

컴퓨터비전

Team Project

-1차 과제-

교 수 명	김진현 교수님
교 과 명	컴퓨터 비전
제 출 일	2024. 05. 28 (화)
조 원	임주형(조장), 이세비, 최하은

8.2(최하은)

내가 잘한 점, 차별화 된 점

: 학습데이터의 수를 적게 설정하여 학습할 데이터에 따라 정확도의 출력에 대해 분석을 해보았다.

1. 문제 요약

knn은 학습한 데이터는 모두 인식하는가?

2. 결론

학습할 데이터가 지나치게 많아질 경우에 정확도가 떨어진다. 랜덤 데이터로 학습할 경우, 데이터의 수가 많아질수록 모델의 성능이 향상되지만, 중복되는 데이터가 많아지고 질적인 문제로 인해 정확도가 떨어질 수 있다.

3. 실험 1: 랜덤 데이터를 이용한 학습과 테스트

- 학습데이터(랜덤)를 10으로 하고 랜덤 범위를 2로 적게 하여 정확도를 출력
- 코드

```
L = 10
data = np.random.randint(0, 2, (L, 2)).astype(np.float32)
labels = np.random.randint(0, 2, (L, 1)).astype(np.float32)

knn = cv2.ml.KNearest_create()

import time
s_time = time.time()
knn.train(data, cv2.ml.ROW_SAMPLE, labels)

from collections import Counter
K = [1]
def print_ret_values(ret_val, rslt_v='no', nfg_v='no', dst_v='no'):
    ret_val, results, neighbours, dist = ret_val
    print('results:', type(results), results.shape)
    if rslt_v != 'no': print(results)
    print('neighbours:', type(neighbours), neighbours.shape)
    if nfg_v != 'no': print(neighbours)
    print('dist:', type(dist), dist.shape)
    if dst_v != 'no': print(dist)
```

```

def get_accuracy(predictions, labels):
    accuracy = (np.squeeze(predictions) == labels).mean()

for k in K:
    print(f'\ntest=train: k={k}, num of test data={len(data)} -----')
    s_time = time.time()
    ret_val = knn.findNearest(data, k)
    e_time = time.time()

    print(f'testing time: whole={e_time - s_time:#.2f},
          unit={(e_time - s_time)/len(data):#.2f}')
    print_ret_values(ret_val, rslt_v='yes', nfg_v='yes', dst_v='yes')
    print(labels)
    ret, results, neighbours, dist = ret_val

    cmp = labels == results
    cmp_f = cmp.flatten()
    dict = Counter(cmp_f)
    print(f'test=train: L={L},
          k={k}: Accuracy={dict[True] * 100 / len(cmp):#.2f}%')

```

- 결과

data	results	neighbours	dist	labels
[[0. 1.]	[[1.]	[[1.]	[[0.]	[[1.]
[1. 1.]	[0.]	[0.]	[0.]	[0.]
[1. 0.]	[1.]	[1.]	[0.]	[1.]
[0. 1.]	[1.]	[1.]	[0.]	[1.]
[1. 0.]	[1.]	[1.]	[0.]	[0.]
[1. 0.]	[1.]	[1.]	[0.]	[0.]
[0. 1.]	[1.]	[1.]	[0.]	[1.]
[0. 0.]	[1.]	[1.]	[0.]	[1.]
[0. 0.]	[1.]	[1.]	[0.]	[1.]
[1. 0.]	[1.]	[1.]	[0.]	[0.]

test=train: L=10, k=1: Accuracy=70.00%

데이터가 [1.0.]일 때 results와 neighbours가 [1.]인데 labels가 [0.]으로 출력되어 불일치하는 것을 확인했다. 이전에 동일한 [1.0] 데이터 값에 대해 labels를 [1.]로 할당했음에도 불구하고, 또다시 같은 데이터 값 [1.0]에 대해 labels를 [0.]로 할당한 것이다. 이는 기존 labels에 할당된 값을 덮어쓰며 새롭게 다시 labels을 부여하여 정확도가 떨어진다.

테스트하려는 데이터가 많아질수록 중복되는 데이터가 많아질 수밖에 없다. 하지만 이 중복된 데이터

가 다 똑같은 값을 할당하지 않기 때문에 정확도가 떨어진다. 또한 랜덤 데이터는 특정 패턴이나 구조가 부족할 수 있기 때문에, 데이터의 특성을 제대로 반영하지 못하고 성능 저하를 초래할 수 있다. 랜덤 데이터는 실험적 또는 시뮬레이션된 데이터로서, 어떠한 구조적 패턴이나 연관성을 반드시 반영하지 않을 수 있다. 이는 종종 데이터의 특성이 불명확하거나, 특정 문제에 대해 잘못된 정보를 제공할 수 있다. 실제 문제와의 관련성이 낮고 노이즈가 많은 랜덤 데이터는 모델에서 유의미한 정보를 제공하지 못하게 하고, 때로는 모델이 불필요하거나 잘못된 패턴으로 학습하게 하는 경우도 있다.

하지만 랜덤 데이터의 범위를 넓히고 학습 데이터의 수를 줄이면 중복 데이터가 줄어들어 정확도는 높아질 것이다. 중복된 값들이 얼마나 정확한 값을 할당하는지에 따라 정확도가 달라진다.

특히 KNN 모델에서는 $k=1$ 일 때, 자기 자신만을 기준으로 거리를 측정하기 때문에 정확도가 높을 것이라고 생각한다. 하지만 학습 데이터의 수가 10개이고 랜덤 범위가 좁을 때, 주변 데이터를 고려하지 못해 이미 존재하는 데이터가 치명적일 수 있다. 따라서 중복 데이터에 동일한 값을 할당하는지에 따라 코딩을 실행할 때마다 정확도가 계속해서 달라진다.

```
test=train: L=10, k=1: Accuracy=60.00%
```

```
test=train: L=10, k=1: Accuracy=40.00%
```

```
test=train: L=10, k=1: Accuracy=90.00%
```

4. 실험 2: 5000개의 문자 데이터 학습 정확도

- 20 x 20 으로 5000개의 문자데이터의 학습정확도 출력
- 코드

```
SIZE_IMAGE_ROW = 20
SIZE_IMAGE_COL = 20
NUMBER_CLASSES = 10

def load_digits_and_labels(big_image):
    digits_img = cv2.imread(big_image, 0)
    number_cols = digits_img.shape[1] / SIZE_IMAGE_COL
    number_rows = digits_img.shape[0] / SIZE_IMAGE_ROW
    rows = np.vsplit(digits_img, number_rows)

    digits = []
    for row in rows:
        row_cells = np.hsplit(row, number_cols)
        for digit in row_cells:
            digits.append(digit)
    digits = np.array(digits)
```

```

    labels = np.repeat(np.arange(NUMBER_CLASSES), len(digits) / NUMBER_CLASSES)
    return digits, labels

def get_accuracy(predictions, labels):
    accuracy = (np.squeeze(predictions) == labels).mean()
    return accuracy * 100

digits, labels = load_digits_and_labels('../data/digits.png')

for i in digits[0]:
    print(i)
shuffle = np.random.permutation(len(digits))
digits, labels = digits[shuffle], labels[shuffle]

raw_descriptors = []
for img in digits:
    raw_descriptors.append(np.float32(img.flatten()))
raw_descriptors = np.array(raw_descriptors)
partition = int(0.5 * len(raw_descriptors))
raw_descriptors_train, raw_descriptors_test = np.split(raw_descriptors,
[partition])

labels_train, labels_test = np.split(labels, [partition])

knn = cv2.ml.KNearest_create()
knn.train(raw_descriptors_train, cv2.ml.ROW_SAMPLE, labels_train)

ret, result, neighbours, dist = knn.findNearest(raw_descriptors_test, 1)
acc = get_accuracy(result, labels_test) # test 데이터를 분류했을 때의 정확도를 확인
print(f"k={1}: Accuracy for test data: {acc:#6.2f}")

ret, result, neighbours, dist = knn.findNearest(raw_descriptors_train, 1)
acc = get_accuracy(result, labels_train) # 학습데이터를 분류했을 때의 정확도를 확인
print(f"k={1}: Accuracy for train data: {acc:#6.2f}")

```

- 결과

[illegible]

```
k=1: Accuracy for test data: 93.12
```

```
k=1: Accuracy for train data: 100.00
```

Digit.png에서는 총 5000개의 데이터에서 중복된 값이 거의 없다는 것을 확인할 수 있다. 그래서 $K=1$ 일 때에는 주변을 생각하지 않고 자기 자신을 검증하기 때문에 당연히 학습 데이터의 정확도는 100%이다.

또한 이미지 데이터는 특정 구조와 패턴을 가지고 있고, 시각적인 정보도 포함되어 있다. 보통 높은 차원의 데이터이며, 다양한 특성을 가지고 있다. 랜덤 데이터보다 매우 구체적인 정보를 포함하고 있어서 많은 정보의 데이터를 통해 대량의 데이터에서도 높은 정확도를 출력할 수 있다.

5. 최종 결론

k-NN 모델은 학습한 데이터를 인식하지만, 아무리 $K=1$ 이라도 학습할 데이터의 질과 중복 여부에 따라 정확도가 달라진다. k-NN 알고리즘 자체에서 자동적으로 일반화가 이루어진다고 보기는 어려우며, 성능을 최적화하기 위해서는 데이터 전처리, 적절한 k 값 선택, 차원 축소 등 다양한 기법을 적용해야 한다. 더하여 학습할 데이터에 대한 깊은 이해도 필요하다.

8.3(이세비)

내가 잘한 점, 차별화 된 점

: get_accuracy() 함수의 로직을 한 줄씩 이해하려고 한 점, 더하여 shape를 파악할 수 있도록 코드를 수정하여 get_accuracy() 함수의 내부 동작 과정을 파악한 점이다. 이런 과정은 보다 명확한 문제 원인 규명을 가능하게 했으며, 두 가지 다른 방법으로 코드를 수정하여 문제를 해결하는 방법을 제시했다.

1. 문제 요약

다른 예제(ex. 2_1_knn_handwritten_digits_recognition_varying_k.py)에서 사용했던 get_accuracy() 함수가 여기서(1_1_checking_knn_model_accuracy.py)는 계속 50%대의 정확도로 고정되어 출력된다. 어떤 오류가 숨겨져 있는지 원인을 파악하여 해결해 보자.

2. 결론

get_accuracy 함수 내부의 데이터 값 비교 과정을 살펴보면 np.squeeze(predictions)과 labels를 비교하여 정확도를 계산하는데, 이 두 데이터의 차원과 형태가 일치하지 않아 정확도에 차이를 초래한다. 정확한 비교를 위해 데이터의 shape를 일치시키는 과정이 필요하며, 이를 통해 정확도를 높일 수 있다.

아래와 같이 코드를 수정하면 원하는 높은 정확도를 확인할 수 있다.

```
# 코드 수정 방법 1
def get_accuracy(predictions, labels):
    accuracy = (np.squeeze(predictions) == np.squeeze(labels)).mean()
    return accuracy * 100


# 코드 수정 방법 2
def get_accuracy(predictions, labels):
    accuracy = (predictions == labels).mean()
    return accuracy * 100
```

3. get_accuracy()의 로직

```
def get_accuracy(predictions, labels):  
    accuracy = (np.squeeze(predictions) == labels).mean()  
    return accuracy * 100
```

이 함수는 예측 결과인 predictions와 실제 레이블인 labels를 비교하여 정확도를 계산한다.

함수의 주요 로직:

1. **np.squeeze(predictions)**: np.squeeze(predictions)를 사용하여 예측 결과를 1차원 배열로 변환한다. 이는 예측 결과가 2차원 배열이거나 다차원 배열인 경우에도 처리할 수 있도록 한다.
 np.squeeze는 NumPy에서 사용되는 함수 중 하나.
이 함수는 배열에서 크기가 1인 차원을 제거하여 배열의 모양을 변경.
2. **(np.squeeze(predictions) == labels)**: 변환된 예측 결과와 실제 레이블 labels를 비교하여 같은지 여부를 나타내는 불리언 배열을 생성한다. 이때, == 연산자를 사용하여 각 요소를 비교해 예측값과 레이블이 일치하는 경우 True를, 그렇지 않은 경우 False를 반환한다.
3. **.mean()**: mean() 함수를 사용하여 불리언 배열의 평균을 계산한다. 이는 True와 False의 비율을 계산하여 정확도를 구하는 것이다. True는 1로 간주되고 False는 0으로 간주되므로, True의 비율은 예측 결과가 실제 레이블과 일치하는 비율, 즉 정확도를 나타낸다.
4. **return accuracy * 100**: 정확도를 백분율로 표시하기 위해 100을 곱하여 반환한다.

위의 과정에서 가장 중요한 부분:

예측값(predictions)과 실제 레이블(labels)을 비교하는 부분이다. 이 부분이 올바르게 작동하지 않으면 함수의 반환값도 올바르게 않을 것이다.

4. 두 예제에서 사용된 get_accuracy 함수의 동작을 비교하는 실험

1) 실험 개요

두 가지 예제에서 사용된 get_accuracy 함수의 동작을 비교하고 그 차이에 대해 설명하기 위해 실행했다. 예제 1은 2_1_knn_handwritten_digits_recognition_varying_k.py이고 예제 2는 1_1_checking_knn_model_accuracy.py이나, 현재 코드로 지칭한다.

2) 실험 방법

두 예제에서는 동일한 get_accuracy 함수를 사용하여 예측값과 실제 레이블 간의 정확도를 계산한다. 그러나 두 예제 간에 정확도가 다르게 나타나는 것을 관찰했다. 특히, 현재 코드에서 정확도가

50%대의 낮은 정확도를 출력하고 있다. 실험을 통해 이러한 차이를 이해하기 위해 get_accuracy 함수의 동작 과정을 확인하였다.

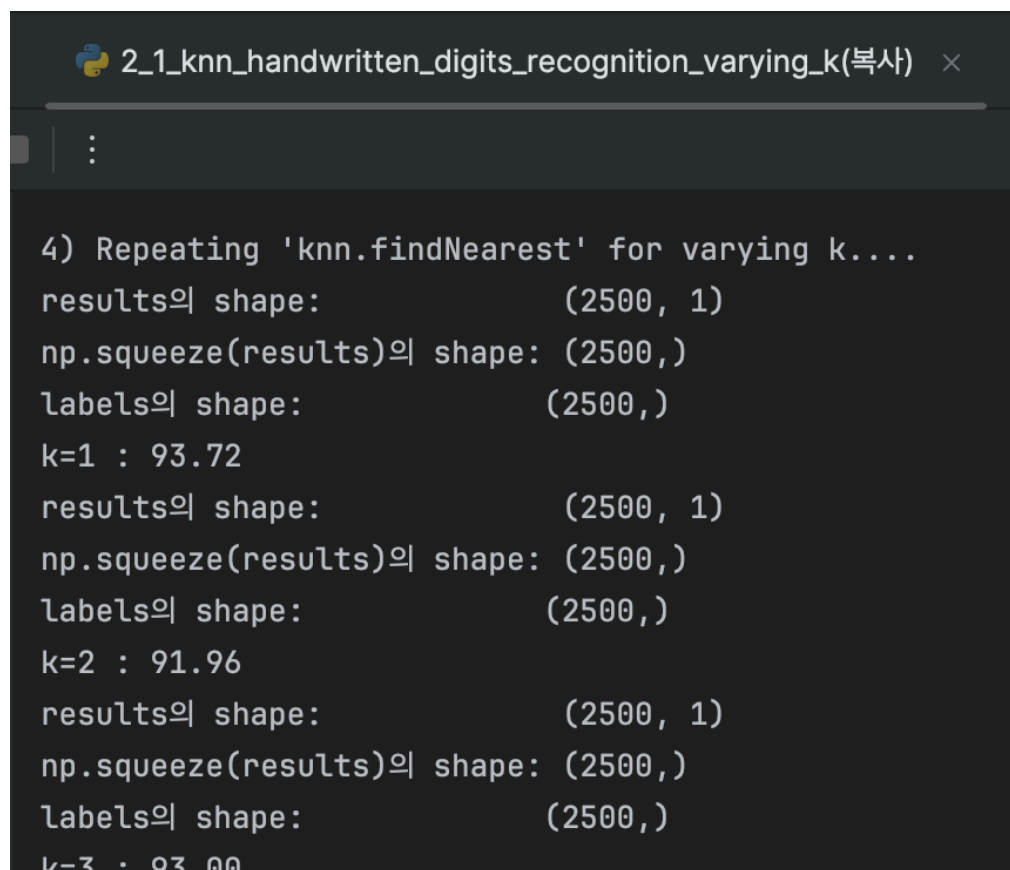
```
def get_accuracy(predictions, labels):  
    print(f'results의 shape: {predictions.shape} ')  
    print(f'np.squeeze(results)의 shape: {np.squeeze(predictions).shape} ')  
    print(f'labels의 shape: {labels.shape} ')  
    accuracy = (np.squeeze(predictions) == labels).mean()  
    return accuracy * 100
```

이처럼 코드를 수정 후, 실행해 확인 과정을 거쳤다.

3) 실험 결과 및 분석

예제 1: 2_1_knn_handwritten_digits_recognition_varying_k.py

- 결과 확인:



```
4) Repeating 'knn.findNearest' for varying k....  
results의 shape: (2500, 1)  
np.squeeze(results)의 shape: (2500,)  
labels의 shape: (2500,)  
k=1 : 93.72  
results의 shape: (2500, 1)  
np.squeeze(results)의 shape: (2500,)  
labels의 shape: (2500,)  
k=2 : 91.96  
results의 shape: (2500, 1)  
np.squeeze(results)의 shape: (2500,)  
labels의 shape: (2500,)  
k=3 : 93.00
```

- 분석:
results는 (2500, 1)의 shape를 가지고 있다. 따라서 np.squeeze(results)를 사용하여 1차원 배열로 변환한 후, 정확도를 계산하면 적절한 결과가 나타난다. 실제 레이블과의 비교 시 차원이 일치하여 정확도를 올바르게 측정할 수 있기 때문이다. 이 경우, 정확도는 90%대로 측정되었다.

현재 코드: 1_1_checking_knn_model_accuracy.py

- 결과 확인:

```
1_1_checking_knn_model_accuracy x
:

Step 3: Checking the knn model according to k selection..

test=train: k=1, num of test data=400 -----
testing time: whole=0.00, unit=0.00
results: <class 'numpy.ndarray'> (400, 1)
neighbours: <class 'numpy.ndarray'> (400, 1)
dist: <class 'numpy.ndarray'> (400, 1)
test=train: L=400, k=1: Accuracy=99.50%
k=1: Accuracy2= 50.24

Process finished with exit code 0
```

- 분석:
results와 labels의 shape가 같다. 이로 인해 np.squeeze(results)를 사용하여 1차원 배열로 변환하여 labels와의 비교 시 차원이 맞지 않아 정확한 비교가 이루어지지 않는다. 결과적으로 정확도는 실제보다 낮은 값을 반환하게 된다. 이러한 차이는 데이터의 차원과 형태가 정확도 계산에 영향을 미침을 보여준다.

4) 결론 및 개선 방향

위 실험을 통해 데이터의 차원과 형태가 정확도에 중요한 영향을 미친다는 것을 확인할 수 있었다. 따라서 정확도를 계산할 때는 results와 labels의 차원과 형태를 일치시키는 것이 중요하다. 이를 위해 데이터의 shape를 확인하고 필요에 따라 reshape 작업을 수행하여 일관된 비교를 보장해야 한다. 이러한 수정은 정확도 계산 함수에 적용되어야 하며, 현재 코드에서는 데이터의 shape를 맞추는 전처리 단계를 추가함으로써 문제를 해결할 수 있다. 이를 통해 모델의 성능을 정확하게 측정할 수 있을 것이다.

5. 해결 방법

```
# 코드 수정 방법 1
def get_accuracy(predictions, labels):
    accuracy = (np.squeeze(predictions) == np.squeeze(labels)).mean()
    return accuracy * 100

# 코드 수정 방법 2
def get_accuracy(predictions, labels):
    accuracy = (predictions == labels).mean()
    return accuracy * 100
```

results와 labels의 차원과 형태를 일치시켜 비교하도록 코드를 수정했다.

```
test=train: k=1, num of test data=400 -----
testing time: whole=0.00, unit=0.00
results: <class 'numpy.ndarray'> (400, 1)
neighbours: <class 'numpy.ndarray'> (400, 1)
dist: <class 'numpy.ndarray'> (400, 1)
test=train: L=400, k=1: Accuracy=99.00%
k=1: Accuracy2= 99.00
```

수정 이후, 50%대이던 Accuracy2가 95% 이상의 높은 정확도로 출력된다. 더하여 Accuracy2값이 다른 방법으로 정확도를 계산했던 Accuracy 값과 일치하는 것을 확인할 수 있다.

8.4(최하은)

내가 잘한 점, 차별화 된 점

: KNN 모델이 저장하고 있는 방식에 대해 공부하여 다양한 함수를 통해서 모델의 크기를 판단하였다.

1. 문제 요약

KNN 모델은 학습한 데이터를 모두 저장하는가?
모델의 크기가 학습 데이터의 양에 따라 달라지는가?

2. 결론

KNN 모델을 저장해 보아서 확인하니 학습 데이터가 적을 때와 많을 때의 파일 크기 달라진다. 이를 통해 학습 데이터의 양이 많아질수록 모델 파일 크기가 증가한다는 것을 알 수 있다. 하지만 학습 데이터가 KNN 모델 내부에 저장되고 있는지는 정확히 확인이 불가능하다.

3. KNN모델이 학습한 데이터를 내부적으로 저장하는지 확인하는 실험

1) 실험 코드

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

L = 400      # 작은 수를 사용하면 프린트하기 수월하다.
data = np.random.randint(0, 100, (L, 2)).astype(np.float32)
labels = np.random.randint(0, 2, (L, 1)).astype(np.float32)
print('data for training:', type(data), data.shape)
print('Labels for training:', type(labels), labels.shape)

knn = cv2.ml.KNearest_create()

import time

s_time = time.time()
knn.train(data, cv2.ml.ROW_SAMPLE, labels)

import os
import sys
from os.path import getsize
import os

#파일 저장
```

```

file_path = 'knn_data.xml'
knn.save(file_path)
f_size = getsize("knn_data.xml")
print(f'knn file size before training={f_size}')
print(f'model size before training= {sys.getsizeof(knn)}')
file_size = os.path.getsize(file_path)
print(f"Model file size: {file_size} bytes")

# 파일 존재 여부 확인
if os.path.exists(file_path):
    # 파일 읽기 권한 확인
    if os.access(file_path, os.R_OK):
        print("File exists and is readable")
        knn_loaded = cv2.ml.KNearest_create()
        knn_loaded.load(file_path)
    else:
        print("File is not readable")
else:
    print("File does not exist")

# 학습 데이터의 양 출력 (원본 데이터 크기를 기반으로)
print("Number of training samples:", len(data))
print(f'model size after training= {sys.getsizeof(knn_loaded)}')
file_size = os.path.getsize(file_path)
print(f"Model file size: {file_size} bytes")

```

2) 실험 결과

- 학습 데이터를 400개를 설정했을 때

```

data for training: <class 'numpy.ndarray'> (400, 2)
Labels for training: <class 'numpy.ndarray'> (400, 1)
knn file size before training=5134
model size before training= 32
Model file size: 5134 bytes
File exists and is readable
Number of training samples: 400
model size after training= 32
Model file size: 5134 bytes

Process finished with exit code 0

```

model file size가 5134 byte로 출력값이 나온다.

- 학습데이터를 40000개로 설정했을 때

```
data for training: <class 'numpy.ndarray'> (40000, 2)
Labels for training: <class 'numpy.ndarray'> (40000, 1)
knn file size before training=471840
model size before training= 32
Model file size: 471840 bytes
File exists and is readable
Number of training samples: 40000
model size after training= 32
Model file size: 471840 bytes

Process finished with exit code 0
```

model file size가 471840 bytes로 늘어난 것을 확인할 수 있다.

Python의 pickle 모듈은 일부 C 기반 라이브러리 객체, 특히 OpenCV 객체 같은 경우 직렬화하는 데 한계가 있다. 이런 객체들은 내부적으로 복잡한 C++ 구조를 가지고 있기 때문에 pickle로 knn 모델을 직렬화할 수 없다. OpenCV의 KNN 모델과 같은 경우, 다른 방법으로 모델 데이터를 저장하고 로드할 수 있다. OpenCV는 이를 위해 cv2.FileStorage를 제공한다. 이를 사용하여 모델의 파라미터를 XML 또는 YAML 파일로 저장하고 불러올 수 있다.

KNN 모델은 학습 단계에서 데이터를 저장하는 방식으로 작동하기 때문에, "학습 전"과 "학습 후"의 데이터양은 변경되지 않는다. KNN은 학습 과정에서 데이터를 변형하거나 압축하지 않고, 전체 데이터 세트를 메모리에 그대로 저장한다.

- **sys.getsizeof()** sys.getsizeof()를 호출하는 것은 모델 객체 자체의 기본 크기만을 제공한다. 크기는 객체 자체의 저장 공간과 함께 객체가 내부적으로 참조하는 다른 객체들의 크기를 포함하지 않는다. 리스트 객체의 크기를 반환할 때 리스트가 담고 있는 항목들의 크기는 포함되지 않고, 리스트 구조 자체의 메모리 크기만을 계산한다.
- **os.path.getsize()** 반면에 os.path.getsize()를 호출하는 것은 파일 시스템에 있는 파일의 크기를 바이트 단위로 반환한다. 이는 실제 파일의 디스크 상의 크기를 의미하며, 파일 내용에 따라 달라진다는 것을 확인했다.

4. 최종 결론

결론적으로, KNN 모델은 xml로 저장하여 학습 데이터가 적을 때와 많을 때의 파일 크기가 달라진다. 학습 데이터가 양이 많아질수록 bytes가 늘어나는 것을 확인할 수 있다. 하지만 학습 데이터가 모델 안에 학습 데이터를 저장하고 있는지는 확인할 수가 없다. 왜냐하면 KNN 모델은 학습 단계에서만 저장하는 기능을 하고 있어 학습 후에 모델 객체가 지정하는 학습 데이터가 포함되는지는 판단이 불가능하기 때문이다.

9.4(임주형)

내가 잘한 점, 차별화 된 점

: k에 따라 서브 플롯에 각각 다른 상태의 화면을 보여주어 비교가 쉽게 되도록 하였다.

1. 문제 요약

query data point를 k=2, 4..일 때, 전 영역으로 확장하여 색칠하여 비교해 보아라.

- 모든 실험에 같은 학습 데이터를 사용하게 만들어야 한다. -> seed() 함수
- 학습 데이터의 레이블의 분포(비율)를 통제할 수 있어야 한다. -> choice() 함수

2. 결론

7:3 비율과 3:7 비율의 결과를 비교해 볼 때 label=0으로 나타나는 범위가 비교적 넓음을 알 수 있었고, k가 짝수일 때 외부 요소가 knn 모델의 결정 시에 영향을 끼칠 수 있겠지만, 개인적인 의견으로는 label의 순서가 앞인 경우 더 우선시 되는 것 같다고 여겨진다.

3. 코드

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

N = 16
K = 5 # 1 ~ 5 범위

np.random.seed(370001)
data = np.random.randint(0, 100, (N, 2)).astype(np.float32)
'''
# label 0 - 7 : label 1 - 3 비율
labels = np.zeros((N, 1), dtype=np.float32)
num_class_1 = int(N * 0.3)
labels[:num_class_1] = 1
rate = "7 : 3"
'''
'''
# label 0 - 3 : label 1 - 7 비율
labels = np.zeros((N, 1), dtype=np.float32)
num_class_1 = int(N * 0.7)
labels[:num_class_1] = 1
rate = "3 : 7"
```

```

'''

# 5 : 5 비율
labels = np.zeros((N, 1), dtype=np.float32)
num_class_1 = N // 2
labels[:num_class_1] = 1
rate = "5 : 5"

# 섞어서 랜덤하게 배치
indices = np.random.choice(N, N, replace=False)
data = data[indices]
labels = labels[indices]

# 서브플롯 생성 (2 행 K//2 + 1 열)
fig, axs = plt.subplots(2, (K + 1) // 2, figsize=(13, 9))
fig.suptitle(f'KNN with different k values, label 0=red triangle, 1=blue rectangle,
rate = {rate}')

# X, Y 범위 설정
x_min, x_max = 0, 100
y_min, y_max = 0, 100

# Grid 생성
xx, yy = np.meshgrid(np.arange(x_min, x_max, 1),
                     np.arange(y_min, y_max, 1))

grid_points = np.c_[xx.ravel(), yy.ravel()]

# 각 k에 대해 예측 및 시각화
for k in range(1, K + 1):
    # KNN 모델 생성 및 훈련
    knn = cv2.ml.KNearest_create()
    knn.train(data, cv2.ml.ROW_SAMPLE, labels)

    # 모든 grid_points에 대해 예측
    _, results, _, _ = knn.findNearest(grid_points.astype(np.float32), k)
    results = results.reshape(xx.shape)

    ax = axs[(k - 1) // ((K + 1) // 2), (k - 1) % ((K + 1) // 2)]
    ax.contourf(xx, yy, results, alpha=0.3, levels=[-1, 0, 1], colors=['red',
'blue'])

    # 데이터 포인트 시각화
    red_triangles = data[labels.ravel() == 0]
    blue_squares = data[labels.ravel() == 1]

    ax.scatter(red_triangles[:, 0], red_triangles[:, 1], 200, 'r', '^')
    ax.scatter(blue_squares[:, 0], blue_squares[:, 1], 200, 'b', 's')

    ax.set_xlim(x_min, x_max)

```



```

ax.set_ylim(y_min, y_max)
ax.set_title(f'k={k}')
ax.grid(True)

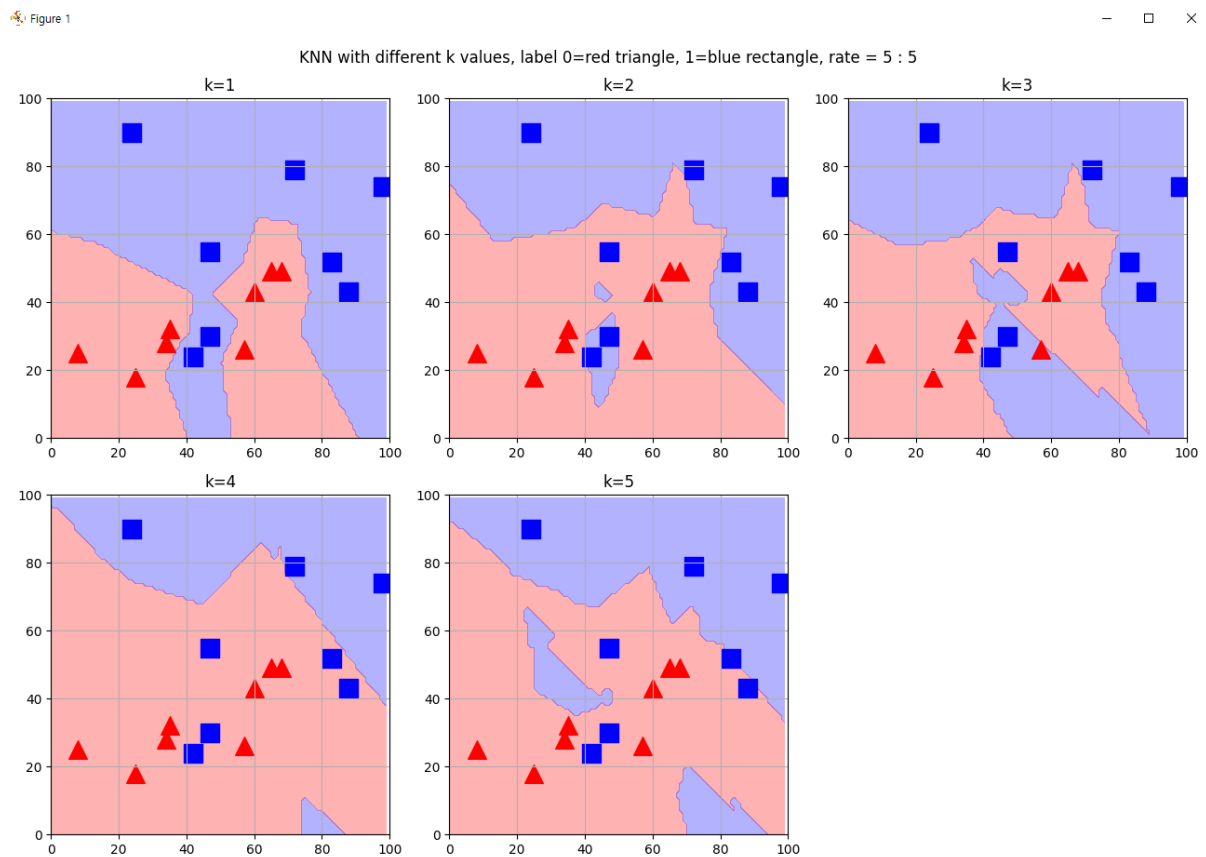
# 숨겨진 서브플롯 제거
if K % 2 != 0:
    fig.delaxes(axes[1, (K) // 2])

plt.tight_layout()
plt.show()

```

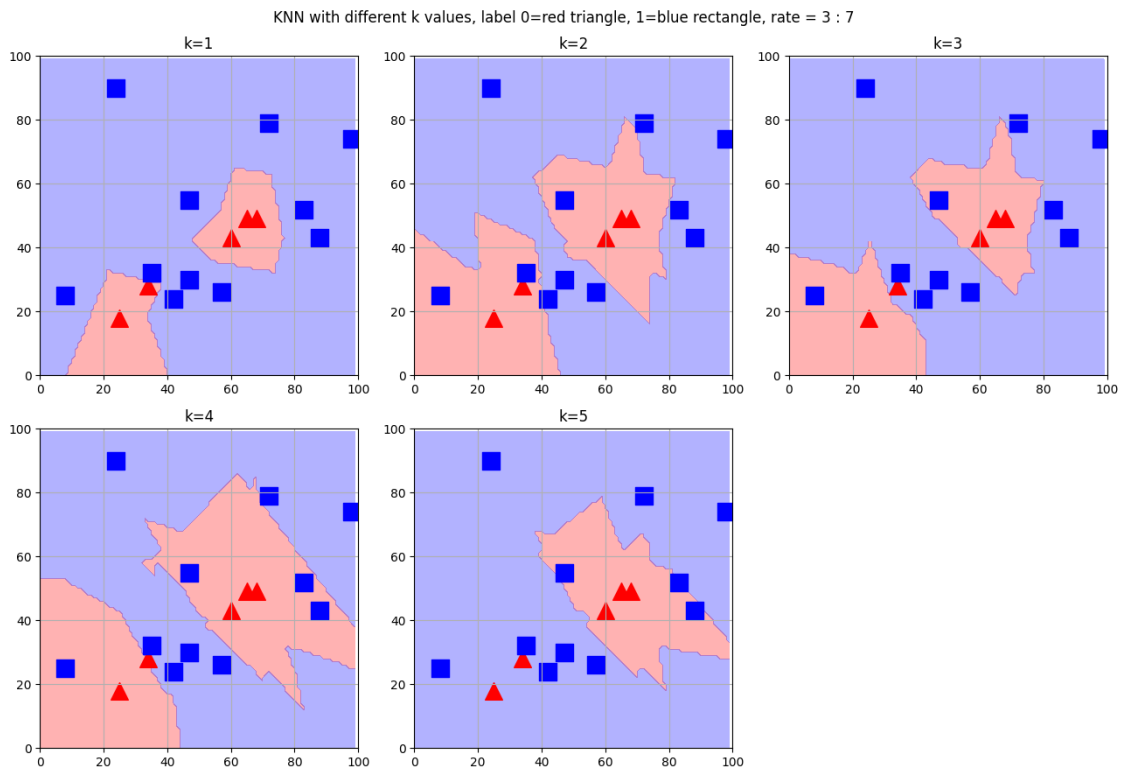
3. 결과

- 실행 초기 상태(5:5 비율)



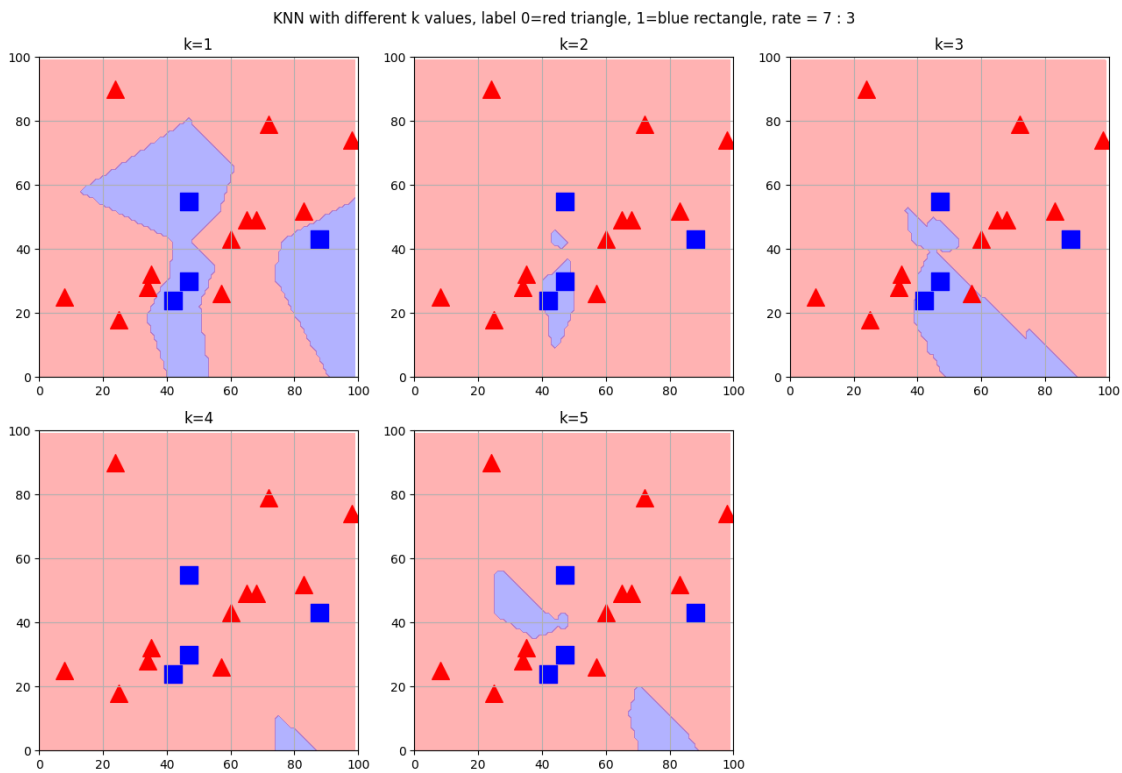
- 3:7 비율

Figure 1



- 7:3 비율

Figure 1



4. 최종 결론

- (0, 0)부터 (100, 100) 까지의 좌표를 범위로 지정.
- 각 좌표에서 k 값에 따라 knn을 수행시켜 해당 좌표가 어느 label을 갖게 되는지 실험.
- 7:3 비율과 3:7 비율의 결과를 비교해 볼 때 label=0 으로 나타나는 범위가 비교적 넓음.
- k가 짝수일 때 외부 요소가 knn 모델의 결정 시에 영향을 끼칠 수 있겠지만, 개인적인 의견으로는 label의 순서가 앞인 경우 더 우선시 되는 것 같다고 여겨짐.
- k = 1~5 까지의 현상을 하나의 pyplot로 표현.
- 7:3, 3:7, 5:5 의 비율을 코드의 수정을 통해 시각화 가능.

SVM1(임주형)

내가 잘한 점, 차별화 된 점

: 트랙바의 갱신에 따라 달라지는 결과 화면을 즉각적으로 보일 수 있게 하였다.

1. 문제 요약

SVM Introduction 1.2절의 RBF(Radial Basic Function) Kernel 참조해 gamma에 따른 분류 특성을 보여주는 사례와 같은 프로그램의 작성하라.

- 단계 1: gamma에 따라 몇 개의 그림을 한 화면에 matplotlib로 도시
- 단계 2: 가능한 클래스를 2~4개 정도 선택 가능하도록 설계 바람.

----- 선택 사항 -----

- 단계 3: trackbar로 감마를 제어 (감마 값은 화면에서 문자로 출력)
- 단계 4: 트랙 바를 하나 더 추가하여 C 값의 설정도 바꿀 수 있음(화면에 값 출력)
- 단계 5: 트랙 바 추가하여 커널 함수를 고를 수 있었으면 좋겠음(2.1절 사례처럼)

3. 코드

```
# 03 조 ( 임주형,이세비,최하은 )

"""
SVM1 - gamma 에 따른 분류 특성을 보여주는 사례와 같은 프로그램의 작성
=> gamma, C, kernel 에 따라 그림을 화면에 matplotlib로 도시
"""

import cv2
import numpy as np
import matplotlib.pyplot as plt

# 랜덤한 네 개의 클래스 데이터 생성
mean1 = [2, 2] # 클래스 1 평균값
cov1 = [[2, 0], [0, 2]] # 클래스 1 x, y 축에 퍼져있는 정도
data1 = np.random.multivariate_normal(mean1, cov1, 100)
label1 = np.zeros((100, 1)) # 해당 레이블에 적용되는 데이터는 0의 레이블을 가짐

mean2 = [6, 6] # 클래스 2 평균값
cov2 = [[2, 0], [0, 2]]
data2 = np.random.multivariate_normal(mean2, cov2, 100)
label2 = np.ones((100, 1)) # 해당 레이블에 적용되는 데이터는 1의 레이블을 가짐
```

```

mean3 = [10, 2] # 클래스 3 평균값
cov3 = [[2, 0], [0, 2]]
data3 = np.random.multivariate_normal(mean3, cov3, 100)
label3 = 2 * np.ones((100, 1)) # 해당 레이블에 적용되는 데이터는 2의 레이블을 가짐

mean4 = [2, 10] # 클래스 4 평균값
cov4 = [[2, 0], [0, 2]]
data4 = np.random.multivariate_normal(mean4, cov4, 100)
label4 = 3 * np.ones((100, 1)) # 해당 레이블에 적용되는 데이터는 3의 레이블을 가짐

# 모든 데이터와 레이블을 하나로 합침
all_data = [data1, data2, data3, data4]
all_labels = [label1, label2, label3, label4]

# 초기 감마 값 설정
gamma = 1.0
# 초기 C 값 설정
C = 1
# 초기 커널 함수 설정
kernel_type = cv2.ml.SVM_RBF
kernel_str = "RBF"
# 선택한 클래스의 수 설정
num_classes = 4

# 트랙바 콜백 함수 (감마)
def on_gamma_trackbar(val):
    global gamma
    gamma = val / 100.0 * 99.99 + 0.01 # 감마 값을 0.01에서 100 사이의 값으로 변환
    update_plot()

# 트랙바 콜백 함수 (C)
def on_c_trackbar(val):
    global C
    C = val / 100.0 * 99.99 + 0.01 # C 값을 0.01에서 100 사이의 값으로 변환
    update_plot()

# 트랙바 콜백 함수 (커널 함수)
def on_kernel_trackbar(val):
    global kernel_type
    global kernel_str
    if val == 0: # 선형 커널
        kernel_type = cv2.ml.SVM_LINEAR
        kernel_str = "LINEAR"
    elif val == 1: # 다항 커널
        kernel_type = cv2.ml.SVM_POLY
        kernel_str = "POLY"
    elif val == 2: # RBF 커널

```

```

        kernel_type = cv2.ml.SVM_RBF
        kernel_str = "RBF"
    elif val == 3: # 시그모이드 커널
        kernel_type = cv2.ml.SVM_SIGMOID
        kernel_str = "SIGMOID"
    update_plot()

# 트랙바 콜백 함수 (클래스 수)
def on_class_trackbar(val):
    global num_classes
    num_classes = val + 2 # 최소 2 개의 클래스를 선택하게 함
    update_plot()

# 결정 경계 업데이트 함수
def update_plot():
    # 선택한 클래스 데이터 병합
    selected_data = np.vstack(all_data[:num_classes])
    selected_labels = np.vstack(all_labels[:num_classes])

    # SVM 모델 생성
    svm = cv2.ml.SVM_create()
    svm.setKernel(kernel_type) # 커널 함수 설정

    # 모델 유형 설정 - C-SVC : 클래스 간의 간격을 최대화하는 선형 분류 수행
    svm.setType(cv2.ml.SVM_C_SVC)
    svm.setC(C) # C 설정
    svm.setGamma(gamma) # gamma 값 적용

    if kernel_type == cv2.ml.SVM_POLY:
        svm.setDegree(3) # 다항 커널의 degree 설정
    if kernel_type == cv2.ml.SVM_SIGMOID or kernel_type == cv2.ml.SVM_POLY:
        svm.setCoef0(0.0) # 다항 및 시그모이드 커널의 coef0 설정

    # SVM 모델 훈련
    svm.train(selected_data.astype(np.float32), cv2.ml.ROW_SAMPLE,
selected_labels.astype(np.int32))

    # 결정 경계 생성
    x1_min, x1_max = selected_data[:, 0].min() - 1, selected_data[:, 0].max() + 1
    x2_min, x2_max = selected_data[:, 1].min() - 1, selected_data[:, 1].max() + 1

    # x1 및 x2 의 최소에서 최대까지 0.1 간격으로 그리드 포인트를 생성
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, 0.1),
        np.arange(x2_min, x2_max, 0.1))

    # xx1 과 xx2 를 펼쳐서 각 포인트를 하나의 특성 벡터로 생성
    mesh_data = np.c_[xx1.ravel(), xx2.ravel()]

```

```

# svm 모델을 사용해서 경계를 생성
_, decision_values = svm.predict(mesh_data.astype(np.float32))

# xx1의 형태 재구성 -> 결정 경계의 시각화 가능
decision_values = decision_values.reshape(xx1.shape)

# 시각화
plt.clf() # plt 갱신
plt.contourf(xx1, xx2, decision_values, alpha=0.5) # 결정 경계의 등고선 시각화

# 선택한 클래스 데이터 시각화
colors = ['red', 'blue', 'green', 'purple']
labels = ['Class 1', 'Class 2', 'Class 3', 'Class 4']
for i in range(num_classes):
    plt.scatter(all_data[i][:, 0], all_data[i][:, 1], color=colors[i],
label=labels[i])

plt.title(f'SVM with {kernel_str} Kernel (C={C:.2f}, gamma={gamma:.2f})')
plt.legend()

# Matplotlib 그래프를 이미지로 변환
plt.savefig('tmp_plot.png')

# OpenCV로 이미지 불러오기
img = cv2.imread('tmp_plot.png')
cv2.imshow('SVM with Kernel', img)

# 초기화 및 트랙바 생성
cv2.namedWindow('SVM with Kernel')
cv2.createTrackbar('Gamma', 'SVM with Kernel', 50, 100, on_gamma_trackbar)
cv2.createTrackbar('C', 'SVM with Kernel', 100, 1000, on_c_trackbar)
cv2.createTrackbar('Kernel', 'SVM with Kernel', 2, 3,
on_kernel_trackbar) # 0: Linear, 1: Polynomial, 2: RBF, 3: Sigmoid
cv2.createTrackbar('Classes', 'SVM with Kernel', 2, 2, on_class_trackbar) # 2: 최소
2개, 최대 4개 클래스 선택

# 트랙바 초기값 설정
cv2.setTrackbarPos('Kernel', 'SVM with Kernel', 2)
cv2.setTrackbarPos('Classes', 'SVM with Kernel', 0)

# 초기 플롯 생성
update_plot()

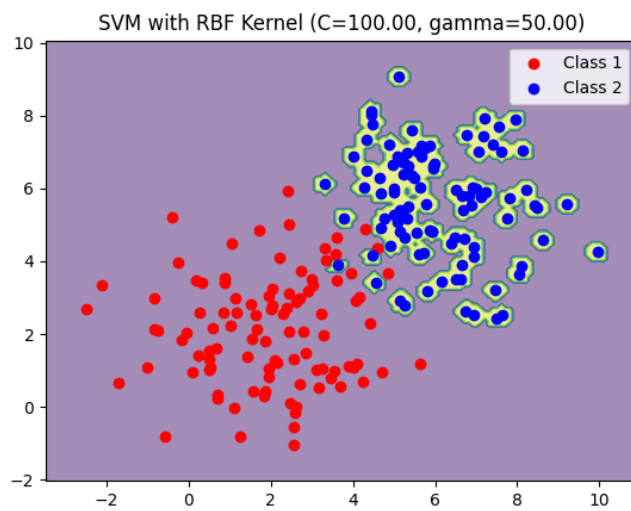
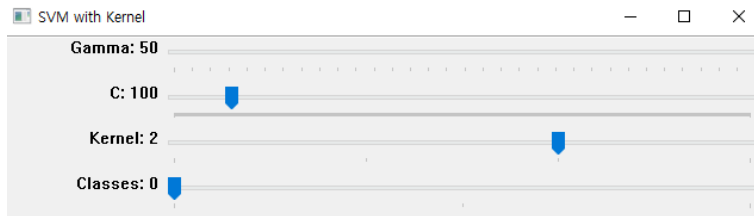
# 트랙바 이벤트 처리
while True:
    key = cv2.waitKey(1) & 0xFF
    if key == 27: # ESC 키를 누르면 종료
        break

```

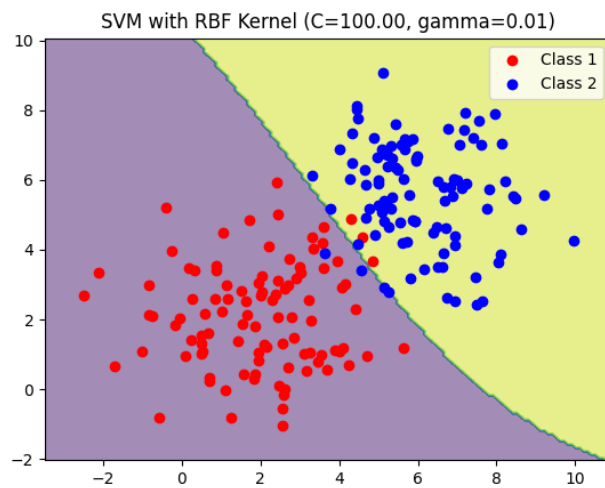
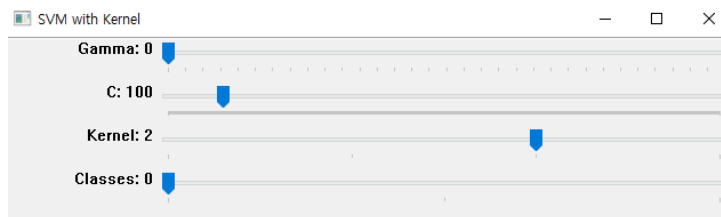
```
cv2.destroyAllWindows()
```

4. 결과

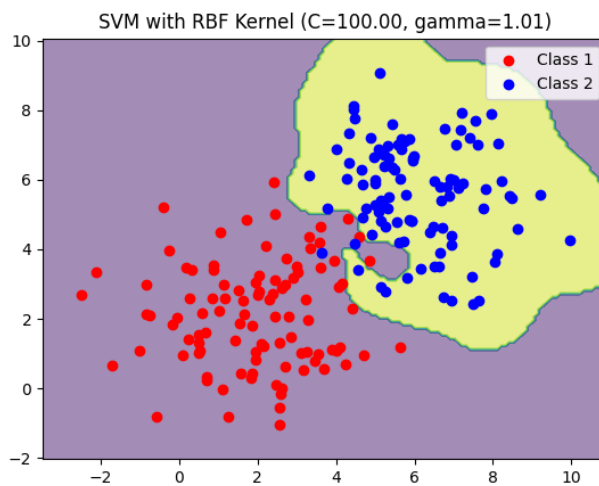
- 실행 초기 상태($\gamma = 50$, $C = 100$, Kernel = RBF)



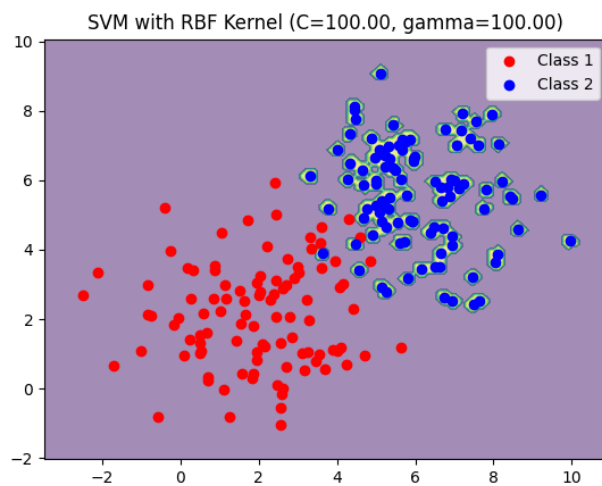
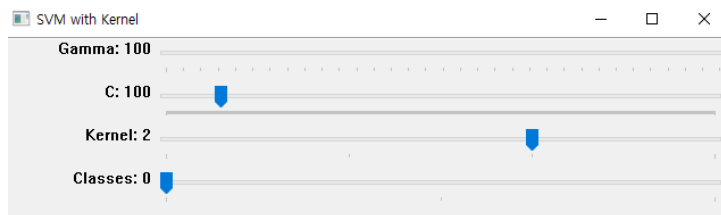
- $\gamma = 0.01$



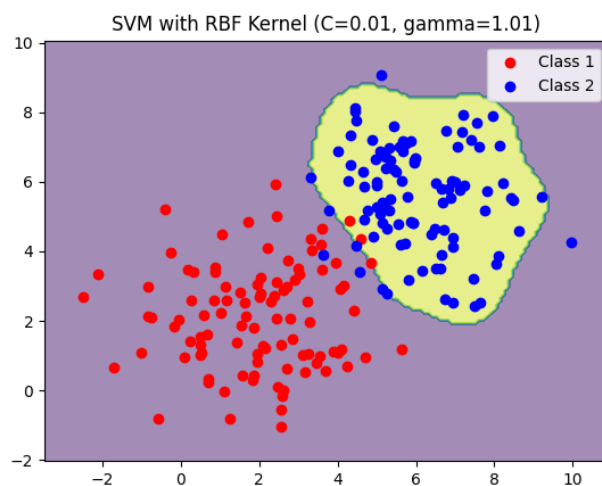
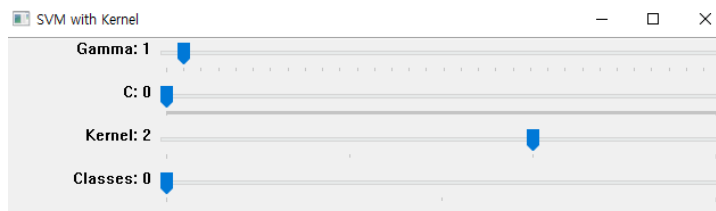
- $\gamma = 1$



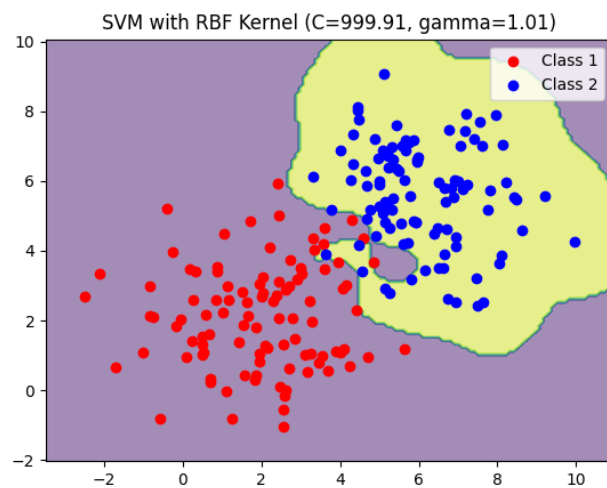
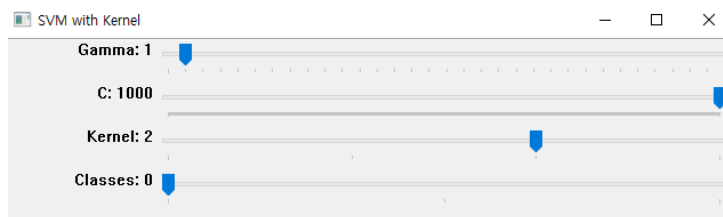
- $\gamma = 100$



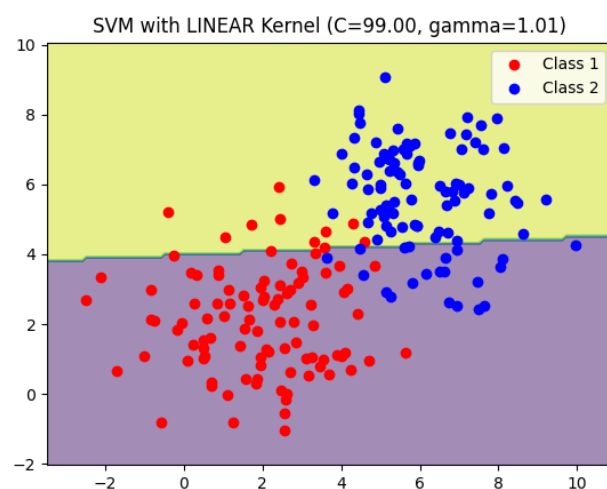
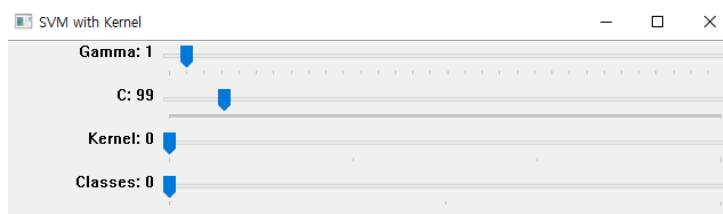
- $C = 0.01$



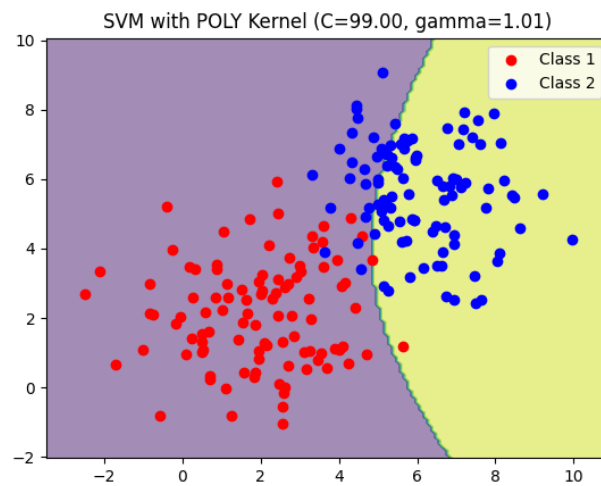
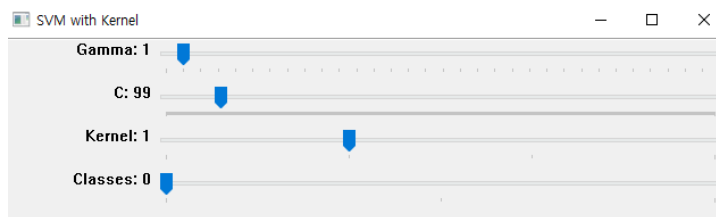
- $C = 1000$



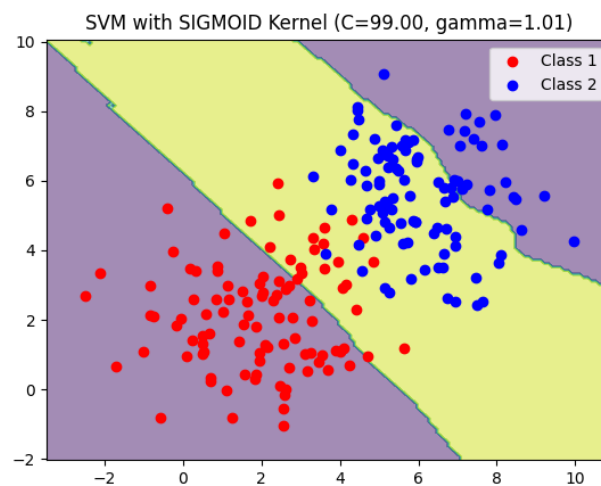
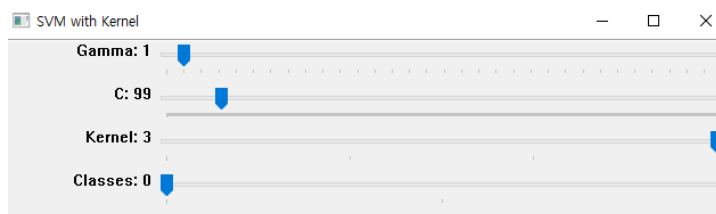
- Kernel = Linear



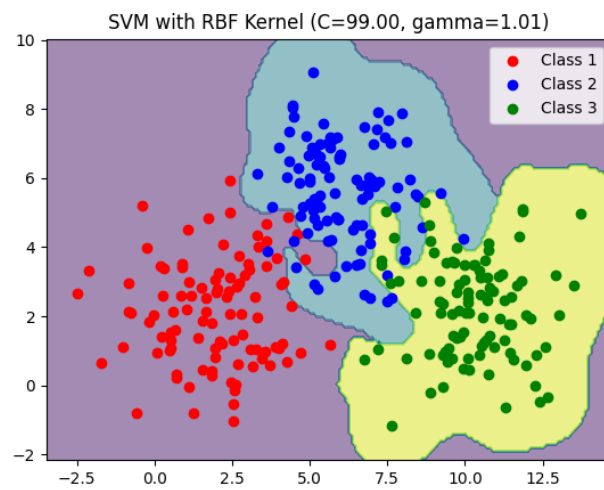
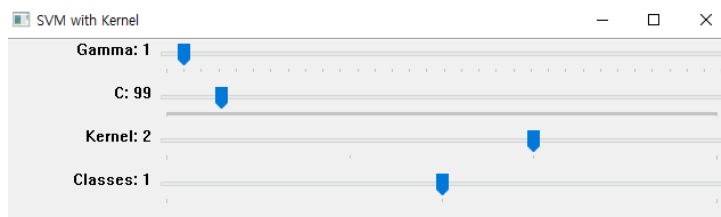
- Kernel = Polynomial



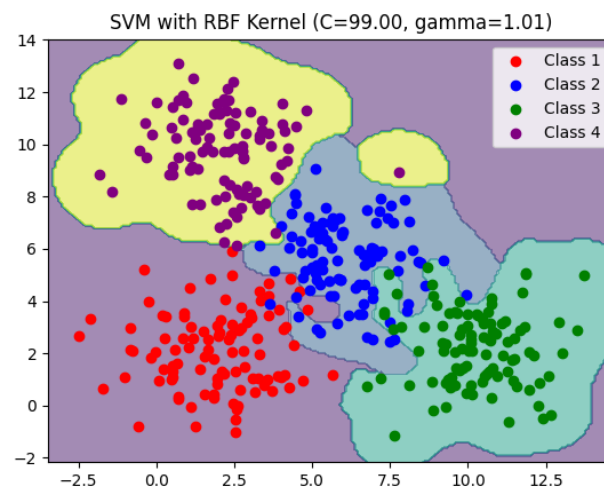
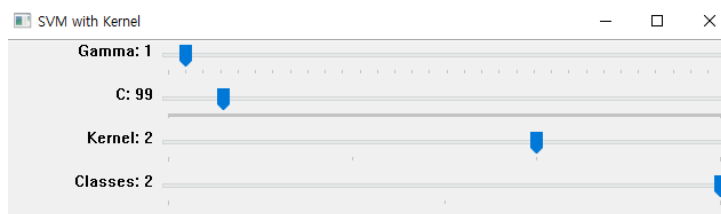
- Kernel = Sigmoid



- Class 3



- Class 4



5. 최종 결론

- 단계 1: 여러 값을 수정하면서 갱신된 상태의 그림을 볼 수 있음 (사진을 저장하고 이를 보여주는 메커니즘).
- 단계 2: 클래스를 트랙 바를 통해 수정 가능
- 단계 3: gamma 수정 가능 및 화면 출력 완료.
- 단계 4: C 수정 가능 및 화면 출력 완료.
- 단계 5: 0: Linear, 1: Polynomial, 2: RBF, 3: Sigmoid

SVM2(이세비)

내가 잘한 점, 차별화 된 점

: 난수 생성 실험(python 코드 작성)을 통해 시드 값에 따라 생성되는 난수를 직접 확인했다. 이 결과로 결과를 통해 시드 값의 영향력을 명확히 설명했다. 더하여 SVM모델의 정확도 차이가 나지 않는다는 점을 발견했으며, 이에 대한 이유를 자세히 설명했다.

1. 문제 요약

5절의 SVM_02 실험과의 비교에서 차이가 발생하는 원인 규명해보자.

의문사항: SVM_03 예제는99.2% 정확도인데, SVM_02 예제에서는99.0% 정확도이다. 의심점: shuffle 과정에서 서로 다른 데이터를 학습데이터와 테스트데이터로 쓸 수도 있다. 반론: RandomState(1234)의seed 값이같다. 이는 KNN에서도 계속 써오던 값이다.

- 그동안의 추정: seed가 같으면 난수 발생 패턴도 같다. True or False?

Seed가 난수 발생의 재연을 하는가 하지 않는가?

2. 결론

seed가 같으면 난수 발생 패턴도 같다. 이 때문에 결과 값도 동일한 정확도인 99.0%로 측정된다.

3. seed 가 같을 때, 난수 발생 패턴을 확인하는 실험

1) 실험 개요

난수 생성기의 재현성을 확인하는 것이다. 이를 통해 동일한 시드(seed) 값을 사용했을 때 난수 생성 패턴이 동일하게 유지되는지, 그리고 다른 시드 값을 사용했을 때 난수 생성 패턴이 달라지는지 확인하고자 한다.

2)실험 방법

두 가지 경우에 대해 실험 수행:

- ㄱ. 같은 시드 값을 사용하여 난수를 생성하고, 두 번의 실행 결과를 비교
- ㄴ. 다른 시드 값을 사용하여 난수를 생성하고, 두 번의 실행 결과를 비교

실험에 사용된 코드는 다음과 같다.

- 같은 시드 값을 사용하는 경우

```
import numpy as np

# 같은 시드를 사용하여 난수를 생성
seed = 1234
random_state1 = np.random.RandomState(seed)
random_numbers1 = random_state1.rand(5)

random_state2 = np.random.RandomState(seed)
random_numbers2 = random_state2.rand(5)

print("같은 시드를 사용한 첫 번째 난수 배열:", random_numbers1)
print("같은 시드를 사용한 두 번째 난수 배열:", random_numbers2)

# 결과 비교
same_seed_same_output = np.array_equal(random_numbers1, random_numbers2)
print("같은 시드를 사용했을 때 결과가 동일한가요?", same_seed_same_output)
```

- 다른 시드 값을 사용하는 경우

```
import numpy as np

# 다른 시드를 사용하여 난수를 생성
random_state3 = np.random.RandomState(5678)
random_numbers3 = random_state3.rand(5)

random_state4 = np.random.RandomState(8765)
random_numbers4 = random_state4.rand(5)

print("다른 시드를 사용한 첫 번째 난수 배열:", random_numbers3)
print("다른 시드를 사용한 두 번째 난수 배열:", random_numbers4)

# 결과 비교
different_seed_different_output = np.array_equal(random_numbers3,
random_numbers4)
print("다른 시드를 사용했을 때 결과가 다른가요?", different_seed_different_output)
```

3) 실험 결과

- 같은 시드를 사용한 경우:

```
같은 시드를 사용한 첫 번째 난수 배열: [0.19151945 0.62210877 0.43772774 0.78535858 0.77997581]
같은 시드를 사용한 두 번째 난수 배열: [0.19151945 0.62210877 0.43772774 0.78535858 0.77997581]
같은 시드를 사용했을 때 결과가 동일한가요? True
```

- 다른 시드를 사용한 경우:


```
다른 시드를 사용한 첫 번째 난수 배열: [0.48932698 0.05933244 0.36620243 0.51886544 0.59822501]
다른 시드를 사용한 두 번째 난수 배열: [0.03278543 0.80595138 0.66984421 0.76030572 0.27496366]
다른 시드를 사용했을 때 결과가 동일한가요? False
```

4) 결론

실험 결과, 동일한 시드 값을 사용하여 난수를 생성했을 때 항상 동일한 난수 배열이 생성됨을 확인할 수 있었다. 이는 난수 생성기의 재현성이 보장됨을 의미한다. 반면, 다른 시드 값을 사용했을 때 생성된 난수 배열은 서로 달랐다. 따라서, 시드 값이 같으면 난수 생성 패턴도 동일하다.

4. SVM_02와 SVM_03의 정확도

1) svm_02

아래와 같이 $C=12.5$, $\gamma=0.50625$ 일 때, 정확도 99.00%가 출력된다.

```
# 6) SVM 모델 객체를 하나 생성한다.
print('Training SVM model ...')
model = svm_init(C=12.5, gamma=0.50625)
```

```
hog descriptor size=144
5.1) 학습데이터: <class 'numpy.ndarray'> (4500, 144) float32
5.2) 학습레이블: <class 'numpy.ndarray'> (4500,)
Training SVM model ...
Evaluating model ...
Percentage Accuracy: 99.00 %
```

2) svm_03

아래의 코드와 같이 $C=12.5$ 에 다양한 γ 의 경우별로 나누어 정확도를 측정했다. 이 때의 결과 중 SVM_02 예제와의 비교를 위해 같은 조건 즉, $C=12.5$, $\gamma=0.50625$ 에 해당하는 accuracy를 확인해보면 99.00%로 동일한 정확도를 출력한다. 더하여 $C=12.5$, $\gamma=0.3$ 일 때, 가장 큰 정확도인 99.40%를 출력하는 것을 확인할 수 있다.

```
# SVM_02 예제와 비교해 보기 위한 설정, 1 회 검증용
gamma_list = [0.1, 0.3, 0.50625, 0.7, 0.9, 1.1, 1.3, 1.5]
C_list = [12.5]
```

```
Training SVM model ...
C=12.5, gamma=0.10000: accuracy=99.00
C=12.5, gamma=0.30000: accuracy=99.40
C=12.5, gamma=0.50625: accuracy=99.00
C=12.5, gamma=0.70000: accuracy=98.80
C=12.5, gamma=0.90000: accuracy=98.60
C=12.5, gamma=1.10000: accuracy=98.20
C=12.5, gamma=1.30000: accuracy=98.40
C=12.5, gamma=1.50000: accuracy=98.20
```

정확도 순으로 소팅해서 출력한 결과입니다.

```
C=12.5, gamma=0.3: accuracy=99.40
C=12.5, gamma=0.1: accuracy=99.00
C=12.5, gamma=0.5: accuracy=99.00
C=12.5, gamma=0.7: accuracy=98.80
C=12.5, gamma=0.9: accuracy=98.60
C=12.5, gamma=1.3: accuracy=98.40
C=12.5, gamma=1.1: accuracy=98.20
C=12.5, gamma=1.5: accuracy=98.20
```

5. 최종 결론

동일한 시드 값(1234)을 사용하여 난수를 생성하였기에 SVM_02와 SVM_03의 정확도는 99.00%로 동일하다. 이는 같은 난수를 생성하기 때문에 결과 값도 동일하게 나타날 수 있었다. 따라서 정확도가 다르게 측정된 것이 아니라 조건이 다른 경우에 대해서만 다른 결과를 얻을 수 있었다. 특히 SVM_03에서는 C=12.5 및 gamma=0.3 조합에서 최고 정확도인 99.4%를 기록했다는 것을 확인할 수 있다.