Regression

ECE30007 Intro to AI Project

한동대학교
HANDONG GLOBAL UNIVERSITY

Contents

- Linear Regression
- Regression in scikit-Learn
  - simple linear regression
  - multiple regression
  - Polynomial
  - Lasso
  - Ridge

Machine Learning Algorithms

https://medium.com/mathpresso/mathpresso-%EB%A8%B8%EC%8B%A0-%EB%9F%AC%EB%8B%9D-%EC%8A%A4%ED%84%B0%EB%94%94-2-%EB%A8%B8%EC%8B%A0-%EB%9F%AC%EB%8B%9D-machine-learning-%EC%9D%98-%EC%B2%AB%EA%B1%B8%EC%9D%8C-58f5d568d7c2

Linear Regression

- What is regression?
  - predicting the target y given a D-dimensional vector x.
  - or estimating the relationship between x and y.
  - or finding a function from x to y.
    - x : independent variable, features, predictors
    - y : dependent variable, outcome

  - Linear regression attempts to model the relationship between two variables by fitting a linear equation to observed data.

$$\hat{y}(w, x) = w_0 + w_1 x_1 + \ldots + w_p x_p$$

where $w = (w_1, \ldots, w_p)$ as coefficient and w_0 as intercept (or bias)

https://scikit-learn.org/stable/modules/linear_model.html

Some keywords

- ***X***: Independent variable (or predictor, explanatory variable)
  - The variable you are using to predict the other variable's value.
- ***Y***: Dependent variable (or response)
  - The variable you want to predict.

$$Y = f(X) = f(X;w) = f_w(X)$$

https://developers.google.com/machine-learning/glossary

Some keywords

- ***Model***: The representation of what a machine learning system has learned from the training data.

$$\hat{y}(w, x) = w_0 + w_1 x_1 + \ldots + w_p x_p$$

- ***Weight***: A coefficient for a feature in a linear model.
- ***Bias***: An intercept or offset from an origin. Bias (also known as the bias term) is referred to as b or w0 in machine learning models.

https://developers.google.com/machine-learning/glossary

Some keywords

- ***Loss***: difference between the values that a model is predicting and the actual values of the labels.
- ***Gradient decent*** : A technique to minimize loss by computing the gradients of loss with respect to the model's parameters, conditioned on training data.
  - Informally, gradient descent iteratively adjusts parameters, gradually finding the best combination of weights and bias to minimize loss.
- ***Back propagation***: The primary algorithm for performing gradient descent on neural networks.
- ***Learning rate***: A scalar used to train a model via gradient descent. During each iteration, the gradient descent algorithm multiplies the learning rate by the gradient.
- ***Epoch***: A full training pass over the entire dataset such that each example has been seen once.

https://developers.google.com/machine-learning/glossary

loss function

- given observed inputs, $X = \{x_1, \dots, x_N\}$, and targets, $\mathbf{Y} = [y_1, \dots, y_N]^{\mathrm{T}}$
- we can simply minimize errors defined by

$$E(w) = \sum_{n=1}^{N} \left(y_n - f(x_n)\right)^2$$

- $E(w)$ can be defined in different ways.

simple problem for simple linear regression

- A simple problem
  - Given X and Y, what is the exact number y for x = 4?
    - X = [1, 2, 3]
    - Y = [2, 4, 6]
  - Obvious answer is a number y = 8.

- What is the model for the problem?
  - Given a simple model $\hat{y}(w, x) = w_0 + w_1 x_1$
  - Based on the data X and Y, the possible model (with parameters) would be $y = 2x$
  - But how does a computer find the model?

forward pass and loss: implementation

- Let's try to find *w* for a simple linear model *f(w,x)*

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

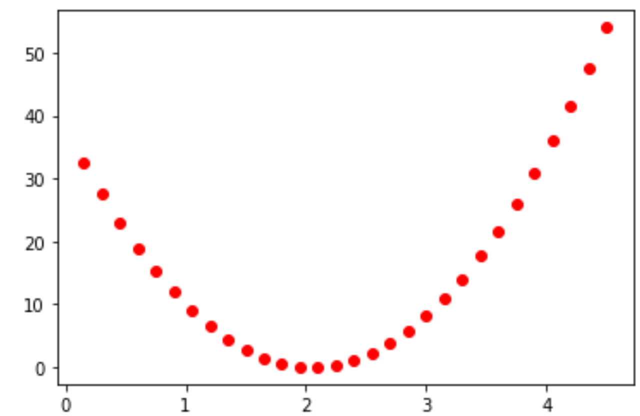
```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

#Our model forward pass
def forward(x):
    return x * w

# Loss function
def loss(y_pred, y):
    return (y_pred - y) * (y_pred - y)

w = 0.00  # initial w
for epoch in range(30):
    for x, y in zip(x_data, y_data):
        y_pred = forward(x)
        l = loss(y_pred, y)
        w = w +0.05
    print("Epcoh", epoch, "\t(x,y): ", x, y, "\tw=", '%.2f'%w, "\tloss=", '%.2f'%l)
    plt.scatter(w,l, color = 'r')
```

note that this is **NOT** a learning process

forward pass and loss

- Let's try find *w* for a simple linear model *f(w,x)*

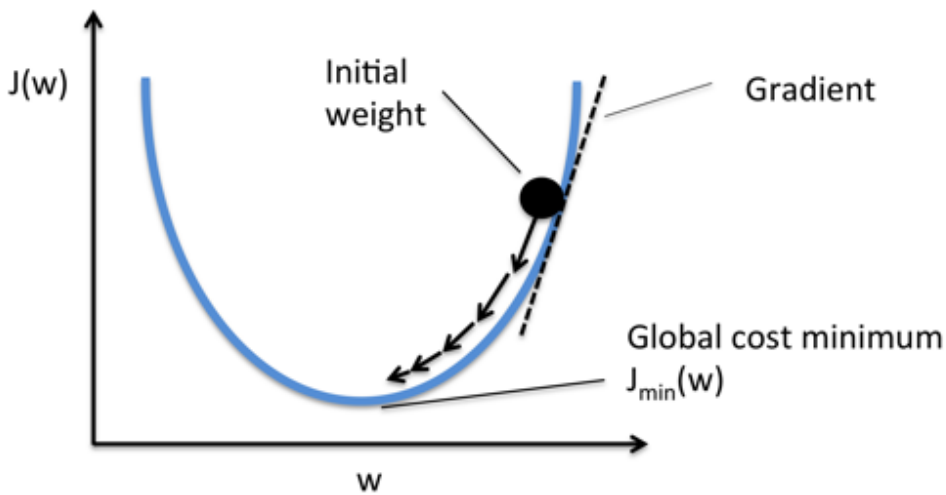
Epcoh 0	(x,y): 3.0 6.0	w= 0.15	loss= 32.49
Epcoh 1	(x,y): 3.0 6.0	w= 0.30	loss= 27.56
Epcoh 2	(x,y): 3.0 6.0	w= 0.45	loss= 23.04
Epcoh 3	(x,y): 3.0 6.0	w= 0.60	loss= 18.92
Epcoh 4	(x,y): 3.0 6.0	w= 0.75	loss= 15.21
Epcoh 5	(x,y): 3.0 6.0	w= 0.90	loss= 11.90
Epcoh 6	(x,y): 3.0 6.0	w= 1.05	loss= 9.00
Epcoh 7	(x,y): 3.0 6.0	w= 1.20	loss= 6.50
Epcoh 8	(x,y): 3.0 6.0	w= 1.35	loss= 4.41
Epcoh 9	(x,y): 3.0 6.0	w= 1.50	loss= 2.72
Epcoh 10	(x,y): 3.0 6.0	w= 1.65	loss= 1.44
Epcoh 11	(x,y): 3.0 6.0	w= 1.80	loss= 0.56
Epcoh 12	(x,y): 3.0 6.0	w= 1.95	loss= 0.09
Epcoh 13	(x,y): 3.0 6.0	w= 2.10	loss= 0.02
Epcoh 14	(x,y): 3.0 6.0	w= 2.25	loss= 0.36
Epcoh 15	(x,y): 3.0 6.0	w= 2.40	loss= 1.10
Epcoh 16	(x,y): 3.0 6.0	w= 2.55	loss= 2.25
Epcoh 17	(x,y): 3.0 6.0	w= 2.70	loss= 3.80
Epcoh 18	(x,y): 3.0 6.0	w= 2.85	loss= 5.76
Epcoh 19	(x,y): 3.0 6.0	w= 3.00	loss= 8.12
Epcoh 20	(x,y): 3.0 6.0	w= 3.15	loss= 10.89
Epcoh 21	(x,y): 3.0 6.0	w= 3.30	loss= 14.06
Epcoh 22	(x,y): 3.0 6.0	w= 3.45	loss= 17.64
Epcoh 23	(x,y): 3.0 6.0	w= 3.60	loss= 21.62
Epcoh 24	(x,y): 3.0 6.0	w= 3.75	loss= 26.01
Epcoh 25	(x,y): 3.0 6.0	w= 3.90	loss= 30.80
Epcoh 26	(x,y): 3.0 6.0	w= 4.05	loss= 36.00
Epcoh 27	(x,y): 3.0 6.0	w= 4.20	loss= 41.60
Epcoh 28	(x,y): 3.0 6.0	w= 4.35	loss= 47.61
Epcoh 29	(x,y): 3.0 6.0	w= 4.50	loss= 54.02



At 13^{th} epoch, w = 2.10 yields the lowest loss
But is it optimal?

Gradient decent approach

- Gradient of *J(w)* at a specific *w* provides important information.
- The direction to move can be determined from the gradient.
- In the following figure, gradient at the black circle, is positive,
  and we can guess that w should decrease to reach the optimal point
  yielding the minimum loss (or error).

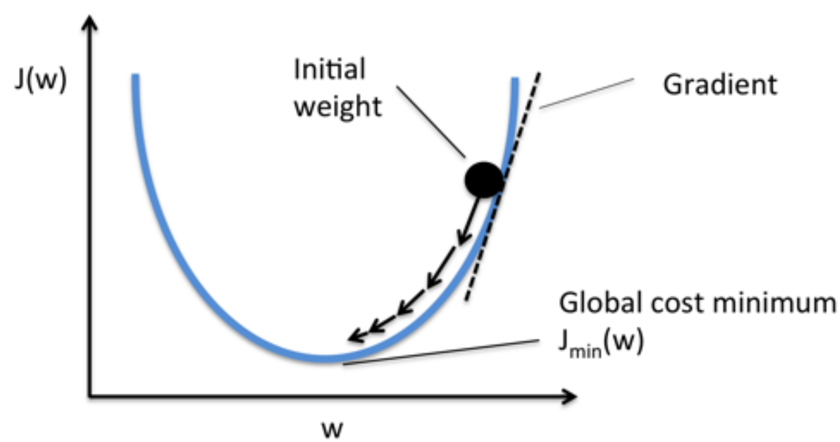


Gradient decent approach

- For a given model $y = f(x, w, b)$
  and loss function $J(w, b, y)$,
  the next w_{i+1} and b_{i+1} can be obtained by

$$w_{i+1} = w_i - \alpha_i \nabla_w J(w_i, b_i, y) \quad \alpha_i\text{: Learning rate}$$
$$b_{i+1} = b_i - \beta_i \nabla_b J(w_i, b_i, y) \quad \beta_i\text{: Learning rate}$$

- In our example,
  - $J(w) = (wx - y)^2$
    $w_{i+1} = w_i - \underbrace{\gamma_i}_{\text{learning rate}}[\overbrace{\boxed{2x(wx - y)}}^{\text{gradient}}]$



Gradient decent: implementation

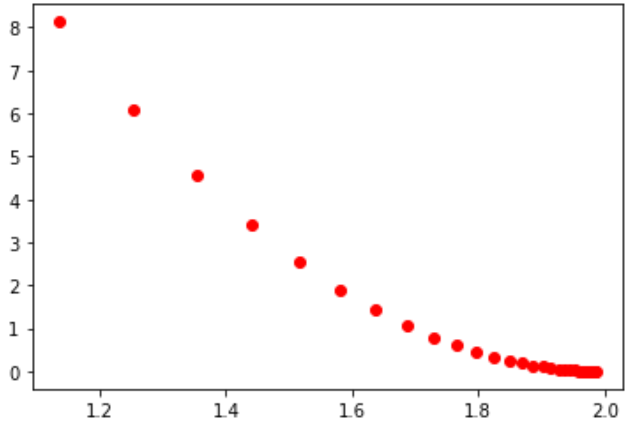
```
# Training Data
x_data = [1, 2, 3]
y_data = [2, 4, 6]

# compute gradient
def gradient(x, y):  # d_loss/d_w
    return 2 * x * (x * w - y)

w = 1.00  # initial w
for epoch in range(30):
    for x, y in zip(x_data, y_data):
        y_pred = forward(x)
        l = loss(y_pred, y)
        grad = gradient(x, y)
        w = w - 0.005 * grad
    print("Epoch", epoch, "\t(x,y): ", x, y, "\tw=", '%.2f'%w, "\tloss=", '%.2f'%l, '\tGrad=', '%.2f'%grad)

    plt.scatter(w,l, color = 'r')
```

$$w_{i+1} = w_i - \gamma_i[2x(wx - y)]$$

this is a learning process

experiments of learning

W converges to 2.0.




```
Epoch 0     (x,y):  3 6   w= 1.14   loss= 8.13   Grad= -17.11
Epoch 1     (x,y):  3 6   w= 1.25   loss= 6.08   Grad= -14.80
Epoch 2     (x,y):  3 6   w= 1.35   loss= 4.55   Grad= -12.80
Epoch 3     (x,y):  3 6   w= 1.44   loss= 3.40   Grad= -11.07
Epoch 4     (x,y):  3 6   w= 1.52   loss= 2.54   Grad= -9.57
Epoch 5     (x,y):  3 6   w= 1.58   loss= 1.90   Grad= -8.28
Epoch 6     (x,y):  3 6   w= 1.64   loss= 1.42   Grad= -7.16
Epoch 7     (x,y):  3 6   w= 1.69   loss= 1.06   Grad= -6.19
Epoch 8     (x,y):  3 6   w= 1.73   loss= 0.80   Grad= -5.36
Epoch 9     (x,y):  3 6   w= 1.77   loss= 0.60   Grad= -4.63
Epoch 10    (x,y):  3 6   w= 1.80   loss= 0.45   Grad= -4.01
Epoch 11    (x,y):  3 6   w= 1.82   loss= 0.33   Grad= -3.46
Epoch 12    (x,y):  3 6   w= 1.85   loss= 0.25   Grad= -3.00
Epoch 13    (x,y):  3 6   w= 1.87   loss= 0.19   Grad= -2.59
Epoch 14    (x,y):  3 6   w= 1.89   loss= 0.14   Grad= -2.24
Epoch 15    (x,y):  3 6   w= 1.90   loss= 0.10   Grad= -1.94
Epoch 16    (x,y):  3 6   w= 1.92   loss= 0.08   Grad= -1.68
Epoch 17    (x,y):  3 6   w= 1.93   loss= 0.06   Grad= -1.45
Epoch 18    (x,y):  3 6   w= 1.94   loss= 0.04   Grad= -1.25
Epoch 19    (x,y):  3 6   w= 1.95   loss= 0.03   Grad= -1.08
Epoch 20    (x,y):  3 6   w= 1.95   loss= 0.02   Grad= -0.94
Epoch 21    (x,y):  3 6   w= 1.96   loss= 0.02   Grad= -0.81
Epoch 22    (x,y):  3 6   w= 1.96   loss= 0.01   Grad= -0.70
Epoch 23    (x,y):  3 6   w= 1.97   loss= 0.01   Grad= -0.61
Epoch 24    (x,y):  3 6   w= 1.97   loss= 0.01   Grad= -0.52
Epoch 25    (x,y):  3 6   w= 1.98   loss= 0.01   Grad= -0.45
Epoch 26    (x,y):  3 6   w= 1.98   loss= 0.00   Grad= -0.39
Epoch 27    (x,y):  3 6   w= 1.98   loss= 0.00   Grad= -0.34
Epoch 28    (x,y):  3 6   w= 1.99   loss= 0.00   Grad= -0.29
Epoch 29    (x,y):  3 6   w= 1.99   loss= 0.00   Grad= -0.25
```

prediction with the trained model

- Prediction (after 30 epochs)

```
# After training
print("Predicted score (after training)",  "When x=4, y will be : ", forward(4))
print("Predicted score (after training)",  "When x=6, y will be : ", forward(6))
```

```
Predicted score (after training) When x=4, y will be :  7.94865542184356
Predicted score (after training) When x=6, y will be :  11.922983132765339
```

- Very close to the answers.
  More training may produce the exact answer.

(after 100 epochs)

```
# After training
print("Predicted score (after training)",  "When x=4, y will be : ", forward(4))
print("Predicted score (after training)",  "When x=6, y will be : ", forward(6))
```

```
Predicted score (after training) When x=4, y will be :  7.999998019193797
Predicted score (after training) When x=6, y will be :  11.999997028790697
```

implementation of simple linear regression

- Step by Step

$$E(w) = \sum_{n=1}^{N} (y_n - (x * w + b))^2$$

```
x_data = [1.0, 2.0, 3.0,4.0, 6.0]
y_data = [2.0, 4.0, 6.0, 8.0, 12.0]
w=0.0
b=0.0

n_data = len(x_data)
epochs = 100
learning_rate = 0.01
for i in range(epochs):
  print("Process " , i+1)
  for x_i, y_i in zip(x_data,y_data):
        y_hat = x_i * w + b
        # using MSE
        loss = ((y_hat - y_i) ** 2 ) / n_data
        grad_w = ((w * x_i - y_i + b) * 2 * x_i)/ n_data
        grad_b = ((w * x_i - y_i + b) * 2)/ n_data

        w -= learning_rate * grad_w
        b -= learning_rate * grad_b
        print("weight  = ",w, "₩t₩t","bias  = ",b)
  plt.scatter(w,loss, color = 'r')
```

Exercise 1 – speed of sound

- Speed of sound at temperature

Temperature X(°C)	Speed Y(m/s)
0	331
1	331.6
2	332.2
3	332.8

- Based on the data (temperature, speed),
  what is the prediction of speed at temperature of 20°C.

Exercise 1 – speed of sound

```
x_data = [0.0, 1.0, 2.0, 3.0]
y_data = [331.0, 331.6, 332.2, 332.8]
w=0.0
b=0.0

epochs = 1000
learning_rate = 0.03
for i in range(epochs):
    for x, y in zip(x_data,y_data):
        y_pred = x * w + b
        loss = ((y_pred - y) ** 2 ) # MSE
        grad_w = ((w * x + b - y) * 2 * x)
        grad_b = ((w * x + b - y) * 2)
        w -= learning_rate * grad_w
        b -= learning_rate * grad_b
```

sklearn Linear Regression

- Import
  - `from sklearn.linear_model import LinearRegression`

- Methods

fit(X, y[, sample_weight])	Fit linear model.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model.
score(X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

Real Estate Data

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.linear_model import LinearRegression
```

```
df = pd.read_excel('data_set_train.xlsx')
df.head()
```

	aptnm(아파트 이름)	yyyyqrt(거래년도 분기별)	price(가격)	con_year(건축년도)	dong(동)	area(면적)	floor(층수)	Latitude(위도)	Longtitude(경도)	gdp	...	dis_su 하철
0	강남역우정에쉐르	2006Q1	9000	2004	역삼동	17.23	7	37.494204	127.043545	225613	...	849
1	강남역우정에쉐르	2006Q1	9000	2004	역삼동	17.23	7	37.494204	127.043545	225613	...	849
2	개포주공1단지	2006Q1	73000	1982	개포동	50.38	3	37.478407	127.061375	225613	...	1486
3	개포주공1단지	2006Q1	70000	1982	개포동	50.64	5	37.484609	127.067275	225613	...	1160
4	개포주공1단지	2006Q1	40000	1982	개포동	35.44	4	37.482445	127.051278	225613	...	650

5 rows × 29 columns

Real Estate Data

```
df_2017q1=df[df['yyyyqrt(거래년도 분기별)']=='2017Q1']
df_2017q1.corr().head(5)
```

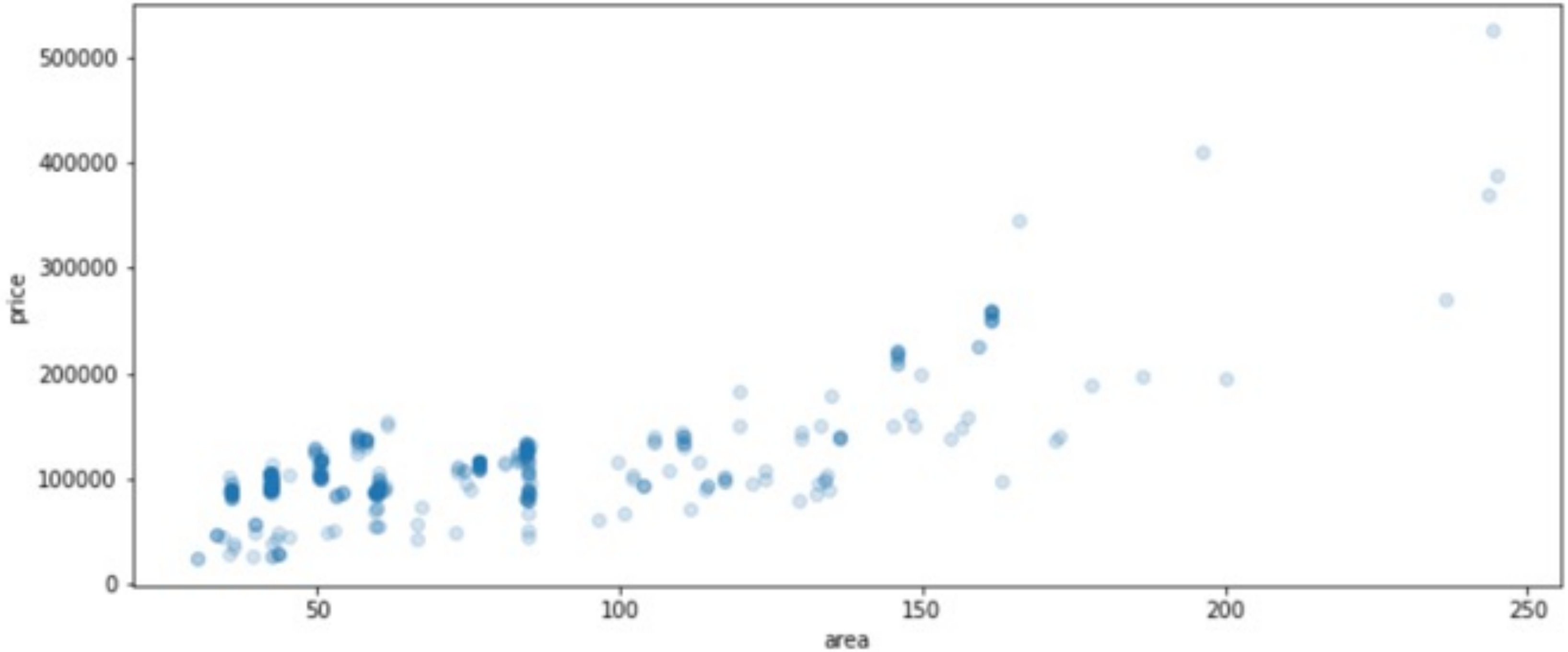
	price(가격)	con_year(건축년도)	area(면적)	floor(층수)	Latitude(위도)	Longtitude(경도)	gdp	e_grwth(경제성장률)	Seoul_I.rate(지가상승률)	house_rate(담보대출금리)	...	dis_hospital(종합 병원과의 거리)
price(가격)	1.000000	0.195676	0.750915	0.135630	0.257352	0.010404	NaN	5.541402e-16	-6.566975e-16	2.531704e-16	...	-0.145817
con_year(건축년도)	0.195676	1.000000	0.523815	0.397941	0.489696	-0.430093	NaN	-5.761812e-15	-7.480498e-15	-7.037531e-15	...	-0.496689
area(면적)	0.750915	0.523815	1.000000	0.248797	0.545653	-0.212950	NaN	6.069479e-15	6.285195e-15	7.559905e-15	...	-0.455760
floor(층수)	0.135630	0.397941	0.248797	1.000000	0.167945	-0.156736	NaN	2.389365e-15	3.082374e-15	-3.251218e-16	...	-0.428012
Latitude(위도)	0.257352	0.489696	0.545653	0.167945	1.000000	-0.431871	NaN	1.052426e-11	1.052426e-11	1.052535e-11	...	-0.578459

5 rows × 24 columns

Price prediction by Linear regression

- Area vs. Price

```
# Area vs. Price
plt.figure(figsize=(12, 5))
plt.scatter(df_2017q1['area(면적)'], df_2017q1['price(가격)'], alpha=0.1)
plt.xlabel('area')
plt.ylabel('price')
plt.show()
```



Price prediction by Linear regression

- Split data into training set and test set

```
from sklearn.model_selection import train_test_split

#Split data into training set and test set
X = np.array(df_2017q1['area(면적)']).reshape(-1, 1)
y = np.array(df_2017q1['price(가격)']).reshape(-1, 1)
train_x, test_x, train_y, test_y = train_test_split(X, y, test_size = 0.10,random_state=2)

print("train_x shape  = ",train_x.shape, "train_y shape  = ", train_y.shape)
print("test_x shape  = ",test_x.shape, "test_y shape  = ", test_y.shape)
```

```
train_x shape  =  (702, 1) train_y shape  =  (702, 1)
test_x shape  =  (78, 1) test_y shape  =  (78, 1)
```

Price prediction by Linear regression

- Training and testing

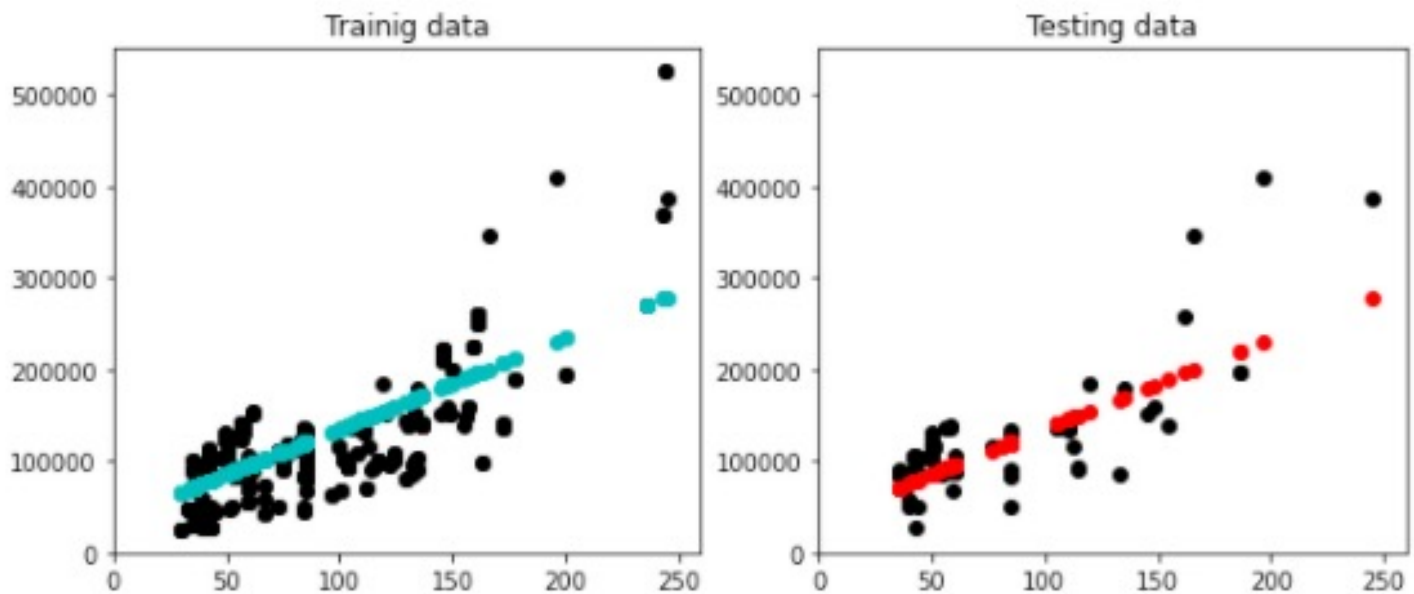
```
# Training
simple_lin_reg = LinearRegression(normalize = True)
simple_lin_reg.fit(train_x, train_y)
print("Fitting results: ", 'b = ', simple_lin_reg.intercept_, 'w = ', simple_lin_reg.coef_ )

# Prediction
test_y_pred=simple_lin_reg.predict(test_x)
print("Testing results: ", 'score = ', simple_lin_reg.score(test_x, test_y))

# Visulization
plt.figure(figsize=(10, 4))
plt.subplot(1,2,1)
plt.scatter(train_x,train_y,color='k')
plt.scatter(train_x,simple_lin_reg.predict(train_x),color='c')
plt.title('Trainig data'); plt.xlim(0,260); plt.ylim(0,550000);

plt.subplot(1,2,2)
plt.scatter(test_x, test_y, color ='k')
plt.scatter(test_x, test_y_pred,color='r')
plt.title('Testing data'); plt.xlim(0,260); plt.ylim(0,550000);
```

```
Fitting results:  b =  [35857.48945078] w =  [[991.57606614]]
Testing results:  score =  0.5938353309667359
```

Price prediction by Linear regression

- Training and testing

```
Fitting results:  b =  [35857.48945078] w =  [[991.57606614]]
Testing results:  score =  0.5938353309667359
```



- Score = $R^2 = (1-u/v)$
- u is the residual sum of squares ((y_true - y_pred) ** 2).sum()
- v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum().
- The best possible score is 1.0.
- it can be negative.

Quadratic regression (a form of linear regression)

a special case of polynomial regression

- These data samples do not seem on a straight line, let's use a different model.

  Quadratic function of x: $y = b + w_1 x + w_2 x^2$

```
# Generating X^2
train_x_s = np.c_[train_x, train_x**2]
test_x_s = np.c_[test_x, test_x**2]

train_x_s
```

```
array([[5.65700000e+01, 3.20016490e+03],
       [1.71940000e+02, 2.95633636e+04],
       [1.61470000e+02, 2.60725609e+04],
       ...,
       [8.48600000e+01, 7.20121960e+03],
       [2.44320000e+02, 5.96922624e+04],
       [4.25500000e+01, 1.81050250e+03]])
```

```
>>> np.c_[np.array([1,2,3]), np.array([4,5,6])]
array([[1, 4],      // concatenation
       [2, 5],
       [3, 6]])
>>> np.c_[np.array([[1,2,3]]), 0, 0, np.array([[4,5,6]])]
array([[1, 2, 3, ..., 4, 5, 6]])
```

Quadratic regression

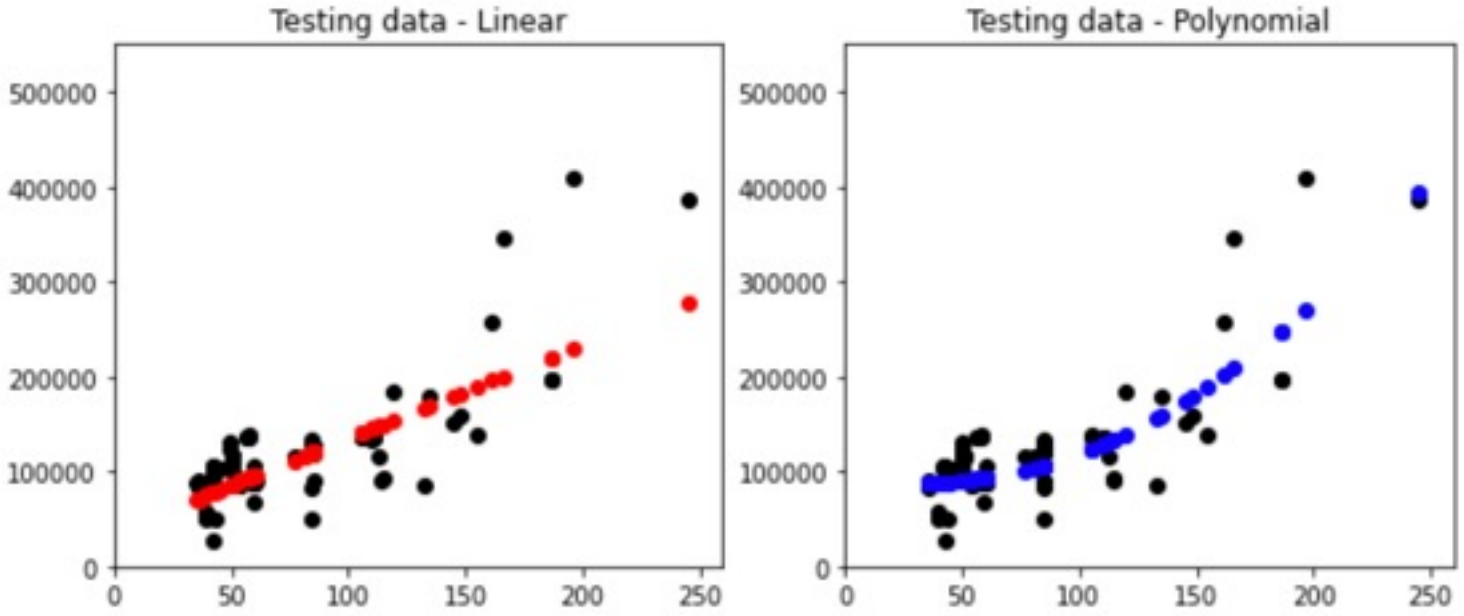
- Regression

```
# Training
lin_reg_poly = LinearRegression(normalize = True)
lin_reg_poly.fit(train_x_s, train_y)
print("Fitting results: ", 'b = ', lin_reg_poly.intercept_, 'w = ', lin_reg_poly.coef_)

# Prediction
test_y_pred_poly = lin_reg_poly.predict(test_x_s)
print("Testing results: ", 'score = ', lin_reg_poly.score(test_x_s, test_y))

# Visualization
plt.figure(figsize=(10, 4))
plt.subplot(1,2,1)
plt.scatter(test_x, test_y, color ='k')
plt.scatter(test_x, test_y_pred, color='r')
plt.title('Testing data - Linear'); plt.xlim(0,260); plt.ylim(0,550000);

plt.subplot(1,2,2)
plt.scatter(test_x[:,0], test_y, color ='k')
plt.scatter(test_x[:,0], test_y_pred_poly, color ='b')
plt.title('Testing data - Polynomial'); plt.xlim(0,260); plt.ylim(0,550000);
```

Quadratic regression

- Result

```
Fitting results:  b =  [94791.84193193] w =  [[-431.72413584    6.73213135]]
Testing results:  score =  0.6961349781442754
```



Multiple Linear Regression

- What if price depends not only on the area.

- Multiple Linear Regression
  - Many independent variables (predictors), and one response
  - $\hat{y}(w,x) = w_0 + \; w_1 x_1 + \; \ldots + \; w_p x_p$

  - e.g., polynomial regression

- cf. multivariate regression
  - from vector input to vector output

https://brilliant.org/wiki/multivariate-regression/

Multiple Linear Regression

- Let's refine the data to have only quantitative values.
- Remove columns showing constant info. in 2017Q1
- Use the mean in the column if nan exists in any item.

```
df_2017q1_mf=df_2017q1.copy()
col=df_2017q1.columns
for c in col:
    if(df_2017q1_mf[c].dtypes == 'object'):
        df_2017q1_mf=df_2017q1_mf.drop(c,axis=1)
        continue
    if(df_2017q1_mf[c].var() == 0):
        df_2017q1_mf=df_2017q1_mf.drop(c,axis=1)
df_2017q1_mf['Yongpae(용적률)']=df_2017q1_mf['Yongpae(용적률)'].replace(np.nan,df_2017q1_mf['Yongpae(용적률)'].mean())
df_2017q1_mf['Gunpae(건폐율)']=df_2017q1_mf['Gunpae(건폐율)'].replace(np.nan,df_2017q1_mf['Gunpae(건폐율)'].mean())
df_2017q1_mf
```

Multiple Linear Regression

- Train and test data

```
X_train, X_test, y_train, y_test = train_test_split( df_2017q1_mf.iloc[:,1:], \
                                                      df_2017q1_mf.iloc[:,0], \
                                                      test_size=0.10, random_state=2)
print("X_train.shape =", X_train.shape)
print("y_train.shape =", y_train.shape)

print("X_test.shape =", X_test.shape)
print("y_test.shape =", y_test.shape)
```

```
X_train.shape = (702, 21)
y_train.shape = (702,)
X_test.shape = (78, 21)
y_test.shape = (78,)
```

Multiple Linear Regression

- Train & Test Result

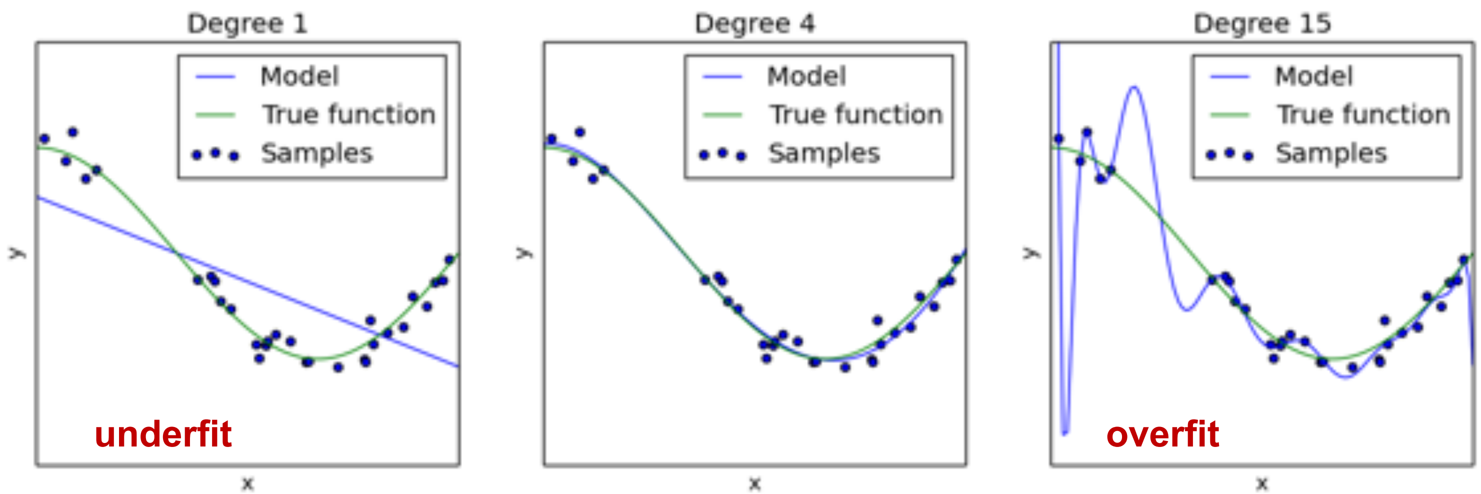
```
# create object
mul_reg = LinearRegression(normalize=True)
# train
mul_reg.fit(X_train, y_train)

# check coefficients and intercept
print("coef_:", mul_reg.coef_)              # w
print("intercept_:", mul_reg.intercept_)    # b
```

```
coef_: [ 1.82057172e+02  1.24331853e+03  7.46420414e+02  7.03658562e+05
 -5.93845750e+05 -4.09059417e+06  5.23292160e+08 -4.49583975e+00
  3.83181929e+00 -7.31375238e+00  1.49634861e+01  1.13172730e+01
  2.62270050e+00 -6.01879150e+00  1.05155808e+03  7.64355121e-01
  5.28651512e+02 -7.80285592e+01  2.50487278e+02 -5.70073837e+02
  3.61334244e+03]
intercept_: 52211546.69883826
```

```
# evaluation
print("Training set score: {:.2f}".format(mul_reg.score(X_train, y_train)))
print("Test set score: {:.2f}".format(mul_reg.score(X_test, y_test)))
```

```
Training set score: 0.80
Test set score: 0.78
```

Overfitting vs Underfitting



- Overfitting - Overfitting happens when a model learns the details and noise in the training dataset to the extent that it negatively impacts the performance of the model on a new dataset.
  - So, a clear sign of overfitting is that its error on the testing or validation dataset is much greater than the error on training dataset.
- Underfitting - It refers to a model that can neither model the training dataset nor generalize to new dataset.
  - An underfit model is not a suitable model with a poor performance even on the training dataset.

Regularization

- Overfitting usually occurs with complex models.
- Regularization normally tries to reduce or penalize the complexity of the model.

- This technique discourages learning a more complex or flexible model, so as to avoid the risk of overfitting.

- L1 Regularization (Lasso) vs. L2 Regularization (Ridge)

Regularization - Ridge

- Ridge
  - ridge regression puts constraint on the coefficients *(w).* The penalty term (lambda) regularizes the coefficients such that if the coefficients take large values the optimization function is penalized.
  - So, ridge regression shrinks the coefficients, and it helps to reduce the model complexity and multi-collinearity.

Regularization - Ridge

- Mean Squared Error (MSE)

$$\sum_{i=1}^{M}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{p} w_j \times x_{ij}\right)^2 \qquad (1.2)$$

- With the Ridge term, the cost function changes to

$$\sum_{i=1}^{M}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{p} w_j \times x_{ij}\right)^2 + \boxed{\lambda \sum_{j=0}^{p} w_j^2} \qquad (1.3)$$

- Import
  - `from sklearn.linear_model import Ridge`

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html

Regularization

- Ridge

```
from sklearn.linear_model import Ridge

ridge = Ridge(alpha=0.01, normalize=True).fit(X_train, y_train)
print("ridge.coef_: {}".format(ridge.coef_))              # w
print("ridge.intercept_: {}".format(ridge.intercept_))    # b
```

```
ridge.coef_: [ 2.27963972e+02  1.22410965e+03  7.33687460e+02  6.51211759e+05
 -3.76429656e+05 -2.50994011e+06  2.97526764e+08 -3.60610453e+00
  2.61484319e+00 -8.45447188e+00  1.19727788e+01  1.16414288e+01
  2.90330002e+00 -4.54672254e+00  9.78316980e+02  1.51797276e+00
  4.80065907e+02 -7.06718315e+01  1.76176512e+02 -5.37562548e+02
  3.33249729e+03]
ridge.intercept_: 25460555.972395957
```

```
print("Training score: {:.2f}".format(ridge.score(X_train, y_train)))
print("Test score: {:.2f}".format(ridge.score(X_test, y_test)))
```

```
Training score: 0.80
Test score: 0.78
```

Regularization - Lasso

- Lasso (least absolute shrinkage and selection operator)
  - The only difference is instead of taking the square of the coefficients, magnitudes are taken into account. This type of regularization (L1) can lead to zero coefficients.
  - That is, some of the features are completely neglected for the evaluation of output. So Lasso regression not only helps in reducing overfitting but it can help us in feature selection.

Regularization - Lasso

- MSE

$$\sum_{i=1}^{M}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{p} w_j \times x_{ij}\right)^2 \tag{1.2}$$

- With Lasso, the cost function changes to

$$\sum_{i=1}^{M}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{p} w_j \times x_{ij}\right)^2 + \boxed{\lambda \sum_{j=0}^{p} |w_j|} \tag{1.4}$$

- Import
  - `from sklearn.linear_model import Lasso`

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

Regularization - Lasso

- Lasso

```
from sklearn.linear_model import Lasso

lasso = Lasso(alpha = 1,normalize=True).fit(X_train, y_train)
print("Training score: {:.2f}".format(lasso.score(X_train, y_train)))
print("Test score: {:.2f}".format(lasso.score(X_test, y_test)))
print("Number of features used:", np.sum(lasso.coef_ != 0))
```

```
Training score: 0.80
Test score: 0.78
Number of features used: 19
```

```
lasso001 = Lasso(0.01, normalize=True).fit(X_train, y_train)
print("Training score: {:.2f}".format(lasso.score(X_train, y_train)))
print("Test score: {:.2f}".format(lasso.score(X_test, y_test)))
print("Number of features used:", np.sum(lasso.coef_ != 0))
```

```
Training score: 0.80
Test score: 0.78
Number of features used: 19
```