Data Mining

k-최근접 이웃 (k-Nearest Neighbor)

학습 목표

- k-최근접 이웃 알고리즘과 구현 코드를 살펴보고, Iris 데이터셋에 k-NN 알고리즘을 적용해서 분류를 수행해본다.

주요 내용

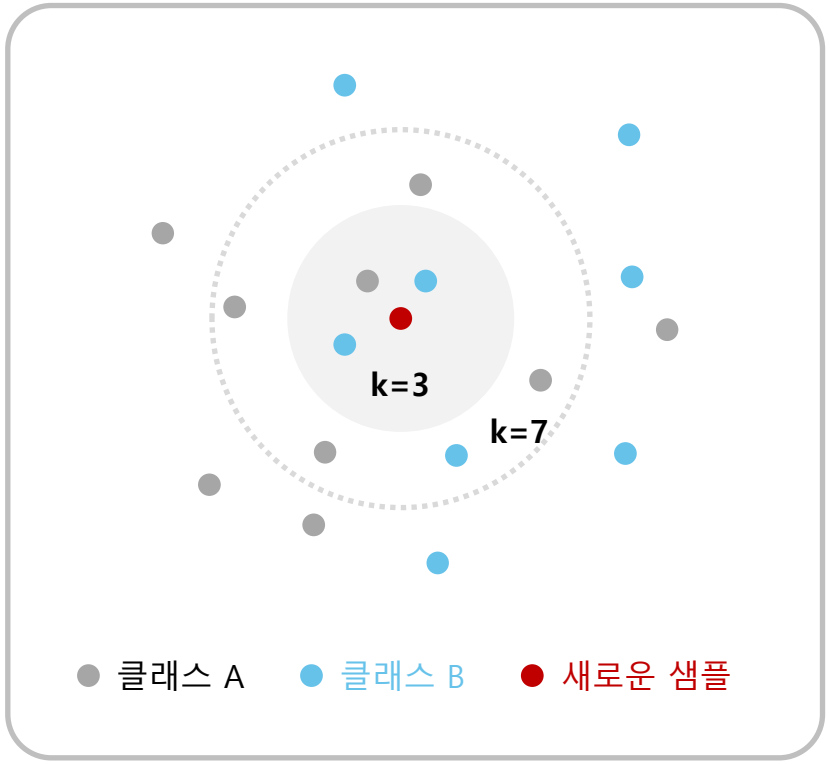
1. k-NN 알고리즘
2. k-NN 구현
3. (Key, Value) 자료 구조
4. k-NN 예시
5. 차원의 저주

1. k-NN 알고리즘

k-최근접 이웃 (k-Nearest Neighbor)

가까운 점들은 유사할 것이라는 가정 하에

예측하려는 점과 가장 가까운 k개의 이웃으로 분류 또는 회귀를 하는 모델



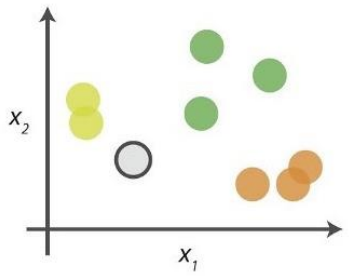
- **훈련 방식**
  - 메모리에 훈련 데이터셋을 저장
  - 별도의 모델 훈련 과정이 없음 (Lazy Learning)

- **분류** : k개 이웃에 대한 투표로 클래스 판별
  - **다수결 투표** (Majority Voting) : 빈도에 따라 클래스 판별
  - **가중 합산 투표** (Weighted Voting) : 가까운 클래스에 가중치를 부여해서 판별

- **회귀** : k개 이웃의 특징의 평균으로 값을 결정

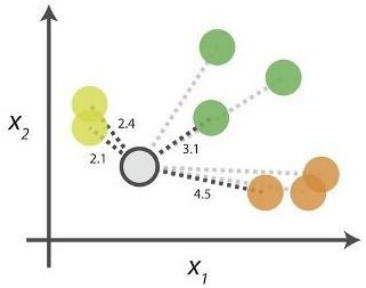
훈련 및 추론

1 훈련 데이터 저장



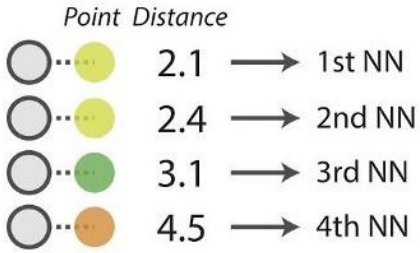
- 메모리에 훈련 데이터셋을 저장

2 거리 계산



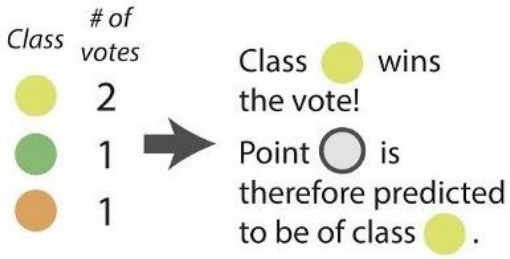
- 새로운 데이터와 훈련 데이터와의 거리 계산

3 k 이웃 선정



- 거리에 따라 정렬
- k개의 이웃을 선정

4 투표

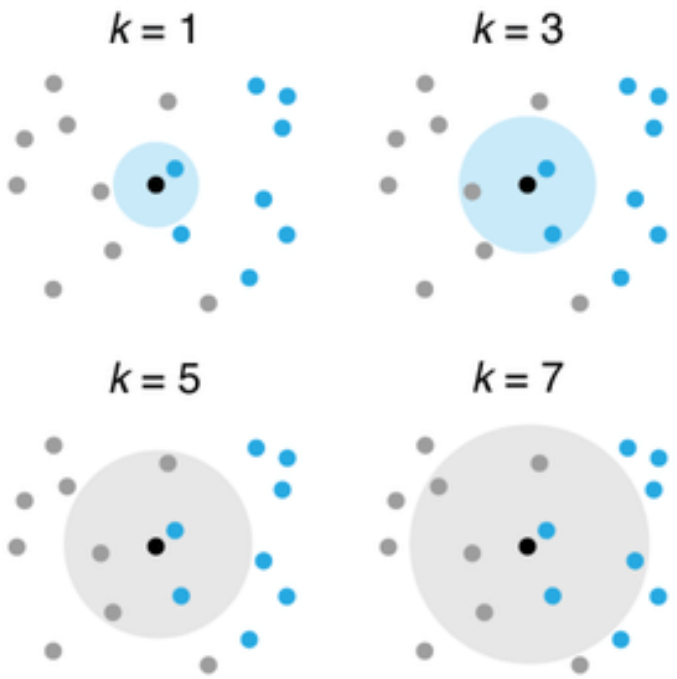


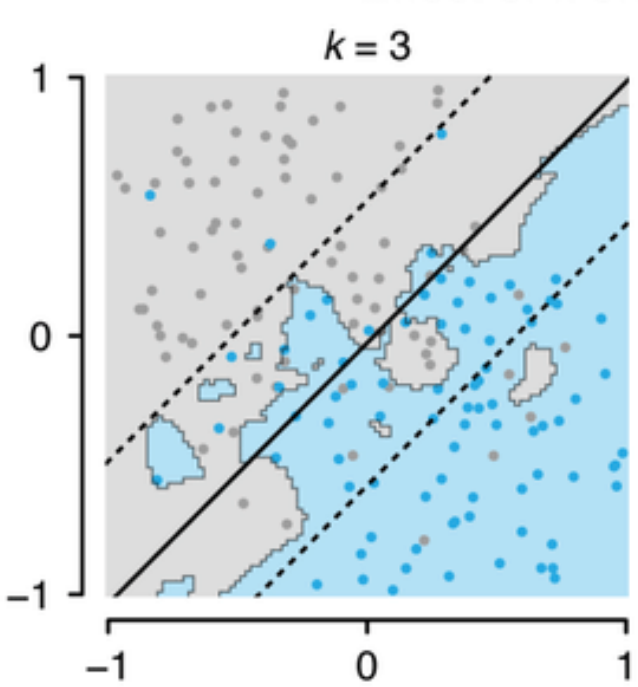
- 각 클래스 별 득표 계산
- 최대 득표를 한 클래스로 예측

그림 : https://towardsdatascience.com/how-to-find-the-optimal-value-of-k-in-knn-35d936e554eb

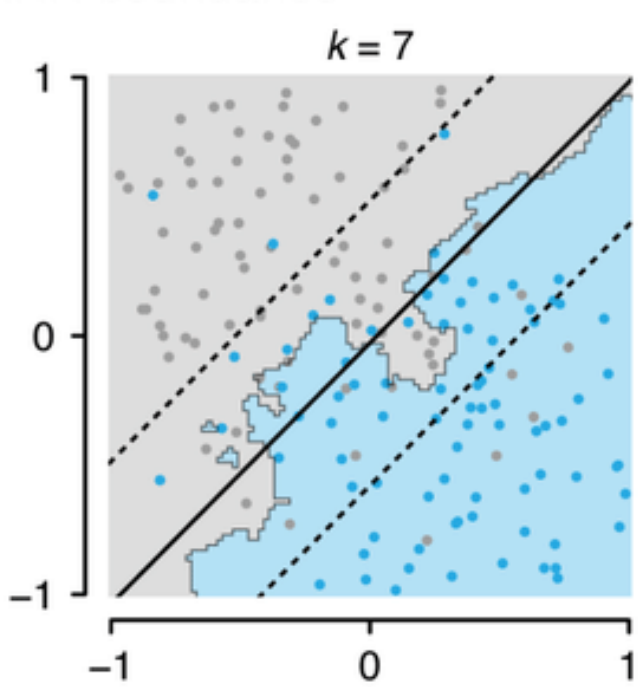
이웃의 수 k 선택

k가 작으면 경계에 잡음이 많고 k가 커지면 경계가 부정확해질 수 있다.





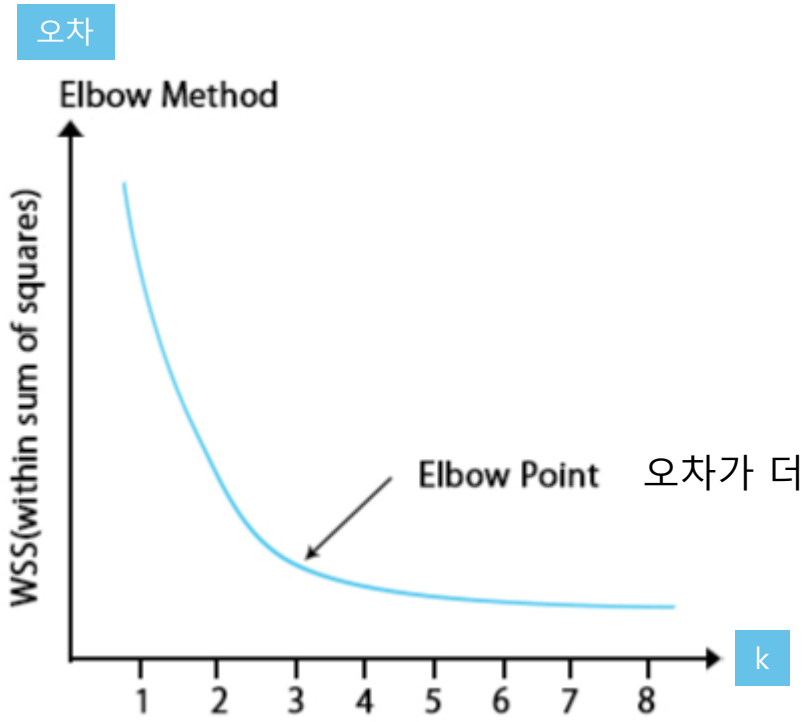
오버피팅 (overfitting)



언더피팅 (underfitting)

엘보 방법 (Elbow method)

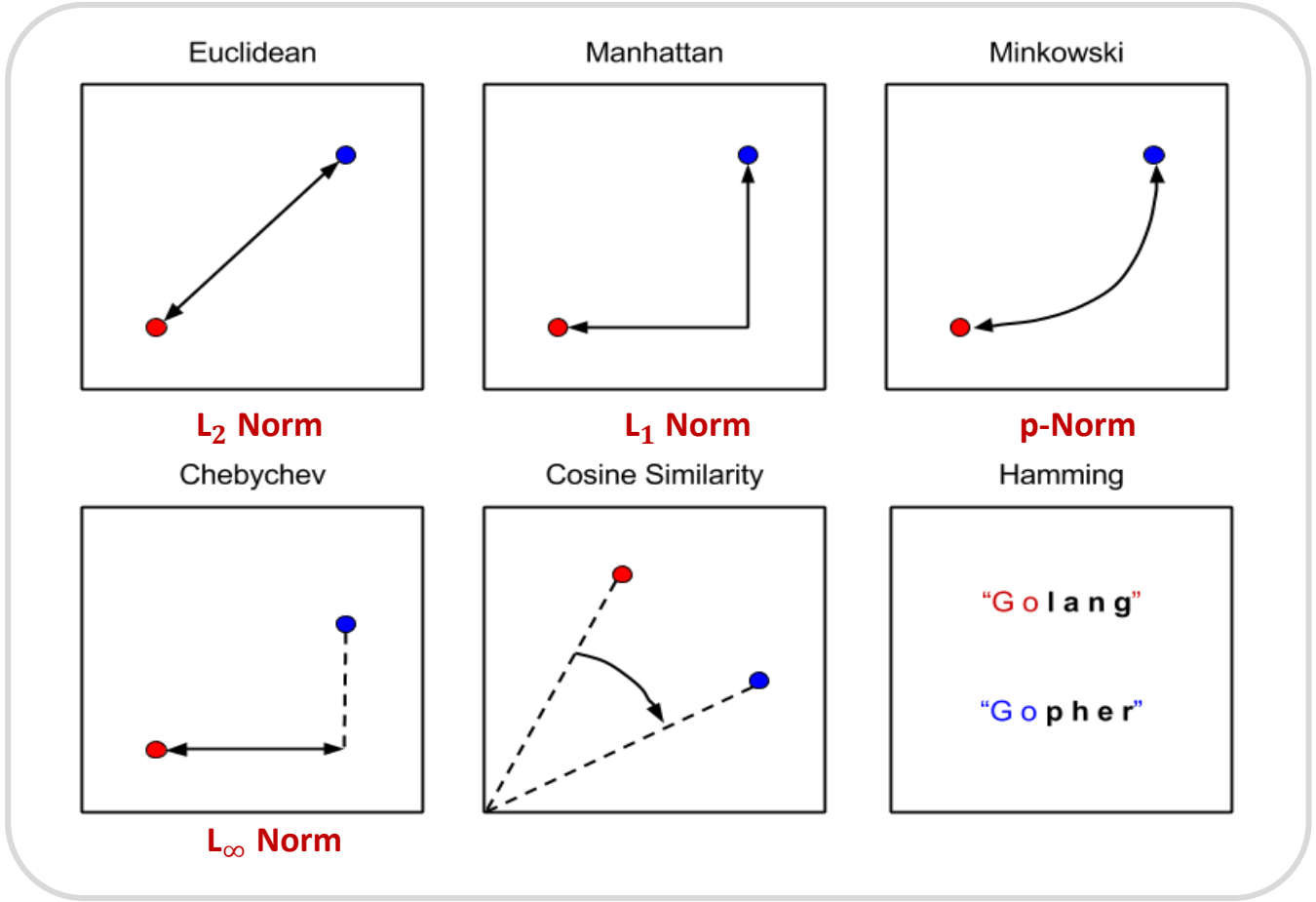
k 값에 따른 오차 그래프를 그려보고 k를 결정하는 방법



오차가 더 이상 내려가지 않는 지점

거리 함수

거리 함수의 종류



- 일반적으로 **유클리드 거리**를 사용
- 문자 분류 : 해밍 거리

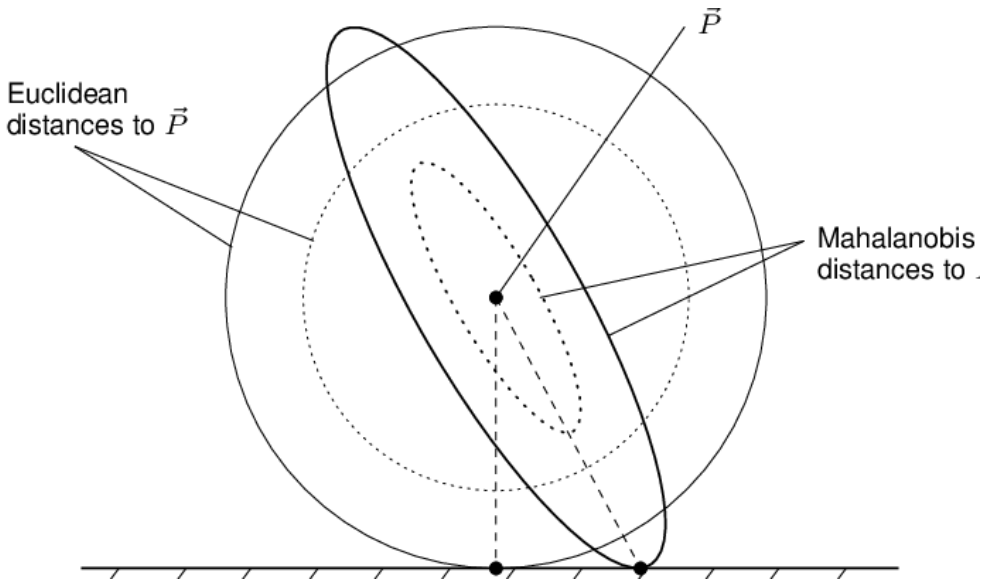
p-Norm

$$\| x \|_p = \left(\sum_{i=0}^{n} |x_i|^p \right)^{1/p} \quad \text{for } P \geq 1$$

- p=1 : L_1 Norm (Manhattan)
- p=2 : L_2 Norm (Euclidean)
- p=∞ : L_∞ Norm (Chebychev)

참고 마할라노비스 거리(Mahalanobis Distance)

마할라노비스 거리 (Mahalanobis Distance)



- 평균과의 거리가 표준편차의 몇 배인지를 나타낸 값이다.
- 즉, 데이터가 얼마나 표준 편차를 얼마나 벗어났는지를 표현

한 점 v의 거리

$$d(\boldsymbol{v}) = \sqrt{(\boldsymbol{v}-\boldsymbol{\mu})^T S^{-1}(\boldsymbol{v}-\boldsymbol{\mu})}$$

$\boldsymbol{v} = [v_1, v_2, \dots, v_N]$

평균 : $\boldsymbol{\mu} = [\mu_1, \mu_2, \dots, \mu_N]$ 공분산 : $\boldsymbol{S}$

두 점 v와 w 사이의 거리

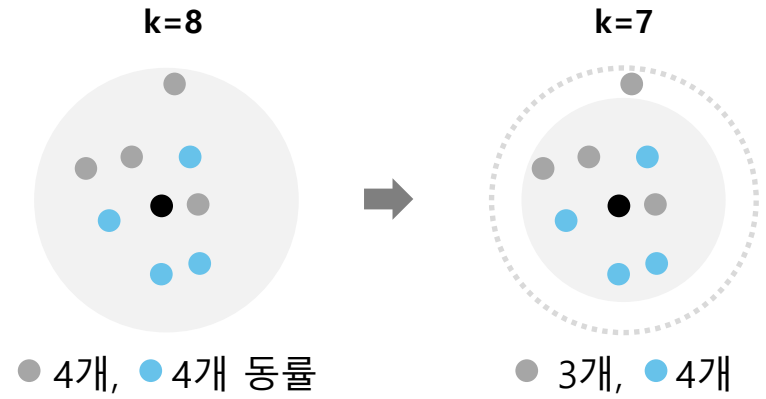
$$d(\boldsymbol{v}, \boldsymbol{w}) = \sqrt{(\boldsymbol{v}-\boldsymbol{w})^T \boldsymbol{S}^{-1}(\boldsymbol{v}-\boldsymbol{w})}$$

$v = [v_1, v_2, \dots, v_N]$, $w = [w_1, w_2, \dots, w_N]$

동률 처리

동률이 나오면 다음과 같은 방식으로 처리한다.

1. **임의로 클래스로 선택**
2. **거리가 가까운 클래스를 선택**
3. **1등이 생길 때까지 k를 줄이기**



- 최악의 경우 k=1이라면 1등이 무조건 존재하게 됨

단, 투표에서 동률이 나오지 않도록 k를 홀수로 정하는 것이 좋다.

장점과 단점

장점

- 알고리즘이 단순함
- 훈련 단계가 빠름
- 데이터 분포에 대한 가정이 없음
- 데이터의 분산까지 고려한 거리로 계산하면 모델이 좀 더 강건해짐

단점

- **차원이 높아질수록 성능이 급격히 낮아짐**
- **특징과 클래스의 관계에 대한 모형이 없으므로 성능에 제약이 있음**
- **추론 시간이 오래 걸림** : 새로운 데이터와 모든 훈련 데이터와의 거리를 매번 계산해야 함
- 데이터의 특성에 맞는 **이웃의 수 k**와 **거리 척도**를 찾아야 함

차원이 높은 경우 1) 데이터를 늘리거나 2) 특징의 개수를 줄이거나 3) 차원 축소를 해야 함

2. k-NN 구현

투표 함수 (Voting)

최대 득표 방식으로 투표를 실시

최대 득표를 한 값을 선정

```
from typing import List
from collections import Counter

def raw_majority_vote(labels: List[str]) -> str:
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner
```

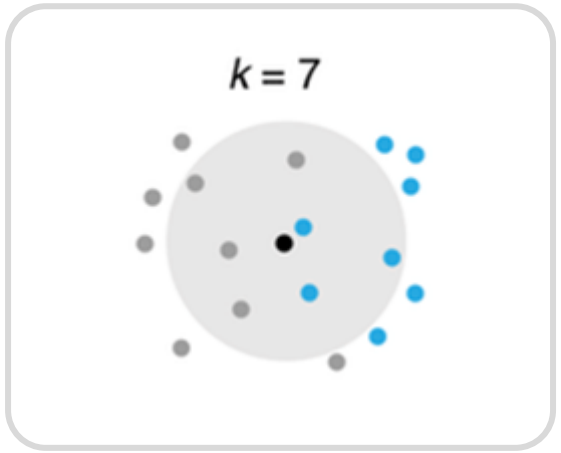
- 전달받은 레이블 리스트 중에 개수가 가장 많은 레이블을 반환

코드 검증

```
assert raw_majority_vote(['a', 'b', 'c', 'b']) == 'b'
```

- 'b ' 가 두 번 나타나므로 winner가 됨

● 5개, ● 3개이므로 ● 로 선택



동률 처리

동률이 발생하면 k를 하나씩 줄이면서 재투표

최대 득표를 한 값을 선정

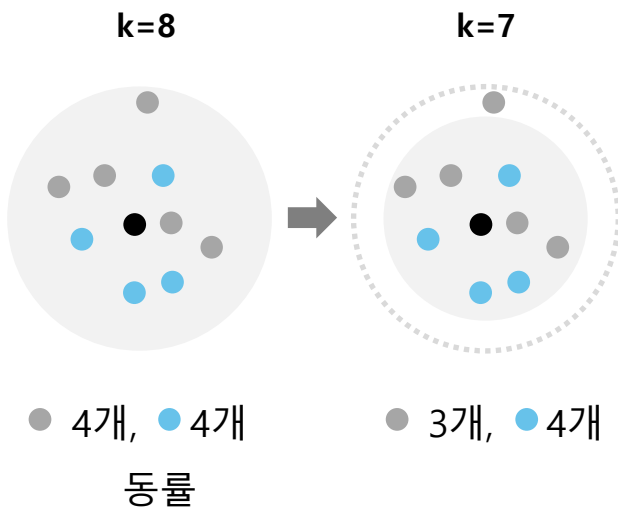
```
def majority_vote(labels: List[str]) -> str:
    """Assumes that labels are ordered from nearest to farthest."""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
```

동률 확인 및 재투표

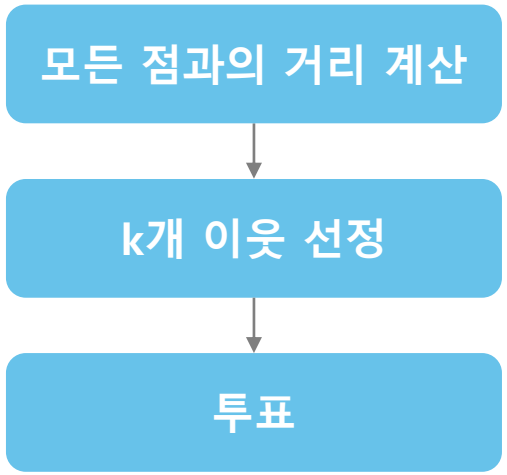
```
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])

    if num_winners == 1:
        return winner                         # unique winner, so return it
    else:
        return majority_vote(labels[:-1]) # try again without the farthest
```

- num_winners는 최대 득표를 한 레이블의 득표 목록
- Num_winnder가 1이면 winner 반환
- 아니면 동률이 발생한 것이므로 labels의 마지막 항목을 빼고 재투표



k-NN 분류기



각 점을 LabeledPoint 형으로 변환해서 knn 분류

```
from typing import NamedTuple
from scratch.linear_algebra import Vector, distance

class LabeledPoint(NamedTuple):
    point: Vector
    label: str
```

- 각 점에 레이블 정보를 추가로 정의

```
def knn_classify(k: int,
                 labeled_points: List[LabeledPoint],
                 new_point: Vector) -> str:

    # Order the labeled points from nearest to farthest.
    by_distance = sorted(labeled_points,
                         key=lambda lp: distance(lp.point, new_point))

    # Find the labels for the k closest
    k_nearest_labels = [lp.label for lp in by_distance[:k]]

    # and let them vote.
    return majority_vote(k_nearest_labels)
```

3. (Key, Value) 자료 구조

- 10장. 데이터 다루기 (Working with Data)

Dictionary 타입의 한계점

1. 메모리를 많이 사용한다.

```
import datetime

stock_price = {'closing_price': 102.06,
               'date': datetime.date(2014, 8, 29),
               'symbol': 'AAPL'}
```

- (key, value)를 동적으로 관리하기 위해서 메모리를 많이 사용한다.

2. 오류가 쉽게 생긴다.

```
# oops, typo
stock_price['cosing_price'] = 103.06
```

- key이름에 오타가 있을 경우 오류가 생기지 않고 새로운 (key, value) 쌍을 생성한다.

3. 타입 어노테이션으로 여러 타입을 갖는 딕셔너리를 표기할 수 없다.

```
prices: Dict[datetime.date, float] = {}
```

(Key, Value) 쌍의 모음을 표현할 수 있으면서
메모리 사용량이 적고 타입 어노테이션이 가능한 자료 구조는?

namedtuple 사용하기

대인 1 **Collections의 namedtuple 이용하기**

```
from collections import namedtuple
StockPrice = namedtuple('StockPrice', ['symbol', 'date', 'closing_price'])
```

typename field_names

- 튜플과 같이 immutable 하지만 각 항목에 이름이 있는 튜플
- 단, 타입 어노테이션을 할 수 없다.

NamedTuple 사용하기

대안 2 **Typing의 NamedTuple 이용하기**

```
from typing import NamedTuple

class StockPrice(NamedTuple):
    symbol: str
    date: datetime.date
    closing_price: float
```

1 NamedTuple에서 상속

각 항목에 대해서 이름과 타입을 명시 2

- 속성 별로 타입을 명시할 수 있다.
- 단, 튜플이기 때문에 속성을 수정할 수 없다.

NamedTuple 사용하기

NamedTuple 사용 예제

```
from typing import NamedTuple

class StockPrice(NamedTuple):
    symbol: str
    date: datetime.date
    closing_price: float

    def is_high_tech(self) -> bool:
        """It's a class, so we can add methods too"""
        return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']

price = StockPrice('MSFT', datetime.date(2018, 12, 14), 106.03)

assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
assert price.is_high_tech()
```

Dataclass 사용하기

대인 3 **dataclasses의 dataclass 이용하기**

```
from dataclasses import dataclass

@dataclass
class StockPrice2:
    symbol: str
    date: datetime.date
    closing_price: float
```

1 @dataclass decorator 사용

각 속성에 대해서 이름과 타입을 명시 2

- 데이터 속성들을 관리하게 쉽도록 만든 클래스
- __init__(), __repr__(), __eq__()와 같은 메서드를 자동으로 생성
- @dataclass(frozen=True)으로 속성 변경이 안되게 할 수 있음
- Python 3.7부터 도입

https://docs.python.org/3/library/dataclasses.html

Dataclass 사용하기

dataclass 사용 예제

```
from dataclasses import dataclass

@dataclass
class StockPrice2:
    symbol: str
    date: datetime.date
    closing_price: float

    def is_high_tech(self) -> bool:
        """It's a class, so we can add methods too"""
        return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']

price = StockPrice2('MSFT', datetime.date(2018, 12, 14), 106.03)
assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
assert price.is_high_tech()
```

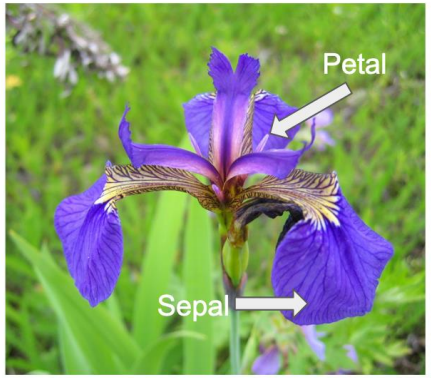
- __init__을 자동으로 생성해 주기 때문에 인스턴스 생성할 때 속성을 순서대로 전달

4. k-NN 예시 (붓꽃 종 분류)

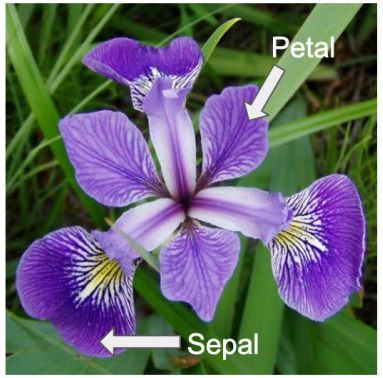
붓꽃 데이터셋 (Iris dataset)

세가지 붓꽃 종에 대한 데이터셋

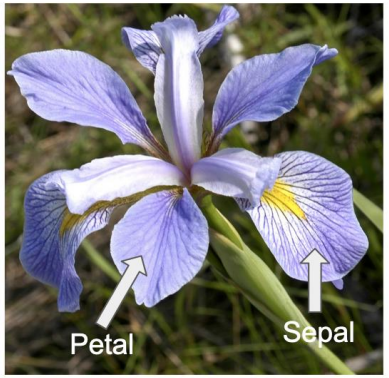
Iris setosa



Iris versicolor



Iris virginica



속성	설명	타입
sepal length (cm)	꽃받침 길이	continuous
sepal width (cm)	꽃받침 폭	continuous
petal length (cm)	꽃잎 길이	continuous
petal width (cm)	꽃잎 폭	continuous
target	붓꽃 종류 • Iris Setosa • Iris Versicolour • Iris Virginica	multi-valued discrete

- 150개 (각 종별로 50개씩)
- 1936년 영국 통계학자이자 생물학자인 도널드 피셔 (Ronald Fisher)의 논문에서 사용됨

https://archive.ics.uci.edu/ml/machine-learning-databases/iris/

붓꽃 데이터셋 (Iris dataset)

데이터 파일 :

5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa

엑셀과 같은 형태로 포맷팅

sepal_length	sepal_width	petal_length	petal_width	species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa

데이터셋 다운로드

데이터 다운로드

```
import requests

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
data = requests.get(url)

with open("iris.data", "w") as f:
    f.write(data.text)
```

- URL에서 데이터를 다운로드해서 "iris.dat" 파일에 저장

csv 파일 읽기

패키지 임포트

```
from typing import Dict
import csv
from collections import defaultdict
```

csv 파일 읽기 및 한 행 씩 파싱

```
with open('iris.data') as f:
    reader = csv.reader(f)
     iris_data = [parse_iris_row(row) for row in reader]
```

- csv 파일 읽기
- 각 row를 파싱해서 데이터들을 LabledPoint 리스트로 구성

https://docs.python.org/3/library/csv.html

데이터 파싱

입력

row =
['5.1', '3.5', '1.4', '0.2', 'Iris-setosa']

- 문자열 리스트

특징과 레이블 분리

masurement = [5.1, 3.5, 1.4, 0.2]

label = 'setosa'

- 특징 : float 로 변환
- 레이블 : “Iris-setosa”에서 앞부분 떼고 "setosa”만 남김

출력

LabeledPoint

point = [5.1, 3.5, 1.4, 0.2]

label = 'setosa'

- NamedTuple 타입

데이터 파싱

```
def parse_iris_row(row: List[str]) -> LabeledPoint:
    """
    sepal_length, sepal_width, petal_length, petal_width, class
    """
    measurements = [float(value) for value in row[:-1]]
    # class is e.g. "Iris-virginica"; we just want "virginica"
    label = row[-1].split("-")[-1]

    return LabeledPoint(measurements, label)
```

- 종 이름을 “Iris-setosa”에서 앞부분 떼고 "setosa”만 남김

종 별 딕셔너리 생성

같은 종끼리 딕셔너리에 모으기

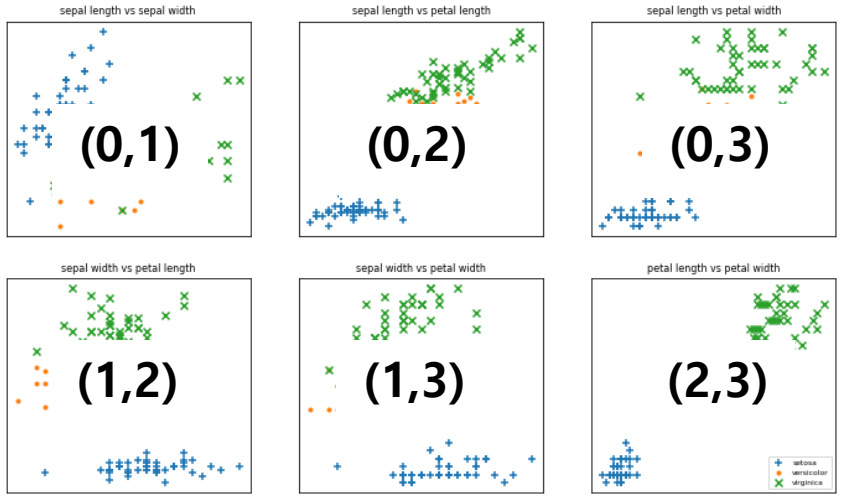
```
# We'll also group just the points by species/label so we can plot them.
points_by_species: Dict[str, List[Vector]] = defaultdict(list)
for iris in iris_data:
    points_by_species[iris.label].append(iris.point)
```

- points_by_species 딕셔너리에 같은 종 별로 데이터 벡터를 리스트 형태로 모으기

데이터 탐색

변수 간의 상관관계를 산점도로 확인

	sepal_length	sepal_width	petal_length	petal_width
sepal_length		(0,1)	(0,2)	(0,3)
sepal_width			(1,2)	(1,3)
petal_length				(2,3)
petal_width				



- Pairs에 변수 인덱스 쌍의 목록에 넣어둠
- Pairs = [(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)]

```
from matplotlib import pyplot as plt
metrics = ['sepal length', 'sepal width', 'petal length', 'petal width']
pairs = [(i, j) for i in range(4) for j in range(4) if i < j]
marks = ['+', '.', 'x']  # we have 3 classes, so 3 markers
```

- mark는 세 개의 클래스 (Setosa, Versicolour, Virginica)별로 +, ., x로 표시

데이터 탐색

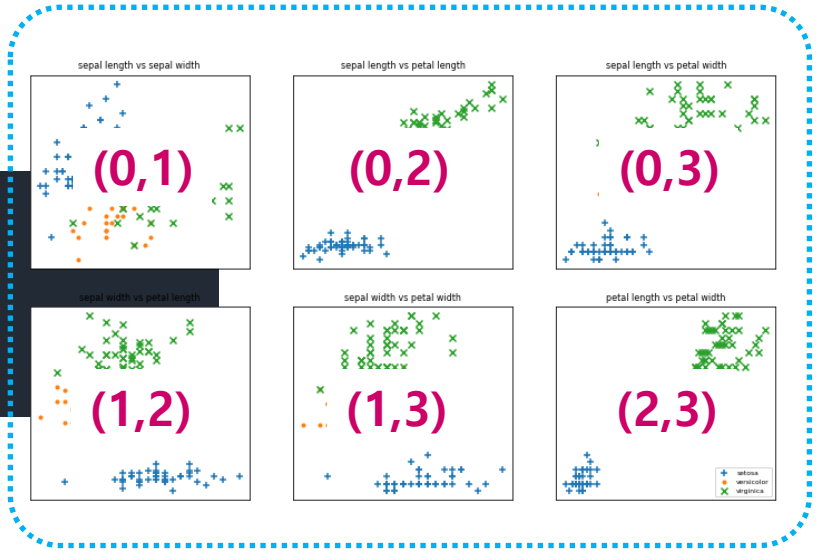
2x3 그리드에 그림 그리기 Pairs = [(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)]

```
fig, ax = plt.subplots(2, 3)

for row in range(2):
    for col in range(3):
        i, j = pairs[3 * row + col]
```

- ax 그리드의 인덱스는 (row, col)이고 metrics의 인덱스는 i와 j

ax 그리드 인덱스 (row, col)



서브 플롯 제목 달기

```
        ax[row][col].set_title(f"{metrics[i]} vs {metrics[j]}", fontsize=8)
        ax[row][col].set_xticks([])
        ax[row][col].set_yticks([])
```

각 종 별로 i번째 특징과 j번째 특징의 산포도 그리기

```
        for mark, (species, points) in zip(marks, points_by_species.items()):
            xs = [point[i] for point in points]
            ys = [point[j] for point in points]
            ax[row][col].scatter(xs, ys, marker=mark, label=species)
```

범례

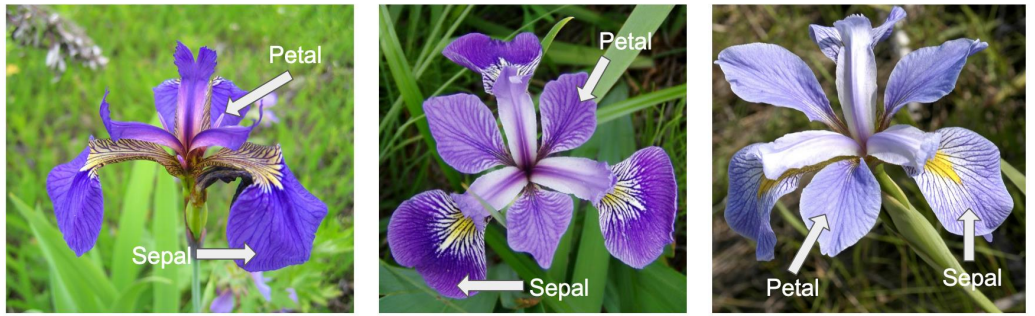
```
ax[-1][-1].legend(loc='lower right', prop={'size': 6})
plt.show()
```

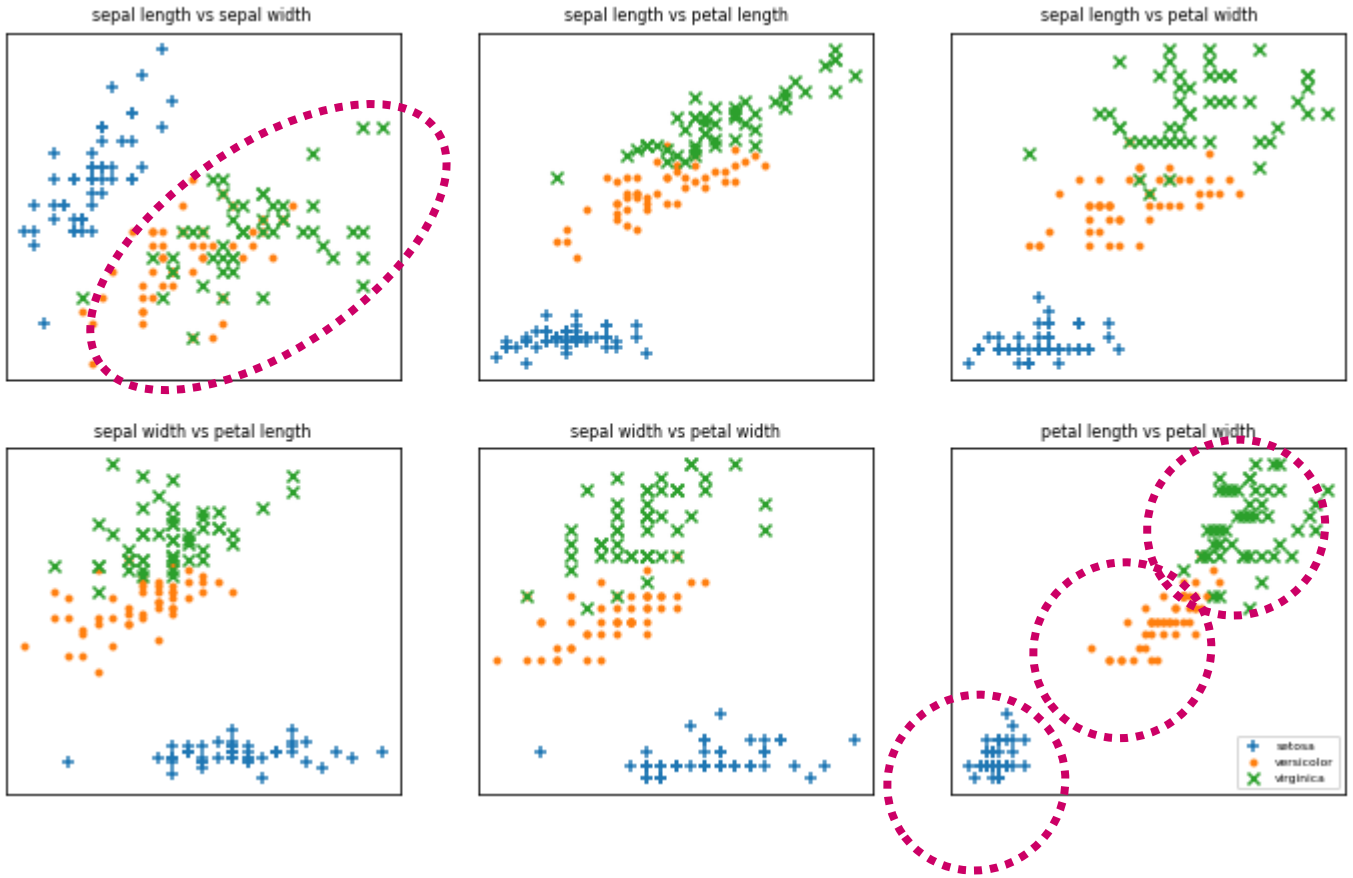
데이터 탐색

꽃받침의 길이와 너비로 'versicolor'와 'virginica' 종이 구분되지 않음

마커 : + • x

Iris setosa *Iris versicolor* *Iris virginica*





이 두 종은 꽃받침이 비슷하다.

꽃잎의 길이와 너비로 종별 구분이 잘됨

데이터셋 분리

데이터셋을 학습 데이터와 평가 데이터로 나눈다.

데이터셋 분리

```
import random
from scratch.machine_learning import split_data

random.seed(12)
iris_train, iris_test = split_data(iris_data, 0.70)
```

- 70%는 훈련 데이터셋으로, 30% 테스트 데이터셋으로 나눈다.

150 샘플 중 105개는 훈련 데이터로, 45개는 테스트 데이터로 나뉨

예측

테스트 및 성능 측정

```
from typing import Tuple

# track how many times we see (predicted, actual)
confusion_matrix: Dict[Tuple[str, str], int] = defaultdict(int)
num_correct = 0

for iris in iris_test:
    predicted = knn_classify(5, iris_train, iris.point)
    actual = iris.label

    if predicted == actual:
        num_correct += 1

    confusion_matrix[(predicted, actual)] += 1

pct_correct = num_correct / len(iris_test)
print(pct_correct, confusion_matrix)
```

- 테스트 데이터를 하나씩 분류
- 예측 값과 실제 타깃을 비교해서 정확도를 계산
- 혼동 행렬을 만들어 나감

예측

정확도 97.7%가 나옴

0.9777777777777777

defaultdict(<class 'int'>,

{('setosa', 'setosa'): 13,

('versicolor', 'versicolor'): 15,

('virginica', 'virginica'): 16,

('virginica', 'versicolor'): 1})

혼동 행렬 (confusion matrix)

실제 클래스
(Actual Class)

	setosa	**versicolor**	**virginica**
setosa	13		
versicolor		15	
virginica		1	16

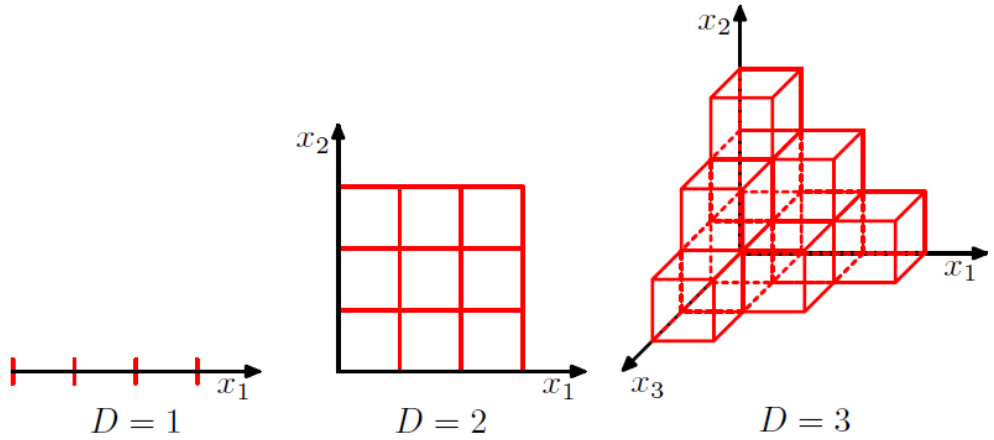
예측 클래스
(Predictive Class)

- **테스트 데이터 총 45건 중 1건만 오류가 있고 나머지는 정확하게 예측**

5. 차원의 저주

차원의 저주 (Curse of dimensionality)

차원이 증가할수록 상황을 설명하는데 필요한 인스턴스가 기하급수적으로 증가하는 현상



각 차원 별로 등간격의 격자를 만들었을 때 기하 급수적으로 증가

차원이 높아질수록 데이터 사이의 거리가 멀어지고 빈공간이 증가하면서 데이터가 희소해지는 현상이 생김

차원과 공간의 밀도

데이터의 특징이 많아질수록 차원이 높아지고,

차원이 높아지면 공간의 밀도가 낮아져 데이터가 희소해짐

1차원에서의 밀도

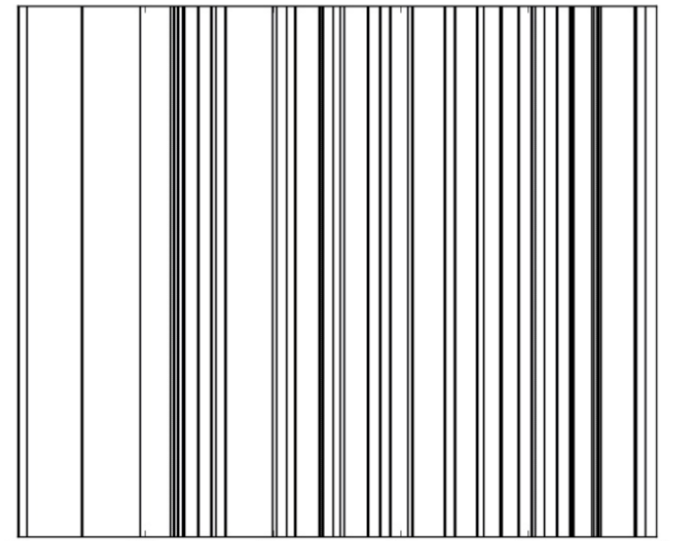


Figure 12-7. Fifty random points in one dimension

- [0,1] 구간에서의 50개 점

1차원에서의 밀도

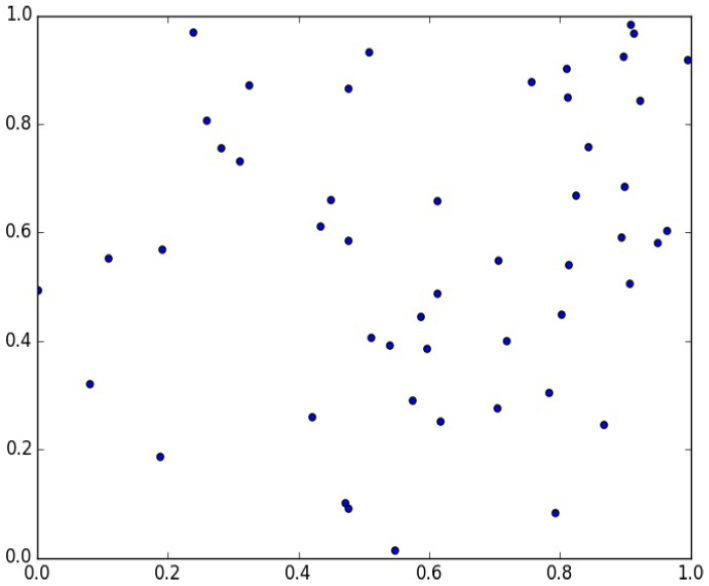


Figure 12-8. Fifty random points in two dimensions

- [0,1, 0,1] 구간에서의 50개 점

3차원에서의 밀도

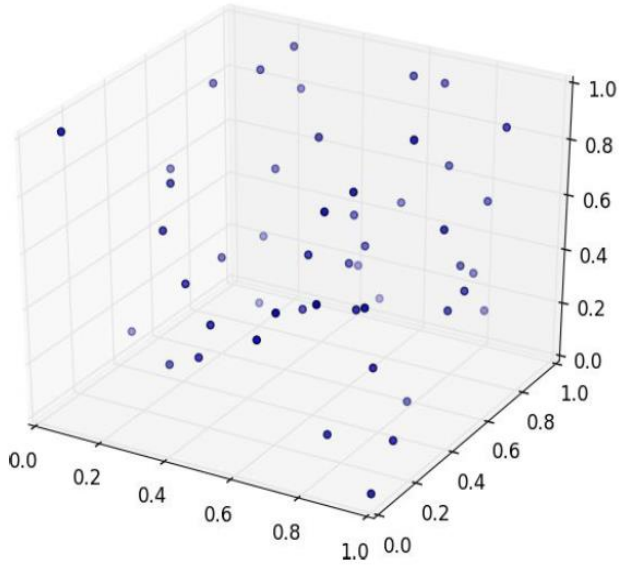


Figure 12-9. Fifty random points in three dimensions

- [0,1, 0,1, 0,1] 구간에서의 50개 점

차원의 저주

차원이 높아질수록 거리가 멀어지는 현상을 확인해보자.

지정된 차원에서 임의의 점 생성

```
import random

def random_point(dim: int) -> Vector:
    return [random.random() for _ in range(dim)]
```

지정된 차원에서 임의의 두 점 간의 거리 측정

```
def random_distances(dim: int, num_pairs: int) -> List[float]:
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

차원의 저주

1차원에서 100차원 생성

```
import tqdm
dimensions = range(1, 101)

avg_distances = []
min_distances = []
```

각 차원 별 평균 거리와 최소 거리 측정

```
random.seed(0)
for dim in tqdm.tqdm(dimensions, desc="Curse of Dimensionality"):
    distances = random_distances(dim, 10000)      # 10,000 random pairs
    avg_distances.append(sum(distances) / 10000)  # track the average
    min_distances.append(min(distances))          # track the minimum
```

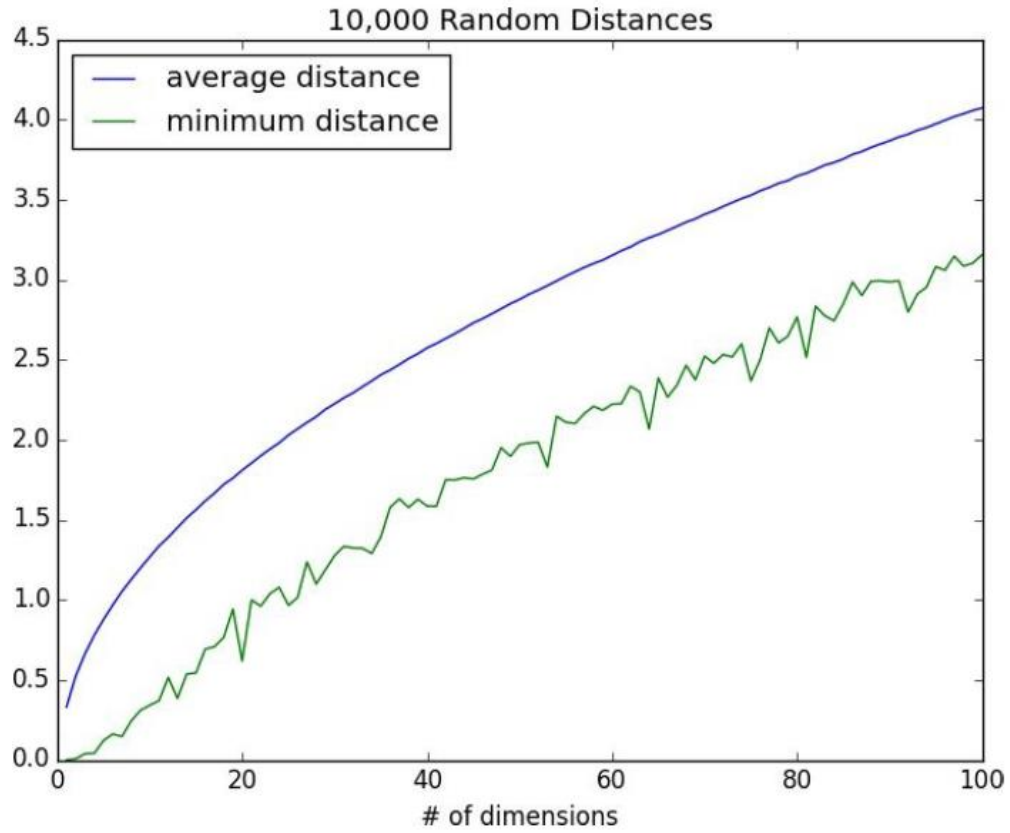
- 각 차원 별로 10,000개의 점에 대해 평균 거리와 최소 거리를 측정

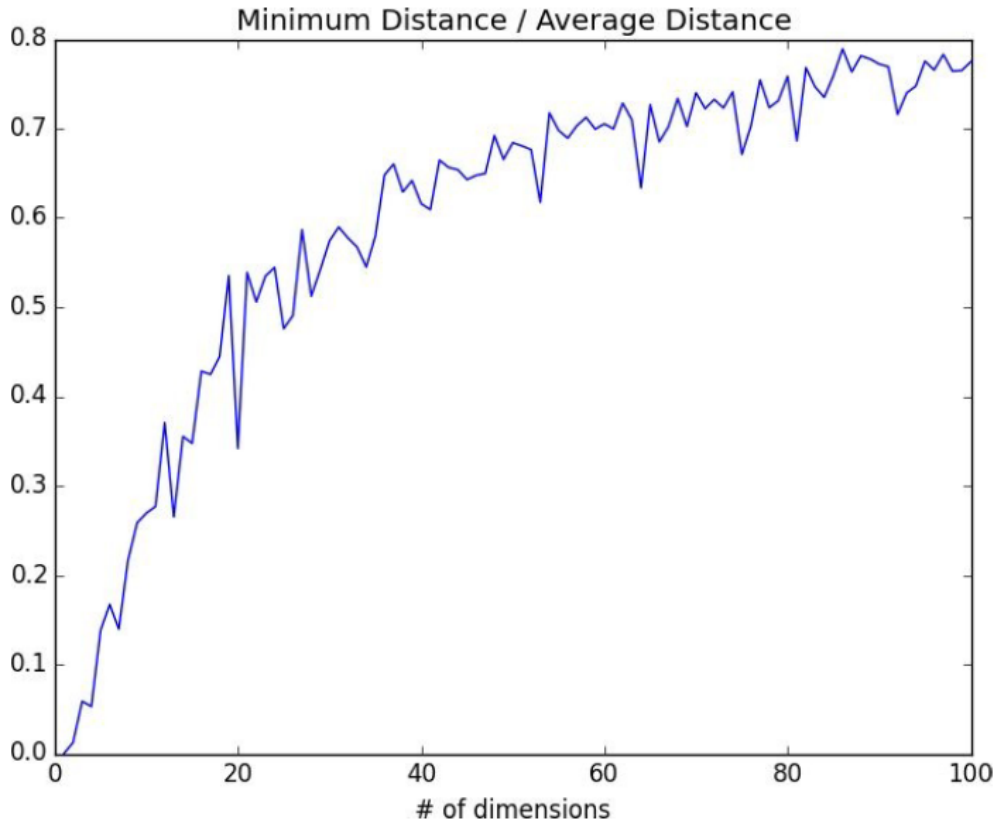
최소거리/평균거리 비율

```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances, avg_distances)]
```

차원의 저주

차원이 높아질수록 두 점 사이의 거리가 멀어지고 분산도 커진다.





Thank you!