3주차 강의 자료

요약

- ❖ 분류 기법 소개
  - 분류 기법의 정의
  - 분류 기법의 특징
  - 대표 분류 기법 소개 (KNN, Decision tree)
- ❖ KNN(K-Nearest Neighbor) 실습 – KNN 기반 KDD 데이터 분류
  - 데이터 로드 및 전처리
  - sklearn 라이브러리를 활용한 학습
  - 결과 확인
- ❖ 의사결정트리(Decision Tree) 실습 – 결정트리 기반 KDD 데이터 분류
  - 데이터 로드 및 전처리
  - sklearn 라이브러리를 활용한 학습
  - 결과 확인
- ❖ MNIST 데이터 분류 및 성능 측정
  - MNIST 데이터 분류
  - 성능 측정 방법

요약

- ❖ 클러스터링 분석
  - 클러스터링 분석이란?
  - 데이터간 유사도
  - 주요 클러스터링 기법
  - 클러스터간 거리 측정
- ❖ Dimensionality reduction
  - Principal Component Analysis(PCA)
- ❖ K-means Algorithm
  - 알고리즘 설명
  - Scikit-learn을 활용한 실습
- ❖ DBSCAN Algorithm
  - 알고리즘 설명
  - Scikit-learn을 활용한 실습
  - 과제 소개

분류

분류 (Classification)

❖ 머신러닝 기법 중 하나로, 기존의 관측치(label)가 있는 training data를 기반으로 학습하는 방법

- 사전에 Categorical 데이터 및 데이터의 label을 요구
- 지도학습으로 기존의 학습을 통하여, 새롭게 관측된 데이터의 category를 예측 가능

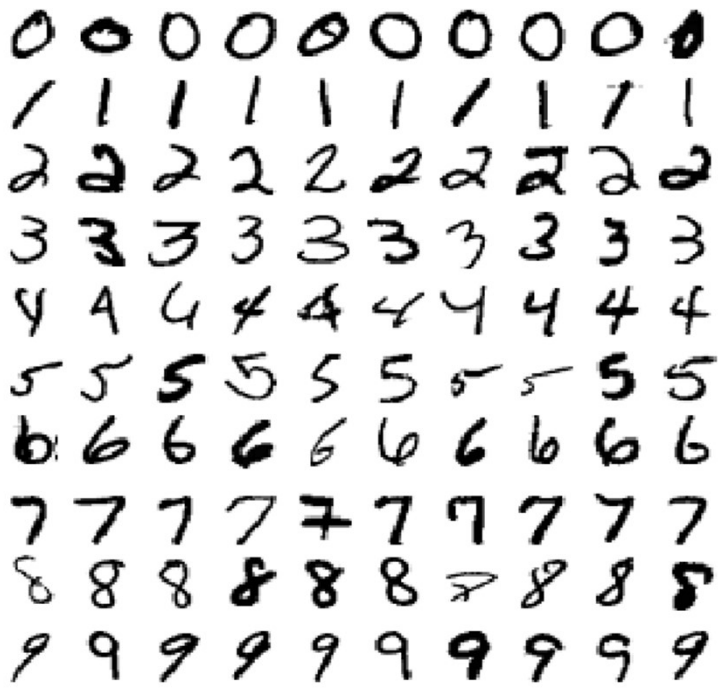
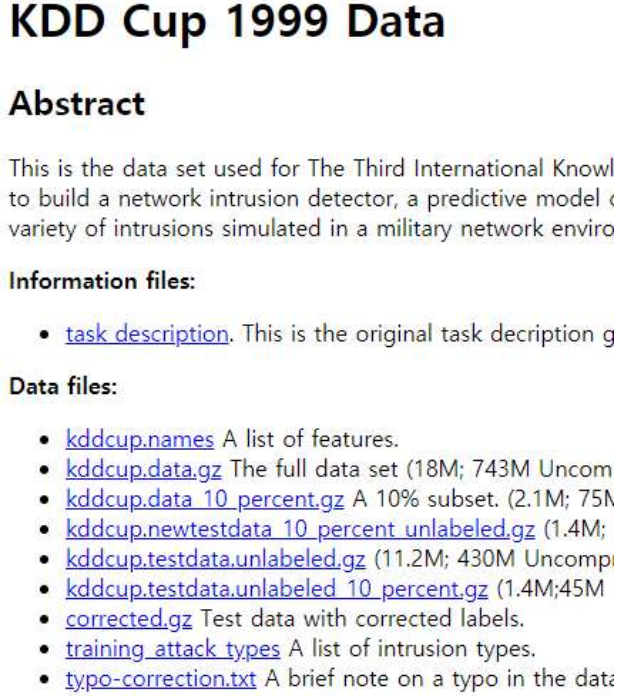


그림 3-1 MNIST 데이터셋에서 추출한 숫자 이미지



KDD Cup 1999 Data

Abstract

This is the data set used for The Third International Knowl
to build a network intrusion detector, a predictive model
variety of intrusions simulated in a military network enviro

Information files:

- task description. This is the original task decription g

Data files:

- kddcup.names A list of features.
- kddcup.data.gz The full data set (18M; 743M Uncom
- kddcup.data_10_percent.gz A 10% subset. (2.1M; 75M
- kddcup.newtestdata_10_percent_unlabeled.gz (1.4M;
- kddcup.testdata.unlabeled.gz (11.2M; 430M Uncompr
- kddcup.testdata.unlabeled_10_percent.gz (1.4M;45M
- corrected.gz Test data with corrected labels.
- training_attack_types A list of intrusion types.
- typo-correction.txt A brief note on a typo in the data

그림 3-2 KDD Cup Data 페이지

KNN (K-Nearest Neighbor)

❖ 분류 기법 중 하나로, 데이터를 거리가 가장 가까운 k개의 데이터를 참고하여 데이터를 분류하는 방법

- 데이터 간의 거리를 측정하기 위해, Euclidian Distance를 사용
- K값에 따른 성능 차이 존재 (일반적으로 총 데이터의 제곱근 값을 사용)

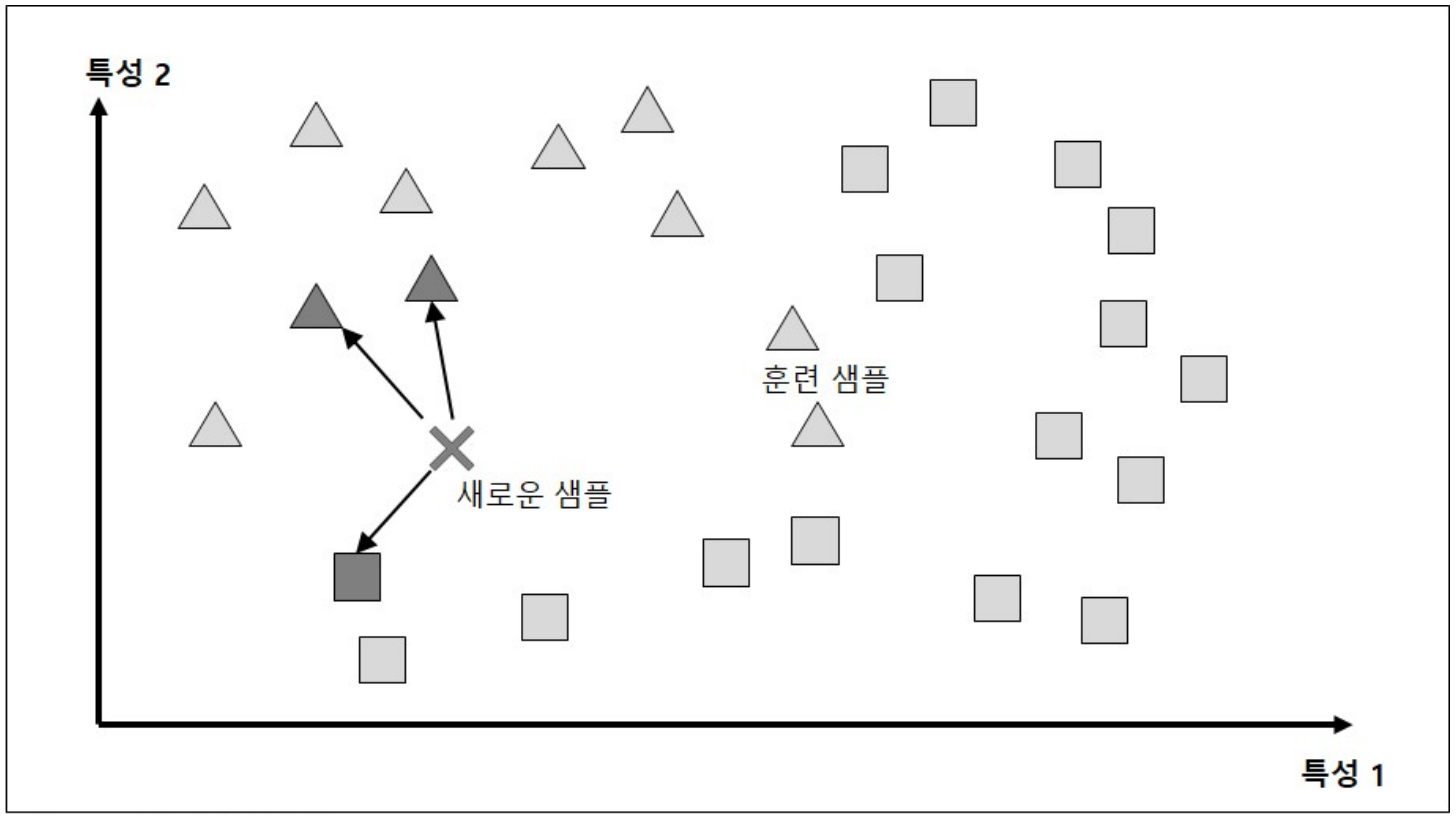


그림 1-15. 사례 기반 학습

KNN 기반 KDD 데이터셋 분류

❖ Dataset load를 위한 import 및 column 이름 설정

```
In [1] : import pandas
         from time import time

         col_names = ["duration", "protocol_type", "service", "flag", "src_bytes", "dst_bytes",
         "land", "wrong_fragment", "urgent", "hot",
                      "num_failed_logins", "logged_in", "num_compromised", "root_shell",
         "su_attempted", "num_root", "num_file_creations",
                      "num_shells", "num_access_files", "num_outbound_cmds", "is_host_login",
         "is_guest_login", "count", "srv_count",
                      "serror_rate", "srv_serror_rate", "rerror_rate", "srv_rerror_rate",
         "same_srv_rate", "diff_srv_rate",
                      "srv_diff_host_rate", "dst_host_count","dst_host_srv_count",
         "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
                      "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
         "dst_host_serror_rate", "dst_host_srv_serror_rate",
                      "dst_host_rerror_rate", "dst_host_srv_rerror_rate", "label"]
```

KNN 기반 KDD 데이터셋 분류

❖ 10percent dataset load 및 확인

- 학습 속도를 위해 10percent dataset을 사용
- names 옵션에 방금 입력한 col_names를 입력
- head 함수를 통해 데이터셋 샘플 조회

```
In [2] : kdd_data_10percent = pandas.read_csv("./kddcup.data_10_percent_corrected", names =
         col_names)
         kdd_data_10percent.head()
```

Out [2] :

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_coun
0	0	tcp	http	SF	181	5450	0	0	0	0	...	9
1	0	tcp	http	SF	239	486	0	0	0	0	...	19
2	0	tcp	http	SF	235	1337	0	0	0	0	...	29
3	0	tcp	http	SF	219	1337	0	0	0	0	...	39
4	0	tcp	http	SF	217	2032	0	0	0	0	...	49

KNN 기반 KDD 데이터셋 분류

❖ Class별 데이터셋 개수 조회

- 딕셔너리 및 value_counts 함수를 활용하여 클래스 별 데이터 개수 조회

```
In [3] :  kdd_data_10percent['label'].value_counts()
```

```
Out [3] : smurf.              280790
          neptune.            107201
          normal.              97278
          back.                 2203
          satan.                1589
          ipsweep.              1247
          portsweep.            1040
          warezclient.          1020
          teardrop.              979
          pod.                   264
          nmap.                  231
          guess_passwd.           53
          buffer_overflow.        30
          land.                   21
          warezmaster.            20
          imap.                   12
          rootkit.                10
          loadmodule.              9
          ftp_write.               8
          multihop.                7
          phf.                     4
          perl.                    3
          spy.                     2
```

KNN 기반 KDD 데이터셋 분류

❖ Categorical data 변환

- 학습에 사용할 categorical data를 조회

```
In [4] :  print(kdd_data_10percent.dtypes)
```

```
Out [4] :  duration                int64
           protocol_type           object
           service                 object
           flag                    object
           src_bytes               int64
           dst_bytes               int64
           land                    int64

           ...
           ...

           dst_host_srv_serror_rate    float64
           dst_host_rerror_rate        float64
           dst_host_srv_rerror_rate    float64
           label                   object
```

KNN 기반 KDD 데이터셋 분류

❖ Categorical data 변환

- pandas의 factorize 함수를 사용하여, categorical data를 정수화

```
In [5] :  kdd_data_10percent['protocol_type'], protocols=
          pandas.factorize(kdd_data_10percent['protocol_type'])
          kdd_data_10percent['service'], services =
          pandas.factorize(kdd_data_10percent['service'])
          kdd_data_10percent['flag'], flags    = pandas.factorize(kdd_data_10percent['flag'])
          kdd_data_10percent['label'], attacks = pandas.factorize(kdd_data_10percent['label'])

          print(kdd_data_10percent.dtypes)
```

```
Out [5] :  duration                      int64
           protocol_type                 int64
           service                       int64
           flag                          int64
           src_bytes                     int64
           dst_bytes                     int64
           land                          int64

           ...
           ...

           dst_host_srv_serror_rate      float64
           dst_host_rerror_rate          float64
           dst_host_srv_rerror_rate      float64
           label                         int64
```

KNN 기반 KDD 데이터셋 분류

❖ Training set Test set 분류

- train_test_split 함수 기반의 train set / test set 분류

In [6] :
```
from sklearn.model_selection import train_test_split

# 기존에 정의했던 col_names를 사용하여 label과 features를 분리
X = kdd_data_10percent[col_names[:len(col_names)-1]]
Y = kdd_data_10percent['label'].copy()

# Train set과 Test set을 8:2 비율로 분리
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
random_state=123)
```

In [7] :
```
# Train set label 조회
Y_train.value_counts()
```

In [8] :
```
# Test set label 조회
Y_test.value_counts()
```

KNN 기반 KDD 데이터셋 분류

❖ 데이터 셋 축소

- 노트북 사용 시, 학습에 상당한 시간이 걸리므로 수업에서는 데이터 셋 10만개만을 사용

In [9] :
```
kdd_data_10percent = kdd_data_10percent[:100000]
```

In [10] :
```
# Train set label 조회
Y_train.value_counts()
```

In [11] :
```
# Test set label 조회
Y_test.value_counts()
```

KNN 기반 KDD 데이터셋 분류

❖ KNN 분류 모델 학습

- n = 5로 학습

In [12]:
```
from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier(n_neighbors = 5)
t0 = time()
clf.fit(X_train, Y_train)
tt = time() - t0
print ("Classifier trained in {} seconds.".format(round(tt, 3)))
```

Out [12]:
```
Classifier trained in 1.763 seconds.
```

KNN 기반 KDD 데이터셋 분류

❖ KNN 분류 모델 학습

- n = 5로 학습 및 결과 확인

In [13]:
```
from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier(n_neighbors = 5)
t0 = time()
clf.fit(X_train, Y_train)
tt = time() - t0
print ("Classifier trained in {} seconds.".format(round(tt, 3)))
```

Out [13]:
```
Classifier trained in 1.763 seconds.
```

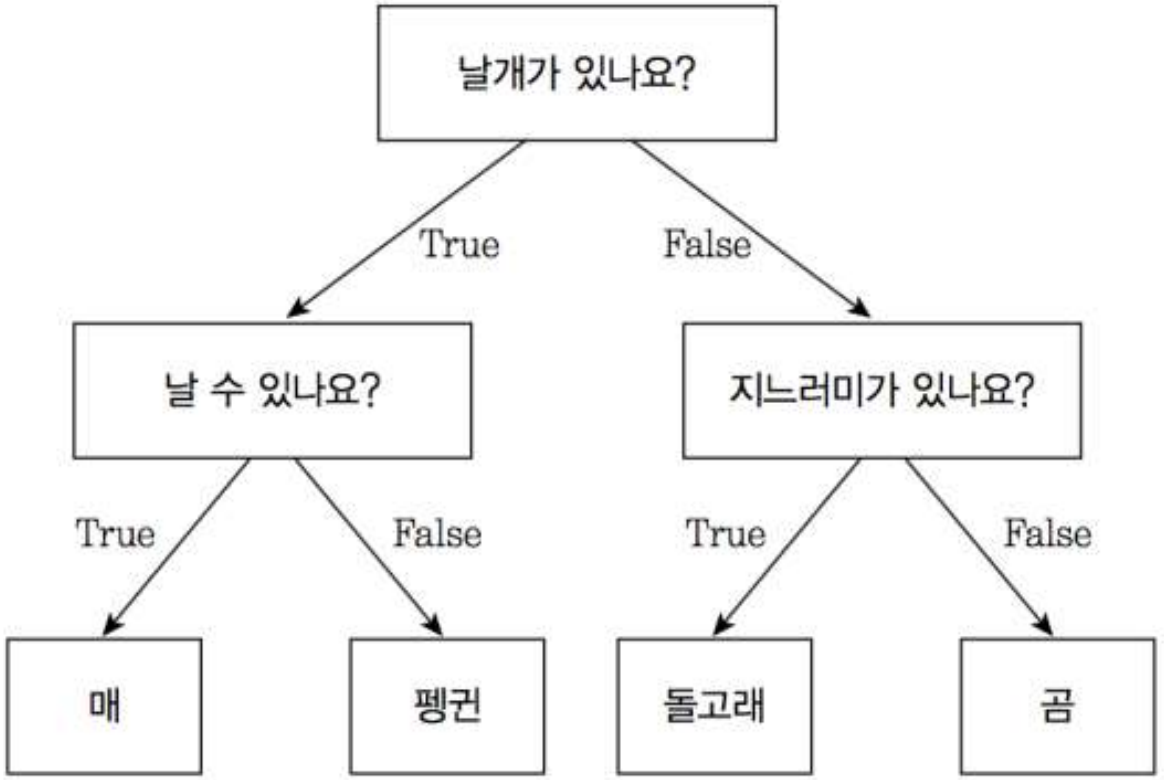
In [14]:
```
print ("Training Score:", clf.score(X_train, Y_train))
print ("Test_Score: ", clf.score(X_test, Y_test))
```

Out [14]:
```
Training Score: 0.9980125
Test_Score:  0.9969
```

의사결정트리 (Decision tree)

❖ Root 노드부터 시작하여, attributes에 따라 true, false를 나누고 데이터를 구분하는 학습 기법

- 데이터를 가장 잘 구분할 수 있는 질문을 기준으로 노드를 나눈 뒤, 이를 recursive하게 반복



의사결정트리 기반 KDD 데이터 분류

❖ sklearn의 DecisionTreeClassifier 함수를 사용

- 기존 KNN 실습에 이어서 데이터 셋을 사용한다

In [15] :

```
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier(random_state=156)

trained_model = clf.fit(X_train, y_train)

print ("Training Score:", trained_model.score(X_train, y_train))
print ("Test_Score: ", trained_model.score(X_test, y_test))
```

Out [15] :

```
Training Score: 1.0
Test_Score:  0.999
```

MNIST 데이터 세트 분석

❖ MNIST 데이터 셋

- 고등학생과 미국 인구 조사국 직원들이 작성한 7,000 개의 작은 숫자 이미지

그림 3-1 MNIST 데이터셋에서 추출한 숫자 이미지

MNIST 데이터 세트 분석

❖ MNIST 데이터 셋

- 사이킷런의 헬퍼 함수를 이용한 데이터셋 다운로드

In [1] :
```
import numpy as np
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1, cache=True)
mnist
```

Out [1] :
```
{'data': pixel1 pixel2 pixel3 pixel4 pixel5 pixel6 pixel7 pixel8 pixel9 ₩
0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
3 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
... ... ... ... ... ... ... ... ... ...
69995 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
69996 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
69997 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
69998 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
69999 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
...
```

MNIST 데이터 세트 분석

❖ MNIST 데이터 셋

- 딕셔너리 구조
  - 데이터 셋을 설명하는 DESCR
  - 셈플이 하나의 행, 특성이 하나의 열로 구성된 배열을 가진 data 키
  - 레이블 배열을 담고 있는 target 키

```
In [2] :  X, y = mnist["data"], mnist["target"]
          X.shape
```

```
Out [2] :  (70000, 784)
```

```
In [3] :  y.shape
```

```
Out [3] :  (70000,)
```

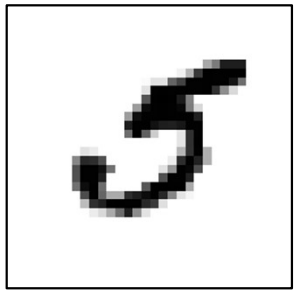
MNIST 데이터 세트 분석

❖ MNIST 데이터 셋

▪ 샘플 한 개를 이미지화

- 샘플의 특성 벡터를 추출해서 28x28배열로 크기를 바꾸고 맷플롯립의 imshow() 함수를 사용

In [4] :
```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
X = np.array(X)
y = np.array(y)
some_digit = X[35]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap = matplotlib.cm.binary, interpolation="nearest")
plt.axis("off")
plt.show()
```



- 실제 레이블 확인

In [5] :
```
y[36000]
```

Out [5] : 5.0

MNIST 데이터 세트 분석

❖ 테스트 세트 생성

- MNIST 데이터 세트는 이미 훈련 세트와 테스트 세트로 나누어 놓음
  - 훈련 세트: 앞쪽 60,000개, 테스트 세트: 뒤쪽 10,000개
  - 훈련 세트를 섞어서 모든 교차 검증 폴드가 비슷해지도록 만들자.
    - 하나의 폴드라도 특정 숫자가 누락되면 안 됨
    - 훈련 샘플의 순서에 민감한 학습 알고리즘도 있으므로 비슷한 샘플이 연이어 나타나면 안 됨

In [6] :
```
# 훈련 데이터와 테스트 데이터를 나누기 위한 코드
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]

#데이터 셔플링
import numpy as np

shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

이진 분류기 훈련

❖ 숫자 5를 식별하는 감지기

- '5' 와 '~5' 두 개의 클래스를 구분할 수 있는 이진 분류기 (binary classifier)
- 타깃 벡터 생성

In [7] :
```
# 5는 True고, 다른 숫자는 모두 False
y_train_5 = (y_train == '5')
y_test_5 = (y_test == '5')
```

- 분류 모델을 선택해서 훈련시킴
  - SGDClassifier 클래스를 사용해 확률적 경사 하강법(Stochastic Gradient Decent: SGD) 분류기 적용
  - 훈련할 때 무작위성을 사용함
    - 결과를 재현하고 싶으면 random_state 매개변수를 사용함

In [8] :
```
# SGDClassifier 모델 생성 및 훈련 코드
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

- 숫자 5의 이미지를 감지

In [9] :
```
sgd_clf.predict([some_digit])
```

Out [9] :
```
array([ True])
```

성능 측정

❖ cross_val_score() 함수로 폴드가 3개인 K(=3)-fold cross validation을 사용해 SGDClassifier 모델을 평가

```
In [11] :   from sklearn.model_selection import cross_val_score
            cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

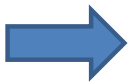
```
Out [11] :   array([0.9619, 0.96645, 0.96645 ])
```

❖ 모든 이미지를 '5', '~5' 클래스로 분류하는 더미 분류기

```
In [12] :   from sklearn.base import BaseEstimator
            class Never5Classifier(BaseEstimator):
                def fit(self, X, y=None):
                    pass
                def predict(self, X):
                    return np.zeros((len(X), 1), dtype=bool)

            never_5_clf = Never5Classifier()
            cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
Out [12] :   array([ 0.90975 ,  0.90745,  0.91175 ])
```



오차 행렬 (confusion matrix)

❖ 오차 행렬을 만들려면 실제 타깃과 비교할 수 있도록 먼저 예측 값을 생성한다.

- cross_val_predice() 메서드를 사용
  - K-fold 교차 검증을 수행하지만 평가 점수를 반환하지 않고 각 테스트 폴드에서 얻은 예측을 반환함

```
In [13] : # 예측값 생성 코드
          from sklearn.model_selection import cross_val_predict

          y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)

          # 오차 행렬 생성 코드
          from sklearn.metrics import confusion_matrix

          confusion_matrix(y_train_5, y_train_pred)
```

```
Out [13] : array([[53918,   661],
                  [ 1447,  3974]], dtype=int64)
```

- 완벽한 분류기

```
In [14] : y_train_perfect_predictions = y_train_5

          confusion_matrix(y_train_5, y_train_perfect_predictions)
```

```
Out [14] : array([[54579,     0],
                  [    0,  5421]], dtype=int64)
```

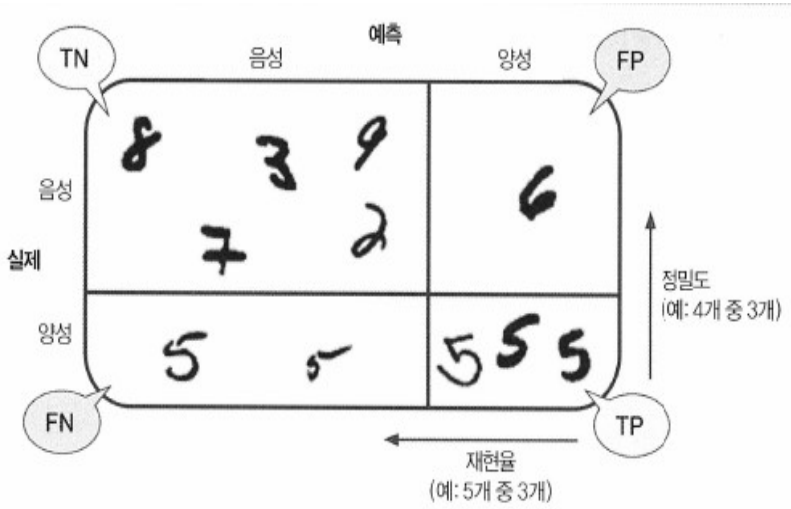


그림 3-2 오차행렬

오차 행렬 (confusion matrix)

❖ 정밀도 (precision)

- 정밀도=$\frac{TP}{TP+FP}$, TP: 진짜 양성의 수, FP: 거짓 양성의 수
- 분류기가 정확하게 감지한 양성 예측의 정확도

In [15] :
```
from sklearn.metrics import precision_score, recall_score

# 정밀도 계산 함수
precision_score(y_train_5, y_train_pred)
```

Out [15] : 0.8573894282632146

❖ 재현율 (recall 혹은 sensitivity)

- 재현율=$\frac{TP}{TP+FN}$, FN: 가짜 음성의 수
- 분류기가 정확하게 감지한 양성 샘플의 비율

In [16] :
```
# 재현율 계산 함수
recall_score(y_train_5, y_train_pred)
```

Out [16] : 0.7330750783988194

➡ 정밀도/재현율 트레이드오프

오차 행렬 (confusion matrix)

❖ F-1 Score (점수)

- 두 분류기를 비교할 때 사용
- 정밀도와 재현율의 조화 평균

$$F_1 = \frac{2}{\frac{1}{\text{정밀도}} + \frac{1}{\text{재현율}}} = 2 \times \frac{\text{정밀도} \times \text{재현율}}{\text{정밀도} + \text{재현율}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

```
In [17] : from sklearn.metrics import f1_score
          f1_score(y_train_5, y_train_pred)
```

```
Out [17] : 0.7903739061256961
```

❖ 정밀도/재현율 트레이드오프

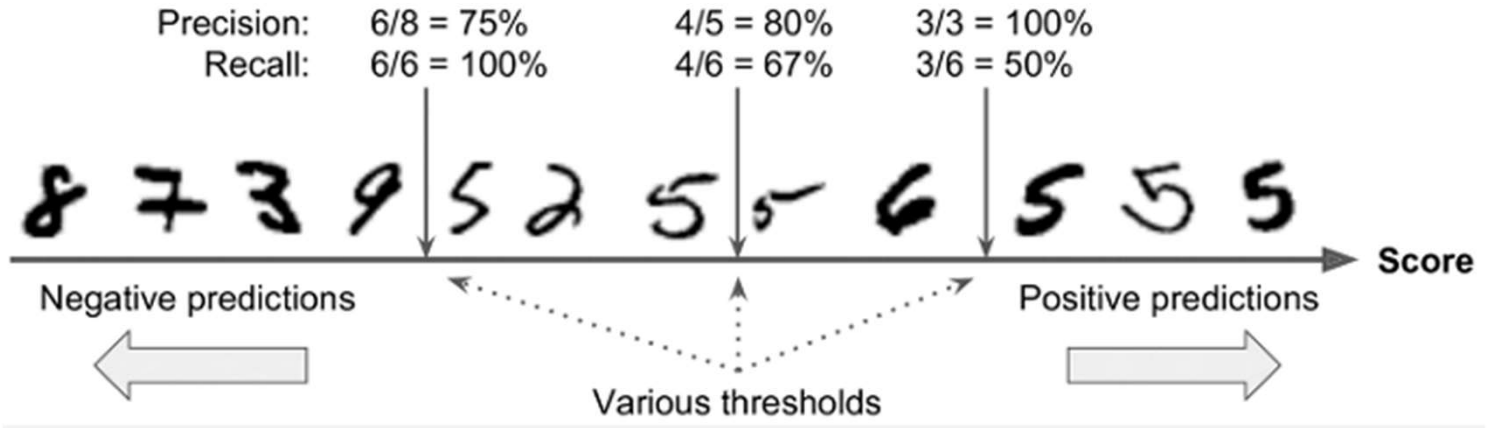


그림 3-3 결정 임곗값과 정밀도/재현율 트레이드오프

정밀도/재현율 트레이드오프

❖ 임계값을 직접 지정할 수는 없지만 예측에 사용한 각 샘플의 점수는 확인 가능

- predict() 메서드 대신 decision_function() 메서드를 호출

```
In [18] :  y_scores = sgd_clf.decision_function([some_digit])
           y_scores
```

```
Out [18] :  array([19516.2894736])
```

```
In [19] :  threshold = 0
           y_some_digit_pred = (y_scores > threshold)
           y_some_digit_pred
```

```
Out [19] :  array([ True])
```

```
In [20] :  threshold = 200000
           y_some_digit_pred = (y_scores > threshold)
           y_some_digit_pred
```

```
Out [20] :  array([ False])
```

정밀도/재현율 트레이드오프

❖ 적절한 임계 값을 정하는 방법

- cross_val_predict() 메소드르 사용해 훈련 세트에 있는 모든 샘플의 점수를 계산
- precision_recall_curve() 메소드를 사용하여 가능한 모든 임계 값에 대해 정밀도와 재현률을 계산
- 맷플롯립을 이용해 임계 값의 함수로 정밀도와 재현율을 그림

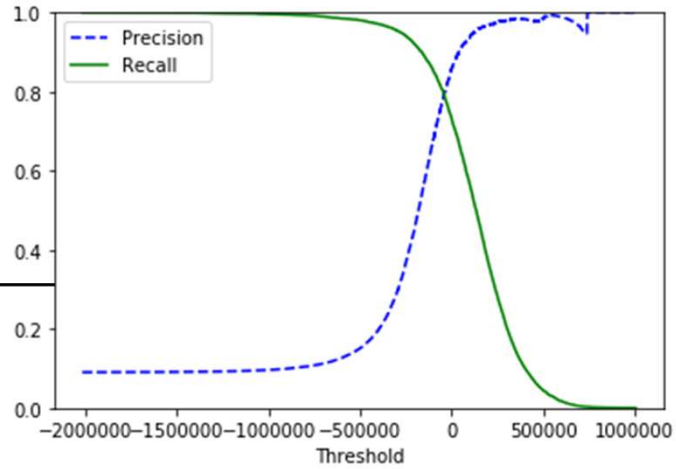
```
In [21] : y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3, method="decision_function")

          from sklearn.metrics import precision_recall_curve

          # 가능한 모든 임곗값에 대해 정밀도와 재현율을 계산하는 코드
          precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)

          # 정밀도와 재현율 그래프 생성 함수
          def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
              plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
              plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
              plt.xlabel("Threshold")
              plt.legend(loc="upper left")
              plt.ylim([0, 1])

          plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
          plt.show()
```



정밀도/재현율 트레이드오프

❖ 적절한 임계 값을 정하는 방법

- 재현율에 대한 정밀도 곡선을 그림

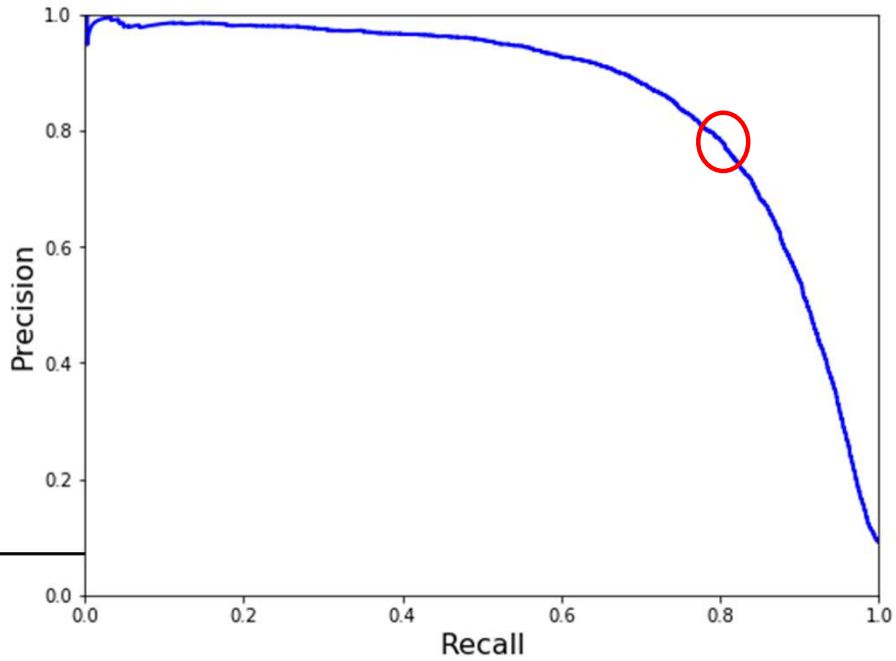
In [22] :

```
y_train_pred_90 = (y_scores > 70000)

precision_score(y_train_5, y_train_pred_90)
recall_score(y_train_5, y_train_pred_90)

#정밀도와 재현율
def plot_precision_vs_recall(precisions, recalls):
    plt.plot(recalls, precisions, "b-", linewidth=2)
    plt.xlabel("Recall", fontsize=16)
    plt.ylabel("Precision", fontsize=16)
    plt.axis([0, 1, 0, 1])

plt.figure(figsize=(8, 6))
plot_precision_vs_recall(precisions, recalls)
plt.show()
```



- 예제
  - 정밀도 90% 달성이 목표

In [23] :

```
#훈련 세트에 대한 예측 만들기
y_train_pred_90 = (y_scores > 70000)

# 임계값이 70,000 이상에서의 정밀도 코드
precision_score(y_train_5, y_train_pred_90)
```

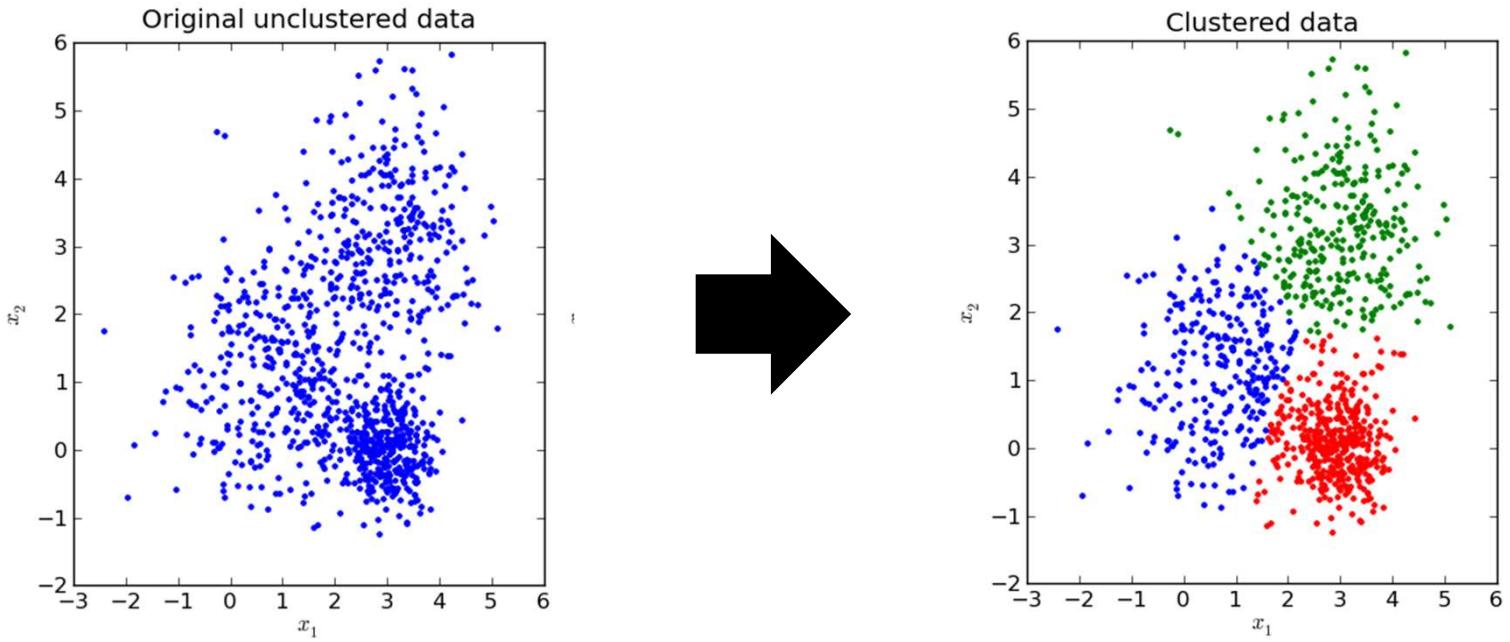
재현율은?

Out [23] : 0.9223995597138139

Cluster Analysis

◆ What is Cluster Analysis?

- 주어진 데이터의 특성을 고려하여 데이터 군집을 형성하여 데이터를 분석하는 기법
- 각 클러스터는 비슷한 특성을 가진 데이터를 포함함



Cluster Analysis

◆ 데이터간 유사도 정의

- **데이터 간 거리(Distance metrics)가 데이터 간 유사도를 나타내는 척도로 널리 사용됨**

Minkowski distance

$$d(i,j) = \sqrt[p]{\sum_{k=1}^{d} (x_{ik} - x_{jk})^p}$$

각각의 오브젝트 i, j 는 d 차원으로 구성됨
p = 1일 때, $d(i,j)$는 Manhattan distance이며. p = 2일 때, $d(i,j)$는 Euclidean distance.

Cosine distance

$$d(i,j) = 1 - cosine\ similarity(i,j) = 1 - \frac{\boldsymbol{i} \cdot \boldsymbol{j}}{\|\boldsymbol{i}\| \|\boldsymbol{j}\|}$$

Cluster Analysis

◆ How to Define Similarity?

- 오브젝트가 items로 구성된 경우(ex. 카테고리)

  Jaccard distance

$$d(i,j) = \frac{|A \cap B|}{|A \cup B|}$$

- 오브젝트가 수치 데이터와 카테고리 데이터를 모두 포함하는 경우

  Weighted distance

$$d(i,j) = \frac{\sum_{k=1}^{d} w_k d_{i,j}^{(f)}}{\sum_{k=1}^{d} w_k}$$

f 는 Mincowski, Cosine, Jaccard, 등의 distance metric

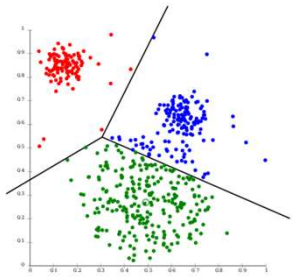
Cluster Analysis

◆ Major Clustering Approaches

- **Partitioning approach**

  특정 기준을 활용하여 데이터 파티션을 구성

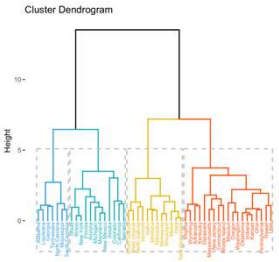
  Algorithms: k-means, k-medoids, minibatch k-means



- **Hierarchical approach**

  특정 기준을 활용하여 데이터를 계층적으로 구성

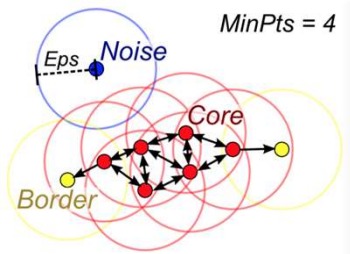
  Algorithms: Agglomerative, BIRCH



- **Density-based Approach**

  밀도 함수를 활용하여 클러스터 형성

  Algorithms: DBSCAN, OPTICS



Cluster Analysis

◆ How to measure the distance between cluster?

- **Single link**

  두 클러스터의 오브젝트 간 거리 중 최소 거리

  $$\min\{d(a,b)|a \in A, b \in B\}$$

- **Complete link**

  두 클러스터의 오브젝트 간 거리 중 최대 거리

  $$max\{d(a,b)|a \in A, b \in B\}$$

- **Average**

  두 클러스터의 모든 오브젝트 간 거리의 평균

  $$\left(\sum_{a \in A}\sum_{b \in B} d(a,b)\right) \Big/ |A| \cdot |B|$$

- **Centroid**

  두 클러스터의 중심점 간 거리

  $$d(A,B) = d(c_A, c_B)$$

- **Medoid**

  두 클러스터의 Medoid 간 거리

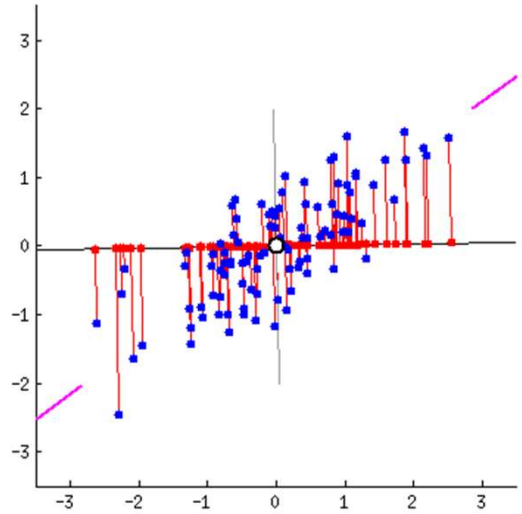
  $$d(A,B) = d(m_A, m_B)$$

  - Medoid : 클러스터 중심점에 가장 가까운 오브젝트

Dimensionality Reduction

◆ Principal Component Analysis(PCA)

- 고차원 데이터의 정보 손실을 최소화 하면서 저차원 데이터로 변환하는 기법
- covariance matrix 의 eigen value 와 vector를 계산하는 방식으로 계산됨



Dimensionality Reduction

◆ How to Calculate PCA

- **Step 1. Standardization**

```
In [1] :  import numpy as np
```

```
In [2] :   d = np.array([[170, 70], [150, 45], [160, 55], [180, 60], [170, 80]])
           d

Out [2] :  array([[170, 70],
                  [150, 45],
                  [160, 55],
                  [180, 60],
                  [170, 80]])
```

```
In [3] :   z = d - np.mean(d, axis=0)
           z

Out [3] :  array([[ 4., 8.],
                  [-16., -17.],
                  [ -6., -7.],
                  [ 14., -2.],
                  [ 4., 18.]])
```

Dimensionality Reduction

◆ How to Calculate PCA

- **Step 2. Calculate covariance vector**

```
In [4] :   sig = np.matmul(z.T, z) / 5
           sig

Out [4] :  array([[104., 78.],
                  [ 78., 146.]])
```

- **Step 3. Calculate SVD**

```
In [5] :   print(np.linalg.svd(sig))


Out [5] :  (array([[-0.60828716, -0.79371704],
                   [-0.79371704, 0.60828716]]),
            array([205.77747211, 44.22252789]),
            array([[-0.60828716, -0.79371704],
                   [-0.79371704, 0.60828716]]))
```

Dimensionality Reduction

◆ How to Calculate PCA

- **Using scikit-learn**

```
In [1] :  from sklearn.decomposition import PCA
```

```
In [2] :   d = np.array([[170, 70], [150, 45], [160, 55], [180, 60], [170, 80]])
           d

Out [2] :  array([[170, 70],
                  [150, 45],
                  [160, 55],
                  [180, 60],
                  [170, 80]])
```

```
In [3] :   d_pca = PCA(n_components=2).fit_transform(d)
           d_pca

Out [3] :  array([[ -8.78288493, -1.69142909],
                  [ 23.22578413, -2.35859097],
                  [ 9.2057422 , -0.50429214],
                  [ -6.9285861 , 12.32861285],
                  [-16.72005531, -7.77430064]])
```

K-means Algorithm

◆ Before We Start…

- 네트워크 보안분야(IDS)에서 널리 사용되는 KDD99 셋을 사용
- **Download link**

  http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

- 약 50만개의 레코드를 보유한 10% 데이터 사용

KDD Cup 1999 Data

Abstract

This is the data set used for The Third International Knowledge Discovery and Data Mining Tools Competition, which was held in conjunction with KDD-99 The Fifth International Conference on Knowledge Discovery and Data Mining. The competition task was to build a network intrusion detector, a predictive model capable of distinguishing between ``bad'' connections, called intrusions or attacks, and ``good'' normal connections. This database contains a standard set of data to be audited, which includes a wide variety of intrusions simulated in a military network environment.

Information files:

- task description. This is the original task decription given to competition participants.

Data files:

- kddcup.names A list of features.
- kddcup.data.gz The full data set (18M; 743M Uncompressed)
- kddcup.data_10_percent.gz A 10% subset. (2.1M; 75M Uncompressed) ← This one!!
- kddcup.newtestdata_10_percent_unlabeled.gz (1.4M; 45M Uncompressed)
- kddcup.testdata.unlabeled.gz (11.2M; 430M Uncompressed)
- kddcup.testdata.unlabeled_10_percent.gz (1.4M;45M Uncompressed)
- corrected.gz Test data with corrected labels.
- training_attack_types A list of intrusion types.
- typo-correction.txt A brief note on a typo in the data set that has been corrected (6/26/07)

K-means Algorithm

◆ K-Means Algorithm

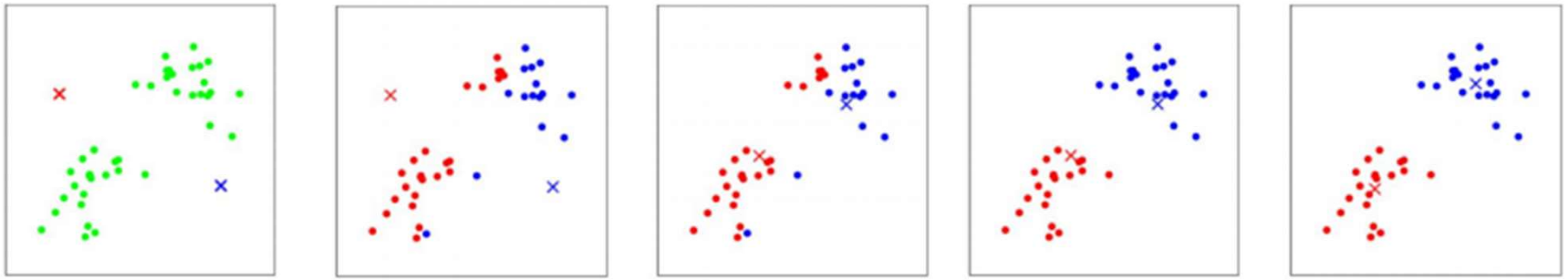
- **Partitioning 기반의 Clustering 알고리즘**
- **총 3단계로 구성되며 주로 마지막 두 단계가 반복 수행됨**
- **Algorithm**

Setp1. Centroids 초기화

Setp2. 오브젝트들을 가장 가까운 Centroid의 클러스터로 지정

Setp3. 각 클러스터의 새로운 Centroid를 계산

Setp4. Centroid가 수렴할 때까지 반복



K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 1. Import all package**

```
In [1] : import csv
         import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt

         from sklearn import metrics
         from sklearn.cluster import KMeans
         from sklearn.decomposition import PCA
         from sklearn.preprocessing import MinMaxScaler
```

- **Step 2. Load KDD-99 Dataset**

```
In [2] : col_names = ["duration", "protocol_type", "service", "flag", "src_bytes", "dst_bytes",
         "land", "wrong_fragment", "urgent", "hot", "num_failed_logins", "logged_in",
         "num_compromised", "root_shell", "su_attempted", "num_root", "num_file_creations",
         "num_shells", "num_access_files", "num_outbound_cmds", "is_host_login",
         "is_guest_login", "count", "srv_count", "serror_rate", "srv_serror_rate",
         "rerror_rate", "srv_rerror_rate", "same_srv_rate", "diff_srv_rate",
         "srv_diff_host_rate", "dst_host_count","dst_host_srv_count", "dst_host_same_srv_rate",
         "dst_host_diff_srv_rate", "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
         "dst_host_serror_rate", "dst_host_srv_serror_rate", "dst_host_rerror_rate",
         "dst_host_srv_rerror_rate", "label"]
```

K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

```
In [3] :  with open('./kddcup.data_10_percent_corrected', 'r') as file:
              data = pd.read_csv(file, names=col_names)
```

```
In [4] :  data['label'].value_counts()

Out [4] : smurf. 280790
          neptune. 107201
          normal. 97278
          back. 2203
          satan. 1589
          ipsweep. 1247
          portsweep. 1040
          warezclient. 1020
          teardrop. 979
          pod. 264
          nmap. 231
          guess_passwd. 53
          buffer_overflow. 30
          land. 21
          warezmaster. 20
          ...
          Name: label, dtype: int64
```

K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 3. Extract data by several per label**

```
In [5] :  unique_labels = data['label'].unique()
          unique_labels

Out [5] : array(['normal.', 'buffer_overflow.', 'loadmodule.', 'perl.', 'neptune.', 'smurf.',
                 'guess_passwd.', 'pod.', 'teardrop.', 'portsweep.', 'ipsweep.', 'land.',
                 'ftp_write.', 'back.', 'imap.', 'satan.', 'phf.', 'nmap.', 'multihop.',
                 'warezmaster.', 'warezclient.', 'spy.', 'rootkit.'], dtype=object)
```

```
In [6] :  selected_data = pd.DataFrame()

          for label in unique_labels:
              selected_data = pd.concat([selected_data, data.loc[data['label'] == label][:200]])

          selected_data
```

Out [6] :

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count	dst_host_same_srv_rate	dst_l
0	0	tcp	http	SF	181	5450	0	0	0	0	...	9	1.00	
1	0	tcp	http	SF	239	486	0	0	0	0	...	19	1.00	
2	0	tcp	http	SF	235	1337	0	0	0	0	...	29	1.00	
3	0	tcp	http	SF	219	1337	0	0	0	0	...	39	1.00	
4	0	tcp	http	SF	217	2032	0	0	0	0	...	49	1.00	

K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 3. Extract data by several per label**

```
In [7] :  selected_data['label'].value_counts()

Out [7] : satan. 200
          ipsweep. 200
          warezclient. 200
          teardrop. 200
          normal. 200
          portsweep. 200
          nmap. 200
          back. 200
          smurf. 200
          neptune. 200
          pod. 200
          guess_passwd. 53
          buffer_overflow. 30
          land. 21
          warezmaster. 20
          imap. 12
          rootkit. 10
          loadmodule. 9
          ...
          Name: label, dtype: int64
```

K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 4. Separate data and label**

```
In [8] :    labels = selected_data['label'].to_numpy()
            labels

Out [8] :   array(['normal.', 'normal.', 'normal.', ..., 'rootkit.', 'rootkit.’,
                   'rootkit.'], dtype=object)
```

```
In [9] :    data = selected_data.drop('label', axis=1)
            data
```

Out [9] :

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_count	dst_host_srv_count	dst_host_sam
0	0	tcp	http	SF	181	5450	0	0	0	0	...	9	9	
1	0	tcp	http	SF	239	486	0	0	0	0	...	19	19	
2	0	tcp	http	SF	235	1337	0	0	0	0	...	29	29	
3	0	tcp	http	SF	219	1337	0	0	0	0	...	39	39	
4	0	tcp	http	SF	217	2032	0	0	0	0	...	49	49	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
142477	0	tcp	ftp_data	SF	0	5636	0	0	0	0	...	1	41	
148154	61	tcp	telnet	SF	294	3929	0	0	0	0	...	255	4	
397011	0	udp	other	SF	32	0	0	0	0	0	...	255	1	

K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 5. Convert nominal data to numeric data**

```
In [10] :   data['protocol_type'], _ = data['protocol_type'].factorize()
            data['service'], _ = data['service'].factorize()
            data['flag'], _ = data['flag'].factorize()

            data
```

Out [10] :

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_count	dst_host_srv_count	dst_host_sam
0	0	0	0	0	181	5450	0	0	0	0	...	9	9	
1	0	0	0	0	239	486	0	0	0	0	...	19	19	
2	0	0	0	0	235	1337	0	0	0	0	...	29	29	
3	0	0	0	0	219	1337	0	0	0	0	...	39	39	
4	0	0	0	0	217	2032	0	0	0	0	...	49	49	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
142477	0	0	2	0	0	5636	0	0	0	0	...	1	41	
148154	61	0	1	0	294	3929	0	0	0	0	...	255	4	
397011	0	2	17	0	32	0	0	0	0	0	...	255	1	
452001	0	2	17	0	4	4	0	0	0	0	...	1	1	
452002	0	2	17	0	4	0	0	0	0	0	...	2	2	

2379 rows × 41 columns

K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 6. Scaling data**

```
In [11] :  scaler = MinMaxScaler()
           data = scaler.fit_transform(data)
```

```
In [12] :   data[0]

Out [12] :  array([0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                   2.61041764e-07, 1.05713002e-03, 0.00000000e+00, 0.00000000e+00,
                   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 1.00000000e+00,
                   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                   0.00000000e+00, 0.00000000e+00, 1.37254902e-02, 1.37254902e-02,
                   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                   1.00000000e+00, 0.00000000e+00, 0.00000000e+00, 3.14960630e-02,
                   3.14960630e-02, 1.00000000e+00, 0.00000000e+00, 1.10000000e-01,
                   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                   0.00000000e+00])
```

K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 7. Dimensionality reduction (41 => 3)**

```
In [13] : data = PCA(n_components=3).fit_transform(data)
```

```
In [14] :   data[0]

Out [14] :  array([[-0.78905185, -0.22631541, 0.69243538],
                   [-0.78385944, -0.22002823, 0.7308531 ],
                   [-0.78303919, -0.21632987, 0.74632718],
                   ...,
                   [-0.73505093, -0.27654194, 0.73775114],
                   [ 0.04080977, 1.2026111 , -0.27340135],
                   [ 0.51334923, 0.87373682, -0.195923 ]])
```

K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 8. Color generation (for color map generation)**

```
In [15] : colors = [plt.cm.Spectral(e) for e in np.linspace(0, 1, len(unique_labels))]
```

```
In [16] :   colors

Out [16] :  [(0.6196078431372549, 0.00392156862745098, 0.25882352941176473, 1.0),
             (0.7126489811610919, 0.10711264898116109, 0.28081507112648985, 1.0),
             (0.8141484044598232, 0.2196847366397539, 0.3048058439061899, 1.0),
             (0.8758169934640523, 0.3045751633986928, 0.2941176470588235 4, 1.0),
             (0.9330257593233372, 0.3913110342176086, 0.2719723183391003 5, 1.0),
             (0.9665513264129182, 0.49742406766628217, 0.295040369088812, 1.0),
             (0.9817762399077278, 0.6073817762399076, 0.3457900807381776, 1.0),
             (0.9928489042675894, 0.716955017301038, 0.40945790080738165, 1.0),
             (0.9946943483275663, 0.8092272202998846, 0.48696655132641287, 1.0),
             (0.9963860053825452, 0.8879661668589004, 0.5610918877354863, 1.0),
             (0.9982314494425221, 0.9451749327181853, 0.6570549788542868, 1.0),
             (0.998077662437524, 0.9992310649750096, 0.7460207612456747, 1.0),
             (0.9557862360630527, 0.9823144944252211, 0.6800461361014996, 1.0),
             (0.9096501345636295, 0.9638600538254518, 0.6080738177623992, 1.0),
             (0.8202998846597465, 0.9275663206459055, 0.6126874279123413, 1.0),
             (0.7114186851211075, 0.8832756632064592, 0.6348327566320646, 1.0),
             (0.5910034602076126, 0.835524798154556, 0.6442906574394464, 1.0),
             ...
```

K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 9. Generate ground-truth color map**

```
In [17] : def gen_color_dict(_unique_labels, _colors):
              color_dict = dict()

              for i in range(len(_unique_labels)):
                  color_dict[_unique_labels[i]] = _colors[i]

              return color_dict
```

```
In [18] : gt_color_dict = gen_color_dict(unique_labels, colors)
```

```
In [19] :   gt_color_dict

Out [19] :  {'normal.': (0.6196078431372549, 0.00392156862745098, 0.25882352941176473, 1.0),
             'buffer_overflow.': (0.7126489811610919, 0.10711264898116109, 0.28081507112648985,
            1.0),
             'loadmodule.': (0.8141484044598232, 0.2196847366397539, 0.3048058439061899, 1.0),
             'perl.': (0.8758169934640523, 0.3045751633986928, 0.29411764705882354, 1.0),
             'neptune.': (0.9330257593233372, 0.3913110342176086, 0.27197231833910035, 1.0),
             'smurf.': (0.9665513264129182, 0.49742406766628217, 0.295040369088812, 1.0),
             'guess_passwd.': (0.9817762399077278, 0.6073817762399076, 0.3457900807381776, 1.0)
            ,
             'pod.': (0.9928489042675894, 0.716955017301038, 0.40945790080738165, 1.0),
             ...
```

K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

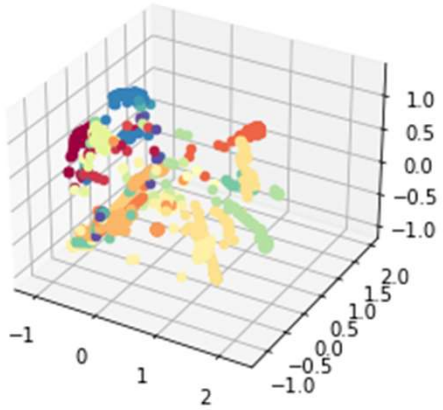
- **Step 10. Plot ground-truth cluster image**

```
In [20] :   fig = plt.figure()
            ax = fig.add_subplot(111, projection='3d')

            for i in range(len(data)):
                ax.scatter(data[i][0], data[i][1], data[i][2], c=[gt_color_dict[labels[i]]])

            plt.show()
```

Out [20] :



K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 11. Apply k-means algorithm and generate k-means cluster color map**

```
In [21] :   n_clusters = len(set(unique_labels))
            kmeans = KMeans(n_clusters=n_clusters)
            kmeans.fit(data)

Out [21] :  KMeans(n_clusters=23)
```

```
In [22] :   kmeans_label = kmeans.labels_
            kmeans_unique_labels = unique_labels
            kmeans_color_dict = gen_color_dict(kmeans_unique_labels, colors)
```

K-means Algorithm

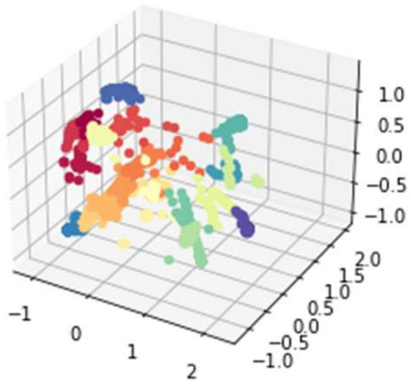
◆ K-means Algorithm with KDD-99 Dataset

- **Step 12. Plot k-means cluster image**

```
In [23] :   fig = plt.figure()
            ax = fig.add_subplot(111, projection='3d')

            for i in range(len(data)):
                ax.scatter(data[i][0], data[i][1], data[i][2],
            c=[kmeans_color_dict[kmeans_label[i]]])

            plt.show()
Out [23] :
```



K-means Algorithm

◆ K-means Algorithm with KDD-99 Dataset

- **Step 13. Check metrics**

```
In [24] :   print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels, kmeans_label))
            print("Completeness: %0.3f" % metrics.completeness_score(labels, kmeans_label))
            print("V-measure: %0.3f" % metrics.v_measure_score(labels, kmeans_label))

Out [24] :  Homogeneity: 0.775
            Completeness: 0.716
            V-measure: 0.744
```

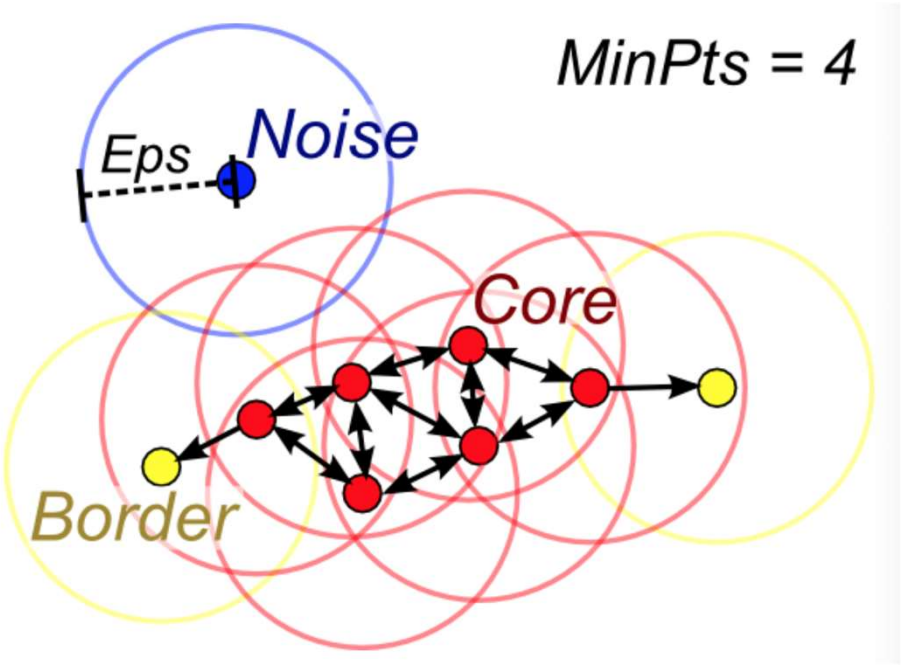
Homogeneity : 각 cluster들이 각 class의 data points만을 포함함
Completeness : 각 class의 모든 data points가 동일한 cluster 내에 있음
V-measure : Homogeneity와 completeness의 조화 평균

DBSCAN Algorithm

◆ DBSCAN Algorithm

- **Density 기반의 clustering 알고리즘**
- **알고리즘을 설명하기위해 다음 5가지 용어를 정의함**

Core	Eps
Noise	MinPts
Border	



DBSCAN Algorithm

◆ DBSCAN Algorithm

- **Algorithm**

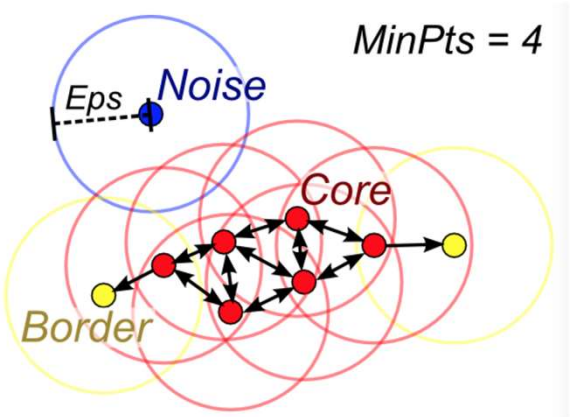
  Setp1. 임의의 오브젝트 ***P***를 선택

  Setp2. **Eps**와 **MinPts**를 만족하는 ***P***와 연결된 모든 포인트를 탐색

  Setp3-1. 만약 ***P***가 **Core** 라면 새로운 클러스터를 생성

  Setp3-2. 만약 ***P***가 **Border** 또는 **Noise** 라면, 데이터베이스 내의 다른 오브젝트를 ***P***로 선택

  Setp4. 모든 오브젝트를 방문할 때까지 반복



DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 1. Import all package**

```
In [1] : import csv
         import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt

         from sklearn import metrics
         from sklearn.cluster import KMeans
         from sklearn.cluster import DBSCAN
         from sklearn.decomposition import PCA
         from sklearn.preprocessing import MinMaxScaler
```

- **Step 2. Load KDD-99 Dataset**

```
In [2] : col_names = ["duration", "protocol_type", "service", "flag", "src_bytes", "dst_bytes",
         "land", "wrong_fragment", "urgent", "hot", "num_failed_logins", "logged_in",
         "num_compromised", "root_shell", "su_attempted", "num_root", "num_file_creations",
         "num_shells", "num_access_files", "num_outbound_cmds", "is_host_login",
         "is_guest_login", "count", "srv_count", "serror_rate", "srv_serror_rate",
         "rerror_rate", "srv_rerror_rate", "same_srv_rate", "diff_srv_rate",
         "srv_diff_host_rate", "dst_host_count","dst_host_srv_count", "dst_host_same_srv_rate",
         "dst_host_diff_srv_rate", "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
         "dst_host_serror_rate", "dst_host_srv_serror_rate", "dst_host_rerror_rate",
         "dst_host_srv_rerror_rate", "label"]
```

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

```
In [3] :  with open('./kddcup.data_10_percent_corrected', 'r') as file:
              data = pd.read_csv(file, names=col_names)
```

```
In [4] :  data['label'].value_counts()

Out [4] : smurf. 280790
          neptune. 107201
          normal. 97278
          back. 2203
          satan. 1589
          ipsweep. 1247
          portsweep. 1040
          warezclient. 1020
          teardrop. 979
          pod. 264
          nmap. 231
          guess_passwd. 53
          buffer_overflow. 30
          land. 21
          warezmaster. 20
          ...
          Name: label, dtype: int64
```

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 3. Extract data by several per label**

```
In [5] :  unique_labels = data['label'].unique()
          unique_labels

Out [5] : array(['normal.', 'buffer_overflow.', 'loadmodule.', 'perl.', 'neptune.', 'smurf.',
                 'guess_passwd.', 'pod.', 'teardrop.', 'portsweep.', 'ipsweep.', 'land.',
                 'ftp_write.', 'back.', 'imap.', 'satan.', 'phf.', 'nmap.', 'multihop.',
                 'warezmaster.', 'warezclient.', 'spy.', 'rootkit.'], dtype=object)
```

```
In [6] :  selected_data = pd.DataFrame()

          for label in unique_labels:
              selected_data = pd.concat([selected_data, data.loc[data['label'] == label][:200]])

          selected_data
```

Out [6] :

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count	dst_host_same_srv_rate	dst_l
0	0	tcp	http	SF	181	5450	0	0	0	0	...	9	1.00	
1	0	tcp	http	SF	239	486	0	0	0	0	...	19	1.00	
2	0	tcp	http	SF	235	1337	0	0	0	0	...	29	1.00	
3	0	tcp	http	SF	219	1337	0	0	0	0	...	39	1.00	
4	0	tcp	http	SF	217	2032	0	0	0	0	...	49	1.00	

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 4. Separate data and label**

```
In [8] :    labels = selected_data['label'].to_numpy()
            labels

Out [8] :   array(['normal.', 'normal.', 'normal.', ..., 'rootkit.', 'rootkit.’,
                   'rootkit.'], dtype=object)
```

```
In [9] :    data = selected_data.drop('label', axis=1)
            data
```

Out [9] :

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_count	dst_host_srv_count	dst_host_sam
0	0	tcp	http	SF	181	5450	0	0	0	0	...	9	9	
1	0	tcp	http	SF	239	486	0	0	0	0	...	19	19	
2	0	tcp	http	SF	235	1337	0	0	0	0	...	29	29	
3	0	tcp	http	SF	219	1337	0	0	0	0	...	39	39	
4	0	tcp	http	SF	217	2032	0	0	0	0	...	49	49	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
142477	0	tcp	ftp_data	SF	0	5636	0	0	0	0	...	1	41	
148154	61	tcp	telnet	SF	294	3929	0	0	0	0	...	255	4	
397011	0	udp	other	SF	32	0	0	0	0	0	...	255	1	

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 5. Convert nominal data to numeric data**

```
In [10] :   data['protocol_type'], _ = data['protocol_type'].factorize()
            data['service'], _ = data['service'].factorize()
            data['flag'], _ = data['flag'].factorize()

            data
```

Out [10] :

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_count	dst_host_srv_count	dst_host_sam
0	0	0	0	0	181	5450	0	0	0	0	...	9	9	
1	0	0	0	0	239	486	0	0	0	0	...	19	19	
2	0	0	0	0	235	1337	0	0	0	0	...	29	29	
3	0	0	0	0	219	1337	0	0	0	0	...	39	39	
4	0	0	0	0	217	2032	0	0	0	0	...	49	49	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
142477	0	0	2	0	0	5636	0	0	0	0	...	1	41	
148154	61	0	1	0	294	3929	0	0	0	0	...	255	4	
397011	0	2	17	0	32	0	0	0	0	0	...	255	1	
452001	0	2	17	0	4	4	0	0	0	0	...	1	1	
452002	0	2	17	0	4	0	0	0	0	0	...	2	2	

2379 rows × 41 columns

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 6. Scaling data**

```
In [11] :  scaler = MinMaxScaler()
           data = scaler.fit_transform(data)
```

```
In [12] :   data[0]

Out [12] :  array([0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                   2.61041764e-07, 1.05713002e-03, 0.00000000e+00, 0.00000000e+00,
                   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 1.00000000e+00,
                   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                   0.00000000e+00, 0.00000000e+00, 1.37254902e-02, 1.37254902e-02,
                   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                   1.00000000e+00, 0.00000000e+00, 0.00000000e+00, 3.14960630e-02,
                   3.14960630e-02, 1.00000000e+00, 0.00000000e+00, 1.10000000e-01,
                   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                   0.00000000e+00])
```

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 7. Dimensionality reduction (41 => 3)**

```
In [13] : data = PCA(n_components=3).fit_transform(data)
```

```
In [14] :   data[0]

Out [14] :  array([[-0.78905185, -0.22631541, 0.69243538],
                   [-0.78385944, -0.22002823, 0.7308531 ],
                   [-0.78303919, -0.21632987, 0.74632718],
                   ...,
                   [-0.73505093, -0.27654194, 0.73775114],
                   [ 0.04080977, 1.2026111 , -0.27340135],
                   [ 0.51334923, 0.87373682, -0.195923 ]])
```

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 8. Color generation (for color map generation)**

```
In [15] : colors = [plt.cm.Spectral(e) for e in np.linspace(0, 1, len(unique_labels))]
```

```
In [16] :   colors

Out [16] :  [(0.6196078431372549, 0.00392156862745098, 0.25882352941176473, 1.0),
             (0.7126489811610919, 0.10711264898116109, 0.28081507112648985, 1.0),
             (0.8141484044598232, 0.2196847366397539, 0.3048058439061899, 1.0),
             (0.8758169934640523, 0.3045751633986928, 0.2941176470588235 4, 1.0),
             (0.9330257593233372, 0.3913110342176086, 0.2719723183391003 5, 1.0),
             (0.9665513264129182, 0.49742406766628217, 0.295040369088812, 1.0),
             (0.9817762399077278, 0.6073817762399076, 0.3457900807381776, 1.0),
             (0.9928489042675894, 0.716955017301038, 0.4094579080738165, 1.0),
             (0.9946943483275663, 0.8092272202998846, 0.4869665513264128 7, 1.0),
             (0.9963860053825452, 0.8879661668589004, 0.5610918877354863, 1.0),
             (0.9982314494425221, 0.9451749327181853, 0.6570549788542868, 1.0),
             (0.998077662437524, 0.9992310649750096, 0.7460207612456747, 1.0),
             (0.9557862360630527, 0.9823144944252211, 0.6800461361014996, 1.0),
             (0.9096501345636295, 0.9638600538254518, 0.6080738177623992, 1.0),
             (0.8202998846597465, 0.9275663206459055, 0.6126874279123413, 1.0),
             (0.7114186851211075, 0.8832756632064592, 0.6348327566320646, 1.0),
             (0.5910034602076126, 0.835524798154556, 0.6442906574394464, 1.0),
             ...
```

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 9. Generate ground-truth color map**

```
In [17] : def gen_color_dict(_unique_labels, _colors):
              color_dict = dict()

              for i in range(len(_unique_labels)):
                  color_dict[_unique_labels[i]] = _colors[i]

              return color_dict
```

```
In [18] : gt_color_dict = gen_color_dict(unique_labels, colors)
```

```
In [19] :   gt_color_dict

Out [19] :  {'normal.': (0.6196078431372549, 0.00392156862745098, 0.25882352941176473, 1.0),
             'buffer_overflow.': (0.7126489811610919, 0.10711264898116109, 0.28081507112648985,
            1.0),
             'loadmodule.': (0.8141484044598232, 0.2196847366397539, 0.3048058439061899, 1.0),
             'perl.': (0.8758169934640523, 0.3045751633986928, 0.29411764705882354, 1.0),
             'neptune.': (0.9330257593233372, 0.3913110342176086, 0.27197231833910035, 1.0),
             'smurf.': (0.9665513264129182, 0.49742406766628217, 0.295040369088812, 1.0),
             'guess_passwd.': (0.9817762399077278, 0.6073817762399076, 0.3457900807381776, 1.0)
            ,
             'pod.': (0.9928489042675894, 0.716955017301038, 0.40945790080738165, 1.0),
             ...
```

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 10. Color generation (for color map generation)**

```
In [24] : colors = [plt.cm.Spectral(e) for e in np.linspace(0, 1, len(unique_labels))]
```

```
In [25] :   colors

Out [25] :  [(0.6196078431372549, 0.00392156862745098, 0.25882352941176473, 1.0),
             (0.7126489811610919, 0.10711264898116109, 0.28081507112648985, 1.0),
             (0.8141484044598232, 0.2196847366397539, 0.3048058439061899, 1.0),
             (0.8758169934640523, 0.3045751633986928, 0.2941176470588235 4, 1.0),
             (0.9330257593233372, 0.3913110342176086, 0.2719723183391003 5, 1.0),
             (0.9665513264129182, 0.49742406766628217, 0.295040369088812, 1.0),
             (0.9817762399077278, 0.6073817762399076, 0.3457900807381776, 1.0),
             (0.9928489042675894, 0.716955017301038, 0.40945790080738165, 1.0),
             (0.9946943483275663, 0.8092272202998846, 0.48696655132641287, 1.0),
             (0.9963860053825452, 0.8879661668589004, 0.5610918877354863, 1.0),
             (0.9982314494425221, 0.9451749327181853, 0.6570549788542868, 1.0),
             (0.998077662437524, 0.9992310649750096, 0.7460207612456747, 1.0),
             (0.9557862360630527, 0.9823144944252211, 0.6800461361014996, 1.0),
             (0.9096501345636295, 0.9638600538254518, 0.6080738177623992, 1.0),
             (0.8202998846597465, 0.9275663206459055, 0.6126874279123413, 1.0),
             (0.7114186851211075, 0.8832756632064592, 0.6348327566320646, 1.0),
             (0.5910034602076126, 0.835524798154556, 0.6442906574394464, 1.0),
             ...
```

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 9. Generate ground-truth color map**

```
In [17] : def gen_color_dict(_unique_labels, _colors):
              color_dict = dict()

              for i in range(len(_unique_labels)):
                  color_dict[_unique_labels[i]] = _colors[i]

              return color_dict
```

```
In [18] : gt_color_dict = gen_color_dict(unique_labels, colors)
```

```
In [19] :   gt_color_dict

Out [19] :  {'normal.': (0.6196078431372549, 0.00392156862745098, 0.25882352941176473, 1.0),
             'buffer_overflow.': (0.7126489811610919, 0.10711264898116109, 0.28081507112648985,
            1.0),
             'loadmodule.': (0.8141484044598232, 0.2196847366397539, 0.3048058439061899, 1.0),
             'perl.': (0.8758169934640523, 0.3045751633986928, 0.29411764705882354, 1.0),
             'neptune.': (0.9330257593233372, 0.3913110342176086, 0.27197231833910035, 1.0),
             'smurf.': (0.9665513264129182, 0.49742406766628217, 0.295040369088812, 1.0),
             'guess_passwd.': (0.9817762399077278, 0.6073817762399076, 0.3457900807381776, 1.0)
            ,
             'pod.': (0.9928489042675894, 0.716955017301038, 0.40945790080738165, 1.0),
             ...
```

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

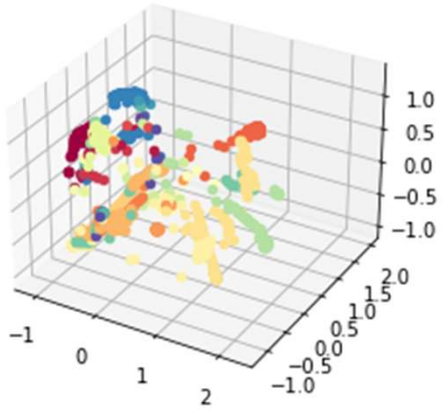
- **Step 10. Plot ground-truth cluster image**

```
In [20] :   fig = plt.figure()
            ax = fig.add_subplot(111, projection='3d')

            for i in range(len(data)):
                ax.scatter(data[i][0], data[i][1], data[i][2], c=[gt_color_dict[labels[i]]])

            plt.show()
```

Out [20] :



DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 13. Apply DBSCAN algorithm and generate k-means cluster color map**

```
In [30] :   dbscan = DBSCAN(eps=0.3, min_samples=10)
            dbscan.fit(data)

Out [30] :  DBSCAN(eps=0.3, min_samples=10)
```

```
In [31] :   dbscan_label = dbscan.labels_
            dbscan_unique_labels = list(set(dbscan_label))
            colors = [plt.cm.Spectral(e) for e in np.linspace(0, 1, len(dbscan_unique_labels))]
            dbscan_color_dict = gen_color_dict(dbscan_unique_labels, colors)
```

DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

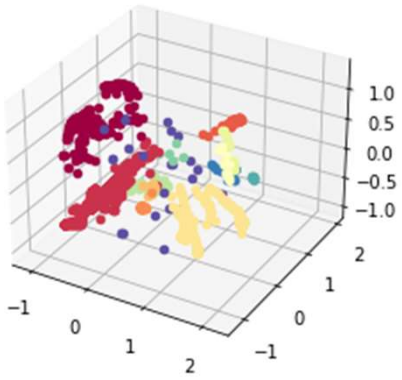
- **Step 14. Plot DBSCAN cluster image**

```
In [32] :   fig = plt.figure()
            ax = fig.add_subplot(111, projection='3d')

            for i in range(len(data)):
                ax.scatter(data[i][0], data[i][1], data[i][2],
            c=[dbscan_color_dict[dbscan_label[i]]])

            plt.show()
```

Out [32] :



DBSCAN Algorithm

◆ DBSCAN Algorithm with KDD-99 Dataset

- **Step 15. Check metrics**

```
In [33] :   n_clusters_ = len(set(dbscan_label)) - (1 if -1 in dbscan_label else 0)

            print('Estimated number of clusters: %d' % n_clusters_)
            print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels, dbscan_label))
            print("Completeness: %0.3f" % metrics.completeness_score(labels, dbscan_label))
            print("V-measure: %0.3f" % metrics.v_measure_score(labels, dbscan_label))
            Estimated number of clusters: 10

Out [33] :  Homogeneity: 0.525
            Completeness: 0.845
            V-measure: 0.648
```

Homogeneity : 각 cluster들이 각 class의 data points만을 포함함
Completeness : 각 class의 모든 data points가 동일한 cluster 내에 있음
V-measure : Homogeneity와 completeness의 조화 평균

Q&A