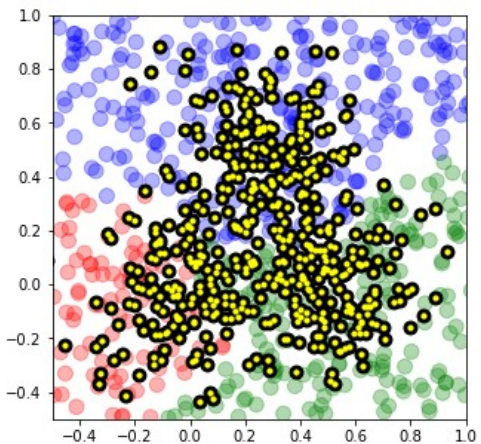


1. K-Nearest Neighbors (KNN)

Part 1: The basics of KNN classification algorithm

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

www.youtube.com/@meanxai

MX-AI

[Classification]

[Regression]

MX-AI

A brief history of KNN

- In statistics, the k-nearest neighbors algorithm (k-NN) is a non-parametric supervised learning method first developed by Evelyn Fix and Joseph Hodges in 1951[1], and later expanded by Thomas Cover[2]. It is used for classification and regression. In both cases, the input consists of the k closest training examples in a data set. The output depends on whether k-NN is used for classification or regression: [source : Wikipedia]

[1]

DISCRIMINATORY ANALYSIS
NONPARAMETRIC DISCRIMINATION: CONSISTENCY PROPERTIES

Evelyn Fix, Ph.D.
J.L. Hodges, Jr., PhD.
University of California, Berkeley

1. Introduction

The discrimination problems (two population case) may be defined as follows: a random variable Z, of observed value z, is distributed over some space (say, p-dimensional) either according to distribution F, or according to distribution G. The problem is to decide, on the basis of z, which of the two distributions Z has.

The problem may be classified in various ways into sub-problems On pertinent method of classification is according to the amount of information assumed to be available about F and G. We may distinguish three stages:

(i) F and G are completely known
(ii) F and G are known except for the values of one or more parameters
(iii) F and G are completely unknown, except possibly for assumptions about existence of densities, etc.

IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. IT-IS, NO. 1, JANUARY 1967 [2]

Nearest neighbor pattern classification

T. M. COVER, MEMBER, IEEE, AND P. E. HART, MEMBER, IEEE

Abstract-The nearest neighbor decision rule assigns to an unclassified sample point the classification of the nearest of a set of previously classified points. This rule is independent of the underlying joint distribution on the sample points and their classifications, and hence the probability of error R of such a rule must be at least as great as the Bayes probability of error R*--the minimum probability of error over all decision rules taking underlying probability structure into account. However, in a large sample analysis, we will show in the M-category case that R* ≤ R ≤ R*(2 - MR*/(M-1)), where these bounds are the tightest possible, for all suitably smooth underlying distributions. Thus for any number of categories, the probability of error of the nearest neighbor rule is bounded above by twice the Bayes probability of error. In this sense, it may be said that half the classification information in an infinite sample set is contained in the nearest neighbor.

I. INTRODUCTION

IN THE CLASSIFICATION problem there are two extremes of knowledge which the statistician may possess. Either he may have complete statistical knowledge of the underlying joint distribution of the observation 2 and the true category ...

MX-AI

KNN Classification – Algorithm

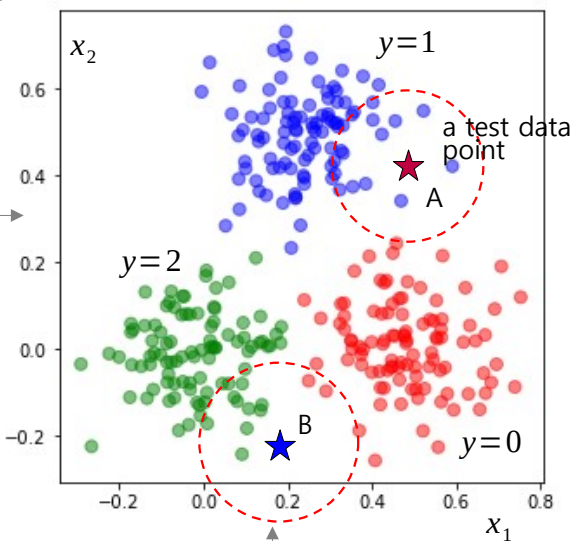
- Goal : Use the training data to estimate the class y of the test data.

[Training data]

feature target class

x_1	x_2	y
0.16	0.01	0
-0.09	-0.03	0
0.06	-0.08	0
0.58	-0.1	2
0.46	-0.07	2
0.23	0.54	1
0.55	0.07	2
0.1	0.42	1
0.18	0.63	1
-0.03	-0.09	0
0.59	-0.14	2
0.32	0.06	2
-0.08	-0.03	0
0.27	0.52	1
⋮	⋮	⋮

feature space (class y in color)



10 data points in this circle

[Test data]

feature target class

	x_1	x_2	y
A	0.5	0.4	? → Estimate to be "1"
B	0.19	-0.21	? → Estimate to be "2"

- **KNN Algorithm**

1. Compute the distances between each test data point and all the training data points.
2. Find K neighboring data points close to each test data point. (ex : K=10)
3. Find all classes (y) of the neighbors.
4. Find the majority class of the nearest neighbors, and then classify the test data point into that class. (Majority Voting).

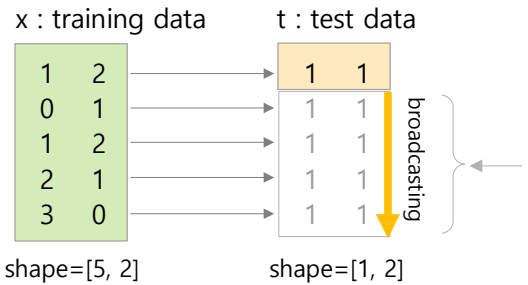
- Since all of the neighboring classes of the test data point A are 1, we can confidently estimate that the class y of this test data point is 1. This data point has similar properties to the data points belonging to class 1.
- For the test data point B, there are 8 data points with class 2, and 2 data points with class 0 inside this circle, so we can estimate that the class y of this test data point is 2 with 80% probability. It has similar properties to the data points belonging to class 2.
- In this example, since the data is two-dimensional, it is easy to estimate the classes even with the naked eye, but this is not the case for high-dimensional data. Therefore, we need a generalized algorithm, like KNN.

* **Issues** : 1. Choosing an appropriate K value.
2. Different types of distances.
3. A curse of dimensionality.
4. Lazy learner.
5. Weighted distance.

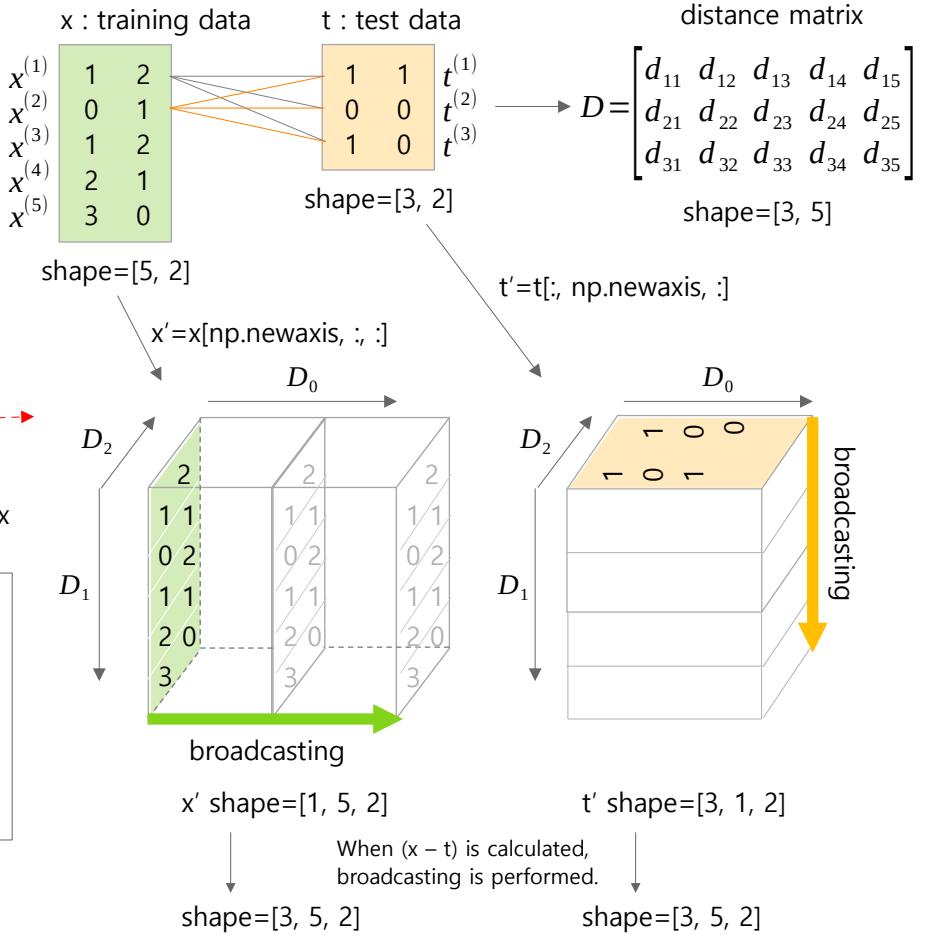
Creating a distance matrix via broadcasting

- The distance matrix (D) between the training and test data points can be found using multiple for loops, but can be easily found using array broadcasting.

  - **Broadcasting**



Computing (x – t) performs broadcasting. shape=[5, 2]



- Python code for calculating the distance matrix (dist) between 5 training data points x and 3 test data points t in the figure on the right.

```
x_exp = x[np.newaxis, :, :]     # Add a new axis at D0 of x. (1, 5, 2)
t_exp = t[:, np.newaxis, :]     # Add a new axis at D1 of t. (3, 1, 2)

# Calculating the distances between the training and test data points.
# The shape of dist will be (3, 5)
dist = np.sqrt(np.sum(np.square(x_exp - t_exp), axis=2))
```

* Distance between the first training data point (1,2) and the first test data point (1,1): → $d = \sqrt{(1-1)^2 + (2-1)^2}$

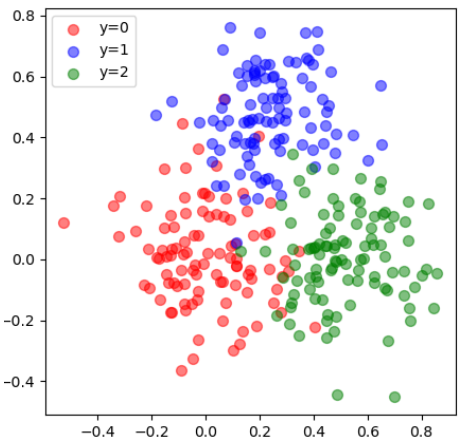
MX-AI

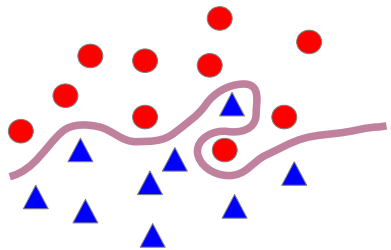
■ Write code to compute distance matrix and perform KNN classification.

```python
# [MXML-1-01] 1.KNN(distance).py
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split

# create a dataset
x, y = make_blobs(n_samples=300, n_features=2,
                  centers=[[0., 0.], [0.25, 0.5], [0.5, 0.]],
                  cluster_std=0.15, center_box=(-1., 1.))

# Visualize the dataset and classes by color
plt.figure(figsize=(5, 5))
for i, color in enumerate(['red', 'blue', 'green']):
    p = x[y==i]
    plt.scatter(p[:, 0], p[:, 1], s=50, c=color,
                label='y=' + str(i), alpha=0.5)
plt.legend()
plt.show()

# split dataset into train and test data
x_train, x_test, y_train, y_test = train_test_split(x, y)
K = 10  # the number of nearest neighbors

# 1. Create a distance matrix.
d_train = x_train[np.newaxis, :, :]  # Add an axis at D0
d_test = x_test[:, np.newaxis, :]    # Add an axis at D1
distance = np.sqrt(np.sum((d_train - d_test) ** 2, axis=2))
```

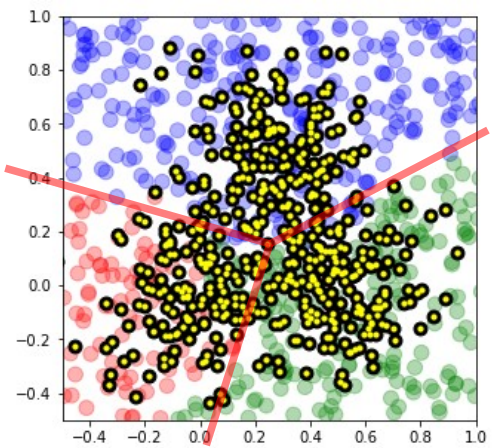
```python
# 2. Find K nearest neighbors
i_near = np.argsort(distance, axis=1)[:, :K]
y_near = y_train[i_near]

# 3. majority voting
y_pred = np.array([np.bincount(p).argmax() for p in y_near])

# Measure the accuracy for test data
print('Accuracy = {:.4f}'.format((y_pred == y_test).mean()))
```

```
distance: shape=(75, 225)
[[0.84, 1.18, 0.95, ..., 0.32, 0.53, 0.62],
 [0.67, 0.23, 0.03, ..., 0.68, 0.62, 0.83],
 [0.06, 0.82, 0.7 , ..., 0.54, 0.32, 0.25],
 ...,
 [0.16, 0.75, 0.59, ..., 0.37, 0.15, 0.22],
 [0.53, 0.26, 0.13, ..., 0.65, 0.54, 0.72],
 [0.58, 0.24, 0.3 , ..., 0.83, 0.68, 0.82]]

i_near[:3]: shape=(75, 10)
[[ 86,  16,  91, 115,  27,  32, 137, 122,  78, 164],
 [  2, 200,  54,  10, 120,  19,  77,  15,  40,  62],
 [149, 112, 143,   0, 101, 202, 159,  65, 118, 201]]

y_near[:3]
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [0, 2, 0, 2, 1, 2, 0, 2, 1, 0],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

y_pred[:3]
[1, 0, 1]

Accuracy = 0.9467
```





Decision boundary



1. K-Nearest Neighbors (KNN)

Part 2: Optimal K value and decision boundary

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

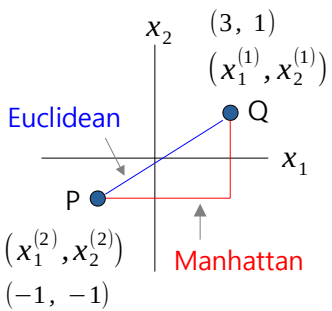
www.youtube.com/@meanxai

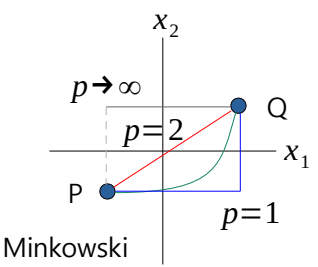
MX-AI

Different types of distances used in KNN

- There are several ways to calculate the distance between two points.
- The **Manhattan** distance between two points is defined as the sum of the absolute differences of their coordinates. In the figure below, Manhattan distance between the point P and the point Q is 6.
- The **Euclidean** distance between two points is defined as the length of the line segment between the two points. In the figure below, Euclidean distance between P and Q is 4.47.
- The **Minkowski** distance is a generalization of the Manhattan and Euclidean distances. When p=1, it becomes Manhattan distance, and when p=2, it becomes Euclidean distance.

- Manhattan distance: $d = |x_1^{(1)} - x_1^{(2)}| + |x_2^{(1)} - x_2^{(2)}|$
- Euclidean distance: $d = \sqrt{(x_1^{(1)} - x_1^{(2)})^2 + (x_2^{(1)} - x_2^{(2)})^2}$
- Minkowski distance: $d = \sqrt[p]{|x_1^{(1)} - x_1^{(2)}|^p + |x_2^{(1)} - x_2^{(2)}|^p}$





Data normalization

- Data normalization is a process that adjusts the observations measured on different scales to a common scale.
- The most commonly used are Min-Max normalization and Z-score normalization.
- Z-score normalization is also called standardization.
- Min-Max normalization transforms data by scaling values so that they fall between a predetermined minimum and maximum value (typically between 0 and 1).
- Z-score normalization transforms data so that it has a mean of 0 and a standard deviation of 1. This process adjusts data values based on how far they deviate from the mean, measured in units of standard deviation.

	feature		class
i	x_1	x_2	y
0	0.16	10.01	0
1	-0.09	-20.03	0
2	0.06	-10.08	0
3	0.58	-13.1	2
4	0.46	-11.07	2
5	0.23	12.54	1
6	0.55	15.07	2
7	0.1	21.42	1
8	0.18	17.63	1
9	-0.03	-19.09	0
10	0.59	-23.14	2
11	0.32	29.06	2
12	-0.08	-18.03	0
13	0.27	16.52	1

- Min-Max normalization between a and b

$$x'^{(i)}_k = \frac{x_k^{(i)} - min(x_k)}{max(x_k) - min(x_k)} \cdot (b-a) + a$$

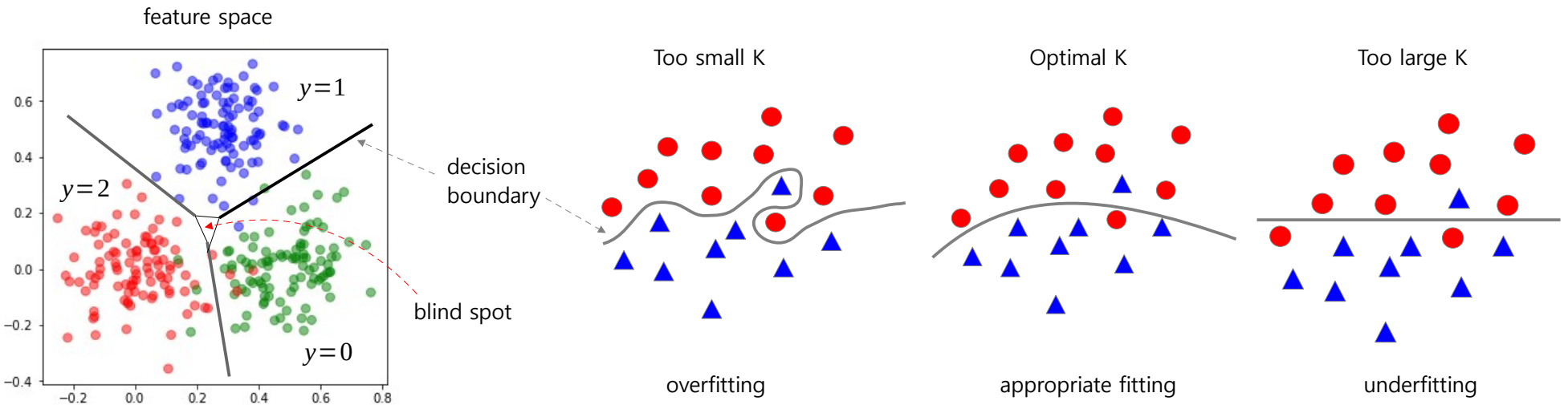
- Z-score normalization

$$x'^{(i)}_k = \frac{x_k^{(i)} - mean(x_i)}{std(x_i)}$$

MX-AI

Decision boundary depending on the value of K

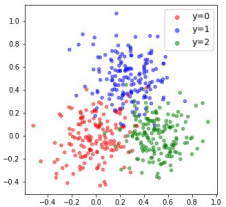
- A decision boundary is a surface that separates different classes in feature space for a classification problem. It's a line in 2D, a plane in 3D, or a hyperplane in higher dimensions.
- The decision boundary depends on the choice of K value in KNN. A smaller value of k considers fewer neighboring data points, which makes the decision boundary more complex and may lead to overfitting. A larger value of k considers more neighboring data points, which results in a smoother decision boundary and may lead to underfitting.
- The optimal value of K can be selected through cross-validation, which can prevent overfitting or underfitting and improve the performance of a model.
- Setting K to sqrt(N) could be a good choice as default, where N=the number of training data points.



MX-AI

- Implementation of a KNN classification model, including how to find optimal K value

```
# [MXML-1-02] 2.KNN(optimal_k).py
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.model_selection import train_tes

# Create dataset
x, y = make_blobs(n_samples=900, n_features=2,
                  centers=[[0., 0.], [0.25, 0.5], [0.5, 0.]],
                  cluster_std=0.2, center_box=(-1., 1.))

# Visualize the dataset and classes by color
plt.figure(figsize=(5, 5))
for i, color in enumerate(['red', 'blue', 'green']):
    p = x[y==i]
    plt.scatter(p[:, 0], p[:, 1], s=20, c=color,
                label='y=' + str(i), alpha=0.5)
plt.legend()
plt.show()

# Split the dataset into training and test data
x_train, x_test, y_train, y_test = \
                 train_test_split(x, y, test_size=0.2)
N = x_train.shape[0]

# Z-score Normalization.
# The feature values in this data set have similar scales
# from -1 to +1, so there is no need to normalize them.
# But let's try this just for practice.
```



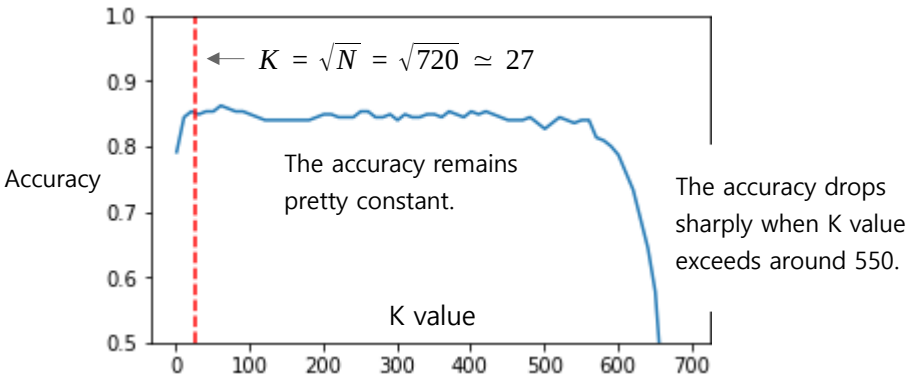

```
# Calculate the mean and standard deviation from the training data
# and apply them to the test data.
mean = x_train.mean(axis=0)
std = x_train.std(axis=0)
z_train = (x_train - mean) / std
z_test = (x_test - mean) / std

# A function for performing the KNN classification algorithm.
def knn_predict(train, test, k):
    # 1. Create a distance matrix.
    d_train = train[np.newaxis, :, :]   # Add a new axis at D0
    d_test = test[:, np.newaxis, :]     # Add a new axis at D1

    p = 2  # Euclidean distance
    d = np.sum(np.abs(d_train - d_test) ** p, axis=-1) ** (1/p)

    # 2. Find K nearest neighbors
    i_nearest = np.argsort(d, axis=1)[:, :k]  # index
    y_nearest = y_train[i_nearest]

    # 3. majority voting
    return np.array([np.bincount(i).argmax() for i in y_nearest])

# Measure the accuracy of the test data while changing K value.
accuracy = []
k_vals = np.arange(1, 700, 10)
for k in k_vals:
    # Estimate the classes of all test data points and measure the accuracy.
    y_pred = knn_predict(z_train, z_test, k)
    accuracy.append((y_pred == y_test).mean())
```

MX-AI

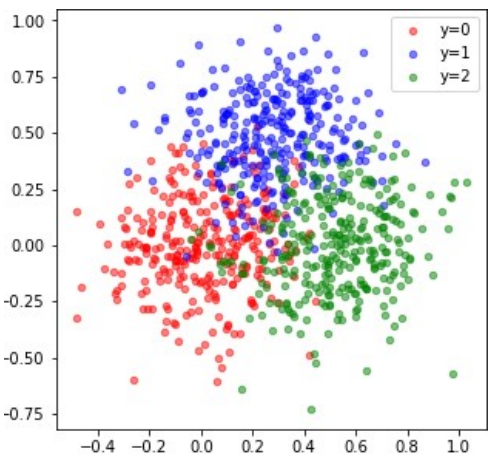
- Implementation of a KNN classification model, including how to find optimal K value

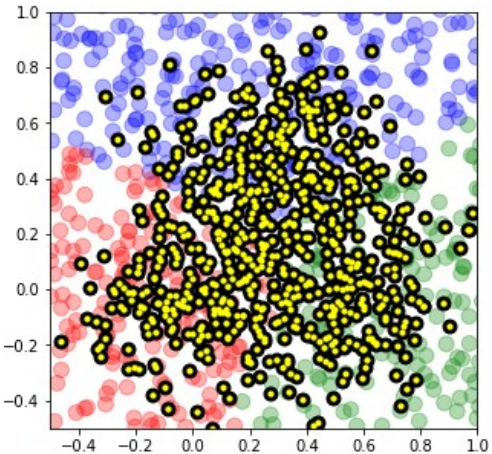
```
# Observe how the accuracy changes as K changes.
plt.figure(figsize=(5, 3))
plt.plot(k_vals, accuracy, 'o-')
plt.axvline(x=np.sqrt(N), c='r', ls='--')
plt.ylim(0.5, 1)
plt.show()

# Generate a large number of test data points and roughly
# determine the decision boundary.
x_many = np.random.uniform(-0.5, 1.5, (1000, 2))
z_many = (x_many - mean) / std
y_many = knn_predict(z_train, z_many, k=int(np.sqrt(N)))

# Check the decision boundary
plt.figure(figsize=(5,5))
color = [['red', 'blue', 'green'][a] for a in y_many]
plt.scatter(x_many[:, 0], x_many[:, 1], s=100, c=color, alpha=0.3)
plt.scatter(x_train[:, 0], x_train[:, 1], s=80, c='black')
plt.scatter(x_train[:, 0], x_train[:, 1], s=10, c='yellow')
plt.xlim(-0.5, 1.0)
plt.ylim(-0.5, 1.0)
plt.show()
```

Changes in accuracy with changes in K value

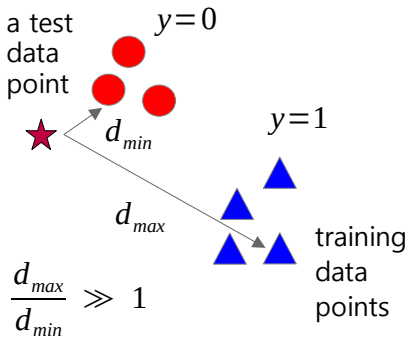


Decision boundaries

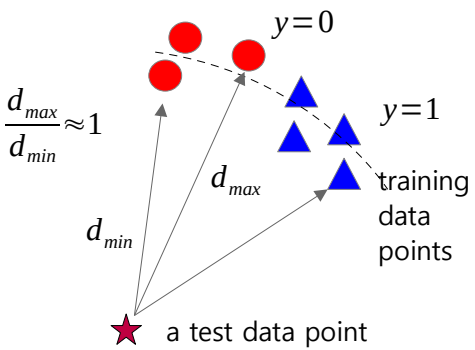




A low-dimensional space



A high-dimensional space



1. K-Nearest Neighbors (KNN)

Part 3: Curse of Dimensionality and Lazy learner

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

www.youtube.com/@meanxai

MX-AI

■ High-dimensional data

- The more features in a dataset, the higher the dimensionality of the feature space.
- For example, the Iris dataset cannot be said to be high-dimensional because it has only 4 features, but the MNIST dataset can be said to be high-dimensional because it has 784 features.

▪ Iris dataset

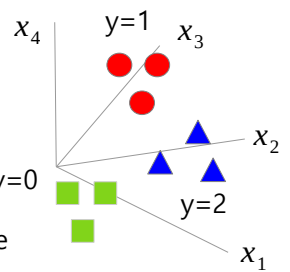
[training data] features target class

	x_1	x_2	x_3	x_4	y
id	sepal length	sepal width	petal length	petal width	spec ies
1	6.1	2.8	4.7	1.2	1
2	6.7	3.1	4.7	1.5	1
3	5.6	3	4.5	1.5	1
4	5.3	3.7	1.5	0.2	0
5	6.5	3.2	5.1	2	2
6	7.7	3.8	6.7	2.2	2
7	6.3	3.3	4.7	1.6	1
8	6.9	3.1	4.9	1.5	1
9	6.2	2.8	4.8	1.8	2
10	4.6	3.2	1.4	0.2	0
11	6.3	3.3	6	2.5	2

[test data]

	x_1	x_2	x_3	x_4	y
id	sepal length	sepal width	petal length	petal width	species
A	6.1	2.8	4.7	1.2	??
B	6.7	3.1	4.7	1.5	??

4 dimensional feature space



distance

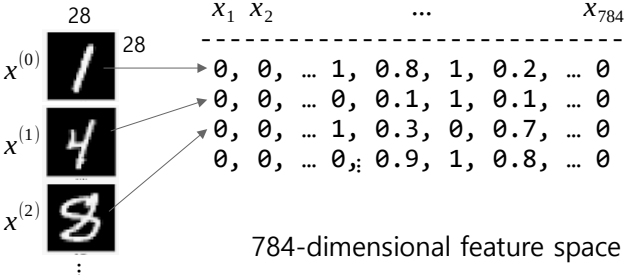
$$d_{i,j} = \sqrt{(x_1^{(i)}-x_1^{(j)})^2+(x_2^{(i)}-x_2^{(j)})^2+(x_3^{(i)}-x_3^{(j)})^2+(x_4^{(i)}-x_4^{(j)})^2}$$

We cannot imagine the shape of a data distribution in a high-dimensional space, but we can mathematically compute the distance between two data points.

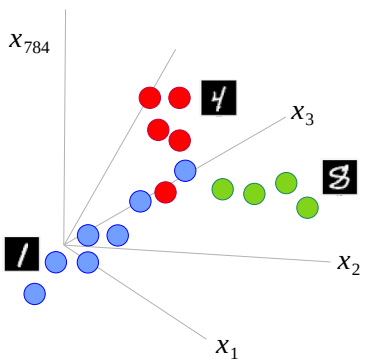
▪ MNIST dataset

original data (2D image)

Convert a 28 x 28 2D image to a 784-dimensional vector.



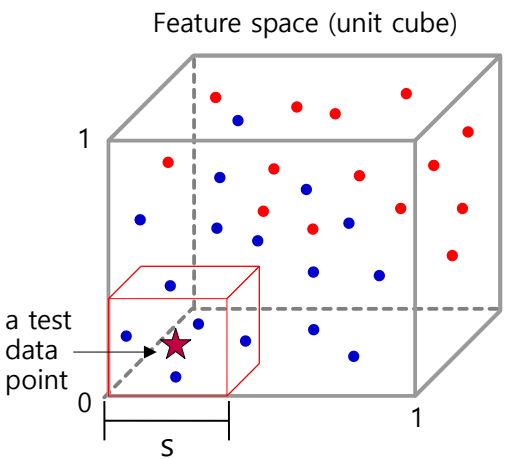
784-dimensional feature space



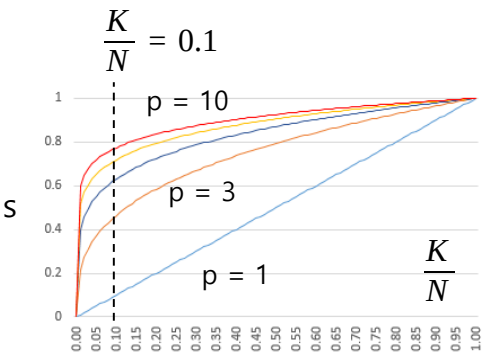
MX-AI

Curse of Dimensionality

- Since KNN is an algorithm that uses the distance between data points in the feature space, its performance may deteriorate as the number of features increases. This phenomenon is called the **curse of dimensionality**. As the number of features in a dataset increases, the amount of data required to cover the feature space increases exponentially.
- Let's say we have a dataset with three features, whose values range from 0 to 1. We want to find neighbors that lie within a cube centered around the test point that contains 10% of the training data points. Then we need a cube with side length 0.464. p=3, K/N = 0.1, the length of a side $s = 0.1^{1/3} = 0.464$. (p: the number of features, N: the number of training data points, K: K value in KNN)
- If p is 10, i.e. a 10-dimensional feature space, s increases to 0.8 ($s = 0.1^{1/10} = 0.8$). This is 80% of the length of one side of a full cube. This means that in high-dimensional space, the neighbors are far away from the test data point.

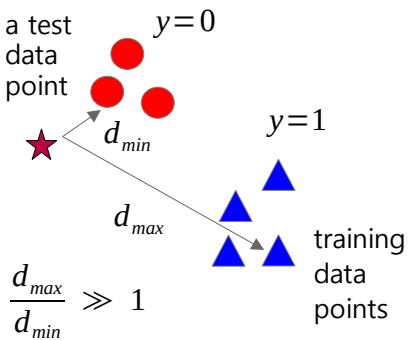


$$s = \left(\frac{K}{N}\right)^{\frac{1}{p}} = 0.1^{\frac{1}{3}} = 0.464$$

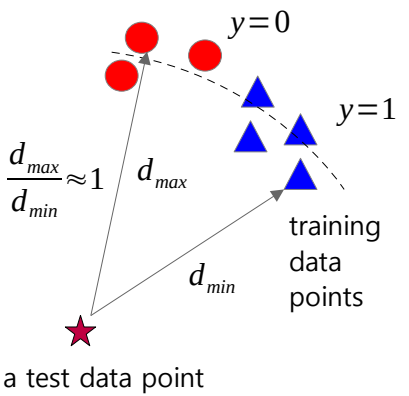


As p increases, s also increases rapidly, so the volume of the cube containing K neighbors also increases rapidly.

a low-dimensional space



a high-dimensional space



* reference: Hastie, et, al. 2009, The Elements of Statistical Learning, p22, Fig 2.6.

MX-AI

■ Observation of the ratio of minimum to maximum distances in a high-dimensional feature space.

```python
# [MXML-01-03] 3.maxmin_ratio.py
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.decomposition import PCA

# Load the MNIST dataset
mnist = fetch_openml('mnist_784', parser='auto')
x = np.array(mnist['data']) / 255

# Compute the distances between a single data point and all other
# data points in a given data set.
def distance(data):
    # Randomly choose a single data point from the dataset.
    i = np.random.randint(0, data.shape[0])
    tp = data[i]

    # Remove the chosen data point from the dataset.
    xp = np.delete(data, i, axis=0)

    # Compute the distances between tp and xp
    d = np.sqrt(np.sum((xp - tp) ** 2, axis=-1))

    # Return the minimum distance and maximum distance
    return d.min(), d.max()
```

```python
# Compute the average ratio of minimum to maximum distances
# in a 784-dimensional feature space
r_maxmin = []
for i in range(10):
    dmin, dmax = distance(x)
    r_maxmin.append(dmax / dmin)
print("max-min ratio (p=784): {0:.2f}".format(np.mean(r_maxmin)))

# Compute the average ratio of minimum to maximum distances
# in a 5-dimensional feature space
pca = PCA(n_components=5)
pca.fit(x)
x_pca = pca.transform(x)

r_maxmin = []
for i in range(10):
    dmin, dmax = distance(x_pca)
    r_maxmin.append(dmax / dmin)
print("max-min ratio (p=5)  : {0:.2f}".format(np.mean(r_maxmin)))
```

```
Results:
max-min ratio (p=784): 4.78
max-min ratio (p=5)  : 36.23

max-min ratio (p=784): 3.50
max-min ratio (p=5)  : 50.18
```

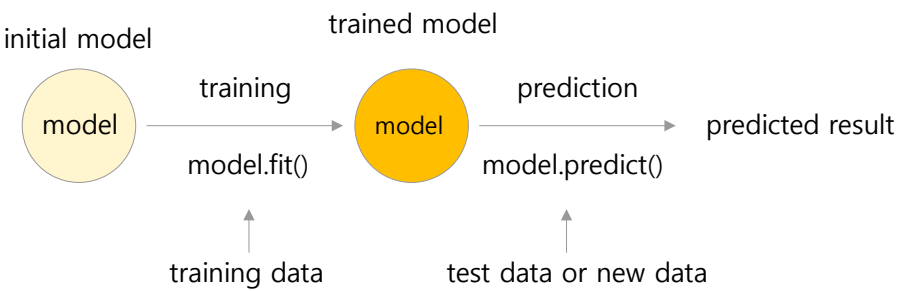
MX-AI

Lazy learner

- KNN is often called a lazy learner. KNN does not build a model during training, but instead simply memorizes all the training data points. Whenever a prediction is needed, KNN needs to measure the distance between the test data and the memorized training data. So it takes a long time to predict.
- It's also known as instance-based learning.

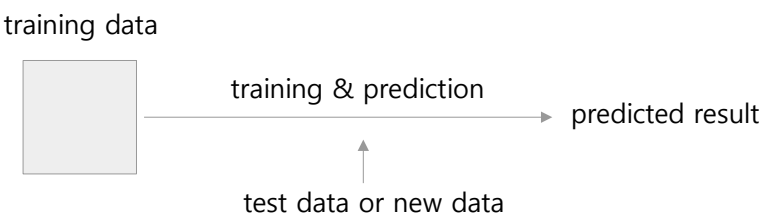
Eager learning (model-based learning)

- A model is built by fitting to the training data and then used to make predictions on the test data or new data.
- Model-based learning takes time to train, but once trained, it can make predictions quickly. It is typically faster than instance-based learning.
- Model-based learning often has higher prediction accuracy than instance-based learning because the model is trained on a large dataset and can generalize to new data.
- Most machine learning models, such as decision trees, support vector machines, and artificial neural networks, work this way.



Lazy learning (instance-based learning)

- In instance-based learning, no model is built first. It simply stores all the training data and uses the stored data to make predictions on test data or new data whenever needed.
- Instance-based learning is typically slower than model-based learning because it must compute something between the new data points and all training data points every time a prediction is made.
- KNN is a typical example of instance-based learning.



MX-AI

- Implementation of a KNN classification model using scikit learn's KNeighborsClassifier

```
# [MXML-01-03] 4.KNN(sklearn).py
import numpy as np
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Load the Iris dataset.
# x: data, the number of samples=150, the number of features=4
# y: target data with class (0,1,2)
x, y = load_iris(return_X_y=True)

# Split the dataset to training, validation and test data
x_train, x_test, y_train, y_test=train_test_split(x, y, \
                                        test_size = 0.4)
x_val, x_test, y_val, y_test=train_test_split(x_test, y_test,\
                                        test_size = 0.5)
# Z-score normalization
mean = x_train.mean(axis=0)
std = x_train.std(axis=0)

x_train = (x_train - mean) / std # Z-score normalization
x_val = (x_val - mean) / std     # use mean and std from x_train
x_test = (x_test - mean) / std   # use mean and std from x_train

# Set K to sqrt(N)
sqr_k = int(np.sqrt(x_train.shape[0]))
```

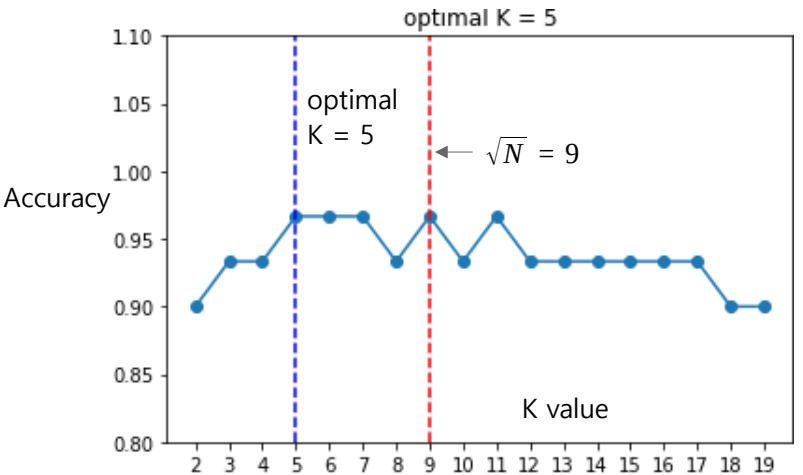
```
# Build a KNN classification model
knn = KNeighborsClassifier(n_neighbors=sqr_k, metric='minkowski', p=2)

# Model fitting. Since KNN is a lazy learner, no learning is performed
# at this step. It simply stores the training data points and the
# parameters.
knn.fit(x_train, y_train)

# Predict the class of validation data.
# The actual learning takes place at this stage, when test or
# validation data is provided.
y_pred = knn.predict(x_val)

# Measure the accuracy on the validation data
accuracy = (y_val == y_pred).mean()
print('\nK: sqr_K = {}, Accuracy on validation data = {:.3f}'\
      .format(sqr_k, accuracy))

# Determine the optimal K.
# Measure the accuracy on the validation data while changing K.
accuracy = []
for k in range(2, 20):
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(x_train, y_train)
    y_pred = knn.predict(x_val)
    accuracy.append((y_val == y_pred).mean())
```

MX-AI

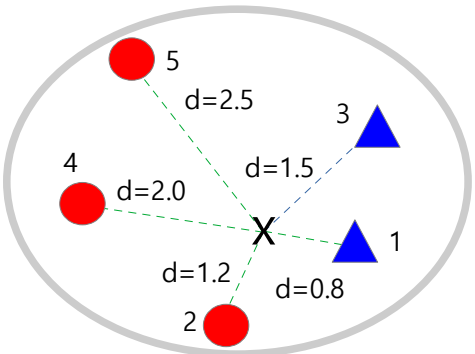
■ Implementation of a KNN classification model using scikit learn's KNeighborsClassifier

```
# Find the optimal K value with the highest accuracy.
opt_k = np.array(accuracy).argmax() + 2

# Observe how the accuracy changes as K changes.
plt.plot(np.arange(2, 20), accuracy, marker='o')
plt.xticks(np.arange(2, 20))
plt.axvline(x = opt_k, c='blue', ls = '--')
plt.axvline(x = sqr_k, c='red', ls = '--')
plt.ylim(0.5, 1.1)
plt.title('optimal K = ' + str(opt_k))
plt.show()

# Finally, we use the test data to measure the final performance
# of the model.
knn = KNeighborsClassifier(n_neighbors = opt_k)
knn.fit(x_train, y_train)
y_pred = knn.predict(x_test)
accuracy = (y_test == y_pred).mean()
print('K: opt_k = {}, Accuracy on test data = {:.3f}'
      .format(opt_k, accuracy))
```

```
K: sqr_K = 9, Accuracy on validation data = 0.967
```




```
K: opt_k = 5, Accuracy on test data = 0.900
```



No	class	distance	Inverse weight
1	▲	0.8	1 / 0.8 = 1.25
2	●	1.2	1 / 1.2 = 0.83
3	▲	1.5	1 / 1.5 = 0.67
4	●	2.0	1 / 2.0 = 0.50
5	●	2.5	1 / 2.5 = 0.40
		sum	3.65

▲ → $\frac{1.25 + 0.67}{3.65} = 0.525$

● → $\frac{0.83 + 0.5 + 0.4}{3.65} = 0.475$

1. K-Nearest Neighbors (KNN)

Part 4: Weighted KNN (WKNN)

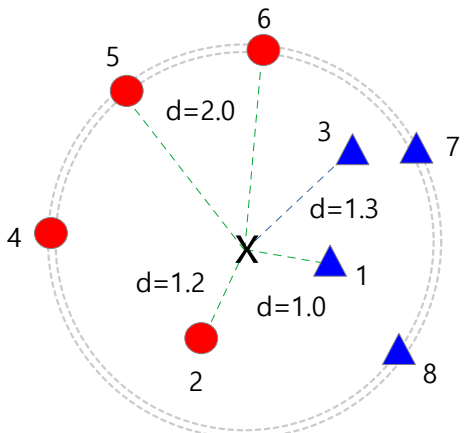
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

■ Different ways to classify test data

- In the following situation, should the test data point X be classified as ● or ▲ ?
- The figure below shows 8 training data points around a test data point. These are the candidates to be the nearest neighbors of the test data point. We want to find the 5 nearest neighbors among these.
- The three candidates 1, 2, 3 are inside a circle. And the other five candidates 4, 5, 6, 7, 8 lie on a circle with a similar distance from the test data point X. For example, the distances are 2.0 ± 0.1. But let's assume that all the distances are 2.0.
- The three candidates (1, 2, 3) can be easily chosen as the neighbors since they are definitely inside the circle. However, choosing two neighbors from the remaining five candidates (4, 5, 6, 7, 8) is not easy because they are all at the same distance from the test data point.
- How can we classify the test data point in this situation?



X : a test data point

● ▲ : training data points

1) Choose randomly

▲ ● ▲ + ● ●

1 2 3 4 6 (If these are chosen)

* X is classified as ●

2) Classify x into the class with smaller sum of the distances.

● : 1.2 + 2.0 * 3 = 7.2

▲ : 1.0 + 1.3 + 2.0 * 2 = 6.3

* X is classified as ▲

3) Classify X into the class with larger sum of the selection probabilities.

No	class	distance	selection probability
1	▲	1.0	1
2	●	1.2	1
3	▲	1.3	1
4	●	2.0	2/5
5	●	2.0	2/5
6	●	2.0	2/5
7	▲	2.0	2/5
8	▲	2.0	2/5

▲ = 1 + 1 + 2/5 + 2/5 =2.8

● = 1 + 2/5 + 2/5 + 2/5 = 2.2

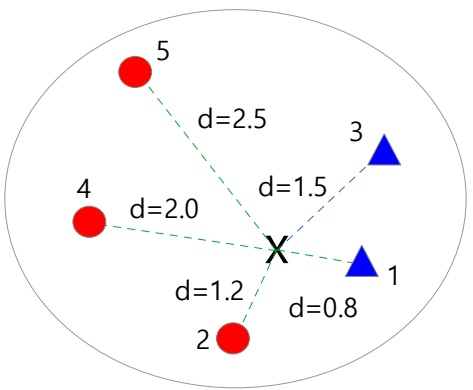
* X is classified as ▲

MX-AI

■ Weighted KNN (WKNN)

- In the following example, when the value of K is 5, does it make sense to classify the data point X as ●, or as ▲?
- In standard KNN, the data point X is classified as ● because ● is the majority. However, since the data point X is closer to ▲, it might make more sense to classify that point as ▲. To do this, you might consider giving a higher weight to ▲.
- Giving higher weights to closer samples is called weighted KNN.
- Since closer neighbors should have higher weights, the weights are defined as the inverse of the distances.

K = 5



X : a test data point

● ▲ : training data points

1) Standard KNN (all neighbors have equal weight)

* X is classified as ●

2) Weighted KNN

No	class	distance	Inverse distance
1	▲	0.8	1 / 0.8 = 1.25
2	●	1.2	1 / 1.2 = 0.83
3	▲	1.5	1 / 1.5 = 0.67
4	●	2.0	1 / 2.0 = 0.50
5	●	2.5	1 / 2.5 = 0.40
		sum	3.65

- The weights of the inverse distance

▲ → $\frac{1.25 + 0.67}{3.65} = 0.525$

● → $\frac{0.83 + 0.5 + 0.4}{3.65} = 0.475$

* X is classified as ▲

MX-AI

■ Implementation of Weighted KNN

```python
# [MXML-1-04] 5.WKNN.py
import numpy as np

# Let's assume that the distance matrix between the test data and
# the training data is given as follows. shape = (5, 10)
dist = np.array(
# train: 0     1    2    3    4    5    6    7    8    9       test
       [[5. , 3.5, 4.3, 3.4, 1.4, 6.5, 2.7, 5.1, 2.9, 2.8],   # i=0
        [4.4, 1.9, 3.6, 3.3, 0.5, 5.5, 2.1, 4.4, 1.3, 2.3],   # i=1
        [4.6, 1. , 3.9, 4.4, 3. , 4.7, 3.2, 4.4, 1.4, 3.5],   # i=2
        [4.7, 0.6, 3.9, 4.1, 1.7, 5.3, 2.7, 4.6, 0.4, 3. ],   # i=3
        [3. , 3.6, 2.4, 1.4, 2.4, 4.8, 1.2, 3.2, 3. , 1.1]])  # i=4

# target class y   (0  1  2  3  4  5  6  7  8  9)
y_train = np.array([0, 1, 1, 0, 1, 0, 1, 1, 0, 0])
C = [0, 1]   # the class y is either 0 or 1
K = 7        # 7-nearest neighbors
T = 5        # the number of test data points

# Find K nearest neighbors
i_near = np.argsort(dist, axis=1)[:, :K]
y_near = y_train[i_near]

# Compute the inverse distance
w_dist = np.array([dist[i, :][i_near[i, :]] for i in range(T)])
w_inv = 1. / w_dist

# Predict the class of test data using the inverse weighted distance
y_pred = []
for i in range(T):
    iw_dist = [w_inv[i][y_near[i] == j].sum() for j in C]
    y_pred.append(np.argmax(iw_dist / w_inv[i].sum()))
```

```
i_near: [[4, 6, 9, 8, 3, 1, 2],
         [4, 8, 1, 6, 9, 3, 2],
         [1, 8, 4, 6, 9, 2, 3],
         [8, 1, 4, 6, 9, 2, 3],
         [9, 6, 3, 2, 4, 0, 8]])

y_near: [[1, 1, 0, 0, 0, 1, 1],
         [1, 0, 1, 1, 0, 0, 1],
         [1, 0, 1, 1, 0, 1, 0],
         [0, 1, 1, 1, 0, 1, 0],
         [0, 1, 0, 1, 1, 0, 0]])

w_dist: [[1.4, 2.7, 2.8, 2.9, 3.4, 3.5, 4.3],
         [0.5, 1.3, 1.9, 2.1, 2.3, 3.3, 3.6],
         [1. , 1.4, 3. , 3.2, 3.5, 3.9, 4.4],
         [0.4, 0.6, 1.7, 2.7, 3. , 3.9, 4.1],
         [1.1, 1.2, 1.4, 2.4, 2.4, 3. , 3. ]])

w_inv:  [[0.71, 0.37, 0.36, 0.34, 0.29, 0.29, 0.23],
         [2.  , 0.77, 0.53, 0.48, 0.43, 0.3 , 0.28],
         [1.  , 0.71, 0.33, 0.31, 0.29, 0.26, 0.23],
         [2.5 , 1.67, 0.59, 0.37, 0.33, 0.26, 0.24],
         [0.91, 0.83, 0.71, 0.42, 0.42, 0.33, 0.33]])

# Compute the weight of class 0 for the first test data point.
w_inv[0][y_near[0] == 0].sum() / w_inv[0].sum() → 0.38

# Compute the weight of class 1 for the first test data point.
# y_pred[0] = 1
w_inv[0][y_near[0] == 1].sum() / w_inv[0].sum() → 0.62

y_pred → [1, 1, 1, 0, 0]  # predicted classes of the test data
```

MX-AI

■ Implementation of Weighted KNN

```python
# [MXML-1-04] 5.WKNN.py
import numpy as np

# Let's assume that the distance matrix between the test data and
# the training data is given as follows. shape = (5, 10)
dist = np.array(
# train: 0     1    2    3    4    5    6    7    8    9       test
       [[5. , 3.5, 4.3, 3.4, 1.4, 6.5, 2.7, 5.1, 2.9, 2.8],   # i=0
        [4.4, 1.9, 3.6, 3.3, 0.5, 5.5, 2.1, 4.4, 1.3, 2.3],   # i=1
        [4.6, 1. , 3.9, 4.4, 3. , 4.7, 3.2, 4.4, 1.4, 3.5],   # i=2
        [4.7, 0.6, 3.9, 4.1, 1.7, 5.3, 2.7, 4.6, 0.4, 3. ],   # i=3
        [3. , 3.6, 2.4, 1.4, 2.4, 4.8, 1.2, 3.2, 3. , 1.1]])  # i=4

# target class y   (0  1  2  3  4  5  6  7  8  9)
y_train = np.array([0, 1, 1, 0, 1, 0, 1, 1, 0, 0])
C = [0, 1]   # the class y is either 0 or 1
K = 7        # 7-nearest neighbors
T = 5        # the number of test data points

# Find K nearest neighbors
i_near = np.argsort(dist, axis=1)[:, :K]
y_near = y_train[i_near]

# Compute the inverse distance
w_dist = np.array([dist[i, :][i_near[i, :]] for i in range(T)])
w_inv = 1. / w_dist

# Predict the class of test data using the inverse weighted distance
y_pred = []
for i in range(T):
    iw_dist = [w_inv[i][y_near[i] == j].sum() for j in C]
    y_pred.append(np.argmax(iw_dist / w_inv[i].sum()))
```

```
i_near: [[4, 6, 9, 8, 3, 1, 2],
         [4, 8, 1, 6, 9, 3, 2],
         [1, 8, 4, 6, 9, 2, 3],
         [8, 1, 4, 6, 9, 2, 3],
         [9, 6, 3, 2, 4, 0, 8]])

y_near: [[1, 1, 0, 0, 0, 1, 1],
         [1, 0, 1, 1, 0, 0, 1],
         [1, 0, 1, 1, 0, 1, 0],
         [0, 1, 1, 1, 0, 1, 0],
         [0, 1, 0, 1, 1, 0, 0]])

w_dist: [[1.4, 2.7, 2.8, 2.9, 3.4, 3.5, 4.3],
         [0.5, 1.3, 1.9, 2.1, 2.3, 3.3, 3.6],
         [1. , 1.4, 3. , 3.2, 3.5, 3.9, 4.4],
         [0.4, 0.6, 1.7, 2.7, 3. , 3.9, 4.1],
         [1.1, 1.2, 1.4, 2.4, 2.4, 3. , 3. ]])

w_inv:  [[0.71, 0.37, 0.36, 0.34, 0.29, 0.29, 0.23],
         [2.  , 0.77, 0.53, 0.48, 0.43, 0.3 , 0.28],
         [1.  , 0.71, 0.33, 0.31, 0.29, 0.26, 0.23],
         [2.5 , 1.67, 0.59, 0.37, 0.33, 0.26, 0.24],
         [0.91, 0.83, 0.71, 0.42, 0.42, 0.33, 0.33]])

# Compute the weight of class 0 for the first test data point.
w_inv[0][y_near[0] == 0].sum() / w_inv[0].sum() → 0.38

# Compute the weight of class 1 for the first test data point.
# y_pred[0] = 1
w_inv[0][y_near[0] == 1].sum() / w_inv[0].sum() → 0.62

y_pred → [1, 1, 1, 0, 0]  # predicted classes of the test data
```

MX-AI

■ Implementation of a WKNN model to classify the Iris dataset

```
# [MXML-1-04] 6.WKNN(iris).py
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load Iris dataset
x, y = load_iris(return_X_y=True)

# Split the dataset to training and test data
x_train, x_test, y_train, y_test = train_test_split(x, y)
N = x_train.shape[0]  # the number of training data points
T = x_test.shape[0]   # the number of test data points
C = np.unique(y)      # categories of y: [0, 1, 2]
K = int(np.sqrt(N))   # appropriate K value

# Z-score Normalization.
mean = x_train.mean(axis=0); std = x_train.std(axis=0)
z_train = (x_train - mean) / std
z_test = (x_test - mean) / std

# Predict the class of test data.
# 1. Compute the distance matrix between test and train data.
d_train = z_train[np.newaxis, :, :]
d_test = z_test[:, np.newaxis, :]
dist = np.sqrt(np.sum((d_train - d_test) ** 2, axis=2))
dist += 1e-8  # To prevent the distance from becoming 0
```

```
# 2. Find K nearest neighbors.
i_near = np.argsort(dist, axis=1)[:, :K]
y_near = y_train[i_near]

# 3. Compute the inverse distance
w_inv = 1. / np.array([dist[i, :][i_near[i, :]] for i in range(T)])

# 4. Predict the class of the test data using the weights of the
#    inverse distance
y_pred1 = []
for i in range(T):
    iw_dist = [w_inv[i][y_near[i] == j].sum() for j in C]
    y_pred1.append(np.argmax(iw_dist / w_inv[i].sum()))
y_pred1 = np.array(y_pred1)

# Measure the accuracy on the test data.
accuracy = (y_test == y_pred1).mean()
print('\nAccuracy on test data = {:.3f}'.format(accuracy))
```

```
Accuracy on test data = 0.974
```

MX-AI

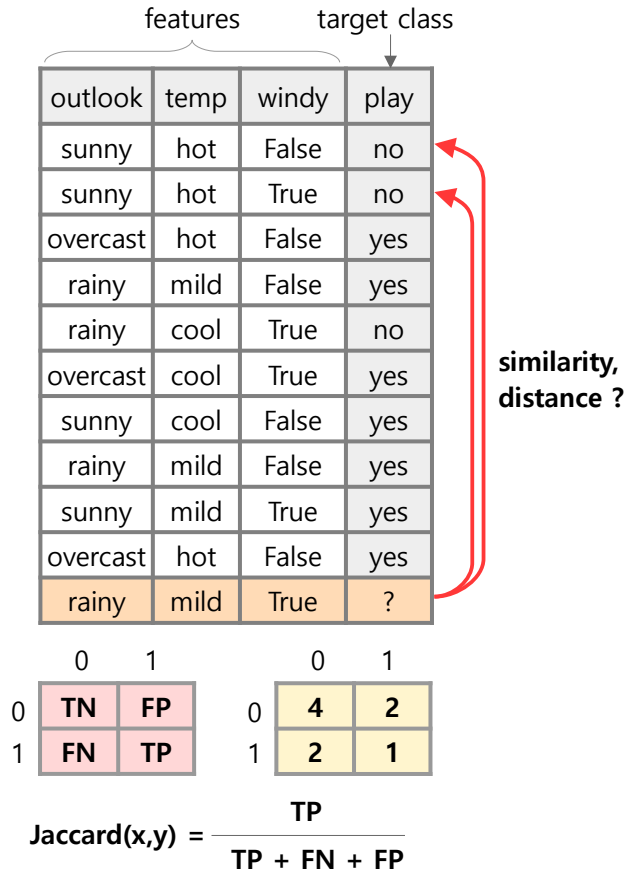
- Implementation of a WKNN model to classify the Iris dataset

```
# Compare with the results of sklearn's KNeighborsClassifier.
from sklearn.neighbors import KNeighborsClassifier

# 'distance': weight points by the inverse of their distance.
# in this case, closer neighbors of a query point will have
# a greater influence than neighbors which are further away.
knn = KNeighborsClassifier(n_neighbors=K, weights='distance')
knn.fit(z_train, y_train)
y_pred2 = knn.predict(z_test)
accuracy = (y_test == y_pred2).mean()
print('Accuracy on test data (sklearn) = {:.3f}'.format(accuracy))
```

```
Accuracy on test data (sklearn) = 0.974
```

```
print('from scratch: y_pred1\n', y_pred1)
print('from sklearn: y_pred2\n', y_pred2)
```

```
from scratch: y_pred1
[0 1 0 2 0 0 2 2 0 2 1 0 2 0 2 2 2 2 0 2 1 0 1 1 1 1 1 1 1 0 0 2 1 0 0 1 2 2]

from sklearn: y_pred2
[0 1 0 2 0 0 2 2 0 2 1 0 2 0 2 2 2 2 0 2 1 0 1 1 1 1 1 1 1 0 0 2 1 0 0 1 2 2]

The two results agree well.
```



1. K-Nearest Neighbors (KNN)

Part 5: Matching coefficient and Jaccard coefficient for categorical data

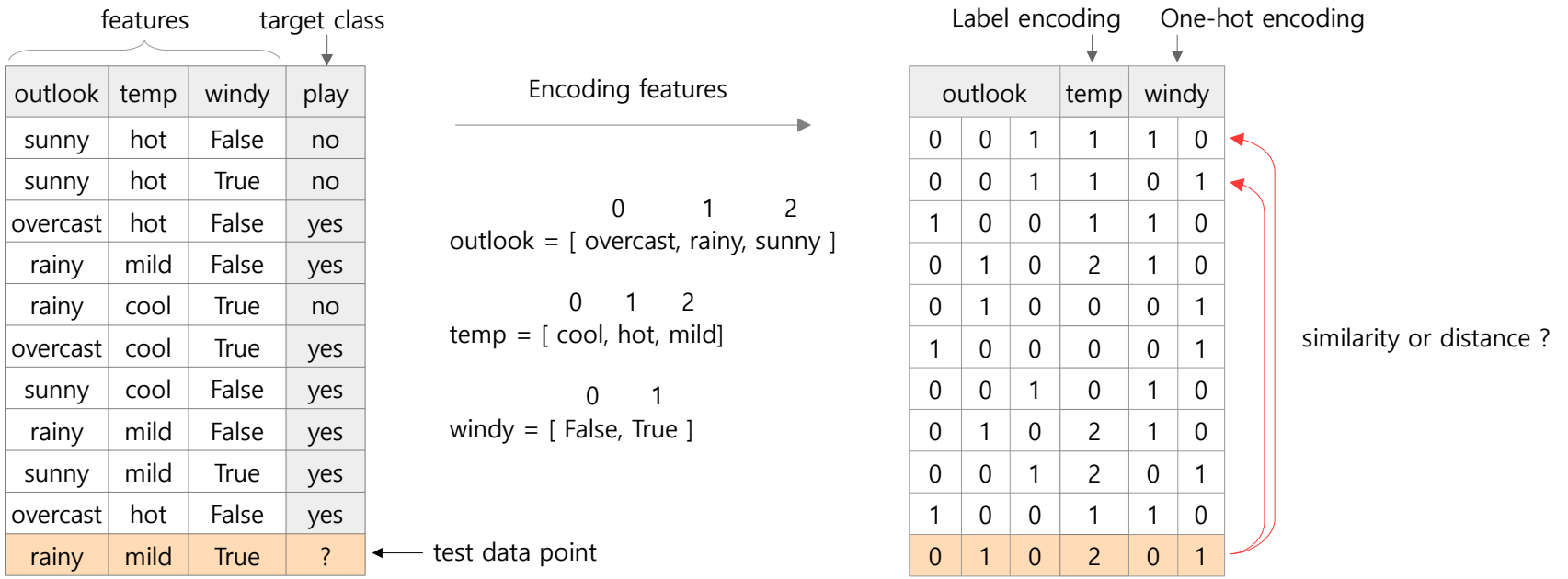
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

Categorical data

- Categorical data includes ordinal and nominal data.
- Ordinal data has a specific order or ranking. It can be sub-categorised in order, such as “low”, “medium”, “high”. The categories have a meaningful sequence, but the distances between the categories are not necessarily equal.
- Nominal data has no inherent order or ranking. The categories are just labels, like “red”, “blue” and “green”. For example, choosing "red" over "blue" does not imply that "red" is better or has a higher ranking.
- Nominal data can be converted into numeric data through one-hot encoding, and ordinal data can be converted into numeric data through label encoding.
- Metrics such as Euclidean distance are not suitable for categorical data, so alternatives such as the Jaccard index are needed.



MX-AI

■ Similarity between categorical data instances: Matching coefficient and Jaccard coefficient

features target class

	outlook	temp	windy	play
A →	sunny	hot	False	no
B →	sunny	hot	True	no
	overcast	hot	False	yes
	rainy	mild	False	yes
	rainy	cool	True	no
	overcast	cool	True	yes
	sunny	mild	True	yes
	overcast	hot	False	yes
C →	rainy	mild	True	?

one-hot encoding

test data point

features

outlook			temp			windy	
f1	f2	f3	f4	f5	f6	f7	f8
0	0	1	0	1	0	1	0
0	0	1	0	1	0	0	1
1	0	0	0	1	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	0	0	1
1	0	0	1	0	0	0	1
0	0	1	0	0	1	0	1
1	0	0	0	1	0	1	0
0	1	0	0	0	1	0	1

▪ **Simple matching coefficient**

$$MC(x,y) = \frac{\text{number of variables with matching values in x and y}}{\text{total number of variables}}$$

MC(C, A) = 2 / 8

MC(C, B) = 4 / 8

▪ **Simple matching distance**

MD = 1 - MC

MD(C, A) = 1 – 2/8 = 0.75

MD(C, B) = 1 – 4/8 = 0.50

▪ **Jaccard coefficient** ▪ **Jaccard distance** : JD = 1 - JC

$$JC(x,y) = \frac{\text{number of variables with matching 1 values in x and y}}{\text{total number of variables - number of variables with matching 0 values}}$$

JC(C, A) = 0 / (8 - 2) = 0

JC(C, B) = 1 / (8 - 3) = 0.2

JD(C, A) = 1 – 0 = 1

JD(C, B) = 1 – 0.2 = 0.8

* Using confusion matrix

x \ y	0	1
0	TN	FP
1	FN	TP

C \ A	0	1
0	2	3
1	3	0

C \ B	0	1
0	3	2
1	2	1

$$JC(x,y) = \frac{TP}{TP + FN + FP}$$

JC(C, A) = 0 / 6 = 0

JC(C, B) = 1 / 5 = 0.2

* Python code:

```
from sklearn.metrics import jaccard_score
A = [0, 0, 1,   0, 1, 0,   1, 0]
B = [0, 0, 1,   0, 1, 0,   0, 1]
C = [0, 1, 0,   0, 0, 1,   0, 1]
J_CA = jaccard_score(C, A, average='binary') # 0.0
J_CB = jaccard_score(C, B, average='binary') # 0.2
```

* MC(C, B) is larger than MC(C, A). The test data point is more similar to B than to A. * JC(C, B) is larger than JC(C, A). The test data point is more similar to B than to A.

MX-AI

■ Similarity between categorical data instances: Matching coefficient and Jaccard coefficient

features | target class

	outlook	temp	windy	play
A →	sunny	hot	False	no
B →	sunny	hot	True	no
	overcast	hot	False	yes
	rainy	mild	False	yes
	rainy	cool	True	no
	overcast	cool	True	yes
	sunny	cool	False	yes
	rainy	mild	False	yes
	sunny	mild	True	yes
	overcast	hot	False	yes
C →	rainy	mild	True	?

label encoding →

test data point

Label encoding

outlook	temp	windy
2	1	0
2	1	1
0	1	0
1	2	0
1	0	1
0	0	1
2	0	0
1	2	0
2	2	1
0	1	0
1	2	1

* Python code:

```
from sklearn.metrics import jaccard_score
A = [2, 1, 0]
B = [2, 1, 1]
C = [1, 2, 1]
J_CA = jaccard_score(C, A, labels=[0,1,2], average='macro',
                     zero_division=0.0)  # 0.0
J_CB = jaccard_score(C, B, labels=[0,1,2], average='macro',
                     zero_division=0.0)  # 0.11
```

▪ **Jaccard coefficient**

$$J(x,y) = \frac{TP}{TP + FN + FP}$$

* One-vs-rest confusion matrix for each label

A

C \	1,2	0
1,2	2	1
0	0	0

[Label 0]

A

C \	0,2	1
0,2	0	1
1	2	0

[Label 1]

A

C \	0,1	2
0,1	1	1
2	1	0

[Label 2]

$J_0(C, A) = 0 / 1 = 0$ $J_1(C, A) = 0 / 3 = 0$ $J_2(C, A) = 0 / 2 = 0$

$JC(C, A) = np.mean([J_0, J_1, J_2]) = 0$

B

C \	1,2	0
1,2	3	0
0	0	0

[Label 0]

B

C \	0,2	1
0,2	0	1
1	1	1

[Label 1]

B

C \	0,1	2
0,1	1	1
2	1	0

[Label 2]

$J_0(C, B) = 0 / 0 = 0$ $J_1(C, B) = 1 / 3 = 0.33$ $J_2(C, B) = 0 / 2 = 0$

$JC(C, B) = np.mean([J_0, J_1, J_2]) = 0.11$

* JC(C, B) is larger than JC(C, A). The test data point is more similar to B than to A.

MX-AI

- Implementation of a KNN classification model for categorical data using Jaccard coefficient.

```
# [MXML-1-05] 7.KNN(Jaccard).py
# KNN classification on categorical data
import numpy as np
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.metrics import jaccard_score

# Golf play dataset
# data source:
# https://www.kaggle.com/datasets/priy998/golf-play-dataset
# columns = [outlook, temperature, humidity, windy, play]
data = np.array(
        [['sunny',    'hot',  'high',   False, 'no'],
         ['sunny',    'hot',  'high',   True,  'no'],
         ['overcast', 'hot',  'high',   False, 'yes'],
         ['rainy',    'mild', 'high',   False, 'yes'],
         ['rainy',    'cool', 'normal', False, 'yes'],
         ['rainy',    'cool', 'normal', True,  'no'],
         ['overcast', 'cool', 'normal', True,  'yes'],
         ['sunny',    'mild', 'high',   False, 'no'],
         ['sunny',    'cool', 'normal', False, 'yes'],
         ['rainy',    'mild', 'normal', False, 'yes'],
         ['sunny',    'mild', 'normal', True,  'yes'],
         ['overcast', 'mild', 'high',   True,  'yes'],
         ['overcast', 'hot',  'normal', False, 'yes'],
         ['rainy',    'mild', 'high',   True,  'no'],
         ['sunny',    'mild', 'high',   True,  'no']])
```

```
# x: one-hot encoded or label encoded feature data
# y: target, k: the number of nearest neighbors
# average: 'binary' or 'macro'
def predict(x, y, k, average):
    match = []
    for t in range(x.shape[0]):
        x_test = x[t]
        y_test = y[t]
        x_train = np.delete(x, t, axis=0)
        y_train = np.delete(y, t, axis=0)

        # Compute the Jaccard similarity between a test data point
        # and all training data points.
        similarities = []
        for i in range(x_train.shape[0]):
            J = jaccard_score(x_train[i], x_test,
                              average=average, zero_division=0.0)
            similarities.append(J)

        # Find the k nearest neighbors of the test data point.
        j = np.argsort(similarities)[::-1][:k]

        # Predict the class of the test data point by majority vote
        y_pred = np.bincount(y_train[j]).argmax()

        # Store whether y_pred and y_test match or not.
        match.append(y_pred == y_test)

        print("True class: {}, Predicted class: {}, is match: {}"\
              .format(y_test, y_pred, match[-1]))
    return np.mean(match)  # return the accuracy
```

MX-AI

- Implementation of a KNN classification model for categorical data using Jaccard coefficient.

```
# One-hot encoding
ohe = OneHotEncoder().fit_transform(data).toarray().astype('int')
x = ohe[:, :-2]  # one-hot encoded features
y = ohe[:, -1]   # target class
K = 5            # 5 nearest neighbors

print("\n* One-hot encoding:")
acc = predict(x, y, K, average='binary')
print("Accuracy: {:.3f}".format(acc))
```

ohe:

```
[0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
[0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0],
[1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1],
[0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
[0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1],
[0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0],
[1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1],
[0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0],
...
```

```
* One-hot encoding:
True class: 0, Predicted class: 0, is match: True
True class: 0, Predicted class: 0, is match: True
True class: 1, Predicted class: 1, is match: True
True class: 1, Predicted class: 0, is match: False
True class: 1, Predicted class: 1, is match: True
True class: 0, Predicted class: 1, is match: False
True class: 1, Predicted class: 1, is match: True
True class: 0, Predicted class: 0, is match: True
True class: 1, Predicted class: 1, is match: True
True class: 1, Predicted class: 1, is match: True
True class: 1, Predicted class: 1, is match: True
True class: 1, Predicted class: 0, is match: False
True class: 1, Predicted class: 1, is match: True
True class: 0, Predicted class: 1, is match: False
True class: 0, Predicted class: 0, is match: True
Accuracy: 0.733
```

```
# Label encoding
le = []
for i in range(data.shape[1]):
    le.append(LabelEncoder().fit_transform(data[:, i]))
le = np.array(le).T

x = le[:, :-1]  # label encoded features
y = le[:, -1]   # target class

print("\n* Label encoding:")
acc = predict(x, y, K, average='macro')
print("Accuracy: {:.3f}".format(acc))
```

le:

```
[2, 1, 0, 0, 0],
[2, 1, 0, 1, 0],
[0, 1, 0, 0, 1],
[1, 2, 0, 0, 1],
[1, 0, 1, 0, 1],
[1, 0, 1, 1, 0],
[0, 0, 1, 1, 1],
[2, 2, 0, 0, 0],
...
```

```
* Label encoding:
True class: 0, Predicted class: 0, is match: True
True class: 0, Predicted class: 0, is match: True
True class: 1, Predicted class: 0, is match: False
True class: 1, Predicted class: 0, is match: False
True class: 1, Predicted class: 1, is match: True
True class: 0, Predicted class: 1, is match: False
True class: 1, Predicted class: 1, is match: True
True class: 0, Predicted class: 0, is match: True
True class: 1, Predicted class: 1, is match: True
True class: 1, Predicted class: 1, is match: True
True class: 1, Predicted class: 1, is match: True
True class: 1, Predicted class: 1, is match: True
True class: 1, Predicted class: 1, is match: True
True class: 0, Predicted class: 1, is match: False
True class: 0, Predicted class: 0, is match: True
Accuracy: 0.733
```

Frequency table : f_k

label	outlook	temp	windy
0	4	4	8
1	4	4	5
2	5	5	0

Label encoding

outlook	temp	windy
2	1	0
2	1	1
0	1	0
1	2	0
1	0	0
1	0	1
0	0	1
2	2	0
1	2	1

IOF similarity

$$if\ (x_k = y_k):$$

$$\text{sim}_k = 1$$

$$else:$$

$$\text{sim}_k = \frac{1}{1 + \log(f_k(x_k)) \cdot \log(f_k(y_k))}$$

1. K-Nearest Neighbors (KNN)

Part 6: Inverse Occurrence Frequency (IOF) similarity

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

www.youtube.com/@meanxai

MX-AI

■ Inverse Occurrence Frequency (IOF) similarity

- The Jaccard coefficient has a drawback in that it does not take into account the frequency of occurrence of each label. For example, if your data contains a lot of label 2, you will have more instances that are similar to the test data point, because they are more likely to match the label 2 of the test data point.
- To solve this problem, we can use the inverse occurrence frequency (IOF) distance, which takes into account the frequency of occurrence of each label.
- The IOF measure was originally constructed for the text mining tasks, later, it was adjusted for categorical variables. The measure assigns higher weight to mismatches on less frequent values and vice versa.

Label encoding

	outlook	temp	windy
	2	1	0
A →	2	1	1
	0	1	0
	1	2	0
	1	0	0
	1	0	1
	0	0	1
	2	2	0
	2	0	0
B →	1	2	0
	2	2	1
	0	2	1
	0	1	0
C →	1	2	1

Frequency table : f_k

label	outlook	temp	windy
0	4	4	8
1	4	4	5
2	5	5	0

* reference : Chandola, V., et, al., 2007, Similarity Measures for Categorical Data – A Comparative Study

$$sim_k = \begin{cases} 1 & if\ (x_k = y_k) \\ \dfrac{1}{1+\log(f_k(x_k))\cdot\log(f_k(y_k))} & otherwise \end{cases}$$

$$sim = \frac{1}{1+d} \qquad d = \frac{1}{sim} - 1$$

$C(outlook)=1, A(outlook)=2$ → $\dfrac{1}{1+\log(4)\times\log(5)}=0.309$

$C(temp)=2, A(temp)=1$ → $\dfrac{1}{1+\log(5)\times\log(4)}=0.309$

$C(windy)=1, A(windy)=1$ → $=1.0$

$\text{sim(C,A)} = \dfrac{0.309+0.309+1.0}{3} = 0.539$

$d(C,A) = \dfrac{1}{0.539} - 1 = 0.855$

$B(outlook)=1$ → $=1.0$

$B(temp)=2$ → $=1.0$

$B(windy)=0$ → $\dfrac{1}{1+\log(5)\times\log(8)}=0.23$

$\text{sim(C,B)} = \dfrac{1.0+1.0+0.23}{3}=0.743$

$d(C,B) = \dfrac{1}{0.743} - 1 = 0.346$

C is closer and more similar to B than to A.

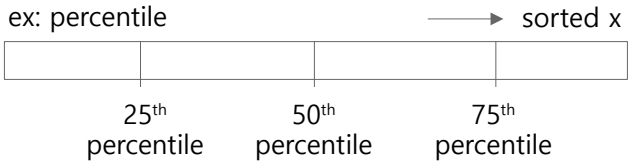
MX-AI

■ Handling mixed categorical and numerical features

Data that contains a mix of categorical and numeric features

	outlook	temp	windy	x	x′
	2	1	0	12.71	0
A →	2	1	1	13.28	1
	0	1	0	27.26	3
	1	2	0	28.46	3
	1	0	0	27.06	3
	1	0	1	18.80	2
	0	0	1	19.71	2
	2	2	0	14.92	1
	2	0	0	31.28	3
B →	1	2	0	10.08	0
	2	2	1	9.22	0
	0	2	1	23.17	2
	0	1	0	15.56	0
C →	1	2	1	30.50	1

1) Convert numerical features to categorical.




```
xp = np.zeros(x.shape)

p1, p2, p3 = np.percentile(x, q=[25,50,75])
A = np.logical_and

xp[np.where((x < p1))[0]] = 0
xp[np.where(A(x >= p1, x < p2))[0]] = 1
xp[np.where(A(x >= p2, x < p3))[0]] = 2
xp[np.where((x > p3))[0]] = 3
```

Example (2)

- Perform KNN on categorical data and find k_1 neighbors for the test data point.
- Perform KNN on numerical data and find k_2 neighbors for the test data point.
- Majority voting on the k_1 + k_2 neighbors.

2) Handle categorical and numerical features separately.

Example (1)

- For categorical features, compute the IOF distances.
- For numerical features, normalize the feature values and compute the distances such as Manhattan.
- Compute the weighted average of the two distances.

IOF distances on [outlook, temp, windy]	Manhattan distances on [x]
$d_{IOF}(C,A) = 0.855$	0.782
$d_{IOF}(C,B) = 0.346$	0.926

```
x = np.array([12.71,13.28,27.26,28.46,...])
t = np.array([30.5])
xn = (x - x.min()) / (x.max() - x.min())
tn = (t - x.min()) / (x.max() - x.min())
d = np.abs(xn - tn)
```

* Weighted average distance:

$$d(C,A) = \frac{3}{4}\times 0.855 + \frac{1}{4}\times 0.782 = 0.837$$

$$d(C,B) = \frac{3}{4}\times 0.346 + \frac{1}{4}\times 0.926 = 0.491$$

MX-AI

■ Implementation of a KNN classification model for categorical data using IOF similarity

```
# [MXML-1-06] 8.KNN(IOF).py
# KNN classification for categorical data using IOF distance
import numpy as np
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.model_selection import train_test_split

# Golf play dataset
# data source:
# https://www.kaggle.com/datasets/priy998/golf-play-dataset
# columns = [outlook, temperature, humidity, windy, play]
data = np.array(
        [['sunny',    'hot',  'high',   False, 'no'],
         ['sunny',    'hot',  'high',   True,  'no'],
         ['overcast', 'hot',  'high',   False, 'yes'],
         ['rainy',    'mild', 'high',   False, 'yes'],
         ['rainy',    'cool', 'normal', False, 'yes'],
         ['rainy',    'cool', 'normal', True,  'no'],
         ['overcast', 'cool', 'normal', True,  'yes'],
         ['sunny',    'mild', 'high',   False, 'no'],
         ['sunny',    'cool', 'normal', False, 'yes'],
         ['rainy',    'mild', 'normal', False, 'yes'],
         ['sunny',    'mild', 'normal', True,  'yes'],
         ['overcast', 'mild', 'high',   True,  'yes'],
         ['overcast', 'hot',  'normal', False, 'yes'],
         ['rainy',    'mild', 'high',   True,  'no'],
         ['sunny',    'mild', 'high',   True,  'no']])

# Compute the IOF similarity between a test data point and
# a training data point
```

```
# example:
#     i =  0  1  2  3  4  5  6  7  8  9
# train = [0, 0, 1, 0, 1, 0, 1, 0, 0, 1]
# test  = [0, 0, 1, 0, 1, 0, 1, 0, 1, 0]
#
# q_table:
#     i:   0   1   2   3   4   5   6   7   8   9
# ohe=0: [10,  9,  9, 10, 11,  7,  7,  7,  7,  7]
# ohe=1: [ 4,  5,  5,  4,  3,  7,  7,  7,  7,  7]
def iof_similarity(train, test, q_table):
    sim = np.ones(shape=(train.shape[0],))
    for i in range(train.shape[0]):
        if (train[i] != test[i]):
            log_x = np.log(q_table[train[i], i] + 1)
            log_y = np.log(q_table[test[i], i] + 1)
            sim[i] = (1. / (1. + log_x * log_y))
    return np.mean(sim)
```

```
[0, 0, 1, 0, 1, 0, 1, 0, 1, 0],   ← the test data point
[0, 0, 1, 0, 1, 0, 1, 0, 0, 1],
[1, 0, 0, 0, 1, 0, 1, 0, 1, 0],
[0, 1, 0, 0, 0, 1, 1, 0, 1, 0],
[0, 1, 0, 1, 0, 0, 0, 1, 1, 0],
[0, 1, 0, 1, 0, 0, 0, 1, 0, 1],
[1, 0, 0, 1, 0, 0, 0, 1, 0, 1],
[0, 0, 1, 0, 0, 1, 1, 0, 1, 0],   } training data points
[0, 0, 1, 1, 0, 0, 0, 1, 1, 0],
[0, 1, 0, 0, 0, 1, 0, 1, 1, 0],
[0, 0, 1, 0, 0, 1, 0, 1, 0, 1],
[1, 0, 0, 0, 0, 1, 1, 0, 0, 1],
[1, 0, 0, 0, 1, 0, 0, 1, 1, 0],
[0, 1, 0, 0, 0, 1, 1, 0, 0, 1],
[0, 0, 1, 0, 0, 1, 1, 0, 0, 1]
```

MX-AI

■ Implementation of a KNN classification model for categorical data using IOF similarity

```
# x: one-hot encoded or label encoded features
# y: target, k: the number of nearest neighbors
def predict(x, y, k):
    match = []
    for t in range(x.shape[0]):
        x_test = x[t]
        y_test = y[t]
        x_train = np.delete(x, t, axis=0)
        y_train = np.delete(y, t, axis=0)
        n = np.unique(x_train).shape[0]

        # An occurrence frequency table
        q = [np.bincount(x_train[:, i], minlength=n) \
             for i in range(x_train.shape[1])]
        q_table = np.array(q).T

        # IOF similarity
        similarities = [iof_similarity(train, x_test, q_table)\
                        for train in x_train]

        # Find the k nearest neighbors of the test data point.
        j = np.argsort(similarities)[::-1][:k]

        # Predict the class of the x_test by majority vote
        y_pred = np.bincount(y_train[j]).argmax()

        # Store whether y_pred and y_test match or not.
        match.append(y_pred == y_test)

        print("True class: {}, Predicted class: {}, match: {}"\
              .format(y_test, y_pred, match[-1]))
    return np.mean(match)  # return the accuracy
```

```
# One-hot encoding
ohe = OneHotEncoder().fit_transform(data).toarray().astype('int')
x = ohe[:, :-2]  # one-hot encoded features
y = ohe[:, -1]   # target
K = 3            # 3 nearest neighbors

print("\n* One-hot encoding:")
acc = predict(x, y, K)
print("\nAccuracy: {:.3f}".format(acc))
```

ohe:

```
[0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
[0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0],
[1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1],
[0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
[0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1],
[0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0],
[1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1],
[0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0],
...
```

```
Results:
* One-hot encoding:
True class: 0, Predicted class: 0, match: True
True class: 0, Predicted class: 0, match: True
True class: 1, Predicted class: 1, match: True
True class: 1, Predicted class: 0, match: False
True class: 1, Predicted class: 1, match: True
True class: 0, Predicted class: 1, match: False
True class: 1, Predicted class: 1, match: True
True class: 0, Predicted class: 0, match: True
True class: 1, Predicted class: 1, match: True
True class: 1, Predicted class: 1, match: True
True class: 1, Predicted class: 0, match: False
True class: 1, Predicted class: 0, match: False
True class: 1, Predicted class: 1, match: True
True class: 0, Predicted class: 1, match: False
True class: 0, Predicted class: 0, match: True

Accuracy: 0.667
```

MX-AI

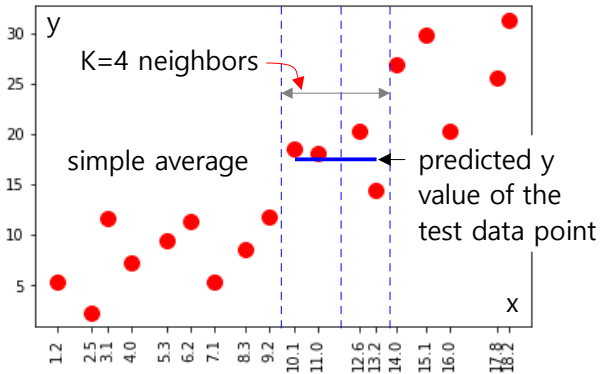
- Implementation of a KNN classification model for categorical data using IOF similarity

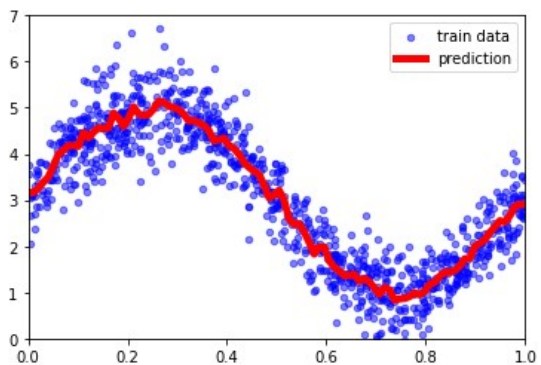
```
# Label encoding
le = []
for i in range(data.shape[1]):
    le.append(LabelEncoder().fit_transform(data[:, i]))
le = np.array(le).T

x = le[:, :-1]  # label encoded features
y = le[:, -1]   # target

print("\n* Label encoding:")
acc = predict(x, y, K)
print("\nAccuracy: {:.3f}".format(acc))
```

le:

```
[2, 1, 0, 0, 0],
[2, 1, 0, 1, 0],
[0, 1, 0, 0, 1],
[1, 2, 0, 0, 1],
[1, 0, 1, 0, 1],
[1, 0, 1, 1, 0],
[0, 0, 1, 1, 1],
[2, 2, 0, 0, 0],
...
```

```
Results:
* Label encoding:
True class: 0, Predicted class: 0, match: True
True class: 0, Predicted class: 0, match: True
True class: 1, Predicted class: 1, match: True
True class: 1, Predicted class: 0, match: False
True class: 1, Predicted class: 1, match: True
True class: 0, Predicted class: 1, match: False
True class: 1, Predicted class: 1, match: True
True class: 0, Predicted class: 0, match: True
True class: 1, Predicted class: 1, match: True
True class: 1, Predicted class: 1, match: True
True class: 1, Predicted class: 0, match: False
True class: 1, Predicted class: 0, match: False
True class: 1, Predicted class: 1, match: True
True class: 0, Predicted class: 1, match: False
True class: 0, Predicted class: 0, match: True

Accuracy: 0.667
```





1. K-Nearest Neighbors (KNN)

Part 7: KNN Regression

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

www.youtube.com/@meanxai

MX-AI

KNN Regression algorithm

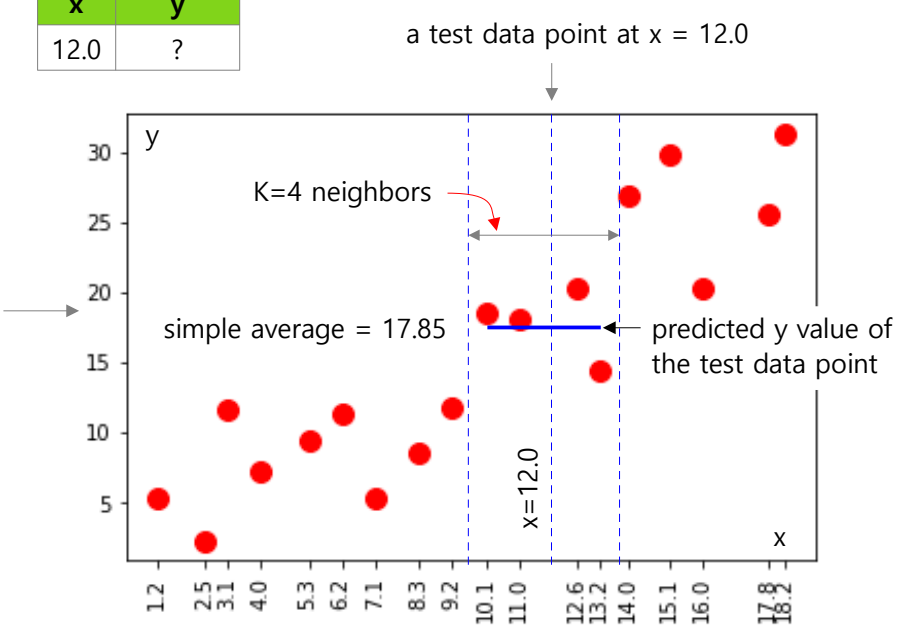
- Goal: predict the y value of a test data point using training data points.

[training data]

x	y
1.2	5.28
6.2	11.38
12.6	20.34
8.3	8.52
16.0	20.27
4.0	7.20
11.0	18.13
18.2	31.34
10.1	18.49
7.1	5.40
15.1	29.84
5.3	9.43
3.1	11.6
14.0	26.96
13.2	14.42
17.8	25.60
2.5	2.33
9.2	11.80

[a test data point]

x	y
12.0	?



1. Simple average method

$$\hat{y} = \frac{18.49+18.13+20.34+14.42}{4} = 17.85$$

2. Weighted average method

4 neighbors around x = 12.0

No	x	y	distance	weight
1	10.1	18.49	|12.0-10.1|=1.9	1 / 1.9 = 0.53
2	11.0	18.13	|12.0-11.0|=1.0	1 / 1.0 = 1.00
3	12.6	20.34	|12.0-12.6|=0.6	1 / 0.6 = 1.67
4	13.2	14.42	|12.0-13.2|=1.2	1 / 1.2 = 0.83
			sum	4.03

The shorter the distance, the greater the weight, and vice versa.

$$\hat{y} = \frac{18.49\times0.53+18.13\times1.00+20.34\times1.67+14.42\times0.83}{4.03} = 18.32$$

- If you have multiple features x, you can use the formula below to compute the distance between a test data point (x'_1, x'_2) and a training data point (x_1, x_2).

$$d = \left(|x'_1 - x_1|^p + |x'_2 - x_2|^p + \ldots \right)^{1/p}$$

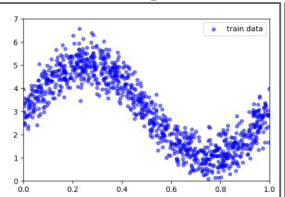
(x'_1, x'_2): a test data point
(x_1, x_2): a training data point

MX-AI

■ Implementation of a KNN regression model using the simple average method

```
[MXML-1-07] 9.KNN(regression).py
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsRegressor

# Generate training and test data
n_train = 1000  # the number of training data points
n_test = 100    # the number of test data points
x_train = np.random.random(n_train).reshape(-1, 1)
y_train = 2.0 * np.sin(2.0 * np.pi * x_train)\
            + np.random.normal(0.0, 0.5, size=(n_train,1))+3.
y_train = y_train.reshape(-1)
x_test = np.linspace(x_train.min(), x_train.max(), n_test)\
            .reshape(-1, 1)

# Generate the distance matrix between x_test and x_train
d_train = x_train[np.newaxis, :, :]
d_test = x_test[:, np.newaxis, :]
dist= np.abs(d_train - d_test).reshape(n_test, n_train)

# Find K nearest neighbors
K = 20
i_near = np.argsort(dist, axis=1)[:, :K]    # (100, 20)
y_near = y_train[i_near]                    # (100, 20)

# Predict the y values of the test data by simple average method
y_pred1 = y_near.mean(axis=1)

# Plot the training and test data points with their predicted
# y values (y_pred1)
```

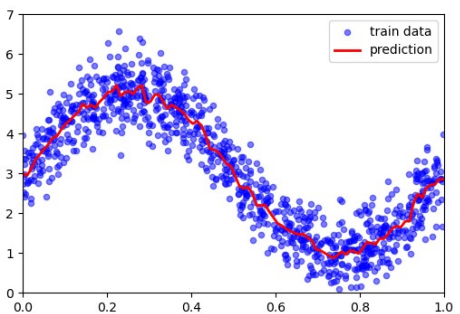
y_train



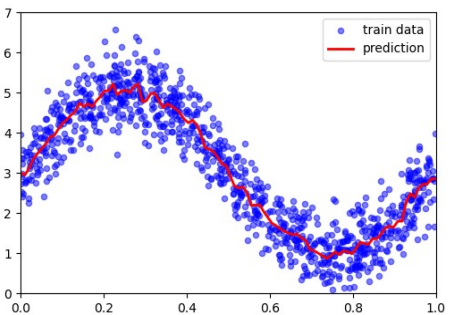
x_train

```
def plot_prediction(y_pred):
    plt.figure(figsize=(6,4))
    plt.scatter(x_train, y_train, c='blue', s=20, alpha=0.5,
                label='train data')
    plt.plot(x_test, y_pred, c='red', lw=2.0, label='prediction')
    plt.xlim(0, 1)
    plt.ylim(0, 7)
    plt.legend()
    plt.show()

plot_prediction(y_pred1)

# Predict the y values of the test data using scikit-learn's
# KNeighborsRegressor
knn = KNeighborsRegressor(n_neighbors=K)
knn.fit(x_train, y_train)
y_pred2 = knn.predict(x_test)
plot_prediction(y_pred2)
```

y_pred1



y_pred2



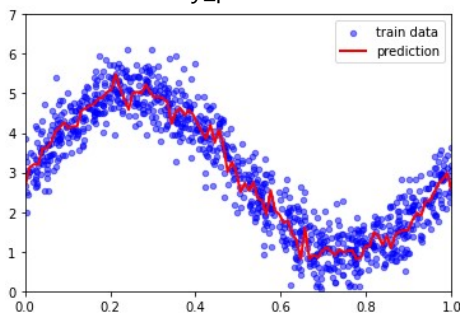
MX-AI

■ Implementation of a KNN regression model using the weighted average method

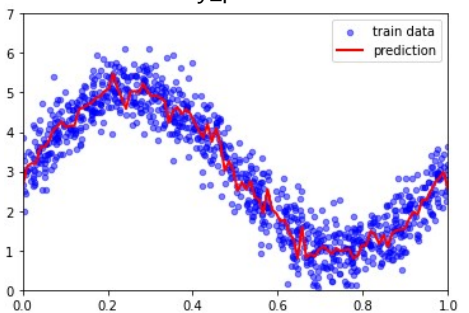
```
# [MXML-1-07] 10.WKNN(regression).py
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsRegressor

# Generate training and test data
n_train = 1000  # the number of training data points
n_test = 100    # the number of test data points
x_train = np.random.random(n_train).reshape(-1, 1)
y_train = 2.0 * np.sin(2.0 * np.pi * x_train)\
            + np.random.normal(0.0, 0.5, size=(n_train,1))+3.
y_train = y_train.reshape(-1)
x_test = np.linspace(x_train.min(), x_train.max(), n_test)\
            .reshape(-1, 1)

# Generate the distance matrix between x_test and x_train
d_train = x_train[np.newaxis, :, :]
d_test = x_test[:, np.newaxis, :]
dist= np.abs(d_train - d_test).reshape(n_test, n_train) + 1e-8

# Find K nearest neighbors
K = 200
i_near = np.argsort(dist, axis=1)[:, :K]    # (100, 200)
y_near = y_train[i_near]                    # (100, 200)

# Compute the weights to apply to the neighbors
w_dist = np.array([dist[i, :][i_near[i, :]] \
                   for i in range(x_test.shape[0])])
w_inv = 1. / w_dist

# Use the weighted average method to predict the y values of
# the test data.
y_pred1 = (y_near * w_inv).sum(axis=1) / w_inv.sum(axis=1)
```

```
# Plot the training and test data points with their predicted
# y values (y_pred)
def plot_prediction(y_pred):
    plt.figure(figsize=(6,4))
    plt.scatter(x_train, y_train, c='blue', s=20, alpha=0.5,
                label='train data')
    plt.plot(x_test, y_pred, c='red', lw=2.0, label='prediction')
    plt.xlim(0, 1)
    plt.ylim(0, 7)
    plt.legend()
    plt.show()

plot_prediction(y_pred1)

# Predict the y values of the test data using KNeighborsRegressor
# from scikit-learn.
knn = KNeighborsRegressor(n_neighbors=K, weights='distance')
knn.fit(x_train, y_train)
y_pred2 = knn.predict(x_test)
plot_prediction(y_pred2)
```

y_pred1

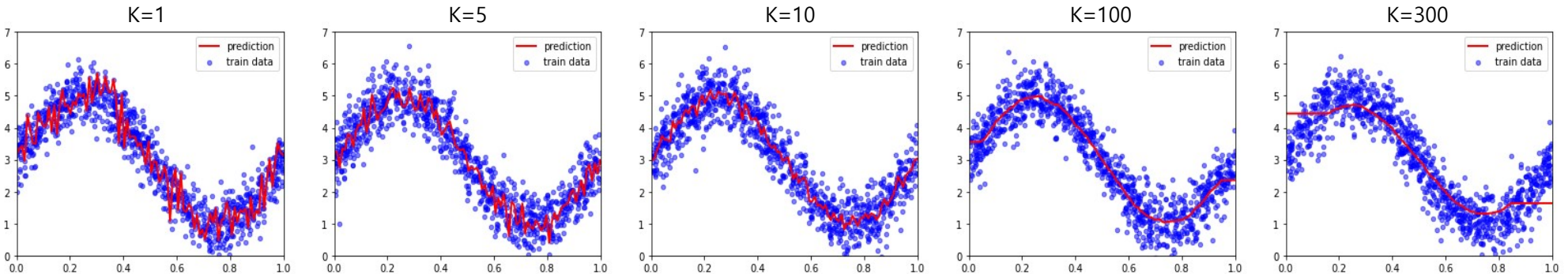


y_pred2

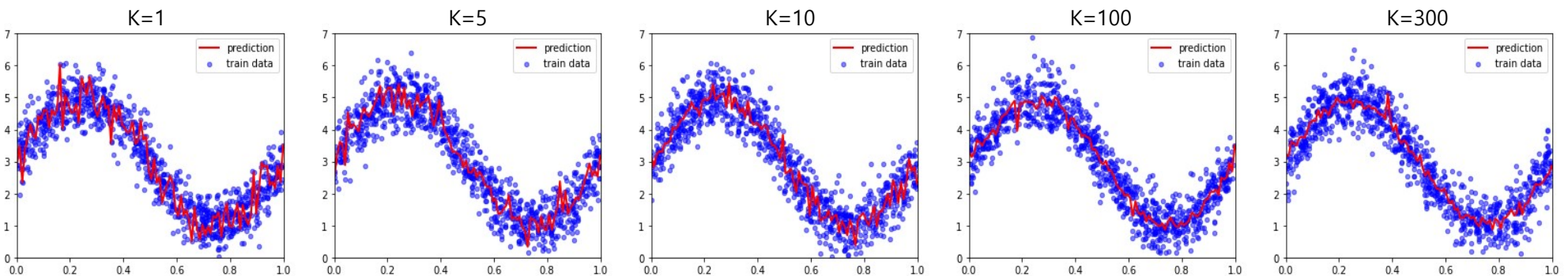


MX-AI

- Different predictions depending on the K value.
  - Simple average method: If K is too small, overfitting occurs, and if K is too large, underfitting occurs.



  - Weighted average method: This model is less sensitive to the value of K because the weights of distant neighbors are smaller. K can be set to a large value.



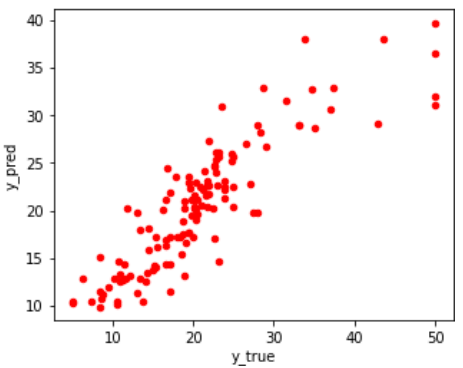
MX-AI

■ Predict the house prices in Boston using KNN regression

```
# [MXML-1-07] 11.KNN(boston).py
# Predict the house prices in Boston using KNN regression
import matplotlib.pyplot as plt
import numpy as np
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import train_test_split
import pickle

# Load 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)

# Z-score Normalization
x_mu = x_train.mean(axis=0); x_sd = x_train.std(axis=0)
y_mu = y_train.mean();       y_sd = y_train.std()
zx_train = (x_train - x_mu) / x_sd
zy_train = (y_train - y_mu) / y_sd
zx_test = (x_test - x_mu) / x_sd
zy_test = (y_test - y_mu) / y_sd

# Visually check the actual and predicted prices
def plot_predictions(y_true, y_pred):
    plt.figure(figsize=(5, 4))
    plt.scatter(y_true, y_pred, s=20, c='r')
    plt.xlabel('y_true')
    plt.ylabel('y_pred')
    plt.show()
```

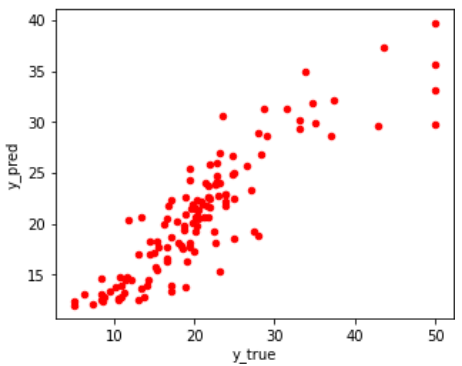
```
# Simple average method
model1 = KNeighborsRegressor(n_neighbors = 10)
model1.fit(zx_train, zy_train)
y_pred1 = model1.predict(zx_test) * y_sd + y_mu
plot_predictions(y_test, y_pred1)
print('KNN R2 = {:.3f}'.format(model1.score(zx_test, zy_test)))

# Weighted average method
model2 = KNeighborsRegressor(n_neighbors = 30, weights='distance')
model2.fit(zx_train, zy_train)
y_pred2 = model2.predict(zx_test) * y_sd + y_mu
plot_predictions(y_test, y_pred2)
print('WKNN R2 = {:.3f}'.format(model2.score(zx_test, zy_test)))
```

Simple average method



KNN R2 = 0.751

Weighted average method



WKNN R2 = 0.741

MX-AI

- Predict Boston house prices using different models and compare their prediction performance.

R^2 score for different models

	Linear Regression				K-Nearest Neighbors (KNN)	
No	OLS	TLS	LWR	RANSAC	Simple average	Weighted average
1	0.741	0.717	0.819	0.599	0.751	0.741
2	0.692	0.78	0.815	0.729	0.671	0.669
3	0.738	0.768	0.749	0.684	0.797	0.757
4	0.678	0.618	0.782	0.725	0.802	0.764
5	0.737	0.657	0.885	0.750	0.737	0.657
6	0.689	0.793	0.831	0.604	0.789	0.703
7	0.653	0.817	0.857	0.620	0.771	0.718
8	0.724	0.746	0.78	0.660	0.735	0.747
9	0.711	0.83	0.831	0.522	0.736	0.682
10	0.692	0.785	0.843	0.670	0.668	0.647
Average	0.7055	0.7511	**0.8192**	0.656	0.746	0.709