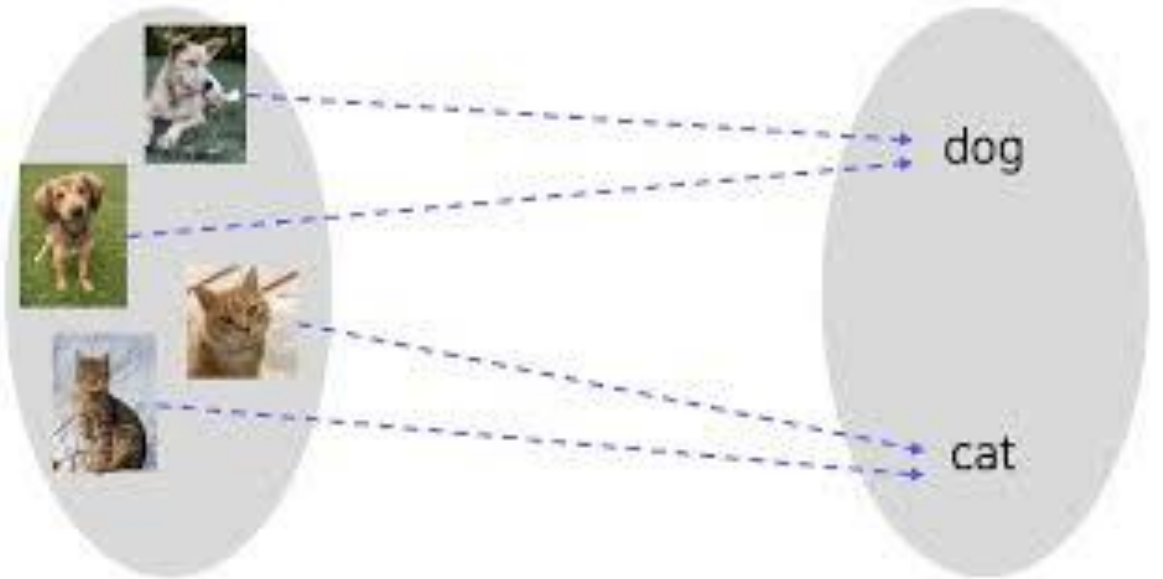
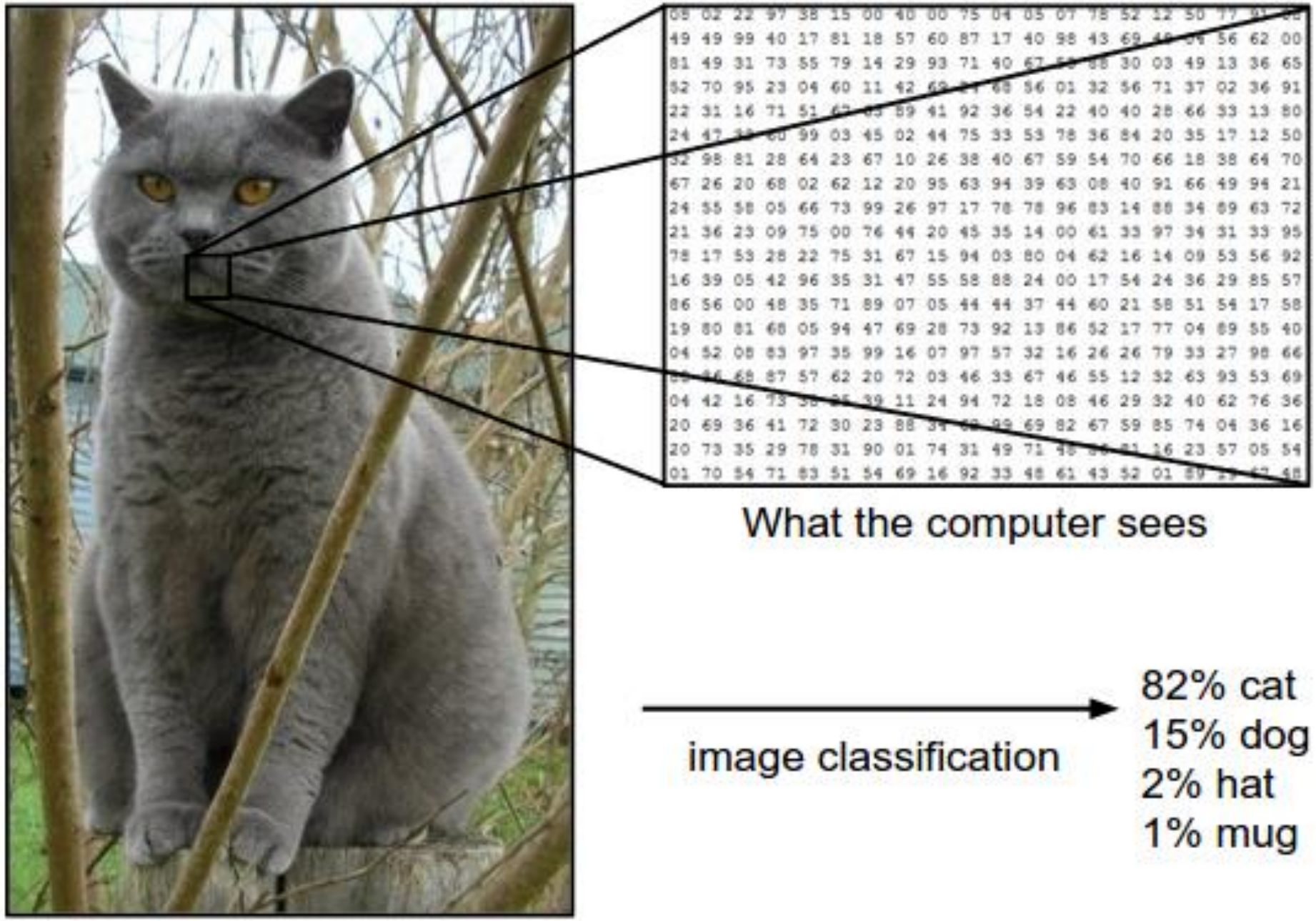
2강: 이미지 분류

이미지 분류

- 카테고리
  - 분류 문제에 있어서 데이터가 속할 수 있는 클래스
  - 카테고리의 예
    - 고양이, 개 등
- 이미지 분류란?
  - 주어진 이미지가 어떤 카테고리에 속하는지 판단하는 문제
  - 각 입력 이미지에 고정된 카테고리 집합 중 하나의 레이블을 할당

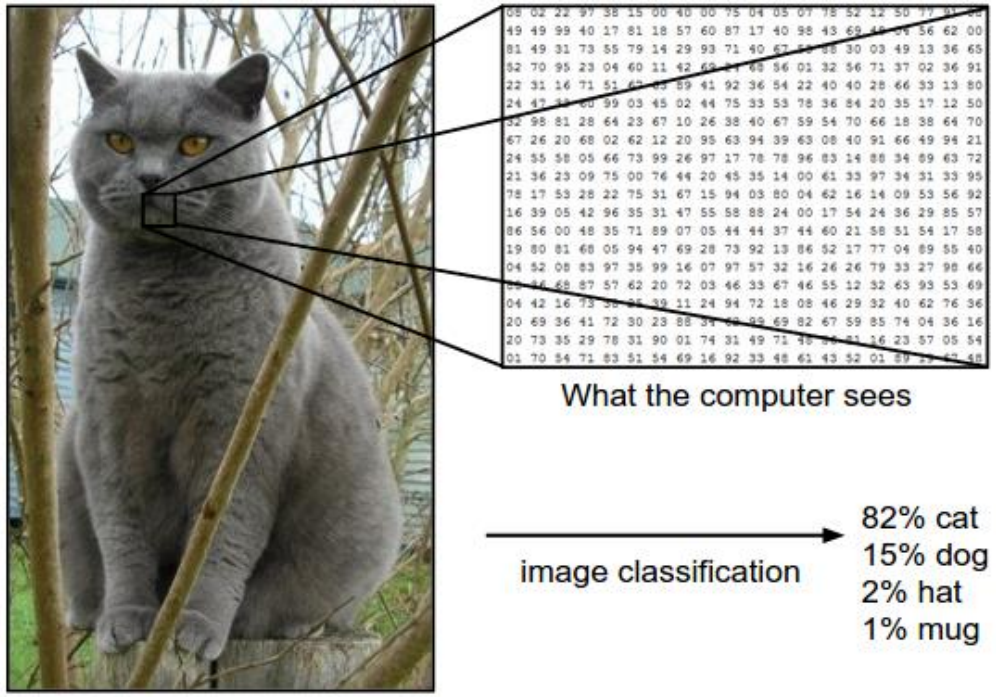


이미지 분류


What the computer sees
image classification
82% cat
15% dog
2% hat
1% mug

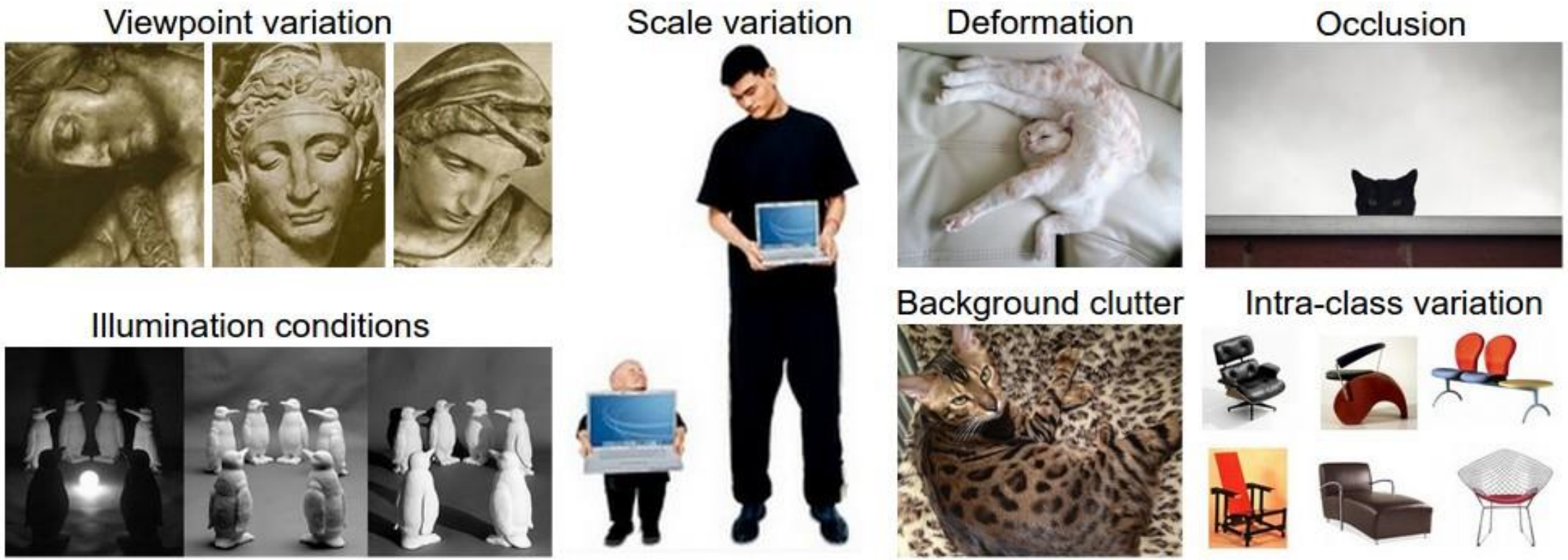
이미지는 3차원 숫자 배열

- 고양이 이미지는 너비가 248 픽셀, 높이가 400 픽셀
- 세 가지 색상 채널(빨강, 초록, 파랑)
- 이 이미지는 총 248x400x3=297,600 개의 숫자 배열
- 각 숫자는 0에서 255까지의 정수
- 이미지 분류: 이 29만개의 숫자를 "고양이"와 같은 레이블로 변환하는 작업



이미지 분류 알고리즘 구현의 어려움

- 시점 변화, 크기 변동, 변형, 가림, 조명 조건, 배경 혼잡, 클래스 내 다양성

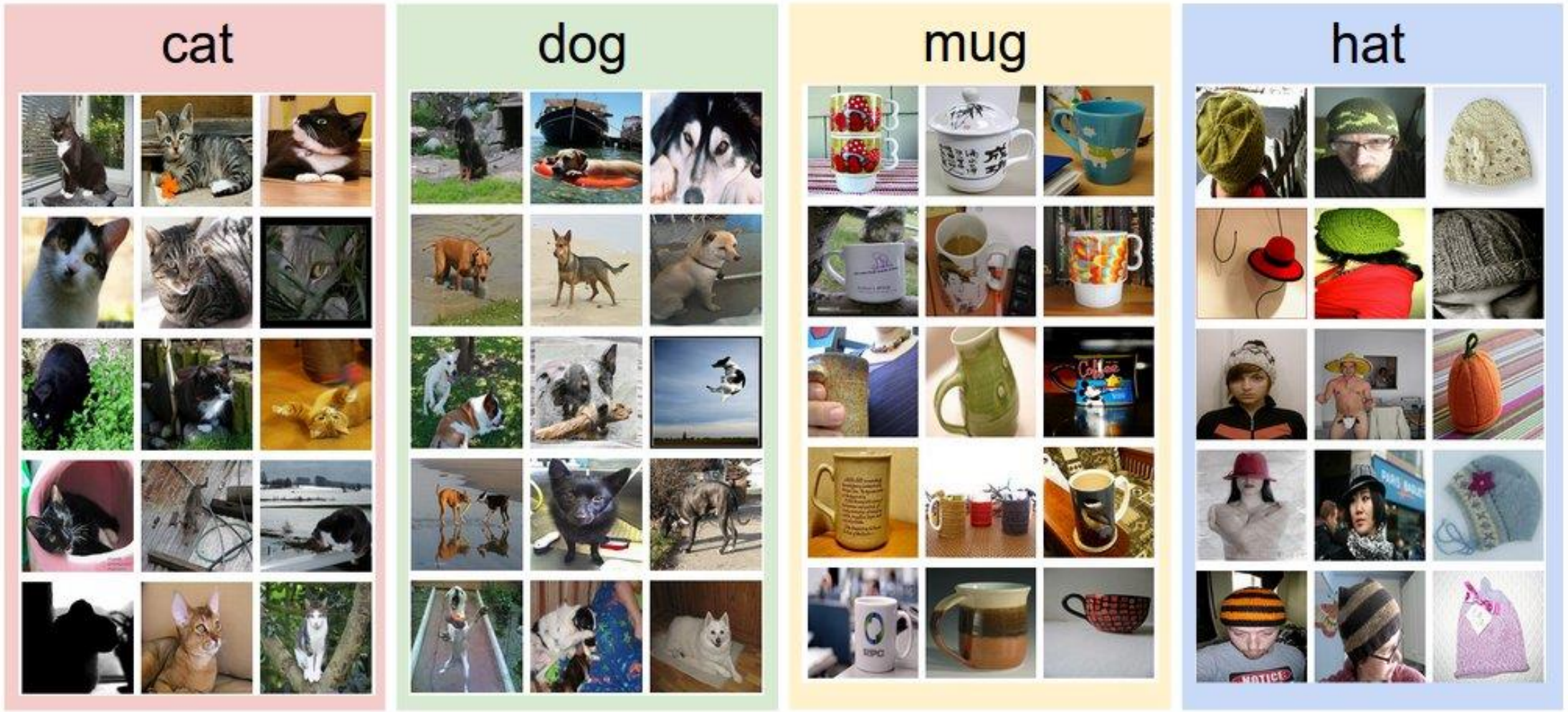


데이터-중심 접근법

- 명시적인 분류 코드 작성 대신 데이터-중심 접근법을 사용
- 데이터-중심 접근법
  - 컴퓨터에 각 클래스의 많은 예시들을 제공
  - 예시들을 보고 각 클래스의 시각적 모양에 대해 학습
  - 학습한 결과를 바탕으로 분류 수행

데이터-중심 접근법

- 네 가지 시각적 카테고리를 위한 훈련 세트



이미지 분류 파이프라인

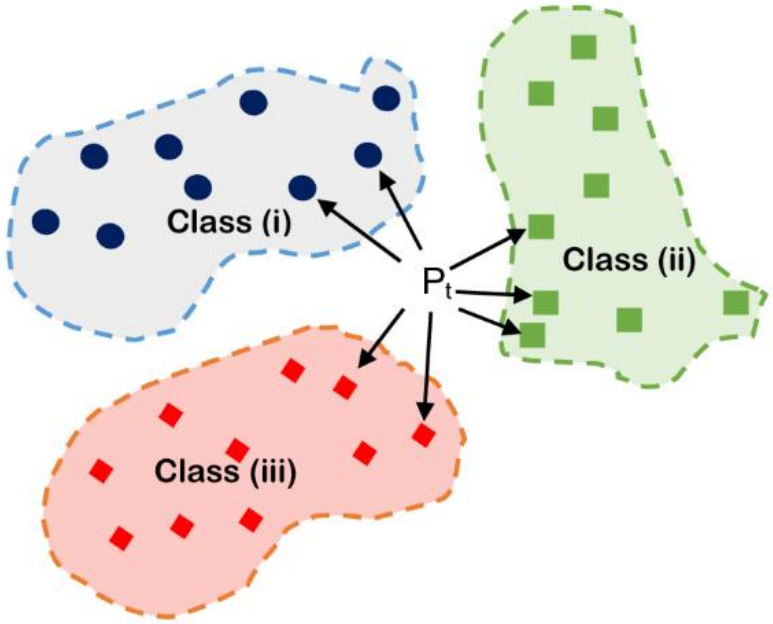
입력 K개의 클래스 중 하나로 레이블이 지정된 N개의 이미지 집합 (훈련 세트)를 입력

훈련 훈련 세트를 사용하여 각 클래스가 어떻게 보이는지 학습

평가 이전에 본적 없는 새로운 이미지들의 레이블을 예측함. 그 후 실제 정답 레이블과 예측 레이블을 비교하여 분류기의 품질을 평가

최근접 이웃 분류기

- 최근접 이웃 분류기(nearest neighbor classifier)로 이미지 분류를 시도
- 이 분류기는 실제로는 거의 사용되지 않지만 이미지 분류에 대한 기본 아이디어를 제공



CIFAR-10 데이터셋

- CIFAR-10은 대표적인 예제 이미지 분류 데이터셋
- 32 픽셀 높이와 너비, 60,000개의 작은 이미지로 구성됨
- 각 이미지는 10개의 클래스 중 하나로 레이블이 지정됨
- 60,000개의 이미지는 50,000개의 훈련 이미지와 10,000개의 테스트 이미지로 나누어짐

CIFAR-10 데이터셋

- 각 클래스의 랜덤 10개 예제 이미지

최근접 이웃 분류기

- 50,000개의 훈련 세트를 사용하여 10,000개의 테스트 이미지의 레이블을 예측하고자 함
- 최근접 이웃 분류기는 테스트 이미지를 훈련 이미지(총 50,000개) 각각과 비교한 후 가장 가까운 훈련 이미지의 레이블로 예측

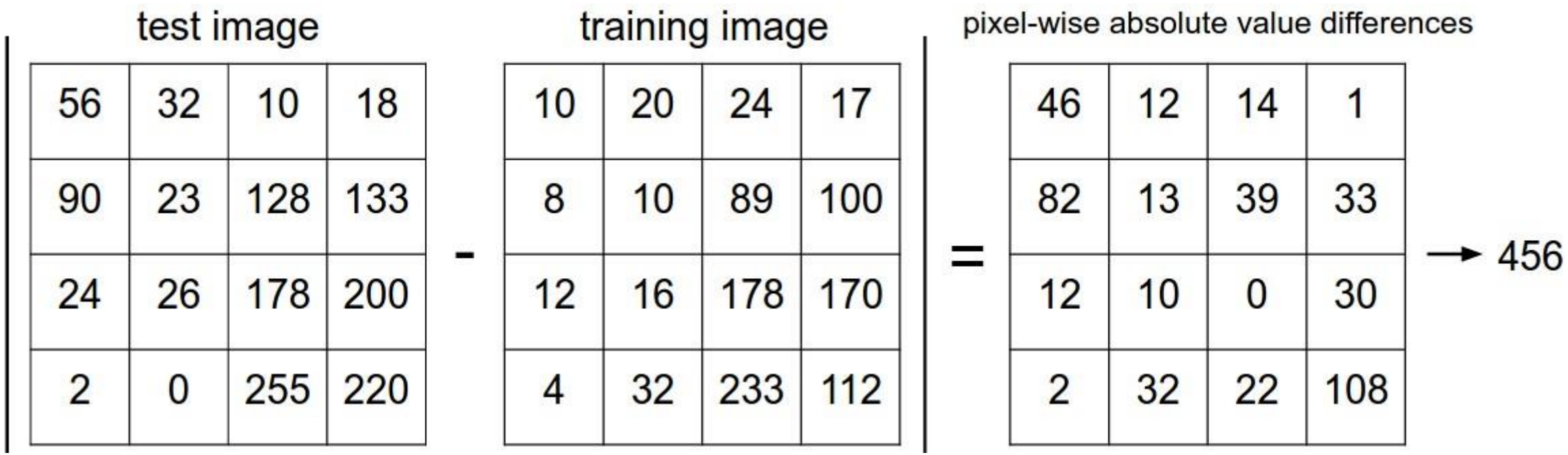
두 이미지의 비교란?

- CIFAR-10의 경우 이미지는 단순히 32x32x3 블록
- 가장 간단한 비교 방법은 두 이미지를 픽셀별로 비교하고 모든 차이를 더함
- L1 거리(L1 distance)

$$I_1, I_2 \in \mathbb{R}^{3072}$$

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

두 이미지의 비교란?


test image
56 32 10 18
90 23 128 133
24 26 178 200
2 0 255 220
-
training image
10 20 24 17
8 10 89 100
12 16 178 170
4 32 233 112
=
pixel-wise absolute value differences
46 12 14 1
82 13 39 33
12 10 0 30
2 32 22 108
→ 456

최근접 이웃 분류기 코드 구현 - CIFAR-10 불러오기

```
Xtr, Ytr, Xte, Yte = load_CIFAR10('data/cifar10/') # a magic function we provide
# flatten out all images to be one-dimensional
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3) # Xtr_rows becomes 50000 x 3072
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3) # Xte_rows becomes 10000 x 3072
```

- CIFAR-10 데이터셋을 불러옴
- Xtr
  - 훈련세트의 모든 이미지 보유
  - 50,000x32x32x3 크기의 배열
- Ytr
  - 훈련세트의 정답 레이블. 크기: 50,000

최근접 이웃 분류기 코드 구현 - CIFAR-10 불러오기

```
Xtr, Ytr, Xte, Yte = load_CIFAR10('data/cifar10/') # a magic function we provide
# flatten out all images to be one-dimensional
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3) # Xtr_rows becomes 50000 x 3072
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3) # Xte_rows becomes 10000 x 3072
```

- Xte
  - 테스트세트의 모든 이미지 보유
  - 크기: 10,000x32x32x3
- Yte
  - 테스트세트의 정답 레이블. 0~9 숫자로 구성됨
  - 크기: 10,000

최근접 이웃 분류기 코드 구현 - CIFAR-10 불러오기

```
Xtr, Ytr, Xte, Yte = load_CIFAR10('data/cifar10/') # a magic function we provide
# flatten out all images to be one-dimensional
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3) # Xtr_rows becomes 50000 x 3072
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3) # Xte_rows becomes 10000 x 3072
```

- 모든 이미지를 행으로 늘어뜨림
- Xtr_rows: 50,000x3072
- Xte_rows: 10,000x3072 배열

최근접 이웃 분류기 코드 구현 - 훈련/평가

```
nn = NearestNeighbor() # create a Nearest Neighbor classifier class
nn.train(Xtr_rows, Ytr) # train the classifier on the training images and labels
Yte_predict = nn.predict(Xte_rows) # predict labels on the test images
# and now print the classification accuracy, which is the average number
# of examples that are correctly predicted (i.e. label matches)
print 'accuracy: %f' % ( np.mean(Yte_predict == Yte) )
```

- 평가 기준으로는 정확도를 사용
  - 정확도 = #(올바로 예측한 테스트 이미지) / #(전체 테스트 이미지)
- 훈련: 훈련 세트의 이미지와 정답 레이블을 사용하여 레이블 예측 모델 구축
- 평가: 훈련된 모델을 사용하여 테스트 세트의 레이블을 예측

최근접 이웃 분류기 구현 예

```
import numpy as np

class NearestNeighbor(object):
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

    # loop over all test rows
    for i in range(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

최근접 이웃 분류기 구현 예

```
def train(self, X, y):
  """ X is N x D where each row is an example. Y is 1-dimension of size N """
  # the nearest neighbor classifier simply remembers all the training data
  self.Xtr = X
  self.ytr = y
```

- X: 50,000x3072 배열
- y: 50,000 배열
- 최근접 이웃 분류기는 훈련 과정이 필요 없음

최근접 이웃 분류기 구현 예

```
def predict(self, X):
  """ X is N x D where each row is an example we wish to predict label for """
  num_test = X.shape[0]
  # lets make sure that the output type matches the input type
  Ypred = np.zeros(num_test, dtype = self.ytr.dtype)
```

- X: 테스트 세트 (10,000x3072 배열)
- num_test : 10,000
- Ypred: 정답 레이블을 저장할 변수. 크기 10,000 배열

최근접 이웃 분류기 구현 예

```
    # loop over all test rows
    for i in range(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

- 0~9999의 i에 대해 평가 진행
- np.sum(np.abs(x-y)): x와 y의 L1 distance 의미
- Xtr: 50,000x3072, X[i,:]: 3072 에 대해 마이너스 연산 수행시 X[i,:]가 자동적으로 3072 -> 50,000x3072 배열로 broadcasting

최근접 이웃 분류기 구현 예

```
    # loop over all test rows
    for i in range(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

- distances: 크기 50,000 배열
- min_index: distance 값이 가장 작은 값의 인덱스. 훈련 세트에서 몇 번째 이미지인지를 나타냄
- Ypred[i]: distance 값이 가장 작은 훈련 세트 이미지의 레이블

최근접 이웃 분류기 구현 결과

- 이 코드 실행 시 CIFAR-10에서 38.6%의 정확도를 달성
- 무작위 예측의 정확도인 10% (10개의 클래스가 있으므로)보다는 높음
- 인간의 성능(약 94%)이나 최첨단 컨볼루션 신경망 성능(약 95%)에는 미치지 못함

거리의 선택

- 벡터 간의 거리로 L1 거리가 아닌 L2 거리도 사용 가능

$$I_1, I_2 \in \mathbb{R}^{3072}$$

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

```
distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
```

```
distances = np.sqrt(np.sum(np.square(self.Xtr - X[i,:]), axis = 1))
```

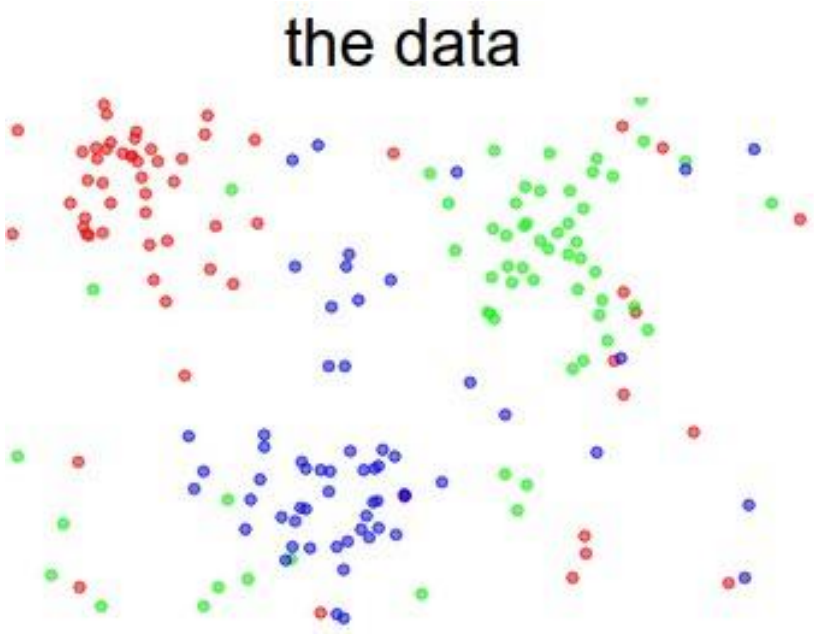
거리의 선택

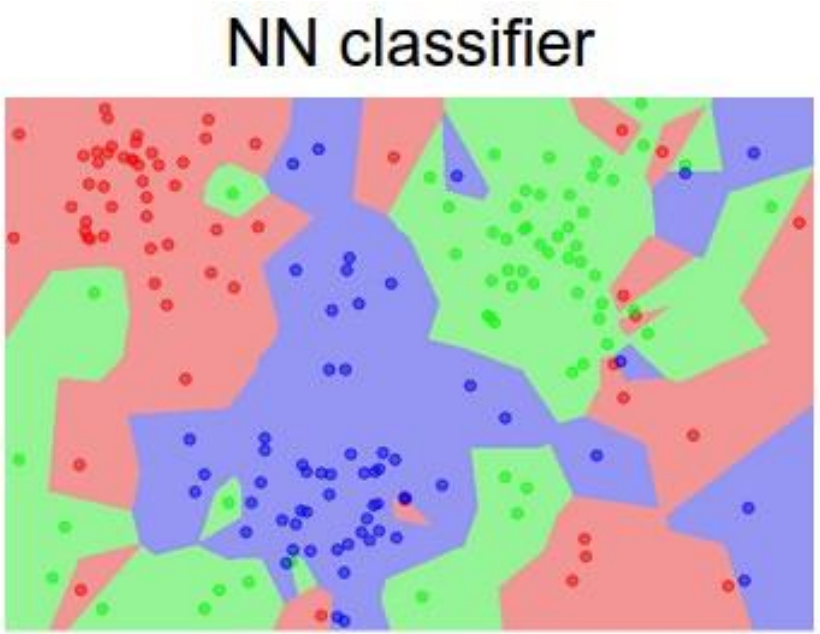
- 실제 최근접 이웃 응용에서는 제곱근 연산 생략 가능
- 거리의 절대 크기는 바뀌지만 순서는 유지하므로 제곱근이 있든 없든 가장 가까운 이웃은 동일
- L2 거리로 최근접 이웃 분류기 실행 결과는 35.4% (L1 거리 결과보다 약간 낮음)

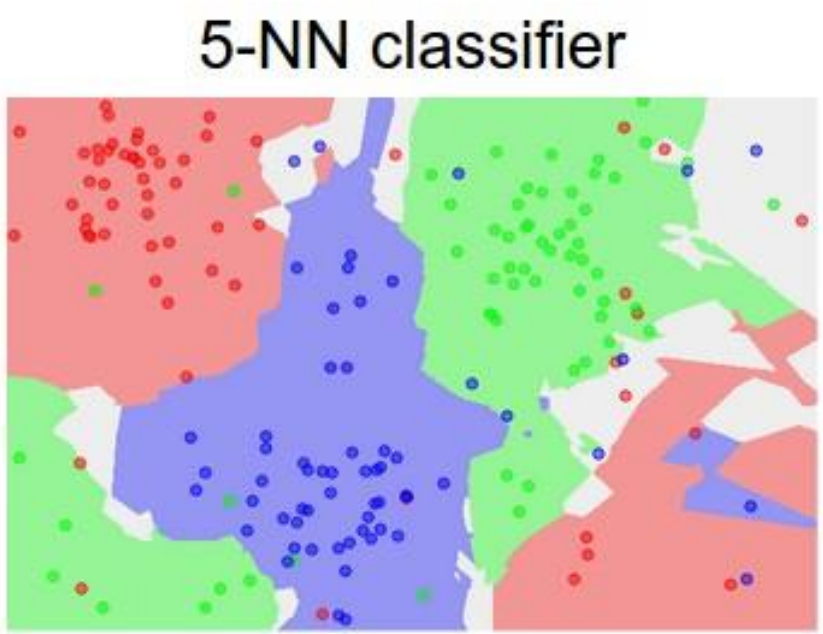
k-최근접 이웃 분류기

- 학습 세트에서 가장 가까운 단일 이미지를 찾는 대신 상위 k개의 가장 가까운 이미지를 찾고 이들 레이블로 테스트 이미지의 예측 레이블을 투표함
- k-최근접 이웃 분류기는 거의 항상 최근접 이웃 분류기(k=1에 해당)보다 좋은 성능을 가짐
- 높은 k는 이상치(outlier)에 대한 저항력 제공

k-최근접 이웃 분류기

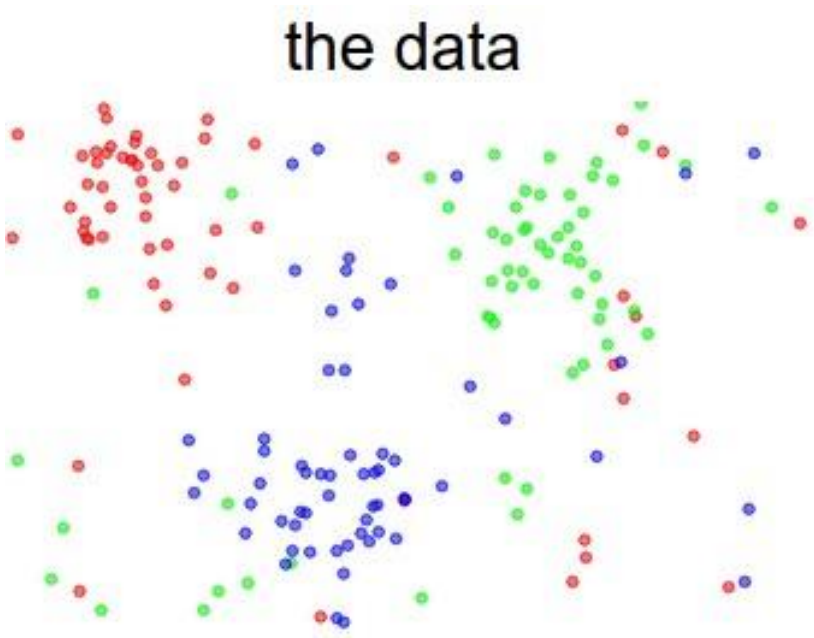


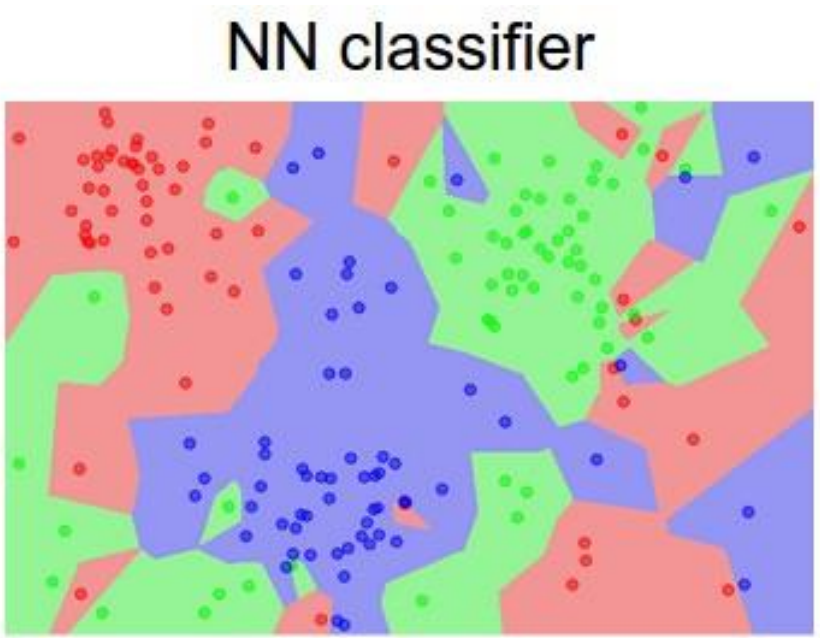


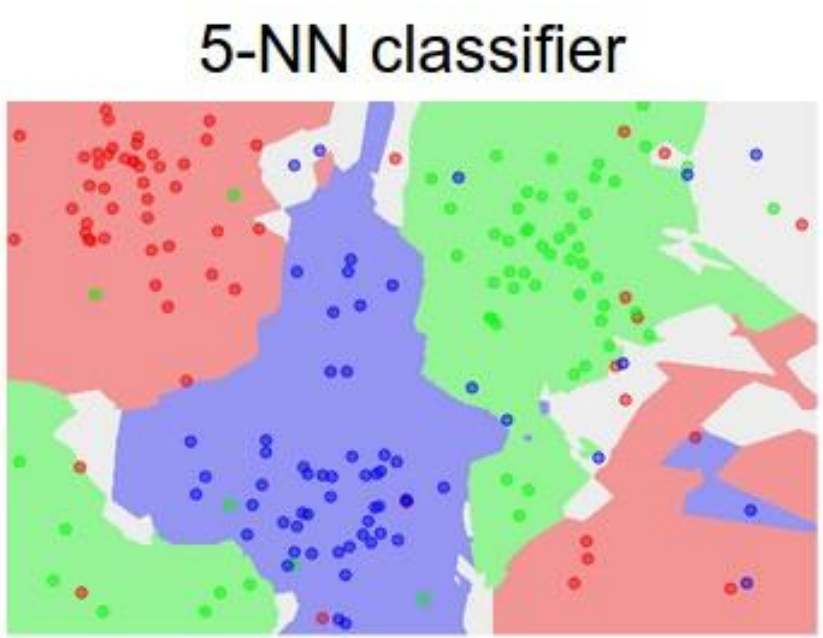


- 파랑, 빨강, 초록: 각 점의 소속 클래스를 나타냄
- 색칠된 영역은 L_2 거리 분류기에 의해 생성된 결정 경계
- 위 그림은 최근접 이웃과 5-최근접 이웃 분류기의 차이를 보여줌

k-최근접 이웃 분류기







- 최근접 이웃 분류기의 경우 잘못된 예측의 작은 섬을 생성
- 5-최근접 이웃 분류기의 경우 이상치에 저항력이 있으므로 더 나은 일반화
- 회색 영역: 최근접 이웃들 사이의 투표에서 동점이 발생한 경우

하이퍼파라미터

- 하이퍼파라미터란?
  - k-최근접 이웃 분류기에서의 k 값 설정 필요
  - L1 norm, L2 norm 등 다양한 거리함수 중 선택 필요
  - 이러한 선택들을 하이퍼파라미터라 부름

하이퍼파라미터

어떤 하이퍼파라미터(k값)을 사용?

하이퍼파라미터 설정 방법

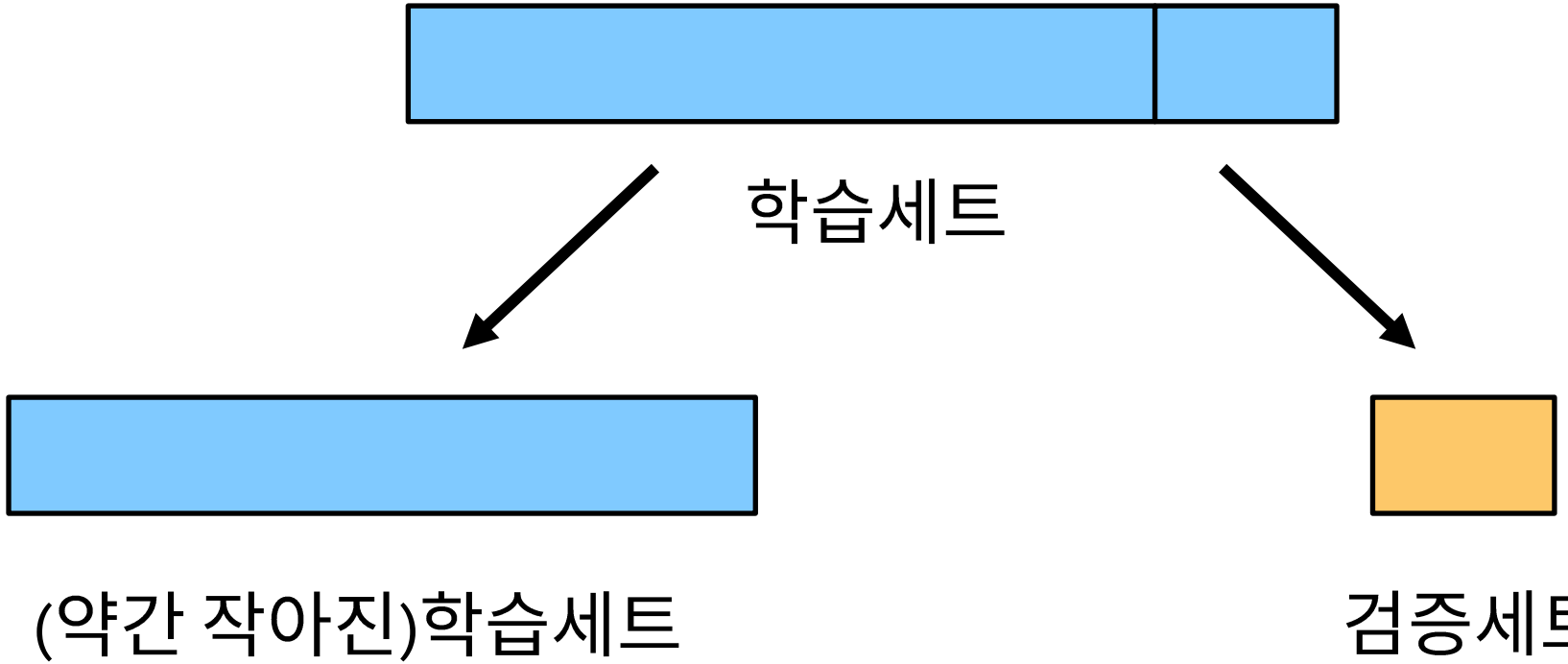
- 많은 다양한 값을 시도해보고 가장 잘 작동하는 값 선택
- 단, 하이퍼파라미터 조정 목적으로 테스트 세트를 사용해서는 안 됨!

테스트 세트

- 머신러닝 알고리즘 설계 시 테스트 세트는 마지막에 한번만 사용해야함
- 테스트 세트에 잘 작동하도록 하이퍼파라미터를 조정하면 실제 모델 배포 시 성능이 크게 감소할 위험이 있음
- 이를 테스트 세트에 과적합(overfit)했다고 표현

검증 세트

- 하이퍼파라미터를 조정하기 위해서는 검증 세트를 사용
- 학습 세트를 두 부분으로 나눔
  - ex) 50,000개 이미지를 49,000개 학습세트와 1,000개 검증세트로 나눔



CIFAR-10에 대한 하이퍼파라미터 조정 예시

```
# assume we have Xtr_rows, Ytr, Xte_rows, Yte as before
# recall Xtr_rows is 50,000 x 3072 matrix
Xval_rows = Xtr_rows[:1000, :] # take first 1000 for validation
Yval = Ytr[:1000]
Xtr_rows = Xtr_rows[1000:, :] # keep last 49,000 for train
Ytr = Ytr[1000:]

# find hyperparameters that work best on the validation set
validation_accuracies = []
for k in [1, 3, 5, 10, 20, 50, 100]:

  # use a particular value of k and evaluation on validation data
  nn = NearestNeighbor()
  nn.train(Xtr_rows, Ytr)
  # here we assume a modified NearestNeighbor class that can take a k as input
  Yval_predict = nn.predict(Xval_rows, k = k)
  acc = np.mean(Yval_predict == Yval)
  print 'accuracy: %f' % (acc,)

  # keep track of what works on the validation set
  validation_accuracies.append((k, acc))
```

CIFAR-10에 대한 하이퍼파라미터 조정 예시

```
Xval_rows = Xtr_rows[:1000, :]
Yval = Ytr[:1000]
Xtr_rows = Xtr_rows[1000:, :]
Ytr = Ytr[1000:]
```

- Xtr_rows: 50,000 x 3072 행렬
- Xval_rows: 1,000 x 3072 행렬 (처음 1,000개 열 선택)
- Xtr_rows: 49,000 x 3072 행렬 (나머지 49,000개 열 선택)
- Ytr: 50,000 배열
- Yval: 1,000 배열 (처음 1,000개 값 선택)
- Ytr: 49,000 배열 (나머지 49,000개 값 선택)

CIFAR-10에 대한 하이퍼파라미터 조정 예시

```
validation_accuracies = []
for k in [1, 3, 5, 10, 20, 50, 100]:
```

- validation_accuracies
  - 다양한 k에 대해 검증세트에 대해 검증한 정확도들을 저장할 리스트
- for문
  - 다양한 k에 대해 반복

CIFAR-10에 대한 하이퍼파라미터 조정 예시

```
nn = NearestNeighbor()
nn.train(Xtr_rows, Ytr)
# here we assume a modified NearestNeighbor
Yval_predict = nn.predict(Xval_rows, k = k)
acc = np.mean(Yval_predict == Yval)
print 'accuracy: %f' % (acc,)
```

- NearestNeighbor
  - k-최근접 분류기 수행에 대한 클래스
- nn.train
  - 훈련 세트와 정답 레이블을 입력으로 넣어주면 훈련시키는 함수
  - 최근접 분류기의 경우 훈련은 단순히 훈련 세트와 정답레이블을 저장 (Xtr_rows: 49,000 x 3072 행렬, Ytr: 49,000 배열)

CIFAR-10에 대한 하이퍼파라미터 조정 예시

```
nn = NearestNeighbor()
nn.train(Xtr_rows, Ytr)
# here we assume a modified NearestNeighbor
Yval_predict = nn.predict(Xval_rows, k = k)
acc = np.mean(Yval_predict == Yval)
print 'accuracy: %f' % (acc,)
```

- nn.predict
  - 검증세트 및 하이퍼파라미터 k값을 입력으로 넣어주면 각 검증세트 이미지에 대해 정답 레이블을 예측
  - Xval_rows: 1000 x 3072
  - Yval_predict: 1000

CIFAR-10에 대한 하이퍼파라미터 조정 예시

```
nn = NearestNeighbor()
nn.train(Xtr_rows, Ytr)
# here we assume a modified NearestNeighbor
Yval_predict = nn.predict(Xval_rows, k = k)
acc = np.mean(Yval_predict == Yval)
print 'accuracy: %f' % (acc,)
```

- Yval_predict == Yval
  - Yval_predict: 1000 배열
  - Yval: 1000 배열
  - == 연산은 두 배열의 각 성분이 같은지 다른지 아닌지 확인. 같으면 True, 아니면 False를 반환 (동일하게 1000 배열)

CIFAR-10에 대한 하이퍼파라미터 조정 예시

```
nn = NearestNeighbor()
nn.train(Xtr_rows, Ytr)
# here we assume a modified NearestNeighbor
Yval_predict = nn.predict(Xval_rows, k = k)
acc = np.mean(Yval_predict == Yval)
print 'accuracy: %f' % (acc,)
```

- np.mean
  - 주어진 배열의 평균 값을 계산함

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
average = np.mean(arr)
print(average)  # 출력: 3.0
```

- acc
  - Yval_predict == Yval 배열은 0과 1의 배열로 인식될 수 있음
  - np.mean 함수를 적용시키면 레이블이 일치할 확률을 얻음
  - 즉, 분류 정확도(accuracy)에 해당됨

CIFAR-10에 대한 하이퍼파라미터 조정 예시

```
validation_accuracies.append((k, acc))
```

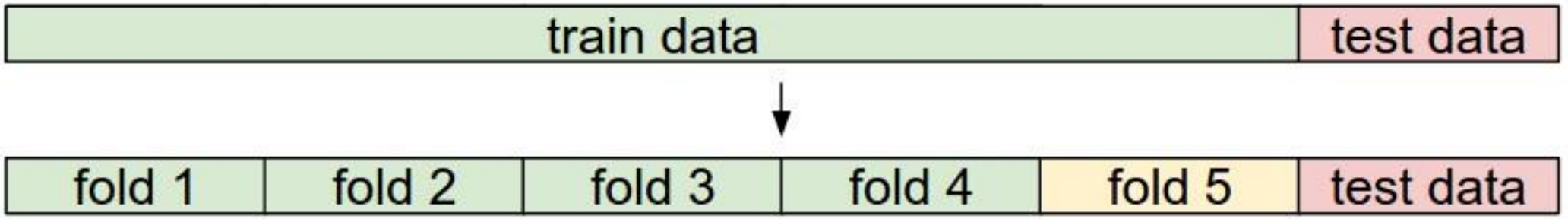
- 각각의 k값과 정확도 acc값의 튜플을 validation_accuracies 리스트에 저장
- 어떤 k값이 가장 효과적인지(정확도가 높은지)를 알 수 있음
- 가장 좋은 k값을 선택하여 실제 테스트 세트에서 한 번만 실행하고 테스트 정확도를 도출

교차 검증(cross-validation)

- 훈련 세트의 크기(그리고 검증 데이터의 크기도)가 작을 경우 종종 교차 검증(cross-validation)이라는 기술을 사용하여 하이퍼파라미터를 조정
- 다양한 검증 세트에 대해 성능 평가하고 평균 값을 구하여 특정 k값의 성능에 대한 노이즈가 덜한 추정치를 얻을 수 있음

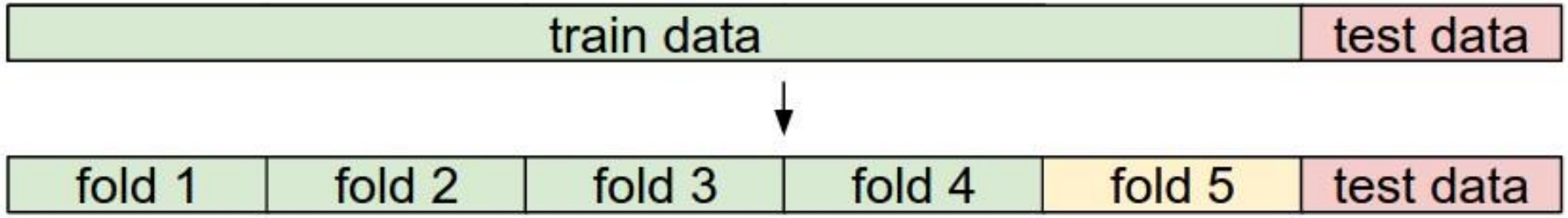
교차 검증

- 예시: 5겹(5-fold) 교차 검증
  - 훈련 세트를 5개의 동일한 폴드(fold)로 나누고, 그 중 4개를 훈련 세트로 사용하며 1개를 검증 세트로 사용
  - 그 다음, 어떤 폴드가 검증 폴드인지 반복하여 성능을 평가하고 평균 성능 도출



교차 검증

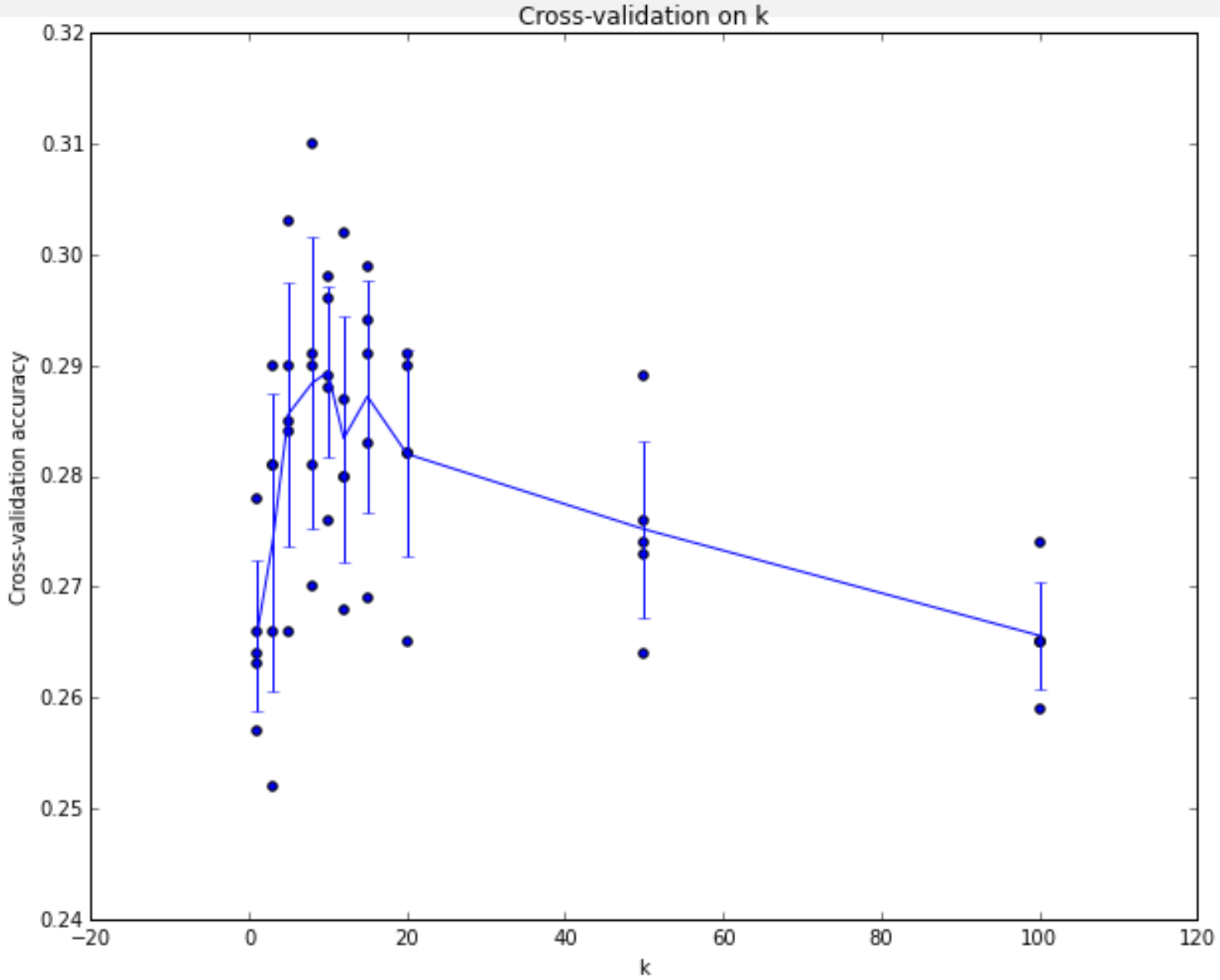
- 예시: 5겹(5-fold) 교차 검증



훈련세트				검증세트	
fold 1	fold 2	fold 3	fold 4	fold 5	val. acc. 1
fold 1	fold 2	fold 3	fold 5	fold 4	val. acc. 2
fold 1	fold 2	fold 4	fold 5	fold 3	val. acc. 3
fold 1	fold 3	fold 4	fold 5	fold 2	val. acc. 4
fold 2	fold 3	fold 4	fold 5	fold 1	val. acc. 5

mean val. acc.

교차 검증


Cross-validation on k
Cross-validation accuracy
k
0.32
0.31
0.30
0.29
0.28
0.27
0.26
0.25
0.24
−20
0
20
40
60
80
100
120

교차 검증 실제 활용

- 실제로는 교차 검증의 큰 계산적 비용 때문에 보통 교차 검증보다 단일 검증 분할을 선호
- 일반적으로 훈련 데이터의 50%에서 90%를 훈련세트로 사용하고 나머지를 검증세트로 사용
- 만약 하이퍼파라미터의 수가 많다면 더 큰 검증 분할을 선호
- 검증세트의 이미지 수가 적으면(ex. 수백개 정도) 교차 검증을 사용하는 것이 안전
- 보통 3-겹, 5-겹, 10-겹 교차 검증이 종종 사용됨

최근접 이웃 분류기의 장단점

장점

1. 구현이 간단하고 이해하기 쉬움
2. 훈련 과정이 필요없음

단점

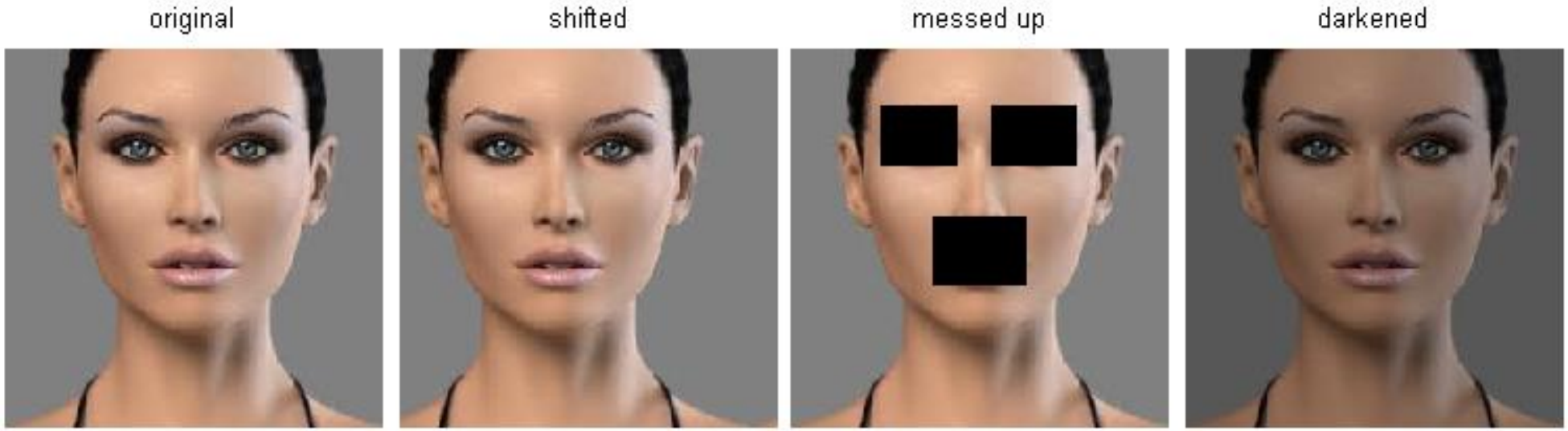
1. 테스트 계산 비용이 매우 큼
2. 모든 훈련 데이터를 저장해야하므로 공간 비용이 매우 큼
2. 이미지에서의 거리는 의미적 유사성과 일치하지 않음

최근접 이웃 분류기의 단점: 큰 테스트 비용

- 테스트세트 이미지를 분류하려면 모든 훈련세트와의 비교가 필요하므로 테스트 비용이 매우 큼
- 훈련 시간보다는 테스트 시간을 줄이는 것이 훨씬 더 바람직함
- 훈련 비용은 크지만, 테스트 시간은 매우 작은 심층 신경망(deep neural networks)과 반대

최근접 이웃 분류기의 단점: 픽셀 별 거리의 한계

- 이미지가 고차원 객체(많은 픽셀을 포함)이므로 고차원 공간에서의 거리는 매우 직관적이지 않음
- 원본 이미지(왼쪽)과 그 옆의 세 개의 다른 이미지들은 L2 픽셀 거리를 기반으로 원본 이미지와 모두 동일하게 떨어져있음
- 픽셀 별 거리는 인식적, 의미적 유사성과 전혀 일치하지 않음



최근접 이웃 분류기의 단점: 픽셀 별 거리의 한계