

Linear Regression Summary

Algorithm	Scikit-Learn Class
Linear regression (closed form)	LinearRegression
Linear regression (SGD)	SGDRegressor
Polynomial regression (closed form)	PolynomialFeatures + LinearRegression
Polynomial regression (SGD)	PolynomialFeatures + SGDRegressor
Ridge regression (closed form)	Ridge
Ridge regression (SGD)	SGDRegressor (penalty='l2')
Lasso regression (closed form)	Lasso
Lasso regression (SGD)	SGDRegressor (penalty='l1')
Elastic net (closed form)	ElasticNet
Elastic net (SGD)	SGDRegressor (penalty='elasticnet')

- Other regression techniques will be discussed later (e.g., regression trees)