

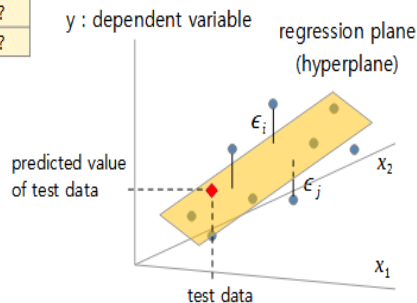


[training data] [test data]

x_1	x_2	y	x_1	x_2	y
1.5	1.2	5.28	1.0	1.8	?
1.9	2.5	2.33	3.7	2.3	?
2.8	3.1	11.6	4.5	3.9	?
3.1	4.0	7.20			
4.3	5.3	9.43			
6.8	6.2	11.38			
7.5	7.1	5.40			
7.3	8.3	8.52			
8.2	9.2	11.80			
12.1	10.1	18.49			
10.0	11.0	18.13			
11.6	12.6	20.34			
13.7	13.2	14.42			
14.8	14.0	26.96			
15.9	15.1	29.84			
16.4	16.0	20.27			

$$y_i = w_1 x_{1,i} + w_2 x_{2,i} + \dots + w_k x_{k,i} + b + \epsilon_i$$

$$\hat{y}_i = w_1 x_{1,i} + w_2 x_{2,i} + \dots + w_k x_{k,i} + b$$



• Assumptions

$$E(\epsilon|x)=0$$

$$\text{Cov}(\epsilon_i, \epsilon_j|x)=0$$

$$\text{Var}(\epsilon|x)=\sigma^2$$

$$\epsilon \sim N(0, \sigma^2)$$

constant

normal distribution

3. Linear Regression

Ordinary Least Square (OLS)

Part 1: Least Squares Method, MLE, R^2

- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

1. Ordinary Least Square (OLS)

- [MXML-3-01] {
 - 1-1. Least Squares Method for univariate and multivariate
 - 1-2. Estimating a regression line using the Maximum Likelihood Estimates (MLE)
 - 1-3. Performance evaluation metrics (MSE, R^2)
- [MXML-3-02] {
 - 1-4. Overfitting and Regularization
 - 1-5. Ridge, Lasso
 - 1-6. Implementation of linear regression using `scipy.optimize`
- [MXML-3-03] {
 - 1-7. Mean-Centering & Feature-Scaling (Normalization, Standardization)
 - 1-8. Prediction of Boston house price using `scipy` and `sklearn`

2. Total Least Squares (TLS)

- [MXML-3-04] {
 - 2-1. OLS vs. TLS
 - 2-2. Objective function for TLS
 - 2-3. Implementation of TLS using `scipy.optimize`
 - 2-4. Prediction of Boston house price using `scipy`

3. Locally Weighted Regression : LWR

- [MXML-3-05] {
 - 3-1. Overview
 - 3-2. Weighted cost function
 - 3-3. Implementing LWR using `scipy` and `scikit-learn`.
 - 3-4. Predicting the Boston house price using LWR

4. Random Sample Consensus (RANSAC)

- [MXML-3-06] {
 - 4-1. Outliers and RANSAC
 - 4-2. The Maximum Number of Attempts to Find a Consensus Set
- [MXML-3-07] {
 - 4-3. Implementation of RANSAC from scratch
 - 4-4. Implementation of RANSAC using `sklearn`'s `RANSACRegressor`
 - 4-5. Prediction of Boston house price using RANSAC
 - 4-6. Comparison of the performance by model

■ Ordinary Least Square (OLS) – Least Squares Method : univariate

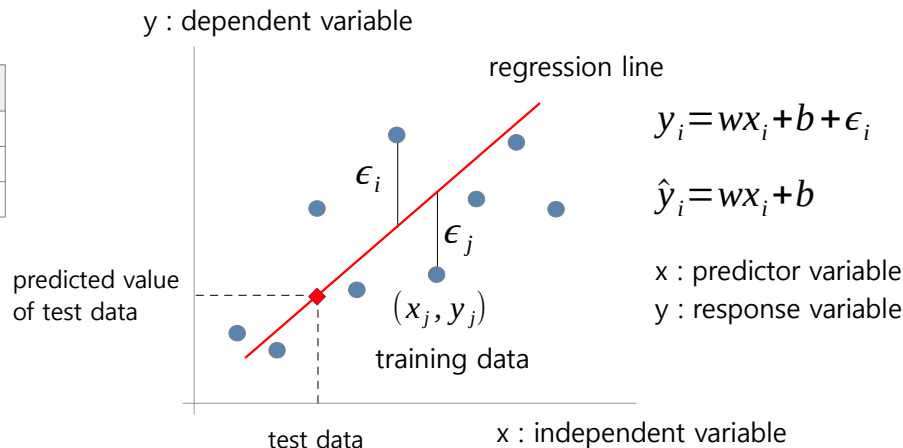
- The goal is to find a straight line that best fits the training data. This is the regression line.
- To find a straight line, you must determine the slope and intercept of that line.
- To determine the slope and intercept, you can use the least squares method.
- To predict the target values for the test data, you can use the regression line.

[training data]

x	y
1.2	5.28
2.5	2.33
3.1	11.6
4.0	7.20
5.3	9.43
6.2	11.38
7.1	5.40
8.3	8.52
9.2	11.80
10.1	18.49
11.0	18.13
12.6	20.34
13.2	14.42
14.0	26.96
15.1	29.84
16.0	20.27

[test data]

x	y
1.8	?
2.3	?
3.9	?



• Assumptions

$$E(\epsilon|x)=0 \quad \text{Cov}(\epsilon_i, \epsilon_j|x)=0$$

$$\text{Var}(\epsilon|x)=\sigma^2 \quad \epsilon \sim N(0, \sigma^2)$$

constant normal distribution

• Objective function

$$\min_{w,b} \sum_i \epsilon_i^2 = \min_{w,b} \sum_i (y_i - \hat{y}_i)^2$$

$$= \min_{w,b} \sum_i (y_i - wx_i - b)^2$$

• Analytical solution

$$\left. \begin{aligned} \frac{\partial}{\partial w} \sum_i (y_i - wx_i - b)^2 &= 0 \\ \frac{\partial}{\partial b} \sum_i (y_i - wx_i - b)^2 &= 0 \end{aligned} \right\} \text{simultaneous equations}$$

• Numerical solution

ex) Quadratic Programming : QP

■ Ordinary Least Square (OLS) – Least Squares Method : multivariate

- When there are multiple features x , find the regression plane that best fits the training data.

[training data]

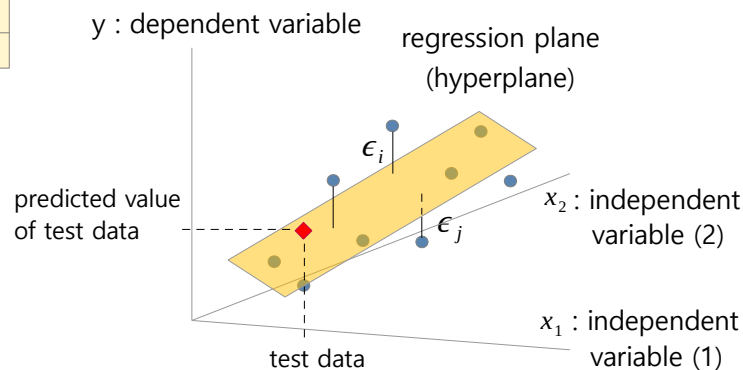
x_1	x_2	y
1.5	1.2	5.28
1.9	2.5	2.33
2.8	3.1	11.6
3.1	4.0	7.20
4.3	5.3	9.43
6.8	6.2	11.38
7.5	7.1	5.40
7.3	8.3	8.52
8.2	9.2	11.80
12.1	10.1	18.49
10.0	11.0	18.13
11.6	12.6	20.34
13.7	13.2	14.42
14.8	14.0	26.96
15.9	15.1	29.84
16.4	16.0	20.27

[test data]

x_1	x_2	y
1.0	1.8	?
3.7	2.3	?
4.5	3.9	?

$$y_i = w_1 x_{1,i} + w_2 x_{2,i} + \dots + w_k x_{k,i} + b + \epsilon_i$$

$$\hat{y}_i = w_1 x_{1,i} + w_2 x_{2,i} + \dots + w_k x_{k,i} + b$$



• Assumptions

$$E(\epsilon|x) = 0$$

$$\text{Var}(\epsilon|x) = \sigma^2$$

constant

$$\text{Cov}(\epsilon_i, \epsilon_j|x) = 0$$

$$\epsilon \sim N(0, \sigma^2)$$

normal distribution

- Objective: least squares

$$\min_{w,b} \sum_i \epsilon_i^2 = \min_{w,b} \sum_i (y_i - \hat{y}_i)^2$$

- Analytical solution

$$\frac{\partial}{\partial w_1} \sum_i (y_i - \hat{y}_i)^2 = 0$$

$$\frac{\partial}{\partial w_2} \sum_i (y_i - \hat{y}_i)^2 = 0$$

⋮

$$\frac{\partial}{\partial b} \sum_i (y_i - \hat{y}_i)^2 = 0$$

simultaneous equations

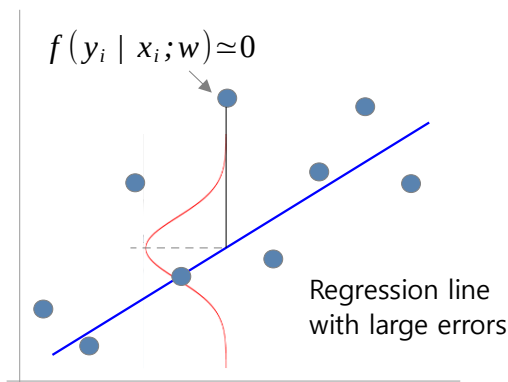
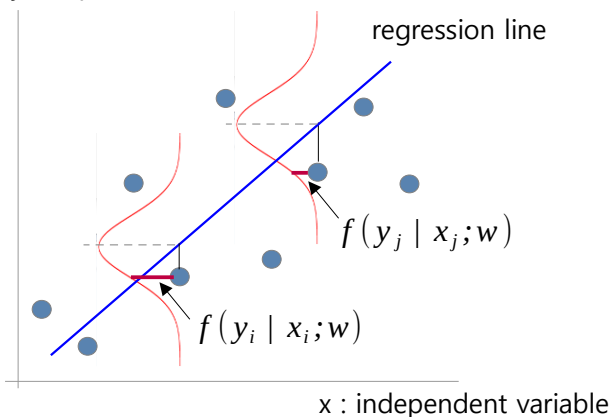
- Numerical solution

ex) Quadratic Programming : QP

■ Estimating a regression line using the Maximum Likelihood Estimates (MLE)

- Suppose we know the data consists of values drawn from $y=wx+b+\epsilon$. ϵ is independent, identically distributed, and normally distributed.
- What are the parameter values w and b for which the observed data have the greatest probability?
- This can be solved with MLE, and the result is the same as the Least Squares Method.

y : dependent variable



$$\hat{y}_i = w_1 x_{1,i} + w_2 x_{2,i} + \dots + w_k x_{k,i} + b$$

$$f(y_i | x_i; w, b) = \frac{1}{\sqrt{2\pi}\sigma} \exp \left(-\frac{(y_i - \hat{y}_i)^2}{2\sigma^2} \right)$$

$$L(w, b) = \prod_{i=1}^N f(y_i | x_i; w, b) \leftarrow \text{likelihood function}$$

log likelihood function

$$\log(L(w, b)) = \sum_{i=1}^N \log(f(y_i | x_i; w, b))$$

$$\log(L(w, b)) = N \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

- Maximize log likelihood

$$\max_{w, b} \log(L(w, b)) = \max_{w, b} \left(-\sum_i (y_i - \hat{y}_i)^2 \right)$$

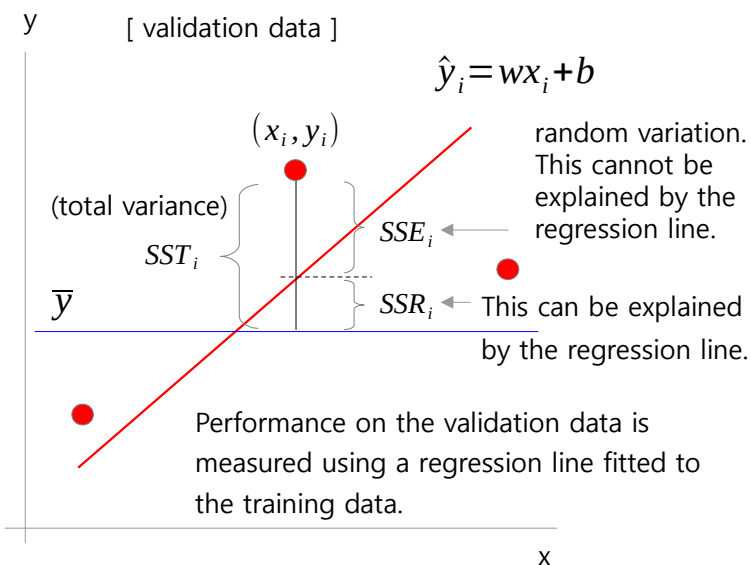
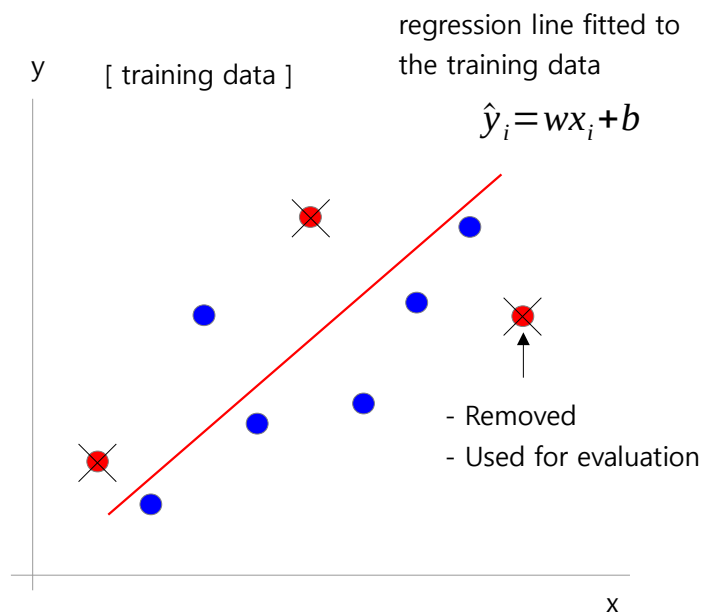
- Minimize squared error
= least squares method

$$\max_{w, b} \left(-\sum_i (y_i - \hat{y}_i)^2 \right) \rightarrow \min_{w, b} \sum_i (y_i - \hat{y}_i)^2$$

- Maximizing the log-likelihood is equivalent to minimizing the squared error.

■ Performance evaluation metrics: MSE and R-squared

- The performance of the linear regression model can be measured by MSE or R^2 .
- Split the data set into training and validation data, and apply the regression line fitted to the training data to the validation data to evaluate the performance of the model.
- R-squared is a measure that determines the proportion of the variance of the dependent variable that can be explained by the independent variables. $R^2 = SSR / SST$



1) mean squared error

$$MSE = \frac{1}{n} \sum_i^n (y_i - \hat{y}_i)^2$$

2) R^2

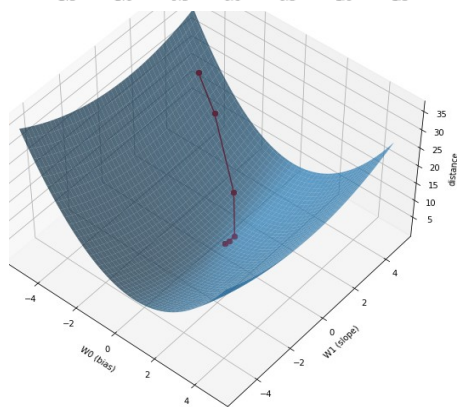
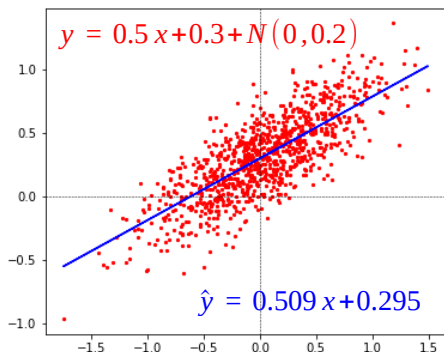
(SS: Sum of Squares)

$$SST = \sum_i^n (y_i - \bar{y})^2 \quad (\text{Total})$$

$$SSE = \sum_i^n (y_i - \hat{y}_i)^2 \quad (\text{Error, Residual})$$

$$SSR = \sum_i^n (\hat{y}_i - \bar{y})^2 \quad (\text{Regression})$$

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



3. Linear Regression

Ordinary Least Square (OLS)

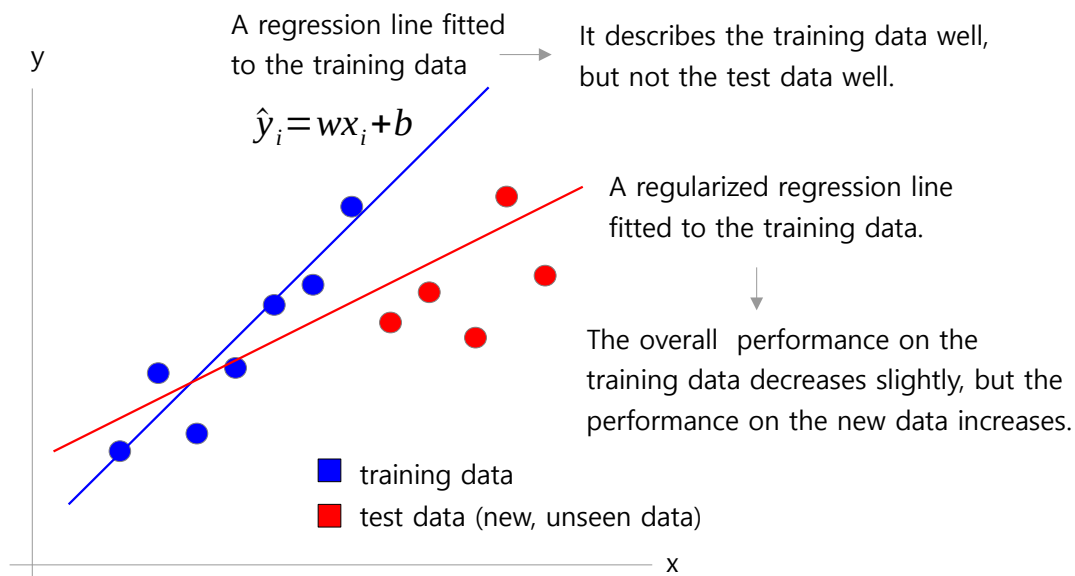
Part 2: Overfitting and Regularization (Lasso and Ridge)

This video was produced in Korean and translated into English,
and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

Overfitting and Regularization

- **Overfitting** occurs when the regression line fits the training data so well that it does not fit the test data, which is new, unseen data.
- On the other hand, underfitting occurs when the regression line is too simple to fit both the training and test data.
- **Regularization** is a key technique that prevents overfitting by adding a penalty term to the model's loss function.
- The penalty term prevents the loss from becoming too small.



$$MSE = \frac{1}{n} \sum_i^n (y_i - \hat{y}_i)^2 +$$

Regularization
Term

↑
This term serves to
reduce losses.

↑
This term serves to prevent
losses from decreasing
excessively.

- In the example on the left, applying regularization has the effect of reducing the slope w and increasing the intercept.
- This produces a regression line that is slightly insensitive to x and not too sensitive.

Regularization: Lasso and Ridge

$$\hat{y}_i = w_1 x_{1,i} + w_2 x_{2,i} + \dots + w_k x_{k,i} + b$$

$$\hat{y}_i = w_0 x_{0,i} + w_1 x_{1,i} + w_2 x_{2,i} + \dots + w_k x_{k,i} \leftarrow (b=w_0, x_{0,i}=1)$$

$$\hat{y}_i = \sum_{j=0}^k w_j x_{j,i} = \vec{w} \cdot \vec{x}$$

L1 regularization (LASSO)

$$\text{loss}(L_1) = \sum_{i=1}^n (y_i - \sum_{j=0}^k w_j x_{j,i})^2 + \lambda \sum_{j=0}^k |w_j|$$

L2 regularization (Ridge)

$$\text{loss}(L_2) = \sum_{i=1}^n (y_i - \sum_{j=0}^k w_j x_{j,i})^2 + \lambda \sum_{j=0}^k w_j^2$$

Regularization term

Regularization constant

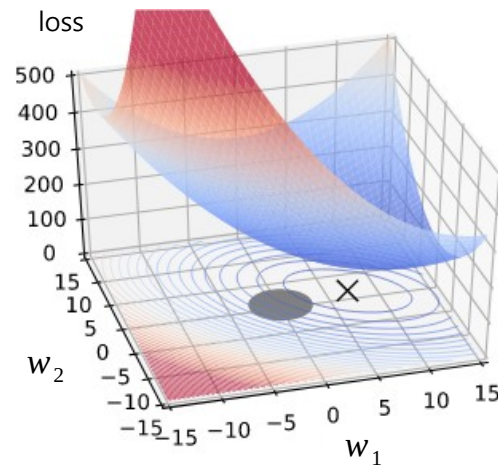
Assuming λ is a constant, λC is also a constant, and we get the following equation. λ is the regularization constant and a hyper-parameter.

Constrained optimization (Lagrangian method)

$$\min_w \sum_{i=1}^n (y_i - \sum_{j=0}^k w_j x_{j,i})^2 \quad \text{subject to} \quad \sum_{i=1}^n w_i^2 \leq C$$

$$\min_w \left[\sum_{i=1}^n (y_i - \sum_{j=0}^k w_j x_{j,i})^2 + \lambda \left(\sum_{j=0}^k w_j^2 - C \right) \right] = \min_w L(w, \lambda)$$

$$\frac{\partial L}{\partial w_j} = 0, \quad \frac{\partial L}{\partial \lambda} = 0 \rightarrow w^*, \lambda^*$$



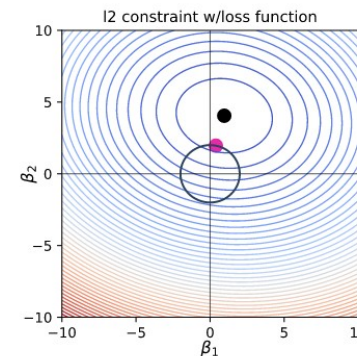
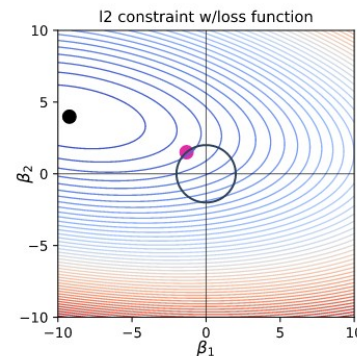
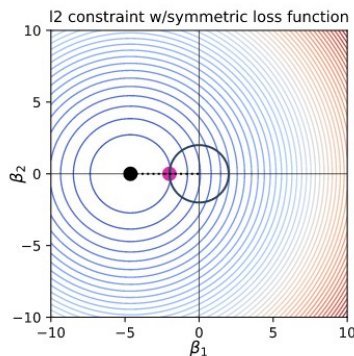
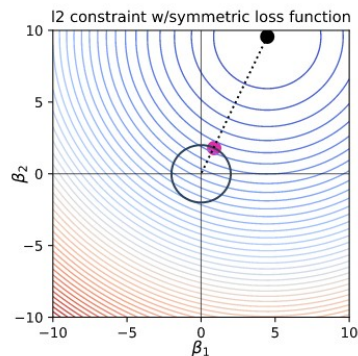
Source : <https://explained.ai/regularization/constraints.html>

Regularization: Lasso and Ridge

L2 regularization (Ridge)

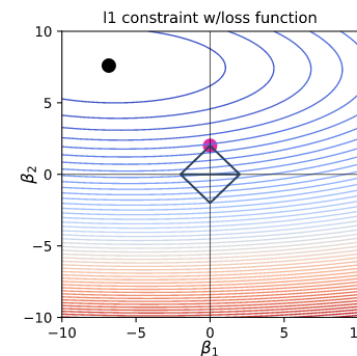
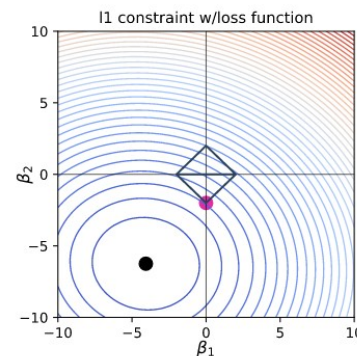
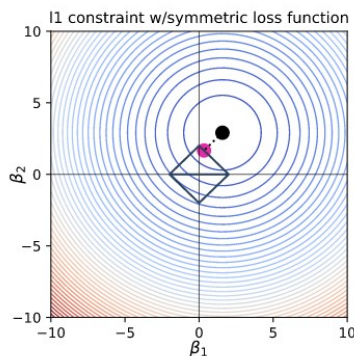
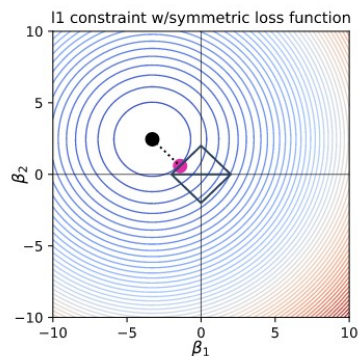
$$\text{loss}(L_2) = \sum_{i=1}^n \left(y_i - \sum_{j=0}^k w_j x_{j,i} \right)^2 + \lambda \sum_{j=0}^k w_j^2$$

Source : <https://explained.ai/regularization/constraints.html>



L1 regularization (LASSO)

$$\text{loss}(L_1) = \sum_{i=1}^n \left(y_i - \sum_{j=0}^k w_j x_{j,i} \right)^2 + \lambda \sum_{j=0}^k |w_j|$$



- Implementing Linear Regression using `scipy.optimize` function. Apply Ridge regularization.

```
# [MXML-3-02] 1.scipy_opt(ols).py
from scipy import optimize
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt
import numpy as np

# y = ax + b + Gaussian noise
def reg_data(a, b, n, s):
    rtn_x, rtn_y = [], []
    for i in range(n):
        x = np.random.normal(0.0, 0.5)
        y = a * x + b + np.random.normal(0.0, s)
        rtn_x.append(x)
        rtn_y.append(y)
    return np.array(rtn_x).reshape(-1,1), np.array(rtn_y)

# Generate 1,000 data points drawn from y = ax + b + noise
# s : standard deviation of the noise distribution
x, y = reg_data(a=0.5, b=0.3, n=1000, s=0.2)

# y = w0 + w1*x1 + ... → w0*x0 + w1*x1 + w2*x2 + ... (x0 = 1)
# y = [w0, w1, w2, ...] * [x0, x1, x2, ...].T (T : transpose)
# y = W * X.T
X = np.hstack([np.ones([x.shape[0], 1]), x])
REG_CONST = 0.01 # regularization constant

# Loss function : Mean Squared Error
# We typically do NOT penalize the intercept term
def ols_loss(W, args):
    e = np.dot(W, X.T) - y
    mse = np.mean(np.square(e)) # mean squared error
    loss = mse + REG_CONST * np.sum(np.square(W[1:]))

# save W and loss
if args[0] == True:
    trace_W.append([W, loss])
return loss
```

```
# Perform optimization process
trace_W = []
result = optimize.minimize(ols_loss, [0., 0.1], args=[True])
print(result)

# Plot the training data and draw the regression line.
y_hat = np.dot(result.x, X.T)
plt.figure(figsize=(6,5))
plt.scatter(x, y, s=5, c='r')
plt.plot(x, y_hat, c='blue')
plt.axvline(x=0, ls='--', lw=0.5, c='black')
plt.axhline(y=0, ls='--', lw=0.5, c='black')
plt.show()

# Draw the loss function and the path to the optimal point.
m = 5
t = 0.1
w0, w1 = np.meshgrid(np.arange(-m, m, t), np.arange(-m, m, t))
zs = np.array([ols_loss([a,b], [False]) \
               for [a, b] in zip(np.ravel(w0), np.ravel(w1))])
z = zs.reshape(w0.shape)

fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111, projection='3d')

# Draw the surface of the loss function
ax.plot_surface(w0, w1, z, alpha=0.7)

# Draw the path to the optimal point.
b = np.array([tw0 for [tw0, tw1], td in trace_W])
w = np.array([tw1 for [tw0, tw1], td in trace_W])
d = np.array([td for [tw0, tw1], td in trace_W])
ax.plot(b, w, d, marker='o', color="r")
```

- Implementing Linear Regression using `scipy.optimize` function. Apply Ridge regularization.

```
ax.set_xlabel('W0 (bias)')
ax.set_ylabel('W1 (slope)')
ax.set_zlabel('distance')
ax.azim = -50
ax.elev = 50
plt.show()

# Check the R2 score
sst = np.sum(np.square(y - np.mean(y)))
sse = np.sum(np.square(y - y_hat))
r2 = 1 - sse / sst
print('\nR2 score = {:.4f}'.format(r2))
print('R2 score = {:.4f}'.format(r2_score(y, y_hat)))
```

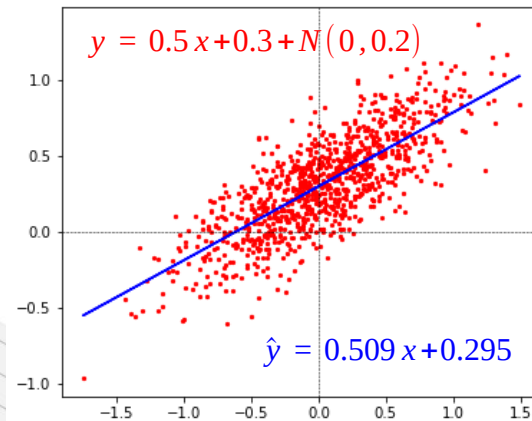
Results :

```
message: Optimization terminated successfully.
success: True
status: 0
fun: 0.04084311103609533
x: [ 2.946e-01  5.090e-01]
nit: 7
jac: [-6.193e-08 -2.910e-07]
hess_inv: [[ 5.021e-01  6.591e-02]
           [ 6.591e-02  2.021e+00]]
nfev: 24
njev: 8
```

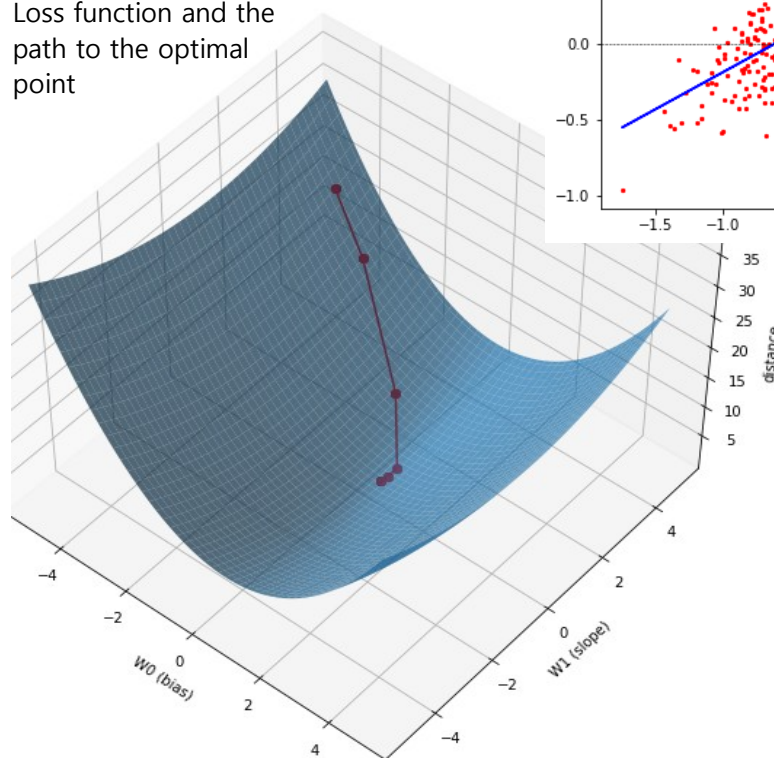
R2 score = 0.6093

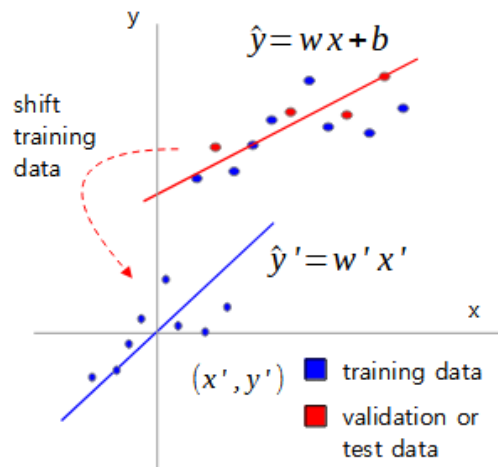
R2 score = 0.6093

training data and
the regression line



Loss function and the
path to the optimal point





3. Linear Regression

Ordinary Least Square (OLS)

Part 3: Mean-Centering & Feature-Scaling (Normalization, Standardization)

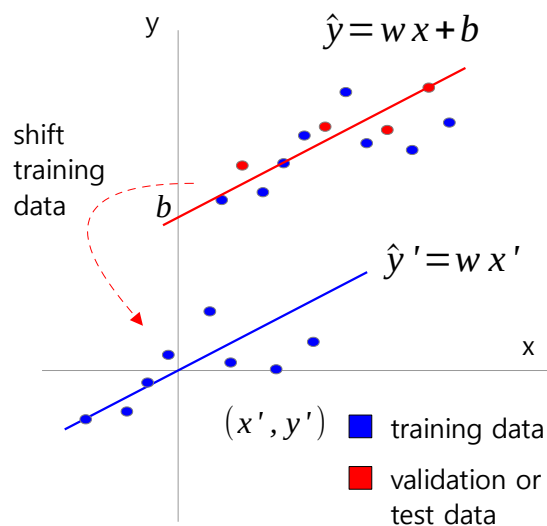
This video was produced in Korean and translated into English,
and the audio was generated by AI (TTS).

$$\begin{aligned} y - \bar{y} &= w(x - \bar{x}) + \epsilon & x' &= \frac{x - \bar{x}}{\sigma_x} & y' &= y - \bar{y} \\ y - \bar{y} &= w \sigma_x \frac{x - \bar{x}}{\sigma_x} + \epsilon & \hat{y}' &= w' x' \\ w' &= w \sigma_x & w &= \frac{w'}{\sigma_x} \\ & & b &= \bar{y} - w \bar{x} \end{aligned}$$

www.youtube.com/@meanxai

■ Mean-Centering and Feature-Scaling (Normalization or Standardization)

- Mean centering is a technique that shifts data closer to the origin. Since the regression line for the shifted data passes through the origin, we only need to estimate w without estimating b . Then we calculate b using w and the average value of the original data.
- Normalization or Standardization is a technique that not only performs mean centering but also feature scaling. In multivariate regression, if the scales of x_i and x_j are significantly different, this can also affect the estimate of w and b , so we need to standardize the scales of all x .
- Normalization is necessary, especially when using regularization. Without normalization, the regularization term may unfairly impose a greater penalty on some coefficients than on others.
- Estimate w' using the normalized data and convert it to the slope of the original data using the formula at the bottom right. Then calculate b using w and the average value of the original data. This is equivalent to mean centering.
- Mean centering or normalization is applied only to the training data. It does not need to be applied to the validation or test data.



$$y = wx + b + \epsilon$$

$$\bar{y} = w\bar{x} + b$$

$$y - \bar{y} = w(x - \bar{x}) + \epsilon$$

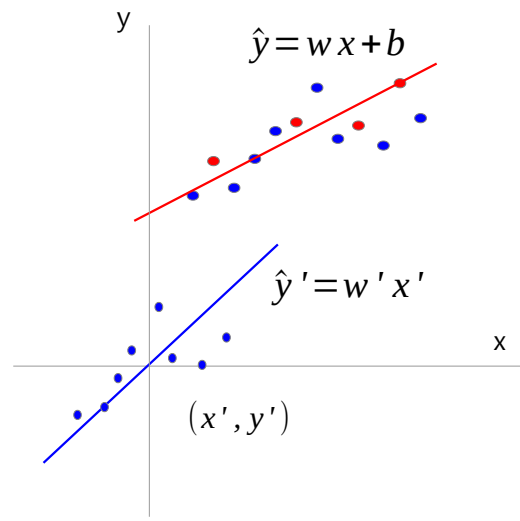
$$x' = x - \bar{x}$$

$$y' = y - \bar{y}$$

$$y' = wx' + \epsilon$$

$$\hat{y}' = wx'$$

$$b = \bar{y} - w\bar{x}$$



$$y - \bar{y} = w(x - \bar{x}) + \epsilon$$

$$y - \bar{y} = w\sigma_x \frac{x - \bar{x}}{\sigma_x} + \epsilon$$

$$w' = w\sigma_x$$

$$x' = \frac{x - \bar{x}}{\sigma_x} \quad y' = y - \bar{y}$$

$$\hat{y}' = w'x'$$

$$w = \frac{w'}{\sigma_x}$$

$$b = \bar{y} - w\bar{x}$$

■ Prediction of Boston house price

- The dataset has 13 features ($x_1 \sim x_{13}$) and 506 samples. The target is the price value, y .

No	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	price
0	0.00632	18	2.31	0	0.538	6.575	65.2	4.09	1	296	15.3	396.9	4.98	24
1	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.9	9.14	21.6
2	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.9	5.33	36.2
5	0.02985	0	2.18	0	0.458	6.43	58.7	6.0622	3	222	18.7	394.12	5.21	28.7
6	0.08829	12.5	7.87	0	0.524	6.012	66.6	5.5605	5	311	15.2	395.6	12.43	22.9
7	0.14455	12.5	7.87	0	0.524	6.172	96.1	5.9505	5	311	15.2	396.9	19.15	27.1
8	0.21124	12.5	7.87	0	0.524	5.631	100	6.0821	5	311	15.2	386.63	29.93	16.5
9	0.17004	12.5	7.87	0	0.524	6.004	85.9	6.5921	5	311	15.2	386.71	17.1	18.9
10	0.22489	12.5	7.87	0	0.524	6.377	94.3	6.3467	5	311	15.2	392.52	20.45	15
11	0.11747	12.5	7.87	0	0.524	6.009	82.9	6.2267	5	311	15.2	396.9	13.27	18.9
12	0.09378	12.5	7.87	0	0.524	5.889	39	5.4509	5	311	15.2	390.5	15.71	21.7
13	0.62976	0	8.14	0	0.538	5.949	61.8	4.7075	4	307	21	396.9	8.26	20.4
14	0.63796	0	8.14	0	0.538	6.096	84.5	4.4619	4	307	21	380.02	10.26	18.2
15	0.62739	0	8.14	0	0.538	5.834	56.5	4.4986	4	307	21	395.62	8.47	19.9
16	1.05393	0	8.14	0	0.538	5.935	29.3	4.4986	4	307	21	386.85	6.58	23.1

- Prediction of Boston house price using `scipy.optimize()`. Applying normalization, Ridge regularization.

```
# [MXML-3-03] 2.boston(ols).py
# prediction of Boston house price
# Applying Mean centering, Normalization, Ridge Regularization
from scipy import optimize
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
import pickle

# Read Boston house price dataset
with open('data/boston_house.pkl', 'rb') as f:
    data = pickle.load(f)

x = data['data']      # features, shape = (506, 13)
y = data['target']    # target, shape = (506,)

# Split the dataset into training and test data
x_train, x_test, y_train, y_test = train_test_split(x, y)

# Mean centering & Normalization are performed on training data.
x_offset = x_train.mean(axis=0)
x_scale = x_train.std(axis=0)
y_offset = y_train.mean()
```

$$x' = \frac{x - \bar{x}}{\sigma_x}$$

$$y' = y - \bar{y}$$

```
xm_train = (x_train - x_offset) / x_scale
ym_train = y_train - y_offset
```

```
# Regularized mean squared error loss function
def ols_loss(W):
    # Calculating MSE using the training data
    d_train = np.dot(W, xm_train.T) - ym_train
    mse = np.mean(np.square(d_train))
    loss = mse + REG_CONST * np.sum(np.square(W))

    # Save the loss history.
    trc_loss.append(loss)
    return loss

# Perform optimization process
trc_loss = []
W0 = np.ones(xm_train.shape[1]) * 0.1 # initial values of W
result = optimize.minimize(ols_loss, W0)

# Check the results
print(result.success)    # check if success = True
print(result.message)

# Visually check the regularized MSE of the training data.
plt.figure(figsize=(6, 4))
plt.plot(trc_loss, label = 'loss_train')
plt.legend()
plt.xlabel('epochs')
plt.show()
```

$$w = \frac{w'}{\sigma_x}$$

$$b = \bar{y} - w \bar{x}$$

```
# Convert result.x to the coef and the intercept
# y_hat = coef * x + intercept
coef = result.x / x_scale
intercept = y_offset - np.dot(x_offset, coef.T)
```


- Prediction of Boston house price using `scipy.optimize()`. Applying normalization, Ridge regularization.

```
# Predict y values of the test data.
y_pred = np.dot(coef, x_test.T) + intercept

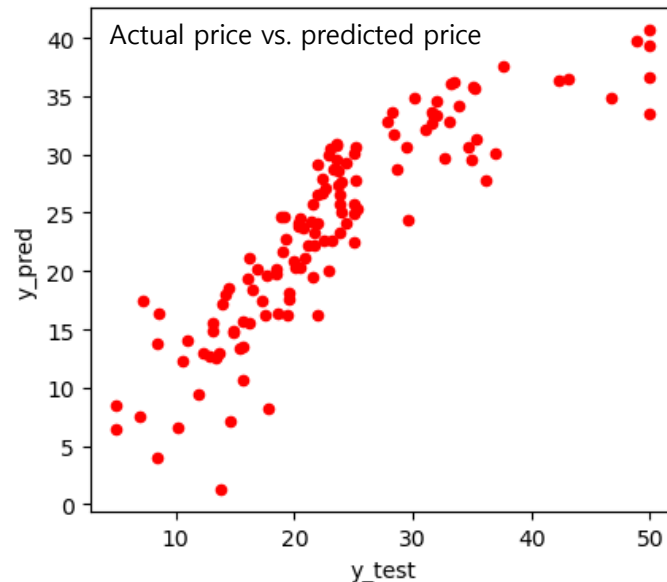
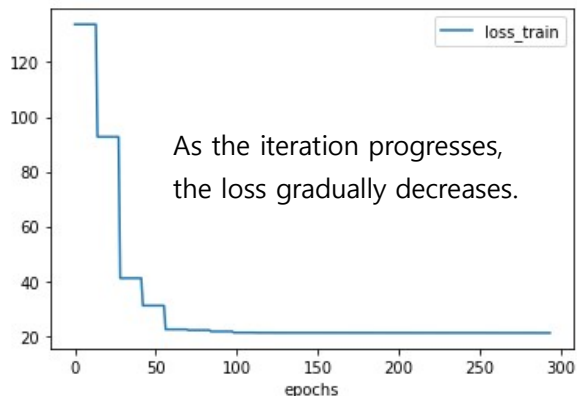
# Visually check the predicted and actual y values of the test data.
plt.figure(figsize=(6, 5))
plt.scatter(y_test, y_pred, s=20, c='r')
plt.xlabel('y_test')
plt.ylabel('y_pred')
plt.show()
```

```
df = pd.DataFrame({'y_test': y_test, 'y_pred': y_pred})
print('\n', df.head(10))
```

```
# Check R2 score of the test data.
print('\nR2 score = {:.4f}'.
      format(r2_score(y_test, y_pred)))
```

R2 score = 0.7602

	y_test	y_pred
0	13.1	15.524781
1	20.5	20.367081
2	20.9	21.092619
3	37.6	37.575247
4	29.6	24.378603
5	16.1	19.348267
6	33.1	32.762967
7	22.6	27.166868
8	25.1	30.630688
9	13.4	12.538836



- Prediction of Boston house price using sklearn's LinearRegression, Ridge, Lasso

```
# [MXML-3-03] 3.boston(sklearn).py
# prediction of Boston house price
# using sklearn's LinearRegression, Ridge, Lasso
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.model_selection import train_test_split
import pickle

# Read Boston house price dataset
with open('data/boston_house.pkl', 'rb') as f:
    data = pickle.load(f)

x = data['data']      # features, shape = (506, 13)
y = data['target']    # target, shape = (506,)

# Split the dataset into training and test data
x_train, x_test, y_train, y_test = train_test_split(x, y)

# 1. LinearRegression()
model = LinearRegression()
model.fit(x_train, y_train)
y_pred = model.predict(x_test)

# Visually check the predicted and actual y values of the
# test data.
plt.figure(figsize=(6, 5))
plt.scatter(y_test, y_pred, s=20, c='r')
plt.xlabel('y_test')
plt.ylabel('y_pred')
plt.show()
```

```
# Check R2 score of the test data.
r2 = model.score(x_test, y_test)
print('\nR2 (LinearRegression) = {:.3f}'.format(r2))
```

```
# 2. Ridge regularization
model = Ridge(alpha=0.01)
model.fit(x_train, y_train)
r2 = model.score(x_test, y_test)
print('R2 (Ridge) = {:.3f}'.format(r2))
```

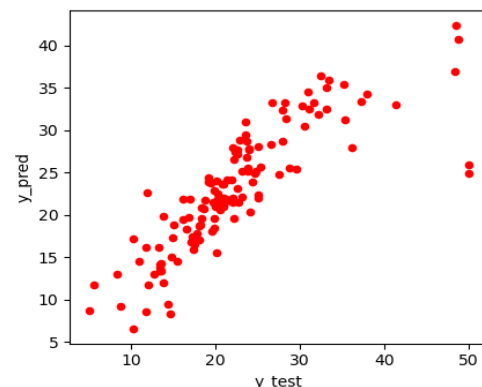
```
# 3. Lasso regularization
model = Lasso(alpha=0.01)
model.fit(x_train, y_train)
r2 = model.score(x_test, y_test)
print('R2 (Lasso) = {:.3f}'.format(r2))
```

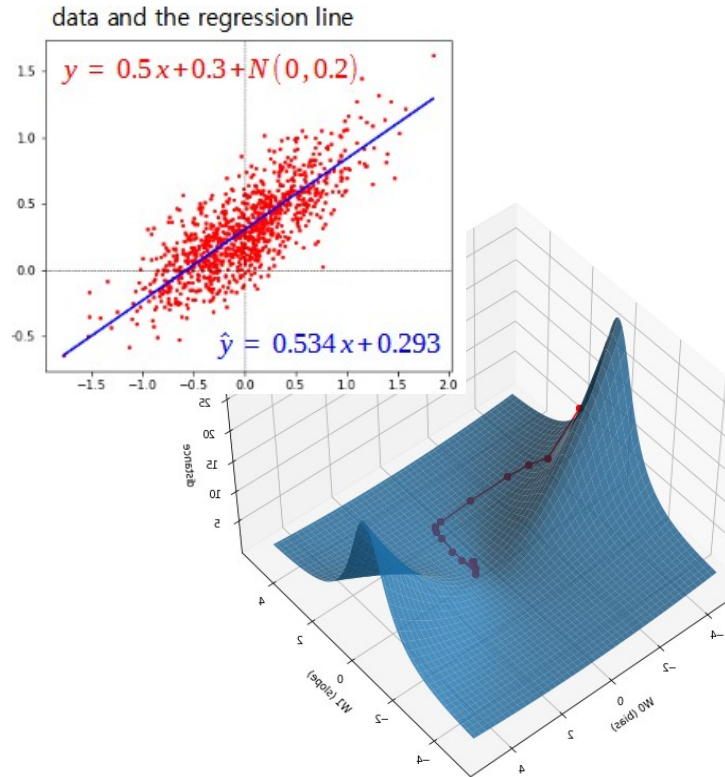
results:

R2 (LinearRegression) = 0.757

R2 (Ridge) = 0.757

R2 (Lasso) = 0.757





3. Linear Regression

Total Least Squares (TLS)

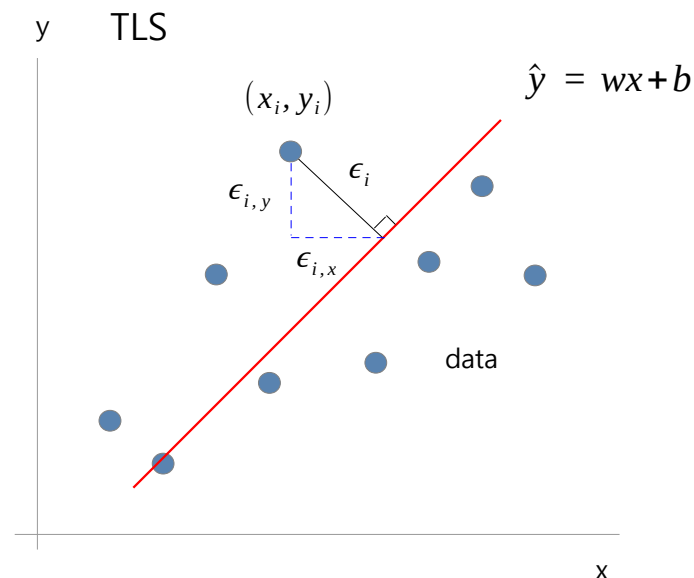
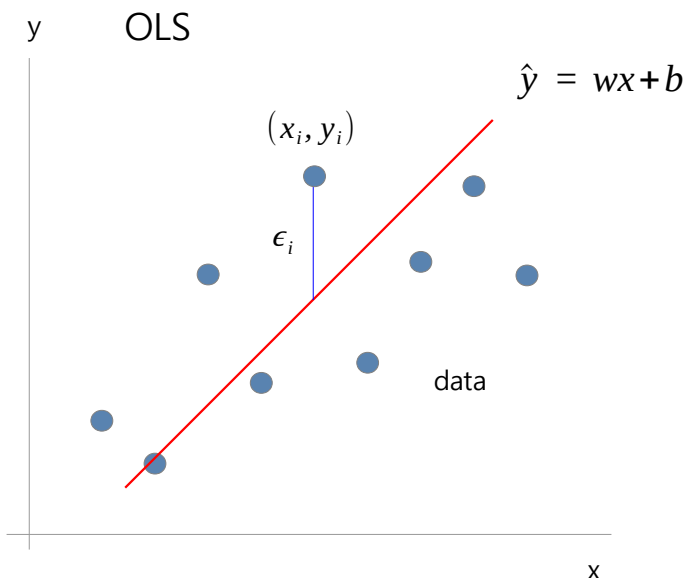
Part 4: Creating an objective function and implementing TLS

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

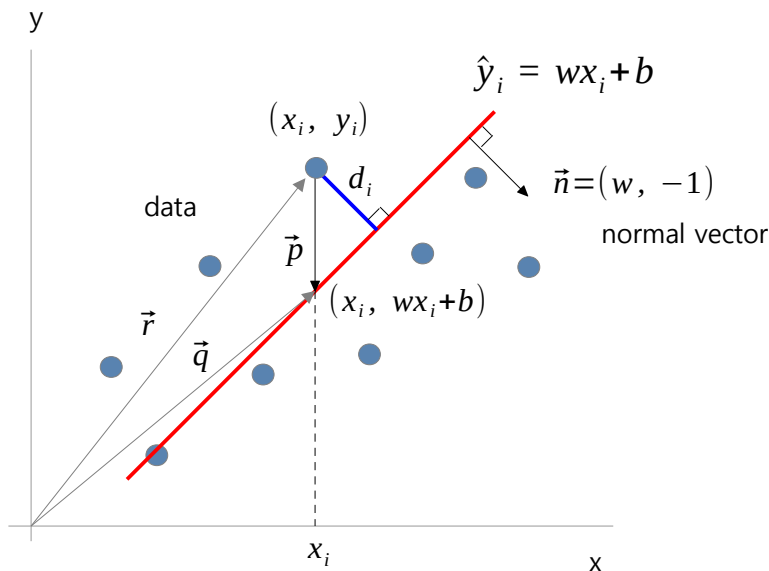
▪ Ordinary Least Squares (OLS) vs. Total Least Squares (TLS)

- OLS assumes that there are no errors in the independent variable x and that there are errors only in the dependent variable y . So we simply measure the errors as the magnitude of $y - \hat{y}$ parallel to the y axis.
- TLS assumes that there are errors in both the independent variable x and the dependent variable y . So, we measure the errors as the perpendicular distances of the data points to the regression line.
- The ordinary TLS assumes that all dependent and independent variables have the same level of uncorrelated Gaussian noise. TLS is a generalized form of least squares regression.



■ Objective function

- TLS is about finding the function that best fits the data points by minimizing the square sum of the perpendicular distances between data points and the regression line.
- The cost function of TLS is non-convex and has a saddle point. TLS is sensitive to initial values and may diverge rather than converge to a solution.
- In this video, we will find a temporary solution using OLS and use it as the initial values to find a TLS solution. We will also apply regularization to prevent the parameter w from becoming too large.



$$d_i = \left| \vec{p} \cdot \frac{\vec{n}}{\sqrt{(w^2 + 1)}} \right|$$

$$d_i = \frac{|\vec{p} \cdot \vec{n}|}{\sqrt{(w^2 + 1)}}$$

$$\vec{p} = \vec{q} - \vec{r}$$

$$\vec{p} = (0, wx_i + b - y_i)$$

$$d_i = \frac{|y_i - wx_i - b|}{\sqrt{w^2 + 1}}$$

• Cost function

$$\min_{w, b} \sum_i d_i^2 = \min_{w, b} \sum_i \frac{(y_i - wx_i - b)^2}{w^2 + 1}$$

■ Implementation of TLS using scipy.optimize. Apply Ridge regularization.

```
# [MXML-3-04] 4.scipy_opt(tls).py
# Implementation of TLS using scipy.optimize. Apply Ridge.
from scipy import optimize
import matplotlib.pyplot as plt
import numpy as np

# y = ax + b + Gaussian noise
def reg_data(a, b, n, s):
    rtn_x, rtn_y = [], []
    for i in range(n):
        x = np.random.normal(0.0, 0.5)
        y = a * x + b + np.random.normal(0.0, s)
        rtn_x.append(x)
        rtn_y.append(y)
    return np.array(rtn_x).reshape(-1,1), np.array(rtn_y)

# Generate 1,000 data points drawn from y = ax + b + noise
x, y = reg_data(a=0.5, b=0.3, n=1000, s=0.2) # s: stdev of noise

# y = w0 + w1*x1 + w2*x2 + ... → w0*x0 + w1*x1 + ... (x0 = 1)
# y = [w0, w1, w2, ...] * [x0, x1, x2, ...].T (T : transpose)
# y = W * X.T
X = np.hstack([np.ones([x.shape[0], 1]), x])
REG_CONST = 0.01 # regularization constant

# Cost function: square sum of the perpendicular distances
# between data points and the regression line.
def tls_loss(W, args):
    numerator = np.square(y - np.dot(W, X.T))
    denominator = np.square(W[1]) + 1
```

$$d_i^2 = \frac{(y_i - wx_i - b)^2}{w^2 + 1}$$

```
d2 = numerator / denominator
msd = np.mean(d2)
loss = msd + REG_CONST * np.sum(np.square(W))
```

```
# save W and loss history
if args[0] == True:
    trace_W.append([W, loss])
return loss
```

```
# Perform optimization process
trace_W = []
result = optimize.minimize(tls_loss, [-4, 0.5], args=[True])
print(result)
```

```
# Plot the training data and draw the regression line.
y_hat = np.dot(result.x, X.T)
plt.figure(figsize=(6,5))
plt.scatter(x, y, s=5, c='r')
plt.plot(x, y_hat, c='blue')
plt.axvline(x=0, ls='--', lw=0.5, c='black')
plt.axhline(y=0, ls='--', lw=0.5, c='black')
plt.show()
```

```
# Draw the loss function and the path to the optimal point.
m = 5
t = 0.1
w0, w1 = np.meshgrid(np.arange(-m, m, t), np.arange(-m, m, t))
```

- Implementation of TLS using scipy.optimize. Apply Ridge regularization.

```
zs = np.array([tls_loss([a,b], [False]) \
               for [a, b] in zip(np.ravel(w0), np.ravel(w1))])
```

```
z = zs.reshape(w0.shape)
fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(w0, w1, z, alpha=0.8) # draw the surface
```

Draw the path to the optimal point.

```
b = np.array([tw0 for [tw0, tw1], td in trace_W[:50]])
w = np.array([tw1 for [tw0, tw1], td in trace_W[:50]])
d = np.array([td for [tw0, tw1], td in trace_W[:50]])
ax.plot(b, w, d, marker='o', color='red')
```

```
ax.set_xlabel('W0 (bias)')
ax.set_ylabel('W1 (slope)')
ax.set_zlabel('distance')
ax.azim = -50
ax.elev = 50
plt.show()
```

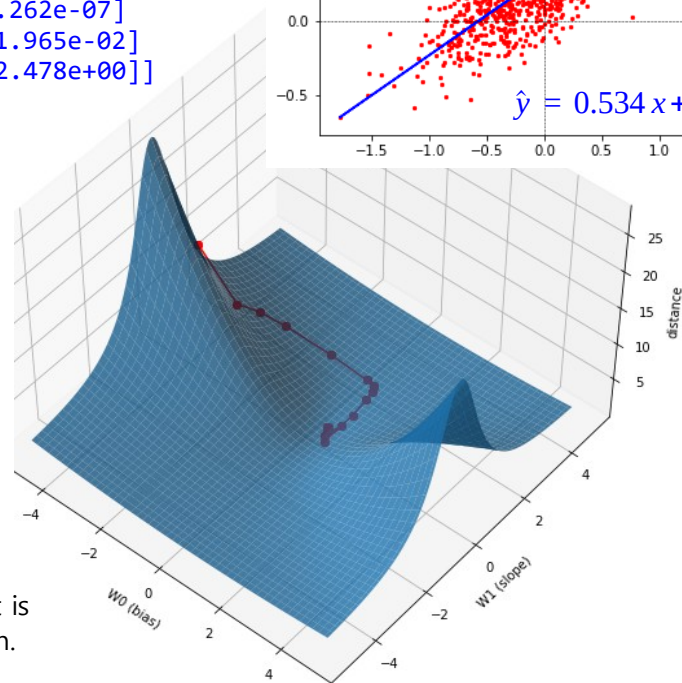
Check the R2 score

```
sst = np.sum(np.square(y - np.mean(y)))
sse = np.sum(np.square(y - y_hat))
r2 = 1 - sse / sst
print('\nR2 score = {:.4f}'.format(r2))
```

```
message: Optimization terminated
        successfully.
success: True
status: 0
fun: 0.03478725077593715
x: [ 2.927e-01  5.336e-01]
nit: 16
jac: [-3.679e-08  9.262e-07]
hess_inv: [[ 6.426e-01 -1.965e-02]
            [-1.965e-02  2.478e+00]]
nfev: 57
njev: 19
```

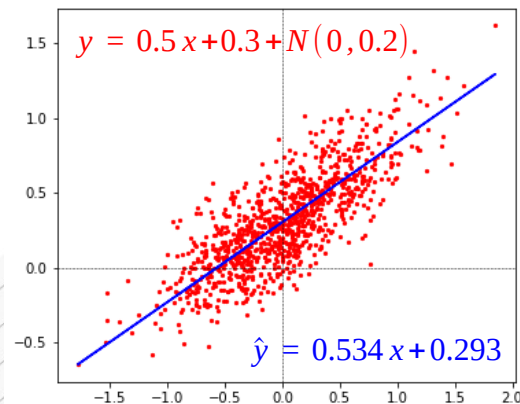
R2 score = 0.6050

loss function and the path
to the optimal point



You can see that it is
non-convex function.

data and the regression line



■ Implementation of TLS using scipy.optimize. Apply Ridge regularization.

- If the initial values are not appropriate or regularization is not applied, the optimal solution may not be found.

```
REG_CONST = 0.0 # regularization constant
```

```
# Cost function
```

```
def tls_loss(W, args):
```

```
    === omitted ===
```

Results may vary depending
on the initial values.

```
# Perform optimization process
```

```
trace_W = []
```

```
result = optimize.minimize(tls_loss, [-4, -1], args=[True]))
```

```
print(result)
```

```
message: Desired error not necessarily achieved due to  
precision loss.
```

```
success: False
```

```
status: 2
```

```
fun: 0.2813413668848516
```

```
x: [-1.112e+03 -6.387e+03] ← (-1112, -6387)
```

```
nit: 39
```

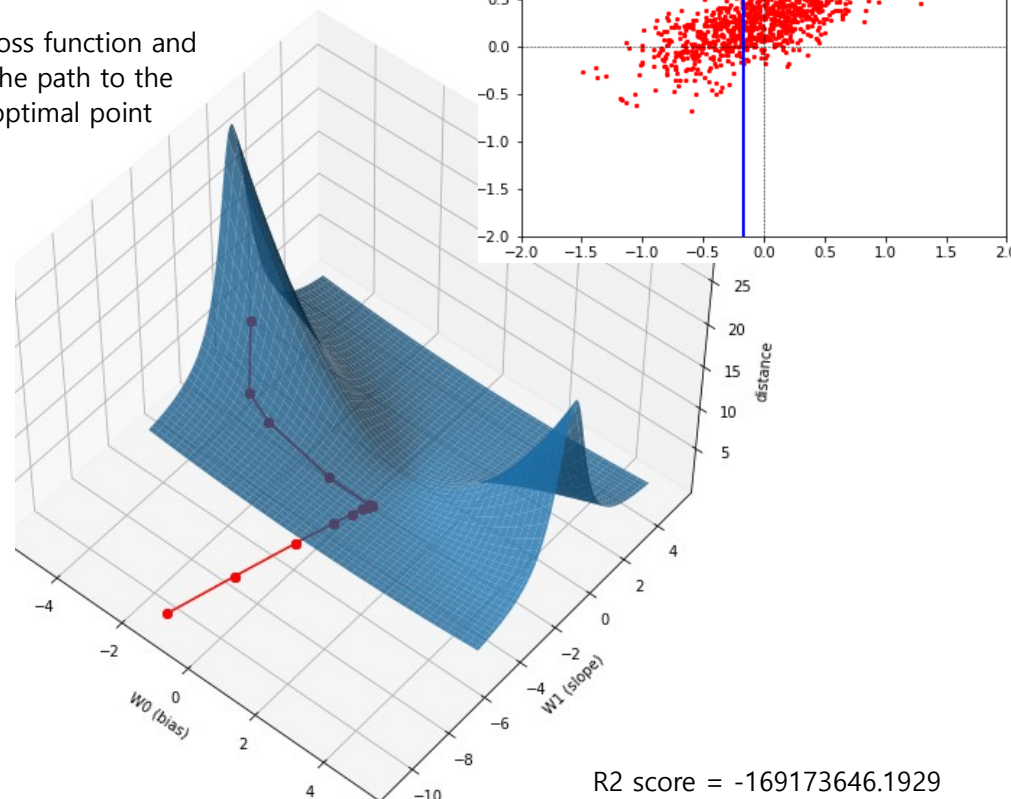
```
jac: [-5.231e-05 9.108e-06]
```

```
hess_inv: [[ 1.324e-01 7.601e-01]  
           [ 7.601e-01 3.693e+01]]
```

```
nfev: 171
```

```
njev: 57
```

loss function and
the path to the
optimal point



R2 score = -169173646.1929

- Prediction of Boston house price using `scipy.optimize()`.
 - Since TLS is sensitive to initial values and may diverge rather than converge to a solution, we will find a temporary solution using OLS and use it as the initial values to find a TLS solution.

```
# [MXML-3-04] 5.boston(tls).py
# prediction of Boston house price by TLS
from scipy import optimize
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
import pickle

# Read Boston house price dataset
with open('data/boston_house.pkl', 'rb') as f:
    data = pickle.load(f)
x = data['data']      # shape = (506, 13)
y = data['target']    # shape = (506,)
x_train, x_test, y_train, y_test = train_test_split(x, y)

# Apply mean-centering to the training data
x_offset = x_train.mean(axis=0)
y_offset = y_train.mean()

xm_train = x_train - x_offset
ym_train = y_train - y_offset

# Apply Ridge regularization
REG_CONST = 0.01 # regularization constant.
```

```
# Cost function for OLS
def ols_loss(W):
    err = np.dot(W, xm_train.T) - ym_train
    mse = np.sqrt(np.mean(np.square(err)))
    loss = mse + REG_CONST * np.sum(np.square(W))
    return loss

# Cost function for TLS
def tls_loss(W):
    numerator = np.square(np.dot(W, xm_train.T) - ym_train)
    denominator = np.sum(np.square(W)) + 1
    d2 = numerator / denominator
    msd = np.sqrt(np.mean(d2))
    loss = msd + REG_CONST * np.sum(np.square(W))

    # save loss history
    trc_loss_train.append(loss)
    return loss

# Perform optimization process
trc_loss_train = []

# Perform OLS
W0 = np.array([0.1] * x_train.shape[1]) # initialize W
result = optimize.minimize(ols_loss, W0)

# Perform TLS
# The optimal W found by OLS is used as the initial value of TLS.
W0 = result.x
result = optimize.minimize(tls_loss, W0)
```

- Prediction of Boston house price using `scipy.optimize()`.

```
print(result.success)    # check if success = True

# Check the loss history
plt.figure(figsize=(6, 4))
plt.plot(trc_loss_train, label = 'loss_train')
plt.legend()
plt.xlabel('epochs')
plt.show()

# y_hat = coef * x + intercept
coef = result.x
intercept = y_offset - np.dot(x_offset, coef.T)

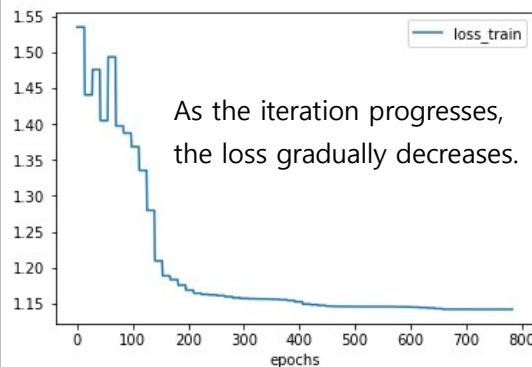
# Predict the y values of the test data
y_pred = np.dot(coef, x_test.T) + intercept

# Visually check the actual and predicted y values of the test data.
plt.figure(figsize=(6, 5))
plt.scatter(y_test, y_pred, s=20, c='r')
plt.xlabel('y_test')
plt.ylabel('y_pred')
plt.show()

df = pd.DataFrame({'y_test': y_test, 'y_pred': y_pred})
print('\n', df.head(10))

# Check the R2 score
print('\nTLS R2 score = {:.4f}'.format(r2_score(y_test, y_pred)))

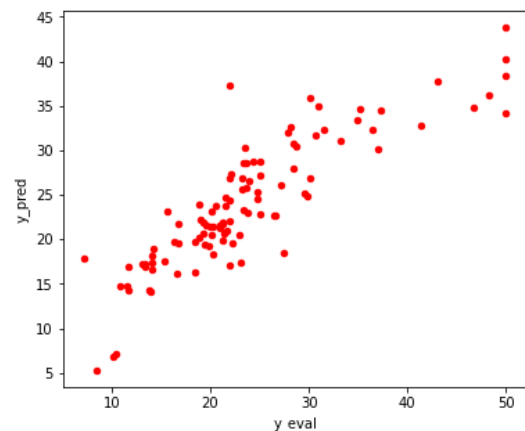
# Check the R2 score from OLS
ols_coef = w0
ols_icept = y_offset - np.dot(x_offset, ols_coef.T)
y_ols_pred = np.dot(ols_coef, x_test.T) + ols_icept
print('OLS R2 score = {:.4f}'.format(r2_score(y_test, y_ols_pred)))
```

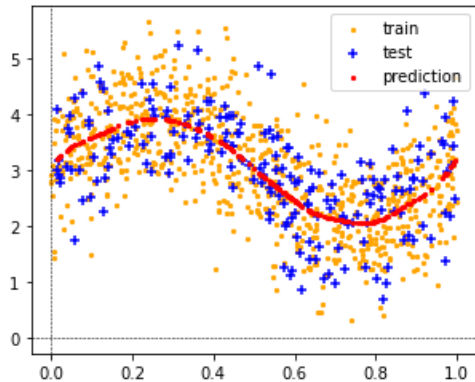
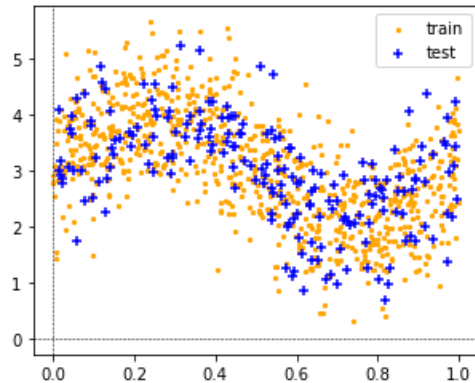


	y_test	y_pred
0	14.1	16.648579
1	19.6	21.635975
2	28.2	32.595285
3	27.9	32.092015
4	26.4	22.719592
5	48.3	36.208407
6	23.6	28.658743
7	28.7	30.469936
8	30.1	26.837508
9	20.0	21.413488

TLS R2 score = 0.7434
OLS R2 score = 0.7204

Actual price vs. predicted price





3. Linear Regression

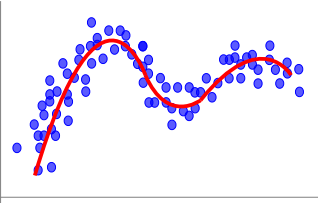
Locally Weighted Regression (LWR)

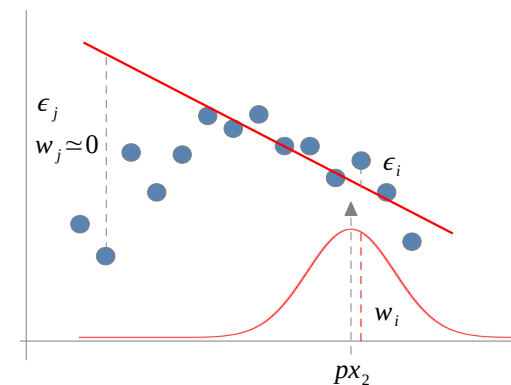
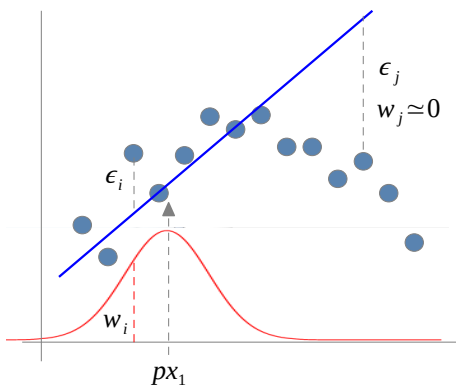
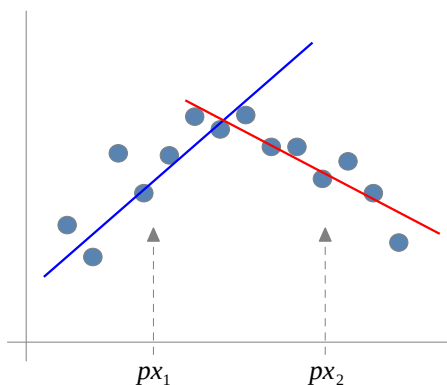
Part 5: Creating an weighted cost function and implementing LWR

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

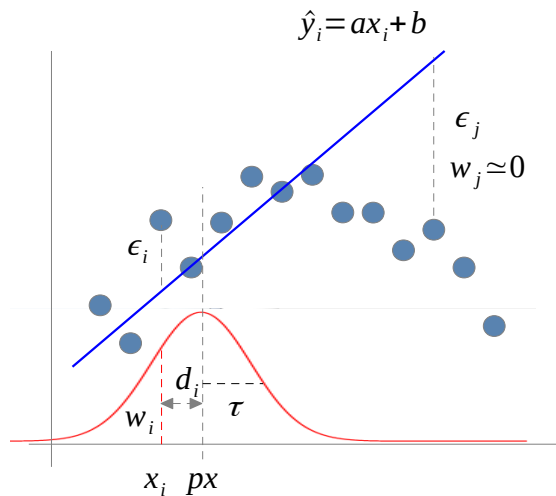
Locally Weighted Regression (LWR)

- To learn the data in the figure on the right, nonlinear regression is needed. We need to choose a nonlinear function and fit that function to the data. Polynomials, sine, log, etc. can be chosen as a nonlinear function. However, it is not easy to choose an appropriate nonlinear function in advance. This method is called a parametric method.
 - LWR is a type of non-parametric method that performs nonlinear regression without assuming a nonlinear function.
- 
- To learn the data in the figure below, nonlinear regression is needed. If you want to perform linear regression using two regression lines, you can fit the blue line to the data in the left region and the red line to the data in the right region.
 - To predict the test data point px_1 , calculate the weights w of each data point using a normal distribution with mean px_1 . Then, linear regression is performed by weighting the errors. The data points on the left have high weight, but the data points on the right have low weight. Therefore, linear regression is performed mainly using the data on the left, and a blue straight line is generated. We can use this line to predict the px_1 . Conversely, test data point px_2 can be predicted using the red straight line.
 - LWR is a non-parametric method capable of nonlinear regression, but it takes a long time because it must be performed for each test data point. LWR is a lazy learner like KNN.



■ Weighted cost function

- Calculate the distance d between the test data point px and all training data points, and calculate weight w for each data point with a normal distribution for d .
- The standard deviation, τ of the normal distribution can be used to adjust the range of neighbors. The τ is a hyper-parameter.
- Applying weights to the OLS cost function produces the weighted cost function used in Locally Weighted Regression.
- The weighted cost function can be written in two forms as follows.



▪ Weighted cost function

- weight function

$$d_i = |px - x_i|$$

$$w_i = \exp\left(-\frac{d_i^2}{2\tau^2}\right) \begin{cases} d_i \rightarrow 0 : w_i \rightarrow 1 \\ d_i \rightarrow \infty : w_i \rightarrow 0 \end{cases}$$

- Cost function

$$\min_{w,b} \sum_i \epsilon_i^2 = \min_{w,b} \sum_i w_i (y_i - \hat{y}_i)^2$$

$$= \min_{w,b} \sum_i w_i (y_i - ax_i - b)^2$$

$$w_i (y_i - ax_i - b)^2 = (\sqrt{w_i} y_i - a \sqrt{w_i} x_i - \sqrt{w_i} b)^2$$

$$x'_i = \sqrt{w_i} x_i$$

$$y'_i = \sqrt{w_i} y_i$$

$$b' = \sqrt{w_i} b$$

$$\min_{w,b} \sum_i \epsilon_i^2 = \min_{w,b} \sum_i (y'_i - ax'_i - b')^2$$

■ Implementing LWR using scipy.optimize

```
# [MXML-3-5] 6.lwr(scipy).py
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
from sklearn.model_selection import train_test_split

# Generate sinusoidal data with Gaussian noise added.
def noisy_sine_data(n, s):
    rtn_x, rtn_y = [], []
    for i in range(n):
        x = np.random.random()
        y = 2.0*np.sin(2.0*np.pi*x)+np.random.normal(0.0, s) + 3.0
        rtn_x.append(x)
        rtn_y.append(y)
    return np.array(rtn_x).reshape(-1,1), np.array(rtn_y)

# Create 1,000 data points for LWR testing.
x, y = noisy_sine_data(n=1000, s=0.7)
x_train, x_test, y_train, y_test = train_test_split(x, y)
x1_train = np.hstack([np.ones([x_train.shape[0], 1]), x_train])

# Visualize the training and test data
plt.figure(figsize=(6,5))
plt.scatter(x_train, y_train, s=5, c='orange', label='train')
plt.scatter(x_test, y_test, marker='+', s=30, c='blue',
            label='test')
plt.legend()
plt.axvline(x=0, ls='--', lw=0.5, c='black')
plt.axhline(y=0, ls='--', lw=0.5, c='black')
plt.show()
```

```
# Find the weight for each data point.
# train: training data, test: test data point to be predicted
def get_weight(train, test, tau):
    d2 = np.sum(np.square(train - test), axis=1)
    w = np.exp(-d2 / (2. * tau * tau))
    return w

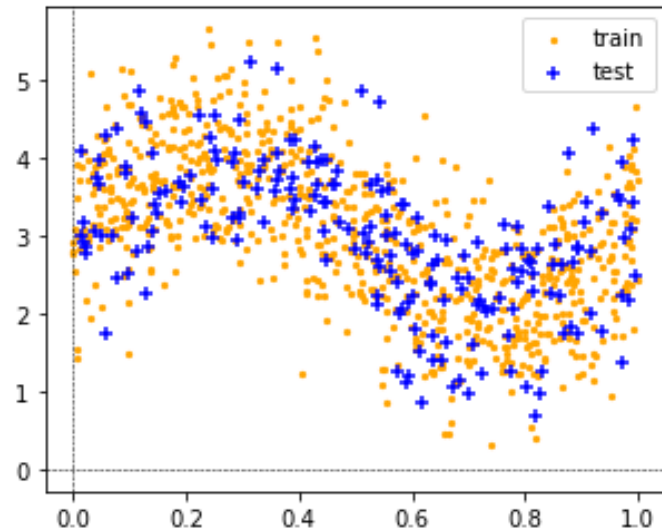
# Weighted cost function
def lwr_loss(W, weight):
    d = np.dot(W, x1_train.T) - y_train
    wmsd = np.mean(weight * np.square(d))
    return wmsd

# predict the target value of the test data
y_pred = []
for tx in x_test:
    weight = get_weight(x_train, tx, 0.05)
    result = optimize.minimize(lwr_loss, [0.1, 0.1], args=weight)
    y_pred.append(np.dot(result.x[1], tx) + result.x[0])
y_pred = np.array(y_pred).reshape(-1,)

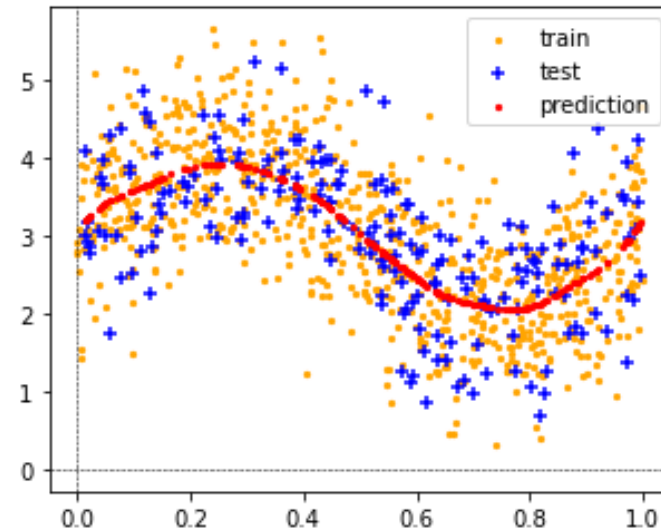
# Visualize the predicted results
plt.figure(figsize=(6,5))
plt.scatter(x_train, y_train, s=5, c='orange', label='train')
plt.scatter(x_test, y_test, marker='+', s=30, c='blue',
            label='test')
plt.scatter(x_test, y_pred, s=5, c='red', label='prediction')
plt.legend()
plt.axvline(x=0, ls='--', lw=0.5, c='black')
plt.axhline(y=0, ls='--', lw=0.5, c='black')
plt.show()
```

- Implementing LWR using scipy.optimize

Training and test data



Prediction result of the test data



■ Implementing LWR using scikit-learn's Ridge library

```
# [MXML-3-5] 7.lwr(sklearn).py
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split

# Generate sinusoidal data with Gaussian noise added.
def noisy_sine_data(n, s):
    rtn_x, rtn_y = [], []
    for i in range(n):
        x= np.random.random()
        y= 2.0*np.sin(2.0*np.pi*x)+np.random.normal(0.0, s) + 3.0
        rtn_x.append(x)
        rtn_y.append(y)
    return np.array(rtn_x).reshape(-1,1), np.array(rtn_y)

# Create 1,000 data points for LWR testing.
x, y = noisy_sine_data(n=1000, s=0.7)
x_train, x_test, y_train, y_test = train_test_split(x, y)
x1_train = np.hstack([np.ones([x_train.shape[0], 1]), x_train])

# Visualize the training and test data
plt.figure(figsize=(6, 5))
plt.scatter(x_train, y_train, s=5, c='orange', label='train')
plt.scatter(x_test, y_test, marker='+', s=30, c='blue',
            label='test')

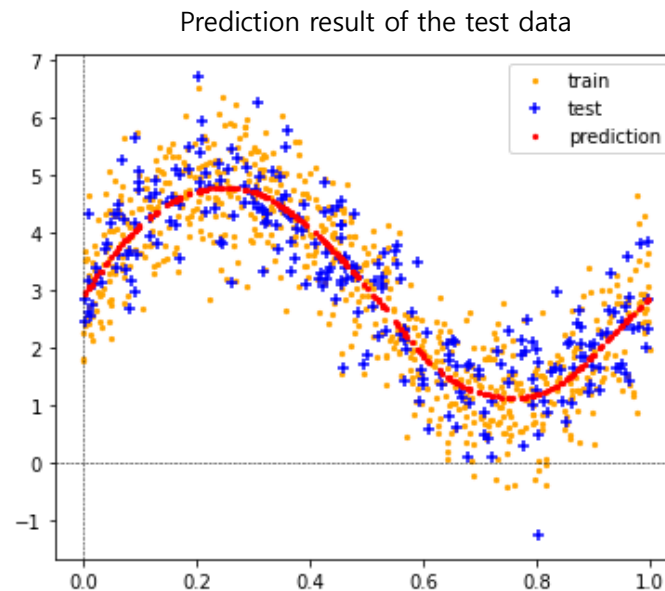
plt.legend()
plt.axvline(x=0, ls='--', lw=0.5, c='black')
plt.axhline(y=0, ls='--', lw=0.5, c='black')
plt.show()
```

```
# Find the weight for each data point.
# train: training data, test: test data point to be predicted
def get_weight(train, test, tau):
    d2 = np.sum(np.square(train - test), axis=1)
    w = np.exp(-d2 / (2. * tau * tau))
    return w

# predict the target value of the test data
y_pred = []
for tx in x_test:
    weight = get_weight(x_train, tx, 0.05)
    model = Ridge(alpha=0.01)
    model.fit(x_train, y_train, sample_weight = weight)
    y_pred.append(model.predict(tx.reshape(-1,1))[0])
y_pred = np.array(y_pred).reshape(-1,)

# Visualize the predicted results
plt.figure(figsize=(6, 5))
plt.scatter(x_train, y_train, s=5, c='orange', label='train')
plt.scatter(x_test, y_test, marker='+', s=30, c='blue',
            label='test')
plt.scatter(x_test, y_pred, s=5, c='red', label='prediction')
plt.legend()
plt.axvline(x=0, ls='--', lw=0.5, c='black')
plt.axhline(y=0, ls='--', lw=0.5, c='black')
plt.show()
```


- Implementing LWR using scikit-learn's Ridge library



■ Predicting the Boston house price using LWR

```
# [MXML-3-5] 8.bostn(lwr).py
# Predicting the Boston house price using LWR
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
import pickle

# Read saved dataset
with open('data/boston_house.pkl', 'rb') as f:
    data = pickle.load(f)
x = data['data']      # shape = (506, 13)
y = data['target']    # shape = (506,)
x_train, x_test, y_train, y_test = train_test_split(x, y)

# Find the weight for each data point.
# train: training data, test: test data point to be predicted
def get_weight(train, test, tau):
    d2 = np.sum(np.square(train - test), axis=1)
    w = np.exp(-d2 / (2. * tau * tau))
    return w

y_pred = []
for tx in x_test:
    weight = get_weight(x_train, tx, 50.0)
    model = Ridge(alpha=0.01)
    model.fit(x_train, y_train, sample_weight = weight)
    y_pred.append(model.predict(tx.reshape(1, -1))[0])
```

```
y_pred = np.array(y_pred).reshape(-1,)

# Visually check the actual and predicted y values of the test data.
plt.figure(figsize=(6, 5))
plt.scatter(y_test, y_pred, s=10, c='r')
plt.xlabel('y_test')
plt.ylabel('y_pred')
plt.show()

print('\nR2 (LWR) = {:.3f}'.format(r2_score(y_test, y_pred)))
```

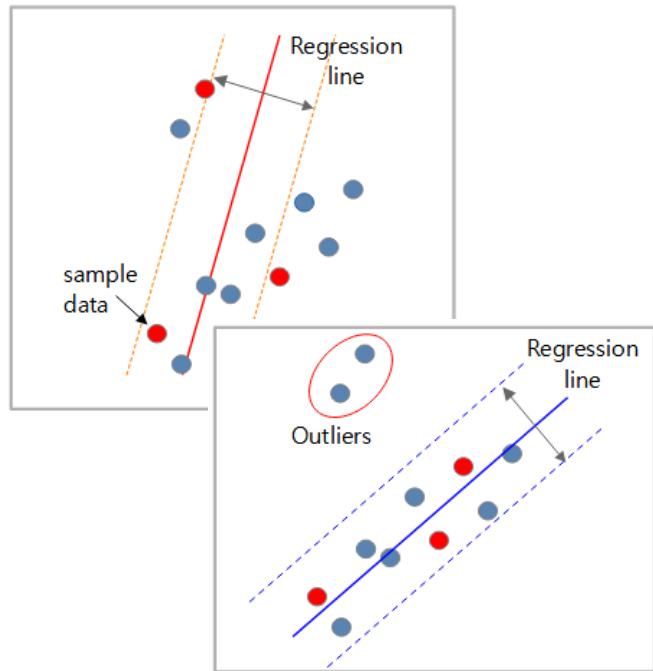
R2 (LWR) = 0.870





3. Linear Regression

Random Sample Consensus (RANSAC)



Part 6: Outliers and RANSAC. The Maximum Number of Attempts to Find a Consensus Set.

This video was produced in Korean and translated into English,
and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

■ Random Sample Consensus (RANSAC) – Fischler, 1981

Graphics and
Image Processing

J. D. Foley
Editor

Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography

Martin A. Fischler and Robert C. Bolles SRI International

A new paradigm, Random Sample Consensus (RANSAC), for fitting a model to experimental data is introduced. RANSAC is capable of interpreting smoothing data containing a significant percentage of gross errors, and is thus ideally suited for applications in automated image analysis where interpretation is based on the data provided by error-prone feature detectors. A major portion of this paper describes the application of RANSAC to the Location Determination Problem (LDP): Given an image depicting a set of landmarks with known locations, determine that point in space from which the image was obtained. In response to a RANSAC requirement, new results are derived on the minimum number of landmarks needed to obtain a solution, and algorithms are presented for computing these minimum-landmark solutions in closed form. These results provide the basis for an automatic system that can solve the LDP under difficult viewing.

I. Introduction

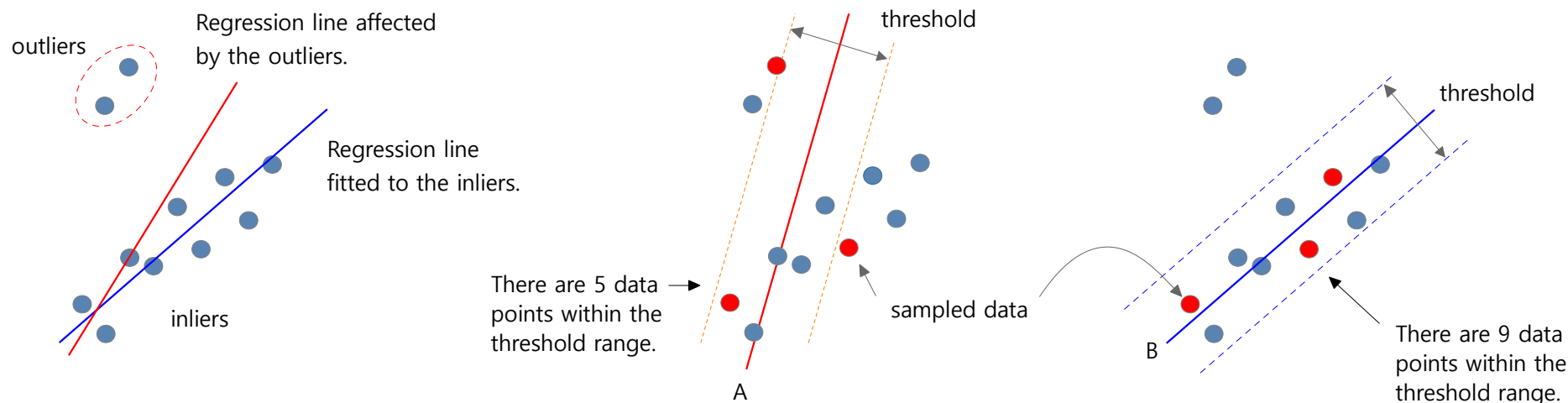
We introduce a new paradigm, Random Sample Consensus (RANSAC), for fitting a model to experimental data; and illustrate its use in scene analysis and automated cartography. The application discussed, the location determination problem (LDP), is treated at a level beyond that of a mere example of the use of the RANSAC paradigm; new basic findings concerning the conditions under which the LDP can be solved are presented and a comprehensive approach to the solution of this problem that we anticipate will have near-term practical applications is described.

To a large extent, scene analysis (and, in fact, science in general) is concerned with the interpretation of sensed data in terms of a set of predefined models. Conceptually, interpretation involves two distinct activities: First, there is the problem of finding the best match between the data and one of the available models (the classification problem); Second, there is the problem of computing the best values for the free parameters of the selected model (the parameter estimation problem). In practice, these two problems are not independent--a solution to the parameter estimation problem is often required to solve the classification problem.

Classical techniques for parameter estimation, such as least squares, optimize (according to a specified objective function) the fit of a functional description (model) to all of the presented data. These techniques have ...

■ Outliers and RANSAC

- Outliers in the data affect the estimation of the regression line. The estimated regression line may be biased toward the outliers, like the red straight line in the figure below.
- RANSAC is an algorithm that finds a regression line using only inliers without being affected by outliers as much as possible.
- RANSAC iterates the process of fitting a regression line to a randomly sampled subset. This results in multiple regression lines. And the one with the most data points near the regression line is chosen as the final regression line. Select B among the regression lines A and B in the figure below.
- If all sampled data points were inliers by chance, most of them would be near the regression line.
- We need to determine the number of iterations so that only inliers can be sampled at least once. This is the main topic of this video, "The Maximum Number of Attempts to Find a Consensus Set" presented in the section II-B of the paper.



■ The Maximum Number of Attempts to Find a Consensus Set – The expected number of trials, $E(k)$

- k : the number of trials or attempts.
- n : The number of data points to sample.
- w : The probability that the selected data point is an inlier.
- $1 - w$: The probability that the selected data point is an outlier.

w^n - The probability that all n selected data points are inliers.

$1 - w^n$ - The probability that n selected data points contain at least one outlier.

Probability distribution table of trials, k

trials (k)	Probability that all selected data points at the k -th iteration are inliers.
1	w^n
2	$(1 - w^n) w^n$
3	$(1 - w^n)^2 w^n$
\vdots	\vdots

$$E(k) = w^n + 2(1 - w^n)w^n + 3(1 - w^n)^2 w^n + \dots$$

$$E(k) = b + 2ab + 3a^2b + 4a^3b + \dots \quad \leftarrow \quad b = w^n, \quad a = 1 - w^n$$

$$E(k) = b(1 + 2a + 3a^2 + 4a^3 + \dots)$$

$$S = a + a^2 + a^3 + \dots = \frac{a}{1 - a} \quad (|a| < 1)$$

$$\frac{dS}{da} = 1 + 2a + 3a^2 + 4a^3 + \dots = \frac{1}{(1 - a)^2}$$

$$E(k) = \frac{b}{(1 - a)^2} = \frac{1}{w^n}$$

Example : if $w = 0.5$, $n = 4$, then $E(k) = 16$

Source : II-B. Maximum Number of Attempts to Find a Consensus Set in the section II-B of the paper.

- The Maximum Number of Attempts to Find a Consensus Set – The standard deviation of trials, SD(k)

$$\text{Var}(k) = E(k^2) - E(k)^2$$

$$E(k) = b(1 + 2a + 3a^2 + 4a^3 + \dots)$$

$$E(k^2) = b(1 + 4a + 9a^2 + 16a^3 + 25a^4 + \dots)$$

$$E(k^2) = b(1 + 2a + 3a^2 + 4a^3 + 5a^4 + \dots) + \rightarrow E(k)$$

$$b(0 + 2a + 6a^2 + 12a^3 + 20a^4 + \dots)$$

$$S = a^2 + 2a^3 + 3a^4 + 4a^5 + \dots$$

$$Sa = a^3 + 2a^4 + 3a^5 + 4a^6 + \dots$$

$$S(1-a) = a^2 + a^3 + a^4 + a^5 + \dots = \frac{a^2}{1-a}$$

$$S = a^2 + 2a^3 + 3a^4 + 4a^5 + \dots = \frac{a^2}{(1-a)^2}$$

$$\frac{dS}{da} = 2a + 6a^2 + 12a^3 + 20a^4 + \dots = \frac{2a}{(1-a)^3}$$

trials (k)	Probability
1	w^n
2	$(1-w^n)w^n$
3	$(1-w^n)^2w^n$
\vdots	\vdots

$$E(k^2) = \frac{b}{(1-a)^2} + \frac{2ab}{(1-a)^3} = \frac{2-b}{b^2}$$

$$\text{Var}(k) = \frac{2-b}{b^2} - \frac{1}{b^2} = \frac{1-b}{b^2}$$

$$\text{SD}(k) = \frac{\sqrt{1-b}}{b} = \frac{\sqrt{1-w^n}}{w^n}$$

$$E(k) = \frac{1}{w^n} \quad \text{SD}(k) = \frac{\sqrt{1-w^n}}{w^n}$$

Example (1) : if $w = 0.5$, $n = 4$, then $E(k) = 16$, $\text{SD}(k) = 15.5$

Example (2) : If we set the appropriate number of iterations to $E(k) + 2 \cdot \text{SD}(k)$,
we get: $k = 16 + 2 \cdot 15.5 = 47$. This is the maximum number of attempts.

Source : II-B. Maximum Number of Attempts to Find a Consensus Set
in the section II-B of the paper.

■ The Maximum Number of Attempts to Find a Consensus Set – Method using the probability z .

- k : the number of trials or attempts.
- n : The number of data points to sample.
- w : The probability that the selected data point is an inlier.
- $1 - w$: The probability that the selected data point is an outlier.
- z : The probability that all n selected data points are inliers at least once in k iterations.

w^n - The probability that all n selected data points are inliers.

$1 - w^n$ - The probability that n selected data points contain at least one outlier.

$(1 - w^n)^k$ - This is the probability that, given k iterations, every iteration will contain at least one outlier.

$1 - (1 - w^n)^k$ - This is the z .

$$z = 1 - (1 - w^n)^k$$

Example (1) : if $w = 0.5$, $n = 4$, $z = 95\%$, then $k = 46.4 \rightarrow 47$

This is the maximum number of attempts.

$$1 - z = (1 - w^n)^k \longrightarrow k = \frac{\log(1 - z)}{\log(1 - w^n)}$$

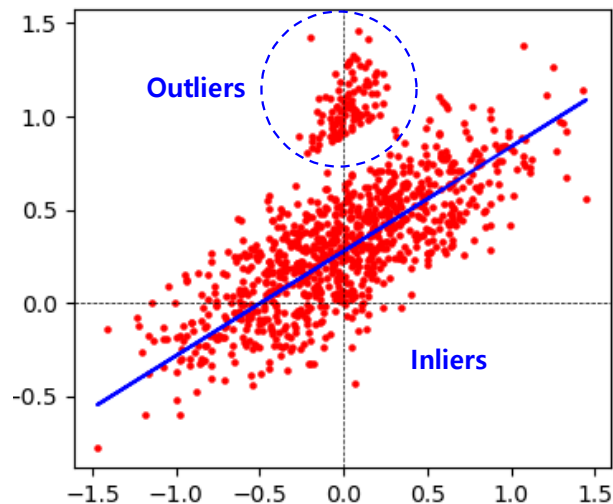
Source : II-B. Maximum Number of Attempts to Find a Consensus Set
in the section II-B of the paper.



3. Linear Regression

Random Sample Consensus (RANSAC)

Part 7: Implementing RANSAC from scratch



This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

■ Implementing RANSAC from scratch

```
# [MXML-3-07] 9.ransac(1).py
# Implementing RANSAC from scratch
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

# Generate n data samples with outliers.
def reg_data_outlier(a, b, n, s, outlier_rate=0.1):
    n1 = int(n * outlier_rate) # the number of outliers
    n2 = n - n1                # the number of inliers

    # Generate normal data points (inliers)
    x2 = np.random.normal(0.0, 0.5, size=n2)
    y2 = a * x2 + b + np.random.normal(0.0, s, size=n2)

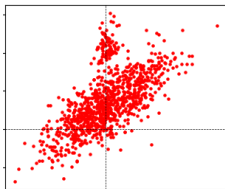
    # Generate abnormal data points (outliers)
    x1 = np.random.normal(0.5, 0.1, size=n1)
    y1 = a * x1 + b * 3 + np.abs(np.random.normal(0.0, s, size=n1))

    x = np.hstack([x2, x1]).reshape(-1,1)
    y = np.hstack([y2, y1])

    return x, y

x, y = reg_data_outlier(a=0.5, b=0.3, n=1000, s=0.2,
                        outlier_rate=0.2)

# 1. OLS
model = LinearRegression()
result = model.fit(x.reshape(-1,1), y)
```



```
# Visualize the data and regression line
w = result.coef_
b = result.intercept_
y_hat = np.dot(w, x.T) + b

plt.figure(figsize=(6,5))
plt.scatter(x, y, s=5, c='r')
plt.plot(x, y_hat, c='blue')
plt.axvline(x=0, ls='--', lw=0.5, c='black')
plt.axhline(y=0, ls='--', lw=0.5, c='black')
plt.show()

print('\nOLS results:')
print('Regression line: y = {:.3f}x + {:.3f}'.format(w[0], b))
print('R2 score = {:.3f}'.format(r2_score(y, y_hat)))

# RANSAC
n_sample = 10      # the number of samples chosen randomly from data
z_prob = 0.99      # the probability z
w_prob = 0.8       # the probability w

$$k = \frac{\log(1-z)}{\log(1-w^n)}$$


# The maximum number of attempts to find a consensus set
k_maxiter = int(np.log(1.0-z_prob) / np.log(1.0-w_prob ** n_sample))

# RANSACRegressor/residual_threshold:
# the threshold is chosen as the MAD (median absolute deviation)
# of the target values y
threshold = np.median(np.abs(y - np.median(y)))

ransac_w = 0      # slope
ransac_b = 0      # intercept
ransac_c = 0      # count within the error tolerance
```

■ Implementing RANSAC from scratch

```
for i in range(k_maxiter):
    # sampling without replacement
    idx = np.random.choice(np.arange(0, x.shape[0]-1),
                           n_sample, replace=False)

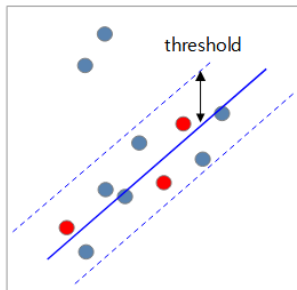
    xs = x[idx]
    ys = y[idx]

    # OLS Regression
    model = LinearRegression()
    result = model.fit(xs, ys)

    # Calculate the absolute value of residuals.
    y_pred = np.dot(result.coef_, x.T) + result.intercept_
    residual = np.abs(y - y_pred)

    # Count the number of times the residual is less than the
    # threshold.
    count = (residual < threshold).sum()

    # Find the regression line where
    # the count is largest.
    if count > ransac_c:
        ransac_c = count
        ransac_w = result.coef_
        ransac_b = result.intercept_
```

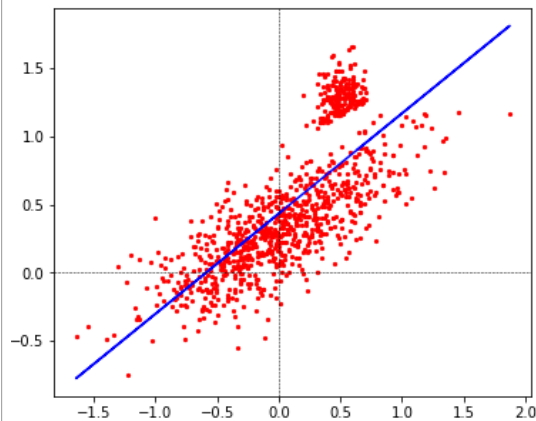


```
y_pred = np.dot(ransac_w, x.T) + ransac_b
```

```
# Visually check the data and final regression line
plt.figure(figsize=(6,5))
plt.scatter(x, y, s=5, c='r')
plt.plot(x, y_pred, c='blue')
plt.axvline(x=0, ls='--', lw=0.5, c='black')
plt.axhline(y=0, ls='--', lw=0.5, c='black')
plt.show()
```

```
print('\nRANSAC results:')
print('The maximum number of k = {}'.format(k_maxiter))
print('Threshold = {:.3f}'.format(threshold))
print('Regression line: y = {:.3f}x + {:.3f}'.\
      format(ransac_w[0], ransac_b))
print('R2 score = {:.3f}'.format(r2_score(y, y_pred)))
```

OLS results

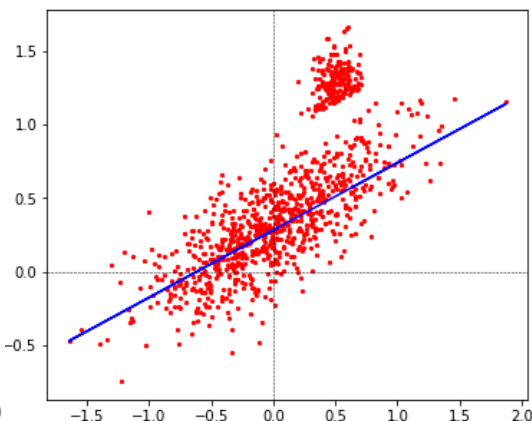


OLS results:

Regression line: $y = 0.735x + 0.433$

R2 score = 0.548

RANSAC results



RANSAC results:

The maximum number of k = 40

Threshold = 0.291

Regression line: $y = 0.459x + 0.283$

R2 score = 0.342

■ Implementing RANSAC using sklearn's RANSACRegressor

```
# [MXML-3-07] 10.ransac(2).py
# Implementing RANSAC using sklearn's RANSACRegressor.
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression, RANSACRegressor
from sklearn.metrics import r2_score

# Generate 1,000 data samples with outliers.
x, y = reg_data_outlier(a=0.5, b=0.3, n=1000, s=0.2,
                        outlier_rate=0.2)

# min_samples:
#   min_samples is chosen as X.shape[1] + 1.
# stop_probability:
#   RANSAC iteration stops if at least one outlier-free set of
#   the training data is sampled in RANSAC. This requires to
#   generate at least N samples (iterations):
# residual_threshold:
#   By default the threshold is chosen as the MAD (median
#   absolute deviation) of the target values y.
model = RANSACRegressor(LinearRegression(),
                        stop_probability = 0.99,      # default
                        residual_threshold = None,    # default
                        min_samples = 10)

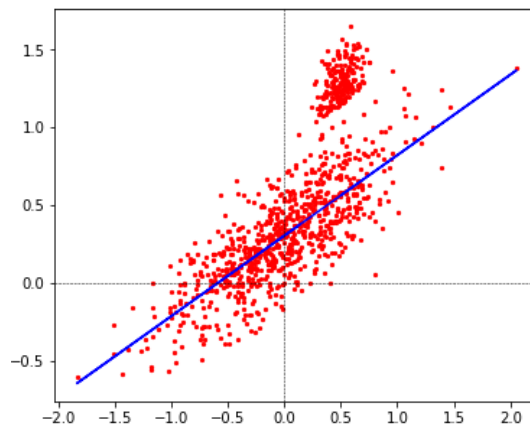
model.fit(x, y)

w = model.estimator_.coef_
b = model.estimator_.intercept_
```

Visually check the data and final regression line

```
y_pred = model.predict(x)
plt.figure(figsize=(6,5))
plt.scatter(x, y, s=5, c='r')
plt.plot(x, y_pred, c='blue')
plt.axvline(x=0, ls='--', lw=0.5, c='black')
plt.axhline(y=0, ls='--', lw=0.5, c='black')
plt.show()

print('\nRANSAC results:')
print('Regression line: y = {:.3f}x + {:.3f}'.format(w[0], b))
print('R2 score = {:.3f}'.format(r2_score(y, y_pred)))
```



RANSAC results:

Regression line: $y = 0.518x + 0.303$

R2 score = 0.430

■ Predict the Boston house prices using RANSAC

```
# [MXML-3-07] 11.boston(ransac).py
# Predict the Boston house prices using RANSAC
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import RANSACRegressor, Ridge
from sklearn.model_selection import train_test_split
import pickle

# Read Boston house price dataset
with open('data/boston_house.pkl', 'rb') as f:
    data = pickle.load(f)
x = data['data']      # shape = (506, 13)
y = data['target']    # shape = (506,)
x_train, x_test, y_train, y_test = train_test_split(x, y)

# min_samples: min_samples is chosen as X.shape[1] + 1.
# stop_probability:
#   RANSAC iteration stops if at least one outlier-free set of
#   the training data is sampled in RANSAC. This requires to
#   generate at least N samples (iterations):
# residual_threshold:
#   By default the threshold is chosen as the MAD (median
#   absolute deviation) of the target values y.
model = RANSACRegressor(Ridge(alpha=0.01),
                        stop_probability = 0.99, # default
                        residual_threshold = None, # default
                        min_samples = 50,
                        max_trials = 1000)
```

```
model.fit(x_train, y_train)
```

```
# Visually check the actual and predicted prices
```

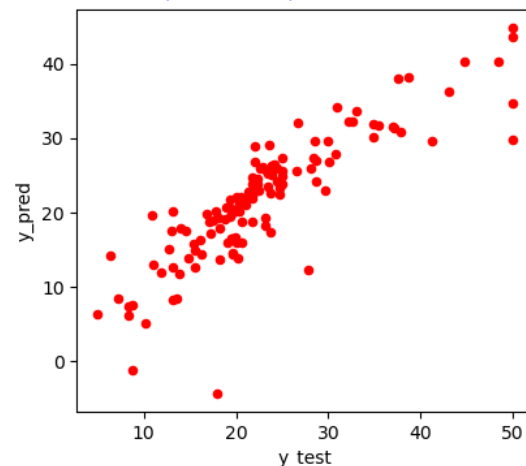
```
y_pred = model.predict(x_test)
plt.figure(figsize=(6, 5))
plt.scatter(y_test, y_pred, s=20, c='r')
plt.xlabel('y_test')
plt.ylabel('y_pred')
plt.show()
```

```
print('RANSAC R2 = {:.3f}'.format(model.score(x_test, y_test)))
```

Results from multiple runs:

1. RANSAC R2 = 0.711
2. RANSAC R2 = 0.492
3. RANSAC R2 = 0.642
4. RANSAC R2 = 0.747
5. RANSAC R2 = 0.663

Relationship between the actual and predicted prices.



- Predict Boston house prices using different models and compare their prediction performance.
 - Randomly generate training and test data, perform linear regression on the training data, and measure the R-squared score on the test data.
 - The linear regression was performed 10 times for each model.
 - The results of measuring the R-squared score are as follows. All have ridge regularization applied.
 - LWR performed best, followed by TLS.

R-squared score by model

No	OLS	TLS	LWR	RANSAC
1	0.741	0.717	0.819	0.599
2	0.692	0.78	0.815	0.729
3	0.738	0.768	0.749	0.684
4	0.678	0.618	0.782	0.725
5	0.737	0.657	0.885	0.750
6	0.689	0.793	0.831	0.604
7	0.653	0.817	0.857	0.620
8	0.724	0.746	0.78	0.660
9	0.711	0.83	0.831	0.522
10	0.692	0.785	0.843	0.670
Average	0.7055	0.7511	0.8192	0.656