

Machine Learning with Python

박태우

강사 소개

■ 박태우

▶ 이력

- 금강대학교 IT Software학과 강의 전담 교수
- Korea Digital Tech의 CTO

▶ 학력

- 박사 – 한국외국어대학교(정보공학)
- 석사 – 루이지애나 주립대학교(전자공학)
- 학사 – 한국외국어대학교(제어계측)

▶ 경력 사항

- CDNS 네트워크 개발
- 2008년 베이징 올림픽, 2010 남아공 월드컵 미주내 한인 위성 방송 송출 프로젝트
- 미주내 인터넷 전화 하나폰, 한국 통신(KT)와 협업하여 출시 및 진행 등

강의 내용

- 파이썬 기반의 머신 러닝 생태계 이해
- 사이킷 런으로 시작하는 머신러닝
- 머신러닝 평가
- 머신러닝 분류
- 회귀
- 비지도 학습 (차원 축소)
- 비지도 학습 (군집화)

2012
2011

Chapter 01

파이썬 기반의 머신 러닝 생태계 이해

파이썬 기반의 머신러닝과 생태계 이해

- 머신 러닝이란?
- 데이터 기반 숨겨진 패턴을 인지해 해결
- Predictive Analysis
- Extensive to : 데이터 마이닝, 영상 인식, 자연어 처리

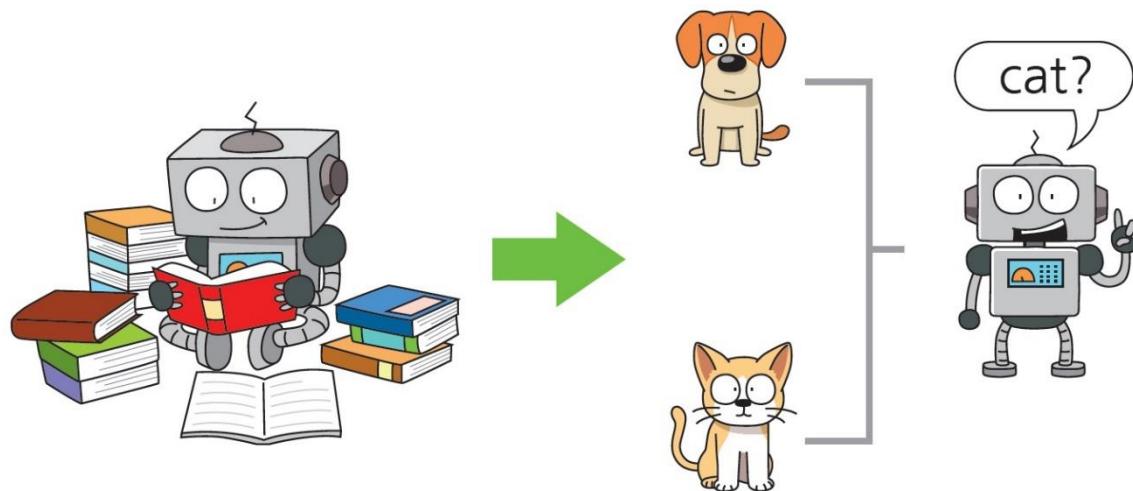
ML?

■ 인공지능의 미래

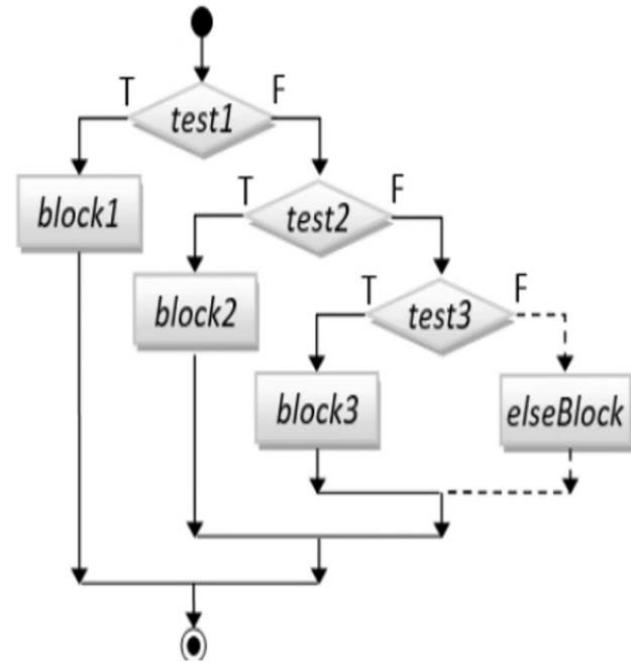
- 모든 산업과 사회 분야에 깊숙이 파고들어 무한한 부를 창출
- 자신의 전문분야는 다른 전공자가 접근하기 어렵기 때문에 자신만이 최적의 맞춤형 인공지능 서비스를 제작할 수 있음
- 인공지능 서비스는 분야별로 특성이 다르기 때문에 빅데이터 분석 방법이나 인공지능 서비스 모델의 적용이 달라짐
- 자신의 전공을 최대한 살리는 방법

기계학습이란

- 현재의 컴퓨터는 스스로 학습할 수 없기 때문에 우리가 컴퓨터에게 어떤 작업을 시키려면 반드시 프로그램을 작성하여 작업을 지시하여야 한다.
- 컴퓨터가 스스로 학습할 수 있다면 컴퓨터는 프로그램 없이도 여러 가지 일을 할 수 있을 것이다.



Why ML?



- 현실 세계의 복잡한 업무와 규칙을 구현하기 위한 매우 복잡하고 방대한 코드
- 수시로 변하는 업무 환경, 정책, 사용자 성향에 따른 애플리케이션 구현의 어려움
- 많은 자원과 비용을 통해서 구현된 애플리케이션의 예측 정확성 문제

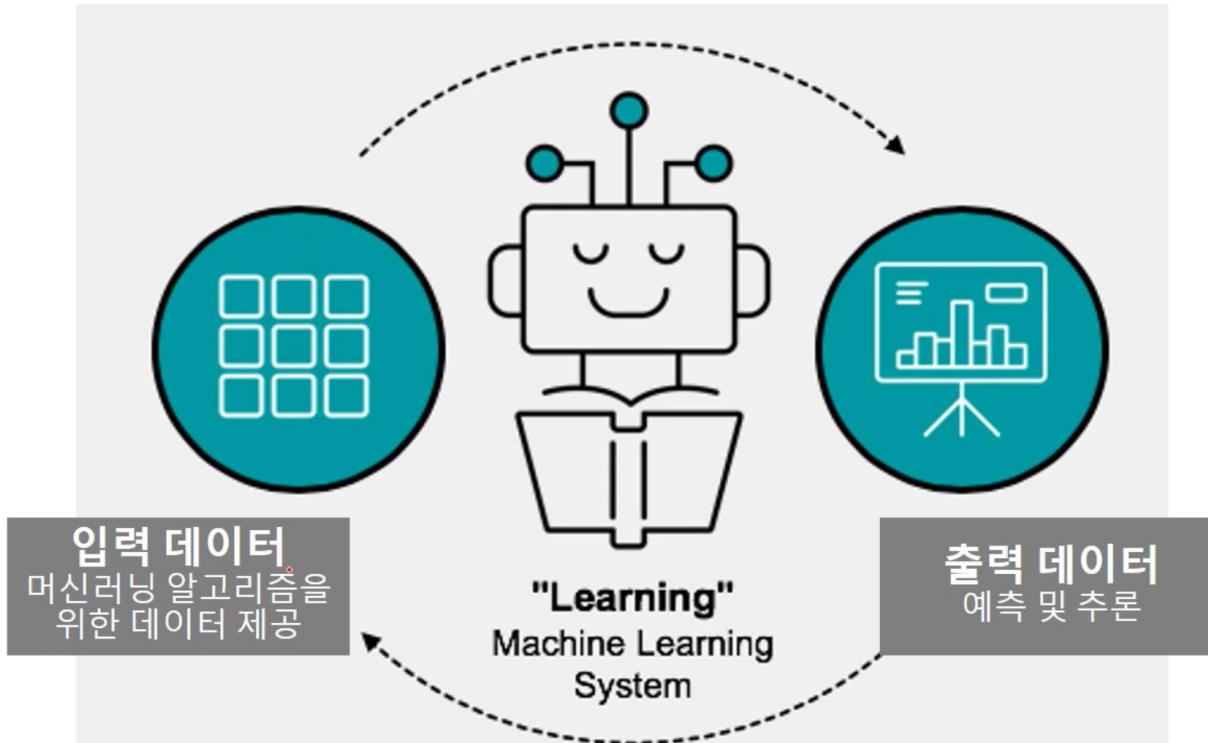
Why ML?



- 동일한 숫자라 하더라도 여러 변형으로 인해 숫자 인식에 필요한 여러 특징 (feature) 들을 if else 와 같은 조건으로 구분하여 숫자를 인식하기 어렵다.

Why ML?

머신러닝을 통한 복잡한 문제의 해결



- 머신러닝은 이러한 복잡한 문제를 데이터를 기반으로 숨겨진 패턴을 인지해 해결합니다.
- 머신러닝 알고리즘은 데이터를 기반으로 통계적인 신뢰도를 강화하고 예측 오류를 최소화하기 위한 다양한 수학적 기법을 적용해 데이터 내의 패턴을 스스로 인지하고 신뢰도 있는 예측 결과를 도출해냅니다

머신러닝의 분류

머신러닝은 지도학습(Supervised Learning)과 비지도학습(Un-supervised Learning), 강화학습(Reinforcement Learning)으로 나뉩니다. 지도 학습은 명확한 결정값이 주어진 데이터를 학습하는 것이며, 비지도 학습은 결정값이 주어지지 않는 데이터를 학습하는 것입니다.

지도 학습

분류

회귀

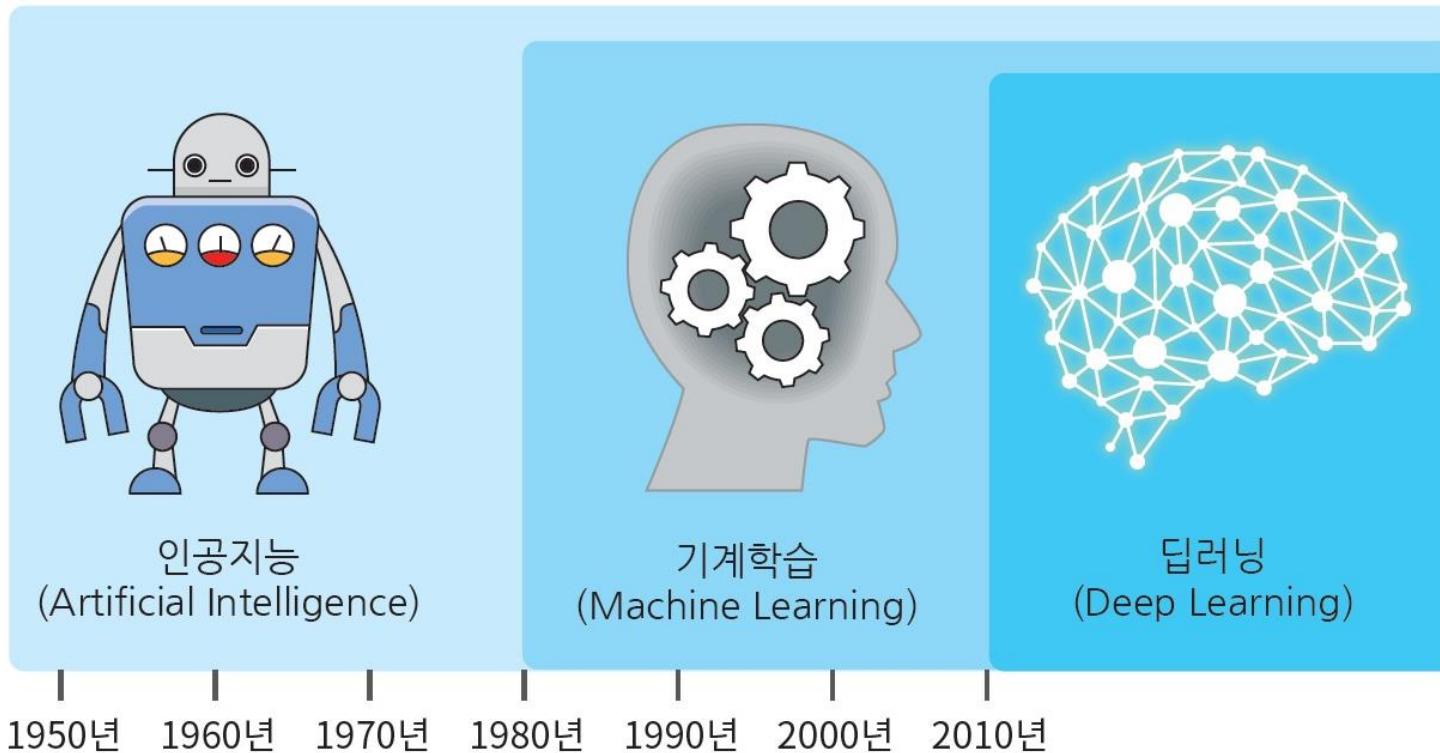
시각/음성 감지/인지

비지도 학습

군집화(클러스터링)

차원 축소

인공지능, 기계 학습, 딥러닝



기계 학습은 어디에 이용되는가

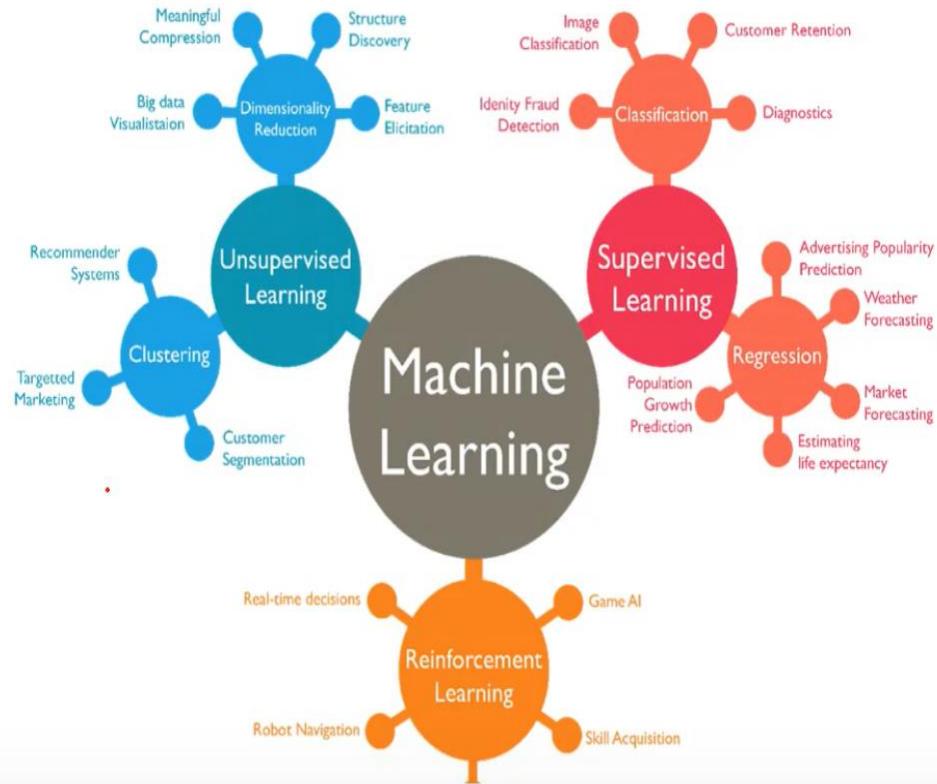


기계 학습은 어디에 이용되는가

- 이들 분야들은 살펴보면 복잡한 데이터들이 있고, 이들 데이터에 기반하여 결정을 내려야 하는 분야이다.
 - 영상 인식, 음성 인식처럼 프로그램으로 작성하기에는 규칙과 공식이 너무 복잡할 때
 - 주식 거래나 에너지 수요 예측, 쇼핑 추세 예측의 경우처럼 데이터 특징이 계속 바뀌고 프로그램을 계속해서 변경해야 하는 상황일 때
 - 구매자가 클릭할 확률이 가장 높은 광고가 무엇인지를 알아내는 시스템
 - 넷플릭스에서 비디오 추천 시스템
 - 자율 주행자동차

ML(Predictive Analysis)

- 머신러닝은 데이터를 관통하는 패턴을 학습하고, 이에 기반한 예측을 수행하면서 데이터 분석 영역에 새로운 혁신을 가져왔습니다.
- 데이터 분석 영역은 재빠르게 머신러닝 기반의 예측 분석(Predictive Analysis)으로 재편되고 있습니다.
- 많은 데이터 분석가와 데이터 과학자가 머신러닝 알고리즘 기반의 새로운 예측 모델을 이용해 더욱 정확한 예측 및 의사 결정을 도출하고 있으며, 데이터에 감춰진 새로운 의미와 인사이트를 발굴해 놀랄 만한 이익으로 연결시키고 있습니다.



<https://www.predictiveanalyticsworld.com>

인공지능의 역사

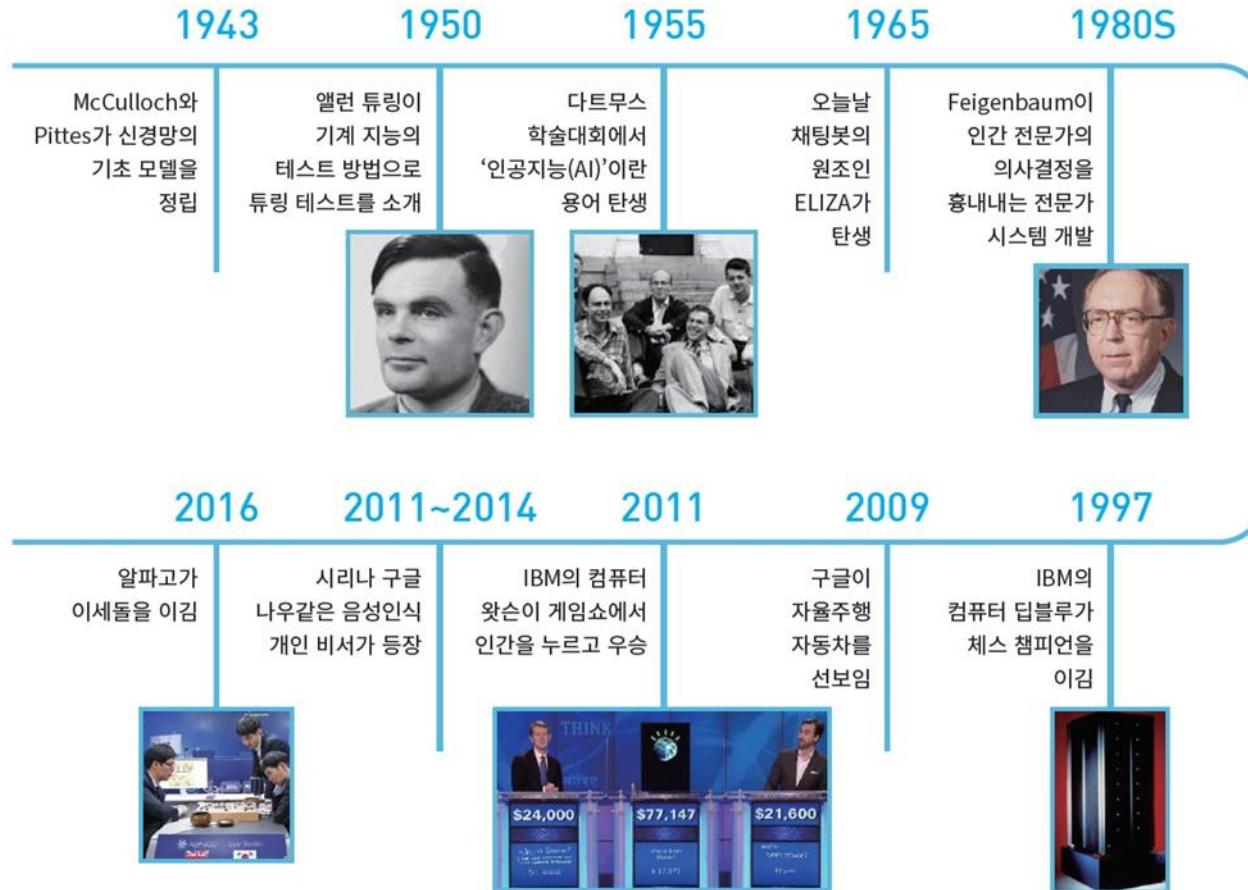
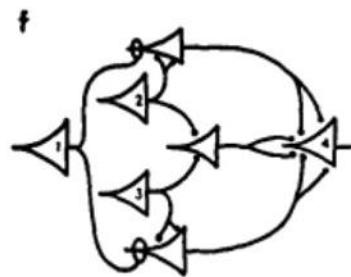
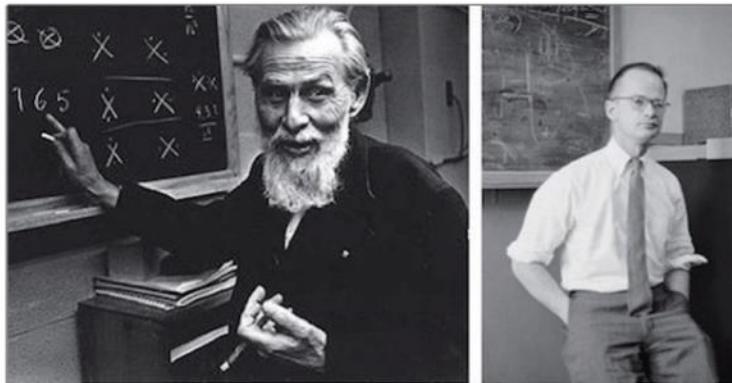


그림 1-17 인공지능의 역사

인공지능의 태동

- 1943년에 Warren McCulloch과 Walter Pitts는 뉴런들의 간단한 네트워크를 분석하고 이것이 간단한 논리 기능을 수행할 수 있음을 보여주었다. 이것들은 나중에 연구자들이 인공 신경망이라고 부르게 되었다.



Warren McCulloch와 Walter Pitts, 그들이 만들었던 신경망

튜링 테스트

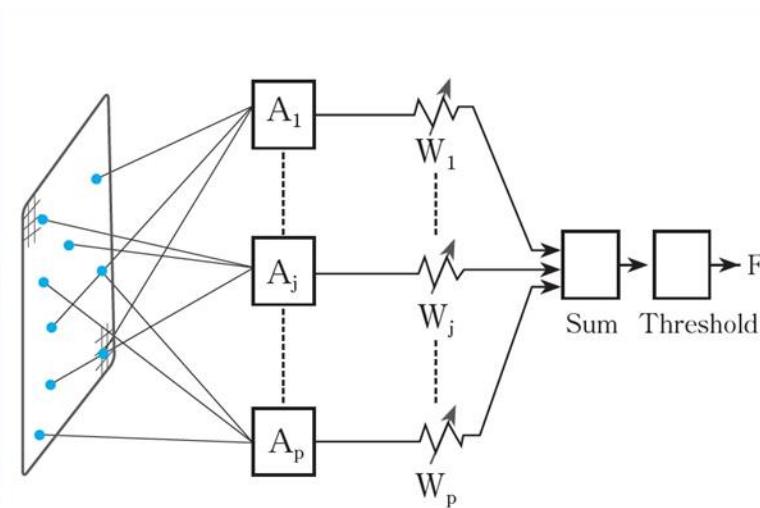


튜링 테스트

- 인공 신경망의 초기 형태인 퍼셉트론(perceptron)을 Frank Rosenblatt가 개발하였다. Rosenblatt는 "퍼셉트론은 궁극적으로 언어를 배우고 결정하며 언어를 번역할 수 있게 될 것"이라고 예측하여 낙관적인 입장을 보였다. Minsky와 Papert의 1969년 저서 '퍼셉트론(Perceptrons)'이 발표되면서 갑작스럽게 중단되었다.



Rosenblatt와 퍼셉트론



다트머스 학술 대회

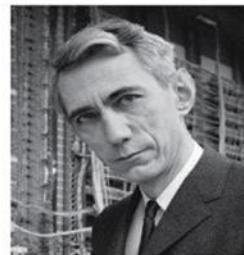
- 1956년에 의해 다트머스 학술 회의가 Marvin Minsky와 John MacCarthy 등에 의하여 조직되었다



John McCarthy



Marvin Minsky



Claude Shannon



Ray Solomonoff



Alan Newell



Herbert Simon



Arthur Samuel



Oliver Selfridge



Nathaniel Rochester



Trenchard More

다트머스 학술 회의 참가자들

“탐색으로 추로하기”, 시대

- 많은 초기의 AI 프로그램은 기본 탐색 알고리즘을 사용했다. 이들 알고리즘은 어떤 목표를 달성하기 위해, 미로를 탐색하는 것처럼 단계별로 진행하였고 막 다른 곳에 도달할 때마다 탐색 트리 상에서 되돌아갔다.

7			2	
8	5	4		
6	3	1		

10^5 개의 상태

조합 폭발

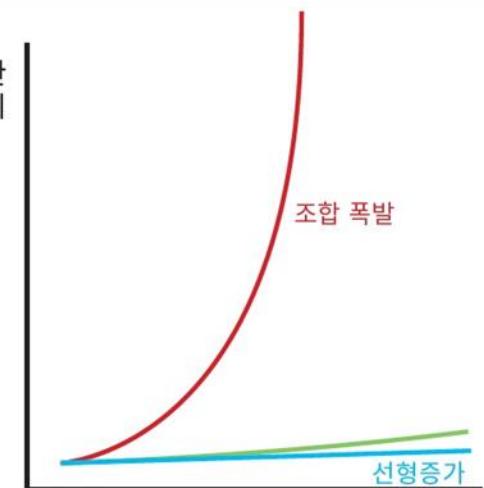
4	10	15	3	
1			13	7
6	2	9	5	
8	14	11	12	

10^{13} 개의 상태

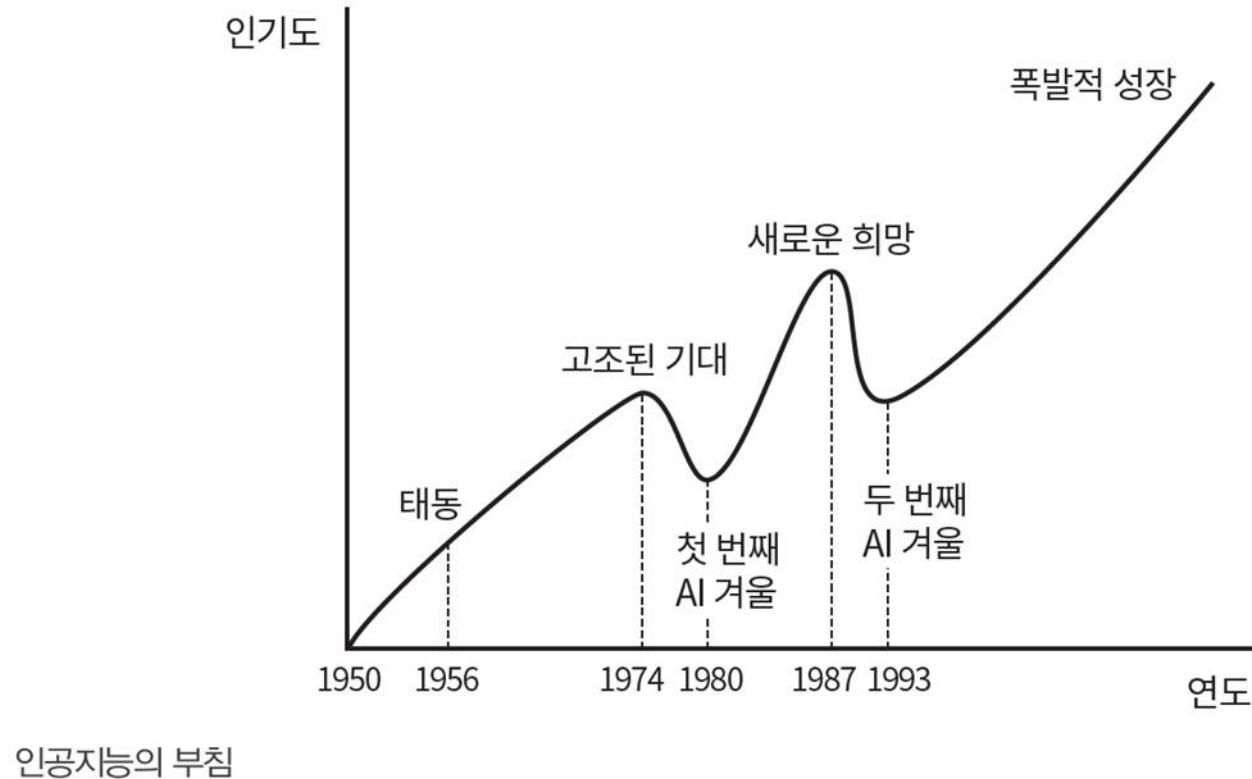
2	6	3	14	17
18	1	13	8	10
16	11	15		24
21	23	9	12	4
7	5	22	20	19

10^{25} 개의 상태

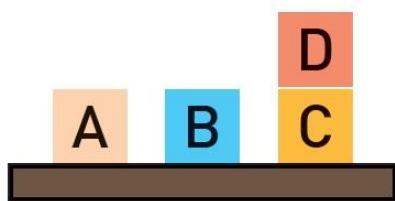
가능한 상태의 개수



첫 번째 AI 겨울

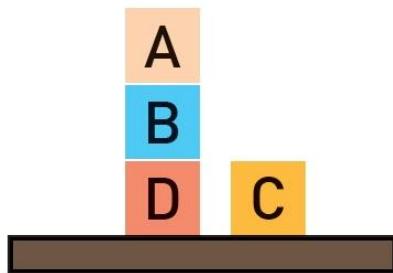


장난감 문제



초기 상태

장난감 문제



목표 상태

7	2	4
5		6
8	3	1

초기 상태

1	2
3	4
6	7

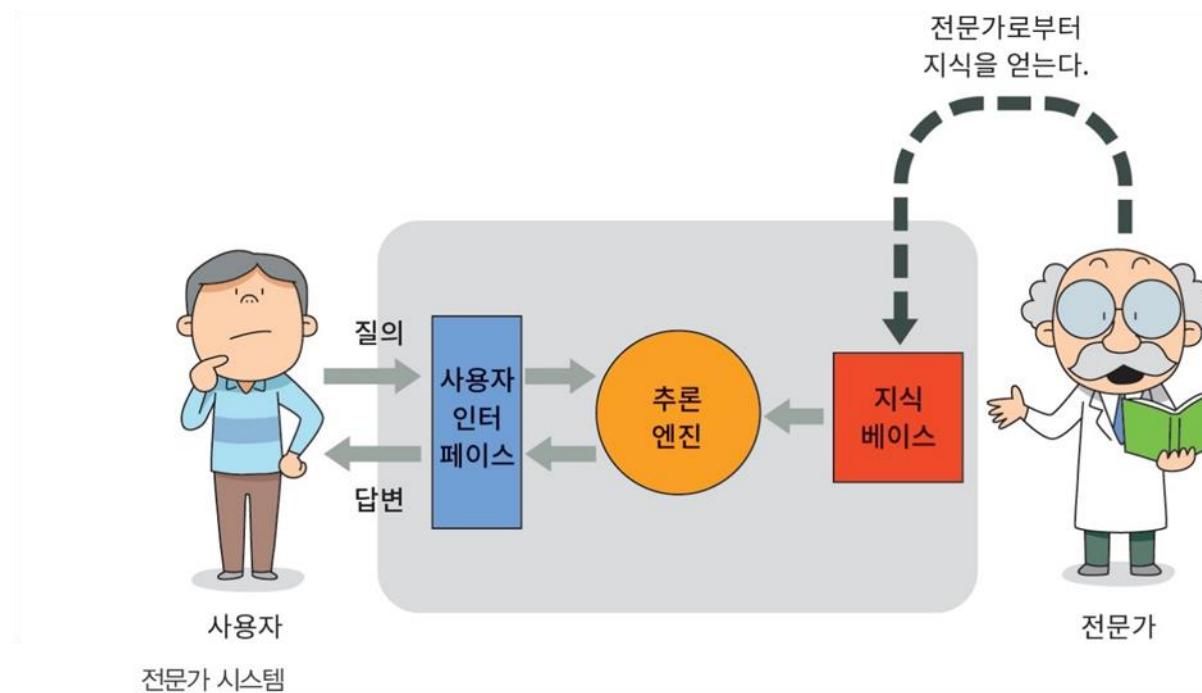
목표 상태

당시의 문제점

- 첫 번째로 1970년대에는 충분한 컴퓨팅 파워가 없었다. 실제로 유용한 결과를 내는데 필요한 CPU의 속도나 충분한 메모리가 없었다.
- 두 번째로 “장난감 문제”가 있다. 인공 지능 분야에서는 지수적 시간에만 풀 수 있는 많은 현실적인 문제가 있다. 따라서 이러한 현실적인 문제에 대한 최적의 솔루션을 찾는 데는 상상할 수 없는 양의 계산 시간이 필요하다.
- 세 번째로 컴퓨터 시각이나 자연어 처리와 같은 많은 인공 지능 응용 프로그램은 전 세계에 대한 엄청난 양의 정보를 필요로 한다. 1970년에는 아무도 이 정도의 데이터베이스를 만들 수 없었고 어떤 프로그램도 이 방대한 정보를 어떻게 학습해야 하는지를 알지 못했다.

전성 시대

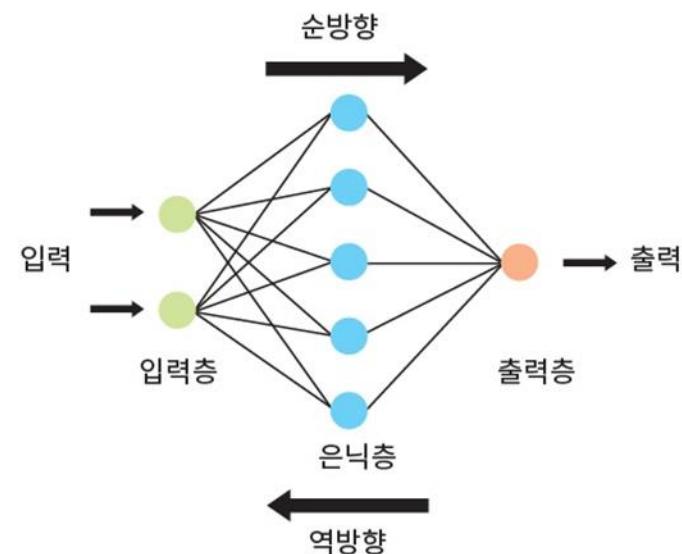
- 연구자들은 이 세상의 모든 문제를 해결할 수 있는 시스템을 개발한다는 생각을 버렸다.
- 이에 새롭게 등장한 시스템이 "전문가 시스템(expert system)"이다.



- **DENDRAL**은 분광계 수치로 화합물을 분석하는 전문가 시스템으로 스탠포드 대학교의 **Edward Feigenbaum**과 그의 학생들에 의해 개발되었다.
- **MYCIN**은 전염성 질환을 진단하고 항생제를 처방하는 전문가 시스템이었다.

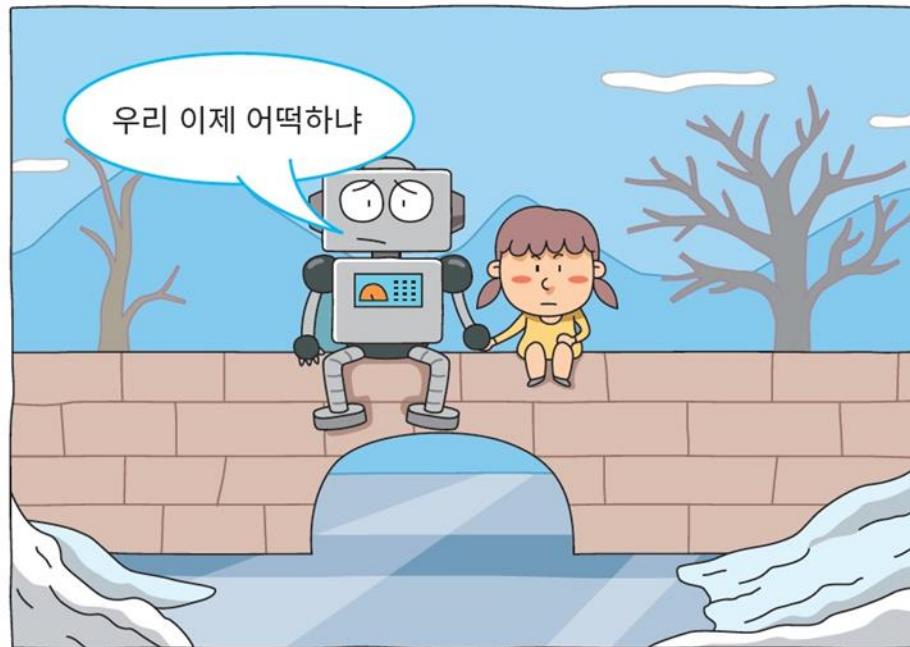
신경망의 부활

- 1982년 물리학자 John Hopfield는 완전히 새로운 방식으로 정보를 학습하고 처리할 수 있는 한 형태의 신경망 (Hopfield Net)을 제안
- Geoffrey Hinton과 David Rumelhart는 "역전파 (backpropagation)"라고 불리는 유명한 학습 방법을 대중화



두 번째 AI 겨울

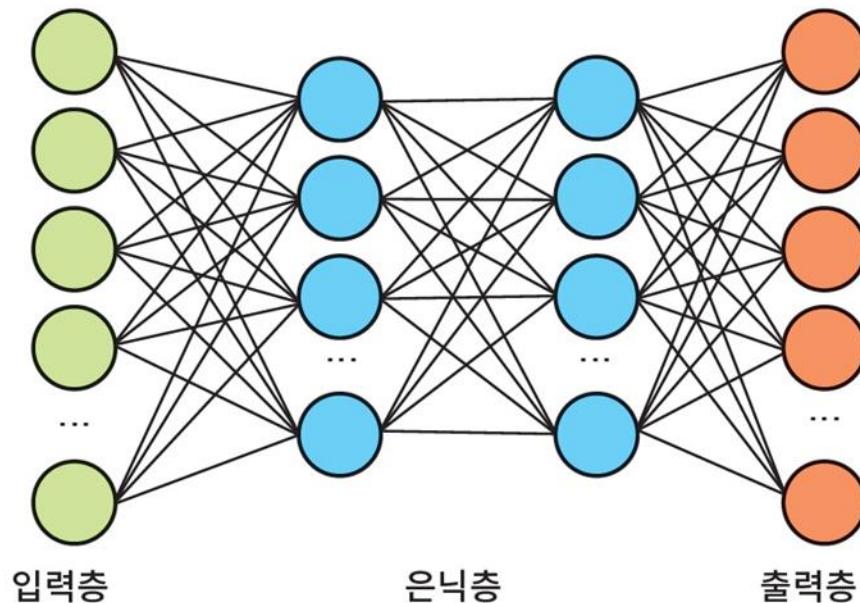
- 전문가 시스템은 유용했지만 몇 가지 특수한 상황에서만 유용함이 밝혀졌다.
- 1980년대 후반, 미국의 전략적 컴퓨팅 구상(**Strategic Computing Initiative**)은 AI에 대한 기금을 잔인하게 삭감했다.



AI의 겨울

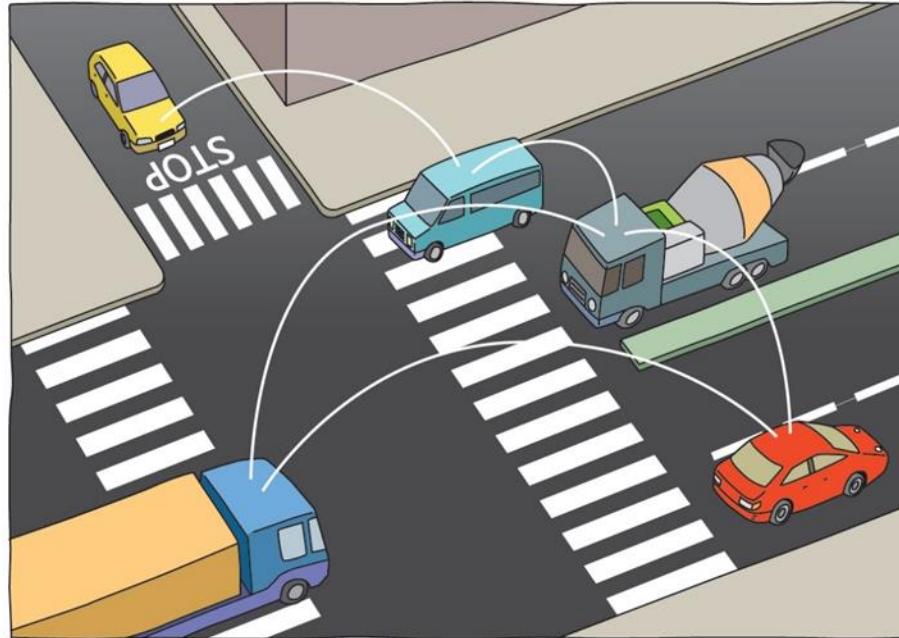
AI의 부활

- 딥러닝(deep learning)은 많은 레이어(layer)가 있는 신경 회로망을 사용하여 데이터의 추상화를 모델링하는 기계 학습의 한 분야이다.



인공지능의 유통 분야

- 자동차 업계에서는 이미지 인식 기술을 바탕으로 한 자율 주행 자동차 개발에 심혈을 기울이고 있다.



연결된 자율주행 자동차의 개념

인공지능의 옆 분야(광고)

- 인공지능은 현재 사용자가 보고 있는 웹사이트의 컨텐츠와 가장 유사한 상품이나 기사를 추천한다.



인공지능 추천 시스템

인공지능의 유행 분야(챗봇)

- 오늘날 챗봇은 Google Assistant 및 Amazon Alexa와 같은 가상 어시스턴트, Facebook Messenger 또는 WeChat과 같은 메시징 앱이나 웹 사이트를 통해 사용된다.



인공지능의 융합 분야 (의료분야)



환자 데이터

의사 노트,
각종 테스트 결과
의학 영상 결과



환자 설문

왓슨의 질문에 대한
환자의 답변



유전자 분석

조직의 DNA
염기서열 정보



진단

환자의 정보를
가능한 모든 의학
정보와 비교



왓슨을 사용한 의료 진단



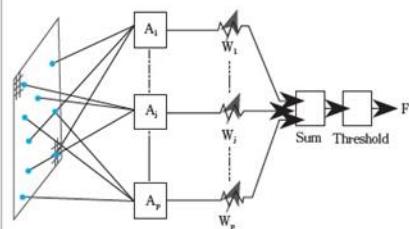
데이터베이스

가이드라인

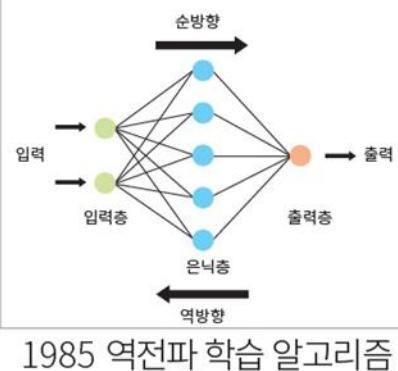
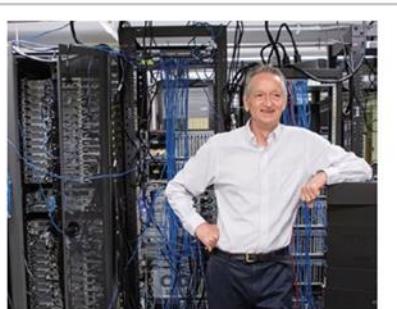
케이스리포트

임상 실험 결과

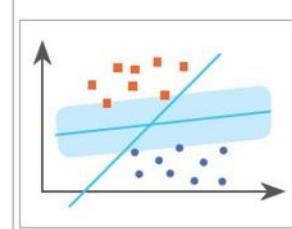
기계 학습의 역사



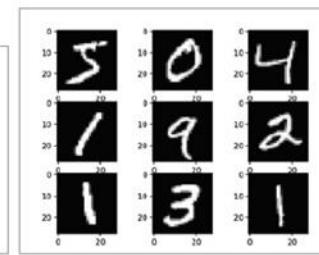
1957 퍼셉트론



1985 역전파 학습 알고리즘



1995 SVM



1998 필기체 인식

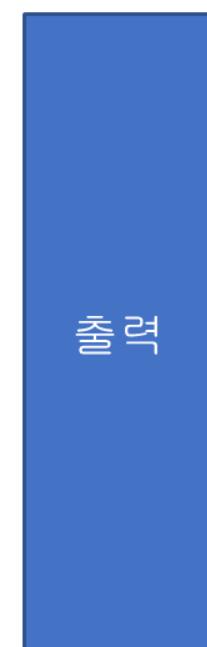
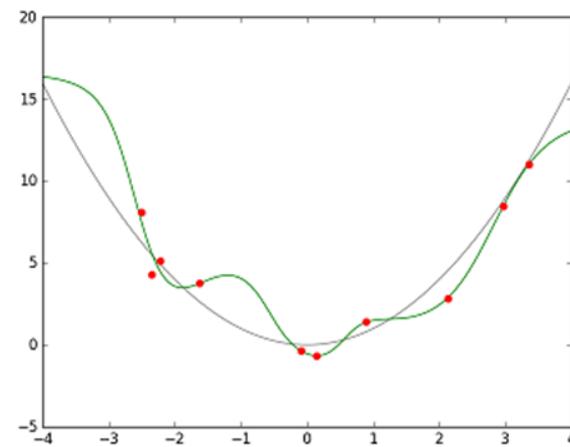
2012
AlexNet wins
ImageNet
IMAGENET

2012
ImageNet에서
우승

기계학습의 역사

기계 학습

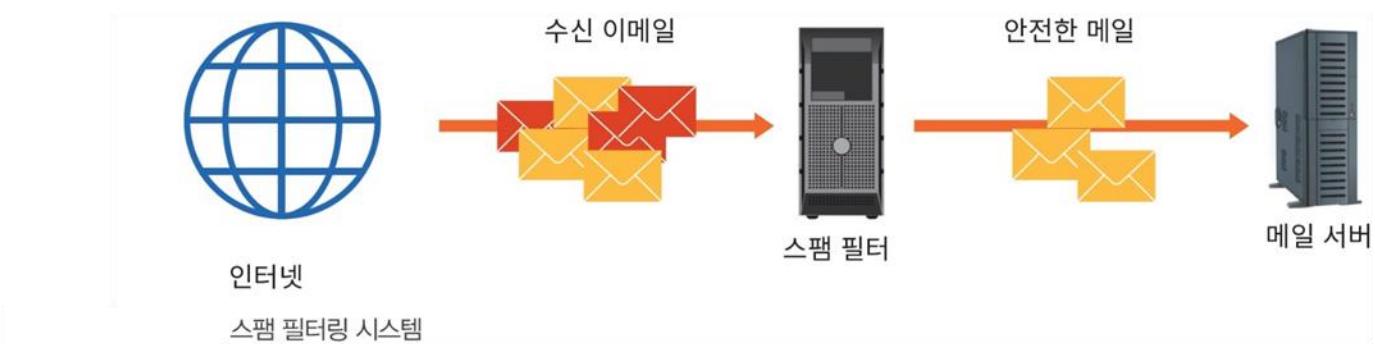
- 기계 학습은 항상 입력을 받아서 출력하는 함수 $y=f(x)$ 를 학습한다고 생각할 수 있다. (함수 근사)



기계 학습(machine learning) == 함수 근사(function approximation)

특징(features)

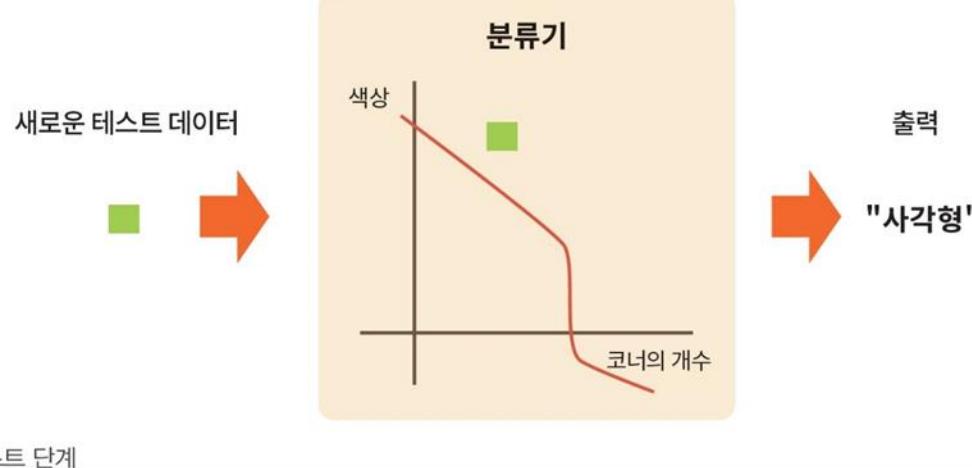
- 특징이란 우리가 학습 모델에게 공급하는 입력이다. 가장 간단한 경우에는 입력 자체가 특징이 된다.
- (예)
 - 이메일에 “검찰”이라는 문자 포함 여부(yes 또는 no)
 - 이메일에 “광고”, “선물 교환권”이나 “이벤트 당첨” 문자열 포함 여부(yes 또는 no)
 - 이메일의 제목이나 본문에 있는 ‘★’과 같은 특수 기호의 개수(정수)
 - ...



레이블과 샘플

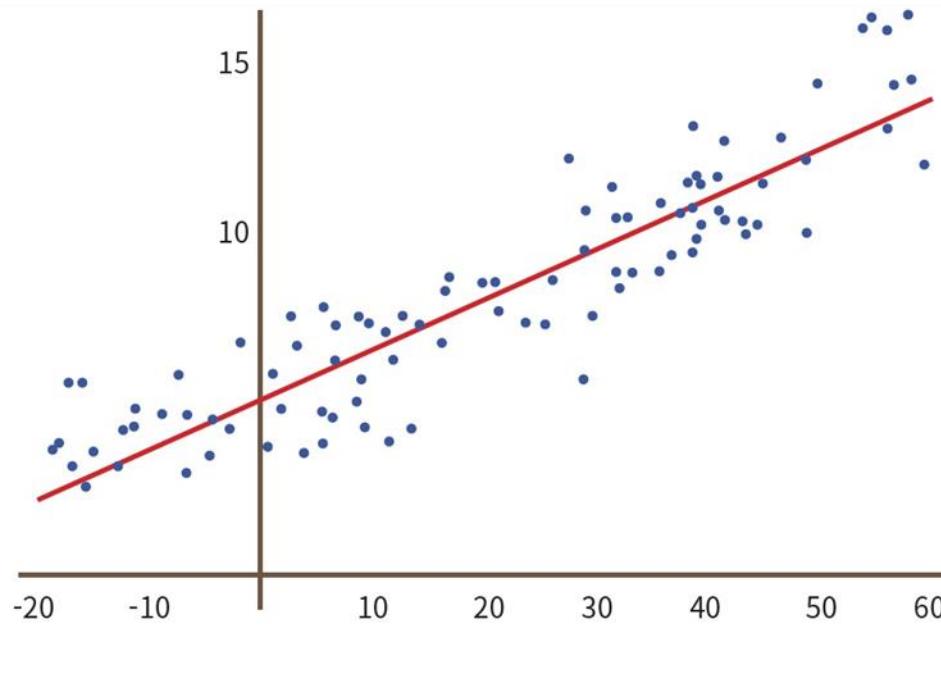
- 레이블(label)
 - $y = f(X)$ 에서 y 변수에 해당한다.
 - 예를 들어서 농작물의 향후 가격, 사진에 표시되는 동물의 종류, 동영상의 의미 등 무엇이든지 레이블이 될 수 있다.
- 샘플, 또는 예제
 - 샘플은 기계 학습에 주어지는 특정한 예이다. $y = f(X)$ 에서 X 에 해당한다. 레이블이 있는 샘플도 있고 레이블이 없는 샘플도 있다. 지도 학습을 시키려면 레이블이 있어야 한다.

학습 데이터와 테스트 데이터



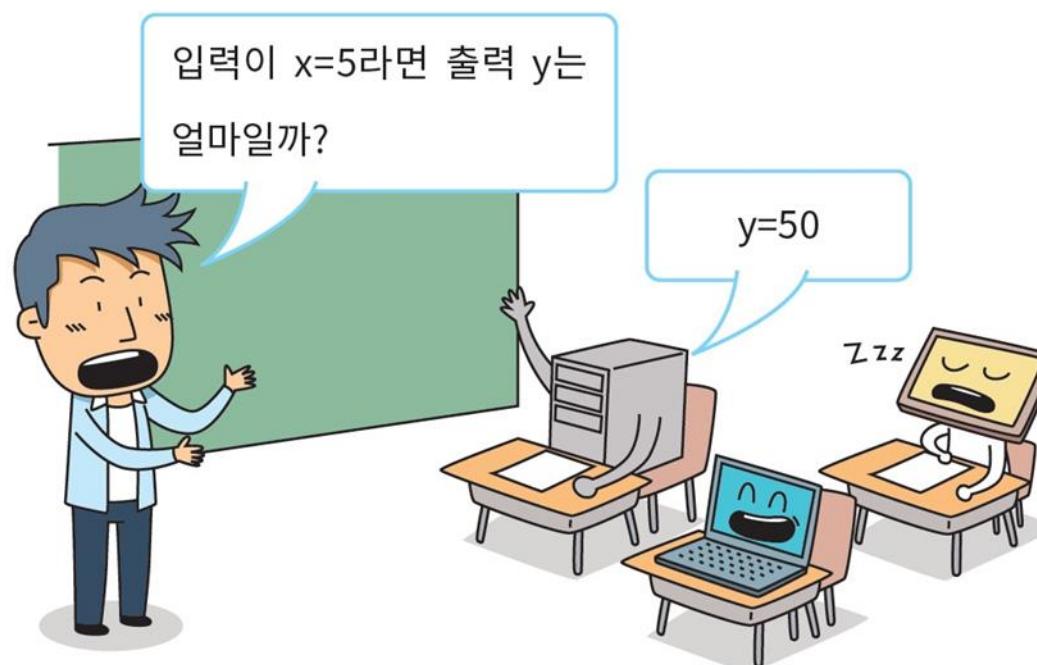
- 학습(learning)은 모델을 만들거나 배우는 것을 의미한다.
- 예측(prediction)은 학습된 모델을 레이블이 없는 샘플에 적용하는 것을 의미한다. 즉 학습된 모델을 사용하여 유용한 예측(y')을 해내는 것이다.

- 회귀(regression) : 회귀에서는 입력과 출력이 모두 실수이다.
 - “사용자가 이 광고를 클릭할 확률이 얼마인가요?”
 -

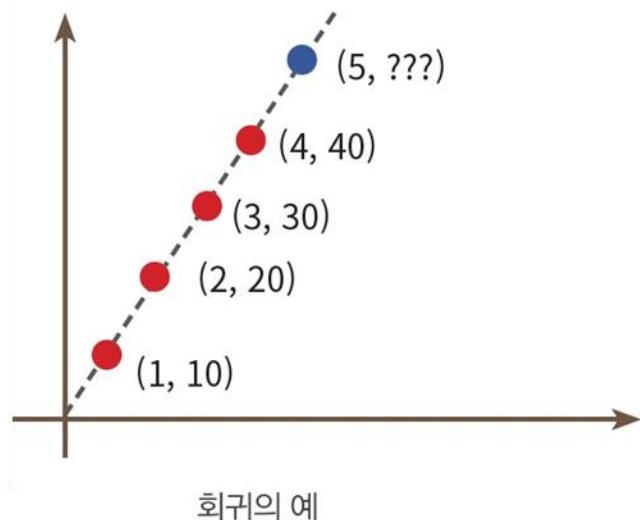


- 회귀는 실수 입력(x)과 실수 출력(y)이 주어질 때, 입력에서 출력으로의 매팅 함수를 학습하는 것이라 할 수 있다.

$$y = f(x)$$



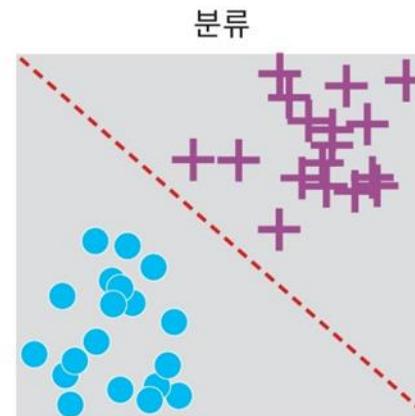
회귀의 예



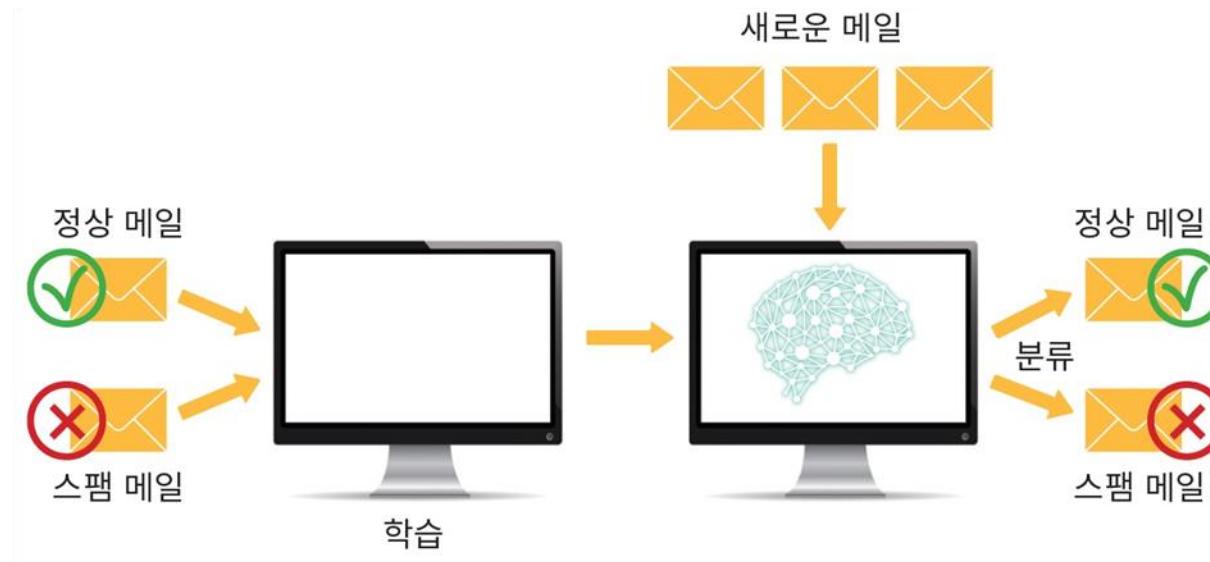
- 분류(classification): 입력을 두 개 이상의 레이블(유형)으로 분할하는 것
- 해당 모델을 학습시킬 때 우리는 레이블을 제공해야 한다.



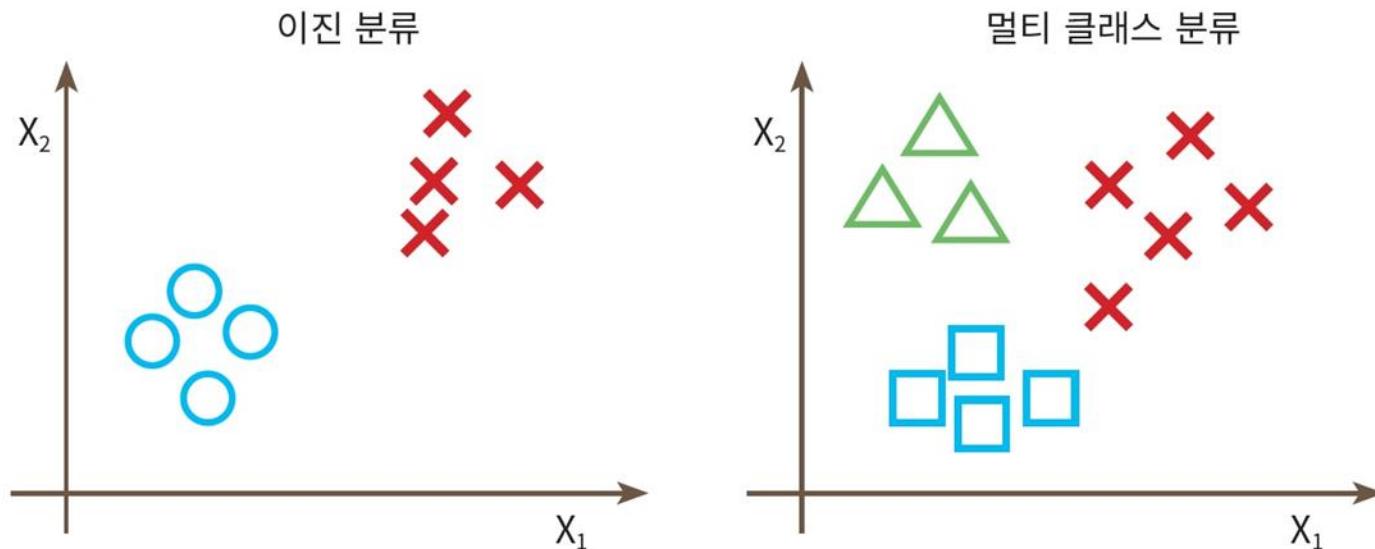
분류



- 앞에 나왔던 식 $y = f(x)$ 에서 출력 y 가 이산적(discrete)인 경우에 이것을 분류 문제(또는 인식 문제)라고 부른다.
- 분류에서는 입력을 2개 이상의 클래스로 나누는 것이다.

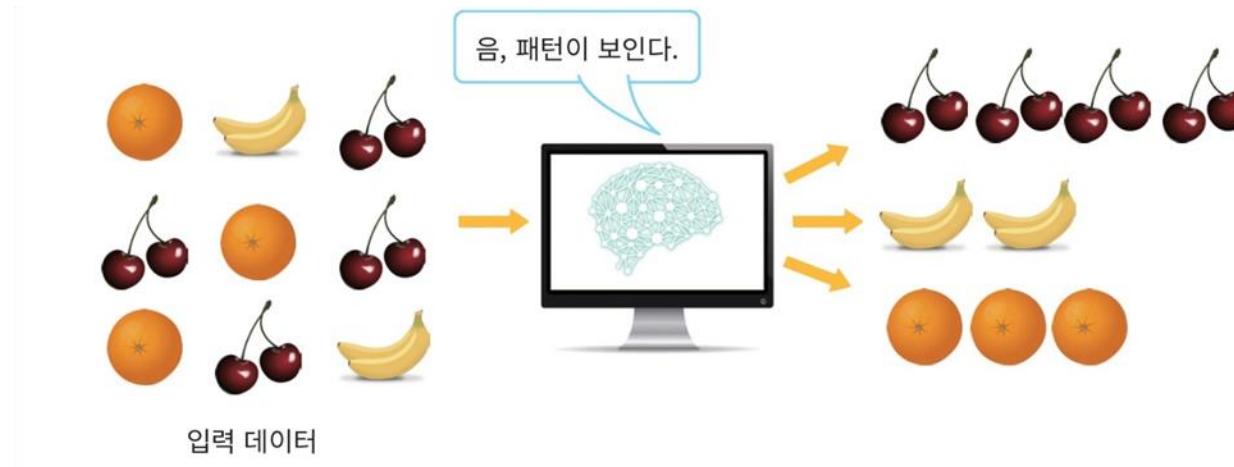


- 분류는 지도 학습의 형태로 이루어지는 것이 일반적이다.
- 분류를 수행하기 위한 일반적인 알고리즘에는 신경망, kNN(k-nearest neighbor), SVM(Support Vector Machine), 의사 결정 트리 등이 포함된다.

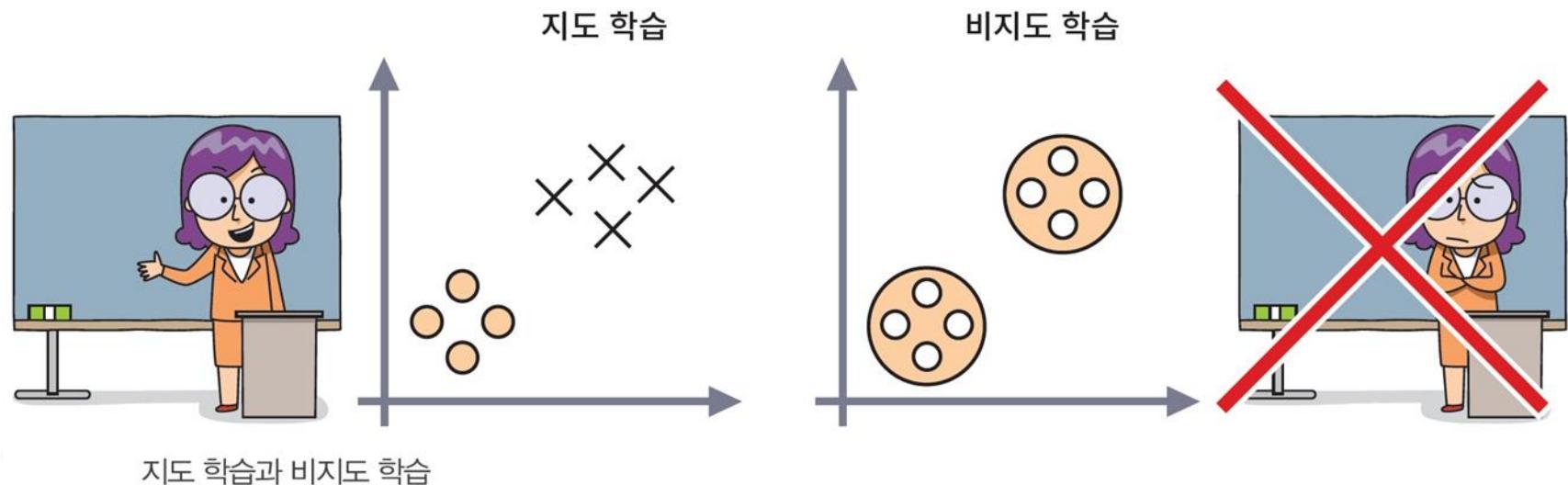


비지도 학습

- 비지도 학습(unsupervised Learning)은 “교사” 없이 컴퓨터가 스스로 입력들을 분류하는 것을 의미한다. 식 $y = f(x)$ 에서 레이블 y 가 주어지지 않는 것이다.
- 데이터들의 상관도를 분석하여 유사한 데이터들을 모을 수 있다.

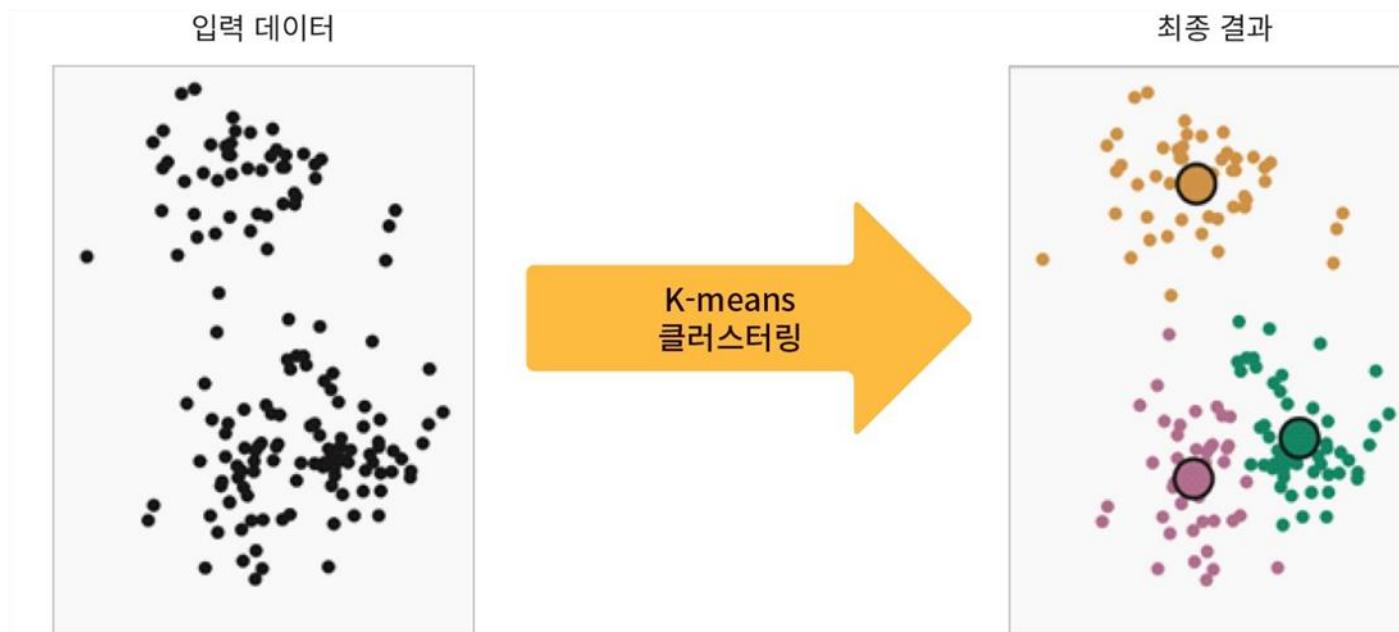


비지도 학습(k-means 클러스터링)

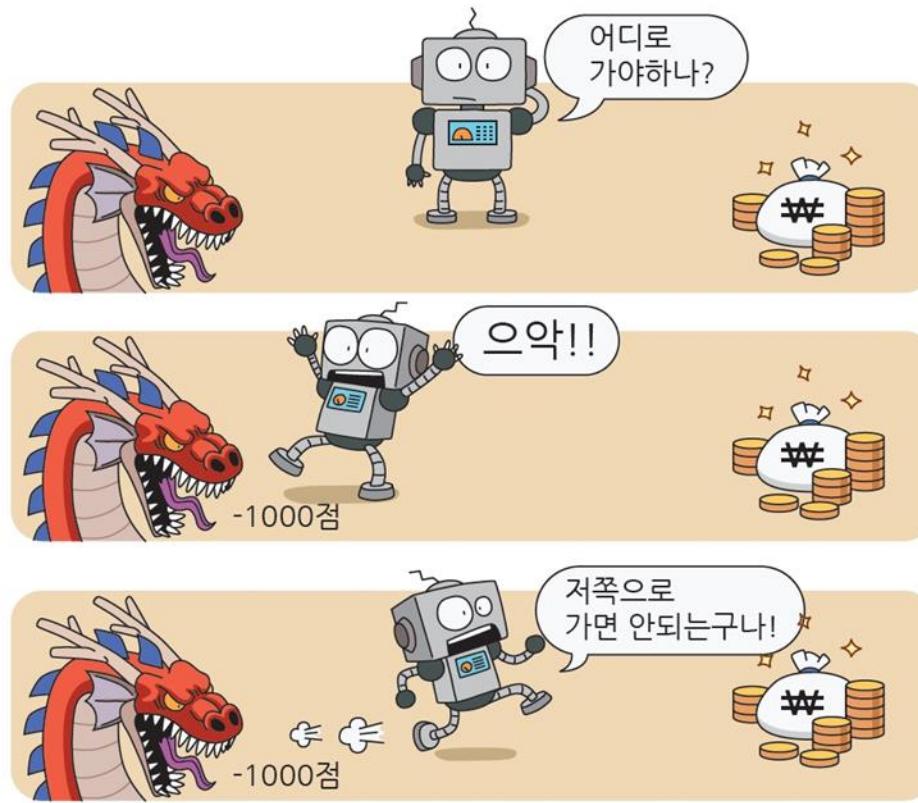


비지도 학습

- 가장 대표적인 비지도 학습이 클러스터링(군집화, clustering)이다.
- 클러스터링이란 데이터간 거리를 계산하여서 입력을 몇 개의 그룹으로 나누는 방법이다.

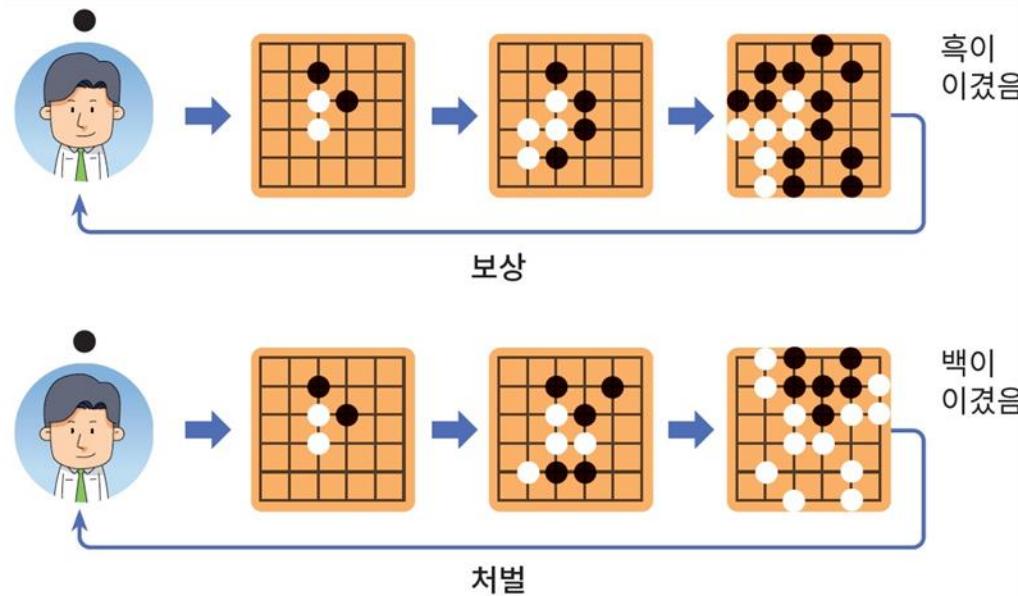


- 강화 학습(Reinforcement Learning)에서는 컴퓨터가 어떤 행동을 취할 때마다 외부에서 처벌이나 보상이 주어진다.



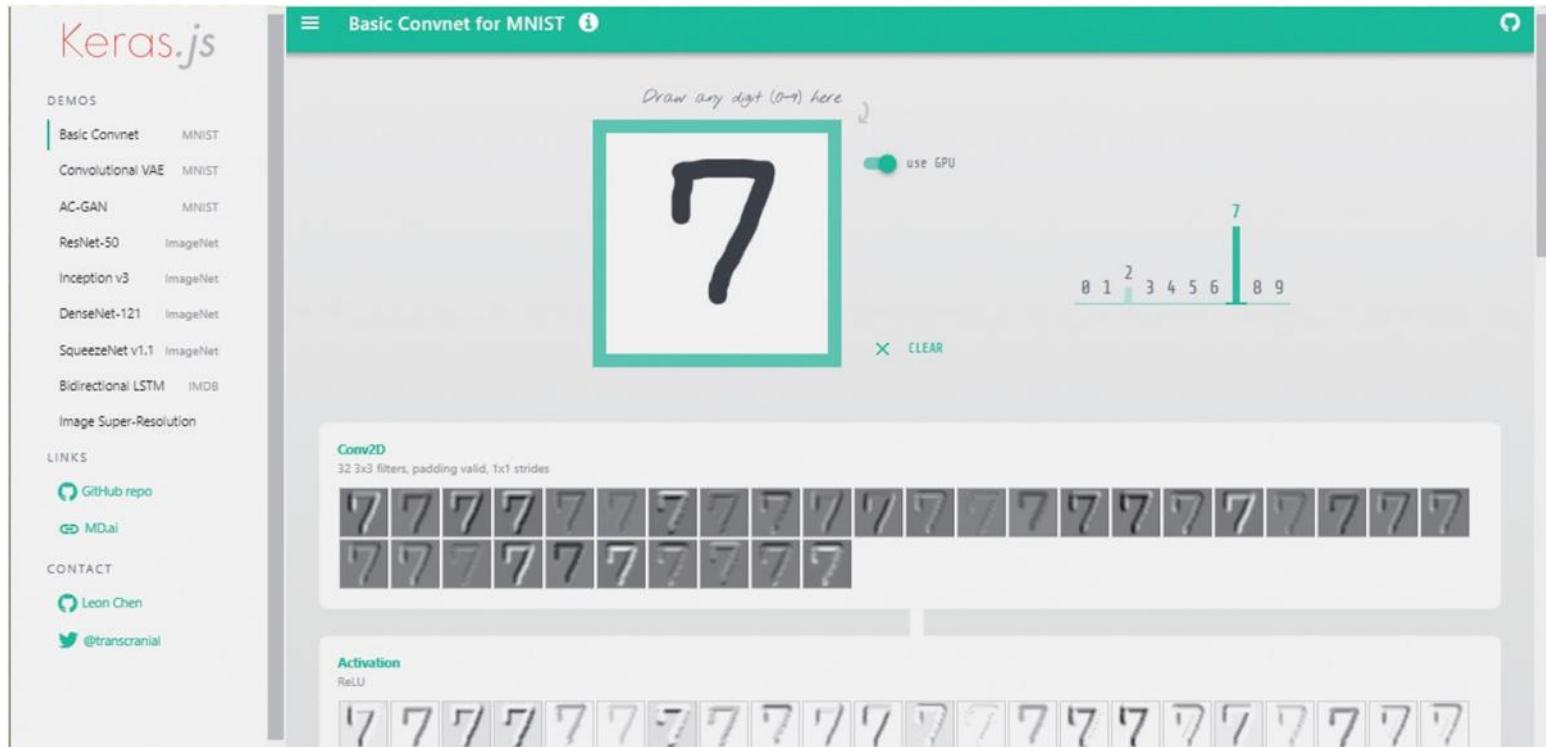
강화 학습

- 알파고 최종 버전도 강화 학습 사용
- 게임에서 많이 사용된다(예: Frozen Lake).



Lab: 기계 학습 체험하기

- <https://transcranial.github.io/keras-js/#/>



Lab: 기계 학습 체험하기

- <https://transcranial.github.io/keras-js/#/>

Keras.js

50-layer Residual Network, trained on ImageNet [i](#)

Enter a valid image URL or select an image from the dropdown.

enter image url
<https://i.imgur.com/0B8y6MR.jpg>

select image
dog

use GPU

visualization
None

inference time: 352.5 ms (2.8 fps)

Newfoundland | 77%

Tibetan mastiff | 21%

Leonberg | 0%

groenendael | 0%

chow | 0%

ResNet-50 | ImageNet

AC-GAN | MNIST

Convolutional VAE | MNIST

Basic Convnet | MNIST

Inception v3 | ImageNet

DenseNet-121 | ImageNet

SqueezeNet v1.1 | ImageNet

Bidirectional LSTM | IMDB

Image Super-Resolution

LINKS

[GitHub repo](#)

[MD.ai](#)

CONTACT

[Leon Chen](#)

[@transcranial](#)

```
graph TD; InputLayer[InputLayer  
shape: [224,224,3]] --> Conv2D[Conv2D  
64 7x7 filters, 2x2  
striding, pad to same  
borders]; Conv2D --> BatchNormalization[BatchNormalization]; BatchNormalization --> Activation[Activation  
relu];
```

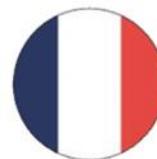
프로그래머로서 기계 학습의 실용적인 가치

- 첫 번째는 프로그래밍 시간을 줄일 수 있다는 점이다.
- 예를 들어서 맞춤법 오류를 수정하는 프로그램을 개발한다고 하자.
 - 전통적인 방법: 많은 맞춤법 규칙을 이용하여 작성할 수 있다. -> 상당한 시간이 필요
 - 기계 학습 이용: 많은 예제만 있다면 학습시켜서 빠른 시간 안에 신뢰성있는 프로그램을 완성할 수 있다.



프로그래머로서 기계 학습의 실용적인 가치

- 두 번째로 맞춤형 제품을 쉽게 개발할 수 있다.
- 예를 들어서 여러분이 한국어 맞춤법 수정 프로그램이 작성하여 가지고 있다고 하자. 제품이 성공적이어서 30개국 언어 버전으로 확장하려고 한다.
 - 전통적인 방법: 각 언어마다 새로 작성하려면 수년 이상의 엄청난 시간이 필요하다.
 - 기계 학습 이용: 너무나도 쉽다 예제만 있으면 된다.



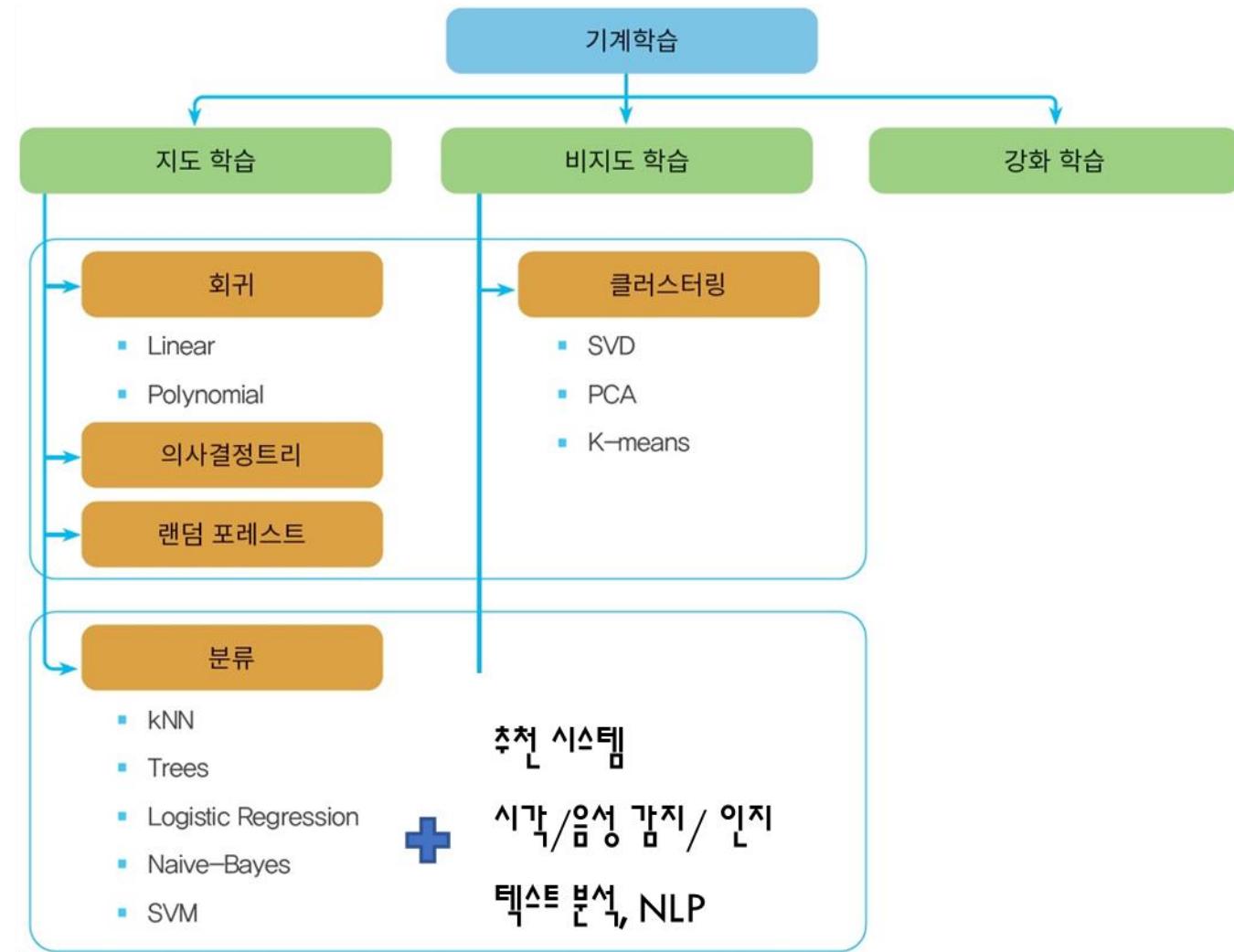
프로그래머로서 기계 학습의 실용적인 가치

- 세 번째로 기계 학습은 프로그래머로 시도할 알고리즘이 떠오르지 않는 문제들을 해결할 수도 있다.
- 예를 들어서 컴퓨터가 사람의 얼굴을 인식하는 프로그램을 작성
 - 전통적인 방법: 이런 문제를 작성하려면 컴퓨터 시각 분야의 수많은 지식과 경험이 필요한 작업이다.
 - 기계 학습 이용: 프로그램에 수많은 예제만 보여주기만 하면 문제가 해결된다. 편리하지 않은가?

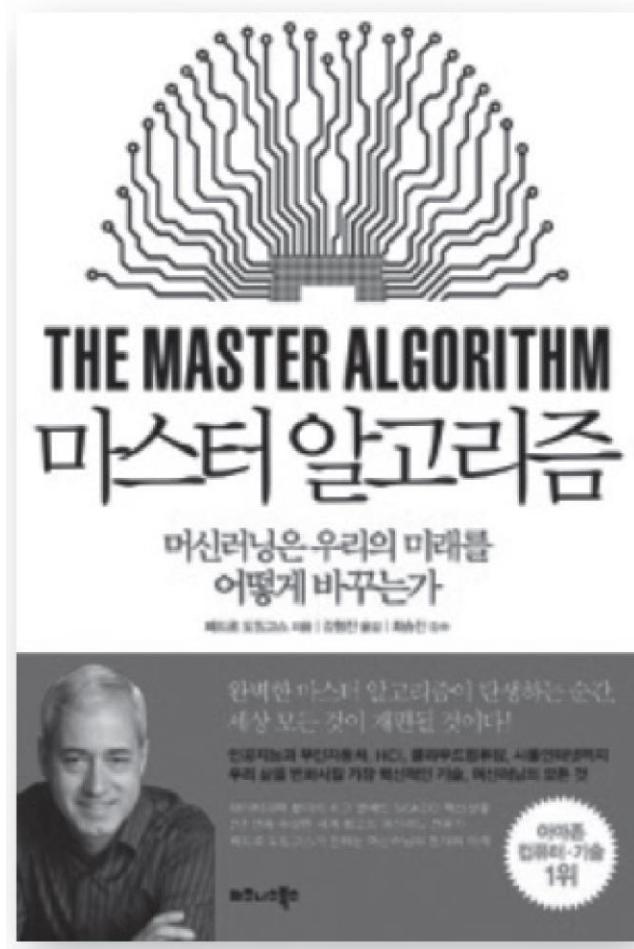


기계 학습의 종류

신경망은 모든 기계학습 분야에서 사용될 수 있습니다!!!



Best Book for normal readers



기호주의: 결정 트리 등

연결주의: 신경망/딥러닝

유전 알고리즘

베이지안 통계

유추주의: KNN, 서포트 벡터 머신

Data 전쟁

- **Which one would be important?**

Data or ML Algorithm?

- Garbage in, Garbage out
- Optimized ML Algorithm and Model parameter is crucial as well as the matter of data !!
- Extensive and Various data can guarantee the qualities.

ML 단점

- 데이터에 너무 의존적입니다. (Garbage In , Garbage Out)
- 학습시에 최적의 결과를 도출하기 위해 수립된 머신러닝 모델은 실제 환경 데이터 적용 시 과적합 되기 쉽습니다.
- 복잡한 머신러닝 알고리즘으로 인해 도출된 결과에 대한 논리적인 이해가 어려울 수 있습니다 (머신러닝은 블랙 박스).
- **데이터만 집어 넣으면 자동으로 최적화된 결과를 도출할 것이라는 것은 환상입니다**(특정 경우에는 개발자가 직접 만든 코드보다 정확도가 더 떨어질수 있습니다). 끊임없이 모델을 개선하기 위한 노력이 필요하기 때문에 데이터의 특성을 파악하고 최적의 알고리즘과 파라미터를 구성할 수 있는 고급 능력이 필요합니다.

왜 데이터 수집에 열광하는가



구글과 페이스북에서 보유하고 있는 데이터로 최적화 된 머신러닝 모델을 다른 회사가 이길 수 있을까? (더 좋은 알고리즘을 가지고 있더라도?)



다양하고 광대한 데이터를 기반으로 만들어진 머신러닝 모델은 더 좋은 품질을 약속합니다. 앞으로 많은 회사의 경쟁력은 어떠한 품질의 머신러닝 모델을 가지고 있느냐에 결정 될 수 있습니다.

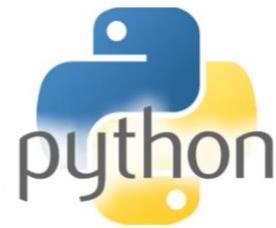


파이썬 VS R for ML (통계분석 관점)

R은 통계 전용 프로그램 언어로서 SPSS,SAS, MATLAB 등 전통적인 통계 및 마이닝 패키지의 고비용으로 신음(?)하던 통계 전문가들이 이를 개선하고자 만든 언어입니다



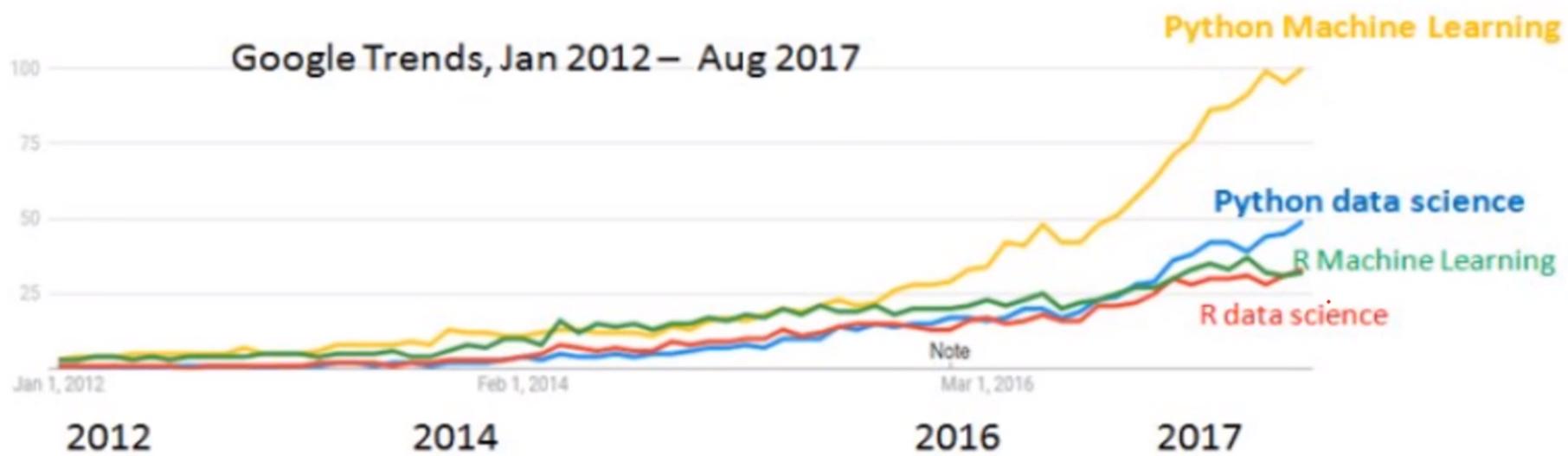
VS



지극히 개인적인 판단이지만, 개발 언어에 익숙하지 않으나 통계 분석에 능한 현업 사용자라면 머신러닝을 위해 R을 선택하는 것이 더 나을 수도 있습니다. 파이썬도 굉장히 직관적인 언어지만, R의 경우 통계 분석을 위해 특화된 언어이며 무엇보다도 오랜 기간 동안 많은 R 사용자들이 생성하고 검증해온 다양하고 많은 통계 패키지를 보유하고 있는 것이 가장 큰 장점입니다.

파이썬 VS R for ML –Google trends

하지만 이제 머신러닝을 시작하려는 사람이라면, 특히 개발자라면 R보다는 파이썬을 권하고 싶습니다

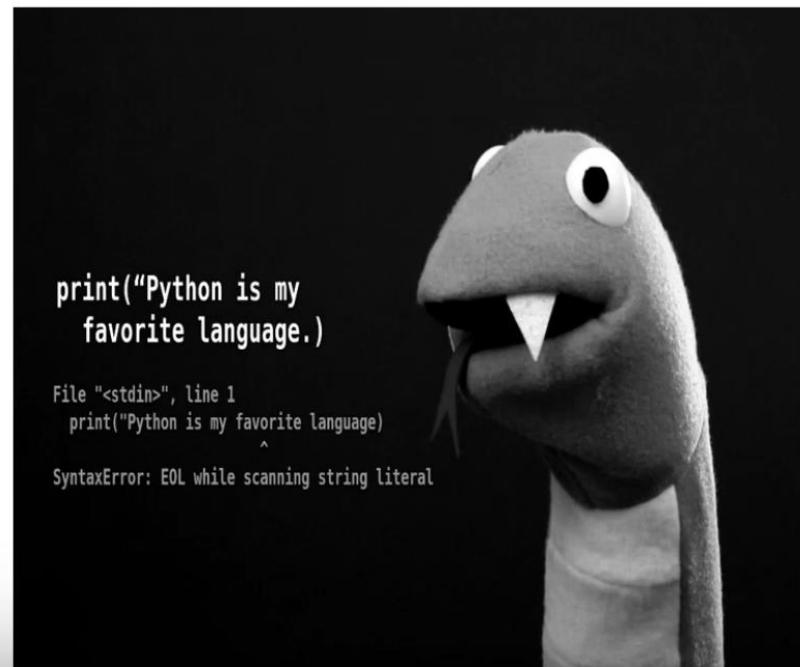


ML+ Python의 인기

Python 은 소리없이 프로그래밍 세계를 점령하고 있는 Language .

복잡한 세상 , 복잡한 코드는 가라

- 쉽고 뛰어난 개발 생산성으로 전 세계의 개발자들 뿐만 아니라
Academy 나 타 영역의 인재들도 Python 선호
- Google , Facebook 등 유수의 IT 업계에서도 Python 의 높은
생산성으로 인해 활용도가 매우 높음. (특히 Google)
- 오픈 소스 계열의 전폭적인 지원을 받고 있음.
- 놀라울 정도의 많은 라이브러리 지원은 어떠한 유형의 개발도
쉽게 가능 (역으로 선택의 자유가 많아서 오히려 머리가 아플 정도)
- Interpreter Language의 특성상 속도는 느리지만 쉽고 유연한
특징으로 인해 데스크탑 , 서버 , 네트워크 , 시스템 , IOT 등 다양한
영역에서 사용되고 있음.



Python 확장성 연계 호환성

많은 라이브러리, 뛰어난 생산성을 가지는 Python 언어



Machine
Learning

- 분석 영역을 넘어서 ML 기반의 다양한 Application 개발이 쉽게 가능
- 기존 Application 과의 연계도 쉬움 (서로 다른 언어로 개발된 Application 의 경우 Rest API)
- Enterprise 아키텍처에도 연계, 확장 가능. Microservice 실시간 연계 등.

Python강점(Deep learning으로 진격)

- 유수의 Deep Learning Framework 이 Python 기반으로 작성(tensorflow Backend 는 성능때문에 C/C++ 로 작성).
- 대부분의 Deep Learning 관련 Tutorial , 설명 자료들이 Python 으로 작성되어 제공.
- 현 시점에서 Deep Learning을 활용하기에 가장 좋은 시작점은 Python



TensorFlow



파이썬 머신러닝 생태계 주요 패키지

머신러닝
패키지

배열/선형대수
/통계 패키지

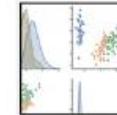
데이터 핸들링

시각화

대화형 파이썬
툴



Seaborn



파이썬 머신러닝을 위한 S/W 설치

anaconda? Or pip?

파이썬 머신러닝을 위한 패키지를 설치하는 가장 쉬운 방법은 anaconda를 이용하는 것입니다

<https://www.anaconda.com/download/>



The screenshot shows the 'Download Anaconda Distribution' page for version 5.3.1. At the top, there are links for 'What is Anaconda?', 'Products', 'Support', 'Resources', 'About', and 'Downloads'. A green 'Downloads' button is highlighted. Below this, the text 'Download Anaconda Distribution' is displayed, along with the release date 'Version 5.3.1 | Release Date: November 19, 2018'. A 'Download For:' section shows icons for Windows, Mac, and Linux. At the bottom, there are three sections: 'High-Performance Distribution', 'Package Management', and 'Portal to Data Science'.

High-Performance Distribution
Easily install 1400+ [data science packages](#)

Package Management
Manage packages, dependencies and environments with [conda](#)

Portal to Data Science
Uncover insights in your data and create interactive visualizations



The screenshot shows the 'Anaconda 5.3.1 For Windows Installer' page. It features two main download options: 'Python 3.7 version *' and 'Python 2.7 version *'. Each option has a green 'Download' button. Below each button, there are two links for '64-Bit Graphical Installer' and '32-Bit Graphical Installer'.

Anaconda 5.3.1 For Windows Installer

Python 3.7 version *

[Download](#)

64-Bit Graphical Installer (633 MB) ⓘ
32-Bit Graphical Installer (510 MB)

Python 2.7 version *

[Download](#)

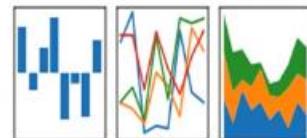
64-Bit Graphical Installer (580 MB) ⓘ
32-Bit Graphical Installer (458 MB)

머신러닝을 위한 넘파이와 판다스의 중요성



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

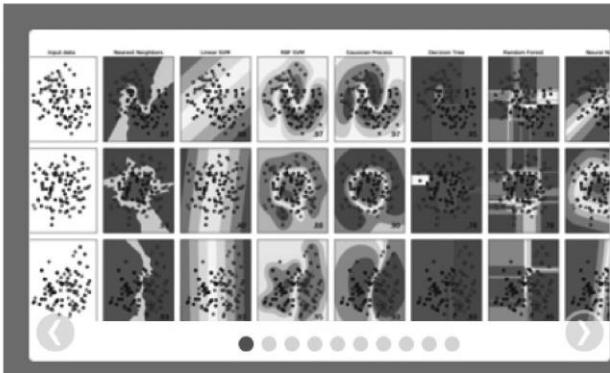


- 머신러닝 애플리케이션 구현에서 다양한 데이터의 추출/가공/변환이 상당한 영역을 차지하고 데이터 처리 부분은 대부분 넘파이와 판다스의 몫.
- 사이킷런이 넘파이 기반에서 작성됐기 때문에 넘파이의 기본 프레임워크를 이해하지 못하면 사이킷런 역시 실제 구현에서 많은 벽에 부딪힐 수 있음
- 사이킷런은 API 구성이 매우 간결하고 직관적이어서 이를 이용한 개발 또한 상대적으로 쉽지만 넘파이와 판다스 API는 더 방대하기 때문에 이를 익히는데 시간이 많이 소모 될 수 있음. 하지만 머신러닝을 위해서 이들을 많은 시간을 들여 전문적으로 공부하는 것은 효율적이지 못함.
- 넘파이와 판다스에 대한 기본 프레임워크와 중요 API만 습득하고, 일단 코드와 부딪쳐 가면서 모르는 API에 대해서는 인터넷 자료를 통해 체득하는 것이 머신러닝뿐만 아니라 넘파이와 판다스에 관한 이해를 넓히는 더 빠른 방법임.

Chapter 02

사이킷 런으로

시작하는 머신러닝



scikit-learn

Machine Learning in Python

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Scikit-learn 소개와 특징.

Classification

Identifying to which category an object belongs to.

Applications: Spam detection, Image recognition.

Algorithms: SVM, nearest neighbors, random forest, ...

— Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.

Algorithms: SVR, ridge regression, Lasso, ...

— Examples

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes.

Algorithms: k-Means, spectral clustering, mean-shift, ...

— Examples

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency

Algorithms: PCA, feature selection, non-negative matrix factorization.

— Examples

Model selection

Comparing, validating and choosing parameters and models.

Goal: Improved accuracy via parameter tuning

Modules: grid-search, cross-validation, metrics.

— Examples

Preprocessing

Feature extraction and normalization.

Application: Transforming input data such as text for use with machine learning algorithms.

Modules: preprocessing, feature extraction.

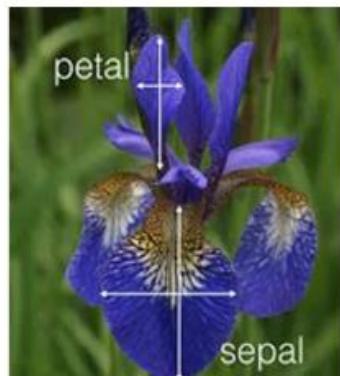
— Examples

- 파이썬 기반의 다른 머신러닝 패키지도 사이킷런 스타일의 API를 지향할 정도로 쉽고 가장 편리한 API를 제공합니다
- 머신러닝을 위한 매우 다양한 알고리즘과 개발을 위한 편리한 프레임워크와 API를 제공합니다
- 오랜 기간 실전 환경에서 검증됐으며, 매우 많은 환경에서 사용되는 성숙한 라이브러리입니다
- 주로 Numpy와 Scipy 기반 위에서 구축된 라이브러리입니다

첫번째 머신러닝 만들어 보기 – Iris 꽃

사이킷런을 통해 첫 번째로 만들어볼 머신러닝 모델은 붓꽃 데이터 세트로 붓꽃의 품종을 분류 (Classification) 하는 것입니다. 붓꽃 데이터 세트는 꽃잎의 길이와 너비, 꽃받침의 길이와 너비 피처 (Feature)를 기반으로 꽃의 품종을 예측하기 위한 것입니다.

붓꽃 데이터 피처



- Sepal length
- Sepal width
- Petal length
- Petal width

붓꽃 데이터 품종(레이블)



Setosa



Vesicolor



Virginica

지도 학습 - 분류

학습
데이터



테스트
데이터

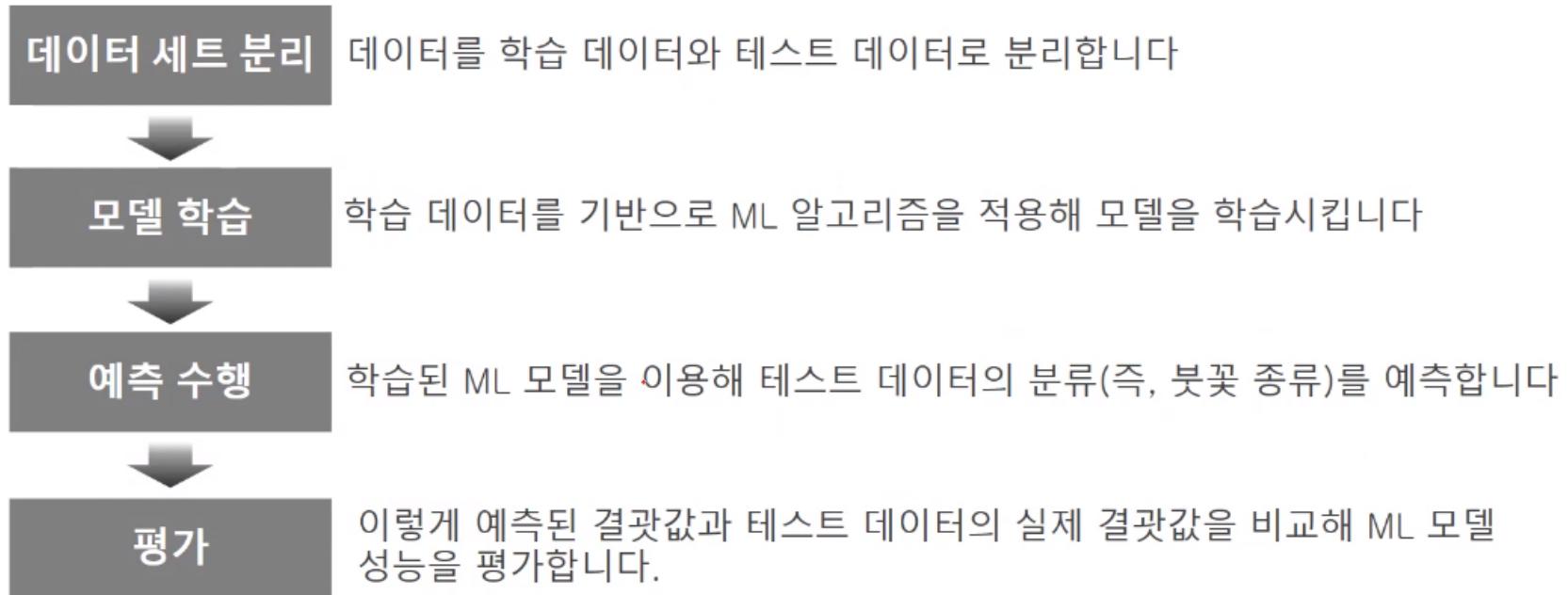
피처				레이블
꽃받침 길이	꽃받침 너비	꽃잎 길이	꽃잎 너비	Iris 꽃 종류
5.1	3.5	1.4	0.2	Setosa
4.9	3.0	1.4	0.2	Setosa
6.4	3.5	4.5	1.2	Versicolor

꽃받침 길이	꽃받침 너비	꽃잎 길이	꽃잎 너비	Iris 꽃 종류는?
5.3	3.2	1.1	0.1	?
4.2	2.0	2.4	0.4	?
6.5	3.8	5.5	1.1	?

분류(Classification)는 대표적인 지도학습(Supervised Learning) 방법의 하나입니다. 지도학습은 학습을 위한 다양한 피처와 분류 결정값인 레이블(Label) 데이터로 모델을 학습한 뒤, 별도의 테스트 데이터 세트에서 미지의 레이블을 예측합니다.

즉 지도학습은 명확한 정답이 주어진 데이터를 먼저 학습한 뒤 미지의 정답을 예측하는 방식입니다. 이 때 학습을 위해 주어진 데이터 세트를 학습 데이터 세트, 머신러닝 모델의 예측 성능을 평가하기 위해 별도로 주어진 데이터 세트를 테스트 데이터 세트로 지칭합니다

붓꽃 데이터 분류 예측 프로세스



붓꽃 데이터 Load

sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	label
0	5.1	3.5	1.4	0
1	4.9	3.0	1.4	0
2	4.7	3.2	1.3	0

```
X_train, X_test, y_train, y_test = train_test_split(iris_data, iris_label,  
                                                test_size=0.2, random_state=11)
```

DecisionTreeClassifier 객체 생성

```
dt_clf = DecisionTreeClassifier(random_state=11)
```

학습 수행

```
dt_clf.fit(X_train, y_train)
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,  
                      max_features=None, max_leaf_nodes=None,  
                      min_impurity_decrease=0.0, min_impurity_split=None,  
                      min_samples_leaf=1, min_samples_split=2,  
                      min_weight_fraction_leaf=0.0, presort=False, random_state=11,  
                      splitter='best')
```

학습이 완료된 DecisionTreeClassifier 객체에서 테스트 데이터 세트로 예측 수행.
pred = dt_clf.predict(X_test)

```
from sklearn.metrics import accuracy_score  
print('예측 정확도: {:.4f}'.format(accuracy_score(y_test, pred)))
```

예측 정확도: 0.9333

Overall

학습
데이터

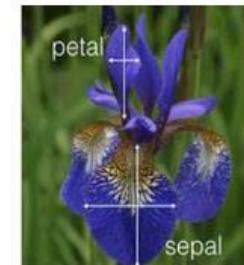
번호	피처				레이블 Iris 꽃 종류
	꽃받침 길이	꽃받침 너비	꽃잎 길이	꽃잎 너비	
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
.....				
50	6.4	3.5	4.5	1.2	Versicolor
.....				
150	5.9	3.0	5.0	1.8	Virginica

테스트
데이터

번호	꽃받침 길이	꽃받침 너비	꽃잎 길이	꽃잎 너비	Iris 꽃 종류는?
1	5.1	3.5	1.4	0.2	?
.....					?
50	6.4	3.5	4.5	1.2	?

학습 데이터로 모델 학습

모델 학습



학습 모델 통해 테스트
데이터의 레이블 값 예측

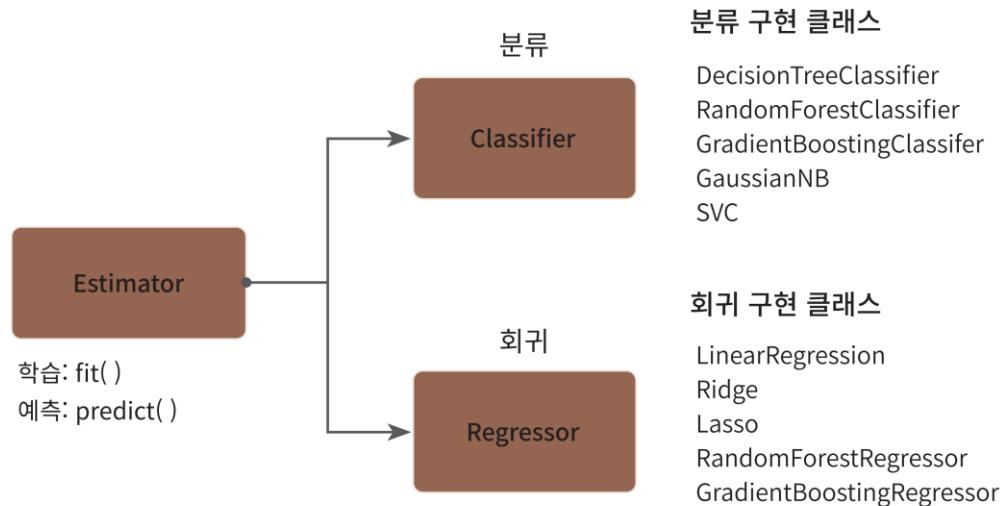
학습된 모델로
테스트 데이터
예측

예측된 레이블 값과 실제
레이블 값 예측 정확도 평가

평가

사이킷런 기반 프레임워크 익히기

- 학습을 위해 `fit()`, 학습된 모델의 예측을 위해 `predict()` 제공
- Classification and Regression의 다양한 알고리즘 구현
- Classifier 와 Regressor가 구현된 class로 Estimator class



사이킷런의 주요 모듈

분류	모듈명	설명
예제 데이터	sklearn.datasets	사이킷런에 내장되어 예제로 제공하는 데이터 세트
데이터 분리, 검증 & 파라미터 튜닝	sklearn.model_selection	교차 검증을 위한 학습용/테스트용 분리, 그리드 서치(Grid Search)로 최적 파라미터 추출 등의 API 제공
	sklearn.preprocessing	데이터 전처리에 필요한 다양한 가공 기능 제공(문자열을 숫자형 코드 값으로 인코딩, 정규화, 스케일링 등)
	sklearn.feature_selection	알고리즘에 큰 영향을 미치는 피처를 우선순위대로 셀렉션 작업을 수행하는 다양한 기능 제공
피처 처리		텍스트 데이터나 이미지 데이터의 벡터화된 피처를 추출하는 데 사용됨.
	sklearn.feature_extraction	예를 들어 텍스트 데이터에서 Count Vectorizer 나 Tf-Idf Vectorizer 등을 생성하는 기능 제공. 텍스트 데이터의 피처 추출은 sklearn.feature_extraction.text 모듈에, 이미지 데이터의 피처 추출은 sklearn.feature_extraction.image 모듈에 지원 API가 있음.
피처 처리 & 차원 축소	sklearn.decomposition	차원 축소와 관련한 알고리즘을 지원하는 모듈임. PCA, NMF, Truncated SVD 등을 통해 차원 축소 기능을 수행할 수 있음

분류	모듈명	설명
평가	sklearn.metrics	분류, 회귀, 클러스터링, 페어와이즈(Pairwise)에 대한 다양한 성능 측정 방법 제공 Accuracy, Precision, Recall, ROC-AUC, RMSE 등 제공
	sklearn.ensemble	앙상블 알고리즘 제공 랜덤 포레스트, 애이다 부스트, 그래디언트 부스팅 등을 제공
	sklearn.linear_model	주로 선형 회귀, 릿지(Ridge), 라쏘(Lasso) 및 로지스틱 회귀 등 회귀 관련 알고리즘을 지원. 또한 SGD(Stochastic Gradient Descent) 관련 알고리즘도 제공
ML 알고리즘	sklearn.naive_bayes	나이브 베이즈 알고리즘 제공. 가우시안 NB, 다행 분포 NB 등.
	sklearn.neighbors	최근접 이웃 알고리즘 제공. K-NN 등
	sklearn.svm	서포트 벡터 머신 알고리즘 제공
	sklearn.tree	의사 결정 트리 알고리즘 제공
	sklearn.cluster	비지도 클러스터링 알고리즘 제공 (K-평균, 계층형, DBSCAN 등)
유틸리티	sklearn.pipeline	피처 처리 등의 변환과 ML 알고리즘 학습, 예측 등을 함께 묶어서 실행할 수 있는 유틸리티 제공

내장된 예제 데이터 세트

분류나 회귀 연습용 예제 데이터

API 명	설명
datasets.load_boston()	회귀 용도이며, 미국 보스턴의 집 피처들과 가격에 대한 데이터 세트
datasets.load_breast_cancer()	분류 용도이며, 위스콘신 유방암 피처들과 악성/음성 레이블 데이터 세트
datasets.load_diabetes()	회귀 용도이며, 당뇨 데이터 세트
datasets.load_digits()	분류 용도이며, 0에서 9까지 숫자의 이미지 픽셀 데이터 세트
datasets.load_iris()	분류 용도이며, 붓꽃에 대한 피처를 가진 데이터 세트

Fetch_covtype, 29newsgroups~~~

분류와 클러스터링을 위한 표본 데이터 생성기

API 명	설명
datasets.make_classifications()	분류를 위한 데이터 세트를 만듭니다. 특히 높은 상관도, 불필요한 속성 등의 노이즈 효과를 위한 데이터를 무작위로 생성해 줍니다.
datasets.make_blobs()	클러스터링을 위한 데이터 세트를 무작위로 생성해 줍니다. 군집 지정 개수에 따라 여러 가지 클러스터링을 위한 데이터 세트를 쉽게 만들어 줍니다.

Data set의 Key의 의미

- Data : 피터의 데이터셋
- Target: 분류시 레이블값, 숫자 결과값
- Target_names : 개별 레이블 이름
- feature_names : 피처의 이름
- DESCR은 데이터 세트의 대한 설명과 피처의 설명

붓꽃 데이터셋 생성

```
from sklearn.datasets import load_iris  
  
iris_data = load_iris()  
print(type(iris_data))
```

```
<class 'sklearn.utils.Bunch'>
```

```
keys = iris_data.keys()  
print('붓꽃 데이터 세트의 키들:', keys)
```

```
붓꽃 데이터 세트의 키들: dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename'])
```

Load_iris()가 반환하는 객체의 값 출력

```
print('#n feature_names 의 type:',type(iris_data.feature_names))
print(' feature_names 의 shape:',len(iris_data.feature_names))
print(iris_data.feature_names)

print('#n target_names 의 type:',type(iris_data.target_names))
print(' target_names 의 shape:',len(iris_data.target_names))
print(iris_data.target_names)

print('#n data 의 type:',type(iris_data.data))
print(' data 의 shape:',iris_data.data.shape)
print(iris_data['data'])

print('#n target 의 type:',type(iris_data.target))
print(' target 의 shape:',iris_data.target.shape)
print(iris_data.target)
```

```
feature_names 의 type: <class 'list'>
feature_names 의 shape: 4
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

```
target_names 의 type: <class 'numpy.ndarray'>
feature_names 의 shape: 3
['setosa' 'versicolor' 'virginica']
```

Module Selection 소개-학습 데이터와 테스트 데이터

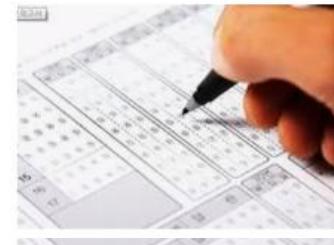
학습 데이터 세트

- 머신러닝 알고리즘의 학습을 위해 사용.
- 데이터의 속성들과 결정값(레이블)값 모두를 가지고 있음
- 학습 데이터를 기반으로 머신러닝 알고리즘이 데이터 속성과 결정값의 패턴을 인지하고 학습



테스트 데이터 세트

- 테스트 데이터 세트에서 학습된 머신러닝 알고리즘을 테스트
- 테스트 데이터는 속성 데이터만 머신러닝 알고리즘에 제공하며, 머신러닝 알고리즘은 제공된 데이터를 기반으로 결정값을 예측
- 테스트 데이터는 학습 데이터와 별도의 데이터 세트로 제공되어야 함.



학습 데이터와 테스트 데이터 분리 – train_test_split()

sklearn.model_selection의 train_test_split()함수

```
X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_data.target, test_size=0.3, random_state=121)
```

- **test_size**: 전체 데이터에서 테스트 데이터 세트 크기를 얼마로 샘플링할 것인가를 결정합니다. 디폴트는 0.25, 즉 25%입니다.
- **train_size**: 전체 데이터에서 학습용 데이터 세트 크기를 얼마로 샘플링할 것인가를 결정합니다. test_size parameter를 통상적으로 사용하기 때문에 train_size는 잘 사용되지 않습니다.
- **shuffle**: 데이터를 분리하기 전에 데이터를 미리 섞을지를 결정합니다. 디폴트는 True입니다. 데이터를 분산시켜서 좀 더 효율적인 학습 및 테스트 데이터 세트를 만드는 데 사용됩니다.
- **random_state**: random_state는 호출할 때마다 동일한 학습/테스트용 데이터 세트를 생성하기 위해 주어지는 난수 값입니다. train_test_split()는 호출 시 무작위로 데이터를 분리하므로 random_state를 지정하지 않으면 수행할 때마다 다른 학습/테스트 용 데이터를 생성합니다

Module Selection 소개

train_test_spilt() 분리?

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

iris = load_iris()
dt_clf = DecisionTreeClassifier()
train_data = iris.data
train_label = iris.target
dt_clf.fit(train_data, train_label)

# 학습 데이터 셋으로 예측 수행
pred = dt_clf.predict(train_data)
print('예측 정확도:',accuracy_score(train_label,pred))
```

예측 정확도: 1.0

Module Selection 소개

train_test_spilt() 분리? 30:70 분리 random_state=121 ?

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

dt_clf = DecisionTreeClassifier()
iris_data = load_iris()

X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_data.target,
                                                    test_size=0.3, random_state=121)
```

```
dt_clf.fit(X_train, y_train)
pred = dt_clf.predict(X_test)
print('예측 정확도: {:.4f}'.format(accuracy_score(y_test, pred)))
```

예측 정확도: 0.9556

교차 검증

학습 데이터를 다시 분할하여 학습 데이터와 학습된 모델의 성능을 일차 평가하는 검증 데이터로 나눔

모든 학습/검증 과정이 완료된 후 최종적으로 성능을 평가하기 위한 데이터 세트



- 일반적인 ML모델은 1차 평가후
최종 테스트 데이터 세트에 적용후 평가.

K 폴드 교차 검증

K=5일 경우

총 5개의
폴드 세트에
5번의 학습과 검증
평가 반복 수행



K 폴드 교차 검증

K 폴드 교차 검증

- 일반 K 폴드
- Stratified K 폴드
 - 불균형한(imbalanced) 분포도를 가진 레이블(결정 클래스) 데이터 집합을 위한 K 폴드 방식.
 - 학습데이터와 검증 데이터 세트가 가지는 레이블 분포도가 유사하도록 검증 데이터 추출

Kfold 예제

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import KFold
import numpy as np

iris = load_iris()
features = iris.data
label = iris.target
dt_clf = DecisionTreeClassifier(random_state=156)

# 5개의 폴드 세트로 분리하는 KFold 객체와 폴드 세트별 정확도를 담을 리스트 객체 생성.
kfold = KFold(n_splits=5)
cv_accuracy = []
print('붓꽃 데이터 세트 크기:', features.shape[0])
```

붓꽃 데이터 세트 크기: 150

Kfloid 예제-인덱스 추출

```
n_iter = 0
```

```
# KFold객체의 split( ) 호출하면 폴드 별 학습용, 검증용 테스트의 로우 인덱스를 array로 반환
for train_index, test_index in kfold.split(features):
    # kfold.split( )으로 반환된 인덱스를 이용하여 학습용, 검증용 테스트 데이터 추출
    X_train, X_test = features[train_index], features[test_index]
    y_train, y_test = label[train_index], label[test_index]
    #학습 및 예측
    dt_clf.fit(X_train, y_train)
    pred = dt_clf.predict(X_test)
    n_iter += 1
    # 반복 시마다 정확도 측정
    accuracy = np.round(accuracy_score(y_test, pred), 4)
    train_size = X_train.shape[0]
    test_size = X_test.shape[0]
    print('#n{0} 교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'
          .format(n_iter, accuracy, train_size, test_size))
    print('#{0} 검증 세트 인덱스:{1}'.format(n_iter, test_index))
    cv_accuracy.append(accuracy)

# 개별 iteration별 정확도를 합하여 평균 정확도 계산
print('##n## 평균 검증 정확도:', np.mean(cv_accuracy))
```

```
#1 교차 검증 정확도 :1.0, 학습 데이터 크기: 120, 검증 데이터 크기: 30
#1 검증 세트 인덱스:[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
  24 25 26 27 28 29]
```

```
#2 교차 검증 정확도 :0.9667, 학습 데이터 크기: 120, 검증 데이터 크기: 30
#2 검증 세트 인덱스:[30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
  54 55 56 57 58 59]
```

```
#3 교차 검증 정확도 :0.8667, 학습 데이터 크기: 120, 검증 데이터 크기: 30
#3 검증 세트 인덱스:[60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
  84 85 86 87 88 89]
```

```
#4 교차 검증 정확도 :0.9333, 학습 데이터 크기: 120, 검증 데이터 크기: 30
#4 검증 세트 인덱스:[ 90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
  108 109 110 111 112 113 114 115 116 117 118 119]
```

```
#5 교차 검증 정확도 :0.7333, 학습 데이터 크기: 120, 검증 데이터 크기: 30
#5 검증 세트 인덱스:[120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137
  138 139 140 141 142 143 144 145 146 147 148 149]
```

```
## 평균 검증 정확도: 0.9
```

Stratified K fold

```
import pandas as pd

iris = load_iris()

iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['label'] = iris.target
iris_df['label'].value_counts()
```

```
2    50
1    50
0    50
Name: label, dtype: int64
```

```
kfold = KFold(n_splits=3)
# kfold.split(X)는 폴드 세트를 3번 반복할 때마다 달라지는 학습/테스트 용 데이터로의 인덱스 번호 반환.
n_iter = 0
for train_index, test_index in kfold.split(iris_df):
    n_iter += 1
    label_train = iris_df['label'].iloc[train_index]
    label_test = iris_df['label'].iloc[test_index]
    print('## 교차 검증: {}'.format(n_iter))
    print('학습 레이블 데이터 분포:', label_train.value_counts())
    print('검증 레이블 데이터 분포:', label_test.value_counts())
```

```
## 교차 검증: 1
학습 레이블 데이터 분포:
2    50
1    50
Name: label, dtype: int64
검증 레이블 데이터 분포:
0    50
Name: label, dtype: int64
## 교차 검증: 2
학습 레이블 데이터 분포:
2    50
0    50
Name: label, dtype: int64
검증 레이블 데이터 분포:
1    50
Name: label, dtype: int64
## 교차 검증: 3
학습 레이블 데이터 분포:
1    50
0    50
Name: label, dtype: int64
검증 레이블 데이터 분포:
2    50
Name: label, dtype: int64
```

Stratified K fold

```
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=3)
n_iter=0

for train_index, test_index in skf.split(iris_df, iris_df['label']):
    n_iter += 1
    label_train=iris_df['label'].iloc[train_index]
    label_test=iris_df['label'].iloc[test_index]
    print('## 교차 검증: {}'.format(n_iter))
    print('학습 레이블 데이터 분포:', label_train.value_counts())
    print('검증 레이블 데이터 분포:', label_test.value_counts())
```

```
## 교차 검증: 1
학습 레이블 데이터 분포:
2    33
1    33
0    33
Name: label, dtype: int64
검증 레이블 데이터 분포:
2    17
1    17
0    17
Name: label, dtype: int64
## 교차 검증: 2
학습 레이블 데이터 분포:
2    33
1    33
0    33
Name: label, dtype: int64
검증 레이블 데이터 분포:
2    17
1    17
0    17
Name: label, dtype: int64
## 교차 검증: 3
학습 레이블 데이터 분포:
2    34
1    34
0    34
Name: label, dtype: int64
검증 레이블 데이터 분포:
2    16
1    16
0    16
Name: label, dtype: int64
```

Stratified K fold

```
dt_clf = DecisionTreeClassifier(random_state=156)

skfold = StratifiedKFold(n_splits=3)
n_iter=0
cv_accuracy=[]

# StratifiedKFold의 split() 호출시 반드시 레이블 데이터 셋도 추가 입력 필요
for train_index, test_index in skfold.split(features, label):
    # split()으로 반환된 인덱스를 이용하여 학습용, 검증용 테스트 데이터 추출
    X_train, X_test = features[train_index], features[test_index]
    y_train, y_test = label[train_index], label[test_index]
    #학습 및 예측
    dt_clf.fit(X_train, y_train)
    pred = dt_clf.predict(X_test)

    # 반복 시마다 정확도 측정
    n_iter += 1
    accuracy = np.round(accuracy_score(y_test, pred), 4)
    train_size = X_train.shape[0]
    test_size = X_test.shape[0]
    print('#{0} 교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'
          .format(n_iter, accuracy, train_size, test_size))
    print('#{0} 검증 세트 인덱스:{1}'.format(n_iter, test_index))
    cv_accuracy.append(accuracy)

# 교차 검증별 정확도 및 평균 정확도 계산
print('## 교차 검증별 정확도:', np.round(cv_accuracy, 4))
print('## 평균 검증 정확도:', np.mean(cv_accuracy))
```

```
#1 교차 검증 정확도 :0.9804, 학습 데이터 크기: 99, 검증 데이터 크기: 51
#1 검증 세트 인덱스:[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 50
  51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 100 101
  102 103 104 105 106 107 108 109 110 111 112 113 114 115 116]
#2 교차 검증 정확도 :0.9216, 학습 데이터 크기: 99, 검증 데이터 크기: 51
#2 검증 세트 인덱스:[ 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 67
  68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 117 118
  119 120 121 122 123 124 125 126 127 128 129 130 131 132 133]
#3 교차 검증 정확도 :0.9792, 학습 데이터 크기: 102, 검증 데이터 크기: 48
#3 검증 세트 인덱스:[ 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 84 85
  86 87 88 89 90 91 92 93 94 95 96 97 98 99 134 135 136 137
  138 139 140 141 142 143 144 145 146 147 148 149]
## 교차 검증별 정확도: [0.9804 0.9216 0.9792]
## 평균 검증 정확도: 0.9604
```

교차 검증을 보다 간편하게 `-cross_val_score()`

KFold 클래스를 이용한 교차 검증 방법

1 폴드 세트 설정

2 For 루프에서 반복적으로 학습/검증 데이터 추출 및 학습과 예측 수행

3 폴드 세트별로 예측 성능을 평균하여 최종 성능 평가



`cross_val_score()` 함수로
폴드 세트 추출, 학습/예측, 평가를
한번에 수행

```
cross_val_score(estimator, X, y=None, scoring=None, cv=None,  
n_jobs=1, verbose=0, fit_params=None, pre_dispatch='2*n_jobs')
```

GridSearchCV – 교차 검증과 최적 하이퍼 파라미터 튜닝을 한번에



사이킷런은 GridSearchCV 를 이용해 Classifier나 Regressor와 같은 알고리즘에 사용되는 하이퍼 파라미터를 순차적으로 입력하면서 편리하게 최적의 파라미터를 도출할 수 있는 방안을 제공합니다

```
grid_parameters = {'max_depth': [1, 2, 3],  
'min_samples_split': [2, 3]}  
}
```

cv 세트가 3 이라면

파라미터 순차 적용 횟수	cv 세트수	학습/검증 총 수행횟수
6	X	3
		18

순번	max_depth	min_samples_split
1	1	2
2	1	3
3	2	2
4	2	3
5	3	2
6	3	3

cross_val_score()

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score, cross_validate
from sklearn.datasets import load_iris

iris_data = load_iris()
dt_clf = DecisionTreeClassifier(random_state=156)

data = iris_data.data
label = iris_data.target

# 성능 지표는 정확도(accuracy), 교차 검증 세트는 3개
scores = cross_val_score(dt_clf, data, label, scoring='accuracy', cv=3)
print('교차 검증별 정확도:', np.round(scores, 4))
print('평균 검증 정확도:', np.round(np.mean(scores), 4))
```

교차 검증별 정확도: [0.9804 0.9216 0.9792]

평균 검증 정확도: 0.9604

GridSearchCV

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

# 데이터를 로딩하고 학습데이터와 테스트 데이터 분리
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.data.target,
                                                    test_size=0.2, random_state=121)
dtree = DecisionTreeClassifier()

### parameter 들을 dictionary 형태로 설정
parameters = {'max_depth':[1,2,3], 'min_samples_split':[2,3]}

import pandas as pd

# param_grid의 하이퍼 파라미터들을 3개의 train, test set fold로 나누어서 테스트 수행 설정.
### refit=True 가 default 일. True이면 가장 좋은 파라미터 설정으로 재 학습 시킴.
grid_dtree = GridSearchCV(dtree, param_grid=parameters, cv=3, refit=True)

# 끝꽃 Train 데이터로 param_grid의 하이퍼 파라미터들을 순차적으로 학습/평가 .
grid_dtree.fit(X_train, y_train)

# GridSearchCV 결과 추출하여 DataFrame으로 변환
scores_df = pd.DataFrame(grid_dtree.cv_results_)
scores_df[['params', 'mean_test_score', 'rank_test_score', '#',
           'split0_test_score', 'split1_test_score', 'split2_test_score']]
```

List형태 입력 (param)
refit: 최적의 parameter시 학습
grid_dtree.cv_results_ : 확인 (많은 parameters)

	params	mean_test_score	rank_test_score	split0_test_score	split1_test_score	split2_test_score
0	{'max_depth': 1, 'min_samples_split': 2}	0.700000	5	0.700	0.7	0.70
1	{'max_depth': 1, 'min_samples_split': 3}	0.700000	5	0.700	0.7	0.70
2	{'max_depth': 2, 'min_samples_split': 2}	0.958333	3	0.925	1.0	0.95
3	{'max_depth': 2, 'min_samples_split': 3}	0.958333	3	0.925	1.0	0.95
4	{'max_depth': 3, 'min_samples_split': 2}	0.966667	1	0.950	1.0	0.95
5	{'max_depth': 3, 'min_samples_split': 3}	0.966667	1	0.950	1.0	0.95

GridSearchCV

```
print('GridSearchCV 최적 파라미터:', grid_dtrees.best_params_)
print('GridSearchCV 최고 정확도: {:.4f}'.format(grid_dtrees.best_score_))
```

```
GridSearchCV 최적 파라미터: {'max_depth': 3, 'min_samples_split': 2}
GridSearchCV 최고 정확도: 0.9750
```

```
# GridSearchCV의 refit으로 이미 학습이 된 estimator 반환
estimator = grid_dtrees.best_estimator_

# GridSearchCV의 best_estimator_는 이미 최적 하이퍼 파라미터로 학습이 된
pred = estimator.predict(X_test)
print('테스트 데이터 세트 정확도: {:.4f}'.format(accuracy_score(y_test, pred)))
```

```
테스트 데이터 세트 정확도: 0.9667
```

cross_val_score()

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score, cross_validate
from sklearn.datasets import load_iris

iris_data = load_iris()
dt_clf = DecisionTreeClassifier(random_state=156)

data = iris_data.data
label = iris_data.target

# 성능 지표는 정확도(accuracy), 교차 검증 세트는 3개
scores = cross_val_score(dt_clf, data, label, scoring='accuracy', cv=3)
print('교차 검증별 정확도:', np.round(scores, 4))
print('평균 검증 정확도:', np.round(np.mean(scores), 4))
```

교차 검증별 정확도: [0.9804 0.9216 0.9792]

평균 검증 정확도: 0.9604

데이터 전처리

- 데이터 클린징
- 결손값 처리(Null/NaN 처리)
- 데이터 인코딩(레이블, 원-핫 인코딩)
- 데이터 스케일링
- 이상치 제거
- Feature 선택, 추출 및 가공

데이터 전처리

- 데이터 전처리는 알고리즘 만큼 중요. (Garbage in, Garbage out)
- 데이터는 문자열을 입력 값으로 허용하지 않는다.
- 문자열을 인코딩하여 숫자로 변화 feature vectorization 기법

데이터 인코딩

- 레이블 인코딩 (Label encoding) 과 원 핫 인코딩(One Hot encoding)

Label encoding

원본 데이터

상품 분류	가격
TV	1,000,000
냉장고	1,500,000
전자렌지	200,000
컴퓨터	800,000
선풍기	100,000
선풍기	100,000
믹서	50,000
믹서	50,000

상품 분류를 레이블 인코딩한 데이터

상품 분류	가격
0	1,000,000
1	1,500,000
4	200,000
5	800,000
3	100,000
3	100,000
2	50,000
2	50,000

```
from sklearn.preprocessing import LabelEncoder
```

```
items=['TV', '냉장고', '전자렌지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']
```

LabelEncoder를 객체로 생성한 후, fit()과 transform()으로 label 인코딩 수행.

```
encoder = LabelEncoder()
```

```
encoder.fit(items)
```

```
labels = encoder.transform(items)
```

```
print('인코딩 변환값:', labels)
```

인코딩 변환값: [0 1 4 5 3 3 2 2]

```
print('인코딩 클래스:', encoder.classes_)
```

인코딩 클래스: ['TV' '냉장고' '믹서' '선풍기' '전자렌지' '컴퓨터']

```
print('디코딩 원본 값:', encoder.inverse_transform([4, 5, 2, 0, 1, 1, 3, 3]))
```

디코딩 원본 값: ['전자렌지' '컴퓨터' '믹서' 'TV' '냉장고' '냉장고' '선풍기' '선풍기']

원-핫 인코딩

피처값에 새로운 피처를 추가후 고유값에 해당되는 칼럼에 1표시 하고 나머지는 0 표시 하는 방식.

원본 데이터

상품 분류
TV
냉장고
전자렌지
컴퓨터
선풍기
선풍기
믹서
믹서

원-핫 인코딩

상품분류_	상품분류_	상품분류_	상품분류_	상품분류_	상품분류_
TV	냉장고	믹서	선풍기	전자렌지	컴퓨터
1	0	0	0	0	0
0	1	0	0	0	0
0	0	0	0	1	0
0	0	0	0	0	1
0	0	0	1	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	0	0	0



One Hot encoding

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np

items=['TV', '냉장고', '전자렌지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']

# 먼저 숫자값으로 변환을 위해 LabelEncoder로 변환합니다.
encoder = LabelEncoder()
encoder.fit(items)
labels = encoder.transform(items)

# 2차원 데이터로 변환합니다.
labels = labels.reshape(-1,1)

# 원-핫 인코딩을 적용합니다.
oh_encoder = OneHotEncoder()
oh_encoder.fit(labels)
oh_labels = oh_encoder.transform(labels)
print('원-핫 인코딩 데이터')
print(oh_labels.toarray())
print('원-핫 인코딩 데이터 차원')
print(oh_labels.shape)
```

원-핫 인코딩 데이터

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

원-핫 인코딩 데이터 차원
(8, 6)

원본 데이터

상품 분류	가격
TV	1,000,000
냉장고	1,500,000
전자렌지	200,000
컴퓨터	800,000
선풍기	100,000
선풍기	100,000
믹서	50,000
믹서	50,000

숫자로 인코딩

상품 분류	가격
0	1,000,000
1	1,500,000
4	200,000
5	800,000
3	100,000
3	100,000
2	50,000
2	50,000

원-핫 인코딩

TV	냉장고	믹서	선풍기	전자렌지	컴퓨터	가격
1	0	0	0	0	0	1,000,000
0	1	0	0	0	0	1,500,000
0	0	0	0	1	0	200,000
0	0	0	0	0	1	800,000
0	0	0	1	0	0	100,000
0	0	0	1	0	0	100,000
0	0	1	0	0	0	50,000
0	0	1	0	0	0	50,000

판다스에서 지원하는 원-핫 인코딩 API get_dummies()

```
import pandas as pd

df = pd.DataFrame({'item':['TV','냉장고','전자렌지','컴퓨터','선풍기','선풍기','믹서','믹서']})
pd.get_dummies(df)
```

	item_TV	item_냉장고	item_믹서	item_선풍기	item_전자렌지	item_컴퓨터
0	1	0	0	0	0	0
1	0	1	0	0	0	0
2	0	0	0	0	1	0
3	0	0	0	0	0	1
4	0	0	0	1	0	0
5	0	0	0	1	0	0
6	0	0	1	0	0	0
7	0	0	1	0	0	0

피쳐 스케일링과 정규화

서로 다른 변수의 값 범위를 일정한 수준으로 맞추는 작업 : feature scaling

-> 표준화 (Standardization)와 정규화 (Normalization)

표준화 : 평균이 0이고 분산이 1인 Gaussian distribution으로 변환

$$x_i_new = \frac{x_i - \text{mean}(x)}{\text{stdev}(x)}$$

정규화 : 서로 다른 피처의 크기를 통일하기 위해 크기를 변환하는 개념.

$$x_i_new = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

사이킷 런의 Normalizer모듈은 선형대수에서의 정규화 개념이 적용, 개별 벡터의 크기를 맞추기 위해 변환

$$x_i_new = \frac{x_i}{\sqrt{x_i^2 + y_i^2 + z_i^2}}$$

사이킷런 피쳐 스케일링 지원

- StandardScaler: 평균이 0이고, 분산이 1인 정규 분포 형태로 변환
- MinMaxScaler: 데이터값을 0과 1사이의 범위 값으로 변환합니다 (음수 값이 있으면 -1에서 1값으로 변환합니다)

StandardScaler

```
from sklearn.datasets import load_iris
import pandas as pd
# 붓꽃 데이터 셋을 로딩하고 DataFrame으로 변환합니다.
iris = load_iris()
iris_data = iris.data
iris_df = pd.DataFrame(data=iris_data, columns=iris.feature_names)

print('feature들의 평균 값')
print(iris_df.mean())
print('feature들의 분산 값')
print(iris_df.var())
```

feature들의 평균 값

sepal length (cm)	5.843333
sepal width (cm)	3.054000
petal length (cm)	3.758667
petal width (cm)	1.198667

dtype: float64

feature들의 분산 값

sepal length (cm)	0.685694
sepal width (cm)	0.188004
petal length (cm)	3.113179
petal width (cm)	0.582414

dtype: float64

표준화후 평균값과 분산 reconfirm!

```
from sklearn.preprocessing import StandardScaler

# StandardScaler 객체 생성
scaler = StandardScaler()
# StandardScaler로 데이터셋 변환. fit()과 transform() 호출.
scaler.fit(iris_df)
iris_scaled = scaler.transform(iris_df)

# transform()과 scale 변환된 데이터셋이 numpy ndarray로 반환되어 이를 DataFrame으로 변환
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
print('feature들의 평균 값')
print(iris_df_scaled.mean())
print('feature들의 분산 값')
print(iris_df_scaled.var())
```

```
feature들의 평균 값
sepal length (cm)    -1.690315e-15
sepal width (cm)     -1.637024e-15
petal length (cm)    -1.482518e-15
petal width (cm)     -1.623146e-15
dtype: float64
```

```
feature들의 분산 값
sepal length (cm)    1.006711
sepal width (cm)     1.006711
petal length (cm)    1.006711
petal width (cm)     1.006711
dtype: float64
```

MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler

# MinMaxScaler 객체 생성
scaler = MinMaxScaler()
# MinMaxScaler로 데이터셋 변환. fit() 과 transform() 호출.
scaler.fit(iris_df)
iris_scaled = scaler.transform(iris_df)

# transform() 시 scale 변환된 데이터셋이 numpy ndarray로 반환되어 이를 DataFrame으로 변환
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
print('feature들의 최소 값')
print(iris_df_scaled.min())
print('feature들의 최대 값')
print(iris_df_scaled.max())
```

feature들의 최소 값

```
sepal length (cm)    0.0
sepal width (cm)    0.0
petal length (cm)   0.0
petal width (cm)    0.0
dtype: float64
```

feature들의 최대 값

```
sepal length (cm)    1.0
sepal width (cm)    1.0
petal length (cm)   1.0
petal width (cm)    1.0
dtype: float64
```

타이타닉 생존자 ML예측 구현

데이터 전처리

- Null 처리
- 불필요한 속성 제거
- 인코딩 수행

모델 학습 및 검증/예측/평가

- 결정트리, 랜덤포레스트,
로지스틱 회귀 학습 비교
- K 폴드 교차 검증
- cross_val_score()와
GridSearchCV() 수행



타이타닉 생존자 ML예측 구현

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

titanic_df = pd.read_csv('./titanic_train.csv')
titanic_df.head(3)
```

PassengerId	Survived	Pclass		Name	Sex	Age	SibSp	Parch		Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S	
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th... Th...	female	38.0	1	0	PC 17599	71.2833	C85	C	
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S	

- Passengerid: 탑승자 데이터 일련번호
- survived: 생존 여부, 0 = 사망, 1 = 생존
- Pclass: 티켓의 선실 등급, 1 = 일등석, 2 = 이등석, 3 = 삼등석
- sex: 탑승자 성별
- name: 탑승자 이름
- Age: 탑승자 나이
- sibsp: 같이 탑승한 형제자매 또는 배우자 인원수
- parch: 같이 탑승한 부모님 또는 어린이 인원수
- ticket: 티켓 번호
- fare: 요금
- cabin: 선실 번호
- embarked: 중간 정착 항구 C = Cherbourg, Q = Queenstown, S = Southampton

타이타닉 생존자 ML예측 구현

```
print('## train 데이터 정보 ##\n')
print(titanic_df.info())
```

train 데이터 정보

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype  
 ---  -- 
 0   PassengerId 891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object  
 4   Sex          891 non-null    object  
 5   Age          714 non-null    float64 
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object  
 9   Fare          891 non-null    float64 
 10  Cabin         204 non-null    object  
 11  Embarked     889 non-null    object  
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
None
```

```
titanic_df['Age'].fillna(titanic_df['Age'].mean(), inplace=True)
titanic_df['Cabin'].fillna('N', inplace=True)
titanic_df['Embarked'].fillna('N', inplace=True)
print('데이터 세트 Null 값 갯수 ',titanic_df.isnull().sum().sum())
```

데이터 세트 Null 값 갯수 0

```
print('데이터 세트 Null 값 갯수 ',titanic_df.isnull().sum())
```

```
데이터 세트 Null 값 갯수  PassengerId  0
Survived      0
Pclass        0
Name          0
Sex           0
Age           0
SibSp         0
Parch         0
Ticket        0
Fare          0
Cabin         0
Embarked      0
dtype: int64
```

타이타닉 생존자 ML예측 구현

```
print('Sex 값 분포 :',titanic_df['Sex'].value_counts())
print('\nCabin 값 분포 :',titanic_df['Cabin'].value_counts())
print('\nEmbarked 값 분포 :',titanic_df['Embarked'].value_counts())
```

Sex 값 분포 :

male 577
female 314
Name: Sex, dtype: int64

Cabin 값 분포 :

N	687
G6	4
C23 C25 C27	4
B96 B98	4
C22 C26	3
F2	3
E101	3
F33	3
D	3
C65	2
B22	2
B28	2
E25	2
D36	2
F G73	2
E24	2
E67	2
B51 B53 B55	2
C78	2
B35	2
F4	2
C125	2
E121	2
D20	2
C98	2
E38	2
B18	2
C52	2
D17	2
E8	2

D56 1
B82 B84 1
B80 1
A5 1
F E69 1
E38 1
E12 1
C50 1
D30 1
D49 1
C54 1
F G63 1
D46 1
A32 1
F38 1
C110 1
E31 1
E34 1
C87 1
D37 1
B86 1
C111 1
C47 1
D45 1
T 1
C101 1
D28 1
B73 1
B102 1
A24 1
Name: Cabin, Length: 148, dtype: int64

Embarked 값 분포 :

S	644
C	168
Q	77
N	2

Name: Embarked, dtype: int64

타이타닉 생존자 ML예측 구현

```
titanic_df['Cabin'] = titanic_df['Cabin'].str[:1]
print(titanic_df['Cabin'].head(3))
```

```
0    N
1    C
2    N
Name: Cabin, dtype: object
```

```
titanic_df['Cabin'].value_counts()
```

```
N    687
C     59
B     47
D     33
E     32
A     15
F     13
G      4
T      1
Name: Cabin, dtype: int64
```

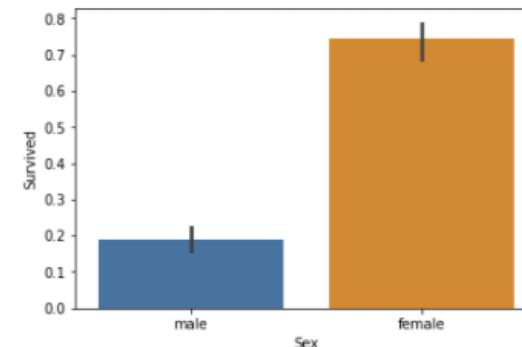
```
titanic_df.groupby(['Sex', 'Survived'])['Survived'].count()
```

Sex	Survived	Count
female	0	81
female	1	233
male	0	468
male	1	109

Name: Survived, dtype: int64

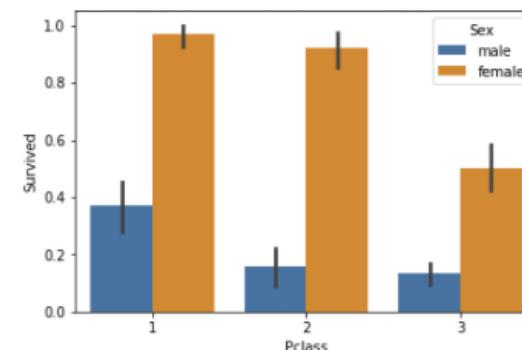
```
sns.barplot(x='Sex', y = 'Survived', data=titanic_df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1be04d1cbe0>
```



```
sns.barplot(x='Pclass', y='Survived', hue='Sex', data=titanic_df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1be04e50d30>
```



타이타닉 생존자 ML예측 구현

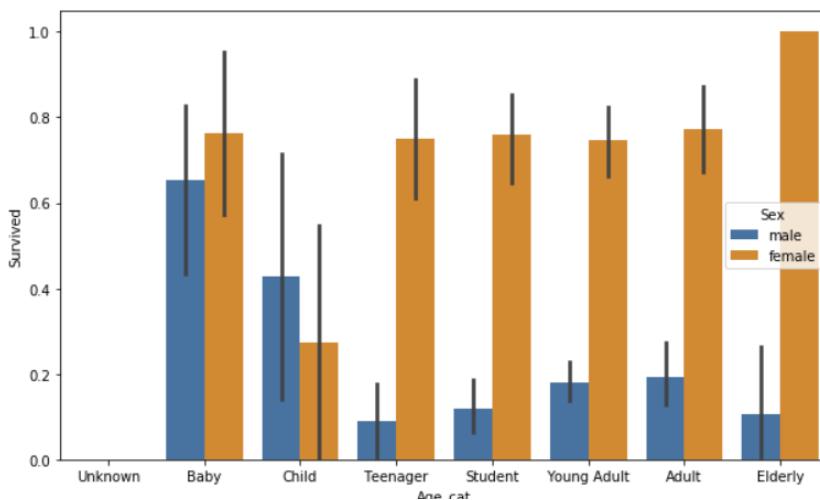
```
# 입력 age에 따라 구분값을 반환하는 함수 설정. DataFrame의 apply lambda식에 사용.
def get_category(age):
    cat = ''
    if age <= -1: cat = 'Unknown'
    elif age <= 5: cat = 'Baby'
    elif age <= 12: cat = 'Child'
    elif age <= 18: cat = 'Teenager'
    elif age <= 25: cat = 'Student'
    elif age <= 35: cat = 'Young Adult'
    elif age <= 60: cat = 'Adult'
    else : cat = 'Elderly'

    return cat

# 막대그래프의 크기 figure를 더 크게 설정
plt.figure(figsize=(10,6))

#X축의 값을 순차적으로 표시하기 위한 설정
group_names = ['Unknown', 'Baby', 'Child', 'Teenager', 'Student', 'Young Adult', 'Adult', 'Elderly']

# lambda식에 위에서 생성한 get_category( ) 함수를 반환값으로 지정
# get_category(X)는 입력값으로 'Age' 컬럼값을 받아서 해당하는 cat 반환
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x: get_category(x))
sns.barplot(x='Age_cat', y = 'Survived', hue='Sex', data=titanic_df, order=group_names)
titanic_df.drop('Age_cat', axis=1, inplace=True)
```



타이타닉 생존자 ML예측 구현

```
from sklearn import preprocessing

def encode_features(dataDF):
    features = ['Cabin', 'Sex', 'Embarked']
    for feature in features:
        le = preprocessing.LabelEncoder()
        le = le.fit(dataDF[feature])
        dataDF[feature] = le.transform(dataDF[feature])

    return dataDF

titanic_df = encode_features(titanic_df)
titanic_df.head()
```

	PassengerId	Survived	Pclass		Name	Sex	Age	SibSp	Parch		Ticket	Fare	Cabin	Embarked
0	1	0	3		Braund, Mr. Owen Harris	1	22.0	1	0		A/5 21171	7.2500	7	3
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th... Th...	0	38.0	1	0		PC 17599	71.2833	2	0	
2	3	1	3		Heikkinen, Miss. Laina	0	26.0	0	0	STON/O2. 3101282	7.9250	7	3	
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	0	35.0	1	0		113803	53.1000	2	3	
4	5	0	3		Allen, Mr. William Henry	1	35.0	0	0		373450	8.0500	7	3

타이타닉 생존자 ML예측 구현

```
from sklearn.preprocessing import LabelEncoder

# Null 처리 함수
def fillna(df):
    df['Age'].fillna(df['Age'].mean(), inplace=True)
    df['Cabin'].fillna('N', inplace=True)
    df['Embarked'].fillna('N', inplace=True)
    df['Fare'].fillna(0, inplace=True)
    return df

# 머신러닝 알고리즘에 불필요한 속성 제거
def drop_features(df):
    df.drop(['PassengerId', 'Name', 'Ticket'], axis=1, inplace=True)
    return df

# 레이블 인코딩 수행.
def format_features(df):
    df['Cabin'] = df['Cabin'].str[:1]
    features = ['Cabin', 'Sex', 'Embarked']
    for feature in features:
        le = LabelEncoder()
        le = le.fit(df[feature])
        df[feature] = le.transform(df[feature])
    return df

# 앞에서 설정한 Data Preprocessing 함수 호출
def transform_features(df):
    df = fillna(df)
    df = drop_features(df)
    df = format_features(df)
    return df
```

타이타닉 생존자 ML예측 구현

원본 데이터를 재로딩 하고, feature데이터 셋과 Label 데이터 셋 추출.

```
titanic_df = pd.read_csv('./titanic_train.csv')
y_titanic_df = titanic_df['Survived']
X_titanic_df = titanic_df.drop('Survived', axis=1)

X_titanic_df = transform_features(X_titanic_df)
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df, test_size=0.2, random_state=42)
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

```
# 결정트리, Random Forest, 로지스틱 회귀를 위한 사이킷런 Classifier
dt_clf = DecisionTreeClassifier(random_state=42)
rf_clf = RandomForestClassifier(random_state=42)
lr_clf = LogisticRegression()
```

```
# DecisionTreeClassifier 학습/예측/평가
dt_clf.fit(X_train, y_train)
dt_pred = dt_clf.predict(X_test)
print('DecisionTreeClassifier 정확도: {:.4f}'.format(accuracy_score(y_test, dt_pred)))
```

```
# RandomForestClassifier 학습/예측/평가
rf_clf.fit(X_train, y_train)
rf_pred = rf_clf.predict(X_test)
print('RandomForestClassifier 정확도: {:.4f}'.format(accuracy_score(y_test, rf_pred)))
```

```
# LogisticRegression 학습/예측/평가
lr_clf.fit(X_train, y_train)
lr_pred = lr_clf.predict(X_test)
print('LogisticRegression 정확도: {:.4f}'.format(accuracy_score(y_test, lr_pred)))
```

```
DecisionTreeClassifier 정확도: 0.7877
RandomForestClassifier 정확도: 0.8324
LogisticRegression 정확도: 0.8659
```

```
from sklearn.model_selection import KFold

def exec_kfold(clf, folds=5):
    # 폴드 세트를 5개의 KFold객체를 생성, 폴드 수만큼 예측결과 저장을 위한 리스트 객체 생성
    kfold = KFold(n_splits=folds)
    scores = []

    # KFold 교차 검증 수행.
    for iter_count, (train_index, test_index) in enumerate(kfold.split(X_titanic_df)):
        # X_titanic_df 데이터에서 교차 검증별로 학습과 검증 데이터를 가리키는 index 생성
        X_train, X_test = X_titanic_df.values[train_index], X_titanic_df.values[test_index]
        y_train, y_test = y_titanic_df.values[train_index], y_titanic_df.values[test_index]

        # Classifier 학습, 예측, 정확도 계산
        clf.fit(X_train, y_train)
        predictions = clf.predict(X_test)
        accuracy = accuracy_score(y_test, predictions)
        scores.append(accuracy)
        print("교차 검증 {}: 정확도: {:.4f}".format(iter_count, accuracy))

    # 5fold에서의 평균 정확도 계산.
    mean_score = np.mean(scores)
    print("평균 정확도: {:.4f}".format(mean_score))

# exec_kfold 호출
exec_kfold(dt_clf, folds=5)
```

```
교차 검증 0 정확도: 0.7542
교차 검증 1 정확도: 0.7809
교차 검증 2 정확도: 0.7865
교차 검증 3 정확도: 0.7697
교차 검증 4 정확도: 0.8202
평균 정확도: 0.7823
```

타이타닉 생존자 ML예측 구현

```
from sklearn.model_selection import KFold

def exec_kfold(clf, folds=5):
    # 폴드 세트를 5개인 KFold 객체를 생성, 폴드 수만큼 예측 결과 저장을 위한 리스트 객체 생성.
    kfold = KFold(n_splits=folds)
    scores = []

    # KFold 교차 검증 수행.
    for iter_count, (train_index, test_index) in enumerate(kfold.split(X_titanic_df)):
        # X_titanic_df 데이터에서 교차 검증별로 학습과 검증 데이터를 가리키는 index 생성
        X_train, X_test = X_titanic_df.values[train_index], X_titanic_df.values[test_index]
        y_train, y_test = y_titanic_df.values[train_index], y_titanic_df.values[test_index]

        # Classifier 학습, 예측, 정확도 계산
        clf.fit(X_train, y_train)
        predictions = clf.predict(X_test)
        accuracy = accuracy_score(y_test, predictions)
        scores.append(accuracy)
        print("교차 검증 {0} 정확도: {1:.4f}".format(iter_count, accuracy))

    # 5개 fold에서의 평균 정확도 계산.
    mean_score = np.mean(scores)
    print("평균 정확도: {0:.4f}".format(mean_score))
# exec_kfold 호출
exec_kfold(dt_clf, folds=5)
```

교차 검증 0 정확도: 0.7542
교차 검증 1 정확도: 0.7809
교차 검증 2 정확도: 0.7865
교차 검증 3 정확도: 0.7697
교차 검증 4 정확도: 0.8202
평균 정확도: 0.7823

타이타닉 생존자 ML예측 구현

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(dt_clf, X_titanic_df, y_titanic_df, cv=5)
for iter_count, accuracy in enumerate(scores):
    print("교차 검증 {} 정확도: {:.4f}".format(iter_count, accuracy))

print("평균 정확도: {:.4f}".format(np.mean(scores)))
```

```
교차 검증 0 정확도: 0.7430
교차 검증 1 정확도: 0.7753
교차 검증 2 정확도: 0.7921
교차 검증 3 정확도: 0.7865
교차 검증 4 정확도: 0.8427
평균 정확도: 0.7879
```

```
from sklearn.model_selection import GridSearchCV

parameters = {'max_depth':[2,3,5,10],
              'min_samples_split':[2,3,5], 'min_samples_leaf':[1,5,8]}

grid_dclf = GridSearchCV(dt_clf, param_grid=parameters, scoring='accuracy', cv=5)
grid_dclf.fit(X_train, y_train)

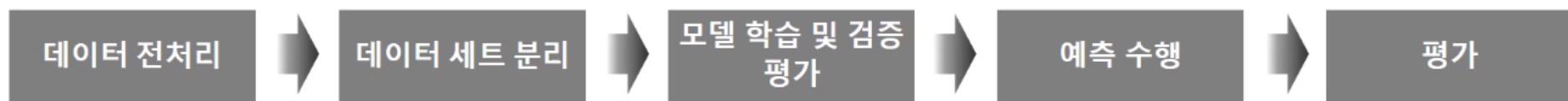
print('GridSearchCV 최적 하이퍼 파라미터 :',grid_dclf.best_params_)
print('GridSearchCV 최고 정확도: {:.4f}'.format(grid_dclf.best_score_))
best_dclf = grid_dclf.best_estimator_

# GridSearchCV의 최적 하이퍼 파라미터로 학습된 Estimator로 예측 및 평가 수행.
dpredictions = best_dclf.predict(X_test)
accuracy = accuracy_score(y_test, dpredictions)
print('테스트 세트에서의 DecisionTreeClassifier 정확도 : {:.4f}'.format(accuracy))
```

```
GridSearchCV 최적 하이퍼 파라미터 : {'max_depth': 3, 'min_samples_leaf': 1, 'min_samples_split': 2}
GridSearchCV 최고 정확도: 0.7992
테스트 세트에서의 DecisionTreeClassifier 정확도 : 0.8715
```

Summary

머신 러닝 지도 학습 프로세스



- 데이터 클린징
- 결손값 처리(Null/Nan 처리)
- 데이터 인코딩(레이블, 원-핫 인코딩)
- 데이터 스케일링
- 이상치 제거
- Feature 선택, 추출 및 가공
- 학습 데이터/테스트 데이터 분리
- 알고리즘 학습
- 교차 검증
- `cross_val_score()`
- `GridSearchCV`
- 테스트 데이터로 예측 수행
- 예측 평가

Chapter 03

머신러닝 평가

분류 (Classification) 성능 평가 지 표

- 정확도(Accuracy)
- 오차행렬(Confusion Matrix)
- 정밀도(Precision)
- 재현율(Recall)
- F1 스코어
- ROC AUC



정확도 (Accuracy)

$$\text{정확도(Accuracy)} = \frac{\text{예측 결과가 동일한 데이터 건수}}{\text{전체 예측 데이터 건수}}$$

- 정확도는 직관적으로 모델 예측 성능을 나타내는 평가 지표입니다. 하지만 이진 분류의 경우 데이터의 구성에 따라 ML 모델의 성능을 왜곡할 수 있기 때문에 정확도 수치 하나만 가지고 성능을 평가하지 않습니다.
- 특히 정확도는 불균형한(imbalanced) 레이블 값 분포에서 ML 모델의 성능을 판단할 경우, 적합한 평가 지표가 아닙니다.

정확도의 문제점

타이타닉 생존자 예측에서 여성은 모두 생존으로 판별

If Sex = '여성'
생존

MNIST 데이터셋을 multi classification에서 binary classification 으로 변경

0	1	2	3	4	False	False	False	False	False
0	1	2	3	4	False	False	True	False	False
5	6	7	8	9					
5	6	7	8	9					

```

import numpy as np
from sklearn.base import BaseEstimator

class MyDummyClassifier(BaseEstimator):
    # fit( ) 메소드는 아무것도 학습하지 않음.
    def fit(self, X, y=None):
        pass

    # predict( ) 메소드는 단순히 Sex feature가 1이면 0, 그렇지 않으면 1로 예측함.
    def predict(self, X):
        pred = np.zeros( ( X.shape[0], 1 ) )
        for i in range ( X.shape[0] ):
            if X['Sex'].iloc[i] == 1:
                pred[i] = 0
            else :
                pred[i] = 1

        return pred

```

BestEstimator에서 상속받은 MyDummyClassifier 생성

동일 코드 생성

```

import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Null 처리/ 채우기
def fillna(df):
    df['Age'].fillna(df['Age'].mean(), inplace=True)
    df['Cabin'].fillna('N', inplace=True)
    df['Embarked'].fillna('N', inplace=True)
    df['Fare'].fillna(0, inplace=True)
    return df

# 머신러닝 알고리즘에 불필요한 속성 제거
def drop_features(df):
    df.drop(['PassengerId', 'Name', 'Ticket'], axis=1, inplace=True)
    return df

# 레이블 인코딩 수행.
def format_features(df):
    df['Cabin'] = df['Cabin'].str[:1]
    features = ['Cabin', 'Sex', 'Embarked']
    for feature in features:
        le = LabelEncoder()
        le = le.fit(df[feature])
        df[feature] = le.transform(df[feature])
    return df

# 앞에서 설정한 Data Preprocessing 함수 호출
def transform_features(df):
    df = fillna(df)
    df = drop_features(df)
    df = format_features(df)
    return df

```

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# 원본 데이터를 재로딩, 데이터 가공, 학습데이터/테스트 데이터 분할.
titanic_df = pd.read_csv('./titanic_train.csv')
y_titanic_df = titanic_df['Survived']
X_titanic_df = titanic_df.drop('Survived', axis=1)
X_titanic_df = transform_features(X_titanic_df)
X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df, #
                                                    test_size=0.2, random_state=0)

# 위에서 생성한 Dummy Classifier를 이용하여 학습/예측/평가 수행.
myclf = MyDummyClassifier()
myclf.fit(X_train, y_train)

mypredictions = myclf.predict(X_test)
print('Dummy Classifier의 정확도는: {:.4f}'.format(accuracy_score(y_test, mypredictions)))
```

Dummy Classifier의 정확도는: 0.7877

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.base import BaseEstimator
from sklearn.metrics import accuracy_score
import numpy as np
import pandas as pd

class MyFakeClassifier(BaseEstimator):
    def fit(self,X,y):
        pass

    # 입력값으로 들어오는 X 데이터 셋의 크기만큼 모두 0값으로 만들어서 반환
    def predict(self,X):
        return np.zeros( (len(X), 1) , dtype=bool)

# 사이킷런의 내장 데이터 셋인 load_digits( )를 이용하여 MNIST 데이터 로딩
digits = load_digits()

print(digits.data)
print("### digits.data.shape:", digits.data.shape)
print(digits.target)
print("### digits.target.shape:", digits.target.shape)

[[ 0.  0.  5. ...  0.  0.  0.]
 [ 0.  0.  0. ... 10.  0.  0.]
 [ 0.  0.  0. ... 16.  9.  0.]
 ...
 [ 0.  0.  1. ...  6.  0.  0.]
 [ 0.  0.  2. ... 12.  0.  0.]
 [ 0.  0. 10. ... 12.  1.  0.]]
### digits.data.shape: (1797, 64)
[0 1 2 ... 8 9 8]
### digits.target.shape: (1797,)
```

```
digits.target == 7
array([False, False, False, ..., False, False])

# digits번호가 7번이면 True이고 0이면 astype(int)로 1로 변환, 7번이 아니면 False이고 0으로 변환.
y = (digits.target == 7).astype(int)
X_train, X_test, y_train, y_test = train_test_split(digits.data, y, random_state=11)
```

```
# 불균형한 레이블 데이터 분포도 확인.
print('레이블 테스트 세트 크기 :', y_test.shape)
print('테스트 세트 레이블 0 과 1의 분포도')
print(pd.Series(y_test).value_counts())

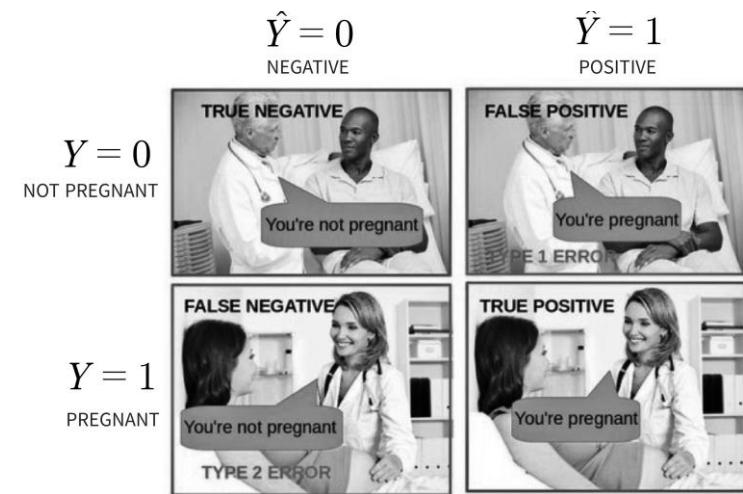
# Dummy Classifier로 학습/예측/정확도 평가
fakeclf = MyFakeClassifier()
fakeclf.fit(X_train, y_train)
fakepred = fakeclf.predict(X_test)
print('모든 예측을 0으로 하여도 정확도는:{:.3f}'.format(accuracy_score(y_test, fakepred)))
```

```
레이블 테스트 세트 크기 : (450,)
테스트 세트 레이블 0 과 1의 분포도
0    405
1     45
dtype: int64
모든 예측을 0으로 하여도 정확도는:0.900
```

Confusion Matrix (오차 행렬)

오차 행렬은 이진 분류의 예측 오류가 얼마인지와 더불어 어떠한 유형의 예측 오류가 발생하고 있는지를 함께 나타내는 지표입니다

예측 클래스(Predicted Class)			
		Negative(0)	Positive (1)
실제 클래스 (Actual Class)	Negative(0)	TN (True Negative)	FP (False Positive)
	Positive(1)	FN (False Negative)	TP (True Positive)



출처: <https://twitter.com/bearda24>

- TN는 예측값을 Negative 값 0으로 예측했고 실제 값 역시 Negative 값 0
- FP는 예측값을 Positive 값 1로 예측했는데 실제 값은 Negative 값 0
- FN은 예측값을 Negative값 0으로 예측했는데 실제 값은 Positive 값 1
- TP는 예측값을 Positive값 1로 예측했는데 실제 값 역시 Positive 값 1

오차 행렬을 통한 정확도 지표 문제점 인지

		예측 클래스	
		Negative	Positive
		TN	FP
Negative	405 개	예측 : Negative (7 이 아닌 Digit)	예측 : Positive (Digit 7)
실제 클래스	실제 : Negative (7 이 아닌 Digit)	0	실제: Negative (7 이 아닌 Digit)
Positive	45 개	FN	TP
	예측 : Negative (7 이 아닌 Digit)	예측: Positive (Digit 7)	0
	실제 : Positive (Digit 7)	실제 : Positive (Digit 7)	

정확도 = 예측 결과와 실제 값이 동일한 건수/전체 데이터 수 = $(TN + TP) / (TN + FP + FN + TP)$

```
from sklearn.metrics import confusion_matrix

# 앞절의 예측 결과인 fakepred와 실제 결과인 y_test의 Confusion Matrix 출력
confusion_matrix(y_test, fakepred)

array([[405,  0],
       [ 45,  0]], dtype=int64)
```

정밀도와 재현율

- 정밀도 = $TP / (FP + TP)$
- 재현율 = $TP / (FN + TP)$

		예측 클래스	
		Negative	Positive
실제 클래스	Negative	TN	FP
	405 개	0	
Positive	Positive	FN	TP
	45 개	0	

- 정밀도는 예측을 Positive로 한 대상 중에 예측과 실제 값이 Positive로 일치한 데이터의 비율을 뜻합니다.
- 재현율은 실제 값이 Positive인 대상 중에 예측과 실제 값이 Positive로 일치한 데이터의 비율을 뜻합니다.

MyFakeClassifier의 예측 결과로 정밀도와 재현율 측정**

```
▶ from sklearn.metrics import accuracy_score, precision_score, recall_score
    print("정밀도:", precision_score(y_test, fakepred))
    print("재현율:", recall_score(y_test, fakepred))
```

정밀도: 0.0
재현율: 0.0

정밀도와 재현율

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matrix

def get_clf_eval(y_test, pred):
    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred)
    recall = recall_score(y_test, pred)
    print('오차 행렬')
    print(confusion)
    print('정확도: {:.4f}, 정밀도: {:.4f}, 재현율: {:.4f}'.format(accuracy, precision, recall))
```

```
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# 원본 데이터를 재로딩, 데이터 가공, 학습데이터/테스트 데이터 분할.
titanic_df = pd.read_csv('./titanic_train.csv')
y_titanic_df = titanic_df['Survived']
X_titanic_df = titanic_df.drop('Survived', axis=1)
X_titanic_df = transform_features(X_titanic_df)

X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df, # 
                                                    test_size=0.20, random_state=11)

lr_clf = LogisticRegression()

lr_clf.fit(X_train, y_train)
pred = lr_clf.predict(X_test)
get_clf_eval(y_test, pred)
```

오차 행렬
[[104 14]
 [13 48]]

정확도: 0.8492, 정밀도: 0.7742, 재현율: 0.7869

Sensitivity : 질병이 있는데 질병이 있다고 판단 (재현율)

Specificity : 질병이 없는데 질병이 없다고 판단

- 재현율이 상대적으로 더 중요한 지표인 경우는 실제 Positive 양성인 데이터 예측을 Negative로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우 : 암 진단, 금융사기 판별
- 정밀도가 상대적으로 더 중요한 지표인 경우는 실제 Negative 음성인 데이터 예측을 Positive 양성으로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우: 스팸 메일

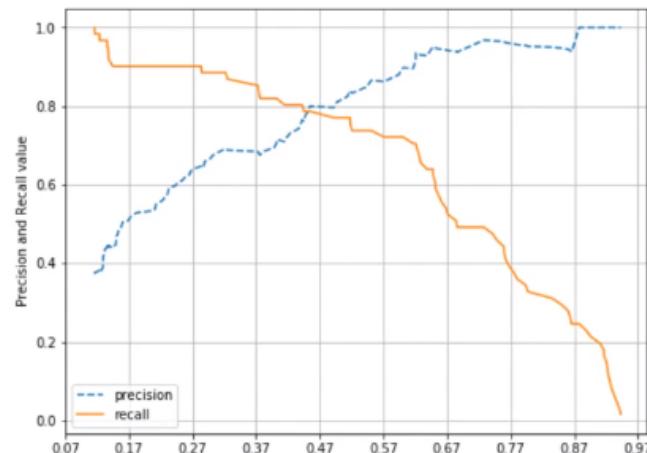
★ 불균형한 레이블 클래스를 가지는 이진 분류 모델에서는 많은 데이터 중에서 중점적으로 찾아야 하는 매우 적은 수의 결괏값에 Positive를 설정해 1값을 부여하고, 그렇지 않은 경우는 Negative로 0 값을 일반적으로 부여합니다.

- **정밀도/재현율 Trade Off**

- 정밀도와 재현율이 강조될 경우 Threshold를 조정해 해당 수치 조정 가능
- 상호 보완적인 수치 이므로 Trade off 작용

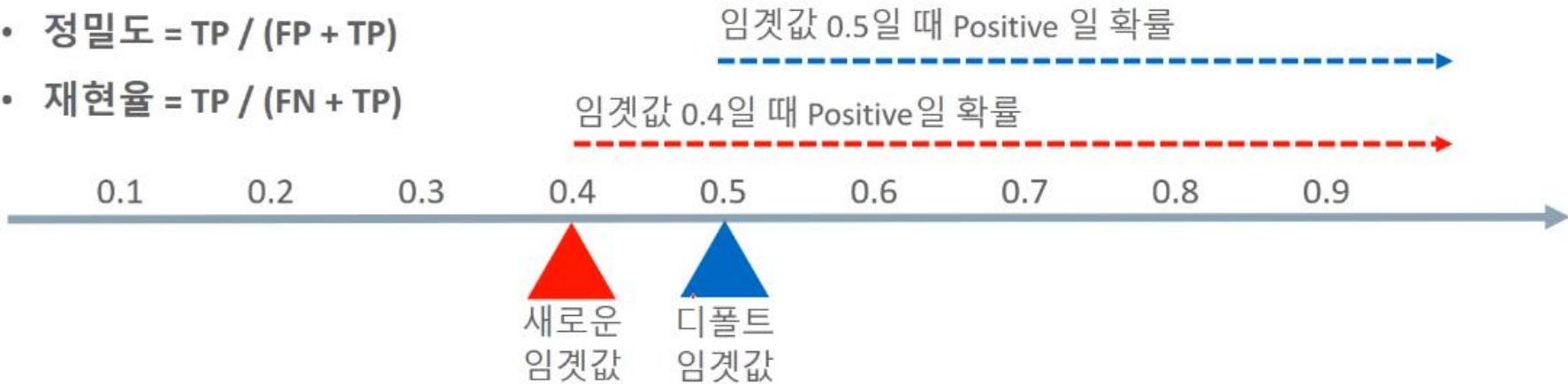
정밀도와 재현율 Trade Off

- 분류하려는 업무의 특성상 정밀도 또는 재현율이 특별히 강조돼야 할 경우 분류의 결정 임곗값(Threshold)을 조정해 정밀도 또는 재현율의 수치를 높일 수 있습니다.
- 하지만 정밀도와 재현율은 상호 보완적인 평가 지표이기 때문에 어느 한쪽을 강제로 높이면 다른 하나의 수치는 떨어지기 쉽습니다. 이를 정밀도/재현율의 트레이드오프(Trade-off)라고 부릅니다.



분류 결정 임곗값에 따른 Positive 예측 확률 변화

- 정밀도 = $TP / (FP + TP)$
- 재현율 = $TP / (FN + TP)$



분류 결정 임곗값이 낮아질 수록 Positive로 예측할 확률이 높아짐. 재현율 증가

- 사이킷런 Estimator 객체의 `predict_proba()` 메소드는 분류 결정 예측 확률을 반환합니다.
- 이를 이용하면 임의로 분류 결정 임곗값을 조정하면서 예측 확률을 변경할 수 있습니다.
- 사이킷런은 `precision_recall_curve()` 함수를 통해 임곗값에 따른 정밀도, 재현율의 변화값을 제공합니다.

분류 결정 임곗값에 따른 Positive 예측 확률 변화

```
pred_proba = lr_clf.predict_proba(X_test)
pred = lr_clf.predict(X_test)
print('pred_proba()결과 Shape : {}'.format(pred_proba.shape))
print('pred_proba array에서 앞 3개만 샘플로 추출 \n:', pred_proba[:3])

# 예측 확률 array 와 예측 결과값 array 를 concatenate 하여 예측 확률과 결과값을 한눈에 확인
pred_proba_result = np.concatenate([pred_proba, pred.reshape(-1,1)],axis=1)
print('두개의 class 중에서 더 큰 확률을 클래스 값으로 예측 \n',pred_proba_result[:3])
```

```
pred_proba()결과 Shape : (179, 2)
pred_proba array에서 앞 3개만 샘플로 추출
: [[0.46191502 0.53808498]
 [0.87867074 0.12132926]
 [0.87715565 0.12284435]]
앞이 0, 뒤가 1이 될 확률
두개의 class 중에서 더 큰 확률을 클래스 값으로 예측
[[0.46191502 0.53808498 1.]
 [0.87867074 0.12132926 0.]
 [0.87715565 0.12284435 0.]]
```

```
from sklearn.preprocessing import Binarizer

X = [[ 1, -1,  2],
      [ 2,  0,  0],
      [ 0,  1.1, 1.2]]                                Binarizer 활용

# threshold 기준값보다 같거나 작으면 0을, 크면 1을 반환
binarizer = Binarizer(threshold=1.1)
print(binarizer.fit_transform(X))

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

분류 결정 임곗값에 따른 Positive 예측 확률 변화

```
from sklearn.preprocessing import Binarizer

#Binarizer의 threshold 설정값. 분류 결정 임곗값임.
custom_threshold = 0.5

# predict_proba( ) 반환값의 두번째 컬럼, 즉 Positive 클래스 컬럼 하나만 추출하여 Binarizer를 적용
pred_proba_1 = pred_proba[:,1].reshape(-1,1)

binarizer = Binarizer(threshold=custom_threshold).fit(pred_proba_1)
custom_predict = binarizer.transform(pred_proba_1)

get_clf_eval(y_test, custom_predict)
```

오차 행렬
[[104 14]
 [13 48]]
정확도: 0.8492, 정밀도: 0.7742, 재현율: 0.7869 앞의 결과와 동일

```
# Binarizer의 threshold 설정값을 0.4로 설정. 즉 분류 결정 임곗값을 0.5에서 0.4로 낮춤
custom_threshold = 0.4      임계값 변동
pred_proba_1 = pred_proba[:,1].reshape(-1,1)
binarizer = Binarizer(threshold=custom_threshold).fit(pred_proba_1)
custom_predict = binarizer.transform(pred_proba_1)

get_clf_eval(y_test, custom_predict)
```

오차 행렬
[[98 20]
 [10 51]]
정확도: 0.8324, 정밀도: 0.7183, 재현율: 0.8361

분류 결정 임곗값에 따른 Positive 예측 확률 변화

```
# 테스트를 수행할 모든 임곗값을 리스트 객체로 저장.
thresholds = [0.4, 0.45, 0.50, 0.55, 0.60]

def get_eval_by_threshold(y_test, pred_proba_c1, thresholds):
    # thresholds list 객체내의 값을 차례로 iteration하면서 Evaluation 수행.
    for custom_threshold in thresholds:
        binarizer = Binarizer(threshold=custom_threshold).fit(pred_proba_c1)
        custom_predict = binarizer.transform(pred_proba_c1)
        print('임곗값:', custom_threshold)
        get_clf_eval(y_test, custom_predict)

get_eval_by_threshold(y_test, pred_proba[:, 1].reshape(-1, 1), thresholds)
```

```
임곗값: 0.4
오차 행렬
[[98 20]
 [10 51]]
정확도: 0.8324, 정밀도: 0.7183, 재현율: 0.8361
임곗값: 0.45
오차 행렬
[[103 15]
 [12 49]]
정확도: 0.8492, 정밀도: 0.7656, 재현율: 0.8033
임곗값: 0.5
오차 행렬
[[104 14]
 [13 48]]
정확도: 0.8492, 정밀도: 0.7742, 재현율: 0.7869
임곗값: 0.55
오차 행렬
[[109 9]
 [15 46]]
정확도: 0.8659, 정밀도: 0.8364, 재현율: 0.7541
임곗값: 0.6
오차 행렬
[[112 6]
 [16 45]]
정확도: 0.8771, 정밀도: 0.8824, 재현율: 0.7377
```

precision_recall_curve() 를 이용하여 임곗값에 따른 정밀도-재현율 값 추출

```
from sklearn.metrics import precision_recall_curve

# 레이블 값이 1일때의 예측 확률을 추출
pred_proba_class1 = lr_clf.predict_proba(X_test)[:, 1]

# 실제값 데이터 셋과 레이블 값이 1일 때의 예측 확률을 precision_recall_curve 인자로 입력
precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_class1)
print('반환된 분류 결정 임곗값 배열의 Shape:', thresholds.shape)
print('반환된 precisions 배열의 Shape:', precisions.shape)
print('반환된 recalls 배열의 Shape:', recalls.shape)

print("thresholds 5 sample:", thresholds[:5])
print("precisions 5 sample:", precisions[:5])
print("recalls 5 sample:", recalls[:5])

#반환된 임계값 배열 로우가 147건이므로 샘플로 10건만 추출하되, 임곗값을 15 Step으로 추출.
thr_index = np.arange(0, thresholds.shape[0], 15)
print('샘플 추출을 위한 임계값 배열의 index 10개:', thr_index)
print('샘플용 10개의 임곗값:', np.round(thresholds[thr_index], 2))

# 15 step 단위로 추출된 임계값에 따른 정밀도와 재현율 값
print('샘플 임계값별 정밀도:', np.round(precisions[thr_index], 3))
print('샘플 임계값별 재현율:', np.round(recalls[thr_index], 3))
```

```
반환된 분류 결정 임곗값 배열의 Shape: (143,)
반환된 precisions 배열의 Shape: (144,)
반환된 recalls 배열의 Shape: (144,)
thresholds 5 sample: [0.10392828 0.10393054 0.1039559 0.10786026 0.10890049]
precisions 5 sample: [0.38853503 0.38461538 0.38709677 0.38961039 0.38562092]
recalls 5 sample: [1. 0.98360656 0.98360656 0.98360656 0.96721311]
샘플 추출을 위한 임계값 배열의 index 10개: [ 0 15 30 45 60 75 90 105 120 135]
샘플용 10개의 임곗값: [0.1 0.12 0.14 0.19 0.28 0.4 0.56 0.67 0.82 0.95]
샘플 임계값별 정밀도: [0.389 0.44 0.466 0.539 0.647 0.729 0.836 0.949 0.958 1.]
샘플 임계값별 재현율: [1. 0.967 0.902 0.902 0.836 0.754 0.607 0.377 0.148]
```

임곗값의 변경에 따른 정밀도-재현율 변화 곡선을 그림

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
%matplotlib inline

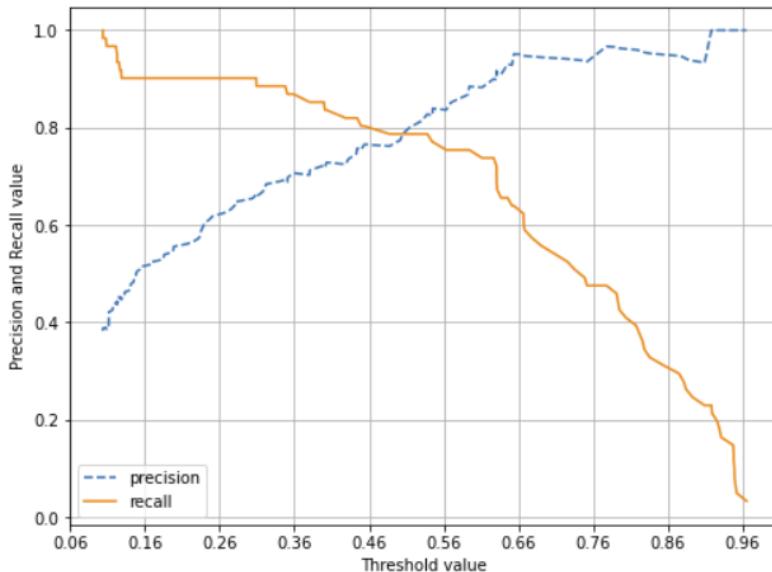
def precision_recall_curve_plot(y_test, pred_proba_c1):
    # threshold ndarray와 0/ threshold에 따른 정밀도, 재현율 ndarray 추출.
    precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_c1)

    # X축을 threshold값으로, Y축은 정밀도, 재현율 값으로 각각 Plot 수행. 정밀도는 점선으로 표시
    plt.figure(figsize=(8,6))
    threshold_boundary = thresholds.shape[0]
    plt.plot(thresholds, precisions[0:threshold_boundary], linestyle='--', label='precision')
    plt.plot(thresholds, recalls[0:threshold_boundary], label='recall')

    # threshold 값 X 축의 Scale을 0.1 단위로 변경
    start, end = plt.xlim()
    plt.xticks(np.round(np.arange(start, end, 0.1),2))

    # x축, y축 label과 legend, 그리고 grid 설정
    plt.xlabel('Threshold value'); plt.ylabel('Precision and Recall value')
    plt.legend(); plt.grid()
    plt.show()

precision_recall_curve_plot(y_test, lr_clf.predict_proba(X_test)[:, 1])
```



정밀도와 재현율의 맹점

정밀도를 100%로 만드는 법

- 확실한 기준이 되는 경우만 Positive로 예측하고 나머지는 모두 Negative로 예측합니다. 정밀도 = $TP / (TP + FP)$ 입니다. 전체 환자 1000명 중 확실한 Positive 징후만 가진 환자는 단 1명이라고 하면 이 한 명만 Positive로 예측하고 나머지는 모두 Negative로 예측하더라도 FP는 0, TP는 1이 되므로 정밀도는 $1/(1+0)$ 으로 100%가 됩니다

재현율을 100%로 만드는 법

- 모든 환자를 Positive로 예측하면 됩니다. 재현율 = $TP / (TP + FN)$ 이므로 전체 환자 1000명을 다 Positive로 예측하는 겁니다. 이 중 실제 양성인 사람이 30명 정도라도 TN이 수치에 포함되지 않고 FN은 아예 0이므로 $30/(30 + 0)$ 으로 100%가 됩니다.

F1 Score

F1 스코어(Score)는 정밀도와 재현율을 결합한 지표입니다. F1 스코어는 정밀도와 재현율이 어느 한쪽으로 치우치지 않는 수치를 나타낼 때 상대적으로 높은 값을 가집니다. F1 스코어의 공식은 다음과 같습니다

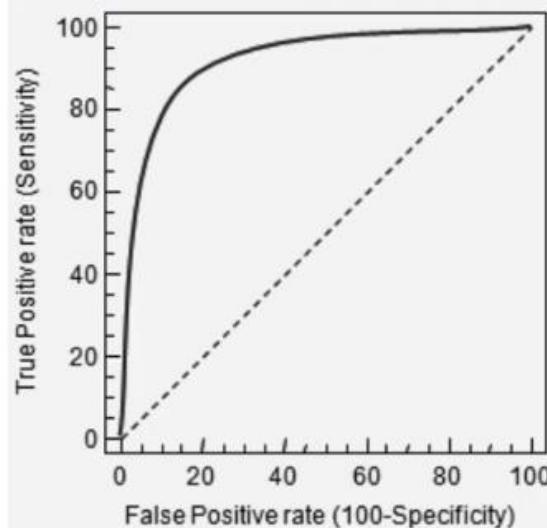
$$F1 = \frac{2}{\frac{1}{recall} + \frac{1}{precision}} = 2 * \frac{precision * recall}{precision + recall}$$

만일 A 예측 모델의 경우 정밀도가 0.9, 재현율이 0.1로 극단적인 차이가 나고, B 예측 모델은 정밀도가 0.5, 재현율이 0.5로 정밀도와 재현율이 큰 차이가 없다면 A 예측 모델의 F1 스코어는 0.18이고, B 예측 모델의 F1 스코어는 0.5로 B 모델이 A 모델에 비해 매우 우수한 F1 스코어를 가지게 됩니다.

사이킷런은 f1 score를 위해 f1_score() 함수를 제공합니다.

ROC 곡선과 AUC

- ROC 곡선(Receiver Operation Characteristic Curve) 과 이에 기반한 AUC 스코어는 이진 분류의 예측 성능 측정에서 중요하게 사용되는 지표입니다. 일반적으로 의학 분야에서 많이 사용되지만, 머신러닝의 이진 분류 모델의 예측 성능을 판단하는 중요한 평가 지표이기도 합니다.

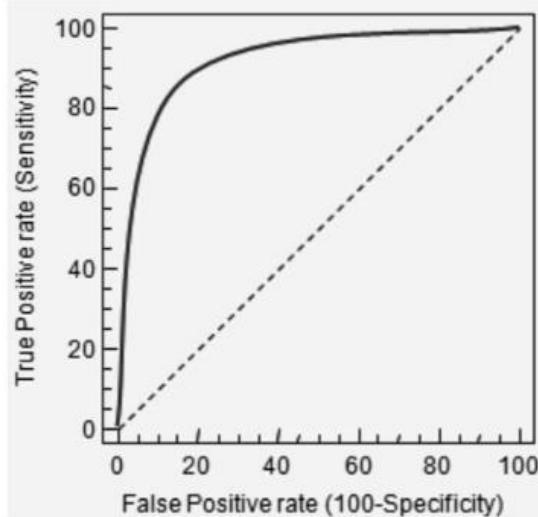


ROC 곡선은 FPR(False Positive Rate)이 변할 때 TPR(True Positive Rate)이 어떻게 변하는지를 나타내는 곡선입니다. FPR을 X 축으로, TPR을 Y 축으로 잡으면 FPR의 변화에 따른 TPR의 변화가 곡선 형태로 나타납니다.

분류의 성능 지표로 사용되는 것은 ROC 곡선 면적에 기반한 AUC 값으로 결정합니다. AUC(Area Under Curve) 값은 ROC 곡선 밑의 면적을 구한 것으로서 일반적으로 1에 가까울수록 좋은 수치입니다

ROC 곡선

FPR의 변화에 따른 TPR의 변화 곡선



- **TPR**은 True Positive Rate의 약자이며, 이는 재현율을 나타냅니다. 따라서 **TPR은 $TP / (FN + TP)$** 입니다. TPR, 즉 재현율은 민감도로도 불립니다.
- **FPR**은 실제 Negative(음성)을 잘못 예측한 비율을 나타냅니다. 즉 실제는 Negative인데 Positive 또는 Negative로 예측한 것 중 Positive로 잘못 예측한 비율입니다. **FPR = $FP / (FP + TN)$** 입니다.

사이킷런 ROC 곡선 및 AUC 스코어

- 사이킷런은 임곗값에 따른 ROC 곡선 데이터를 `roc_curve()`로, AUC 스코어를 `roc_auc_score()` 함수로 제공

API 명	입력 파라미터	반환 값
<code>roc_curve(y_true, y_score)</code>	<ul style="list-style-type: none"><code>y_true</code>: 실제 클래스 값 array (array shape = [데이터 건수])<code>y_score</code>: <code>predict_prob()</code>의 반환 값 array에서 Positive 컬럼의 예측 확률이 보통 사용됨. Binary 분류 시 shape = [n_samples]	<code>fpr</code> : <code>fpr</code> 값을 array로 반환 <code>tpr</code> : <code>tpr</code> 값을 array로 반환 <code>thresholds</code> : <code>threshold</code> 값을 array
<code>roc_auc_score(y_true, y_score)</code>	<ul style="list-style-type: none"><code>y_true</code>: 실제 클래스 값 array (array shape = [데이터 건수])<code>y_score</code>: <code>predict_prob()</code>의 반환 값 array에서 Positive 컬럼의 예측 확률이 보통 사용됨. Binary 분류 시 shape = [n_samples]	AUC 스코어 값

사이킷런 ROC 곡선 및 AUC 스코어

```
from sklearn.metrics import f1_score
f1 = f1_score(y_test, pred)
print('F1 스코어: {:.4f}'.format(f1))
```

F1 스코어: 0.7805

```
def get_clf_eval(y_test, pred):
    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred)
    recall = recall_score(y_test, pred)
    # F1 스코어 추가
    f1 = f1_score(y_test, pred)
    print('오차 행렬')
    print(confusion)
    # f1 score print 추가
    print('정확도: {:.4f}, 정밀도: {:.4f}, 재현율: {:.4f}, F1: {:.4f}'.format(accuracy, precision, recall, f1))
```

```
thresholds = [0.4, 0.45, 0.5, 0.55, 0.60]
pred_proba = lr_clf.predict_proba(X_test)
get_eval_by_threshold(y_test, pred_proba[:, 1].reshape(-1, 1), thresholds)
```

임곗값: 0.4
오차 행렬
[[99 19]
 [10 51]]
정확도: 0.8380, 정밀도: 0.7286, 재현율: 0.8361, F1: 0.7786
임곗값: 0.45
오차 행렬
[[103 15]
 [12 49]]
정확도: 0.8492, 정밀도: 0.7656, 재현율: 0.8033, F1: 0.7840
임곗값: 0.5
오차 행렬
[[104 14]
 [13 48]]
정확도: 0.8492, 정밀도: 0.7742, 재현율: 0.7869, F1: 0.7805
임곗값: 0.55
오차 행렬
[[109 9]
 [15 46]]
정확도: 0.8659, 정밀도: 0.8364, 재현율: 0.7541, F1: 0.7931
임곗값: 0.6
오차 행렬
[[112 6]
 [16 45]]
정확도: 0.8771, 정밀도: 0.8824, 재현율: 0.7377, F1: 0.8036

사이킷런 ROC 곡선 및 AUC 스코어

```
from sklearn.metrics import roc_curve

# 레이블 값이 1일때의 예측 확률을 추출
pred_proba_class1 = lr_clf.predict_proba(X_test)[:, 1]

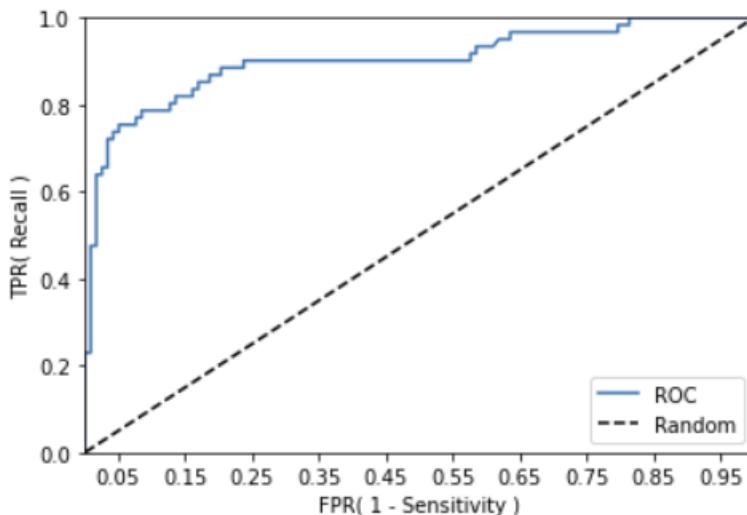
fprs, tprs, thresholds = roc_curve(y_test, pred_proba_class1)
# 반환된 임곗값 배열에서 샘플로 데이터를 추출하되, 임곗값을 5 Step으로 추출.
# thresholds[0]은 max(예측확률)+1로 임의 설정됨. 이를 제외하기 위해 np.arange는 1부터 시작
thr_index = np.arange(1, thresholds.shape[0], 5)
print('샘플 추출을 위한 임곗값 배열의 index:', thr_index)
print('샘플 index로 추출한 임곗값: ', np.round(thresholds[thr_index], 2))

# 5 step 단위로 추출된 임계값에 따른 FPR, TPR 값
print('샘플 임곗값별 FPR: ', np.round(fprs[thr_index], 3))
print('샘플 임곗값별 TPR: ', np.round(tprs[thr_index], 3))
```

```
샘플 추출을 위한 임곗값 배열의 index: [ 1  6 11 16 21 26 31 36 41 46 51]
샘플 index로 추출한 임곗값:  [0.97 0.65 0.63 0.56 0.45 0.4  0.35 0.15 0.13 0.11 0.11]
샘플 임곗값별 FPR:  [0.      0.017 0.034 0.076 0.127 0.169 0.203 0.466 0.585 0.686 0.797]
샘플 임곗값별 TPR:  [0.033 0.639 0.721 0.754 0.803 0.836 0.885 0.902 0.934 0.967 0.984]
```

사이킷런 ROC 곡선 및 AUC 스코어

```
def roc_curve_plot(y_test, pred_proba_c1):  
    # 임곗값에 따른 FPR, TPR 값을 반환 받음.  
    fprs, tprs, thresholds = roc_curve(y_test, pred_proba_c1)  
  
    # ROC Curve를 plot 곡선으로 그림.  
    plt.plot(fprs, tprs, label='ROC')  
    # 가운데 대각선 직선을 그림.  
    plt.plot([0, 1], [0, 1], 'k--', label='Random')  
  
    # FPR X 축의 Scale을 0.1 단위로 변경, X,Y 축명 설정 등  
    start, end = plt.xlim()  
    plt.xticks(np.round(np.arange(start, end, 0.1), 2))  
    plt.xlim(0, 1); plt.ylim(0, 1)  
    plt.xlabel('FPR( 1 - Sensitivity )'); plt.ylabel('TPR( Recall )')  
    plt.legend()  
    plt.show()  
  
roc_curve_plot(y_test, lr_clf.predict_proba(X_test)[:, 1])
```



```
from sklearn.metrics import roc_auc_score  
  
#pred = lr_clf.predict(X_test)  
#roc_score = roc_auc_score(y_test, pred)  
  
pred_proba = lr_clf.predict_proba(X_test)[:, 1]  
roc_score = roc_auc_score(y_test, pred_proba)  
print('ROC AUC 값: {:.4f}'.format(roc_score))
```

ROC AUC 값: 0.9024

피마 인디언 당뇨병 예측

피마 인디언 당뇨병(Pima Indian Diabetes) 데이터 세트를 이용해 당뇨병 여부를 판단하는 머신러닝 예측 모델을 수립하고, 지금까지 설명한 평가 지표를 적용해 보겠습니다.



Data_Set

kaggle Search  Competitions Datasets Notebooks Discussion Courses ...

 Dataset

Pima Indians Diabetes Database
Predict the onset of diabetes based on diagnostic measures

 UCI Machine Learning • updated 3 years ago (Version 1)

Data Kernels (620) Discussion (11) Activity Metadata Download (9 KB) **New Notebook** 

Your Dataset download has started.
Show your appreciation with an upvote  667

 Usability 8.8  License CC0: Public Domain  Tags healthcare, society, india, health conditions, endocrine conditions

Description

Data_set>Loading

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, roc_auc_score
from sklearn.metrics import f1_score, confusion_matrix, precision_recall_curve, roc_curve
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

diabetes_data = pd.read_csv('diabetes.csv')
print(diabetes_data['Outcome'].value_counts())
diabetes_data.head(3)
```

```
0    500
1    268
Name: Outcome, dtype: int64
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome	
0	6	148	72	35	0	33.6		0.627	50	1
1	1	85	66	29	0	26.6		0.351	31	0
2	8	183	64	0	0	23.3		0.672	32	1

```
diabetes_data.info( )
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
Pregnancies          768 non-null int64
Glucose              768 non-null int64
BloodPressure        768 non-null int64
SkinThickness        768 non-null int64
Insulin              768 non-null int64
BMI                  768 non-null float64
DiabetesPedigreeFunction 768 non-null float64
Age                  768 non-null int64
Outcome              768 non-null int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

- Pregnancies: 임신 횟수
- Glucose: 포도당 부하 검사 수치
- BloodPressure: 혈압(mm Hg)
- SkinThickness: 팔 삼두근 뒤쪽의 피하지방 측정값(mm)
- Insulin: 혈청 인슐린(mu U/ml)
- BMI: 체질량지수(체중(kg)/(키(m))^2)
- DiabetesPedigreeFunction: 당뇨 내력 가중치 값
- Age: 나이
- Outcome: 클래스 결정 값(0또는 1)

Logistic Regression으로 학습 및 예측

```
# 피처 데이터 세트 X, 레이블 데이터 세트 y를 추출.  
# 맨 끝이 Outcome 컬럼으로 레이블 값임. 컬럼 위치 -1을 이용해 추출  
X = diabetes_data.iloc[:, :-1]  
y = diabetes_data.iloc[:, -1]  
  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 156, stratify=y)  
  
# 로지스틱 회귀로 학습, 예측 및 평가 수행.  
lr_clf = LogisticRegression()  
lr_clf.fit(X_train, y_train)  
pred = lr_clf.predict(X_test)  
pred_proba = lr_clf.predict_proba(X_test)[:, 1]  
  
get_clf_eval(y_test, pred, pred_proba)
```

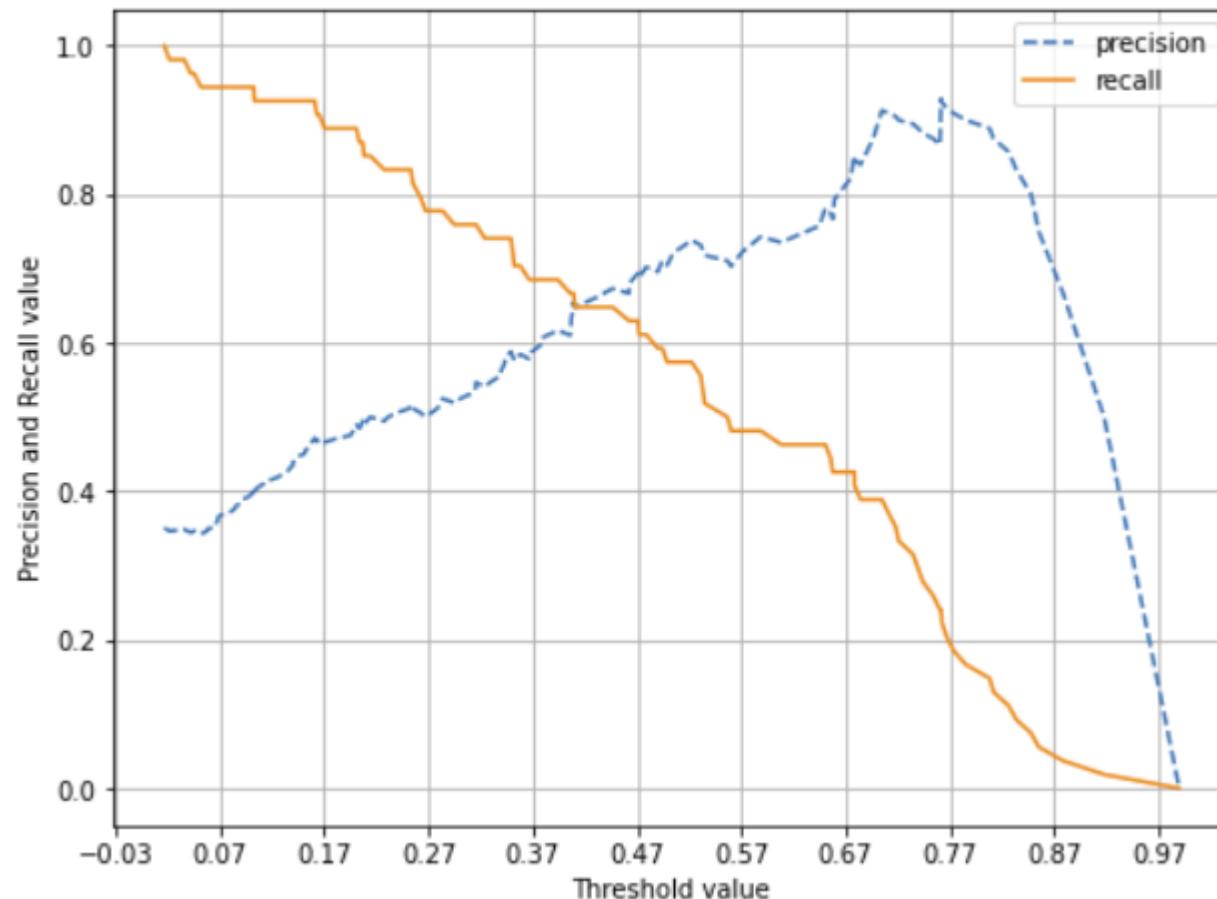
오차 행렬

```
[[88 12]  
 [23 31]]
```

정확도: 0.7727, 정밀도: 0.7209, 재현율: 0.5741, F1: 0.6392, AUC: 0.7919

Precision recall 곡선 그림.

```
pred_proba_c1 = lr_clf.predict_proba(X_test)[:, 1]
precision_recall_curve_plot(y_test, pred_proba_c1)
```



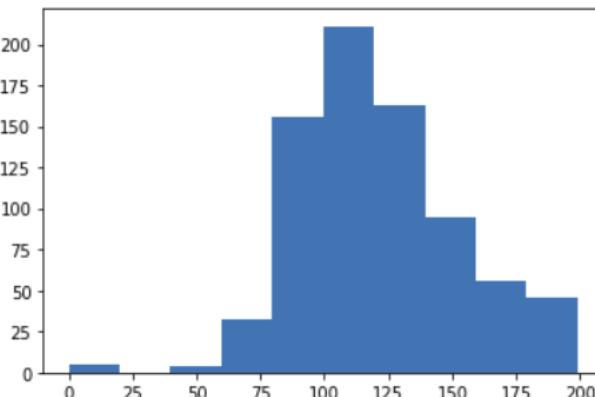
Data_set_확인

```
diabetes_data.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

```
plt.hist(diabetes_data['Glucose'], bins=10)
```

```
(array([ 5.,  0.,  4., 32., 156., 211., 163.,  95.,  56.,  46.]),
 array([ 0., 19.9, 39.8, 59.7, 79.6, 99.5, 119.4, 139.3, 159.2,
 179.1, 199.]),
 <a list of 10 Patch objects>)
```



Tune dataset!

```
# 0값을 검사할 피처명 리스트 객체 설정
zero_features = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']

# 전체 데이터 건수
total_count = diabetes_data['Glucose'].count()

# 피처별로 반복하면서 데이터 값이 0인 데이터 건수 추출하고, 퍼센트 계산
for feature in zero_features:
    zero_count = diabetes_data[diabetes_data[feature] == 0][feature].count()
    print('{0} 0 건수는 {1}, 퍼센트는 {2:.2f} %'.format(feature, zero_count, 100*zero_count/total_count))
```

Glucose 0 건수는 5, 퍼센트는 0.65 %
BloodPressure 0 건수는 35, 퍼센트는 4.56 %
SkinThickness 0 건수는 227, 퍼센트는 29.56 %
Insulin 0 건수는 374, 퍼센트는 48.70 %
BMI 0 건수는 11, 퍼센트는 1.43 %

```
# zero_features 리스트 내부에 저장된 개별 피처들에 대해서 0값을 평균 값으로 대체
diabetes_data[zero_features] = diabetes_data[zero_features].replace(0, diabetes_data[zero_features].mean())
```

```
X = diabetes_data.iloc[:, :-1]
y = diabetes_data.iloc[:, -1]

# StandardScaler 클래스를 이용해 피처 데이터 세트에 일괄적으로 스케일링 적용
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size = 0.2, random_state = 156, stratify=y)

# 로지스틱 회귀로 학습, 예측 및 평가 수행.
lr_clf = LogisticRegression()
lr_clf.fit(X_train, y_train)
pred = lr_clf.predict(X_test)
pred_proba = lr_clf.predict_proba(X_test)[:, 1]

get_clf_eval(y_test, pred, pred_proba)
```

오차 행렬

[[90 10]

[21 33]]

정확도: 0.7987, 정밀도: 0.7674, 재현율: 0.6111, F1: 0.6804, AUC: 0.8433

Tune dataset!

```
from sklearn.preprocessing import Binarizer

def get_eval_by_threshold(y_test, pred_proba_cl, thresholds):
    # thresholds 리스트 각체내의 값을 차례로 iteration하면서 Evaluation 수행.
    for custom_threshold in thresholds:
        binarizer = Binarizer(threshold=custom_threshold).fit(pred_proba_cl)
        custom_predict = binarizer.transform(pred_proba_cl)
        print('임곗값:', custom_threshold)
        get_clf_eval(y_test, custom_predict, pred_proba_cl)
```

```
thresholds = [0.3, 0.33, 0.36, 0.39, 0.42, 0.45, 0.48, 0.50]
pred_proba = lr_clf.predict_proba(X_test)
get_eval_by_threshold(y_test, pred_proba[:, 1].reshape(-1, 1), thresholds)
```

```
임곗값: 0.3
오차 행렬
[[65 35]
 [11 43]]
정확도: 0.7013, 정밀도: 0.5513, 재현율: 0.7963, F1: 0.6515, AUC: 0.8433
임곗값: 0.33
오차 행렬
[[71 29]
 [11 43]]
정확도: 0.7403, 정밀도: 0.5972, 재현율: 0.7963, F1: 0.6825, AUC: 0.8433
임곗값: 0.36
오차 행렬
[[76 24]
 [15 39]]
정확도: 0.7468, 정밀도: 0.6190, 재현율: 0.7222, F1: 0.6667, AUC: 0.8433
임곗값: 0.39
오차 행렬
[[78 22]
 [16 38]]
정확도: 0.7532, 정밀도: 0.6333, 재현율: 0.7037, F1: 0.6667, AUC: 0.8433
임곗값: 0.42
오차 행렬
[[84 16]
 [18 36]]
정확도: 0.7792, 정밀도: 0.6923, 재현율: 0.6667, F1: 0.6792, AUC: 0.8433
임곗값: 0.45
오차 행렬
[[85 15]
 [18 36]]
정확도: 0.7857, 정밀도: 0.7059, 재현율: 0.6667, F1: 0.6857, AUC: 0.8433
임곗값: 0.48
오차 행렬
[[88 12]
 [19 35]]
정확도: 0.7987, 정밀도: 0.7447, 재현율: 0.6481, F1: 0.6931, AUC: 0.8433
임곗값: 0.5
오차 행렬
[[90 10]
 [21 33]]
정확도: 0.7987, 정밀도: 0.7674, 재현율: 0.6111, F1: 0.6804, AUC: 0.8433
```

```
# 임곗값을 0.48로 설정한 Binarizer 생성
binarizer = Binarizer(threshold=0.48)

# 위에서 구한 lr_clf의 predict_proba() 예측 확률 array에서 1에 해당하는 원형값을 Binarizer 변환.
pred_th_048 = binarizer.fit_transform(pred_proba[:, 1].reshape(-1, 1))

get_clf_eval(y_test, pred_th_048, pred_proba[:, 1])
```

```
오차 행렬
[[88 12]
 [19 35]]
정확도: 0.7987, 정밀도: 0.7447, 재현율: 0.6481, F1: 0.6931, AUC: 0.8433
```



```
임곗값: 0.48
오차 행렬
[[88 12]
 [19 35]]
정확도: 0.7987, 정밀도: 0.7447, 재현율: 0.6481, F1: 0.6931, AUC: 0.8433
임곗값: 0.5
오차 행렬
[[90 10]
 [21 33]]
정확도: 0.7987, 정밀도: 0.7674, 재현율: 0.6111, F1: 0.6804, AUC: 0.8433
```

평가 Summary

- 이진 분류에서 정밀도, 재현율, F1 스코어, AUC 스코어가 주로 성능 평가 지표가 활용됩니다.
- 오차 행렬은 실제 클래스 값과 예측 클래스 값의 True, False에 따라 TN, FP, FN, TP로 매핑 되는 4분면 행렬을 제공합니다.
- 정밀도와 재현율은 Positive 데이터 세트의 예측 성능에 좀 더 초점을 맞춘 지표이며, 분류 결정 임계 값을 조정해 정밀도 또는 재현율은 수치를 높이거나 낮출 수 있습니다.
- F1 스코어는 정밀도와 재현율이 어느 한쪽으로 치우치지 않을 때 좋은 값을 가집니다.
- AUC 스코어는 ROC 곡선 밑의 면적을 구한 것으로 1에 가까울 수록 좋은 수치입니다.

Chapter 04

머신러닝 분류

분류 알고리즘

분류(Classification)는 학습 데이터로 주어진 데이터의 피처와 레이블값(결정 값, 클래스 값)을 머신러닝 알고리즘으로 학습해 모델을 생성하고, 이렇게 생성된 모델에 새로운 데이터 값이 주어졌을 때 미지의 레이블 값을 예측하는 것입니다.

대표적인 분류 알고리즘들

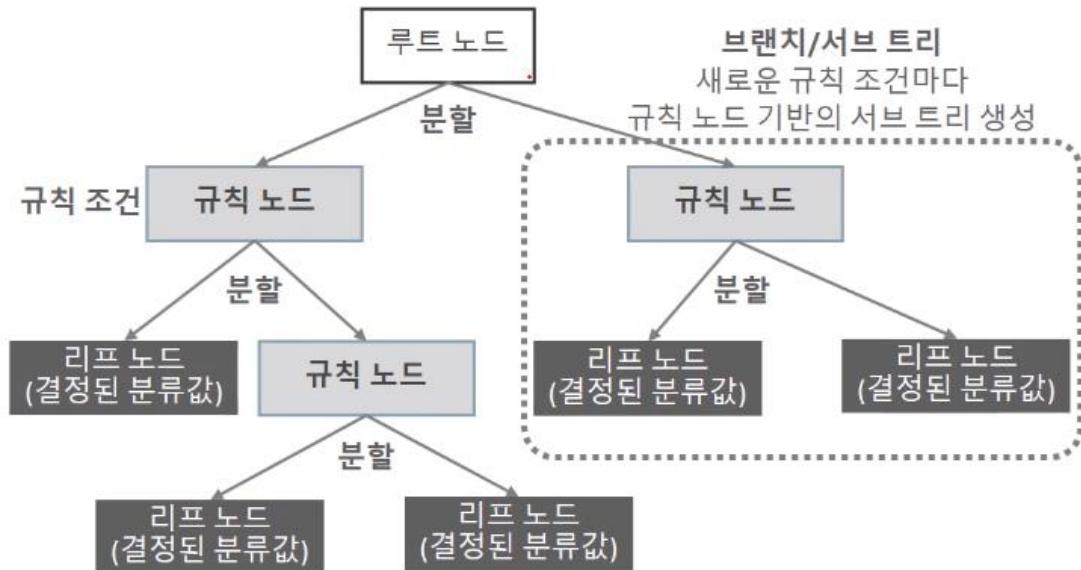
- 베이즈(Bayes) 통계와 생성 모델에 기반한 나이브 베이즈(Naïve Bayes)
- 독립변수와 종속변수의 선형 관계성에 기반한 로지스틱 회귀(Logistic Regression)
- 데이터 균일도에 따른 규칙 기반의 결정 트리(Decision Tree)
- 개별 클래스 간의 최대 분류 마진을 효과적으로 찾아주는 서포트 벡터 머신(Support Vector Machine)
- 근접 거리를 기준으로 하는 최소 근접(Nearest Neighbor) 알고리즘
- 심층 연결 기반의 신경망(Neural Network)
- 서로 다른(또는 같은) 머신러닝 알고리즘을 결합한 앙상블(Ensemble)

결정트리와 앙상블

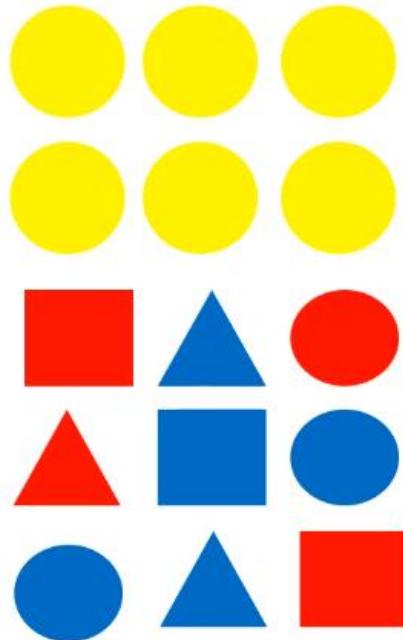
- 결정 트리는 매우 쉽고 유연하게 적용될 수 있는 알고리즘입니다. 또한 데이터의 스케일링이나 정규화 등의 사전 가공의 영향이 매우 적습니다. 하지만 예측 성능을 향상시키기 위해 복잡한 규칙 구조를 가져야 하며, 이로 인한 과적합(overfitting)이 발생해 반대로 예측 성능이 저하될 수도 있다는 단점이 있습니다.
- 하지만 이러한 단점이 앙상블 기법에서는 오히려 장점으로 작용합니다. 앙상블은 매우 많은 여러 개의 약한 학습기(즉, 예측 성능이 상대적으로 떨어지는 학습 알고리즘)를 결합해 확률적 보완과 오류가 발생한 부분에 대한 가중치를 계속 업데이트하면서 예측 성능을 향상시키는데, 결정 트리가 좋은 약한 학습기가 되기 때문입니다(GBM, XGBoost, LightGBM 등)

Decision Tree

- 결정 트리 알고리즘은 데이터에 있는 규칙을 학습을 통해 자동으로 찾아내 트리(Tree) 기반의 분류 규칙을 만듭니다(If-Else 기반 규칙)
- 따라서 데이터의 어떤 기준을 바탕으로 규칙을 만들어야 가장 효율적인 분류가 될 것인가가 알고리즘의 성능을 크게 좌우합니다



균일도 기반 규칙 조건



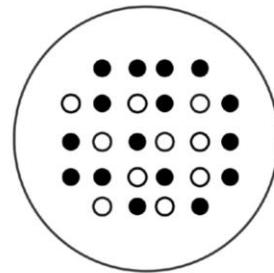
노랑색 블록의 경우 모두 동그라미로 구성되고, 빨강과 파랑 블록의 경우는 동그라미, 네모, 세모가 골고루 섞여 있다고 한다면 각 레고 블록을 분류하고자 할 때 가장 첫 번째로 만들어져야 하는 규칙 조건은?

if 색깔 == '노란색'.

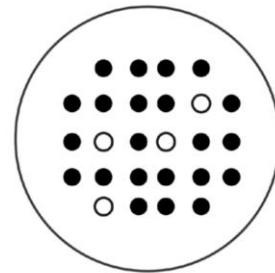
정보 균일도 측정 방법

데이터 혼잡도의 예

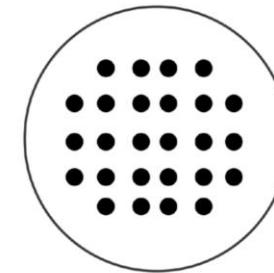
데이터 세트 A



데이터 세트 B



데이터 세트 C



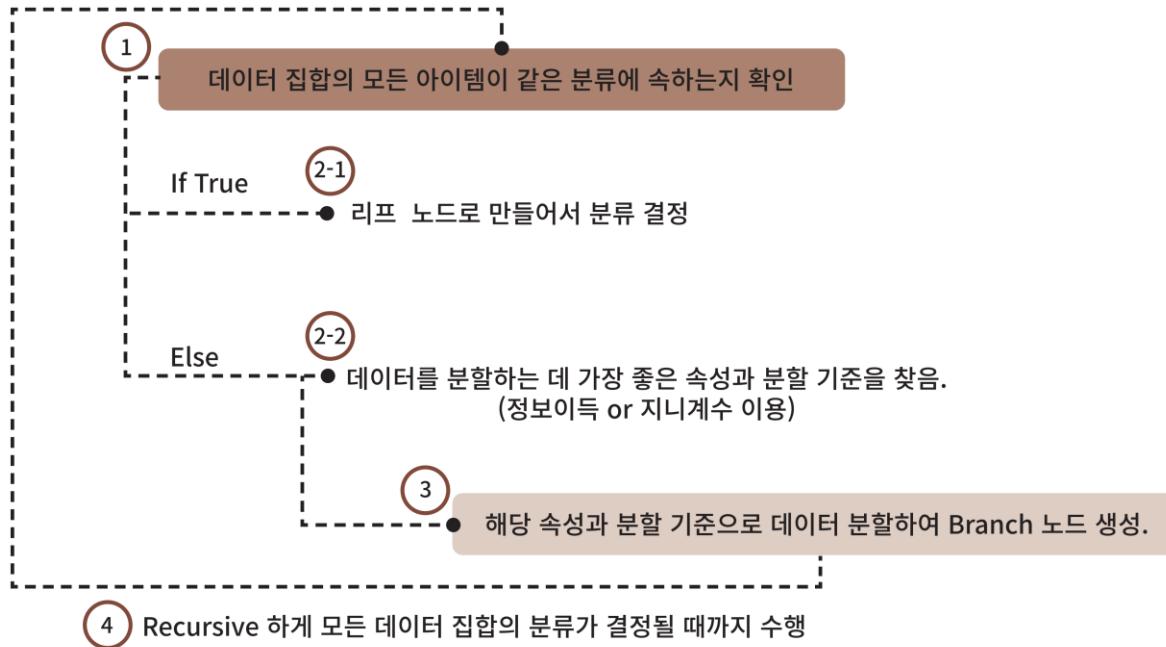
정보 이득(Information Gain)

정보 이득은 엔트로피라는 개념을 기반으로 합니다. 엔트로피는 주어진 데이터 집합의 혼잡도를 의미하는데, 서로 다른 값이 섞여 있으면 엔트로피가 높고, 같은 값이 섞여 있으면 엔트로피가 낮습니다. 정보 이득 지수는 1에서 엔트로피 지수를 뺀 값입니다. 즉, 1-엔트로피 지수입니다. 결정 트리는 이 정보 이득 지수로 분할 기준을 정합니다. 즉, 정보 이득이 높은 속성을 기준으로 분할합니다.

지니 계수

지니 계수는 원래 경제학에서 불평등 지수를 나타낼 때 사용하는 계수입니다. 경제학자인 코라도 지니(Corrado Gini)의 이름에서 따 계수로서 0이 가장 평등하고 1로 갈수록 불평등합니다. 머신러닝에 적용될 때는 의미론적으로 재해석돼 데이터가 다양한 값을 가질수록 평등하며 특정 값으로 쓸릴 경우에는 불평등한 값이 됩니다. 즉, 다양성이 낮을 수록 균일도가 높다는 의미로서, 1로 갈수록 균일도가 높으므로 지니 계수가 높은 속성을 기준으로 분할하는 것입니다.

Decision Tree



- 기본적으로 지니 계수를 이용해 데이터 세트를 분할
- 정보이득이 높거나 지니 계수가 낮은 조건을 찾아서 자시트리 노드에 분할
- 데이터가 모두 특정 분류에 속하게 되면 분할을 멈추고 분류 결정

Characteristics

- “균일도” 직관적이고 쉽다.
- 트리의 크기를 사전에 제한하는 것이 성능 튜닝에 효과적
 - 모든 데이터를 만족하는 완벽한 규칙은 만들수 없기에

결정 트리 장점	결정 트리 단점
<ul style="list-style-type: none">• 쉽다. 직관적이다• 피처의 스케일링이나 정규화 등의 사전 가공 영향도가 크지 않음.	<ul style="list-style-type: none">• 과적합으로 알고리즘 성능이 떨어진다. 이를 극복하기 위해 트리의 크기를 사전에 제한하는 튜닝 필요.

Decision Tree Parameter

파라미터 명	설명
max_depth	<ul style="list-style-type: none">트리의 최대 깊이를 규정.디폴트는 <code>None</code>. <code>None</code>으로 설정하면 완벽하게 클래스 결정 값이 될 때까지 깊이를 계속 키우며 분할하거나 노드가 가지는 데이터 개수가 <code>min_samples_split</code>보다 작아질 때까지 계속 깊이를 증가시킴.깊이가 깊어지면 <code>min_samples_split</code> 설정대로 최대 분할하여 과적합할 수 있으므로 적절한 값으로 제어 필요.
max_features	<ul style="list-style-type: none">최적의 분할을 위해 고려할 최대 피처 개수. 디폴트는 <code>None</code>으로 데이터 세트의 모든 피처를 사용해 분할 수행.<code>int</code> 형으로 지정하면 대상 피처의 개수, <code>float</code> 형으로 지정하면 전체 피처 중 대상 피처의 퍼센트임'<code>sqrt</code>'는 전체 피처 중 $\sqrt{\text{전체 피처 개수}}$ 만큼 선정'<code>auto</code>'로 지정하면 <code>sqrt</code>와 동일'<code>log</code>'는 전체 피처 중 $\log_2(\text{전체 피처 개수})$ 선정'<code>None</code>'은 전체 피처 선정
min_samples_split	<ul style="list-style-type: none">노드를 분할하기 위한 최소한의 샘플 데이터 수로 과적합을 제어하는 데 사용됨.디폴트는 2이고 작게 설정할수록 분할되는 노드가 많아져서 과적합 가능성 증가과적합을 제어. 1로 설정할 경우 분할되는 노드가 많아져서 과적합 가능성 증가
min_samples_leaf	<ul style="list-style-type: none">말단 노드(Leaf)가 되기 위한 최소한의 샘플 데이터 수<code>Min_samples_split</code>과 유사하게 과적합 제어 용도. 그러나 비대칭적(imbalanced) 데이터의 경우 특정 클래스의 데이터가 극도로 작을 수 있으므로 이 경우는 작게 설정 필요.
max_leaf_nodes	<ul style="list-style-type: none">말단 노드(Leaf)의 최대 개수

Decision Tree Visualization

- Graphviz 설치 (www.graphviz.org)
-C/C++ package
- Graphviz 설치후 파이썬과 인터페이스 가능한 wrapper 모듈 설치
- 윈도우 버전의 Graphviz 설치
<https://www2.graphviz.org/Packages/stable/windows/10/cmake/Release/Win32/>

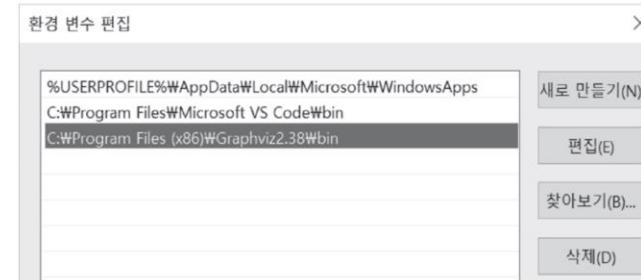
```
(base) C:\Windows\system32>pip install graphviz
Collecting graphviz
  Downloading https://files.pythonhosted.org/packages/1f/e2/ef2581b5b86625657af32030f90cf2717456c1d2b711ba074bf007c0f1a
/graphviz-0.10.1-py2.py3-none-any.whl
Installing collected packages: graphviz
Successfully installed graphviz-0.10.1
```

Decision Tree Visualization

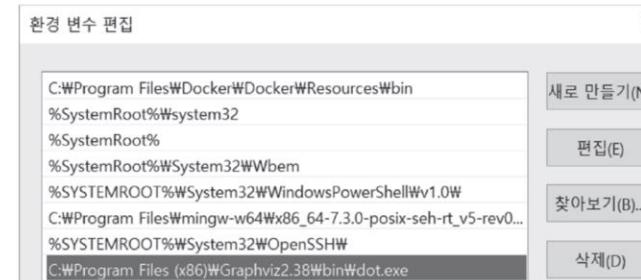
환경 Path 설정



사용자 변수
Path 값 설정



시스템 변수
Path 값 설정



Tree 생성

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')

# DecisionTree Classifier 생성
dt_clf = DecisionTreeClassifier(random_state=156)

# 붂꽃 데이터를 로딩하고, 학습과 테스트 데이터 셋으로 분리
iris_data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_data.target,
                                                    test_size=0.2, random_state=11)

# DecisionTreeClassifier 학습.
dt_clf.fit(X_train, y_train)

DecisionTreeClassifier(random_state=156)
```

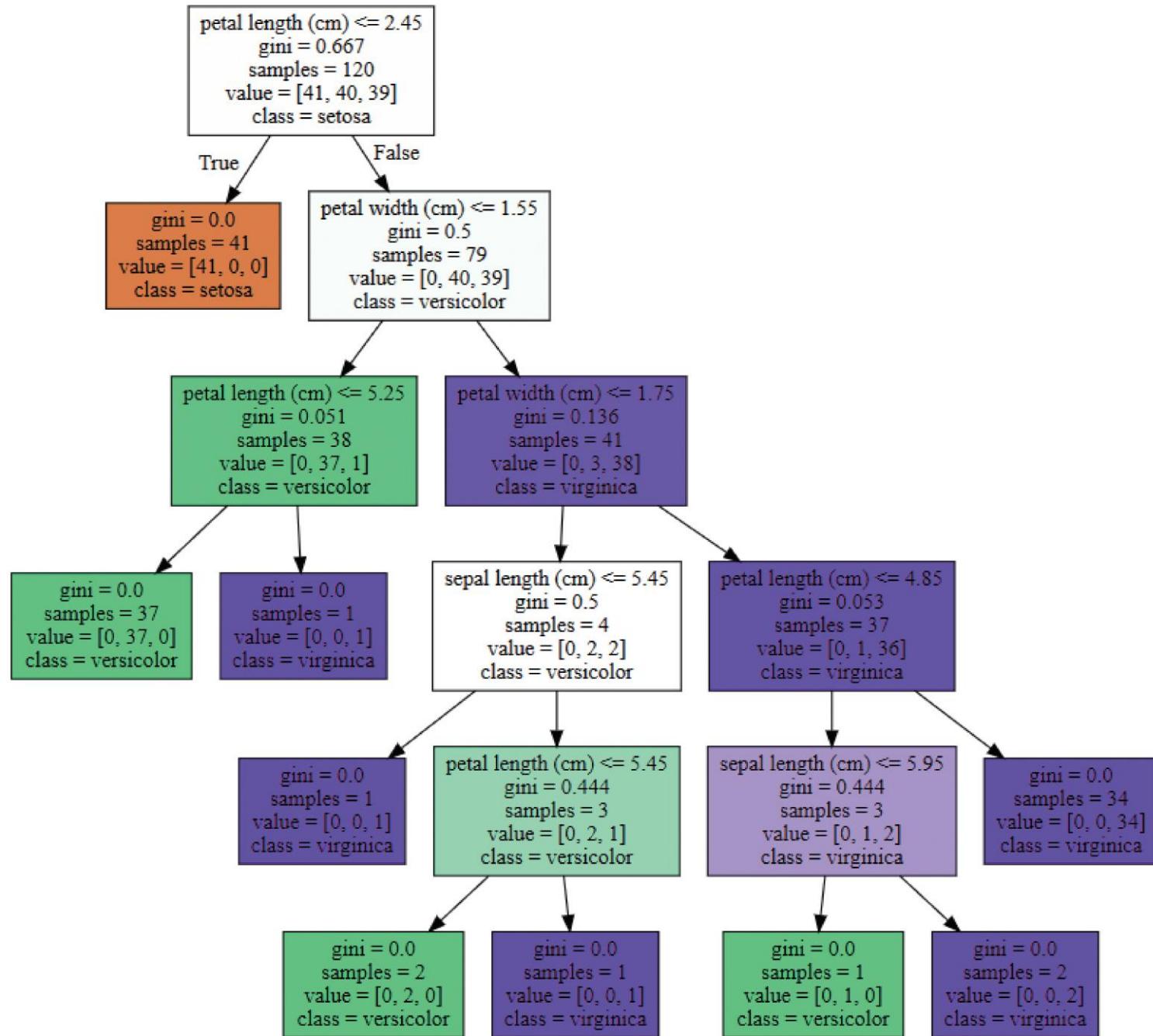
```
from sklearn.tree import export_graphviz

# export_graphviz()의 호출 결과로 out_file로 지정된 tree.dot 파일을 생성함.
export_graphviz(dt_clf, out_file="tree.dot", class_names=iris_data.target_names, #
feature_names = iris_data.feature_names, impurity=True, filled=True)
```

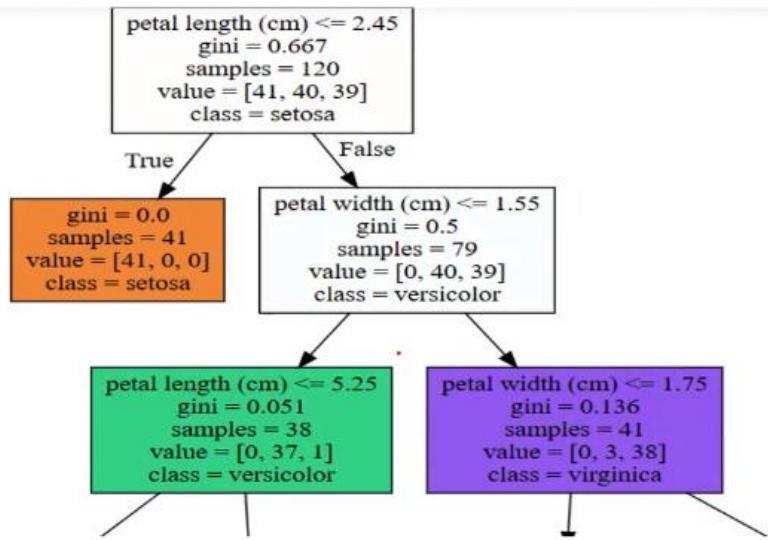
```
import graphviz

# 위에서 생성된 tree.dot 파일을 Graphviz 읽어서 Jupyter Notebook상에서 시각화
with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

Tree

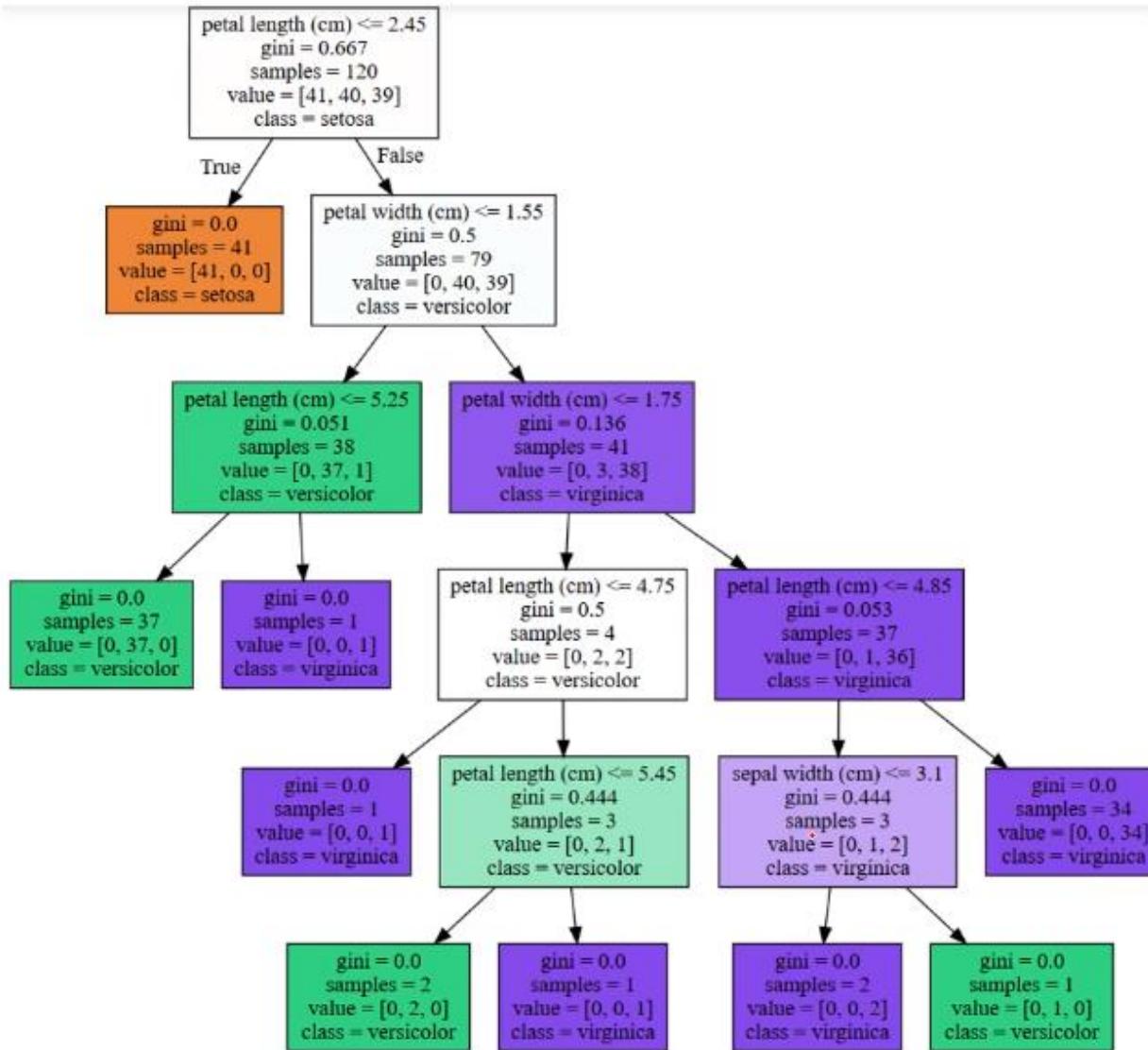


Graphviz의 시각화 노드



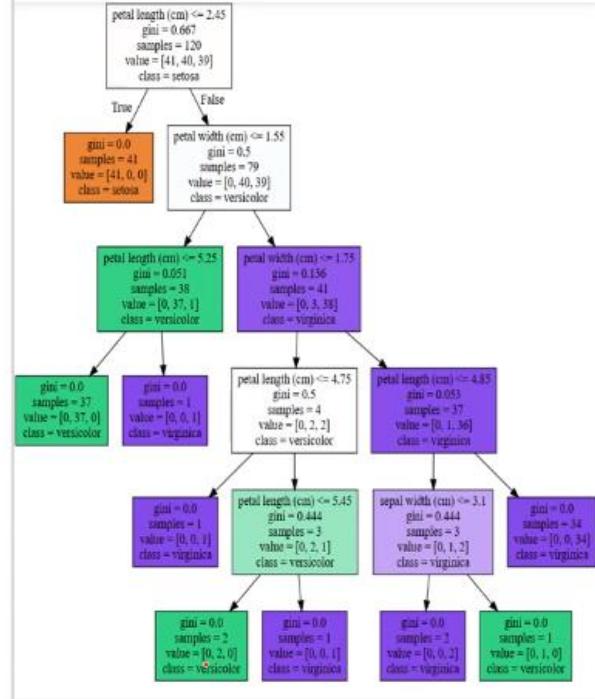
- petal length(cm) <= 2.45와 같이 피처의 조건이 있는 것은 자식 노드를 만들기 위한 규칙 조건입니다. 이 조건이 없으면 리프 노드입니다.
- gini는 다음의 value=[]로 주어진 데이터 분포에서의 지니 계수입니다.
- samples는 현 규칙에 해당하는 데이터 건수입니다.
- value = []는 클래스 값 기반의 데이터 건수입니다. 붓꽃 데이터 세트는 클래스 값으로 0, 1, 2를 가지고 있으며, 0 : Setosa, 1: Versicolor, 2: Virginica 품종을 가리킵니다. 만일 Value = [41, 40, 39]라면 클래스 값의 순서로 Setosa 41개, Versicolor 40개, Virginica 39개로 데이터가 구성돼 있다는 의미입니다.
- class는 value 리스트내에 가장 많은 건수를 가진 결정값입니다.

Graphviz를 이용한 결정 트리 모델의 시각화

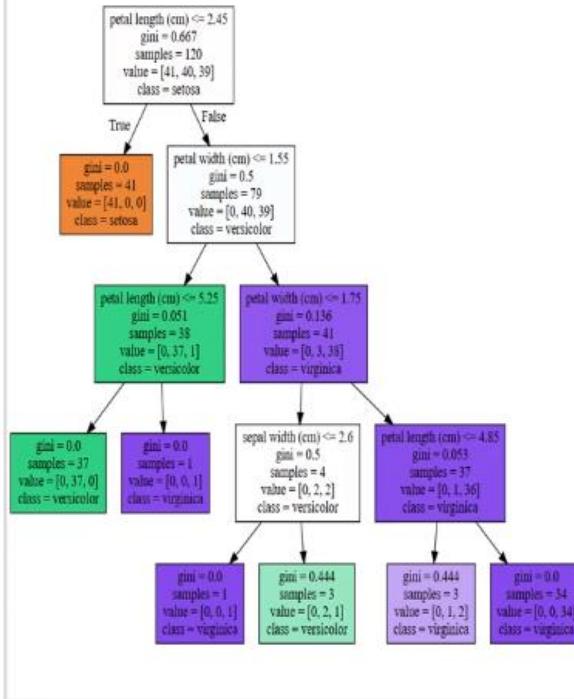


Max_depth에 따른 결정 트리 구조

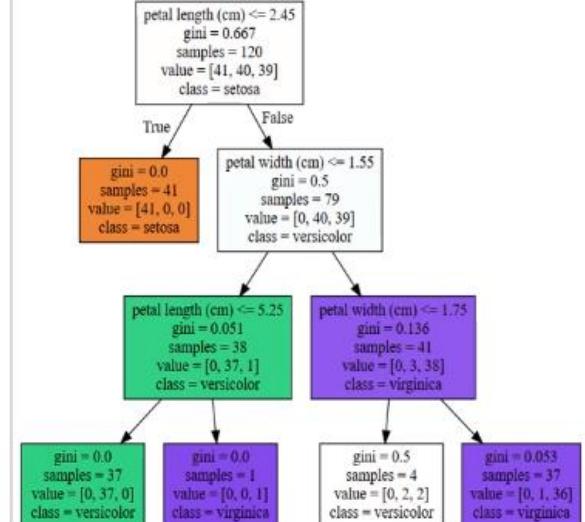
max_depth 제약 없음



max_depth = 4

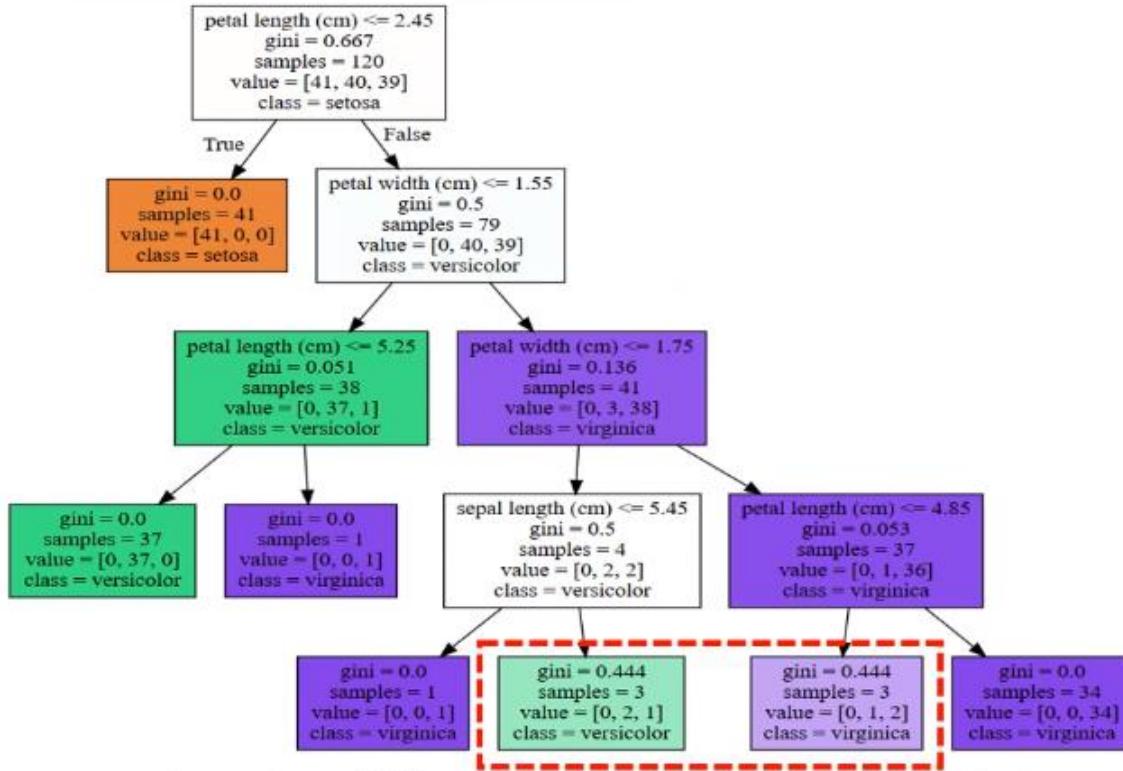


max_depth = 3



Min_samples_spilt에 따른 결정 트리 구조

min_samples_split = 4, 정확도 0.933

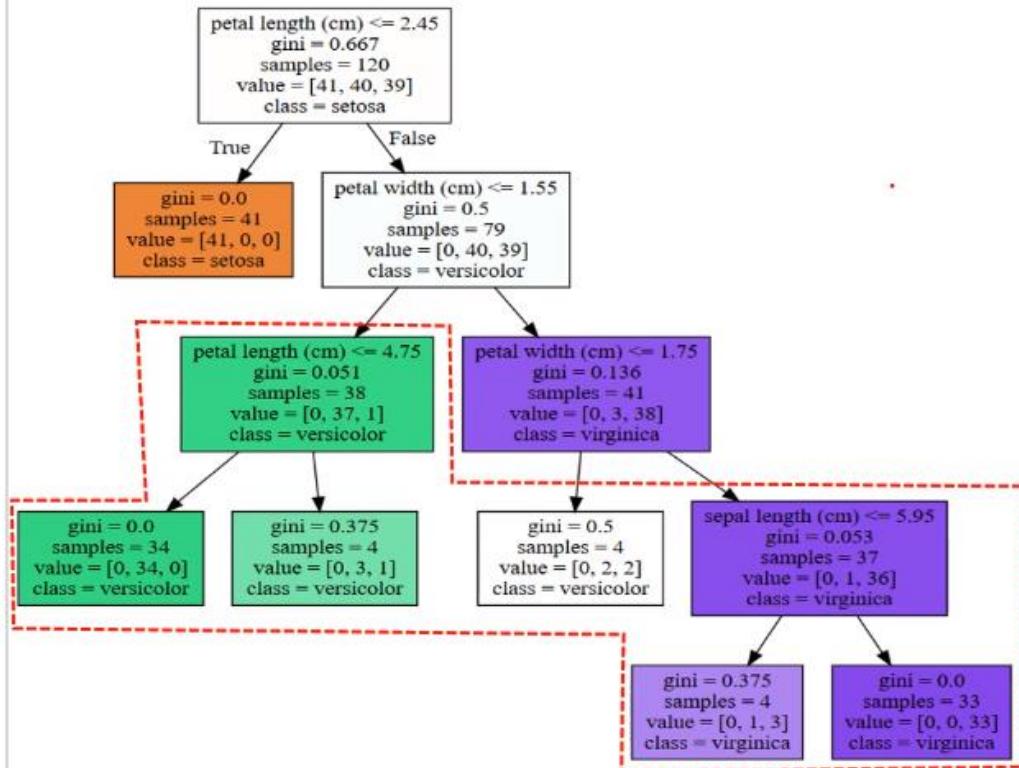


min_samples_split=4 인데 Samples가 3개 이므로 서로 다른 Class

값이 있어도 Split 하지 않습니다.

Min_samples_leaf에 따른 결정 트리 구조

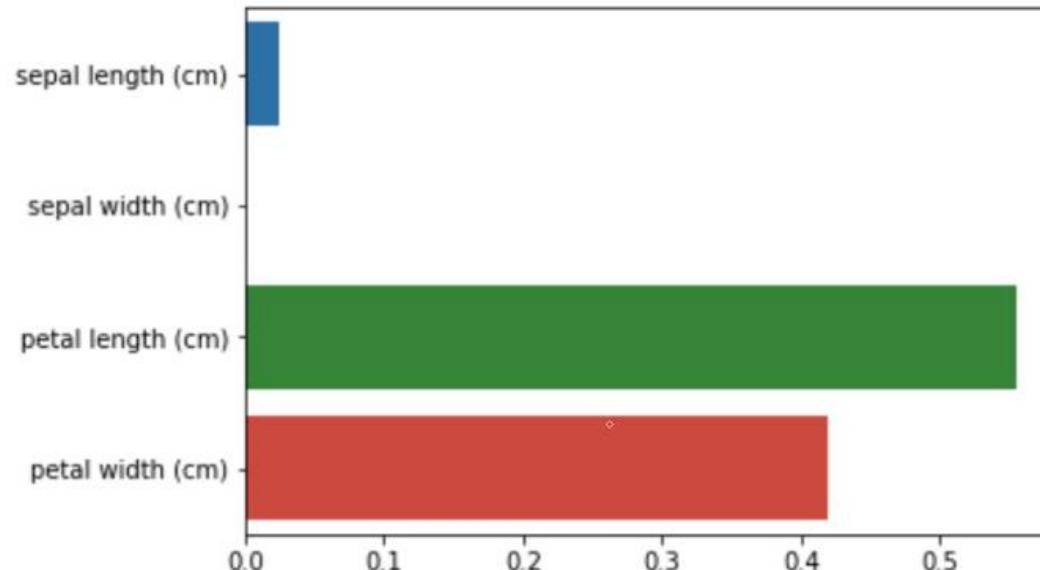
min_samples_leaf = 4, 정확도 0.933



Sample 0이 4 이상인 노드는 리프 클래스 노드가 될 수 있으므로 규칙이 sample 4 인 노드를 만들수 있는 상황을 반영하여 변경됩니다.

결정 트리의 Feature 선택 중요도

사이킷런의 DecisionClassifier 객체는 `feature_importances_` 을 통해 학습/예측을 위해서 중요한 Feature들을 선택할 수 있게 정보를 제공합니다.



결정 트리의 Feature 선택 중요도

```
import seaborn as sns
import numpy as np
%matplotlib inline

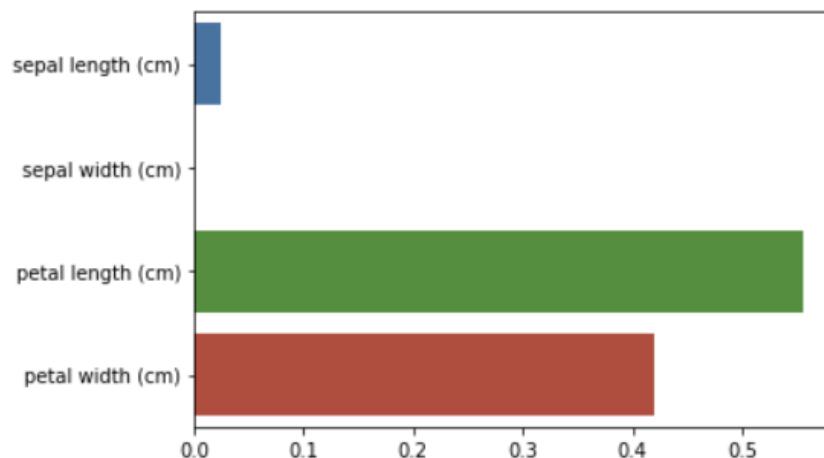
# feature importance 추출
print("Feature importances: \n{0}".format(np.round(dt_clf.feature_importances_, 3)))

# feature별 importance 출력
for name, value in zip(iris_data.feature_names, dt_clf.feature_importances_):
    print('{0} : {1:.3f}'.format(name, value))

# feature importance를 column 별로 시각화 하기
sns.barplot(x=dt_clf.feature_importances_, y=iris_data.feature_names)
```

Feature importances:
[0.025 0.555 0.42]
sepal length (cm) : 0.025
sepal width (cm) : 0.000
petal length (cm) : 0.555
petal width (cm) : 0.420

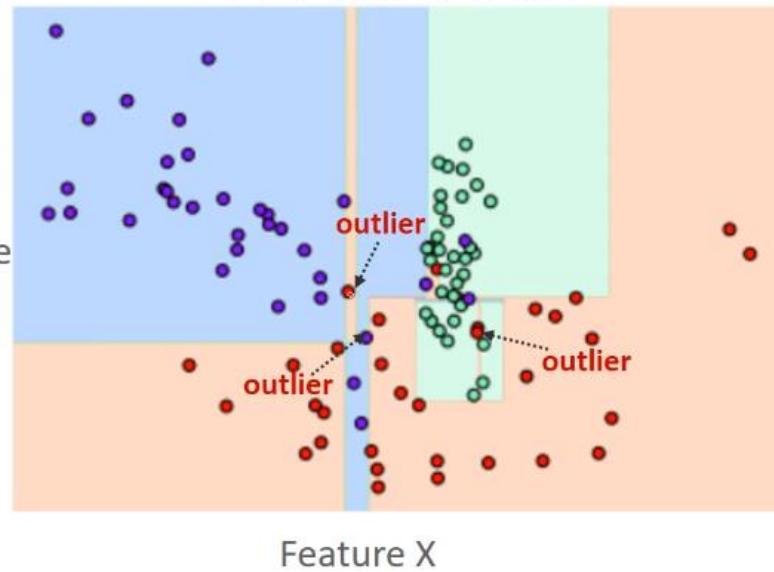
<matplotlib.axes._subplots.AxesSubplot at 0x1f009c343a0>



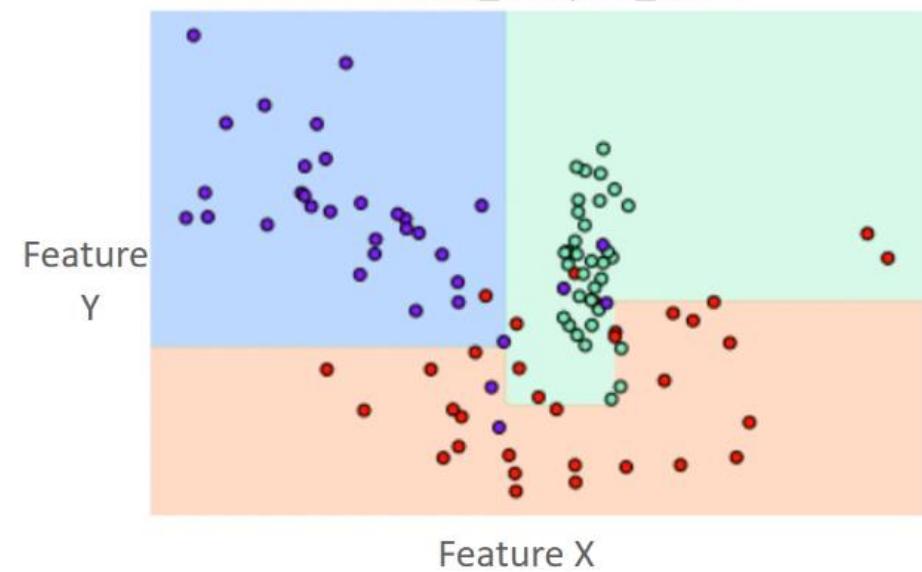
결정 트리 과적합 (Overfitting)

2개의 Feature로 된 3개의 결정 클래스를 가지도록 `make_classification()` 함수를 이용하여 임의 데이터를 생성한 후 트리 생성 제약이 없는 경우와 `min_samples_leaf=6`으로 제약을 주었을 때 분류 기준선의 변화

트리 생성 제약 없음



`min_samples_leaf=6`



결정 트리 과적합 (Overfitting)

- Syntheticdata : make_classification()함수 제공

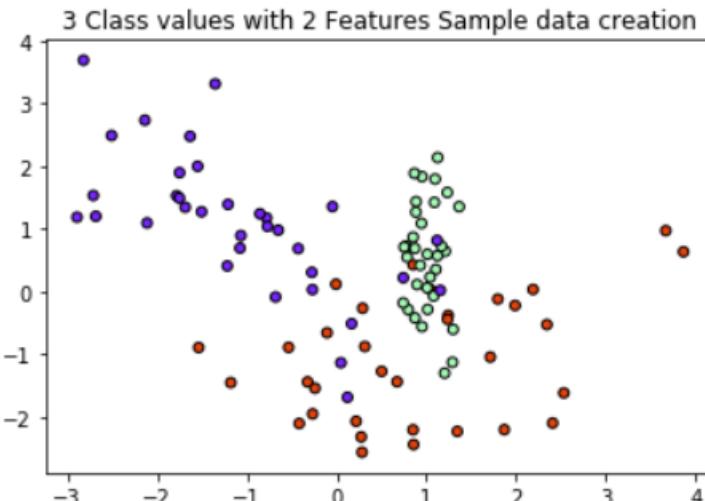
```
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
%matplotlib inline

plt.title("3 Class values with 2 Features Sample data creation")

# 2차원 시각화를 위해서 feature는 2개, 결정값 클래스는 3가지 유형의 classification 샘플 데이터 생성.
X_features, y_labels = make_classification(n_features=2, n_redundant=0, n_informative=2,
                                            n_classes=3, n_clusters_per_class=1, random_state=0)

# plot 형태로 2개의 feature로 2차원 좌표 시각화, 각 클래스값은 다른 색깔로 표시됨.
plt.scatter(X_features[:, 0], X_features[:, 1], marker='o', c=y_labels, s=25, cmap='rainbow', edgecolor='k')

<matplotlib.collections.PathCollection at 0x1f364f85080>
```



visualize_boundary()생성

```
import numpy as np

# Classifier의 Decision Boundary를 시각화 하는 함수
def visualize_boundary(model, X, y):
    fig,ax = plt.subplots()

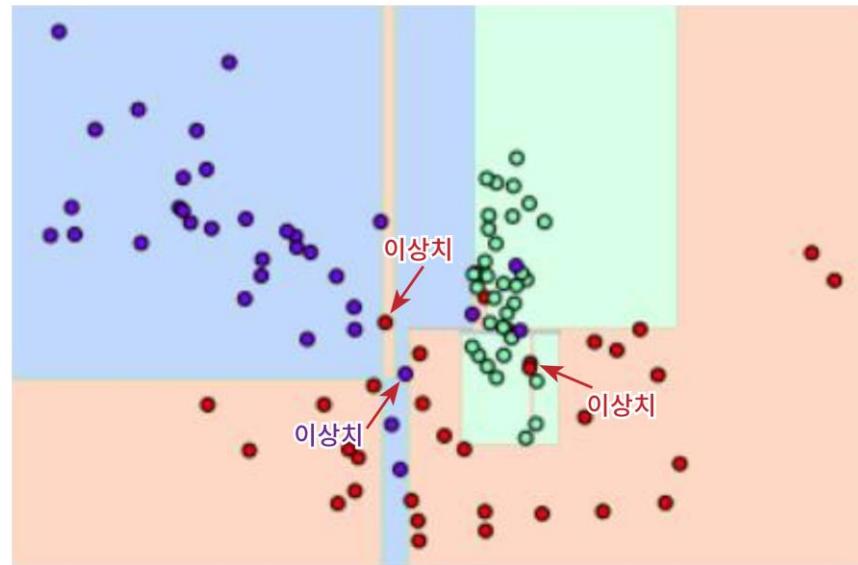
    # 학습 데이터 scatter plot으로 나타내기
    ax.scatter(X[:, 0], X[:, 1], c=y, s=25, cmap='rainbow', edgecolor='k',
               clim=(y.min(), y.max()), zorder=3)
    ax.axis('tight')
    ax.axis('off')
    xlim_start , xlim_end = ax.get_xlim()
    ylim_start , ylim_end = ax.get_ylim()

    # 호출 파라미터로 들어온 training 데이터로 model 학습 .
    model.fit(X, y)
    # meshgrid 형태의 모든 좌표값으로 예측 수행.
    xx, yy = np.meshgrid(np.linspace(xlim_start,xlim_end, num=200),np.linspace(ylim_start,ylim_end, num=200))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # contourf()를 이용하여 class boundary를 visualization 수행.
    n_classes = len(np.unique(y))
    contours = ax.contourf(xx, yy, Z, alpha=0.3,
                          levels=np.arange(n_classes + 1) - 0.5,
                          cmap='rainbow', clim=(y.min(), y.max()),
                          zorder=1)
```

visualize_boundary()생성

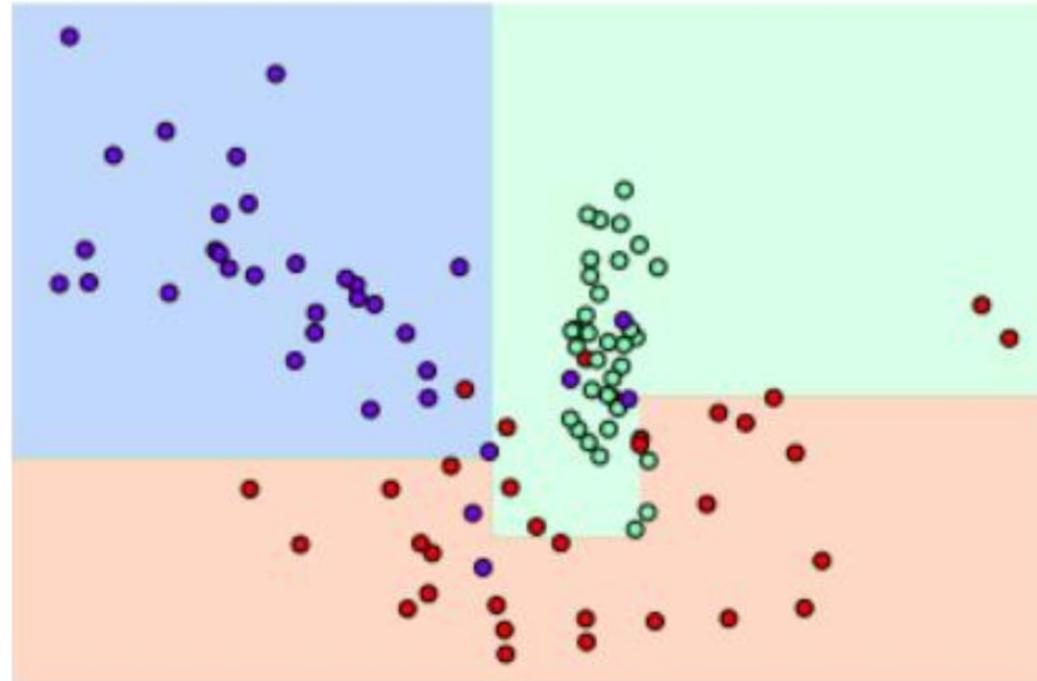
```
from sklearn.tree import DecisionTreeClassifier  
  
# 특정한 트리 생성 제약없는 결정 트리의 Decision Boundary 시각화.  
dt_clf = DecisionTreeClassifier().fit(X_features, y_labels)  
visualize_boundary(dt_clf, X_features, y_labels)
```



- Hyperparameter -defaulted

visualize_boundary()생성

```
# min_samples_leaf=6 으로 트리 생성 조건을 제약한 Decision Boundary 시각화  
dt_clf = DecisionTreeClassifier( min_samples_leaf=6).fit(X_features, y_labels)  
visualize_boundary(dt_clf, X_features, y_labels)
```



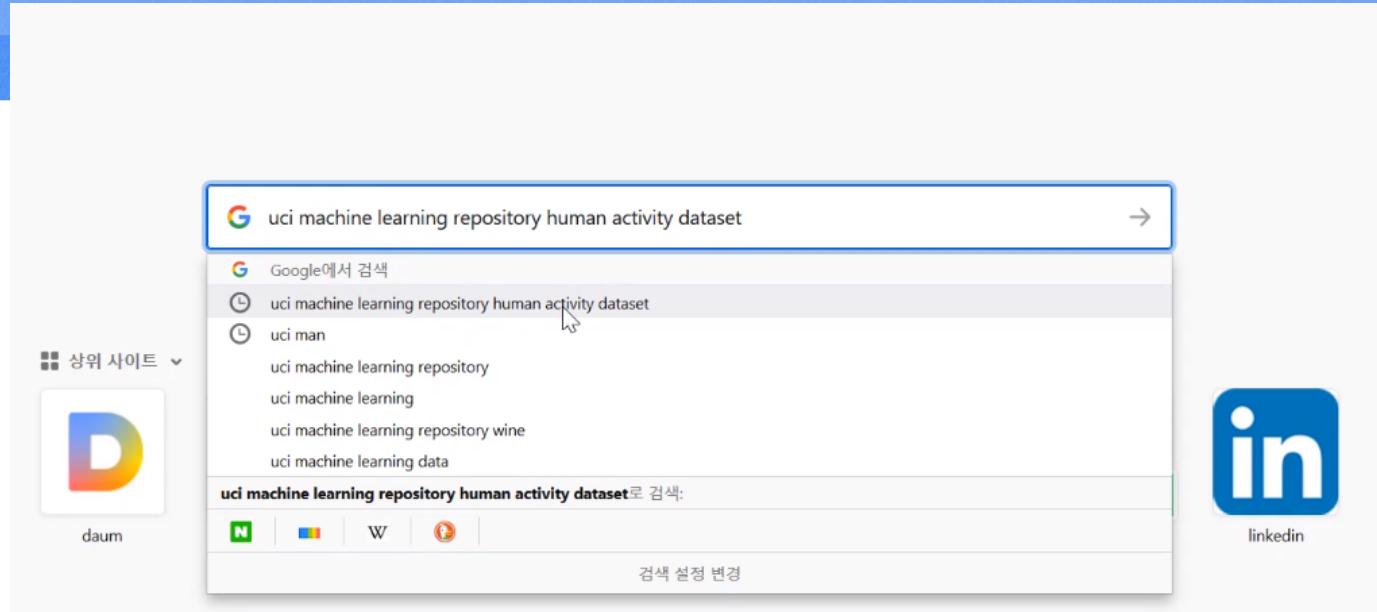
- Min_samples_leaf =6 -> 성능이 뛰어난 가능성이 높음 왜?

결정 트리 실습 -사용자 행동 인식 데이터 세트

사용자 행동 인식 데이터는 30명에게 스마트폰 센서를 장착한 뒤 사람의 동작과 관련된 여러 가지 피처를 수집한 데이터입니다. 수집된 피처 세트를 기반으로 결정 트리를 이용해 어떠한 동작인지 예측해 보겠습니다.



결정 트리 실습 - 사용자 행동 인식 데이터 세트



폴더명 변경

-> Human_activity

CSV 파일이 아니라 Text파일에 공백 이용

결정 트리 실습 – 사용자 행동 인식 데이터 세트

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

# features.txt 파일에는 피처 이름 index와 피처명이 공백으로 분리되어 있음. 이를 DataFrame으로 로드.
feature_name_df = pd.read_csv('./human_activity/features.txt', sep=' ', header=None, names=['column_index', 'column_name'])

# 피처명 index를 제거하고, 피처명만 리스트 객체로 생성한 뒤 샘플로 10개만 추출
feature_name = feature_name_df.iloc[:, 1].values.tolist()
print('전체 피처명에서 10개만 추출:', feature_name[:10])
```

전체 피처명에서 10개만 추출: ['tBodyAcc-mean()-X', 'tBodyAcc-mean()-Y', 'tBodyAcc-mean()-Z', 'tBodyAcc-std()-X', 'tBodyAcc-std()-Y', 'tBodyAcc-std()-Z', 'tBodyAcc-mad()-X', 'tBodyAcc-mad()-Y', 'tBodyAcc-mad()-Z', 'tBodyAcc-max()-X']

feature_name_df.head(20)



column_index	column_name
0	tBodyAcc-mean()-X
1	tBodyAcc-mean()-Y
2	tBodyAcc-mean()-Z
3	tBodyAcc-std()-X
4	tBodyAcc-std()-Y
5	tBodyAcc-std()-Z
6	tBodyAcc-mad()-X
7	tBodyAcc-mad()-Y
8	tBodyAcc-mad()-Z
9	tBodyAcc-max()-X
10	tBodyAcc-max()-Y
11	tBodyAcc-max()-Z
12	tBodyAcc-min()-X
13	tBodyAcc-min()-Y
14	tBodyAcc-min()-Z
15	tBodyAcc-sma()
16	tBodyAcc-energy()-X
17	tBodyAcc-energy()-Y
18	tBodyAcc-energy()-Z
19	tBodyAcc-iqr()-X

결정 트리 실습 - 사용자 행동 인식 데이터 세트

```
feature_dup_df = feature_name_df.groupby('column_name').count()
print(feature_dup_df[feature_dup_df['column_index'] > 1].count())
feature_dup_df[feature_dup_df['column_index'] > 1].head()
```

```
column_index    42
dtype: int64
```

중복된 피처명을 확인

column_name	column_index
fBodyAcc-bandsEnergy()-1,16	3
fBodyAcc-bandsEnergy()-1,24	3
fBodyAcc-bandsEnergy()-1,8	3
fBodyAcc-bandsEnergy()-17,24	3
fBodyAcc-bandsEnergy()-17,32	3

원본 데이터에 중복된 Feature 명으로 인하여 신규 버전의 Pandas에서 Duplicate name 에러를 발생.
중복 feature명에 대해서 원본 feature 명에 '_1(또는2)'를 추가로 부여하는 함수인 `get_new_feature_name_df()` 생성

```
def get_new_feature_name_df(old_feature_name_df):
    feature_dup_df = pd.DataFrame(data=old_feature_name_df.groupby('column_name').cumcount(),
                                    columns=['dup_cnt'])
    feature_dup_df = feature_dup_df.reset_index()
    new_feature_name_df = pd.merge(old_feature_name_df.reset_index(), feature_dup_df, how='outer')
    new_feature_name_df['column_name'] = new_feature_name_df[['column_name', 'dup_cnt']].apply(lambda x : x[0]+'_'+str(x[1])
                                                                 if x[1] >0 else x[0], axis=1)
    new_feature_name_df = new_feature_name_df.drop(['index'], axis=1)
    return new_feature_name_df
```

결정 트리 실습 - 사용자 행동 인식 데이터 세트

```
import pandas as pd

def get_human_dataset( ):

    # 각 데이터 파일들은 공백으로 분리되어 있으므로 read_csv에서 공백 문자를 sep으로 할당.
    feature_name_df = pd.read_csv('./human_activity/features.txt', sep='\s+', header=None, names=['column_index', 'column_name'])

    # 중복된 피처명을 수정하는 get_new_feature_name_df()를 이용, 신규 피처명 DataFrame 생성.
    new_feature_name_df = get_new_feature_name_df(feature_name_df)

    # DataFrame에 피처명을 컬럼으로 부여하기 위해 리스트 객체로 다시 변환
    feature_name = new_feature_name_df.iloc[:, 1].values.tolist()

    # 학습 피처 데이터 셋과 테스트 피처 데이터를 DataFrame으로 로딩. 컬럼명은 feature_name 적용
    X_train = pd.read_csv('./human_activity/train/X_train.txt', sep='\s+', names=feature_name)
    X_test = pd.read_csv('./human_activity/test/X_test.txt', sep='\s+', names=feature_name)

    # 학습 레이블과 테스트 레이블 데이터를 DataFrame으로 로딩하고 컬럼명은 action으로 부여
    y_train = pd.read_csv('./human_activity/train/y_train.txt', sep='\s+', header=None, names=['action'])
    y_test = pd.read_csv('./human_activity/test/y_test.txt', sep='\s+', header=None, names=['action'])

    # 로드된 학습/테스트용 DataFrame을 모두 반환
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = get_human_dataset()
```

데이터 사이 간격 공백으로 구분

결정 트리 실습 -사용자 행동 인식 데이터 세트

```
print('## 학습 피처 데이터셋 info()')
print(X_train.info())
```

```
## 학습 피처 데이터셋 info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7352 entries, 0 to 7351
Columns: 561 entries, tBodyAcc-mean()-X to angle(Z,gravityMean)
dtypes: float64(561)
memory usage: 31.5 MB
None
```

```
print(y_train['action'].value_counts())
```

```
6    1407
5    1374
4    1286
1    1226
2    1073
3     986
Name: action, dtype: int64
```

```
X_train.isna().sum().sum()
```

Null값 확인

0

결정 트리 실습 – 사용자 행동 인식 데이터 세트

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# 예제 반복 시마다 동일한 예측 결과 도출을 위해 random_state 설정
dt_clf = DecisionTreeClassifier(random_state=156)
dt_clf.fit(X_train, y_train)
pred = dt_clf.predict(X_test)
accuracy = accuracy_score(y_test, pred)
print('결정 트리 예측 정확도: {:.4f}'.format(accuracy))

# DecisionTreeClassifier의 하이퍼 파라미터 추출
print('DecisionTreeClassifier 기본 하이퍼 파라미터: ', dt_clf.get_params())
```

결정 트리 예측 정확도: 0.8548

DecisionTreeClassifier 기본 하이퍼 파라미터:

```
{'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features': None, 'max_leaf_nodes': None, 'min_impurity_decrease': 0.0, 'min_impurity_split': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'presort': 'deprecated', 'random_state': 156, 'splitter': 'best'}
```

```
from sklearn.model_selection import GridSearchCV

params = {
    'max_depth' : [ 6, 8, 10, 12, 16, 20, 24]
}

grid_cv = GridSearchCV(dt_clf, param_grid=params, scoring='accuracy', cv=5, verbose=1)
grid_cv.fit(X_train, y_train)
print('GridSearchCV 최고 평균 정확도 수치: {:.4f}'.format(grid_cv.best_score_))
print('GridSearchCV 최적 하이퍼 파라미터: ', grid_cv.best_params_)
```

Fitting 5 folds for each of 7 candidates, totalling 35 fits

```
GridSearchCV 최고 평균 정확도 수치: 0.8513
GridSearchCV 최적 하이퍼 파라미터: {'max_depth': 16}
```

결정 트리 실습 – 사용자 행동 인식 데이터 세트

```
# GridSearchCV 객체의 cv_results_ 속성을 DataFrame으로 생성.
cv_results_df = pd.DataFrame(grid_cv.cv_results_)

# max_depth 파라미터 값과 그때의 테스트(Evaluation)셋, 학습 데이터 셋의 정확도 수치 추출
cv_results_df[['param_max_depth', 'mean_test_score']]
```

	param_max_depth	mean_test_score
0	6	0.850791
1	8	0.851069
2	10	0.851209
3	12	0.844135
4	16	0.851344
5	20	0.850800
6	24	0.849440

```
max_depths = [6, 8, 10, 12, 16, 20, 24]
# max_depth 값을 변화 시키면서 그때마다 학습과 테스트 셋에서의 예측 성능 측정
for depth in max_depths:
    dt_clf = DecisionTreeClassifier(max_depth=depth, random_state=156)
    dt_clf.fit(X_train, y_train)
    pred = dt_clf.predict(X_test)
    accuracy = accuracy_score(y_test, pred)
    print('max_depth = {} 정확도: {:.4f}'.format(depth, accuracy))
```

```
max_depth = 6 정확도: 0.8558
max_depth = 8 정확도: 0.8707
max_depth = 10 정확도: 0.8673
max_depth = 12 정확도: 0.8646
max_depth = 16 정확도: 0.8575
max_depth = 20 정확도: 0.8548
max_depth = 24 정확도: 0.8548
```

결정 트리 실습 – 사용자 행동 인식 데이터 세트

```
params = {  
    'max_depth' : [ 8 , 12, 16 ,20],  
    'min_samples_split' : [16,24],  
}  
  
grid_cv = GridSearchCV(dt_clf, param_grid=params, scoring='accuracy', cv=5, verbose=1 )  
grid_cv.fit(X_train, y_train)  
print('GridSearchCV 최고 평균 정확도 수치: {:.4f}'.format(grid_cv.best_score_))  
print('GridSearchCV 최적 하이퍼 파라미터:', grid_cv.best_params_)
```

Fitting 5 folds for each of 8 candidates, totalling 40 fits

```
GridSearchCV 최고 평균 정확도 수치: 0.8550  
GridSearchCV 최적 하이퍼 파라미터: {'max_depth': 8, 'min_samples_split': 16}
```

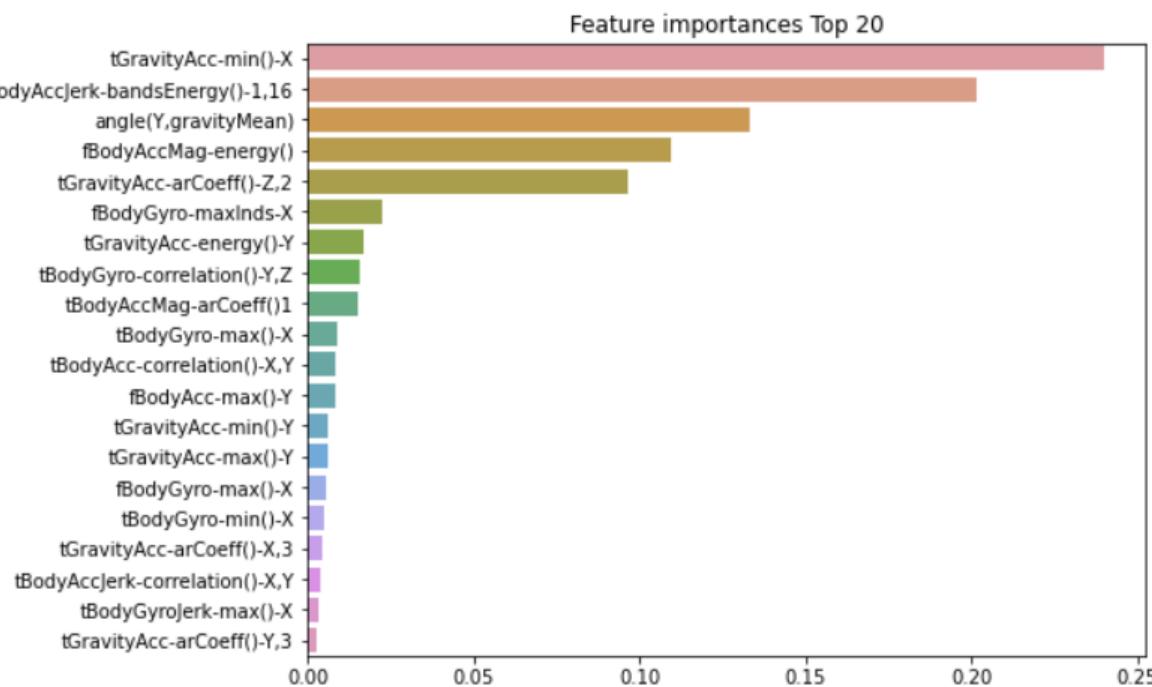
```
best_dt_clf = grid_cv.best_estimator_  
pred1 = best_dt_clf.predict(X_test)  
accuracy = accuracy_score(y_test , pred1)  
print('결정 트리 예측 정확도:{0:.4f}'.format(accuracy))
```

결정 트리 예측 정확도:0.8575

결정 트리 실습 – 사용자 행동 인식 데이터 세트

```
import seaborn as sns

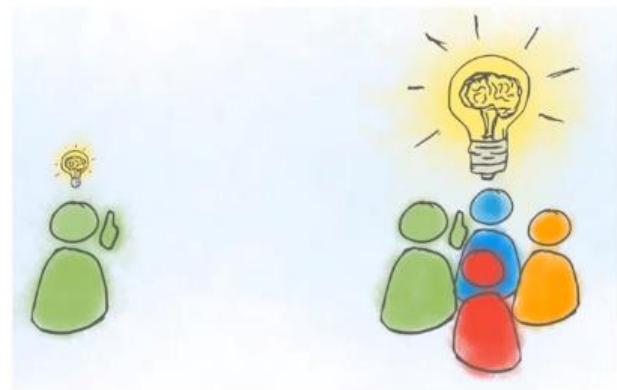
ftr_importances_values = best_df_clf.feature_importances_
# Top 중요도로 정렬을 쉽게 하고, 시본(Seaborn)의 막대그래프로 쉽게 표현하기 위해 Series변환
ftr_importances = pd.Series(ftr_importances_values, index=X_train.columns )
# 중요도값 순으로 Series를 정렬
ftr_top20 = ftr_importances.sort_values(ascending=False)[:20]
plt.figure(figsize=(8,6))
plt.title('Feature importances Top 20')
sns.barplot(x=ftr_top20 , y = ftr_top20.index)
plt.show()
```



Ensemble Learning

앙상블 학습(Ensemble Learning)을 통한 분류는 여러 개의 분류기(Classifier)를 생성하고 그 예측을 결합함으로써 보다 정확한 최종 예측을 도출하는 기법을 말합니다.

어려운 문제의 결론을 내기 위해 여러 명의 전문가로 위원회를 구성해 다양한 의견을 수렴하고 결정하듯이 앙상블 학습의 목표는 다양한 분류기의 예측 결과를 결합함으로써 단일 분류기보다 신뢰성이 높은 예측값을 얻는 것입니다.



Ensemble Learning-유형

- 양상블의 유형은 일반적으로는 보팅(Voting), 배깅(Bagging), 부스팅(Boosting)으로 구분할 수 있으며, 이외에 스태킹(Stacking)등의 기법이 있습니다.
- 대표적인 배깅은 랜덤 포레스트(Random Forest)알고리즘이 있으며, 부스팅은 에이다 부스팅, 그래디언트 부스팅, XGBoost, LightGBM 등이 있습니다. 정형 데이터의 분류나 회귀에서는 GBM 부스팅 계열의 양상블이 전반적으로 높은 예측 성능을 나타냅니다.
- 넓은 의미로는 서로 다른 모델을 결합한 것들을 양상블로 지칭하기도 합니다.

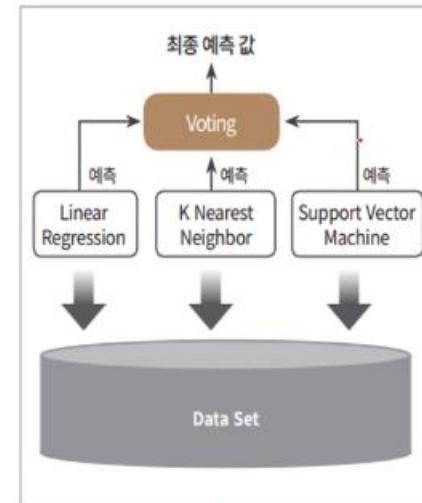
앙상블의 특징

- 단일 모델의 약점을 다수의 모델들을 결합하여 보완
- 뛰어난 성능을 가진 모델들로만 구성하는 것보다 성능이 떨어지더라도 서로 다른 유형의 모델을 섞는 것이 오히려 전체 성능이 도움이 될 수 있음.
- 랜덤 포레스트 및 뛰어난 부스팅 알고리즘들은 모두 결정 트리 알고리즘을 기반 알고리즘으로 적용함.
- 결정 트리의 단점인 과적합(오버 피팅)을 수십~수천개의 많은 분류기를 결합해 보완하고 장점인 직관적인 분류 기준은 강화됨.

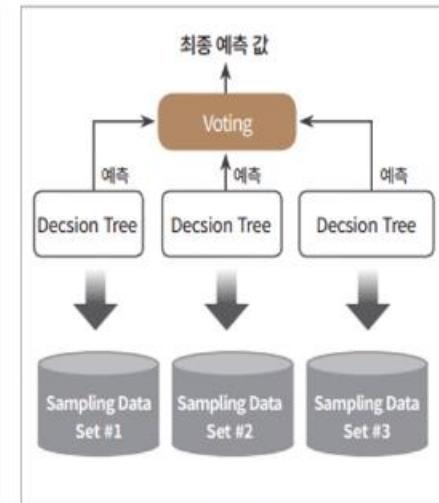


Voting vs Boosting vs Bagging

- 보팅과 배깅은 여러 개의 분류기가 투표를 통해 최종 예측 결과를 결정하는 방식입니다.
- 보팅과 배깅의 다른 점은 보팅의 경우 일반적으로 서로 다른 알고리즘을 가진 분류기를 결합하는 것이고, 배깅의 경우 각각의 분류기가 모두 같은 유형의 알고리즘 기반이지만, 데이터 샘플링을 서로 다르게 가져가면서 학습을 수행해 보팅을 수행하는 것입니다.



Voting 방식



Bagging 방식

Voting Type

Hard Voting은 다수의 classifier 간 다수결로 최종 class 결정

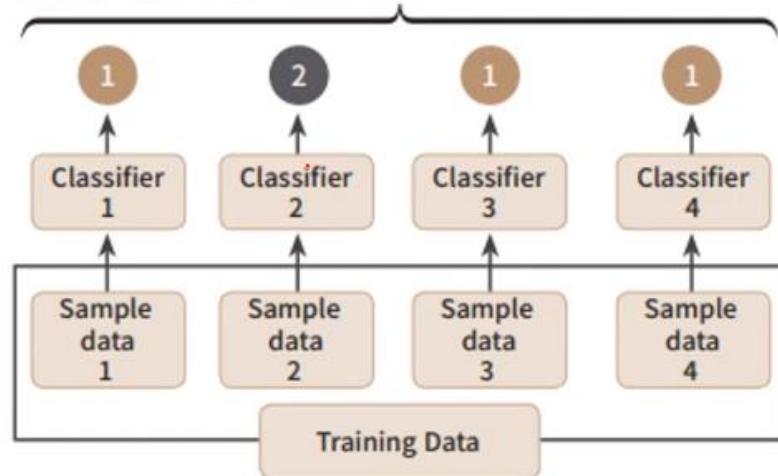
클래스 값 1로 예측

classifier 1, 3, 4는

클래스 값 1로 예측

classifier 2는 클래스 값 2로 예측

1



<하드 보팅>

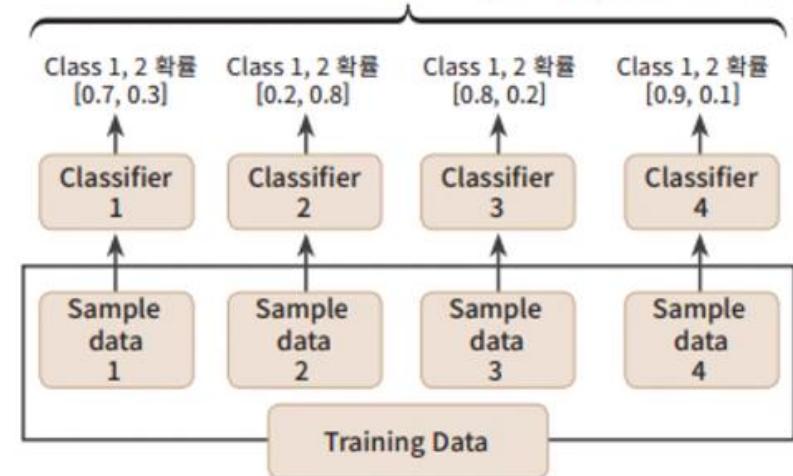
Soft Voting은 다수의 classifier 들의 class 확률을 평균하여 결정

클래스 값 1로 예측

클래스 값 1일 확률: 0.65

클래스 값 2일 확률: 0.35

1 predict_proba() 메소드를
이용하여 class 별 확률 결정



<소프트 보팅>

- 일반적으로 하드 보팅보다는 소프트 보팅이 예측 성능이 상대적으로 우수하여 주로 사용됨.
- 사이킷런은 **VotingClassifier** 클래스를 통해 보팅(Voting)을 지원

Voting Classifier

Load_breast_cancer()
위스콘신 유방암 데이터 세트

```
import pandas as pd

from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

cancer = load_breast_cancer()

data_df = pd.DataFrame(cancer.data, columns=cancer.feature_names)
data_df.head(3)
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst radius	worst texture	worst perimeter	worst area	smoc
0	17.99	10.38	122.8	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	25.38	17.33	184.6	2019.0	
1	20.57	17.77	132.9	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	24.99	23.41	158.8	1956.0	
2	19.69	21.25	130.0	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	23.57	25.53	152.5	1709.0	

Voting Classifier

Load_breast_cancer()
위스콘신 유방암 데이터 세트

```
# 개별 모델은 로지스틱 회귀와 KNN 입.
lr_clf = LogisticRegression()
knn_clf = KNeighborsClassifier(n_neighbors=8)

# 개별 모델을 소프트 보팅 기반의 앙상블 모델로 구현한 분류기
vo_clf = VotingClassifier( estimators=[('LR',lr_clf),('KNN',knn_clf)] , voting='soft' )

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    test_size=0.2 , random_state= 156)

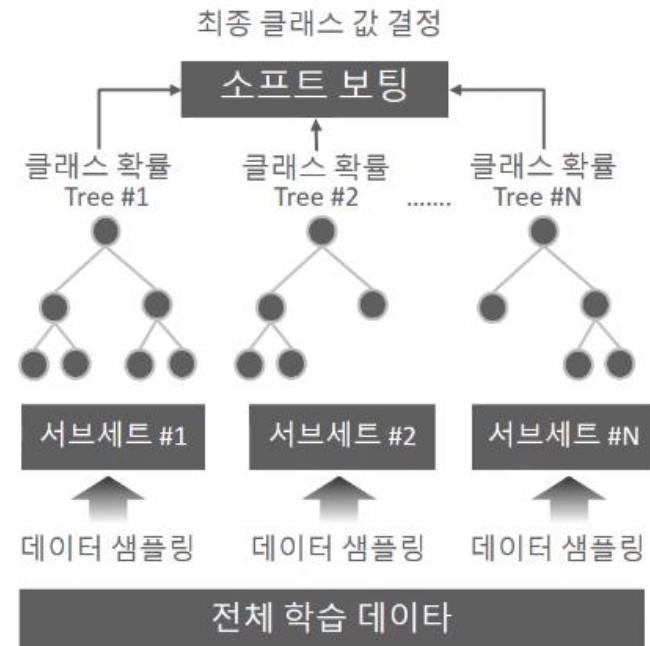
# VotingClassifier 학습/예측/평가.
vo_clf.fit(X_train , y_train)
pred = vo_clf.predict(X_test)
print('Voting 분류기 정확도: {:.4f}'.format(accuracy_score(y_test , pred)))

# 개별 모델의 학습/예측/평가.
classifiers = [lr_clf, knn_clf]
for classifier in classifiers:
    classifier.fit(X_train , y_train)
    pred = classifier.predict(X_test)
    class_name= classifier.__class__.__name__
    print('{0} 정확도: {:.4f}'.format(class_name, accuracy_score(y_test , pred)))
```

Voting 분류기 정확도: 0.9561
LogisticRegression 정확도: 0.9474
KNeighborsClassifier 정확도: 0.9386

배깅 (Bagging) – 랜덤 포레스트

- 배깅의 대표적인 알고리즘은 랜덤 포레스트입니다.
- 랜덤 포레스트는 다재 다능한 알고리즘입니다. 앙상블 알고리즘 중 비교적 빠른 수행 속도를 가지고 있으며, 다양한 영역에서 높은 예측 성능을 보이고 있습니다.
- 랜덤 포레스트는 여러 개의 결정 트리 분류기가 전체 데이터에서 배깅 방식으로 각자의 데이터를 샘플링해 개별적으로 학습을 수행한 뒤 최종적으로 모든 분류기가 보팅을 통해 예측 결정을 하게 됩니다.



Random Forest의 Bootstrapping

- 랜덤 포레스트는 개별적인 분류기의 기반 알고리즘은 결정 트리이지만 개별 트리가 학습하는 데이터 세트는 전체 데이터에서 일부가 중첩되게 샘플링된 데이터 세트입니다. 이렇게 여러 개의 데이터 세트를 중첩되게 분리하는 것을 부트스트래핑(bootstrapping) 분할 방식이라고 합니다(그래서 배깅(Bagging)이 bootstrap aggregating의 줄임말입니다).
- 원본 데이터의 건수가 10개인 학습 데이터 세트에 랜덤 포레스트를 3개의 결정 트리 기반으로 학습하려고 n_estimators=3으로 하이퍼 파라미터를 부여하면 다음과 같이 데이터 서브세트가 만들어 집니다.



Random Forest Hyper parameter

사이킷런은 랜덤 포레스트 분류를 위해 `RandomForestClassifier` 클래스를 제공합니다.

`RandomForestClassifier` 하이퍼 파라미터 .

- `n_estimators`: 랜덤 포레스트에서 결정 트리의 개수를 지정합니다. 디폴트는 10개입니다. 많이 설정할수록 좋은 성능을 기대할 수 있지만 계속 증가시킨다고 성능이 무조건 향상되는 것은 아닙니다. 또한 늘릴수록 학습 수행 시간이 오래 걸리는 것도 감안해야 합니다.
- `max_features`는 결정 트리에 사용된 `max_features` 파라미터와 같습니다. 하지만 `RandomForestClassifier`의 기본 `max_features`는 'None'이 아니라 'auto', 즉 'sqrt'와 같습니다. 따라서 랜덤 포레스트의 트리를 분할하는 피처를 참조할 때 전체 피처가 아니라 $\sqrt{\text{전체 피처 개수}}$ 만큼 참조합니다(전체 피처가 16개라면 분할을 위해 4개 참조).
- `max_depth`나 `min_samples_leaf`와 같이 결정 트리에서 과적합을 개선하기 위해 사용되는 파라미터가 랜덤 포레스트에도 똑같이 적용될 수 있습니다.

Random Forest

결정트리에서 사용한 사용자 행동인지 데이터세트 사용

```
def get_new_feature_name_df(old_feature_name_df):
    feature_dup_df = pd.DataFrame(data=old_feature_name_df.groupby('column_name').cumcount(),
                                    columns=['dup_cnt'])
    feature_dup_df = feature_dup_df.reset_index()
    new_feature_name_df = pd.merge(old_feature_name_df.reset_index(), feature_dup_df, how='outer')
    new_feature_name_df[['column_name', 'dup_cnt']].apply(lambda x : x[0] + '_' + str(x[1])
                                                          if x[1] > 0 else x[0], axis=1)
    new_feature_name_df = new_feature_name_df.drop(['index'], axis=1)
    return new_feature_name_df
```

```
import pandas as pd

def get_human_dataset( ):
    # 각 데이터 파일들은 공백으로 분리되어 있으므로 read_csv에서 공백 문자를 sep으로 활용.
    feature_name_df = pd.read_csv('./human_activity/features.txt',sep='\s+', header=None, names=['column_index','column_name'])

    # 중복된 피처명을 수정하는 get_new_feature_name_df()를 이용, 신규 피처명 DataFrame생성.
    new_feature_name_df = get_new_feature_name_df(feature_name_df)

    # DataFrame에 피처명을 컬럼으로 부여하기 위해 리스트 객체로 다시 변환
    feature_name = new_feature_name_df.iloc[:, 1].values.tolist()

    # 학습 피처 데이터 셋과 테스트 피처 데이터를 DataFrame으로 로딩. 컬럼명은 feature_name 적용
    X_train = pd.read_csv('./human_activity/train/X_train.txt',sep='\s+', names=feature_name )
    X_test = pd.read_csv('./human_activity/test/X_test.txt',sep='\s+', names=feature_name)

    # 학습 레이블과 테스트 레이블 데이터를 DataFrame으로 로딩하고 컬럼명은 action으로 부여
    y_train = pd.read_csv('./human_activity/train/y_train.txt',sep='\s+', header=None, names=['action'])
    y_test = pd.read_csv('./human_activity/test/y_test.txt',sep='\s+', header=None, names=['action'])

    # 로드된 학습/테스트용 DataFrame을 모두 반환
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = get_human_dataset()
```

Random Forest

결정트리에서 사용한 사용자 행동인지 데이터세트 사용

```
def get_new_feature_name_df(old_feature_name_df):
    feature_dup_df = pd.DataFrame(data=old_feature_name_df.groupby('column_name').cumcount(),
                                    columns=['dup_cnt'])
    feature_dup_df = feature_dup_df.reset_index()
    new_feature_name_df = pd.merge(old_feature_name_df.reset_index(), feature_dup_df, how='outer')
    new_feature_name_df[['column_name', 'dup_cnt']].apply(lambda x : x[0] + '_' + str(x[1])
                                                          if x[1] > 0 else x[0], axis=1)
    new_feature_name_df = new_feature_name_df.drop(['index'], axis=1)
    return new_feature_name_df
```

```
import pandas as pd

def get_human_dataset( ):
    # 각 데이터 파일들은 공백으로 분리되어 있으므로 read_csv에서 공백 문자를 sep으로 활용.
    feature_name_df = pd.read_csv('./human_activity/features.txt',sep='\s+', header=None, names=['column_index','column_name'])

    # 중복된 피처명을 수정하는 get_new_feature_name_df()를 이용, 신규 피처명 DataFrame생성.
    new_feature_name_df = get_new_feature_name_df(feature_name_df)

    # DataFrame에 피처명을 컬럼으로 부여하기 위해 리스트 객체로 다시 변환
    feature_name = new_feature_name_df.iloc[:, 1].values.tolist()

    # 학습 피처 데이터 셋과 테스트 피처 데이터를 DataFrame으로 로딩. 컬럼명은 feature_name 적용
    X_train = pd.read_csv('./human_activity/train/X_train.txt',sep='\s+', names=feature_name )
    X_test = pd.read_csv('./human_activity/test/X_test.txt',sep='\s+', names=feature_name)

    # 학습 레이블과 테스트 레이블 데이터를 DataFrame으로 로딩하고 컬럼명은 action으로 부여
    y_train = pd.read_csv('./human_activity/train/y_train.txt',sep='\s+', header=None, names=['action'])
    y_test = pd.read_csv('./human_activity/test/y_test.txt',sep='\s+', header=None, names=['action'])

    # 로드된 학습/테스트용 DataFrame을 모두 반환
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = get_human_dataset()
```

Random Forest

결정트리에서 사용한 사용자 행동인지 데이터세트 사용

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

# 결정 트리에서 사용한 get_human_dataset()을 이용해 학습/테스트용 DataFrame 반환
X_train, X_test, y_train, y_test = get_human_dataset()

# 랜덤 포레스트 학습 및 별도의 테스트 셋으로 예측 성능 평가
rf_clf = RandomForestClassifier(random_state=0)
rf_clf.fit(X_train, y_train)
pred = rf_clf.predict(X_test)
accuracy = accuracy_score(y_test, pred)
print('랜덤 포레스트 정확도: {:.4f}'.format(accuracy))
```

랜덤 포레스트 정확도: 0.9253

```
from sklearn.model_selection import GridSearchCV

params = {
    'n_estimators':[100],
    'max_depth' : [6, 8, 10, 12],
    'min_samples_leaf' : [8, 12, 18 ],
    'min_samples_split' : [8, 16, 20]
}
# RandomForestClassifier 객체 생성 후 GridSearchCV 수행
rf_clf = RandomForestClassifier(random_state=0, n_jobs=-1)      pc의 모든 resources를 다 사용
grid_cv = GridSearchCV(rf_clf, param_grid=params, cv=2, n_jobs=-1)
grid_cv.fit(X_train, y_train)

print('최적 하이퍼 파라미터:\n', grid_cv.best_params_)
print('최고 예측 정확도: {:.4f}'.format(grid_cv.best_score_))
```

최적 하이퍼 파라미터:

```
{'max_depth': 10, 'min_samples_leaf': 8, 'min_samples_split': 8, 'n_estimators': 100}
```

최고 예측 정확도: 0.9180

Random Forest

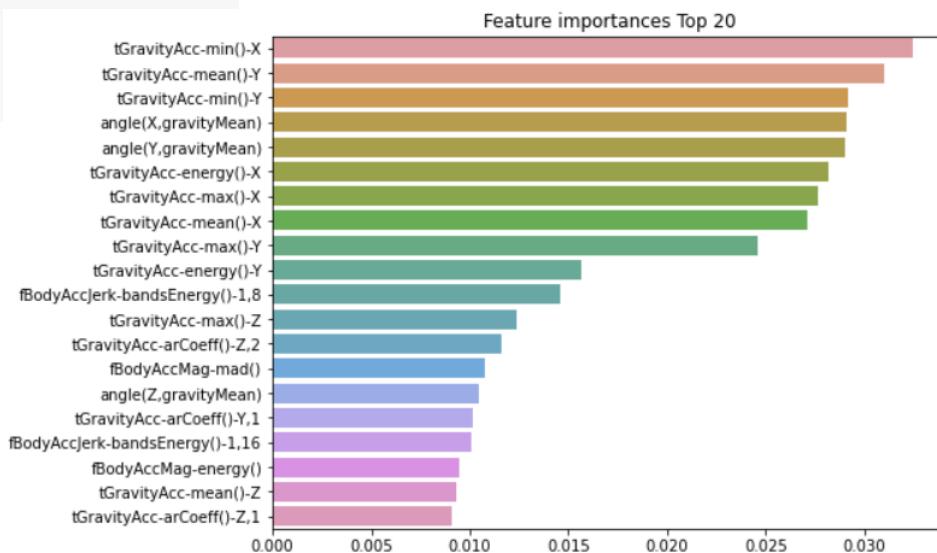
결정트리에서 사용한 사용자 행동인지 데이터세트 사용

```
rf_clf1 = RandomForestClassifier(n_estimators=300, max_depth=10, min_samples_leaf=8, #  
                                min_samples_split=8, random_state=0)  
rf_clf1.fit(X_train, y_train)  
pred = rf_clf1.predict(X_test)  
print('예측 정확도: {:.4f}'.format(accuracy_score(y_test, pred)))
```

예측 정확도: 0.9165

```
import matplotlib.pyplot as plt  
import seaborn as sns  
%matplotlib inline  
  
ftr_importances_values = rf_clf1.feature_importances_  
ftr_importances = pd.Series(ftr_importances_values, index=X_train.columns )  
ftr_top20 = ftr_importances.sort_values(ascending=False)[:20]  
  
plt.figure(figsize=(8,6))  
plt.title('Feature importances Top 20')  
sns.barplot(x=ftr_top20, y=ftr_top20.index)  
plt.show()
```

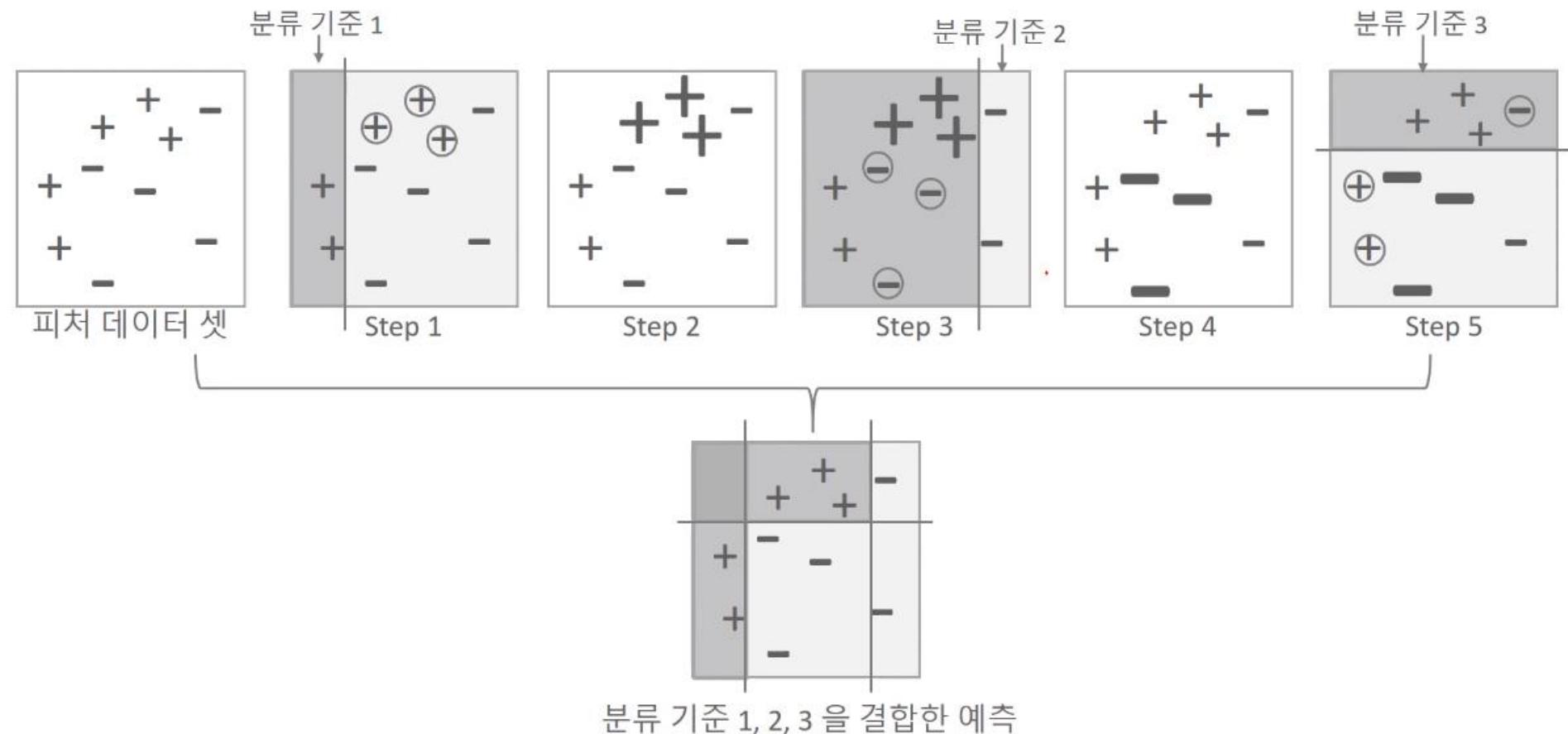
튜닝된 하이퍼 파라미터로 재학습 및 예측/평가



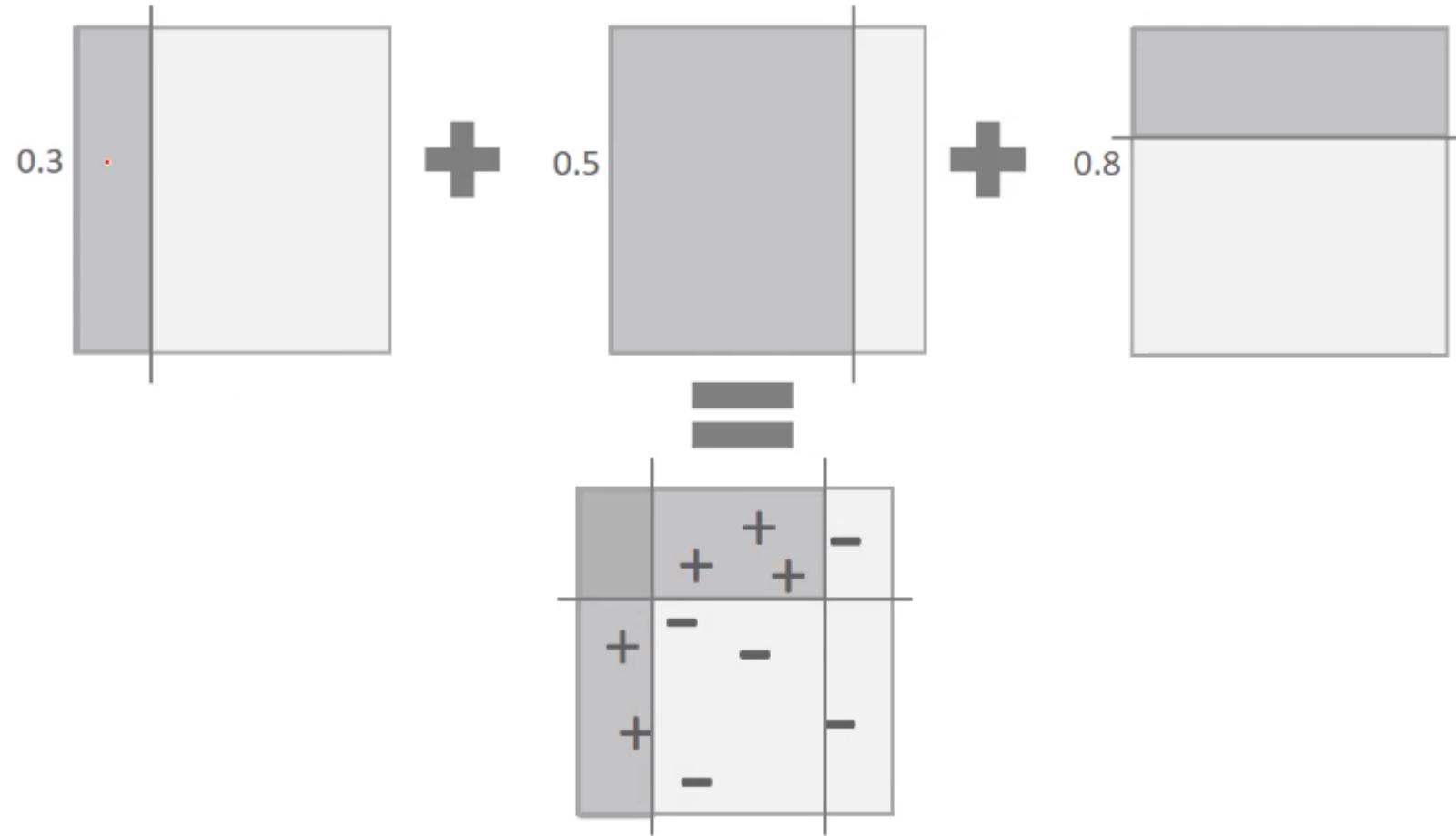
Boosting

- 부스팅 알고리즘은 여러 개의 약한 학습기(weak learner)를 순차적으로 학습-예측하면서 잘못 예측한 데이터에 가중치 부여를 통해 오류를 개선해 나가면서 학습하는 방식입니다.
 - 부스팅의 대표적인 구현은 AdaBoost(Adaptive boosting)와 그래디언트 부스트가 있습니다
- 수행시간이 엄청 걸린다.

Ada Boost 학습/예측 프로세스 -1



Ada Boost 학습/예측 프로세스 -1



GBM(Gradient Boost Machine)개요

GBM(Gradient Boost Machine)도 에이다부스트와 유사하나, 가중치 업데이트를 경사 하강법 (Gradient Descent)을 이용하는 것이 큰 차이입니다. 오류 값은 실제 값 – 예측값입니다. 분류의 실제 결괏값을 y , 피처를 X_1, X_2, \dots, X_n , 그리고 이 피처에 기반한 예측 함수를 $F(x)$ 함수라고 하면 오류식 $h(x) = y - F(x)$ 이 됩니다. 이 오류식 $h(x) = y - F(x)$ 을 최소화하는 방향성을 가지고 반복적으로 가중치 값을 업데이트하는 것이 경사 하강법(Gradient Descent)입니다. 경사 하강법은 반복 수행을 통해 오류를 최소화할 수 있도록 가중치의 업데이트 값을 도출하는 기법으로서 머신러닝에서 중요한 기법 중 하나입니다

GBM Hyper Parameter

사이킷런은 GBM 분류를 위해 `GradientBoostingClassifier` 클래스를 제공합니다.

- **loss:** 경사 하강법에서 사용할 비용 함수를 지정합니다. 특별한 이유가 없으면 기본값인 'deviance'를 그대로 적용합니다.
- **learning_rate:** GBM이 학습을 진행할 때마다 적용하는 학습률입니다. Weak learner가 순차적으로 오류 값을 보정해 나가는 데 적용하는 계수입니다. 0~1 사이의 값을 지정할 수 있으며 기본값은 0.1입니다. 너무 작은 값을 적용하면 업데이트 되는 값이 작아져서 최소 오류 값을 찾아 예측 성능이 높아질 가능성이 높습니다. 하지만 많은 weak learner는 순차적인 반복이 필요해서 수행 시간이 오래 걸리고, 또 너무 작게 설정하면 모든 weak learner의 반복이 완료돼도 최소 오류 값을 찾지 못할 수 있습니다. 반대로 큰 값을 적용하면 최소 오류 값을 찾지 못하고 그냥 지나쳐 버려 예측 성능이 떨어질 가능성이 높아지지만, 빠른 수행이 가능합니다.
- **n_estimators:** weak learner의 개수입니다. weak learner가 순차적으로 오류를 보정하므로 개수가 많을수록 예측 성능이 일정 수준까지는 좋아질 수 있습니다. 하지만 개수가 많을수록 수행 시간이 오래 걸립니다. 기본값은 100입니다.
- **subsample:** weak learner가 학습에 사용하는 데이터의 샘플링 비율입니다. 기본값은 1이며, 이는 전체 학습 데이터를 기반으로 학습한다는 의미입니다(0.5이면 학습 데이터의 50%). 과적합이 염려되는 경우 `subsample`을 1보다 작은 값으로 설정합니다.

GBM Hyper Parameter

사이킷런은 GBM 분류를 위해 `GradientBoostingClassifier` 클래스를 제공합니다.

- **loss:** 경사 하강법에서 사용할 비용 함수를 지정합니다. 특별한 이유가 없으면 기본값인 'deviance'를 그대로 적용합니다.
- **learning_rate:** GBM이 학습을 진행할 때마다 적용하는 학습률입니다. Weak learner가 순차적으로 오류 값을 보정해 나가는 데 적용하는 계수입니다. 0~1 사이의 값을 지정할 수 있으며 기본값은 0.1입니다. 너무 작은 값을 적용하면 업데이트 되는 값이 작아져서 최소 오류 값을 찾아 예측 성능이 높아질 가능성이 높습니다. 하지만 많은 weak learner는 순차적인 반복이 필요해서 수행 시간이 오래 걸리고, 또 너무 작게 설정하면 모든 weak learner의 반복이 완료돼도 최소 오류 값을 찾지 못할 수 있습니다. 반대로 큰 값을 적용하면 최소 오류 값을 찾지 못하고 그냥 지나쳐 버려 예측 성능이 떨어질 가능성이 높아지지만, 빠른 수행이 가능합니다.
- **n_estimators:** weak learner의 개수입니다. weak learner가 순차적으로 오류를 보정하므로 개수가 많을수록 예측 성능이 일정 수준까지는 좋아질 수 있습니다. 하지만 개수가 많을수록 수행 시간이 오래 걸립니다. 기본값은 100입니다.
- **subsample:** weak learner가 학습에 사용하는 데이터의 샘플링 비율입니다. 기본값은 1이며, 이는 전체 학습 데이터를 기반으로 학습한다는 의미입니다(0.5이면 학습 데이터의 50%). 과적합이 염려되는 경우 `subsample`을 1보다 작은 값으로 설정합니다.

GBM Hyper Parameter

```
from sklearn.ensemble import GradientBoostingClassifier
import time
import warnings
warnings.filterwarnings('ignore')

X_train, X_test, y_train, y_test = get_human_dataset()

# GBM 수행 시간 측정을 위한 시작 시간 설정.
start_time = time.time()

gb_clf = GradientBoostingClassifier(random_state=0)
gb_clf.fit(X_train, y_train)
gb_pred = gb_clf.predict(X_test)
gb_accuracy = accuracy_score(y_test, gb_pred)

print('GBM 정확도: {:.4f}'.format(gb_accuracy))
print("GBM 수행 시간: {:.1f} 초 ".format(time.time() - start_time))
```

GBM 정확도: 0.9376

GBM 수행 시간: 169.6 초

GBM Hyper Parameter

```
from sklearn.model_selection import GridSearchCV

params = {
    'n_estimators':[100, 500],
    'learning_rate' : [ 0.05, 0.1]
}
grid_cv = GridSearchCV(gb_clf , param_grid=params , cv=2 ,verbose=1)
grid_cv.fit(X_train , y_train)
print('최적 하이퍼 파라미터 :',grid_cv.best_params_)
print('최고 예측 정확도: {:.4f}'.format(grid_cv.best_score_))
```

일반적으로 30분 소요

Fitting 2 folds for each of 4 candidates, totalling 8 fits

[Parallel(n_jobs=1)]: Done 8 out of 8 | elapsed: 18.2min finished

최적 하이퍼 파라미터 :

{'learning_rate': 0.05, 'n_estimators': 500}

최고 예측 정확도: 0.9010

```
# GridSearchCV를 이용하여 최적으로 학습된 estimator로 predict 수행.
gb_pred = grid_cv.best_estimator_.predict(X_test)
gb_accuracy = accuracy_score(y_test, gb_pred)
print('GBM 정확도: {:.4f}'.format(gb_accuracy))
```

GBM 정확도: 0.9406

(Easy Classification Method)

서포트 벡터 머신

■ 서포트 벡터 머신(SVM: Support Vector Machine)

- 분류와 회귀 방식에 모두 사용 가능
- 선형적인 경우뿐만 아니라 초평면(Hyper plane)과 같은 비선형적인 경우에도 사용할 수 있음
- 주어진 데이터 집합을 바탕으로 테스트 데이터가 비확률적인 이진 선형 분류 모델에도 적용할 수 있음

명령문	<pre>from sklearn import datasets iris = datasets.load_iris() X = iris.data y = iris.target from sklearn.model_selection import train_test_split X_train, X_test, y_train, y_test = train_test_split (X, y, test_size=.5)</pre>
결과의 예	0.9866666666666667 # 높은 예측 결과를 보임.

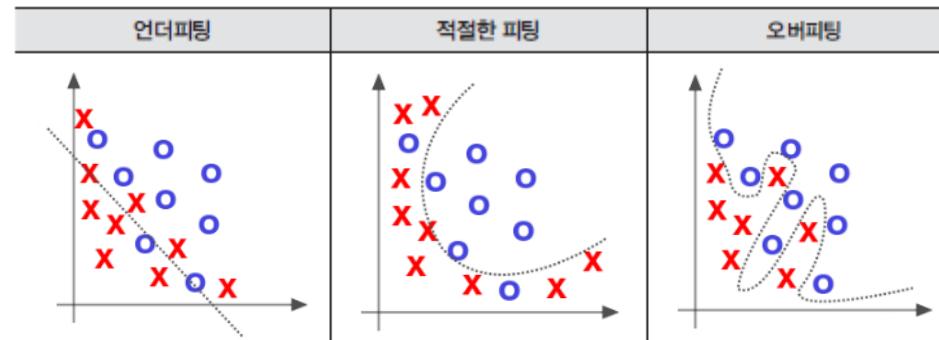
서포트 벡터 머신

■ 감마

- 감마의 값을 증가시키면 결정 경계가 더욱 복잡해지며 서포트 벡터 머신에서는 과도한 연산을 초래할 수 있지만 인식률을 향상시킬 수 있음
- 데이터의 양과 특성에 적절한 값을 설정

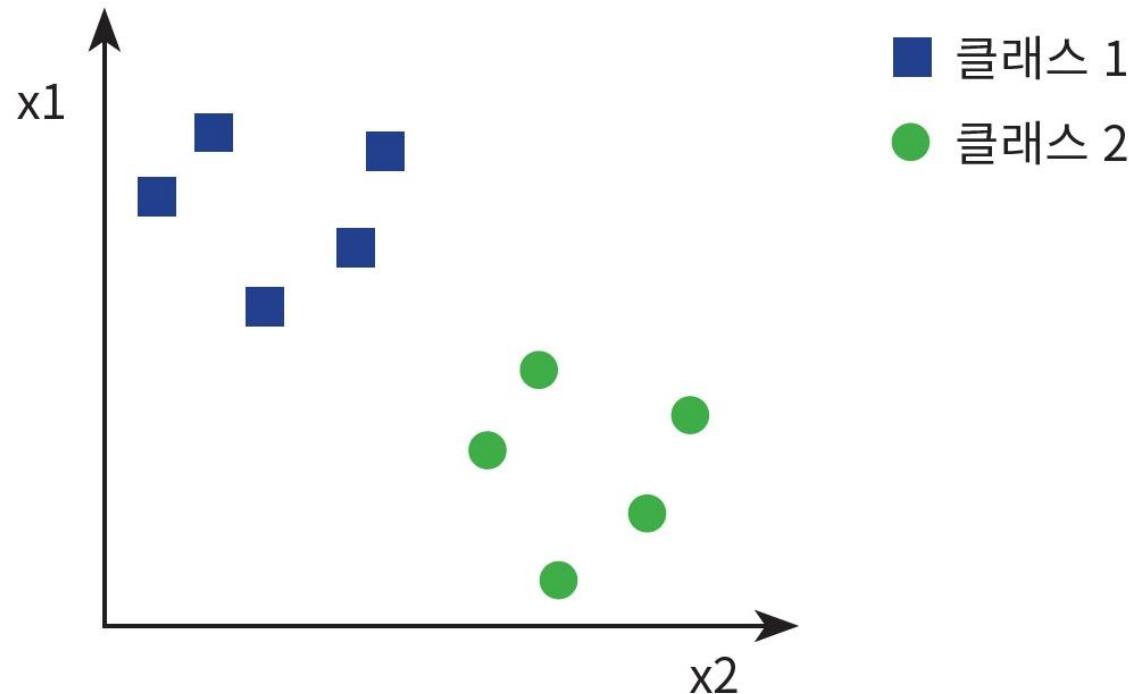
■ C

- C의 값은 학습 시 잘못된 분류를 생성할 때 부여하는 벌점(penalty)을 의미
- 큰 값을 설정하면 분류의 결과에 오버피팅(Over fitting)을 초래
- 작은 값을 설정하면 언더피팅(Under fitting)이 발생



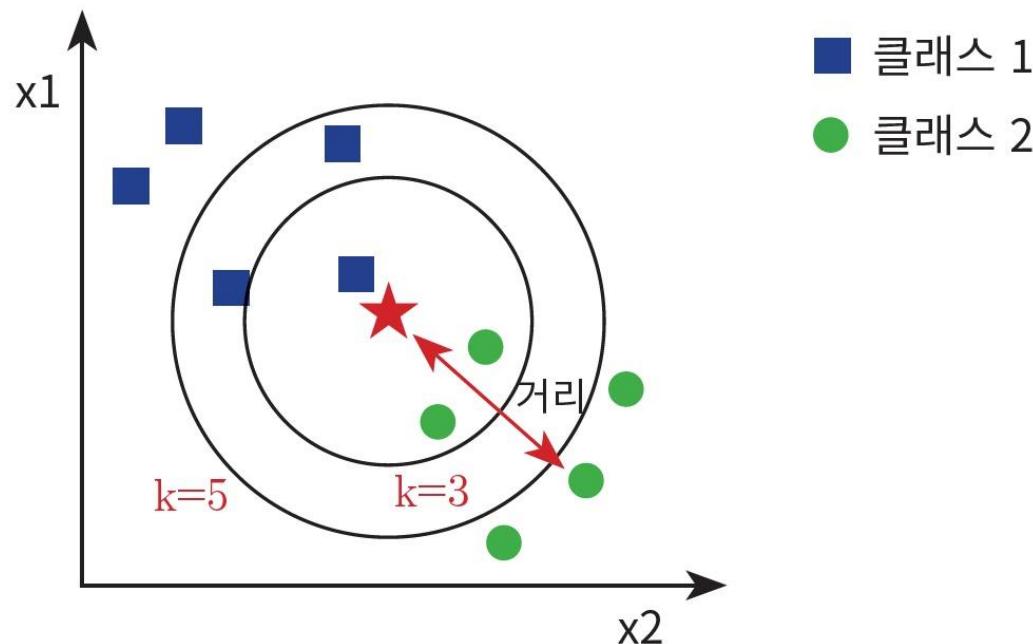
kNN 알고리즘

- k-Nearest Neighbor(kNN) 알고리즘은 모든 기계 학습 알고리즘 중에서도 가장 간단하고 이해하기 쉬운 분류 알고리즘



kNN 알고리즘

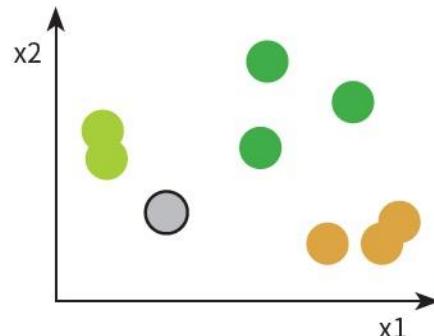
- 이제 새로운 데이터가 입력되어서 그래프 상에 별표로 표시되었다고 하자. 별표는 파랑색 사각형과 빨강색 원 중에서 하나에 속해야 한다. 이것을 분류(classification)라고 한다.



kNN 알고리즘

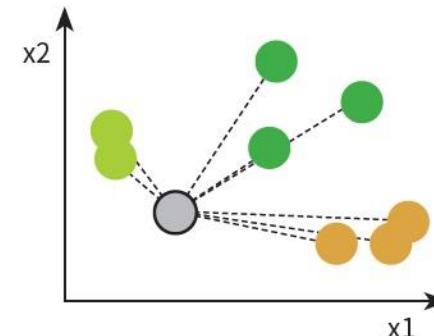
kNN 알고리즘

1. 데이터를 관찰한다.



회색 원은 어디에 속해야 할까?

2. 거리를 계산한다.



회색 원과 다른 원들간의 거리를 계산한다.

3. 이웃을 찾는다.

점	거리	
●	2.1	→ 1등
●	2.4	→ 2등
●	3.1	→ 3등
●	4.5	→ 4등

거리에 따라서 이웃 원들을 정렬한다.

4. 새로운 데이터에 대하여 투표한다.

클래스 투표수

●	2
●	1
●	1

●색 원이 가장 많았으므로 새로운 원은 ●에 속한다.

가장 가까운 k개의 이웃 중에서 가장 많은 표를 얻은 클래스로 분류한다.

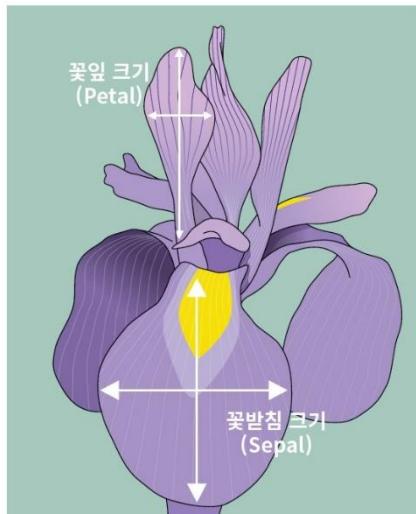
kNN 알고리즘의 장점과 단점

- 특정 공간에 있는 모든 데이터에 대한 정보가 필요하다. 왜냐하면, 가장 가까운 이웃을 찾기 위해 새로운 데이터에서 모든 기존 데이터까지의 거리를 확인해야 하기 때문이다. 데이터와 클래스가 많이 있다면, 많은 메모리 공간과 계산 시간이 필요하다.
- 어떤 종류의 학습이나 준비 시간이 필요 없다.

학습 모델

■ 아이리스(붓꽃)의 세부 분류 예제

- 아이리스의 종류:
 - 세토사(Setosa)
 - 버시컬러(Versicolor)
 - 버지니카(Virginica)
- 4개의 특징(feature)인 꽃잎과 꽃받침의 길이와 폭에 의해서 분류됨
- 150개의 아이리스 꽃의 정보를 가지고 있는 토이 데이터 세트(Toy Dataset)를 인터넷에서 구할 수 있음



5.1,3.8,1.9,0.4,Iris-setosa
4.8,3.0,1.4,0.3,Iris-setosa
5.1,3.8,1.6,0.2,Iris-setosa
4.6,3.2,1.4,0.2,Iris-setosa
5.3,3.7,1.5,0.2,Iris-setosa
5.0,3.3,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
5.7,2.8,4.5,1.3,Iris-versicolor

(아이리스 꽃: 꽃받침 길이와 너비, 꽃잎 길이와 너비)

About 학습 모델

■ 아이리스 데이터의 일부를 출력

명령문	<pre>from sklearn.datasets import load_iris # iris 데이터 세트 가져오기 iris = load_iris() # iris 토이 데이터 세트를 iris변수에 저장 print(iris.feature_names) # 특징(feature)들을 출력 print(iris.target_names) # 학습 라벨에 해당하는 target을 출력 print(iris.data[0]) # 첫 번째 데이터에 저장된 feature값 출력 print(iris.target[0]) # 첫 번째 데이터에 저장된 target값 출력</pre>
결과	<pre>['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)'] ['setosa' 'versicolor' 'virginica'] [5.1 3.5 1.4 0.2] 0</pre>

target값:
• 세토사(0)
• 버시컬러(1)
• 버지니카(2)

■ 아이리스 데이터 세트에 저장된 내용을 모두 출력

명령문	<pre>from sklearn.datasets import load_iris iris = load_iris() for i in range(len(iris.target)): print("Example %d: label %s, features %s" % (i, iris. target[i], iris.data[i]))</pre>
결과	<pre>Example 0: label 0, features [5.1 3.5 1.4 0.2] Example 1: label 0, features [4.9 3. 1.4 0.2] (중간 생략) Example 50: label 1, features [7. 3.2 4.7 1.4] Example 51: label 1, features [6.4 3.2 4.5 1.5] (중간 생략) Example 148: label 2, features [6.2 3.4 5.4 2.3] Example 149: label 2, features [5.9 3. 5.1 1.8]</pre>

0~49: 세토사

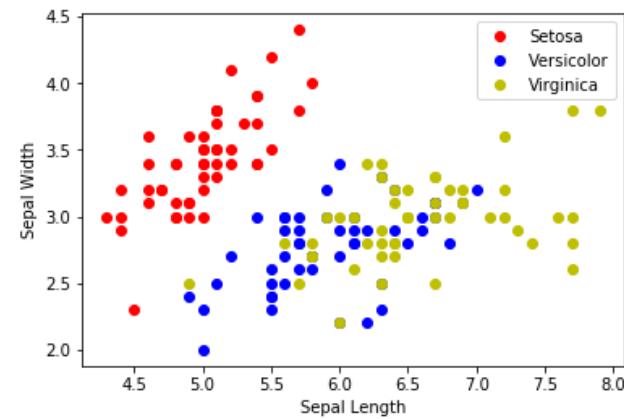
50~99: 버시컬러

100~149: 버지니카

■ 아이리스 데이터 세트의 그래프

명령문

```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
iris = load_iris()
sepal = iris.data[:, 0:2]
kind = iris.target
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.plot(sepal[kind==0][:,0], sepal[kind==0][:,1],
         "ro", label='Setosa')
plt.plot(sepal[kind==1][:,0], sepal[kind==1][:,1],
         "bo", label='Versicolor')
plt.plot(sepal[kind==2][:,0], sepal[kind==2][:,1],
         "yo", label='Virginica')
plt.legend()
```



kNN 학습

```
from sklearn.datasets import load_iris
iris = load_iris()
#print(iris.data)

from sklearn.model_selection import train_test_split

X = iris.data
y = iris.target

# (80:20)으로 분할한다.
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=4)
```

```
print(X_train.shape)
print(X_test.shape)
```

```
(120, 4)
(30, 4)
```

kNN 하나의 가까운

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
scores = metrics.accuracy_score(y_test, y_pred)
scores
```

0.9666666666666667

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X, y)

# 0 = setosa, 1=versicolor, 2=virginica
classes = {0:'setosa',1:'versicolor',2:'virginica'}

# 아직 보지 못한 새로운 데이터를 제시해보자.
x_new = [[3,4,5,2],
          [5,4,2,2]]
y_predict = knn.predict(x_new)

print(classes[y_predict[0]])
print(classes[y_predict[1]])
```

versicolor
setosa

Chapter 05

회귀 (Regression)

회귀소개

- 회귀는 현대 통계학을 이루는 큰 축
- 회귀 분석은 유전적 특성을 연구하던 영국의 통계학자 갈톤(Galton)이 수행한 연구에서 유래했다는 것이 일반론

“부모의 키가 크더라도 자식의 키가 대를 이어 무한정 커지지 않으며, 부모의 키가 작더라도 대를 이어 자식의 키가 무한정 작아지지 않는다”

회귀 분석은 이처럼 데이터 값이 평균과 같은 일정한 값으로 돌아가려는 경향을 이용한 통계학 기법입니다



회귀의 개요

회귀는 여러 개의 독립변수와 한 개의 종속변수 간의 상관관계를 모델링하는 기법을 통칭합니다

아파트 가격은 ?

방 개수

아파트
크기

주변 학군

근처
지하철 역
갯수

$$Y = W_1 * X_1 + W_2 * X_2 + W_3 * X_3 + \dots + W_n * X_n$$

Y 는 종속변수, 즉 아파트 가격

$X_1, X_2, X_3, \dots, X_n$ 은 방 개수, 아파트 크기, 주변 학군등의 독립 변수

$W_1, W_2, W_3 \dots W_n$ 은 이 독립변수의 값에 영향을 미치는 회귀 계수(Regression coefficients)

머신러닝 회귀 예측의 핵심은 주어진 피처와 결정 값 데이터 기반에서 학습을 통해 최적의 회귀 계수를 찾아내는 것입니다

회귀의 유형

- 회귀는 회귀 계수의 선형/비선형 여부, 독립변수의 개수, 종속변수의 개수에 따라 여러 가지 유형으로 나눌 수 있습니다. 회귀에서 가장 중요한 것은 바로 회귀 계수입니다. 이 회귀 계수가 '선형이나 아니냐'에 따라 선형 회귀와 비선형 회귀로 나눌 수 있습니다. 그리고 독립변수의 개수가 한 개인지 여러 개인지에 따라 단일 회귀, 다중 회귀로 나뉩니다

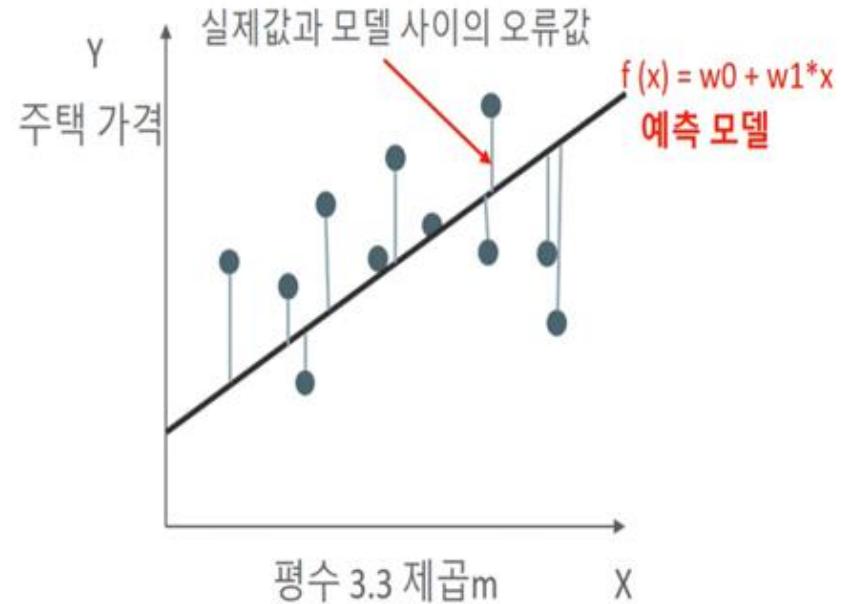
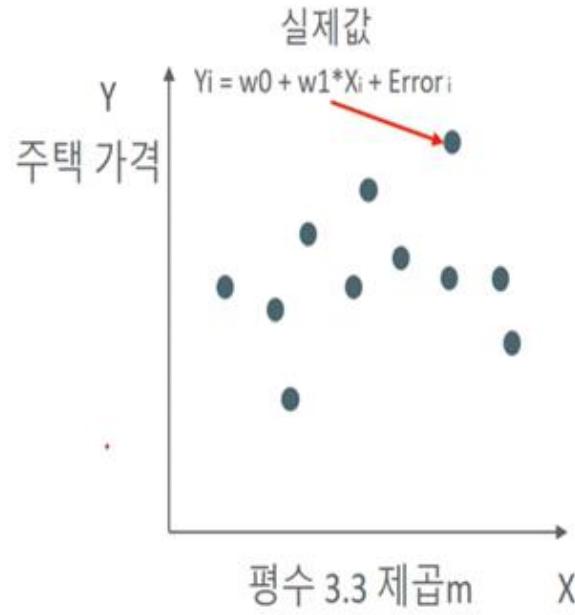
독립변수 개수	회귀 계수의 결합
1개: 단일 회귀	선형: 선형 회귀
여러 개: 다중 회귀	비선형: 비선형 회귀

선형회귀의 종류

- 일반 선형 회귀: 예측값과 실제 값의 RSS(Residual Sum of Squares)를 최소화할 수 있도록 회귀 계수를 최적화하며, 규제(Regularization)를 적용하지 않은 모델입니다.
- 릿지(Ridge): 릿지 회귀는 선형 회귀에 L2 규제를 추가한 회귀 모델입니다.
- 라쏘(Lasso): 라쏘 회귀는 선형 회귀에 L1 규제를 적용한 방식입니다
- 엘라스틱넷(ElasticNet): L2, L1 규제를 함께 결합한 모델입니다.
- 로지스틱 회귀(Logistic Regression): 로지스틱 회귀는 회귀라는 이름이 붙어 있지만, 사실은 분류에 사용되는 선형 모델입니다.

단순선형 회귀(Simple Regression)를 통한 회귀의 이해

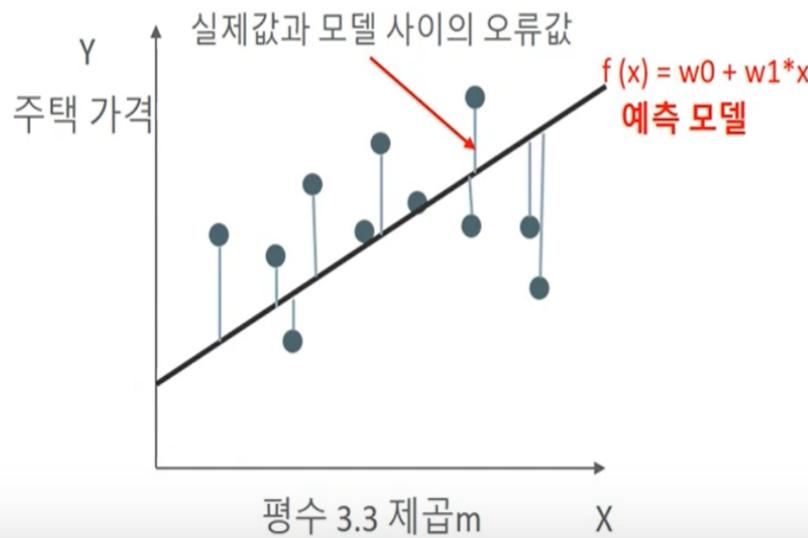
주택 가격이 주택의 크기로만 결정되는 단순 선형 회귀로 가정하면 다음과 같이 주택 가격은 주택 크기에 대해 선형(직선 형태)의 관계로 표현할 수 있습니다.



RSS 기반의 회귀 오류 측정

RSS

오류 값의 제곱을 구해서 더하는 방식입니다. 일반적으로 미분 등의 계산을 편리하게 하기 위해서 RSS 방식으로 오류 합을 구합니다. 즉, $Error = RSS$ 입니다



$$\begin{aligned} RSS = & (\#1 \text{ 주택 가격} - (w_0 + w_1 * \#1 \text{ 주택 크기}))^2 \\ & + (\#2 \text{ 주택 가격} - (w_0 + w_1 * \#2 \text{ 주택 크기}))^2 \\ & + (\#3 \text{ 주택 가격} - (w_0 + w_1 * \#3 \text{ 주택 크기}))^2 \\ & + \dots (\text{모든 학습 데이터에 대해 RSS 수행}) \end{aligned}$$

RSS의 이해

- RSS는 이제 변수가 W_0, W_1 인 식으로 표현할 수 있으며, 이 RSS를 최소로 하는 W_0, W_1 , 즉 회귀 계수를 학습을 통해서 찾는 것이 머신러닝 기반 회귀의 핵심 사항입니다.
- RSS는 회귀식의 독립변수 X, 종속변수 Y가 중심 변수가 아니라 w 변수(회귀 계수)가 중심 변수임을 인지하는 것이 매우 중요합니다(학습 데이터로 입력되는 독립변수와 종속변수는 RSS에서 모두 상수로 간주합니다).
- 일반적으로 RSS는 학습 데이터의 건수로 나누어서 다음과 같이 정규화된 식으로 표현됩니다.

$$RSS(w_0, w_1) = \frac{1}{N} \sum_{i=1}^N (y_i - (w_0 + w_1 * x_i))^2$$

(i는 1부터 학습 데이터의 총 건수 N까지)

RSS 회귀의 비용 함수

회귀의 비용 함수(Cost function)

$$RSS(w_0, w_1) = \frac{1}{N} \sum_{i=1}^N (y_i - (w_0 + w_1 * x_i))^2$$

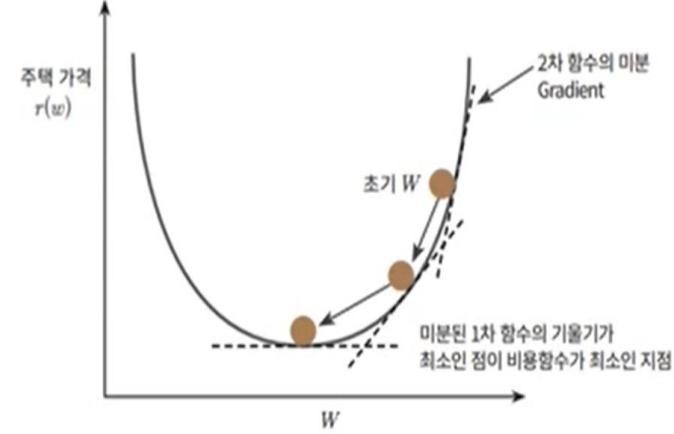
(i 는 1부터 학습 데이터의 총 건수 N 까지)

회귀에서 이 RSS는 비용(Cost)이며 w 변수(회귀 계수)로 구성되는 RSS를 **비용 함수**라고 합니다. 머신 러닝 회귀 알고리즘은 데이터를 계속 학습하면서 이 비용 함수가 반환하는 값(즉, 오류 값)을 지속해서 감소시키고 최종적으로는 더 이상 감소하지 않는 최소의 오류 값을 구하는 것입니다. 비용 함수를 손실함수(loss function)라고도 합니다.

비용 최소화 하기 – 경사 하강법 (Gradient Descent)

W 파라미터의 개수가 적다면 고차원 방정식으로 비용 함수가 최소가 되는 W 변수값을 도출할 수 있겠지만, W 파라미터가 많으면 고차원 방정식을 동원하더라도 해결하기가 어렵습니다. 경사 하강법은 이러한 고차원 방정식에 대한 문제를 해결해 주면서 비용 함수 RSS를 최소화하는 방법을 직관적으로 제공하는 뛰어난 방식입니다.

많은 W 파라미터가 있는 경우에 경사 하강법은 보다 간단하고 직관적인 비용함수 최소화 솔루션을 제공



경사 하강법의 사전적 의미인 '점진적인 하강'이라는 뜻에서도 알 수 있듯이, '**점진적으로**' 반복적인 계산을 통해 W 파라미터 값을 업데이트하면서 오류 값이 최소가 되는 W 파라미터를 구하는 방식입니다

비용 최소화 하기 –경사 하강법 (Gradient Descent)

- 경사 하강법은 반복적으로 비용 함수의 반환 값, 즉 예측값과 실제 값의 차이가 작아지는 방향성을 가지고 W 파라미터를 지속해서 보정해 나갑니다.
- 최초 오류 값이 100이었다면 두 번째 오류 값은 100보다 작은 90, 세 번째는 80과 같은 방식으로 지속해서 오류를 감소시키는 방향으로 W 값을 계속 업데이트해 나갑니다.
- 그리고 오류 값이 더 이상 작아지지 않으면 그 오류 값을 최소 비용으로 판단하고 그때의 W 값을 최적 파라미터로 반환합니다



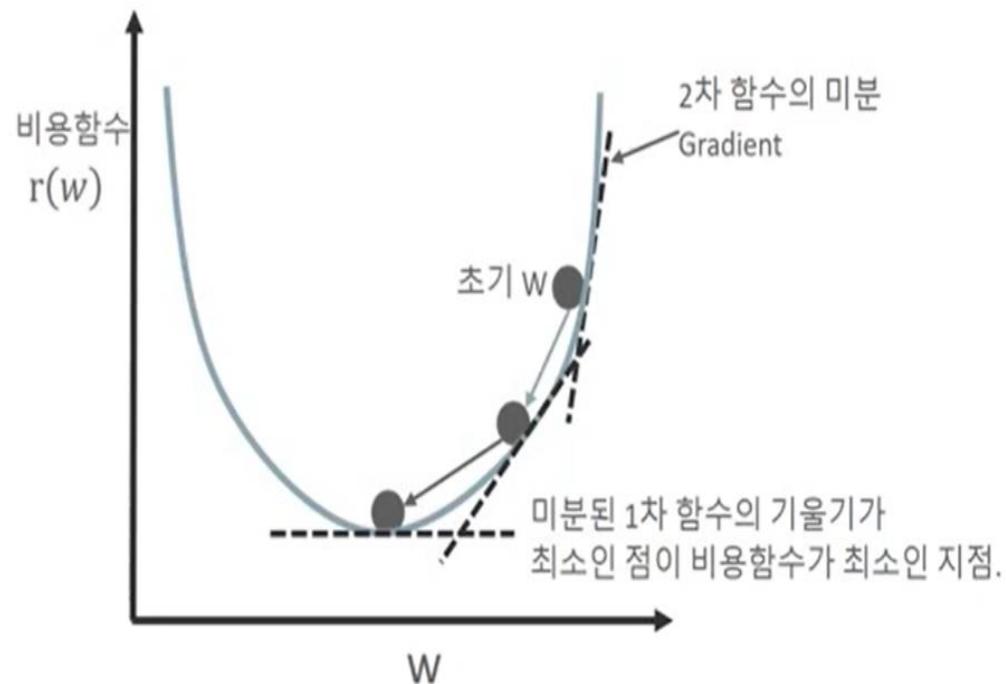
경사 하강법의 핵심은 “어떻게 하면 오류가 작아지는 방향으로 W 값을 보정할 수 있을까?”입니다.

미분을 통해 비용 함수의 최소 값 찾기

어떻게 하면 오류가 작아지는 방향으로 w 값을 보정할 수 있을까?

미분은 증가 또는 감소의 방향성을 나타냅니다.

비용 함수가 다음 그림과 같은 포물선 형태의 2차
함수라면 경사 하강법은 최초 w 에서부터 미분을 적용한
뒤 이 미분 값이 계속 감소하는 방향으로 순차적으로 w 를
업데이트 합니다.
마침내 더 이상 미분된 1차 함수의 기울기가 감소하지
않는 지점을 비용 함수가 최소인 지점으로 간주하고
그때의 w 를 반환합니다



RSS의 편미분

$R(w)$ 는 변수가 w 파라미터로 이뤄진 함수이며, $R(w) = \sum_{i=1}^n (y_i - (w_0 + w_1 * x_i))^2$ 입니다.

$R(w)$ 를 미분해 미분 함수의 최솟값을 구해야 하는데, $R(w)$ 는 두 개의 w 파라미터인 w_0 와 w_1 을 각각 가지고 있기 때문에 일반적인 미분을 적용할 수가 없고, w_0 , w_1 각 변수에 편미분을 적용해야 합니다. $R(w)$ 를 최소화하는 w_0 와 w_1 의 값은 각각 $r(w)$ 를 w_0 , w_1 으로 순차적으로 편미분을 수행해 얻을 수 있습니다.

$$\frac{\partial R(w)}{\partial w_1} = \frac{2}{N} \sum_{i=1}^N -x_i * (y_i - (w_0 + w_1 x_i)) = -\frac{2}{N} \sum_{i=1}^N x_i * (\text{실제값}_i - \text{예측값}_i)$$

$$\frac{\partial R(w)}{\partial w_0} = \frac{2}{N} \sum_{i=1}^N -(y_i - (w_0 + w_1 x_i)) = -\frac{2}{N} \sum_{i=1}^N (\text{실제값}_i - \text{예측값}_i)$$

**w 파라미터들(w_0 , w_1) 각각에 대해 RSS를 미분해서(편미분)
RSS를 최소화하는 w_0 , w_1 값을 구한다**

RSS의 편미분

$$\begin{aligned} R(w_0, w_1) &= \frac{1}{N} \sum_{i=1}^N (y_i - (w_0 + w_1 x_i))^2 \\ &= \frac{1}{N} \sum_{i=1}^N y_i^2 - 2y_i(w_0 + w_1 x_i) + (w_0 + w_1 x_i)^2 \\ &= \frac{1}{N} \sum_{i=1}^N y_i^2 - 2y_i(w_0 + w_1 x_i) + w_0^2 + 2w_0 w_1 x_i + w_1^2 x_i^2 \end{aligned}$$

① w_1 으로 편미분

$$\begin{aligned} \frac{dR(w_0, w_1)}{d w_1} &= \frac{1}{N} \sum_{i=1}^N -2y_i x_i + 2w_0 x_i + 2w_1 x_i^2 \\ &= \frac{2}{N} \sum_{i=1}^N -x_i * (y_i - w_0 - w_1 x_i) \\ &= \frac{2}{N} \sum_{i=1}^N -x_i * (y_i - (w_0 + w_1 x_i)) \\ &= \frac{-2}{N} \sum_{i=1}^N x_i * (y_i - (w_0 + w_1 x_i)) \\ &= \frac{2}{N} \sum_{i=1}^N x_i * (\text{실제값}_i - \text{예측값}_i) \end{aligned}$$

RSS의 편미분

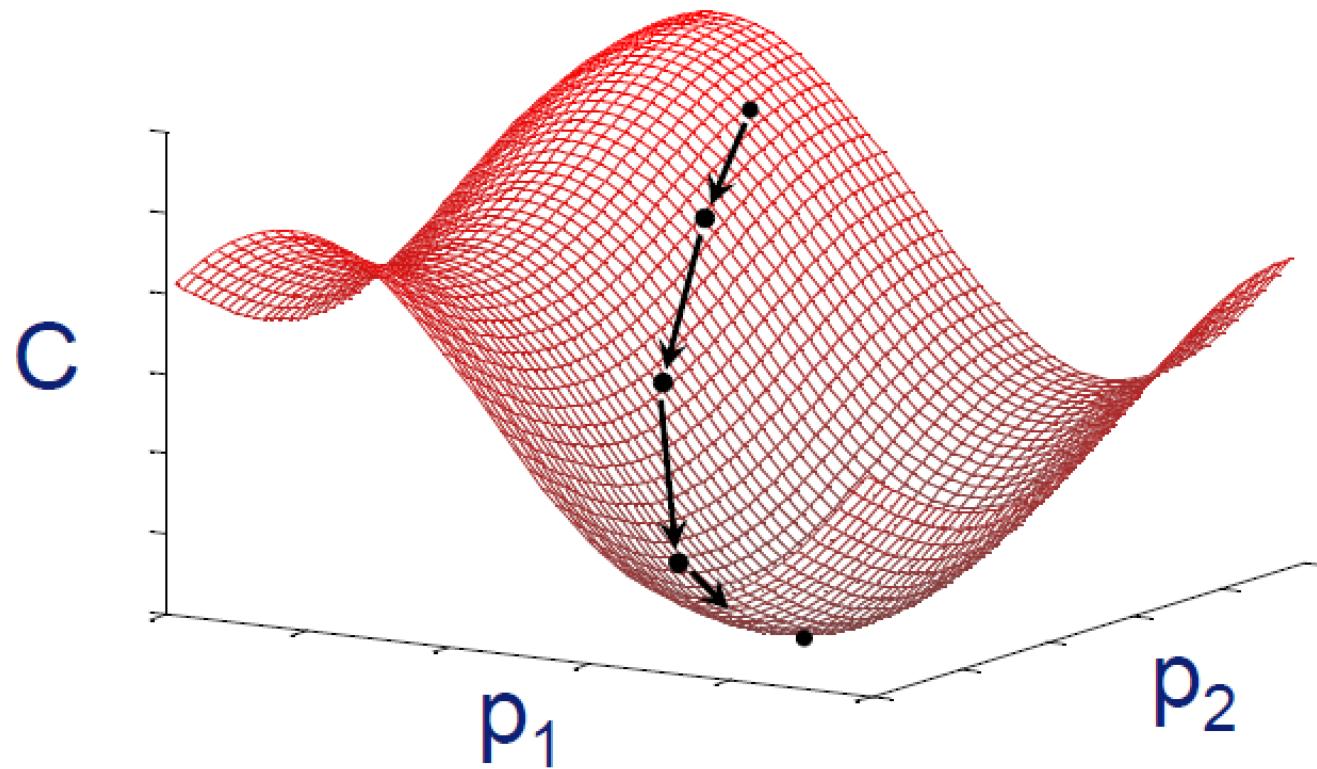
$$\begin{aligned} R(w_0, w_1) &= \frac{1}{N} \sum_{i=1}^N (y_i - (w_0 + w_1 x_i))^2 \\ &= \frac{1}{N} \sum_{i=1}^N y_i^2 - 2y_i(w_0 + w_1 x_i) + (w_0 + w_1 x_i)^2 \\ &= \frac{1}{N} \sum_{i=1}^N y_i^2 - 2y_i(w_0 + w_1 x_i) + w_0^2 + 2w_0 w_1 x_i + w_1^2 x_i^2 \end{aligned}$$

② w_0 로 편미분

$$\begin{aligned} \frac{\partial R(w_0, w_1)}{\partial w_0} &= \frac{1}{N} \sum_{i=1}^N -2y_i + 2w_0 + 2w_1 x_i \\ &= \frac{2}{N} \sum_{i=1}^N -cy - (w_0 + w_1 x_i) \\ &= \frac{-2}{N} \sum_{i=1}^N (\text{실제값}_i - \text{예측값}_i) \end{aligned}$$

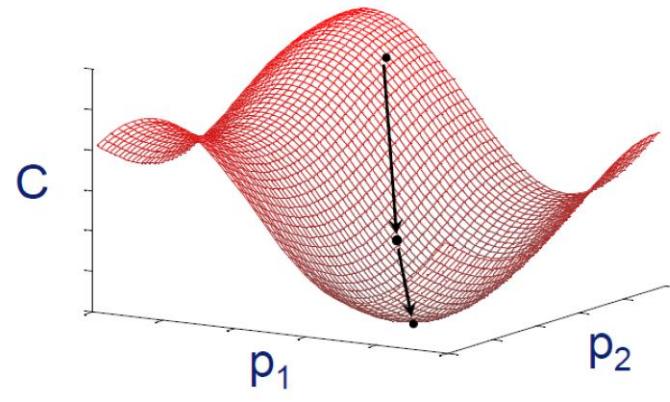
Fundamentals of Gradient Descent Methods

gradient descent

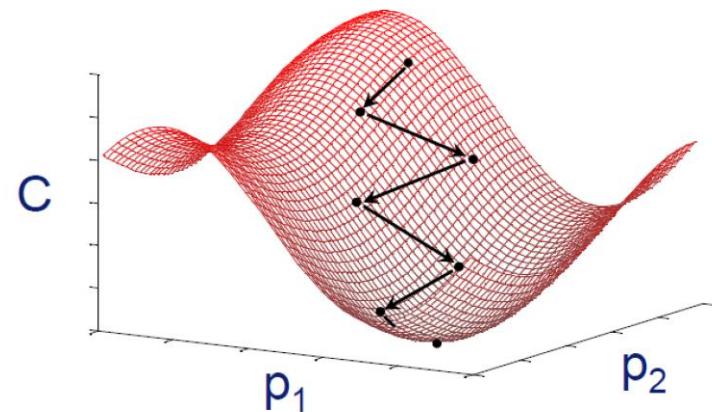


Fundamentals of Gradient Descent Methods

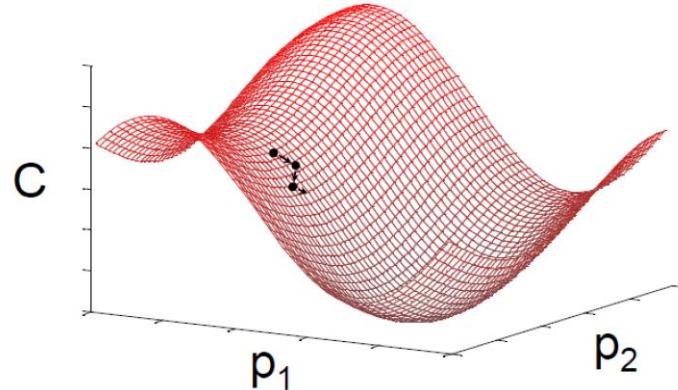
smarter steps



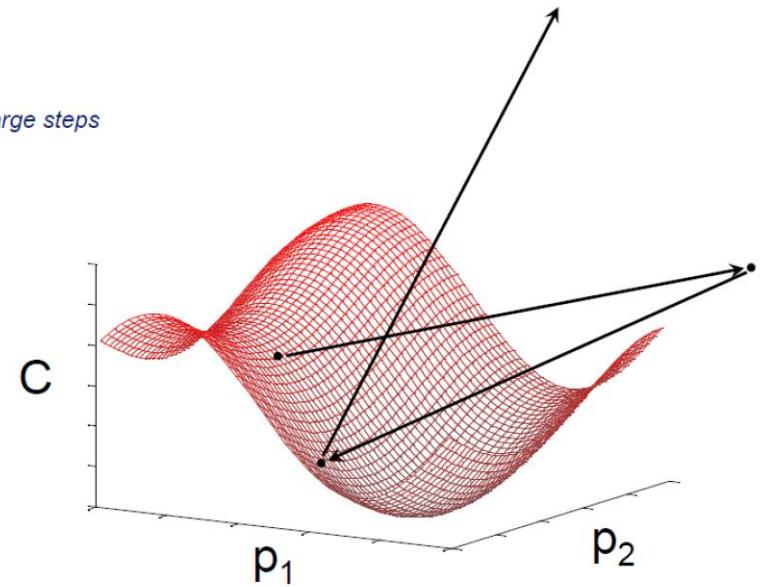
cheaper steps



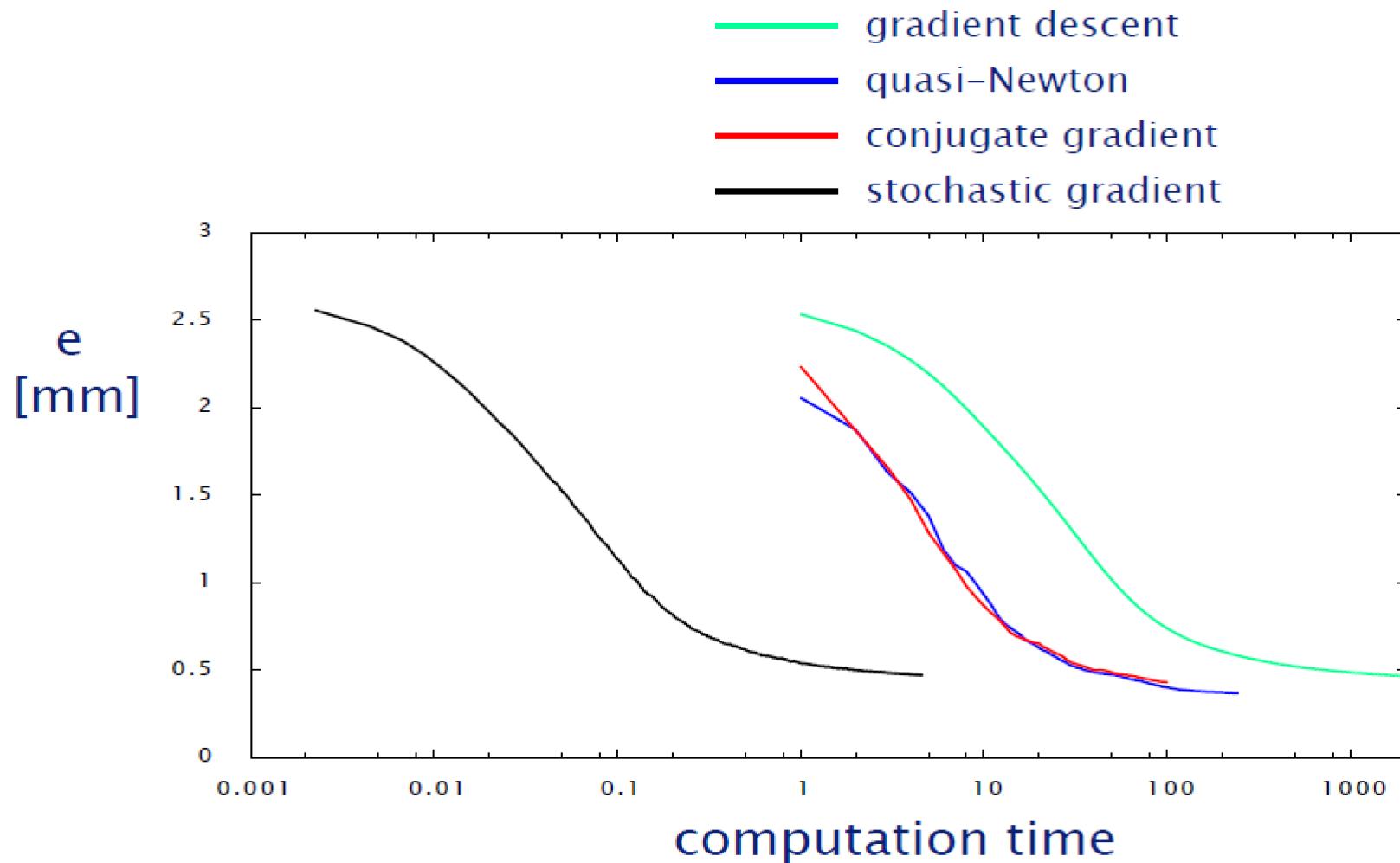
Too small steps



Too large steps



Fundamentals of Gradient Descent Methods



경사 하강법 정리

$$\frac{\partial R(w)}{\partial w_1} = \frac{2}{N} \sum_{i=1}^N -x_i * (y_i - (w_0 + w_1 x_i)) = -\frac{2}{N} \sum_{i=1}^N x_i * (\text{실제값}_i - \text{예측값}_i)$$

$$\frac{\partial R(w)}{\partial w_0} = \frac{2}{N} \sum_{i=1}^N -(y_i - (w_0 + w_1 x_i)) = -\frac{2}{N} \sum_{i=1}^N (\text{실제값}_i - \text{예측값}_i)$$

w_1, w_0 의 편미분 결과값인 $-\frac{2}{N} \sum_{i=1}^N x_i * (\text{실제값}_i - \text{예측값}_i)$ 와 $-\frac{2}{N} \sum_{i=1}^N (\text{실제값}_i - \text{예측값}_i)$ 을 반복적으로 보정하면서 w_1, w_0 값을 업데이트하면 비용함수 $R(W)$ 가 최소가 되는 w_1, w_0 값을 구할 수 있습니다. 하지만 실제로는 위 편미분 값이 너무 클 수 있기 때문에 보정계수 η 를 곱하는데, 이를 “학습률”이라고 합니다.

경사 하강법은 아래와 같은 새로운 w_1 , 새로운 w_0 를 반복적으로 업데이트하면서 비용 함수가 최소가 되는 값을 찾습니다.

$$\text{새로운 } w_1 = \text{이전 } w_1 - \eta \frac{2}{N} \sum_{i=1}^N x_i * (\text{실제값}_i - \text{예측값}_i)$$

$$\text{새로운 } w_0 = \text{이전 } w_0 - \eta \frac{2}{N} \sum_{i=1}^N (\text{실제값}_i - \text{예측값}_i)$$

비용 함수가 최소가 되는 값을 찾게 되면, 그 때가 최적의 w_1, w_0 가 된다.

경사하강법 수행 프로세스

- Step 1: w_1, w_0 를 임의의 값으로 설정하고 첫 비용 함수의 값을 계산합니다.
- Step 2: w_1 을 $w_1 - \eta \frac{2}{N} \sum_{i=1}^N x_i * (\text{실제값}_i - \text{예측값}_i)$, w_0 을 $w_0 - \eta \frac{2}{N} \sum_{i=1}^N (\text{실제값}_i - \text{예측값}_i)$ 으로 업데이트한 후 다시 비용 함수의 값을 계산합니다.
- Step 3: 비용 함수의 값이 감소했으면 다시 Step 2를 반복합니다. 더 이상 비용 함수의 값이 감소하지 않으면 그때의 w_1, w_0 를 구하고 반복을 중지합니다.

Implement of easy instance

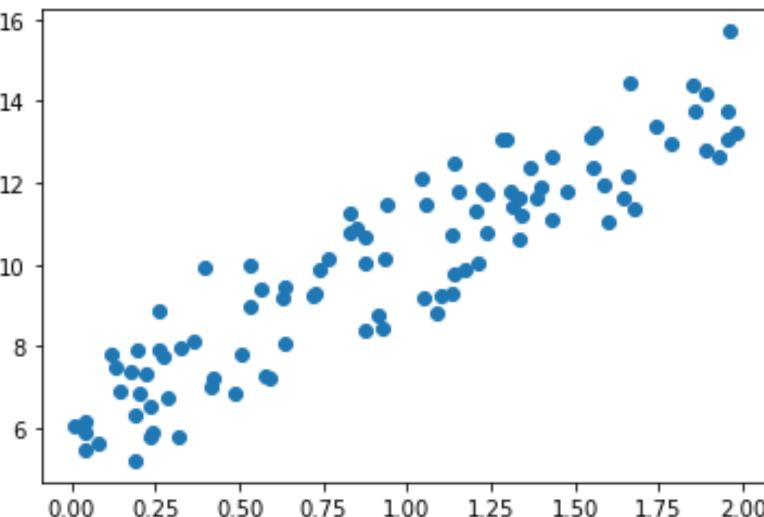
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

np.random.seed(0)
#  $y = 4X + 6$  식을 근사( $w1=4$ ,  $w0=6$ ). random 값은 Noise를 위해 만듬
X = 2 * np.random.rand(100,1)
y = 6 +4 * X+ np.random.randn(100,1)

# X, y 데이터 셋 scatter plot으로 시각화
plt.scatter(X, y)
```

X.shape, y.shape

((100, 1), (100, 1))



코딩으로 경사하강법 구현

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
np.random.seed(0)

#  $y = 4X + 6$  식을 근사(w1=4, w0=6). random 값은 Noise를 위해 만듬
X = 2*np.random.rand(100, 1)
y = 6 + 4*X + np.random.randn(100, 1)
```

```
print(len(X))
X
```

100

```
array([[1.09762701],
       [1.43037873],
       [1.20552675],
       [1.08976637],
       [0.8473096 ],
       [1.29178823],
       [0.87517442],
```

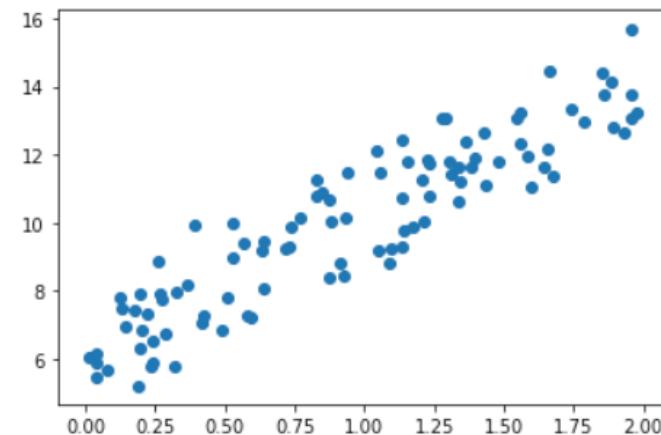
```
print(len(y))
y
```

100

```
array([[ 9.22535819],
       [12.62234142],
       [11.28776945],
       [ 8.82282178],
       [10.87749059],
       [13.06304208],
       [10.67947726],
```

```
# X, y 데이터 셋 scatter plot으로 시각화
plt.scatter(X, y)
```

```
<matplotlib.collections.PathCollection at 0x7f
```



코딩으로 경사하강법 구현

```
# w1 과 w0 를 업데이트 할 w1_update, w0_update를 반환.
def get_weight_updates(w1, w0, X, y, learning_rate=0.01):
    N = len(y)
    # 먼저 w1_update, w0_update를 각각 w1, w0의 shape와 동일한 크기를 가진 0 값으로 초기화
    w1_update = np.zeros_like(w1)
    w0_update = np.zeros_like(w0)
    # 예측 배열 계산하고 예측과 실제 값의 차이 계산
    y_pred = np.dot(X, w1.T) + w0
    diff = y - y_pred

    # w0_update를 dot 행렬 연산으로 구하기 위해 모두 1값을 가진 행렬 생성
    w0_factors = np.ones((N,1))

    # w1과 w0를 업데이트할 w1_update와 w0_update 계산
    w1_update = -(2/N)*learning_rate*(np.dot(X.T, diff))
    w0_update = -(2/N)*learning_rate*(np.dot(w0_factors.T, diff))

    return w1_update, w0_update
```

```
# 입력 인자 iters로 주어진 횟수만큼 반복적으로 w1과 w0를 업데이트 적용함.
def gradient_descent_steps(X, y, iters=10000):
    # w0와 w1을 모두 0으로 초기화.
    w0 = np.zeros((1,1))
    w1 = np.zeros((1,1))

    # 인자로 주어진 iters 만큼 반복적으로 get_weight_updates() 호출하여 w1, w0 업데이트 수행.
    for ind in range(iters):
        w1_update, w0_update = get_weight_updates(w1, w0, X, y, learning_rate=0.01)
        w1 = w1 - w1_update
        w0 = w0 - w0_update

    return w1, w0
```

```
def get_cost(y, y_pred):
    N = len(y)
    cost = np.sum(np.square(y - y_pred))/N  # 오차값
    return cost

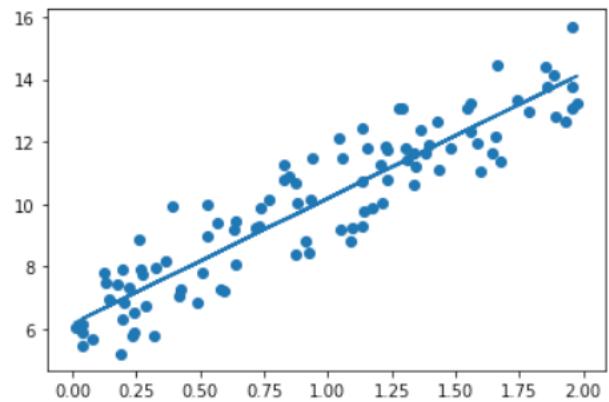
w1, w0 = gradient_descent_steps(X, y, iters=1000)
print("w1:{0:.3f} w0:{1:.3f}".format(w1[0,0], w0[0,0]))
y_pred = w1[0,0]*X + w0
print('Gradient Descent Total Cost:{0:.4f}'.format(get_cost(y, y_pred)))
```

```
w1:4.022 w0:6.162
```

```
Gradient Descent Total Cost:0.9935
```

```
plt.scatter(X, y)
plt.plot(X, y_pred)
```

```
[<matplotlib.lines.Line2D at 0x7fe8b862d150>]
```



미니 배치 확률적 경사 하강법

```
# 샘플링으로 데이터를 추출하여 시간 단축
def stochastic_gradient_descent_steps(X, y, batch_size=10, iters=1000):
    w0 = np.zeros((1,1))
    w1 = np.zeros((1,1))
    prev_cost = 100000
    iter_index = 0

    for ind in range(iters):
        np.random.seed(ind)
        # 전체 x, y 데이터에서 랜덤하게 batch_size만큼 데이터 추출하여 sample_x, sample_y로 저장
        stochastic_random_index = np.random.permutation(X.shape[0])
        sample_X = X[stochastic_random_index[0:batch_size]]
        sample_y = y[stochastic_random_index[0:batch_size]]
        # 랜덤하게 batch_size만큼 추출된 데이터 기반으로 w1_update, w0_update 계산 후 업데이트
        w1_update, w0_update = get_weight_updates(w1, w0, sample_X, sample_y, learning_rate=0.01)
        w1 = w1 - w1_update
        w0 = w0 - w0_update

    return w1, w0
```

```
np.random.permutation(X.shape[0])
```

```
array([53, 70, 30, 34, 84, 7, 62, 27, 52, 86, 56, 55, 59, 93, 51, 94, 22,
       49, 39, 58, 3, 74, 99, 73, 40, 57, 76, 45, 64, 32, 43, 33, 92, 68,
       82, 18, 71, 90, 60, 17, 63, 38, 36, 28, 46, 96, 54, 88, 21, 20, 98,
       24, 29, 47, 75, 0, 42, 97, 26, 65, 41, 44, 78, 15, 10, 87, 91, 31,
       13, 9, 81, 67, 19, 14, 72, 5, 50, 25, 8, 95, 61, 77, 69, 85, 23,
       16, 48, 80, 83, 66, 2, 79, 11, 89, 37, 6, 1, 4, 12, 35])
```

```
w1, w0 = stochastic_gradient_descent_steps(X, y, iters=1000)
print("w1:", round(w1[0,0],3), "w0:", round(w0[0,0],3))
y_pred = w1[0,0] * X + w0
print('Stochastic Gradient Descent Total Cost:{0:.4f}'.format(get_cost(y, y_pred)))
```

```
w1: 4.028 w0: 6.156
Stochastic Gradient Descent Total Cost:0.9937
```

사이킷런 Linear Regression 클래스

LinearRegression 클래스

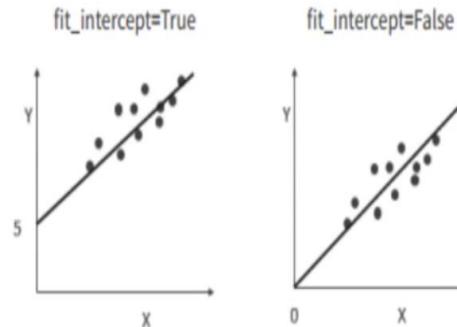
```
class sklearn.linear_model.LinearRegression(fit_intercept=True, normalize=False, copy_X=True, n_jobs=1)
```

LinearRegression 클래스는 예측값과 실제 값의 RSS(Residual Sum of Squares)를 최소화해 OLS(Ordinary Least Squares) 추정 방식으로 구현한 클래스입니다.

LinearRegression 클래스는 fit() 메서드로 X, y 배열을 입력 받으면 회귀 계수(Coefficients)인 W를 coef_ 속성에 저장합니다.

fit_intercept: 불린 값으로, 디폴트는 True입니다. Intercept(절편) 값을 계산할 것인지 말지를 지정합니다. 만일 False로 지정하면 intercept가 사용되지 않고 0으로 지정됩니다.

입력 파라미터



normalize: 불린 값으로 디폴트는 False입니다. fit_intercept가 False인 경우에는 이 파라미터가 무시됩니다. 만일 True이면 회귀를 수행하기 전에 입력 데이터 세트를 정규화합니다.

속성

coef_: fit() 메서드를 수행했을 때 회귀 계수가 배열 형태로 저장하는 속성. Shape는 (Target 값 개수, 피처 개수).
intercept_: intercept 값

선형 회귀의 다중 공선성 문제

선형 회귀의 다중 공선성 문제

- 일반적으로 선형 회귀는 입력 피처의 독립성에 많은 영향을 받습니다. 피처간의 상관관계가 매우 높은 경우 분산이 매우 커져서 오류에 매우 민감해집니다. 이러한 현상을 다중 공선성(multi-collinearity) 문제라고 합니다. 일반적으로 상관관계가 높은 피처가 많은 경우 독립적인 중요한 피처만 남기고 제거하거나 규제를 적용합니다.

회귀의 평가 지표

평가 지표	설명	수식
MAE	Mean Absolute Error(MAE)이며 실제 값과 예측값의 차이를 절댓값으로 변환해 평균한 것입니다.	$MAE = \frac{1}{n} \sum_{i=1}^n Y_i - \hat{Y}_i $
MSE	Mean Squared Error(MSE)이며 실제 값과 예측값의 차이를 제곱해 평균한 것입니다.	$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$
RMSE	MSE 값은 오류의 제곱을 구하므로 실제 오류 평균보다 더 커지는 특성이 있으므로 MSE에 루트를 씌운 것이 RMSE(Root Mean Squared Error)입니다.	$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2}$
R ²	분산 기반으로 예측 성능을 평가합니다. 실제 값의 분산 대비 예측값의 분산 비율을 지표로 하며, 1에 가까울수록 예측 정확도가 높습니다.	$R^2 = \frac{\text{예측값 Variance}}{\text{실제값 Variance}}$

사이킷런 회귀 평가 API

- 사이킷런은 아쉽게도 RMSE를 제공하지 않습니다. RMSE를 구하기 위해서는 MSE에 제곱근을 씌워서 계산하는 함수를 직접 만들어야 합니다.
- 다음은 각 평가 방법에 대한 사이킷런의 API 및 cross_val_score나 GridSearchCV에서 평가 시 사용되는 scoring 파라미터의 적용 값입니다

평가 방법	사이킷런 평가 지표 API	Scoring 함수 적용 값
MAE	metrics.mean_absolute_error	'neg_mean_absolute_error'
MSE	metrics.mean_squared_error	'neg_mean_squared_error'
R ²	metrics.r2_score	'r2'

사이킷런 Scoring 함수 회귀 평가 적용시 유의 사항

cross_val_score, GridSearchCV와 같은 Scoring 함수에 회귀 평가 지표를 적용 시 유의 사항

- MAE의 사이킷런 scoring 파라미터 값은 'neg_mean_absolute_error' 입니다. 이는 Negative(음수) 값을 가진다는 의미인데, MAE는 절댓값의 합이기 때문에 음수가 될 수 없습니다.
- Scoring 함수에 'neg_mean_absolute_error'를 적용해 음수값을 반환하는 이유는 사이킷런의 Scoring 함수가 score값이 클수록 좋은 평가 결과로 자동 평가하기 때문입니다. 따라서 -1을 원래의 평가 지표 값에 곱해서 음수(Negative)를 만들어 작은 오류 값이 더 큰 숫자로 인식하게 합니다. 예를 들어 $10 > 1$ 이지만 음수를 곱하면 $-1 > -10$ 이 됩니다.
- metrics.mean_absolute_error()와 같은 사이킷런 평가 지표 API는 정상적으로 양수의 값을 반환합니다. 하지만 Scoring 함수의 scoring 파라미터 값 'neg_mean_absolute_error'가 의미하는 것은 $-1 * \text{metrics.mean_absolute_error}()$ 이니 주의가 필요합니다.

Retreat _선형 회귀 예제

- 인간의 키와 몸무게는 어느 정도 비례할 것으로 예상된다. 아래와 같은 데이터가 있을 때, 선형 회귀를 이용하여 학습시키고 키가 165cm일 때의 예측값을 얻어보자.



```
import matplotlib.pyplot as plt
from sklearn import linear_model

reg = linear_model.LinearRegression()

X = [[174], [152], [138], [128], [186]]
y = [71, 55, 46, 38, 88]
reg.fit(X, y) # 학습

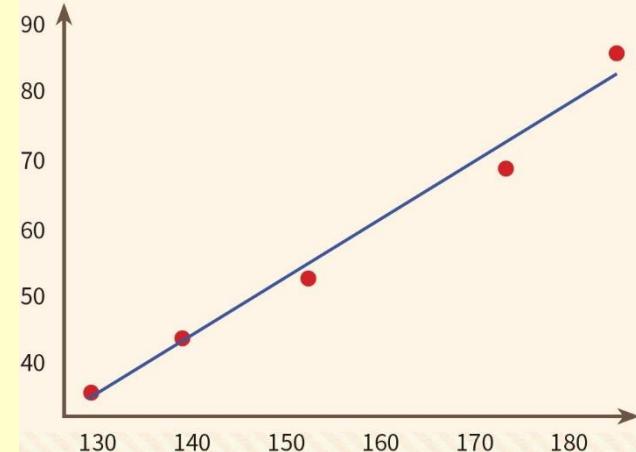
print(reg.predict([[165]]))
```

학습 데이터와 y 값을 산포도로 그린다.
plt.scatter(X, y, color='black')

학습 데이터를 입력으로 하여 예측값을 계산한다.
y_pred = reg.predict(X)

학습 데이터와 예측값으로 선그래프로 그린다.
계산된 기울기와 y 절편을 가지는 직선이 그려진다.
plt.plot(X, y_pred, color='blue', linewidth=3)
plt.show()

[67.30998637]



당뇨병 예제

- sklearn 라이브러리에는 당뇨병 환자들의 데이터가 기본적으로 포함되어 있다.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn import datasets

# 당뇨병 데이터 세트를 적재한다.
diabetes = datasets.load_diabetes()

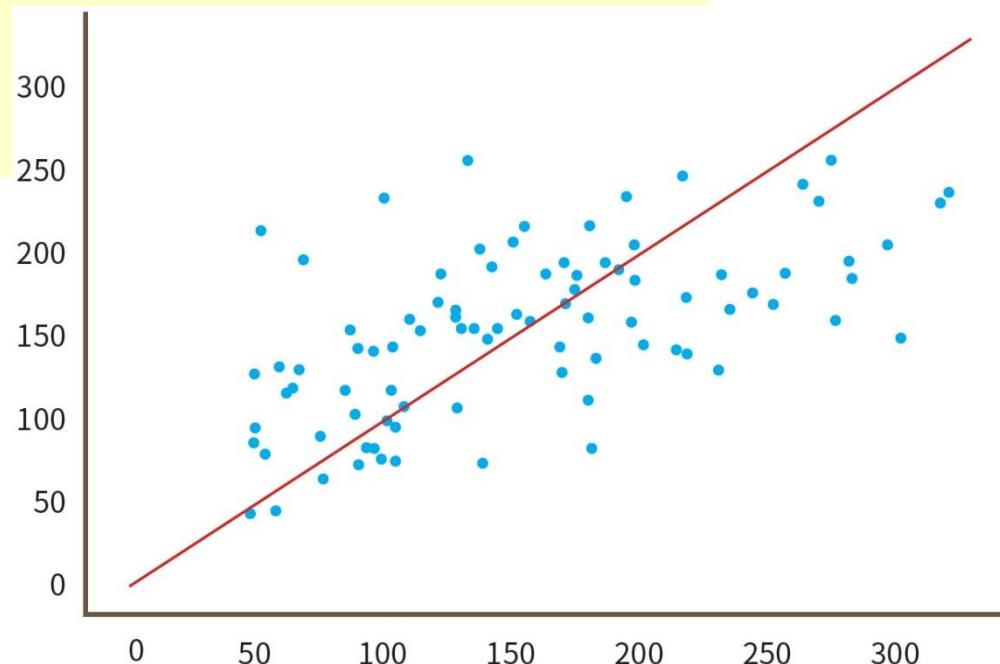
# 학습 데이터와 테스트 데이터를 분리한다.
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(diabetes.data, diabetes.target,
test_size=0.2, random_state=0)

선형 회귀 모델로 학습을 수행한다.
model = LinearRegression()
model.fit(X_train, y_train)
```

```
# 테스트 데이터로 예측해보자.  
y_pred = model.predict(X_test)
```

```
# 실제 데이터와 예측 데이터를 비교해보자.  
plt.plot(y_test, y_pred, '.')
```

```
# 직선을 그리기 위하여 완벽한 선형 데이터를 생성한다.  
x = np.linspace(0, 330, 100)  
y = x  
plt.plot(x, y)  
plt.show()
```



LinearRegression 이용한 보스턴 주택가격 예측

1. 데이터 로드 및 확인

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from scipy import stats
from sklearn.datasets import load_boston
%matplotlib inline

# boston 데이터셋 로드
boston = load_boston()

# boston 데이터셋 DataFrame 변환
bostonDF = pd.DataFrame(boston.data, columns = boston.feature_names)

# boston dataset의 target array는 주택 가격. PRICE 컬럼으로 DataFrame에 존재
bostonDF['PRICE'] = boston.target

print(bostonDF.shape)
bostonDF.head()

(506, 14)
```

속성별 설명

- CRIM: 지역별 범죄 발생률
- ZN: 25,000평방피트를 초과하는 거주 지역의 비율
- INDUS: 비상업 지역 넓이 비율
- CHAS: 찰스강에 대한 더미 변수(강의 경계에 위치한 경우는 1, 아니면 0)
- NOX: 일산화질소 농도
- RM: 거주할 수 있는 방 개수
- AGE: 1940년 이전에 건축된 소유 주택의 비율
- DIS: 5개 주요 고용센터까지의 가중 거리
- RAD: 고속도로 접근 용이도
- TAX: 10,000달러당 재산세율
- PTRATIO: 지역의 교사와 학생 수 비율
- B: 지역의 흑인 거주 비율
- LSTAT: 하위 계층의 비율
- MEDV: 본인 소유의 주택 가격(중앙값)

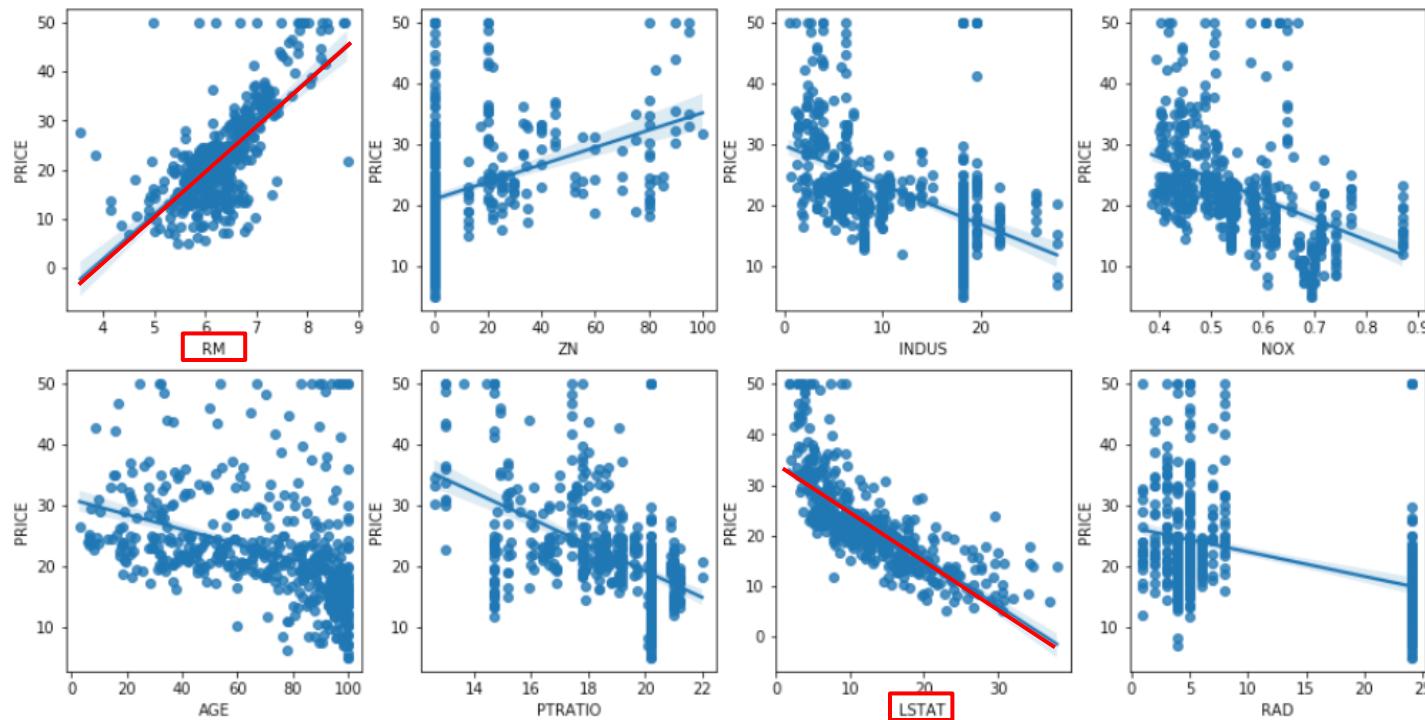
예측하고자 하는 종속 변수
: 주택 가격

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

LinearRegression 이용한 보스턴 주택가격 예측

2. 피처/타겟값 상관관계 파악 - seaborn의 regplot 이용하면 산점도와 선형 회귀 직선을 함께 나타내준다.

```
# 2개의 행과 4개의 열을 가진 subplots를 이용. axes는 4x2개의 ax를 가짐.
fig, axes = plt.subplots(figsize=(16,8) , ncols=4 , nrows=2)
lm_features = ['RM','ZN','INDUS','NOX','AGE','PTRATIO','LSTAT','RAD']
for i , feature in enumerate(lm_features):
    row = int(i/4)
    col = i%4
    # 기본의 regplot을 이용해 산점도와 선형 회귀 직선을 함께 표현
    sns.regplot(x=feature , y='PRICE',data=bostonDF , ax=axes[row][col])
```



LinearRegression 이용한 보스턴 주택가격 예측

3. 학습/테스트 데이터 분리하고 Linear Regression 학습/예측/평가 수행

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

y_target = bostonDF['PRICE']
X_data = bostonDF.drop(['PRICE'], axis=1, inplace=False)

X_train, X_test, y_train, y_test = train_test_split(X_data, y_target, test_size=0.3, random_state=156)

# Linear Regression OLS로 학습/예측/평가 수행.
lr = LinearRegression()
lr.fit(X_train, y_train)
y_preds = lr.predict(X_test)
mse = mean_squared_error(y_test, y_preds)
rmse = np.sqrt(mse)

print('MSE : {0:.3f} , RMSE : {1:.3F}'.format(mse , rmse))
print('Variance score : {0:.3f}'.format(r2_score(y_test, y_preds)))
```

```
MSE : 17.297 , RMSE : 4.159
Variance score : 0.757
```

```
print('절편 값:', lr.intercept_)
print('회귀 계수값:', np.round(lr.coef_, 1))
```

```
절편 값: 40.995595172164336
회귀 계수값: [ -0.1   0.1   0.    3.   -19.8   3.4    0.   -1.7   0.4   -0.   -0.9   0.
 -0.6 ]
```

LinearRegression 이용한 보스턴 주택가격 예측

회귀 계수(모델에 영향을 미치는 정도)가 큰 순서대로 정렬해보면,

```
# 회귀 계수를 큰 값 순으로 정렬하기 위해 Series로 생성. index가 컬럼명에 유의
coeff = pd.Series(data=np.round(lr.coef_, 1), index=X_data.columns)
coeff.sort_values(ascending=False)
```

```
RM           3.4
CHAS          3.0
RAD           0.4
ZN            0.1
B             0.0
TAX          -0.0
AGE           0.0
INDUS          0.0
CRIM          -0.1
LSTAT          -0.6
PTRATIO        -0.9
DIS            -1.7
NOX          -19.8
dtype: float64
```

NOX 회귀 계수 값이 매우 크다.

-> NOX값에 따라서 예측 오류 값이 많은 영향을 받을 것으로 예상됨

LinearRegression 이용한 보스턴 주택가격 예측

5 Fold 셋으로 교차 검증을 수행하여 MSE, RMSE 구하기

```
from sklearn.model_selection import cross_val_score

y_target = bostonDF['PRICE']
X_data = bostonDF.drop(['PRICE'],axis=1,inplace=False)
lr = LinearRegression()

# cross_val_score( )로 5 Fold 셋으로 MSE 를 구한 뒤 이를 기반으로 다시 RMSE 구함.
neg_mse_scores = cross_val_score(lr, X_data, y_target, scoring="neg_mean_squared_error", cv = 5)
rmse_scores = np.sqrt(-1 * neg_mse_scores)
avg_rmse = np.mean(rmse_scores)

# cross_val_score(scoring="neg_mean_squared_error")로 반환된 값은 모두 음수
print(' 5 folds 의 개별 Negative MSE scores: ', np.round(neg_mse_scores, 2))
print(' 5 folds 의 개별 RMSE scores : ', np.round(rmse_scores, 2))
print(' 5 folds 의 평균 RMSE : {0:.3f} '.format(avg_rmse))

5 folds 의 개별 Negative MSE scores:  [-12.46 -26.05 -33.07 -80.76 -33.31]
5 folds 의 개별 RMSE scores :  [3.53 5.1 5.75 8.99 5.77]
5 folds 의 평균 RMSE : 5.829
```

다항 선형 회귀

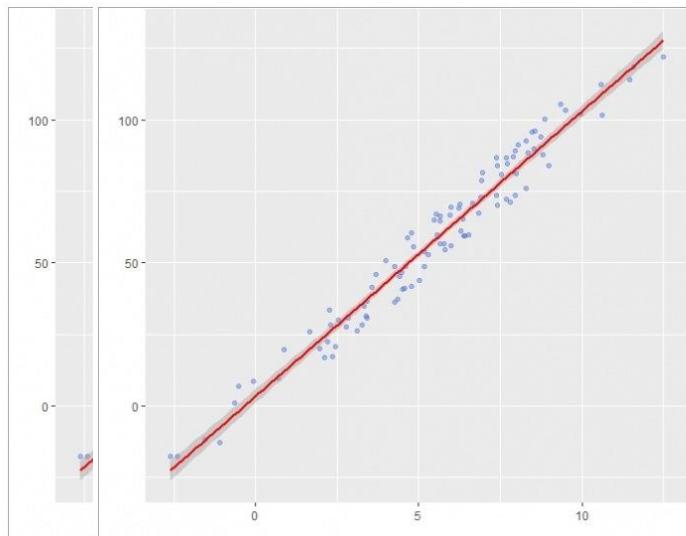
다중 회귀 vs 단항 회귀

지금까지 우리가 살펴본 회귀는

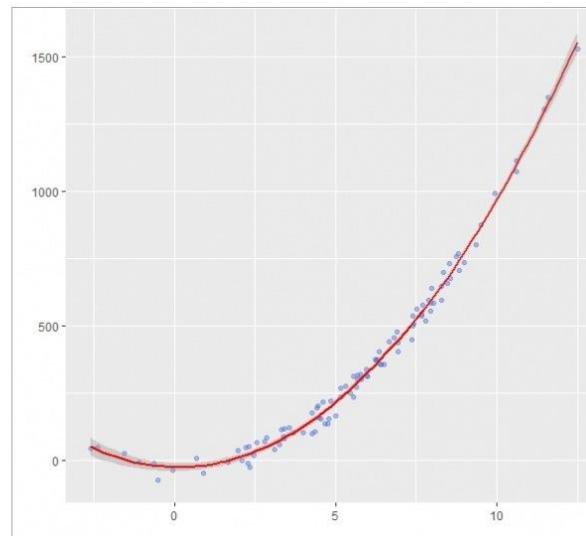
다중 선형 회귀(multiple linear regression) : 일차 방정식

$$y = w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n$$

하지만 세상 모든 관계는 직선으로만 표현할 수 없습니다.



다중 선형 회귀



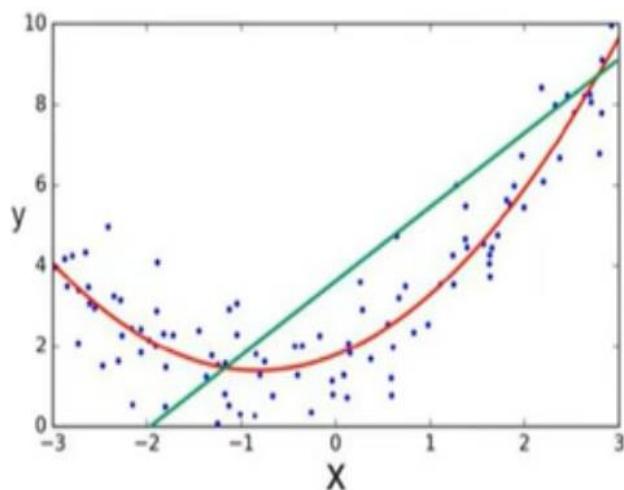
단항 선형 회귀

단항 선형 회귀(polynomial linear regression) : 2차, 3차 방정식

$$y = w_0 + w_1w_1 + w_2x_2 + w_3x_1x_2 + w_4x_1^2 + w_5x_2^2$$

다항 회귀 개요 (Polynomial Regression)

다항 회귀는 $y = w_0 + w_1*x_1 + w_2*x_2 + w_3*x_1*x_2 + w_4*x_1^2 + w_5*x_2^2$ 과 같이 회귀식이 독립변수의 단항식이 아닌 2차, 3차 방정식과 같은 다항식으로 표현되는 것을 지칭합니다.



데이터 세트에 대해서 피처 X에 대해 Target Y 값의 관계를 단순 선형 회귀
직선형으로 표현한 것보다 다항 회귀 곡선형으로 표현한 것이 더 예측 성능이
높습니다.

선형회귀와 비선형 회귀의 구분

선형 회귀

$$Y = w_0 + w_1 * x_1 + w_2 * x_2 + w_3 * x_1 * x_2 + w_4 * x_1^2 + w_5 * x_2^2$$

새로운 변수인 z 를 $z = [x_1, x_2, x_1 * x_2, x_1^2, x_2^2]$ 로 한다면

$$Y = w_0 + w_1 * z_1 + w_2 * z_2 + w_3 * z_3 + w_4 * z_4 + w_5 * z_5$$

비선형 회귀

$$Y = w_1 * \cos(X + w_4) + w_2 * \cos(2 * X + w_4) + w_3$$

$$Y = w_1 * X^{w_2}$$

다항 회귀는 선형 회귀입니다. 회귀에서 선형 회귀/비선형 회귀를 나누는 기준은 회귀 계수가 선형/비선형인지에 따른 것이지 독립변수의 선형/비선형 여부는 무관합니다

사이킷런에서의 다항회귀

사이킷런은 다항회귀를 바로 API로 제공하지 않습니다.

대신 `PolynomialFeatures` 클래스로 원본 단항 피처들을 다항 피처들로 변환한 데이터 세트에 `LinearRegression` 객체를 적용하여 다항회귀 기능을 제공합니다.

PolynomialFeatures

원본 피처 데이터 세트를 기반으로 `degree` 차수에 따른 다항식을 적용하여 새로운 피처들을 생성하는 클래스
피처 엔지니어링의 기법중의 하나임.

단항 피처 $[x_1, x_2]$ 를 `Degree = 2`, 즉 2차 다항 피처로 변환한다면?

$(x_1 + x_2)^2$ 의 식 전개에 대응되는 $[1, x_1, x_2, x_1x_2, x_1^2, x_2^2]$ 의

다항 피처들로 변환



단항 피처 $[x_1, x_2]$ 를 `Degree = 3`, 즉 3차 다항 피처로 변환한다면?

$(x_1 + x_2)^3$ 의 식 전개에 대응되는

$[1, x_1, x_2, x_1x_2, x_1^2, x_2^2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^3]$ 의 다항 피처들로 변환

1차 단항 피처들의 값이 $[x_1, x_2] = [0 1]$ 일 경우

2차 다항 피처들의 값은 $[1, x_1 = 0, x_2 = 1, x_1x_2 = 0, x_1^2 = 0, x_2^2 = 1]$

형태인 $[1, 0, 1, 0, 0, 1]$ 로 변환

사이킷런에서의 다항회귀

사이킷런은 다항회귀를 바로 API로 제공하지 않습니다.

대신 `PolynomialFeatures` 클래스로 원본 단항 피처들을 다항 피처들로 변환한 데이터 세트에 `LinearRegression` 객체를 적용하여 다항회귀 기능을 제공합니다.

PolynomialFeatures
변환

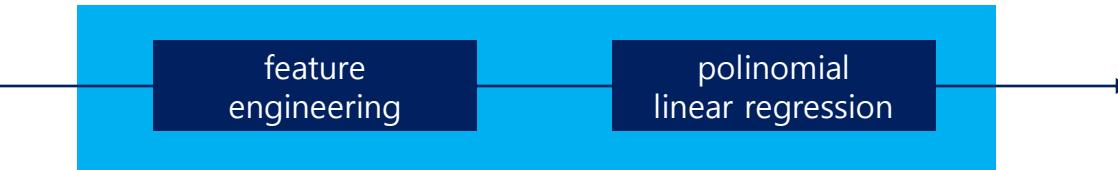
LinearRegression 학습

단항 피처 $[x_1, x_2]$ 를 2차 다항 피처 $[1, x_1, x_2, x_1x_2, x_1^2, x_2^2]$ 로 변경
(degree=2로 가정)

PolynomialFeatures로 변환된 X 피처들을 LinearRegression
객체로 학습

사이킷런에서는 일반적으로 `Pipeline` 클래스를 이용하여
`PolynomialFeatures` 변환과 `LinearRegression` 학습/예측을 결합하여 다항 회귀를 구현합니다.

pipe line



실습 : 다항 회귀

1차 단항 피처들의 값이 $[x_1, x_2] = [0, 1]$ 일 경우

2차 다항 피처들의 값은 $[1, x_1 = 0, x_2 = 1, x_1x_2 = 0, x_1^2 = 0, x_2^2 = 1]$
형태인 $[1, 0, 1, 0, 0, 1]$ 로 변환

```
from sklearn.preprocessing import PolynomialFeatures
import numpy as np

# 다항식으로 변환한 단항식 생성, [[0,1],[2,3]]의 2x2 행렬 생성
X = np.arange(4).reshape(2,2)
print('일차 단항식 계수 feature:\n', X)
```

일차 단항식 계수 feature:

```
[[0 1]
 [2 3]]
```

```
# degree = 2 인 2차 다항식으로 변환하기 위해 PolynomialFeatures를 이용하여 변환
poly = PolynomialFeatures(degree=2)
poly.fit(X)
poly_ftr = poly.transform(X)
print('변환된 2차 다항식 계수 feature:\n', poly_ftr)
```

변환된 2차 다항식 계수 feature:

```
[[1. 0. 1. 0. 0. 1.]
 [1. 2. 3. 4. 6. 9.]]
```

실습 : 다항 회귀

Linear Regression에 3차 다항식 계수 feature와 3차 다항식 결정값으로 학습 후 회귀 계수 확인

```
def polynomial_func(X):
    y = 1 + 2*X + X**2 + X**3
    return y

X = np.arange(4).reshape(2,2)
print('일차 단항식 계수 feature: \n', X)
y = polynomial_func(X)
print('삼차 다항식 결정값: \n', y)
```

일차 단항식 계수 feature:

```
[[0 1]
 [2 3]]
```

삼차 다항식 결정값:

```
[[ 1  5]
 [17 43]]
```

```
from sklearn.linear_model import LinearRegression
```

```
# 3 차 다항식 변환
poly_ftr = PolynomialFeatures(degree=3).fit_transform(X)
print('3차 다항식 계수 feature: \n', poly_ftr)
```

Linear Regression에 3차 다항식 계수 feature와 3차 다항식 결정값으로 학습 후 회귀 계수 확인

```
model = LinearRegression()
model.fit(poly_ftr, y)
print('Polynomial 회귀 계수\n', np.round(model.coef_, 2))
print('Polynomial 회귀 Shape :', model.coef_.shape)
```

3차 다항식 계수 feature:

```
[[ 1.  0.  1.  0.  0.  1.  0.  0.  0.  1.]
 [ 1.  2.  3.  4.  6.  9.  8. 12. 18. 27.]]
```

Polynomial 회귀 계수

```
[[0.    0.02 0.02 0.05 0.07 0.1  0.1  0.14 0.22 0.31]
 [0.    0.06 0.06 0.11 0.17 0.23 0.23 0.34 0.51 0.74]]
```

Polynomial 회귀 Shape : (2, 10)

실습 : 파이프라인 이용 다항 회귀

파이프라인을 이용해서 3차 다항 회귀 실습

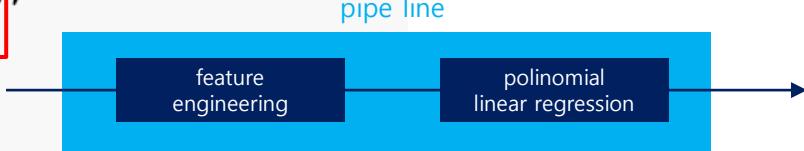
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
import numpy as np

def polynomial_func(X):
    y = 1 + 2*X + X**2 + X**3
    return y

# Pipeline 객체로 Streamline하게 Polynomial Feature변환과 Linear Regression을 연결
model = Pipeline([('poly', PolynomialFeatures(degree=3)),
                  ('linear', LinearRegression())])

X = np.arange(4).reshape(2,2)
y = polynomial_func(X)

model = model.fit(X, y)
print('Polynomial 회귀 계수\n', np.round(model.named_steps['linear'].coef_, 2))
```



Polynomial 회귀 계수

```
[[0.    0.02 0.02 0.05 0.07 0.1   0.1   0.14 0.22 0.31]
 [0.    0.06 0.06 0.11 0.17 0.23 0.23 0.34 0.51 0.74]]
```

실습 : 다항 회귀로 보스턴 주택 가격 예측

다항회귀 이용해서 보스턴 주택가격 예측

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
import numpy as np

# boston 데이터셋 로드
boston = load_boston()

# boston 데이터셋 DataFrame 변환
bostonDF = pd.DataFrame(boston.data, columns = boston.feature_names)

# boston dataset의 target array는 주택 가격. PRICE 컬럼으로 DataFrame에 존재
bostonDF['PRICE'] = boston.target

print(bostonDF.shape)
bostonDF.head()

(506, 14)
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

```
y_target = bostonDF['PRICE']
X_data = bostonDF.drop(['PRICE'], axis=1, inplace=False)

X_train, X_test, y_train, y_test = train_test_split(
    X_data, y_target, test_size=0.3, random_state=156)
```

```
# Pipeline 객체로 Streamline 하기 Polynomial Feature변환과 Linear Regression을 연결
p_model = Pipeline([('poly', PolynomialFeatures(degree=2, include_bias=False)),
                    ('linear', LinearRegression())))
```

```
p_model
```

```
Pipeline(memory=None,
          steps=[('poly',
                  PolynomialFeatures(degree=2, include_bias=False,
                                     interaction_only=False, order='C')),
                 ('linear',
                  LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                                     normalize=False))],
          verbose=False)
```

```
p_model.fit(X_train, y_train)
y_preds = p_model.predict(X_test)
mse = mean_squared_error(y_test, y_preds)
rmse = np.sqrt(mse)

print('MSE : {:.3f}, RMSE : {:.3f}'.format(mse, rmse))
print('Variance score : {:.3f}'.format(r2_score(y_test, y_preds)))

MSE : 15.556, RMSE : 3.944
Variance score : 0.782
```

파이프라인
구축

일반 회귀와 비슷한 RMSE값이 나온다

실습 : 다항 회귀로 보스턴 주택 가격 예측

다항 회귀에서 degree 수를 높일수록 오버피팅 될 수 있다는 점은 주의해야 한다.

```
# Pipeline 객체로 Streamline 하기 Polynomial Feature변환과 Linear Regression을 연결
p_model = Pipeline([('poly', PolynomialFeatures(degree=3, include_bias=False)),
                    ('linear', LinearRegression()))])
p_model
```

다항 회귀에서 차수(degree)를 높이면 오버피팅될 우려가 있다

```
Pipeline(memory=None,
         steps=[('poly',
                  PolynomialFeatures(degree=3, include_bias=False,
                                     interaction_only=False, order='C')),
                ('linear',
                  LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                                    normalize=False))],
         verbose=False)
```

```
p_model.fit(X_train, y_train)
y_preds = p_model.predict(X_test)
mse = mean_squared_error(y_test, y_preds)
rmse = np.sqrt(mse)

print('MSE : {:.3f}, RMSE : {:.3f}'.format(mse, rmse))
print('Variance score : {:.3f}'.format(r2_score(y_test, y_preds)))
```

MSE : 79625.593, RMSE : 282.180
Variance score : -1116.598

오버피팅이 된다.

```
# degree=2로 변환된 다항 회귀의 피처들을 살펴보면, 기존의 피처 13개를 조합해서 피처가 104개로 늘어난 것을 확인할 수 있다.
# 다항식에서 degree가 높아지면 오버피팅이 일어날 수 있다
X_train_poly = PolynomialFeatures(degree=2, include_bias=False).fit_transform(X_train, y_train)
print(X_train_poly.shape, X_train.shape)
```

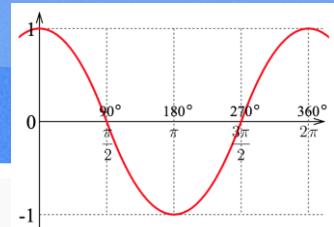
(354, 104) (354, 13)

Underfitting vs Overfitting



Install User Guide API Examples More ▾

코사인 곡선 시뮬레이션 case



Prev Up Next

scikit-learn 0.23.2

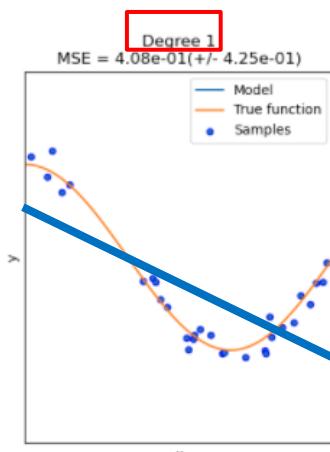
Other versions

Please [cite us](#) if you use the software.

Underfitting vs. Overfitting

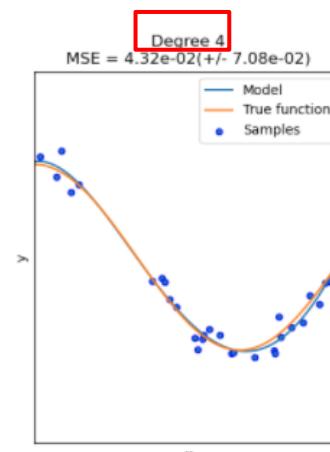
Underfitting vs. Overfitting

This example demonstrates the problems of underfitting and overfitting and how we can use linear regression with polynomial features to approximate nonlinear functions. The plot shows the function that we want to approximate, which is a part of the cosine function. In addition, the samples from the real function and the approximations of different models are displayed. The models have polynomial features of different degrees. We can see that a linear function (polynomial with degree 1) is not sufficient to fit the training samples. This is called **underfitting**. A polynomial of degree 4 approximates the true function almost perfectly. However, for higher degrees the model will **overfit** the training data, i.e. it learns the noise of the training data. We evaluate quantitatively **overfitting / underfitting** by using cross-validation. We calculate the mean squared error (MSE) on the validation set, the higher, the less likely the model generalizes correctly from the training data.



MSE=0.408

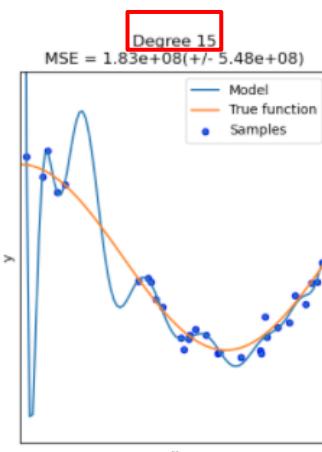
다항 회귀에서 차수(degree)를 높였을 때
오버피팅될 우려가 있다



MSE=0.0432

언더피팅이 된 상태

너무 심플하게
모델을 만들



MSE=1.83⁸

코사인 곡선과 상당히 유사
하게 선이 그어짐

최적의 모델

모든 점을 예측하여서
오버피팅이 된 상태

너무 복잡하게
모델을 만들

실습 : 다항 회귀의 차수(degree)를 변화시키면서 회귀 계수와 MSE값 구해보고 추세선 그리기

```
# noise값이 추가된 코사인 시뮬레이션 점 찍기
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
%matplotlib inline

# random 값으로 구성된 x값에 대해 cosine 변환값을 반환.
def true_fun(X):
    return np.cos(1.5 * np.pi*X)

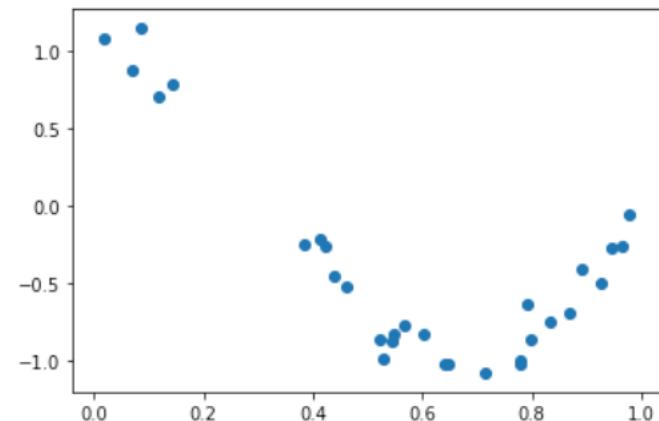
# X는 0 부터 1까지 30개의 random 값을 순서대로 sampling 한 데이터 입니다.
np.random.seed(0)
n_samples = 30
X = np.sort(np.random.rand(n_samples))

# y 값은 cosine 기반의 true_fun()에서 약간의 Noise 변동값을 더한 값입니다.
y = true_fun(X) + np.random.randn(n_samples)*0.1
y

array([ 1.0819082 ,  0.87027612,  1.14386208,  0.70322051,  0.78494746,
       -0.25265944, -0.22066063, -0.26595867, -0.4562644 , -0.53001927,
       -0.86481449, -0.99462675, -0.87458603, -0.83407054, -0.77090649,
       -0.83476183, -1.03080067, -1.02544303, -1.0788268 , -1.00713288,
       -1.03009698, -0.63623922, -0.86230652, -0.75328767, -0.70023795,
       -0.41043495, -0.50486767, -0.27907117, -0.25994628, -0.06189804])
```

```
plt.scatter(X, y)
```

```
<matplotlib.collections.PathCollection at 0x7f...
```



실습 : 다항 회귀의 차수(degree)를 변화시키면서 회귀 계수와 MSE값 구해보고 추세선 그리기

```
# 다항 회귀의 차수(degree)를 변화시키면서 회귀 계수와 MSE값 구해보고 그레프로 나타내기
plt.figure(figsize=(14, 5))
degrees = [1, 4, 15]

# 다항 회귀의 차수(degree)를 1, 4, 15로 각각 변화시키면서 비교합니다.
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)
    plt.setp(ax, xticks=(), yticks=())

    # 개별 degree별로 Polynomial 변환합니다.
    polynomial_features = PolynomialFeatures(degree=degrees[i], include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([('polynomial_features', polynomial_features),
                        ('linear_regression', linear_regression)])
    pipeline.fit(X.reshape(-1, 1), y)

    # 교차 검증으로 다항 회귀를 평가합니다.
    scores = cross_val_score(pipeline, X.reshape(-1, 1), y, scoring="neg_mean_squared_error", cv=10)
    coefficients = pipeline.named_steps['linear_regression'].coef_
    print('\nDegree {} 회귀 계수는 {}입니다.'.format(degrees[i], np.round(coefficients, 2)))
    print('Degree {} MSE 는 {:.2f}입니다.'.format(degrees[i], -1 * np.mean(scores)))

    # 0 부터 1까지 테스트 데이터 세트를 100개로 나눠 예측을 수행합니다.
    # 테스트 데이터 세트에 회귀 예측을 수행하고 예측 곡선과 실제 곡선을 그려서 비교합니다.
    X_test = np.linspace(0, 1, 100)
    # 예측값 곡선
    plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
    # 실제 값 곡선
    plt.plot(X_test, true_fun(X_test), '--', label="True function")
    plt.scatter(X, y, edgecolor='b', s=20, label="Samples")

    plt.xlabel("x"); plt.ylabel("y"); plt.xlim((0, 1)); plt.ylim((-2, 2)); plt.legend(loc="best")
    plt.title("Degree {}\\nMSE = {:.2e}(+/- {:.2e})".format(degrees[i], -scores.mean(), scores.std()))

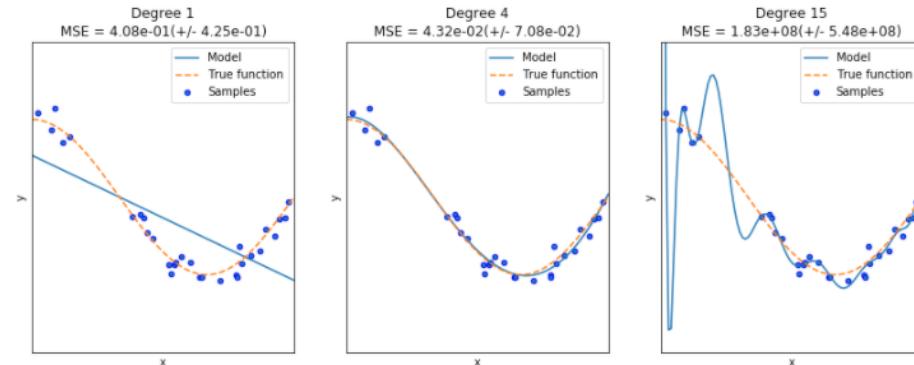
plt.show()
```

Degree 1 회귀 계수는 [-2.]입니다.
Degree 1 MSE 는 0.41입니다.

Degree 4 회귀 계수는 [0. -18. 24. -7.]입니다.
Degree 4 MSE 는 0.04입니다.

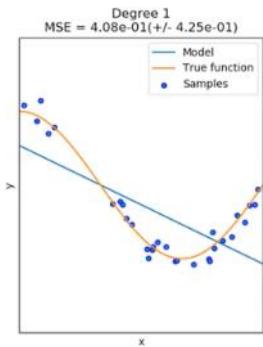
Degree 15 회귀 계수는 [-2.9830000e+03 1.0390000e+05 -1.87417100e+06 2.03717220e+07
-1.44873987e+08 7.09318780e+08 -2.47066977e+09 6.24564048e+09
-1.15677067e+10 1.56895696e+10 -1.54006776e+10 1.06457788e+10
-4.91379977e+09 1.35920330e+09 -1.70381654e+08]입니다.
Degree 15 MSE = 182815433.56입니다.

degree가 15면 회귀 계수와 오차가 너무 크다
(모델을 너무 복잡하게 만들었다)



Bias-Variance (편향 -분산)Trade off

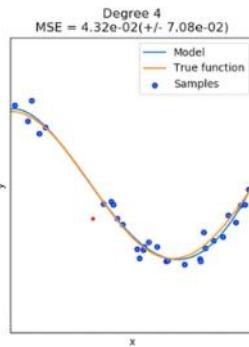
과소적합



너무 심플한 모델

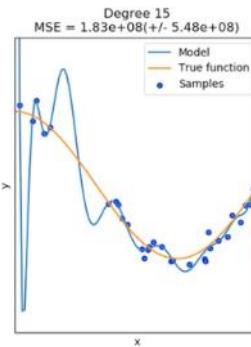
변동성 저
편향 고

과대 적합



최적의 모델
(well-fit model)

변동성 적당히 저
편향 적당히 저



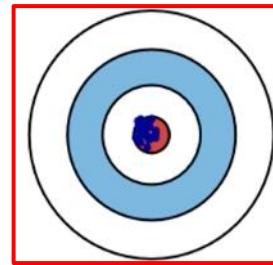
너무 복잡한 모델

변동성 고
편향 저

최적의 모델

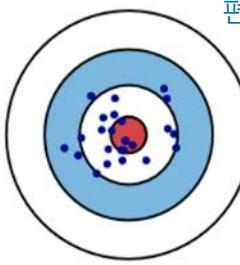
변동성 적당히 저
편향 적당히 저

Low Variance



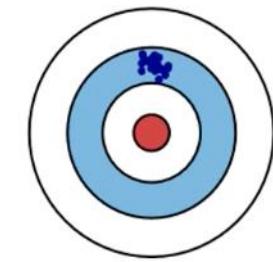
Low Bias

High Variance



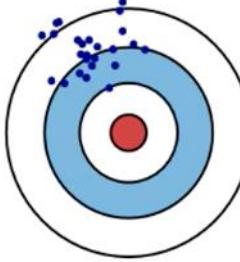
변동성 고
편향 저

High Bias



과소 적합

변동성 저
편향 고



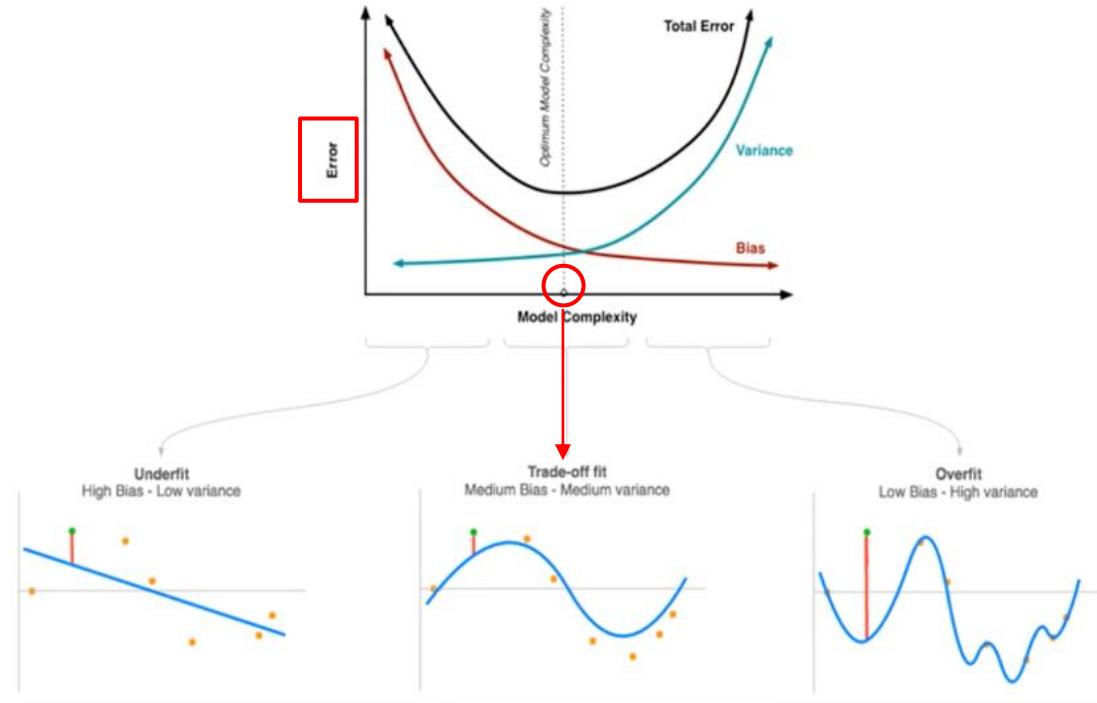
영망 모델

변동성 고
편향 고

Bias-Variance (편향 -분산)Trade off

편향이 높으면 분산은 낮아지고
분산이 높으면 편향이 낮아짐
↑ ↓
과소적합

분산이 높으면 편향이 낮아짐
↑ ↓
과대적합



최적의 편향, 분산 값을
찾으면 오차가 최소가 되는
최적의 머신러닝 모델을 만
들 수 있다

규제 선형 회귀 (릿지, 라쏘, 엘라스틱넷)

규제 선형 회귀 개요

과대 적합을 방지하기 위해서 규제는 필요하다

앞의 예제에서 $\text{Degree}=15$ 의 다항회귀는 지나치게 모든 데이터에 적합한 회귀식을 만들기 위해서 다항식이 복잡해지고 회귀 계수가 매우 크게 설정이 되면서 과대적합이 되고 평가 데이터 세트에 대해서 형편없는 예측 성능을 보였습니다. 따라서 회귀 모델은 적절히 데이터에 적합하면서도 회귀 계수가 기하급수적으로 커지는 것을 제어할 수 있어야 합니다.



최초 목표는 RSS(오차)를 최소화하는 것이었지만, 그러다보니 회귀 계수가 커져서 과대적합이라는 문제를 만나게 되었다.

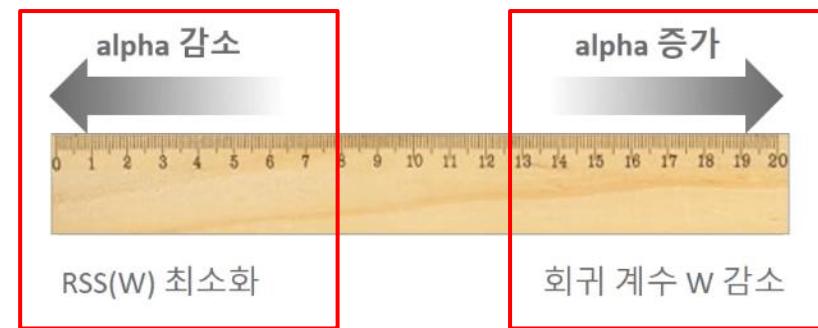
그래서 RSS와 더불어 회귀 계수 크기를 밸런스있게 제어하는 것이 필요하게 되었다.

-> 비용 함수의 목표가 밸런스 조절(RSS값 최소화, 회계 계수 값 제어)이 됨

알파값으로 비용 함수의 회귀 계수 조절

- alpha가 0(또는 매우 작은 값)이라면 비용 함수 식은 기존과 동일한 $\text{Min}(\text{RSS}(W) + 0)$ 이 될 것입니다.
- 반면에 alpha가 무한대(또는 매우 큰 값)라면 비용 함수 식은 $\text{RSS}(W)$ 에 비해 $\text{alpha} * ||W||_2^2$ 값이 너무 커지게 되므로 W 값을 0(또는 매우 작게)으로 만들어야 Cost가 최소화되는 비용 함수 목표를 달성할 수 있습니다.
- 즉, alpha 값을 크게 하면 비용 함수는 회귀 계수 W 의 값을 작게 해 과적합을 개선할 수 있으며 alpha 값을 작게 하면 회귀 계수 W 의 값이 커져도 어느 정도 상쇄가 가능하므로 학습 데이터 적합을 더 개선할 수 있습니다

$$\text{비용 함수 목표} = \text{Min} (\text{RSS}(W) + \text{alpha} * ||W||_2^2)$$



- $\text{alpha} = 0$ 인 경우는 W 가 커도 $\text{alpha} * ||W||_2^2$ 가 0이 되어 비용 함수는 $\text{Min}(\text{RSS}(W))$
- $\text{alpha} = \text{무한대인 경우 } \text{alpha} * ||W||_2^2$ 도 무한대가 되므로 비용 함수는 W 를 0에 가깝게 최소화 해야 함.

알파값으로 비용 함수의 회귀 계수 조절

- 이처럼 비용 함수에 α 값으로 페널티를 부여해 회귀 계수 값의 크기를 감소시켜 과적합을 개선하는 방식을 규제(Regularization)라고 부릅니다.
- 규제는 크게 L2 방식과 L1 방식으로 구분됩니다. L2 규제는 위에서 설명한 바와 같이 $\alpha * \|W\|_2^2$ 와 같이 W 의 제곱에 대해 페널티를 부여하는 방식을 말합니다. L2 규제를 적용한 회귀를 릿지(Ridge) 회귀라고 합니다.
- 라쏘(Lasso) 회귀는 L1 규제를 적용한 회귀입니다. L1 규제는 $\alpha * \|W\|_1$ 와 같이 W 의 절댓값에 대해 페널티를 부여합니다. L1 규제를 적용하면 영향력이 크지 않은 회귀 계수 값을 0으로 변환합니다.
- ElasticNet: L2, L1 규제를 함께 결합한 모델입니다. 주로 피처가 많은 데이터 세트에서 적용되며, L1 규제로 피처의 개수를 줄임과 동시에 L2 규제로 계수 값의 크기를 조정합니다.

릿지 회귀로 보스턴 집값 예측

```
# 앞의 LinearRegression에서 분할한 feature 데이터 셋인 X_data과 Target 데이터 셋인 Y_target 데이터셋을 그대로 이용
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

# boston 데이터셋 로드
boston = load_boston()

# boston 데이터셋 DataFrame 변환
bostonDF = pd.DataFrame(boston.data, columns = boston.feature_names)

# boston dataset의 target array는 주택 가격. PRICE 컬럼으로 DataFrame에 존재
bostonDF['PRICE'] = boston.target

y_target = bostonDF['PRICE']
X_data = bostonDF.drop(['PRICE'],axis=1,inplace=False)

# 릴지 클래스
ridge = Ridge(alpha = 10)
neg_mse_scores = cross_val_score(ridge, X_data, y_target, scoring="neg_mean_squared_error", cv = 5)
rmse_scores = np.sqrt(-1 * neg_mse_scores)
avg_rmse = np.mean(rmse_scores)
print(' 5 folds 의 개별 Negative MSE scores: ', np.round(neg_mse_scores, 3))
print(' 5 folds 의 개별 RMSE scores : ', np.round(rmse_scores,3))
print(' 5 folds 의 평균 RMSE : {0:.3f} '.format(avg_rmse))

5 folds 의 개별 Negative MSE scores:  [-11.422 -24.294 -28.144 -74.599 -28.517]
5 folds 의 개별 RMSE scores :  [3.38  4.929 5.305 8.637 5.34 ]
5 folds 의 평균 RMSE : 5.518
```

규제를 적용하지 않은 선형 회귀는 RMSE가 5.829였으므로 모델 성능이 향상된 것을 확인할 수 있다

alpha값을 0, 0.1, 1, 10, 100으로 변경하면서 RMSE값 측정

```
# Ridge에 사용될 alpha 파라미터의 값을 정의
alphas = [ 0 , 0.1 , 1 , 10 , 100]

# alphas list 값을 iteration하면서 alpha에 따른 평균 rmse 구함.
for alpha in alphas :
    ridge = Ridge(alpha = alpha)

    #cross_val_score를 이용하여 5 fold의 평균 RMSE 계산
    neg_mse_scores = cross_val_score(ridge, X_data, y_target, scoring="neg_mean_squared_error", cv = 5)
    avg_rmse = np.mean(np.sqrt(-1 * neg_mse_scores))
    print('alpha {0} 일 때 5 folds 의 평균 RMSE : {1:.3f} '.format(alpha,avg_rmse))

alpha 0 일 때 5 folds 의 평균 RMSE : 5.829
alpha 0.1 일 때 5 folds 의 평균 RMSE : 5.788
alpha 1 일 때 5 folds 의 평균 RMSE : 5.653
alpha 10 일 때 5 folds 의 평균 RMSE : 5.518
alpha 100 일 때 5 folds 의 평균 RMSE : 5.330
```

알파값이 증가할수록 모델 성능이 향상되고 있다

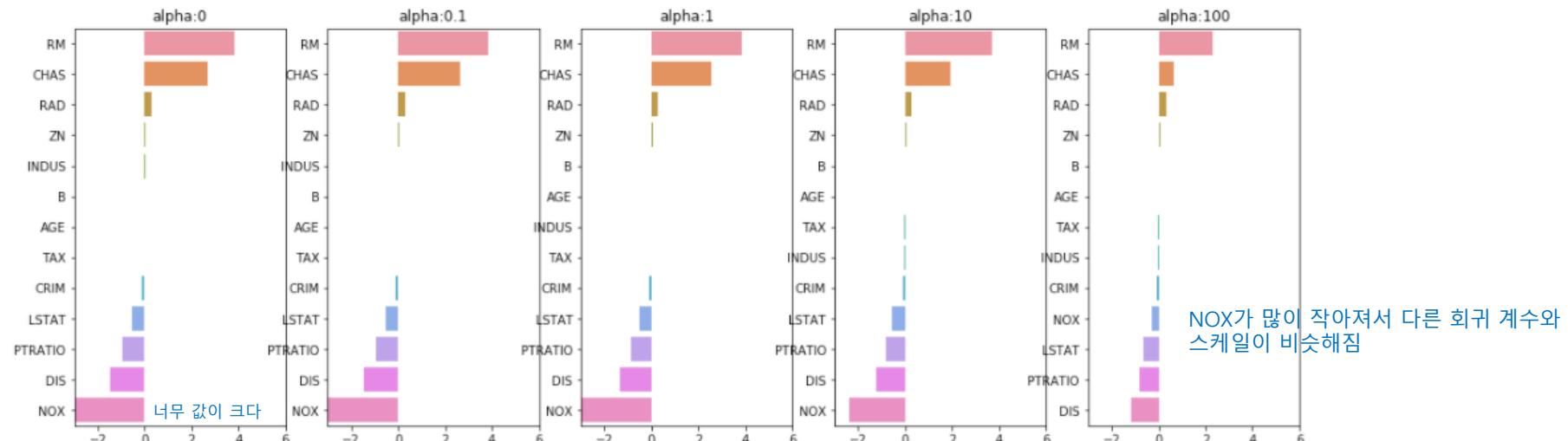
릿지 회귀로 보스턴 집값 예측

알파 값이 증가할수록 실제로 회귀 계수 값이 작아지는지 확인해보기

```
# 각 alpha에 따른 회귀 계수 값을 시각화하기 위해 5개의 열로 된 맷플롯립 축 생성
fig, axs = plt.subplots(figsize=(18,6), nrows=1, ncols=5)
# 각 alpha에 따른 회귀 계수 값을 데이터로 저장하기 위한 DataFrame 생성
coeff_df = pd.DataFrame()

# alphas 리스트 값을 차례로 입력해 회귀 계수 값 시각화 및 데이터 저장. pos는 axis의 위치 지정
for pos, alpha in enumerate(alphas) :
    ridge = Ridge(alpha = alpha)
    ridge.fit(X_data, y_target)
    # alpha에 따른 피처별 회귀 계수를 Series로 변환하고 이를 DataFrame의 컬럼으로 추가.
    coeff = pd.Series(data=ridge.coef_, index=X_data.columns)
    colname = 'alpha:' + str(alpha)
    coeff_df[colname] = coeff
    # 막대 그래프로 각 alpha 값에서의 회귀 계수를 시각화. 회귀 계수값이 높은 순으로 표현
    coeff = coeff.sort_values(ascending=False) # 회귀 계수 값이 높은 순으로 정렬
    axs[pos].set_title(colname)
    axs[pos].set_xlim(-3,6)
    sns.barplot(x=coeff.values, y=coeff.index, ax=axs[pos])

# for 문 바깥에서 맷플롯립의 show 호출 및 alpha에 따른 피처별 회귀 계수를 DataFrame으로 표시
plt.show()
```



릿지 회귀로 보스턴 집값 예측

알파 값이 증가할수록 실제로 회귀 계수 값이 작아지는지 확인해보기

```
# 알파값에 따른 회귀 계수 출력
ridge_alphas = [0, 0.1, 1, 10, 100]
sort_column = 'alpha:' + str(ridge_alphas[0])
coeff_df.sort_values(by=sort_column, ascending=False)
```

	alpha:0	alpha:0.1	alpha:1	alpha:10	alpha:100
RM	3.809865	3.818233	3.854000	3.702272	2.334536
CHAS	2.686734	2.670019	2.552393	1.952021	0.638335
RAD	0.306049	0.303515	0.290142	0.279596	0.315358
ZN	0.046420	0.046572	0.047443	0.049579	0.054496
INDUS	0.020559	0.015999	-0.008805	-0.042962	-0.052826
B	0.009312	0.009368	0.009673	0.010037	0.009393
AGE	0.000692	-0.000269	-0.005415	-0.010707	0.001212
TAX	-0.012335	-0.012421	-0.012912	-0.013993	-0.015856
CRIM	-0.108011	-0.107474	-0.104595	-0.101435	-0.102202
LSTAT	-0.524758	-0.525966	-0.533343	-0.559366	-0.660764
PTRATIO	-0.952747	-0.940759	-0.876074	-0.797945	-0.829218
DIS	-1.475567	-1.459626	-1.372654	-1.248808	-1.153390
NOX	-17.766611	-16.684645	-10.777015	-2.371619	-0.262847

릿지 회귀에서 알파값이 증가할수록 NOX값의 감소하여 다른 회귀 계수 값들과 스케일이 비슷해졌다. 이로 인해 기존 회귀 모델의 성능이 개선되었다.

라쏘 회귀

W의 절댓값에 페널티를 부여하는 L1 규제를 선형 회귀에 적용한 것이 라쏘(Lasso) 회귀입니다. 즉 L1 규제는 $\alpha * \|W\|_1$ 을 의미하며, 라쏘 회귀 비용함수의 목표는 $\text{RSS}(W) + \alpha * \|W\|_1$ 식을 최소화 하는 W를 찾는 것입니다. L2규제가 회귀 계수의 크기를 감소시키는 데 반해, L1규제는 불필요한 회귀 계수를 급격하게 감소시켜 0으로 만들고 제거합니다. 이러한 측면에서 L1 규제는 적절한 피처만 회귀에 포함시키는 피처 셀렉션의 특성을 가지고 있습니다.

사이킷런은 Lasso 클래스를 통해 라쏘 회귀를 구현하였습니다

L2 규제 vs L1 규제

L2의 경우에는 가중치의 값을 이용합니다. 어느 정도 튕는 값에 대해 대응할 수 있다는 소리죠. 따라서, 이상치나 노이즈가 있는 데이터에 대한 학습을 진행할 때 사용하면 좋습니다. 특히 선형 모델의 일반화에 좋습니다.

-> L2규제는 회귀 계수 값의 증감을 관리한다.

L1의 경우에는 가중치의 크기에 상관없이 상수값을 뺍니다. 이는 대체적으로 불필요한 가중치의 수치를 0으로 만들도록 하는 방향으로 적용됩니다.

즉, 중요한 가중치만을 취하기 때문에 sparse feature에 대한 모델을 구성하는데 적합합니다.

-> L1규제는 회귀에 적절한 피처만 포함시키는 피처 셀렉션의 특성을 가진다

엘라스틱넷 회귀

- 엘라스틱넷(Elastic Net) 회귀는 L2 규제와 L1 규제를 결합한 회귀입니다. 따라서 엘라스틱넷 회귀 비용함수의 목표는 $\text{RSS}(W) + \alpha_2 * \|W\|_2^2 + \alpha_1 * \|W\|_1$ 식을 최소화 하는 W 를 찾는 것입니다.
- 엘라스틱넷은 라쏘 회귀가 서로 상관관계가 높은 피처들의 경우에 이들 중에서 중요 피처만을 선택하고 다른 피처들은 모두 회귀 계수를 0으로 만드는 성향이 강합니다. 특히 이러한 성향으로 인해 α 값에 따라 회귀 계수의 값이 급격히 변동 할 수도 있는데, 엘라스틱넷 회귀는 이를 완화하기 위해 L2 규제를 라쏘 회귀에 추가한 것입니다.

엘라스틱넷 회귀

사이킷런은 ElasticNet 클래스를 통해서 엘라스틱넷 회귀를 구현합니다.

ElasticNet 클래스의 주요 생성 파라미터는 alpha 와 l1_ratio 입니다. ElasticNet 클래스의 alpha는 Ridge와 Lasso 클래스의 alpha값과는 다릅니다.

엘라스틱넷의 규제는 $a L1\text{규제} + b L2\text{규제}$ 로 정의될 수 있습니다.

ElasticNet alpha 파라미터

이 때 a는 L1규제의 alpha값, b는 L2 규제의 alpha 값입니다.

따라서 ElasticNet 클래스의 alpha파라미터 값은 $a + b$ 입니다

ElasticNet l1_ratio 파라미터

ElasticNet 클래스의 l1_ratio 파라미터 값은 $a / (a + b)$ 입니다.

l1_ratio가 0 이면 a 가 0 이므로 L2 규제와 동일합니다.

l1_ratio가 1이면 b가 0 이므로 L1 규제와 동일합니다.

$0 < l1_ratio < 1$ 이면 L1과 L2 규제를 함께 적절히 적용합니다.

만일 ElasticNet의 alpha가 10, l1_ratio가 0.7 이라면

$l1_ratio = 0.7 = a / a + b = 7/10$ 이므로 $a=7$ 이고 L1 alpha값은 7, L2 alpha값은 3입니다.

실습 : 라쏘 회귀

평균 RMSE, 회귀 계수 값들을 반환해주는 함수 이용

```
from sklearn.linear_model import Lasso, ElasticNet

# alpha값에 따른 회귀 모델의 폴드 평균 RMSE를 출력하고, 회귀 계수값들을 DataFrame으로 반환해주는 함수
def get_linear_reg_eval(model_name, params=None, X_data_n=None, y_target_n=None, verbose=True):
    coeff_df = pd.DataFrame()
    if verbose : print('#####', model_name, '#####')
    for param in params:
        if model_name =='Ridge': model = Ridge(alpha=param)
        elif model_name =='Lasso': model = Lasso(alpha=param)
        elif model_name =='ElasticNet': model = ElasticNet(alpha=param, l1_ratio=0.7)
        neg_mse_scores = cross_val_score(model, X_data_n,
                                         y_target_n, scoring="neg_mean_squared_error", cv = 5)
        avg_rmse = np.mean(np.sqrt(-1 * neg_mse_scores))
        print('alpha {0}일 때 5 폴드 세트의 평균 RMSE: {1:.3f}'.format(param, avg_rmse))

        # cross_val_score는 evaluation metric만 반환하므로 모델을 다시 학습하여 회귀 계수 추출
        model.fit(X_data, y_target)
        # alpha에 따른 피처별 회귀 계수를 Series로 변환하고 이를 DataFrame의 컬럼으로 추가.
        coeff = pd.Series(data=model.coef_, index=X_data.columns )
        colname='alpha:'+str(param)
        coeff_df[colname] = coeff
    return coeff_df
# end of get_linear_regre_eval
```

```
# 라쏘에 사용될 alpha 파라미터의 값을 정의하고 get_linear_reg_eval() 함수 호출
lasso_alphas = [0.07, 0.1, 0.5, 1, 3]
coeff_lasso_df = get_linear_reg_eval('Lasso', params=lasso_alphas, X_data_n=X_data, y_target_n=y_target)
```

```
#####
alpha 0.07일 때 5 폴드 세트의 평균 RMSE: 5.612 알파값이 0.07일 때 RMSE가 가장 좋다
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.615
alpha 0.5일 때 5 폴드 세트의 평균 RMSE: 5.669
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.776
alpha 3일 때 5 폴드 세트의 평균 RMSE: 6.189
```

실습 : 라쏘 회귀

라쏘는 알파 값이 증가되면 특정 피처의 회귀 계수를 0으로 만들어서 해당 피처를 회귀식에서 제외시켜 버린다.

```
# 반환된 coeff_lasso_df를 첫번째 컬럼순으로 내림차순 정렬하여 회귀계수 DataFrame 출력
sort_column = 'alpha:' + str(lasso_alphas[0])
coeff_lasso_df.sort_values(by=sort_column, ascending=False)
```

	alpha:0.07	alpha:0.1	alpha:0.5	alpha:1	alpha:3
RM	3.789725	3.703202	2.498212	0.949811	0.000000
CHAS	1.434343	0.955190	0.000000	0.000000	0.000000
RAD	0.270936	0.274707	0.277451	0.264206	0.061864
ZN	0.049059	0.049211	0.049544	0.049165	0.037231
B	0.010248	0.010249	0.009469	0.008247	0.006510
NOX	-0.000000	-0.000000	-0.000000	-0.000000	0.000000
AGE	-0.011706	-0.010037	0.003604	0.020910	0.042495
TAX	-0.014290	-0.014570	-0.015442	-0.015212	-0.008602
INDUS	-0.042120	-0.036619	-0.005253	-0.000000	-0.000000
CRIM	-0.098193	-0.097894	-0.083289	-0.063437	-0.000000
LSTAT	-0.560431	-0.568769	-0.656290	-0.761115	-0.807679
PTRATIO	-0.765107	-0.770654	-0.758752	-0.722966	-0.265072
DIS	-1.176583	-1.160538	-0.936605	-0.668790	-0.000000

알파값 0.5 이상일 때
CHAS 회귀 계수 값을 0으로 만들어 버림

알파값 0.07 부터
NOX 회귀 계수 값을 0으로 만들어 버림

실습 : 엘라스틱넷 회귀

```
# 엘라스틱넷에 사용될 alpha 파라미터의 값들을 정의하고 get_linear_reg_eval() 함수 호출
# l1_ratio는 0.7로 고정
elastic_alphas = [ 0.07, 0.1, 0.5, 1, 3]
coeff_elastic_df =get_linear_reg_eval('ElasticNet', params=elastic_alphas,
X_data_n=X_data, y_target_n=y_target)

##### ElasticNet #####
alpha 0.07일 때 5 폴드 세트의 평균 RMSE: 5.542
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.526
alpha 0.5일 때 5 폴드 세트의 평균 RMSE: 5.467
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.597
alpha 3일 때 5 폴드 세트의 평균 RMSE: 6.068
```

```
# 반환된 coeff_elastic_df를 첫번째 컬럼순으로 내림차순 정렬하여 회귀계수 DataFrame 출력
sort_column = 'alpha:'+str(elastic_alphas[0])
coeff_elastic_df.sort_values(by=sort_column, ascending=False)
```

	alpha:0.07	alpha:0.1	alpha:0.5	alpha:1	alpha:3
RM	3.574162	3.414154	1.918419	0.938789	0.000000
CHAS	1.330724	0.979706	0.000000	0.000000	0.000000
RAD	0.278880	0.283443	0.300761	0.289299	0.146846
ZN	0.050107	0.050617	0.052878	0.052136	0.038268
B	0.010122	0.010067	0.009114	0.008320	0.007020
AGE	-0.010116	-0.008276	0.007760	0.020348	0.043446
TAX	-0.014522	-0.014814	-0.016046	-0.016218	-0.011417
INDUS	-0.044855	-0.042719	-0.023252	-0.000000	-0.000000
CRIM	-0.099468	-0.099213	-0.089070	-0.073577	-0.019058
NOX	-0.175072	-0.000000	-0.000000	-0.000000	-0.000000
LSTAT	-0.574822	-0.587702	-0.693861	-0.760457	-0.800368
PTRATIO	-0.779498	-0.784725	-0.790969	-0.738672	-0.423065
DIS	-1.189438	-1.173647	-0.975902	-0.725174	-0.031208

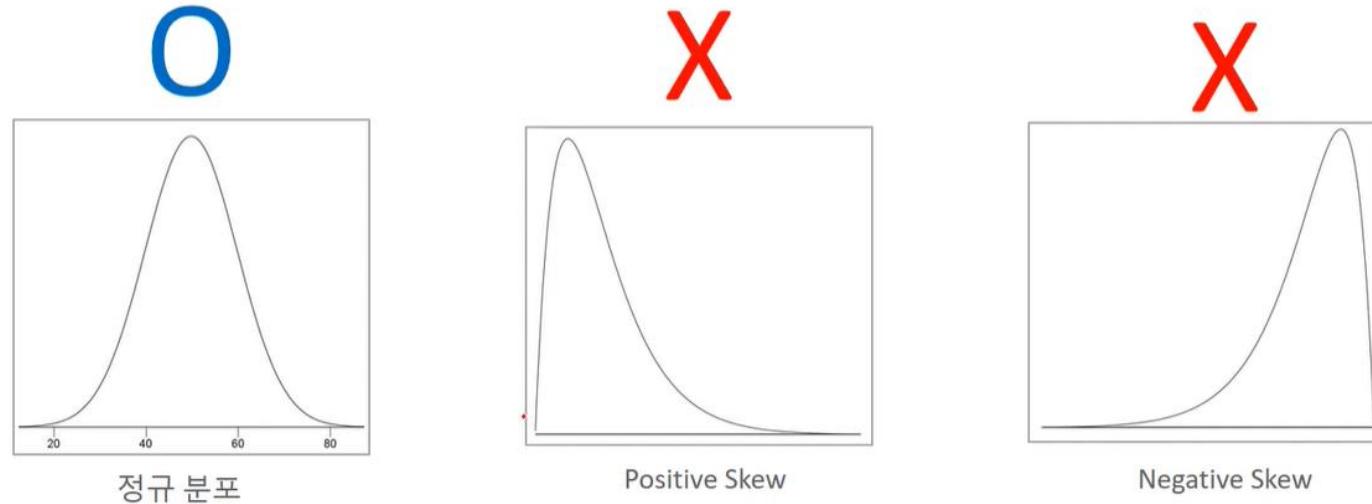
엘라스틱넷 회귀는
릿지와 라쏘가 적절히 조합된 회귀를 만들어 준다.

데이터 전처리

(정규화, 로그 변환, 스케일러,
원-핫 인코딩)

선형회귀 모델을 위한 데이터 변환

회귀 모델과 같은 선형 모델은 일반적으로 피처와 타깃값 간에 선형의 관계가 있다고 가정하고, 이러한 최적의 선형함수를 찾아내 결과값을 예측합니다. 또한 선형 회귀 모델은 피처값과 타깃값의 분포가 정규 분포(즉 평균을 중심으로 종 모양으로 데이터 값이 분포된 형태) 형태를 매우 선호합니다.



로그 변환, 스케일러, 다향 특성 적용

변환 대상	설명
타깃값 변환	<p>회귀에서 타깃값은 반드시 정규 분포를 가져야 함. 이를 위해 주로 로그 변환을 적용</p> <p>StandardScaler 클래스를 이용해 평균이 0, 분산이 1인 표준 정규 분포를 가진 데이터 세트로 변환하거나 MinMaxScaler 클래스를 이용해 최솟값이 0이고 최댓값이 1인 값으로 정규화를 수행합니다</p>
피처값 변환	<p>스케일링/정규화를 수행한 데이터 세트에 다시 다향 특성을 적용하여 변환하는 방법입니다. 보통 1번 방법을 통해 예측 성능에 향상이 없을 경우 이와 같은 방법을 적용합니다.</p> <p>원래 값에 log 함수를 적용하면 보다 정규 분포에 가까운 형태로 값이 분포됩니다. 로그 변환은 매우 유용한 변환이며, 실제로 선형 회귀에서는 앞에서 소개한 1, 2 번 방법보다 로그 변환이 훨씬 많이 사용되는 변환 방법입니다. 왜냐하면 1번 방법의 경우 예측 성능 향상을 크게 기대하기 어려운 경우가 많으며 2번 방법의 경우 피처의 개수가 매우 많을 경우에는 다향 변환으로 생성되는 피처의 개수가 기하급수로 늘어나서 과적합의 이슈가 발생할 수 있기 때문입니다.</p>

회귀를 위한 데이터 변환 방법 -인코딩

카테고리
고양이 원숭이 강아지 호랑이
1 2 3 4

선형 회귀의 데이터 인코딩은 일반적으로 레이블 인코딩이 아니라 **원-핫 인코딩**을 적용합니다.

	고양이	원숭이	강아지	호랑이
고양이	0	0	0	1
원숭이	0	1	0	0

실습 : 피처 데이터 변환에 따른 예측 성능 비교

변환 유형	alpha값			
	alpha=0.1	alpha=1	alpha=10	alpha=100
원본 데이터				
표준 정규 분포				
표준 정규 분포 + 2차 다항식				
최솟값/최댓값 정규화				
최솟값/최댓값 정규화 + 2차 다항식				
로그 변환				

예측 모델 성능

?

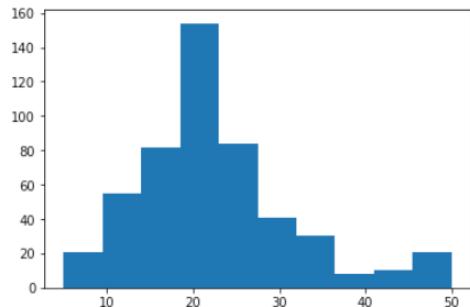
알고리즘 : 릿지 회귀
cross validation : 5 fold

실습 : 피처 데이터 변환에 따른 예측 성능 비교

타겟값은 비교적 정규화가 잘 되어 있다. 피처값만 정규화를 해주면 된다.

```
print(y_target.shape)
plt.hist(y_target, bins=10)

(506,)
(array([ 21.,  55.,  82., 154.,  84.,  41.,  30.,   8.,  10.,  21.]),
 array([ 5. ,  9.5, 14. , 18.5, 23. , 27.5, 32. , 36.5, 41. , 45.5, 50. ]),
 <a list of 10 Patch objects>)
```



데이터 전처리(정규화, 로그변환)을 해주는 함수 정의

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, PolynomialFeatures

# method는 표준 정규 분포 변환(Standard), 최대값/최소값 정규화(MinMax), 로그변환(Log) 결정
# p_degree는 다항식 특성을 추가할 때 적용. p_degree는 2이상 부여하지 않음.
def get_scaled_data(method='None', p_degree=None, input_data=None):
    if method == 'Standard':
        scaled_data = StandardScaler().fit_transform(input_data)
    elif method == 'MinMax':
        scaled_data = MinMaxScaler().fit_transform(input_data)
    elif method == 'Log':
        scaled_data = np.log1p(input_data)
    else:
        scaled_data = input_data

    if p_degree != None:
        scaled_data = PolynomialFeatures(degree=p_degree,
                                         include_bias=False).fit_transform(scaled_data)

    return scaled_data
```

실습 : 피처 데이터 변환에 따른 예측 성능 비교

결론

```
# Ridge의 alpha값을 다르게 적용하고 다양한 데이터 변환방법에 따른 RMSE 추출.
alphas = [0.1, 1, 10, 100]

#변환 방법은 모두 6개, 원본 그대로, 표준정규분포, 표준정규분포+다항식 특성
# 최대/최소 정규화, 최대/최소 정규화+다항식 특성, 로그변환
scale_methods=[(None, None), ('Standard', None), ('Standard', 2),
               ('MinMax', None), ('MinMax', 2), ('Log', None)]

for scale_method in scale_methods:
    X_data_scaled = get_scaled_data(method=scale_method[0], p_degree=scale_method[1], input_data=X_data)
    print('\n## 변환 유형:{0}, Polynomial Degree:{1}'.format(scale_method[0], scale_method[1]))

    # alpha값에 따른 회귀 모델의 폴드 평균 RMSE를 출력하고, 회귀 계수값들을 DataFrame으로 반환해주는 함수
    get_linear_reg_eval('Ridge', params=alphas, X_data_n=X_data_scaled,
                        y_target_n=y_target, verbose=False)
```

```
## 변환 유형:None, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.788
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.653
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.518
alpha 100일 때 5 폴드 세트의 평균 RMSE: 5.330
```

원본

```
## 변환 유형:Standard, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.826
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.803
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.637
alpha 100일 때 5 폴드 세트의 평균 RMSE: 5.421

## 변환 유형:Standard, Polynomial Degree:2
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 8.827
alpha 1일 때 5 폴드 세트의 평균 RMSE: 6.871
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.485
alpha 100일 때 5 폴드 세트의 평균 RMSE: 4.634
```

```
## 변환 유형:MinMax, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.764
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.465
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.754
alpha 100일 때 5 폴드 세트의 평균 RMSE: 7.635
```

```
## 변환 유형:MinMax, Polynomial Degree:2
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.298
alpha 1일 때 5 폴드 세트의 평균 RMSE: 4.323
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.185
alpha 100일 때 5 폴드 세트의 평균 RMSE: 6.538
```

```
## 변환 유형:Log, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 4.770
alpha 1일 때 5 폴드 세트의 평균 RMSE: 4.676
alpha 10일 때 5 폴드 세트의 평균 RMSE: 4.836
alpha 100일 때 5 폴드 세트의 평균 RMSE: 6.241
```

보통 로그 변환만 해도 충분히 RMSE가 좋아지게 만들 수 있다

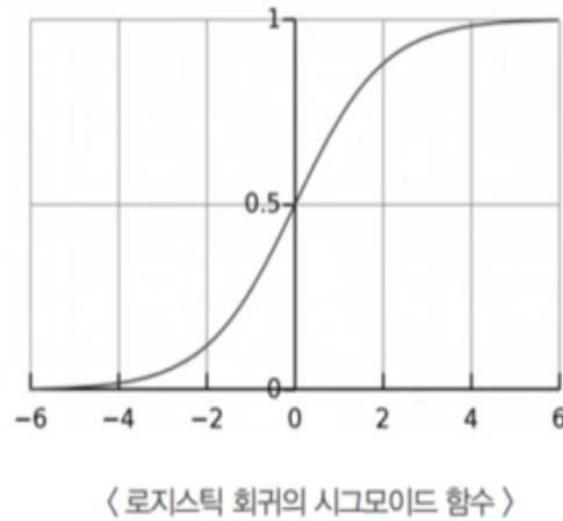
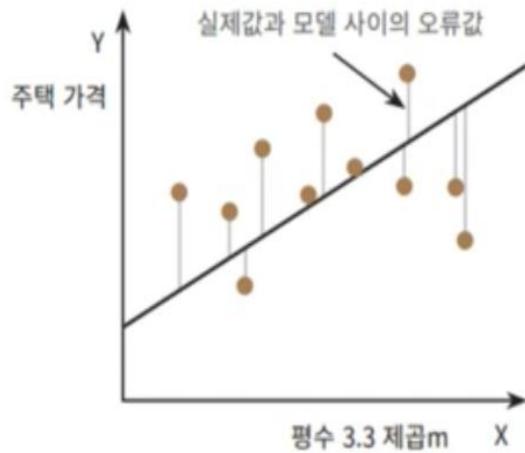
실습 : 피처 데이터 변환에 따른 예측 성능 비교

변환 유형	alpha값			
	alpha=0.1	alpha=1	alpha=10	alpha=100
원본 데이터	5.796	5.659	5.524	5.332
표준 정규 분포	5.834	5.810	5.643	5.424
표준 정규 분포 + 2차 다항식	8.776	6.849	5.487	4.631
최솟값/최댓값 정규화	5.770	5.468	5.755	7.635
최솟값/최댓값 정규화 + 2차 다항식	5.294	4.320	5.186	6.538
로그 변환	4.772	4.676	4.835	6.244

로지스틱 회귀 개요

로지스틱 회귀는 선형 회귀 방식을 분류에 적용한 알고리즘입니다. 즉, 로지스틱 회귀는 분류에 사용됩니다.

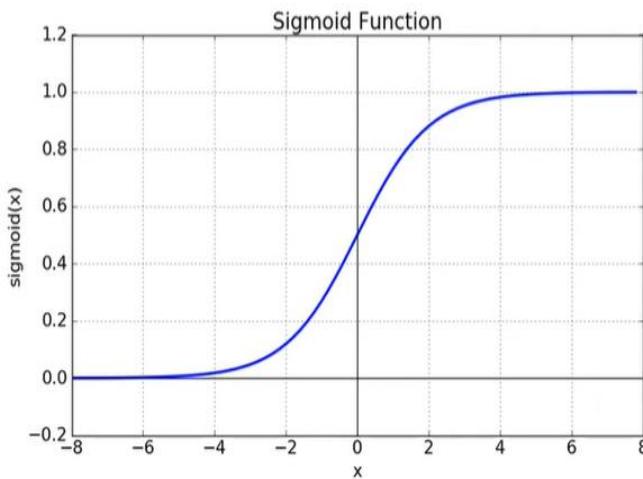
로지스틱 회귀가 선형 회귀와 다른 점은 학습을 통해 선형 함수의 회귀 최적선을 찾는 것이 아니라 시그모이드(Sigmoid) 함수 최적선을 찾고 이 시그모이드 함수의 반환 값을 확률로 간주해 확률에 따라 분류를 결정한다는 것입니다



로지스틱 회귀 예측

로지스틱 회귀 주로 이진 분류(0과 1)에 사용됩니다(물론 다중 클래스 분류에도 적용될 수 있습니다). 로지스틱 회귀에서 예측 값은 예측 확률을 의미하며, 예측 값 즉 예측 확률이 0.5 이상이면 1로, 0.5 이하이면 0으로 예측합니다. 로지스틱 회귀의 예측 확률은 시그모이드 함수의 출력값으로 계산됩니다.

시그모이드 함수 $y = \frac{1}{1+e^{-x}}$



파이썬 머신러닝 완벽 가이드

단순 선형 회귀: $y=w_1x+w_0$ 가 있다고 할 때

로지스틱 회귀는 0과 1을 예측하기에 단순 회귀식은 의미가 없습니다.
하지만 Odds(성공확률/실패확률)을 통해 선형 회귀식에 확률을 적용할 수 있습니다.

$$\text{Odds}(p) = p/(1-p)$$

하지만 확률p의 범위가 (0, 1)이므로 선형 회귀의 반환값인 (-무한대, +무한대)에 대응하기 위하여 로그 변환을 수행하고 이 값에 대한 선형 회귀를 적용합니다.

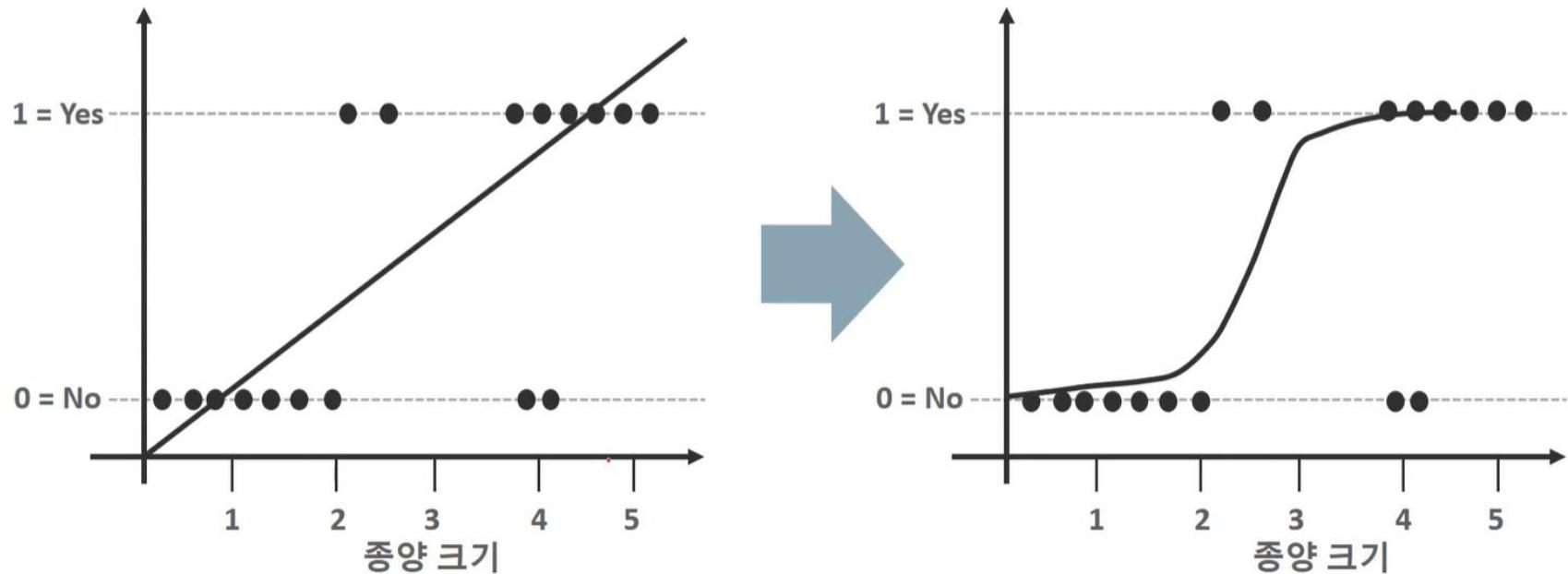
$$\text{Log(Odds}(p)\text{)} = w_1x + w_0$$

해당 식을 데이터 값 x의 확률 p로 정리하면 아래와 같습니다.

$$p(x) = \frac{1}{1+e^{-(w_1x+w_0)}}$$

로지스틱 회귀는 학습을 통해 시그모이드 함수의 w 를 최적화하여 예측하는 것입니다.

시그모이드를 이용한 로지스틱 회귀 예측



- 로지스틱 회귀는 가볍고 빠르지만, 이진 분류 예측 성능도 뛰어납니다. 이 때문에 로지스틱 회귀를 이진 분류의 기본 모델로 사용하는 경우가 많습니다. 또한 로지스틱 회귀는 희소한 데이터 세트 분류에도 뛰어난 성능을 보여서 텍스트 분류에서도 자주 사용됩니다

로지스틱 회귀 특징

- 로지스틱 회귀는 가볍고 빠르지만, 이진 분류 예측 성능도 뛰어납니다. 이 때문에 로지스틱 회귀를 이진 분류의 기본 모델로 사용하는 경우가 많습니다. 또한 로지스틱 회귀는 희소한 데이터 세트 분류에도 뛰어난 성능을 보여서 텍스트 분류에서도 자주 사용됩니다
- 사이킷런은 `LogisticRegression` 클래스로 로지스틱 회귀를 구현합니다. 주요 하이퍼 파라미터로 `penalty`와 `C`가 있습니다. Penalty는 규제(Regularization)의 유형을 설정하며 'l2'로 설정 시 L2 규제를, 'l1'으로 설정 시 L1 규제를 뜻합니다. 기본은 'l2'입니다. `C`는 규제 강도를 조절하는 `alpha`값의 역수입니다. 즉 $C = 1/\alpha$ 입니다. `C`값이 작을 수록 규제 강도가 큽니다.

실습: 로지스틱 회귀

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression

# 위스콘신 유방암 데이터 불러오기
cancer = load_breast_cancer()

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# StandardScaler( )로 평균이 0, 분산 1로 데이터 분포도 변환
scaler = StandardScaler()
data_scaled = scaler.fit_transform(cancer.data)

X_train, X_test, y_train, y_test = train_test_split(
    data_scaled, cancer.target, test_size=0.3, random_state=0)

from sklearn.metrics import accuracy_score, roc_auc_score

# 로지스틱 회귀를 이용하여 학습 및 예측 수행.
lr_clf = LogisticRegression()

lr_clf.fit(X_train, y_train)

lr_preds = lr_clf.predict(X_test)

# accuracy와 roc_auc 측정
print('accuracy: {:.3f}'.format(accuracy_score(y_test, lr_preds)))
print('roc_auc: {:.3f}'.format(roc_auc_score(y_test, lr_preds)))

accuracy: 0.977
roc_auc: 0.972
```



```
from sklearn.model_selection import GridSearchCV

params={'penalty':['l2', 'l1'],
        'C':[0.01, 0.1, 1, 1, 5, 10]}

grid_clf = GridSearchCV(lr_clf, param_grid=params, scoring='accuracy', cv=3 )
grid_clf.fit(data_scaled, cancer.target)
print('최적 하이퍼 파라미터:{0}, 최적 평균 정확도:{1:.3f}'.format(grid_clf.best_params_,
                                                               grid_clf.best_score_))

최적 하이퍼 파라미터:{'C': 1, 'penalty': 'l2'}, 최적 평균 정확도:0.975
```

회기 트리 개요

회귀 트리 : 트리 기반의 회귀 방식

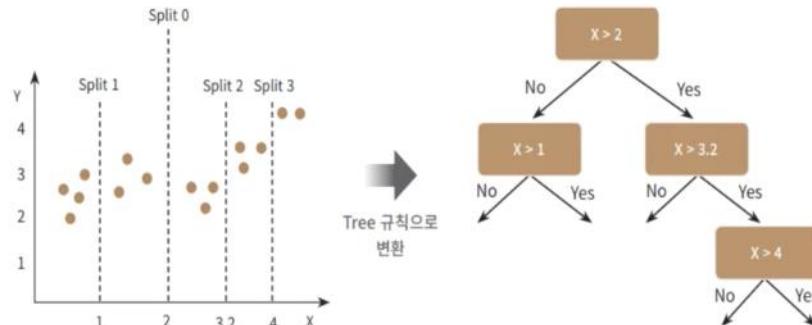
- 사이킷런의 결정 트리 및 결정 트리 기반의 앙상블 알고리즘은 분류 뿐만 아니라 회귀도 가능합니다.
- 이는 트리가 **CART(Classification and Regression Tree)**를 기반으로 만들어 졌기 때문입니다. CART는 분류 뿐만 아니라 회귀도 가능한 트리 분할 알고리즘입니다.
- CART 회귀 트리는 분류와 유사하게 분할을 하며, 분할 기준은 RSS(SSE)가 최소가 될 수 있는 기준을 찾아서 분할 됩니다.
- 최종 분할이 완료 된 후에 각 분할 영역에 있는 데이터 결정값들의 평균 값으로 학습/예측합니다.

회기 트리 프로세스

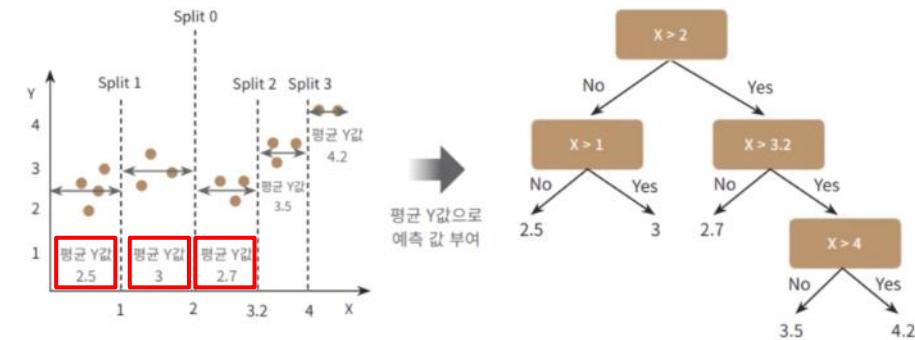
CART : Classification And Regression Trees



1 RSS를 최소화 하는 규칙 기준에 따라 분할



2 최종 분할된 영역에 있는 데이터들의 평균값들로 학습/예측

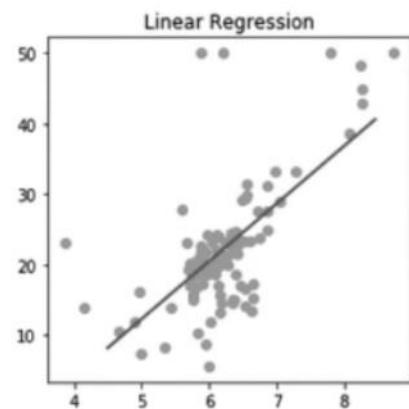


사이킷런의 회귀 트리 클래스

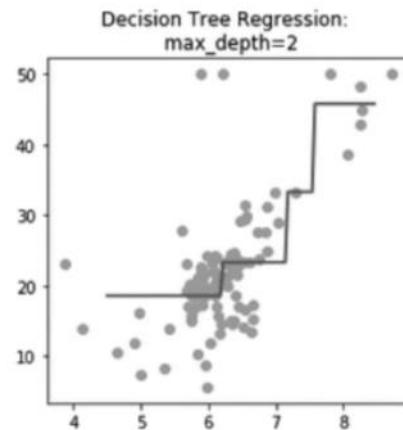
알고리즘	회귀 Estimator 클래스	분류 Estimator 클래스
Decision Tree	DecisionTreeRegressor	DecisionTreeClassifier
Gradient Boosting	GradientBoostingRegressor	GradientBoostingClassifier
XGBoost	XGBRegressor	XGBClassifier
LightGBM	LGBMRegressor	LGBMClassifier

회귀 트리의 오버피팅(과대적합)

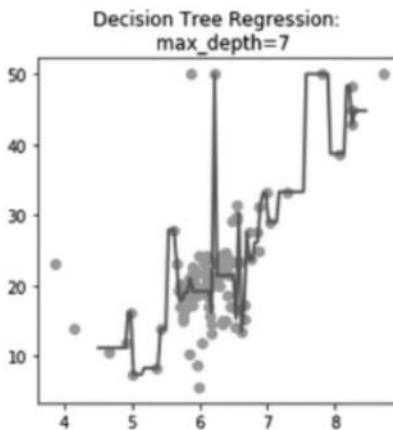
회귀 트리 역시 복잡한 트리 구조를 가질 경우 오버 피팅하기 쉬우므로 **트리의 크기**와 **노드 개수**의 제한 등의 방법을 통해 오버 피팅을 개선 할 수 있습니다.



과소 적합



최적 모델



과대 적합

실습 : 회귀 트리로 보스턴 집값 예측

데이터 불러오기

```
from sklearn.datasets import load_boston
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor
import pandas as pd
import numpy as np

# 보스턴 데이터 세트 로드
boston = load_boston()
bostonDF = pd.DataFrame(boston.data, columns = boston.feature_names)

bostonDF['PRICE'] = boston.target
y_target = bostonDF['PRICE']
X_data = bostonDF.drop(['PRICE'], axis=1, inplace=False)

rf = RandomForestRegressor(random_state=0, n_estimators=1000)
neg_mse_scores = cross_val_score(rf, X_data, y_target, scoring="neg_mean_squared_error", cv = 5)
rmse_scores = np.sqrt(-1 * neg_mse_scores)
avg_rmse = np.mean(rmse_scores)

print('5 교차 검증의 개별 Negative MSE scores: ', np.round(neg_mse_scores, 2))
print('5 교차 검증의 개별 RMSE scores : ', np.round(rmse_scores, 2))
print('5 교차 검증의 평균 RMSE : {:.3f}'.format(avg_rmse))
```

5 교차 검증의 개별 Negative MSE scores: [-7.88 -13.14 -20.57 -46.23 -18.88]

5 교차 검증의 개별 RMSE scores : [2.81 3.63 4.54 6.8 4.34]

5 교차 검증의 평균 RMSE : 4.423

실습 : 회귀 트리로 보스턴 집값 예측

3개 회귀 트리 모델로 회귀 수행

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import GradientBoostingRegressor

dt_reg = DecisionTreeRegressor(random_state=0, max_depth=4)
rf_reg = RandomForestRegressor(random_state=0, n_estimators=1000)
gb_reg = GradientBoostingRegressor(random_state=0, n_estimators=1000)

# 트리 기반의 회귀 모델을 반복하면서 평가 수행
models = [dt_reg, rf_reg, gb_reg]
for model in models:
    get_model_cv_prediction(model, X_data, y_target)

##### DecisionTreeRegressor #####
5 교차 검증의 평균 RMSE : 5.978
##### RandomForestRegressor #####
5 교차 검증의 평균 RMSE : 4.423
##### GradientBoostingRegressor #####
5 교차 검증의 평균 RMSE : 4.269
```

GradientBoostingRegressor의 RMSE 값이 가장 작게 나왔다

회귀 트리의 피처 중요도 파악

회귀 트리는 feature_importances_로 피처 중요도를 파악한다 (선형 회귀의 회귀 계수 역할)

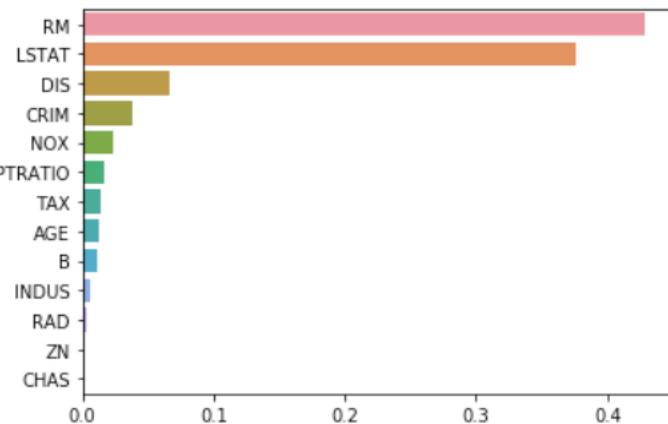
```
import seaborn as sns
%matplotlib inline

rf_reg = RandomForestRegressor(n_estimators=1000)

# 앞 예제에서 만들어진 X_data, y_target 데이터 셋을 적용하여 학습합니다.
rf_reg.fit(X_data, y_target)

feature_series = pd.Series(data=rf_reg.feature_importances_, index=X_data.columns )
feature_series = feature_series.sort_values(ascending=False)
sns.barplot(x= feature_series, y=feature_series.index)
```

<matplotlib.axes._subplots.AxesSubplot at 0x7f8d68345dd0>



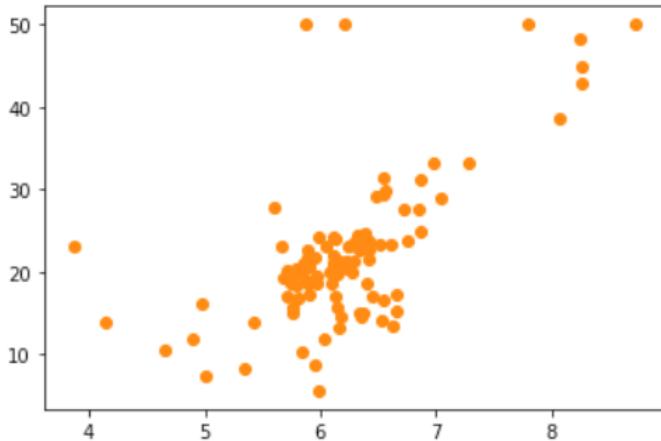
회기 트리의 오버 피팅 시각화

```
import matplotlib.pyplot as plt
%matplotlib inline

bostonDF_sample = bostonDF[['RM', 'PRICE']]
bostonDF_sample = bostonDF_sample.sample(n=100,random_state=0)
print(bostonDF_sample.shape)
plt.figure()
plt.scatter(bostonDF_sample.RM , bostonDF_sample.PRICE,c="darkorange")
```

(100, 2)

<matplotlib.collections.PathCollection at 0x7f8dba56d9d0>



회기 트리의 오버 피팅 시각화

```
import numpy as np
from sklearn.linear_model import LinearRegression

# 선형 회귀와 결정 트리 기반의 Regressor 생성. DecisionTreeRegressor의 max_depth는 각각 2, 7
lr_reg = LinearRegression()
rf_reg2 = DecisionTreeRegressor(max_depth=2)
rf_reg7 = DecisionTreeRegressor(max_depth=7)

# 실제 예측을 적용할 테스트용 데이터셋을 4.5 ~ 8.5 까지 100개 데이터셋 생성.
X_test = np.arange(4.5, 8.5, 0.04).reshape(-1, 1)

# 보스턴 주택가격 데이터에서 시각화를 위해 피처는 RM만, 그리고 결과 데이터인 PRICE 추출
X_feature = bostonDF_sample['RM'].values.reshape(-1, 1)
y_target = bostonDF_sample['PRICE'].values.reshape(-1, 1)

# 학습과 예측 수행.
lr_reg.fit(X_feature, y_target)
rf_reg2.fit(X_feature, y_target)
rf_reg7.fit(X_feature, y_target)

pred_lr = lr_reg.predict(X_test)
pred_rf2 = rf_reg2.predict(X_test)
pred_rf7 = rf_reg7.predict(X_test)

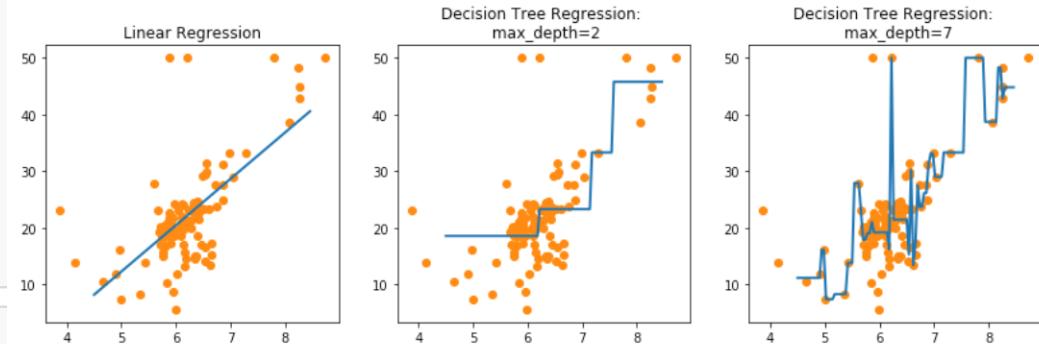
fig, (ax1, ax2, ax3) = plt.subplots(figsize=(14,4), ncols=3)

# x축값을 4.5 ~ 8.5로 변환하여 입력했을 때, 선형 회귀와 결정 트리 회귀 예측 선 시각화
# 선형 회귀로 학습된 모델 회귀 예측선
ax1.set_title('Linear Regression')
ax1.scatter(bostonDF_sample.RM, bostonDF_sample.PRICE, c="darkorange")
ax1.plot(X_test, pred_lr, label="linear", linewidth=2)

# DecisionTreeRegressor의 max_depth를 2로 했을 때 회귀 예측선
ax2.set_title('Decision Tree Regression: \nmax_depth=2')
ax2.scatter(bostonDF_sample.RM, bostonDF_sample.PRICE, c="darkorange")
ax2.plot(X_test, pred_rf2, label="max_depth:2", linewidth=2)

# DecisionTreeRegressor의 max_depth를 7로 했을 때 회귀 예측선
ax3.set_title('Decision Tree Regression: \nmax_depth=7')
ax3.scatter(bostonDF_sample.RM, bostonDF_sample.PRICE, c="darkorange")
ax3.plot(X_test, pred_rf7, label="max_depth:7", linewidth=2)

[<matplotlib.lines.Line2D at 0x7f8d89119bd0>]
```



선형 회귀는 직선으로 예측 회귀선을 표현

회귀 트리는 분할되는 데이터 지점에 따라 계단 형태로 회귀선 표현

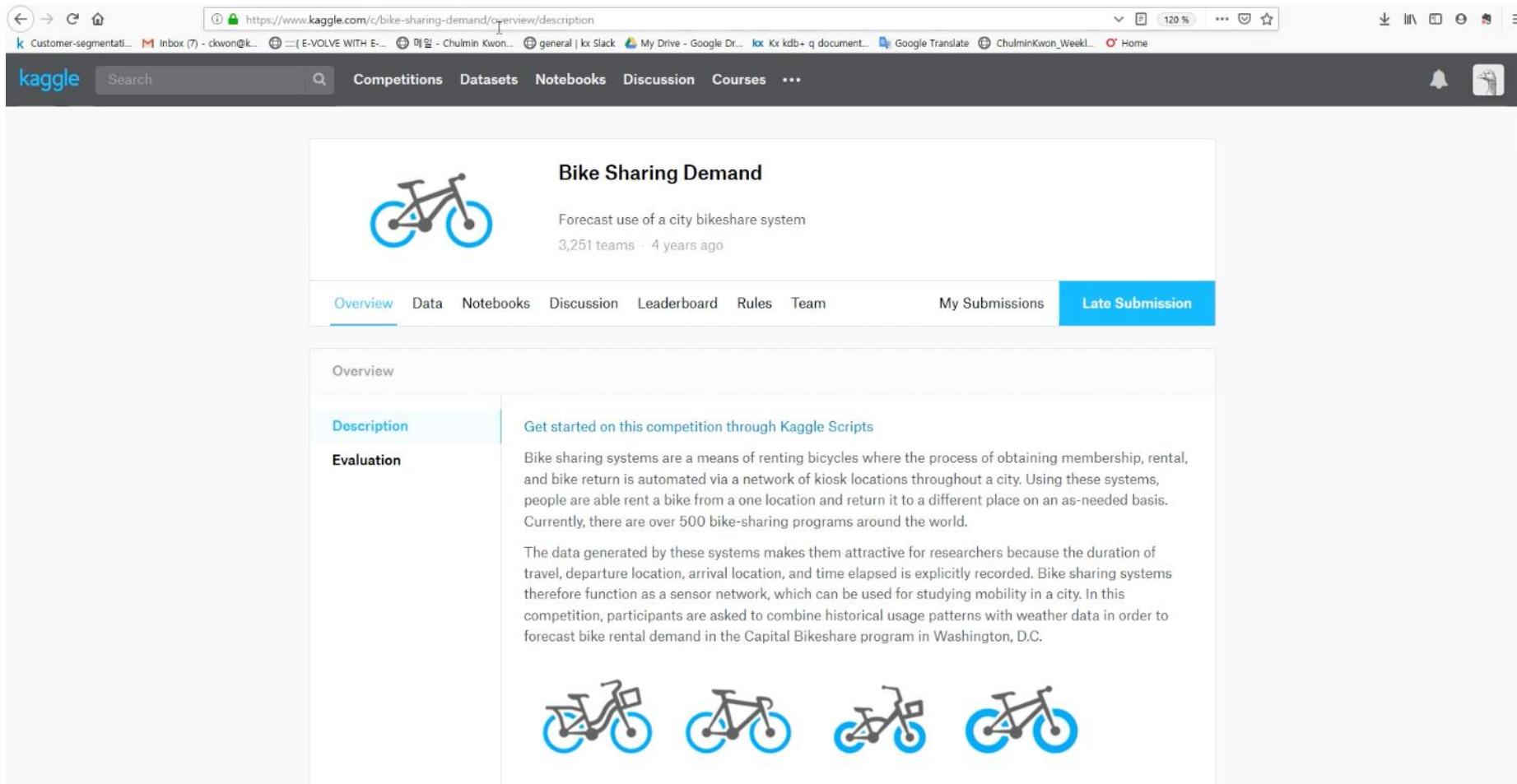
DecisionTreeRegressor의 max_depth=2인 경우
어느 정도 분류가 잘 됨

DecisionTreeRegressor의 max_depth=7인 경우
학습 데이터 세트의 이상치(outlier) 데이터도 학습하면서 복잡한 계단 형태의 회귀선을 만들어 과적합 모델을 만듦.

회귀 실습: 자전거 대여 수요 예측

캐글 : bike-sharing-demand

<https://www.kaggle.com/c/bike-sharing-demand>



The screenshot shows the 'Bike Sharing Demand' competition page on Kaggle. The page features a large image of a bicycle on the left and a title 'Bike Sharing Demand' in the center. Below the title, it says 'Forecast use of a city bikeshare system' and '3,251 teams - 4 years ago'. The navigation bar includes 'Overview', 'Data', 'Notebooks', 'Discussion', 'Leaderboard', 'Rules', 'Team', 'My Submissions', and a highlighted 'Late Submission' tab. The 'Overview' section contains tabs for 'Description' and 'Evaluation'. The 'Description' tab includes a sub-section 'Get started on this competition through Kaggle Scripts'. The 'Evaluation' tab contains a detailed description of bike sharing systems and their data. At the bottom, there are four small bicycle icons.

Bike Sharing Demand

Forecast use of a city bikeshare system

3,251 teams - 4 years ago

Overview Data Notebooks Discussion Leaderboard Rules Team My Submissions Late Submission

Description

Get started on this competition through Kaggle Scripts

Evaluation

Bike sharing systems are a means of renting bicycles where the process of obtaining membership, rental, and bike return is automated via a network of kiosk locations throughout a city. Using these systems, people are able rent a bike from a one location and return it to a different place on an as-needed basis. Currently, there are over 500 bike-sharing programs around the world.

The data generated by these systems makes them attractive for researchers because the duration of travel, departure location, arrival location, and time elapsed is explicitly recorded. Bike sharing systems therefore function as a sensor network, which can be used for studying mobility in a city. In this competition, participants are asked to combine historical usage patterns with weather data in order to forecast bike rental demand in the Capital Bikeshare program in Washington, D.C.

자전거 데이터 확인

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)

bike_df = pd.read_csv('./train.csv')

print(bike_df.shape)
bike_df.head(3)
```

(10886, 12)

	날짜	계절	휴일	주중/주 말	날씨	온도	체감온도	상대습도	풍속	비사전등록자 대여 수	사전등록자 대여 수	총 대여 횟수
	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32

datetime: hourly date + timestamp

season: 1 = 봄, 2 = 여름, 3 = 가을, 4 = 겨울

holiday: 1 = 토, 일요일의 주말을 제외한 국경일 등의 휴일, 0 = 휴일이 아닌 날

workingday: 1 = 토, 일요일의 주말 및 휴일이 아닌 주중, 0 = 주말 및 휴일

weather:

- 1 = 맑음, 약간 구름 낀 흐림
- 2 = 안개, 안개 + 흐림
- 3 = 가벼운 눈, 가벼운 비 + 천둥
- 4 = 심한 눈/비, 천둥/번개

temp: 온도(섭씨)

atemp: 체감온도(섭씨)

humidity: 상대습도

windspeed: 풍속

casual: 사전에 등록되지 않는 사용자가 대여한 횟수

registered: 사전에 등록된 사용자가 대여한 횟수

count: 대여 횟수

데이터 타입, null값 확인

데이터에 null값은 없다

```
bike_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   datetime    10886 non-null   object 
 1   season      10886 non-null   int64  
 2   holiday     10886 non-null   int64  
 3   workingday  10886 non-null   int64  
 4   weather     10886 non-null   int64  
 5   temp        10886 non-null   float64
 6   atemp       10886 non-null   float64
 7   humidity    10886 non-null   int64  
 8   windspeed   10886 non-null   float64
 9   casual      10886 non-null   int64  
 10  registered  10886 non-null   int64  
 11  count       10886 non-null   int64  
dtypes: float64(3), int64(8), object(1)
memory usage: 1020.7+ KB
```

년,월,일,시 각각 분해해서 데이터를 분석해보는게 필요

데이터 타입 변경 삭제

```
# 문자열을 datetime 타입으로 변경.
```

```
bike_df['datetime'] = bike_df.datetime.apply(pd.to_datetime)
```

```
# datetime 타입에서 년, 월, 일, 시간 추출
```

```
bike_df['year'] = bike_df.datetime.apply(lambda x : x.year)
```

```
bike_df['month'] = bike_df.datetime.apply(lambda x : x.month)
```

```
bike_df['day'] = bike_df.datetime.apply(lambda x : x.day)
```

```
bike_df['hour'] = bike_df.datetime.apply(lambda x: x.hour)
```

```
bike_df.head(3)
```

	datetime	season	holiday	workingday	weather	temp	atemp
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635

Data type 확인 후 불필요한 목록 삭제

```
bike_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 10886 entries, 0 to 10885
```

```
Data columns (total 16 columns):
```

#	Column	Non-Null Count	Dtype
0	datetime	10886 non-null	datetime64[ns]
1	season	10886 non-null	int64
2	holiday	10886 non-null	int64
3	workingday	10886 non-null	int64
4	weather	10886 non-null	int64
5	temp	10886 non-null	float64
6	atemp	10886 non-null	float64
7	humidity	10886 non-null	int64
8	windspeed	10886 non-null	float64
9	casual	10886 non-null	int64
10	registered	10886 non-null	int64
11	count	10886 non-null	int64
12	year	10886 non-null	int64
13	month	10886 non-null	int64
14	day	10886 non-null	int64
15	hour	10886 non-null	int64

```
dtypes: datetime64[ns](1), float64(3), int64(12)
```

```
memory usage: 1.3 MB
```

```
drop_columns = ['datetime', 'casual', 'registered']  
bike_df.drop(drop_columns, axis=1, inplace=True)
```

에러 함수들 정의 후 선형회귀 학습/예측

```
from sklearn.metrics import mean_squared_error, mean_absolute_error

# log 값 변환 시 NaN등의 이유로 log() 가 아닌 log1p() 를 이용하여 RMSLE 계산
def rmsle(y, pred):
    log_y = np.log1p(y)
    log_pred = np.log1p(pred)
    squared_error = (log_y - log_pred) ** 2
    rmsle = np.sqrt(np.mean(squared_error))
    return rmsle

# 사이킷런의 mean_square_error() 를 이용하여 RMSE 계산
def rmse(y, pred):
    return np.sqrt(mean_squared_error(y, pred))

# MAE, RMSE, RMSLE 를 모두 계산
def evaluate_regr(y, pred):
    rmsle_val = rmsle(y, pred)
    rmse_val = rmse(y, pred)
    # MAE 는 scikit learn의 mean_absolute_error() 로 계산
    mae_val = mean_absolute_error(y, pred)
    print('RMSLE: {:.3f}, RMSE: {:.3f}, MAE: {:.3f}'.format(rmsle_val, rmse_val, mae_val))
```

```
# 학습 데이터, 테스트 데이터 분리
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LinearRegression, Ridge, Lasso

y_target = bike_df['count']
X_features = bike_df.drop(['count'], axis=1, inplace=False)

X_train, X_test, y_train, y_test = train_test_split(X_features, y_target, test_size=0.3, random_state=0)

# 선형회귀 적용 후 학습/예측/평가
lr_reg = LinearRegression()
lr_reg.fit(X_train, y_train)
pred = lr_reg.predict(X_test)

evaluate_regr(y_test, pred)

RMSLE: 1.165, RMSE: 140.900, MAE: 105.924
```

RMSLE에 비해 RMSE값이 매우 크게 나왔다. 예측 에러가 매우 큰 값들이 섞여 있기 때문

예측값과 실제값 오차 확인

실제 값과 예측 값의 차이가 매우 큰 것을 확인할 수 있다.

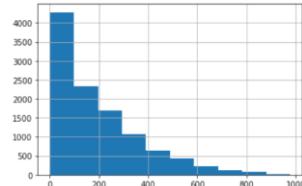
```
: def get_top_error_data(y_test, pred, n_tops = 5):
    # DataFrame에 컬럼들로 실제 대여횟수(count)와 예측 값을 서로 비교 할 수 있도록 생성.
    result_df = pd.DataFrame(y_test.values, columns=['real_count'])
    result_df['predicted_count'] = np.round(pred)
    result_df['diff'] = np.abs(result_df['real_count'] - result_df['predicted_count'])
    # 예측값과 실제값이 가장 큰 데이터 순으로 출력.
    print(result_df.sort_values('diff', ascending=False)[:n_tops])

get_top_error_data(y_test,pred,n_tops=5)

  real_count  predicted_count      diff
1618        890            322.0    568.0
3151        798            241.0    557.0
966         884            327.0    557.0
412         745            194.0    551.0
2817        856            310.0    546.0
```

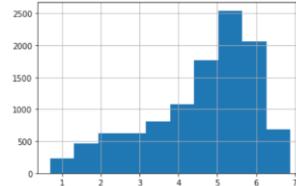
타겟값에 로그를 취해서 정규화

```
y_target.hist()  
<matplotlib.axes._subplots.AxesSubplot at 0x7fcff8f5dc90>
```



정규화 전 타겟값 분포

```
y_log_transform = np.log1p(y_target)  
y_log_transform.hist()  
<matplotlib.axes._subplots.AxesSubplot at 0x7fd0284a9c90>
```



로그 변환 후 타겟값 분포

-> 어느 정도 정규분포를 이룬다

```
# 타겟 컬럼인 count 값을 log1p 로 Log 변환  
y_target_log = np.log1p(y_target)  
  
# 로그 변환된 y_target_log를 반영하여 학습/테스트 데이터 셋 분할  
X_train, X_test, y_train, y_test = train_test_split(X_features, y_target_log, test_size=0.3, random_state=0)  
lr_reg = LinearRegression()  
lr_reg.fit(X_train, y_train)  
pred = lr_reg.predict(X_test)  
  
# 테스트 데이터 셋의 Target 값은 Log 변환되었으므로 다시 expml를 이용하여 원래 scale로 변환  
y_test_exp = np.expml(y_test)  
  
# 예측 값 역시 Log 변환된 타겟 기반으로 학습되어 예측되었으므로 다시 expml로 scale변환  
pred_exp = np.expml(pred)  
  
evaluate_regr(y_test_exp, pred_exp)
```

RMSLE: 1.017, RMSE: 162.594, MAE: 109.286

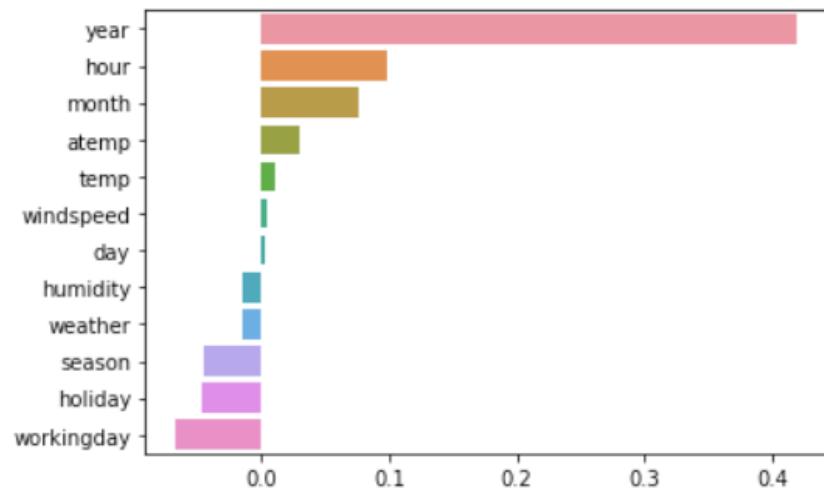
아직도 RMSLE에 비해 RMSE값이 매우 크게 나왔다.

피처 별 회귀 계수 확인

피처 별 회귀 계수 확인

```
coef = pd.Series(lr_reg.coef_, index=X_features.columns)
coef_sort = coef.sort_values(ascending=False)
sns.barplot(x=coef_sort.values, y=coef_sort.index)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd012883690>
```



year(2011, 2012)가 영향력이 큰 것을 볼 수 있다.
-> 해당 데이터는 2011년에 창업한 스타트업으로
2012년부터 더 성장해서 대여 수요량이 늘어난 것

원-핫 인코딩 후 다시 학습/예측

```
# 'year', 'month', 'hour', 'season', 'weather' feature들을 One Hot Encoding
X_features_ohe = pd.get_dummies(X_features, columns=['year', 'month', 'hour', 'holiday',
                                                     'workingday', 'season', 'weather'])

# 원-핫 인코딩이 적용된 feature 데이터 세트 기반으로 학습/예측 데이터 분할.
X_train, X_test, y_train, y_test = train_test_split(X_features_ohe, y_target_log,
                                                    test_size=0.3, random_state=0)

# 모델과 학습/테스트 데이터 셋을 입력하면 성능 평가 수치를 반환
def get_model_predict(model, X_train, X_test, y_train, y_test, is_expm1=False):
    model.fit(X_train, y_train)
    pred = model.predict(X_test)
    if is_expm1 :
        y_test = np.expm1(y_test)
        pred = np.expm1(pred)
    print('###',model.__class__.__name__,'###')
    evaluate_regr(y_test, pred)
# end of function get_model_predict

# model 별로 평가 수행
lr_reg = LinearRegression()
ridge_reg = Ridge(alpha=10)
lasso_reg = Lasso(alpha=0.01)

for model in [lr_reg, ridge_reg, lasso_reg]:
    get_model_predict(model,X_train, X_test, y_train, y_test, is_expm1=True)

### LinearRegression ####
RMSLE: 0.589, RMSE: 97.484, MAE: 63.106
### Ridge ####
RMSLE: 0.589, RMSE: 98.407, MAE: 63.648
### Lasso ####
RMSLE: 0.634, RMSE: 113.031, MAE: 72.658
```

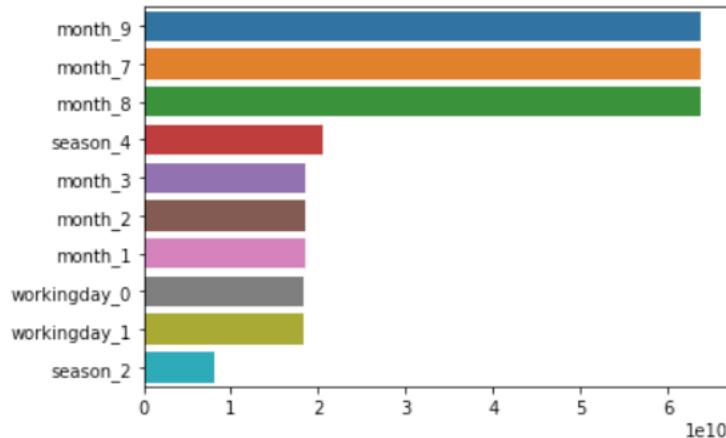
기본 선형회귀와 릿지, 라쏘 모델에 대해 성능 평가를 해주는 함수

이전보다 RMSE가 많이 줄은 것을 볼 수 있다

원-핫 인코딩 후 회귀 계수 확인

```
coef = pd.Series(lr_reg.coef_ , index=X_features_ohe.columns)
coef_sort = coef.sort_values(ascending=False)[:10]
sns.barplot(x=coef_sort.values , y=coef_sort.index)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd0128e5310>
```



회귀 트리 사용

```
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

# 랜덤 포레스트, GBM, XGBoost, LightGBM model 별로 평가 수행
rf_reg = RandomForestRegressor(n_estimators=500)
gbm_reg = GradientBoostingRegressor(n_estimators=500)
|
for model in [rf_reg, gbm_reg]:
    # XGBoost의 경우 DataFrame이 입력 될 경우 버전에 따라 오류 발생 가능. ndarray로 변환.
    get_model_predict(model, X_train.values, X_test.values, y_train.values, y_test.values, is_expm1=True)

### RandomForestRegressor ####
RMSLE: 0.354, RMSE: 50.371, MAE: 31.139
### GradientBoostingRegressor ####
RMSLE: 0.330, RMSE: 53.330, MAE: 32.738
```

Chapter 06

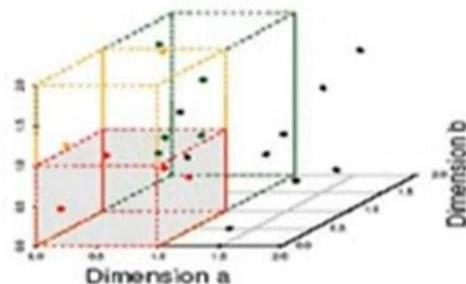
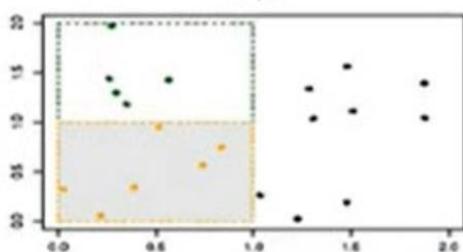
Unsupervised

Learning

(차원 축소)



차원의 저주



특정 입체 공간에 4개의 데이터

차원이 커질수록



- 데이터 포인트들간 거리가 크게 늘어남
- 데이터가 희소화(Sparse) 됨

- 수백~수천개 이상의 피처로 구성된 포인트들간 거리에 기반한 ML 알고리즘이 무력화됨.
- 또한 피처가 많을 경우 개별 피처간에 상관관계가 높아 선형 회귀와 같은 모델에서는 다중 공선성 문제로 모델의 예측 성능이 저하될 가능성이 높음

차원의 축소의 장점

수십~수백개의 피처들을 작은 수의 피처
들로 축소한다면?

- 학습 데이터 크기를 줄여서 학습 시간 절약
- 불필요한 피처들을 줄여서 모델 성능 향상에 기여(주로 이미지 관련 데이터)
- 다차원의 데이터를 3차원 이하의 차원 축소를 통해서 시각적으로 보다 쉽게
데이터 패턴 인지

어떻게 하면 원본 데이터의 정보를 최대한으로 유지한 채로 차원 축소를 수행할 것인가?

피처 선택과 피처 추출

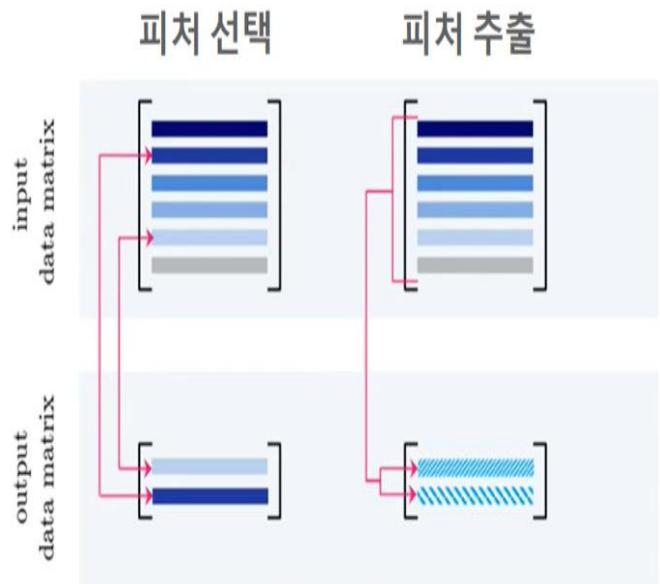
일반적으로 차원 축소는 피처 선택(feature selection)과 피처 추출(feature extraction)로 나눌 수 있습니다.

피처 선택 (Feature Selection)

특정 피처에 종속성이 강한 불필요한 피처는 아예 제거하고, 데이터의 특징을 잘 나타내는 주요 피처만 선택하는 것입니다.

피처 추출 (Feature Extraction)

피처(특성) 추출은 기존 피처를 저차원의 중요 피처로 압축해서 추출하는 것입니다. 이렇게 새롭게 추출된 중요 특성은 기존의 피처를 반영해 압축된 것이지만 새로운 피처로 추출하는 것입니다.



피처 추출 (Feature Extraction)

피처 추출은 기존 피처를 단순 압축이 아닌, 피처를 함축적으로 더 잘 설명할 수 있는 또 다른 공간으로 매핑해 추출하는 것입니다

모의고사 성적

종합 내신 성적

수능성적

봉사활동

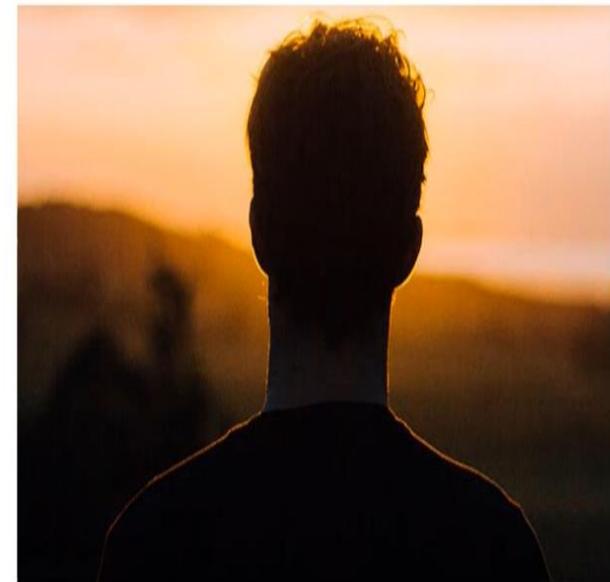
대외활동

수상 경력

학업 성취도

커뮤니케이션

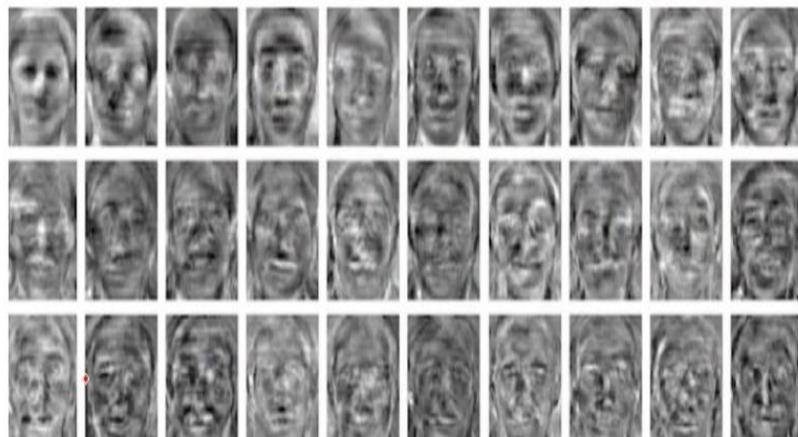
문제 해결력



차원의 축소의 의미

차원 축소는 단순히 데이터의 압축을 의미하는 것이 아닙니다. 더 중요한 의미는 차원 축소를 통해 좀 더 데이터를 잘 설명할 수 있는 잠재적(Latent)인 요소를 추출하는 데에 있습니다

- 추천 엔진
- 이미지 분류 및 변환
- 문서 토픽 모델링



PCA(Principal Component Analysis)의 이해

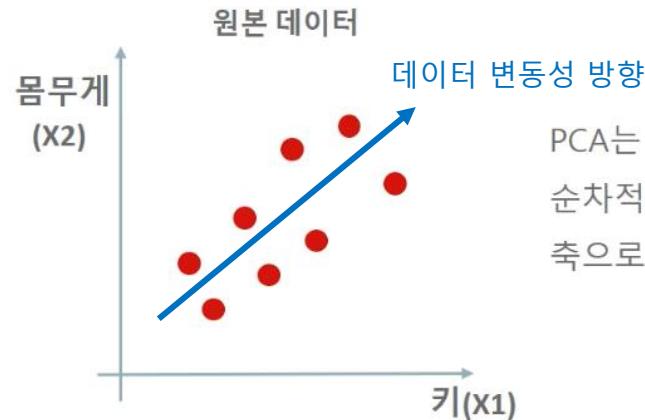
PCA(주성분 분석, Principal Component Analysis)

- 고차원의 원본 데이터를 저 차원의 부분 공간으로 투영하여 데이터를 축소하는 기법
- 예를 들어 10차원의 데이터를 2차원의 부분 공간으로 투영하여 데이터를 축소
- PCA는 원본 데이터가 가지는 데이터 변동성을 가장 중요한 정보로 간주하며 이 변동성에 기반한 원본 데이터 투영으로 차원 축소를 수행

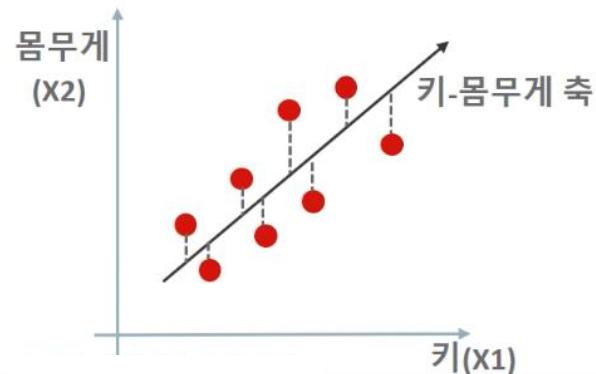


PCA 원리

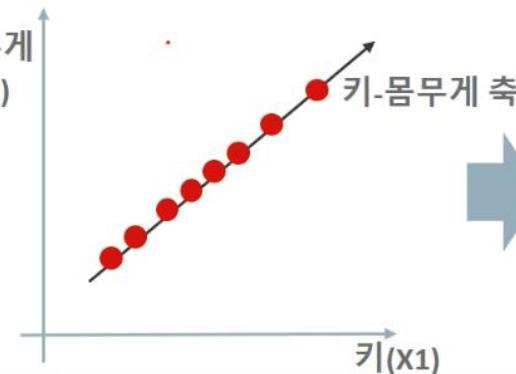
키, 몸무게 2개의 축을 가지는 2차원 원본 데이터



A. 데이터 변동성이 가장 큰 방향으로 축 생성



B. 새로운 축으로 데이터 투영



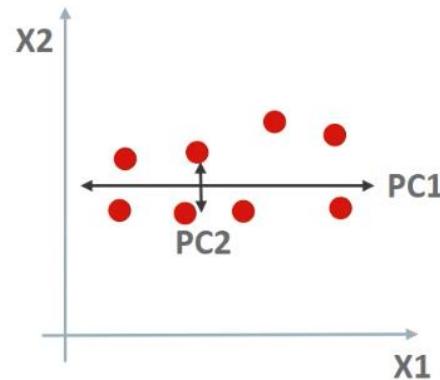
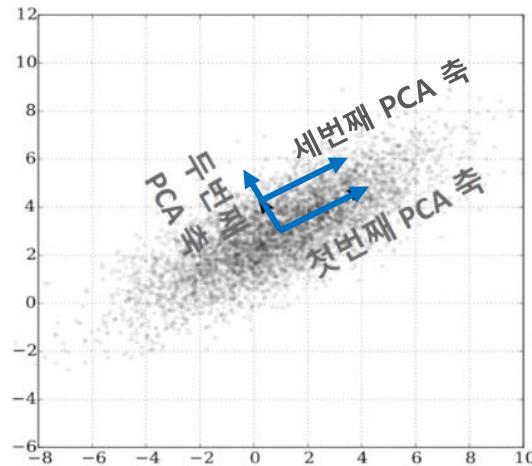
C. 새로운 축 기준으로 데이터 표현



새롭게 생긴 키-몸무게 축에 매핑되는 1차원 데이터로 차원 축소되었다

PCA 원리

PCA는 제일 먼저 원본 데이터에 가장 큰 데이터 변동성(Variance)을 기반으로 첫 번째 벡터 축을 생성하고, 두 번째 축은 첫 번째 축을 제외하고 그 다음으로 변동성이 큰 축을 설정하는 데 이는 첫 번째 축에 직각이 되는 벡터(직교 벡터)축입니다. 세 번째 축은 다시 두 번째 축과 직각이 되는 벡터를 설정하는 방식으로 축을 생성합니다. 이렇게 생성된 벡터 축에 원본 데이터를 투영하면 **벡터 축의 개수만큼의 차원**으로 원본 데이터가 차원 축소됩니다



PCA, 즉 주성분 분석은 이처럼 원본 데이터의 피처 개수에 비해 매우 작은 주성분으로 원본 데이터의 총 변동성을 대부분 설명할 수 있는 분석법입니다

PCA 프로세스

PCA를 선형대수 관점에서 해석해 보면, 입력 데이터의 **공분산 행렬(Covariance Matrix)**을 고유값 분해하고, 이렇게 구한 고유벡터에 입력 데이터를 선형 변환하는 것입니다.



- **고유벡터**는 PCA의 주성분 벡터로서 입력 데이터의 분산이 큰 방향을 나타냅니다.
- **고윳값(eigenvalue)**은 바로 이 고유벡터의 크기를 나타내며, 동시에 입력 데이터의 분산을 나타냅니다.

공분산 행렬

보통 분산은 한 개의 특정한 변수의 데이터 변동을 의미하나, 공분산은 두 변수 간의 변동을 의미합니다. 즉, 사람 키 변수를 X , 몸무게 변수를 Y 라고 하면 공분산 $\text{Cov}(X, Y) > 0$ 은 X (키)가 증가할 때 Y (몸무게)도 증가한다는 의미입니다.

공분산 행렬 예시

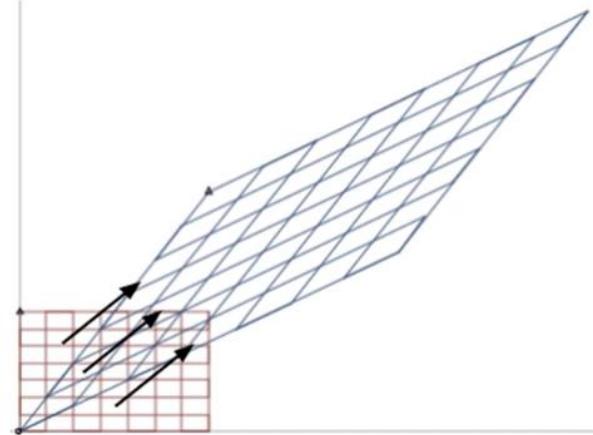
공분산	X	Y	Z
X	3.0 <small>X 분산</small>	-0.71	-0.24
Y	-0.71	4.5 <small>Y 분산</small>	0.28
Z	-0.24	0.28	0.91 <small>Z 분산</small>

공분산 행렬은 여러 변수와 관련된 공분산을 포함하는 정방형 행렬이며 대칭 행렬입니다.

정방행렬은 열과 행이 같은 행렬을 지칭하는데, 정방행렬 중에서 대각 원소를 중심으로 원소 값이 대칭되는 행렬, 즉 $A^T = A$ 인 행렬을 대칭행렬이라고 부릅니다

선형 변환과 고유 벡터/고유값

- 일반적으로 선형 변환은 특정 벡터에 행렬 A 를 곱해 새로운 벡터로 변환하는 것을 의미합니다. 이를 특정 벡터를 하나의 공간에서 다른 공간으로 투영하는 개념으로도 볼 수 있으며, 이 경우 이 행렬을 바로 공간으로 가정하는 것입니다.
- 고유벡터는 행렬 A 를 곱하더라도 방향이 변하지 않고 그 크기만 변하는 벡터를 지칭합니다. 즉, $Ax = ax$ (A 는 행렬, x 는 고유벡터, a 는 스칼라값)입니다. 이 고유벡터는 여러 개가 존재하며, 정방 행렬은 최대 그 차원 수만큼의 고유벡터를 가질 수 있습니다. 예를 들어 2×2 행렬은 두 개의 고유벡터를, 3×3 행렬은 3개의 고유벡터를 가질 수 있습니다. 이렇게 고유벡터는 행렬이 작용하는 힘의 방향과 관계가 있어서 행렬을 분해하는 데 사용됩니다



<https://deeplearning4j.org/kr/eigenvector>

공분산 행렬의 고유값 분해

	x	y	z
x	3.0	-0.71	-0.24
y	-0.71	4.5	0.28
z	-0.24	0.28	0.91

- 공분산 행렬은 정방행렬(Diagonal Matrix)이며 대칭행렬(Symmetric Matrix)입니다. 정방행렬은 열과 행이 같은 행렬을 지칭하는데, 정방행렬 중에서 대각 원소를 중심으로 원소 값이 대칭되는 행렬, 즉 $A^T = A$ 인 행렬을 대칭행렬이라고 부릅니다
- 대칭행렬은 고유값 분해와 관련해 매우 좋은 특성이 있습니다. 대칭행렬은 항상 고유벡터를 직교행렬(orthogonal matrix)로, 고유값을 정방 행렬로 대각화할 수 있다는 것입니다.

$$C = P \sum P^T$$



- P 는 $n \times n$ 의 직교행렬이며, Σ 는 $n \times n$ 정방행렬, P^T 는 행렬 P 의 전치 행렬입니다

공분산 행렬 $C = [e_1 \dots e_n]$ PCA 축 고유 벡터
직교행렬 고유값
정방행렬 고유 벡터
직교행렬의
전치행렬

$$\begin{bmatrix} \lambda_1 & \dots & 0 \\ \dots & \dots & \dots \\ 0 & \dots & \lambda_n \end{bmatrix} \begin{bmatrix} e_1^t \\ \dots \\ e_n^t \end{bmatrix}$$

- 공분산 C 는 고유벡터 직교 행렬, 고유값 정방 행렬 * 고유벡터 직교 행렬의 전치 행렬로 분해됩니다.
- e_i 는 i 번째 고유벡터를, λ_i 는 i 번째 고유벡터의 크기를 의미합니다. 고유 벡터는 바로 PCA의 축입니다.
- e_1 는 가장 분산이 큰 방향을 가진 고유벡터이며, e_2 는 e_1 에 수직이면서 다음으로 가장 분산이 큰 방향을 가진 고유벡터입니다.

PCA 요약

PCA 변환

입력 데이터의 공분산 행렬이 고유벡터와 고유값으로 분해될 수 있으며, 이렇게 분해된 고유벡터를 이용해 입력 데이터를 선형 변환하는 방식

PCA 변환 수행 절차

1. 입력 데이터 세트의 공분산 행렬을 생성합니다.
2. 공분산 행렬의 고유벡터와 고유값을 계산합니다.
3. 고유값이 가장 큰 순으로 K개(PCA 변환 차수만큼)만큼 고유벡터를 추출합니다.
4. 고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터를 변환합니다

사이킷런 PCA 클래스

사이킷런은 PCA를 위해 PCA 클래스를 제공합니다.

`sklearn.decomposition.PCA(n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', random_state=None)`

- `n_components`은 PCA 축의 개수 즉 변환 차원을 의미합니다.
 - PCA를 적용하기 전에 입력 데이터의 개별 피처들을 스케일링해야 합니다. PCA는 여러 피처들의 값을 연산해야 하므로 피처들의 스케일에 영향을 받습니다. 따라서 여러 속성을 PCA로 압축하기 전에 각 피처들의 값을 동일한 스케일로 변환하는 것이 필요합니다. 일반적으로 평균이 0, 분산이 1인 표준 정규 분포로 변환합니다.
- fit.transform으로 PCA 실행**
- PCA 변환이 완료된 사이킷런 PCA 객체는 전체 변동성에서 개별 PCA 컴포넌트별로 차지하는 변동성 비율을 `explained_variance_ratio_` 속성으로 제공합니다.

실습 : 붓꽃 데이터 PCA

데이터 읽어오기

```
from sklearn.datasets import load_iris
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

# 사이킷런 내장 데이터 셋 API 호출
iris = load_iris()

# 넘파이 데이터 셋을 Pandas DataFrame으로 변환
columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
irisDF = pd.DataFrame(iris.data, columns=columns)
irisDF['target'] = iris.target

print(irisDF.shape)
irisDF.head(3)
```

(150, 5)

4개의 피처

품종

0 : setosa
1 : versicolor
2 : virginica

	sepal_length	sepal_width	petal_length	petal_width	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0

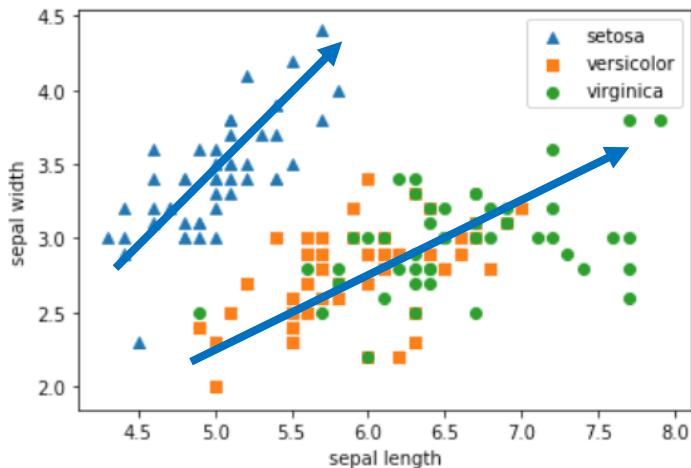
데이터 시각화 확인

2 개의 속성(sepal_length, sepal_width)으로 타겟 산포도 시각화

```
#setosa는 세모, versicolor는 네모, virginica는 동그라미로 표현
markers=['^', 's', 'o'] # 세모, 네모, 동그라미

#setosa의 target 값은 0, versicolor는 1, virginica는 2. 각 target 별로 다른 shape으로 scatter plot
for i, marker in enumerate(markers):
    x_axis_data = irisDF[irisDF['target']==i]['sepal_length']
    y_axis_data = irisDF[irisDF['target']==i]['sepal_width']
    plt.scatter(x_axis_data, y_axis_data, marker=marker, label=iris.target_names[i])

plt.legend()
plt.xlabel('sepal length')
plt.ylabel('sepal width')
plt.show()
```



타겟 값들을 sepal_length, sepal_width 속성 변수들을 이용해서 2차원 평면에 뿌려봄

PCA 수행(n_components=2)

정규화(평균 0, 분산 1)

```
from sklearn.preprocessing import StandardScaler  
  
iris_scaled = StandardScaler().fit_transform(irisDF)
```

PCA 수행(n_components=2)

```
from sklearn.decomposition import PCA  
  
pca = PCA(n_components=2)  
  
#fit( )과 transform( ) 을 호출하여 PCA 변환 데이터 반환  
pca.fit(iris_scaled)  
iris_pca = pca.transform(iris_scaled)  
  
# 차원이 2차원으로 변환된 것 확인  
print(iris_pca.shape)
```

(150, 2) 2차원 PCA가 수행되었다

```
# PCA 변환된 데이터의 컬럼명을 각각 pca_component_1, pca_component_2로 명명  
pca_columns=['pca_component_1','pca_component_2']  
  
irisDF_pca = pd.DataFrame(iris_pca,columns=pca_columns)  
irisDF_pca['target']=iris.target  
irisDF_pca.head(3)
```

PCA가
수행된
데이터

	pca_component_1	pca_component_2	target
0	-2.576120	0.474499	0
1	-2.415322	-0.678092	0
2	-2.659333	-0.348282	0

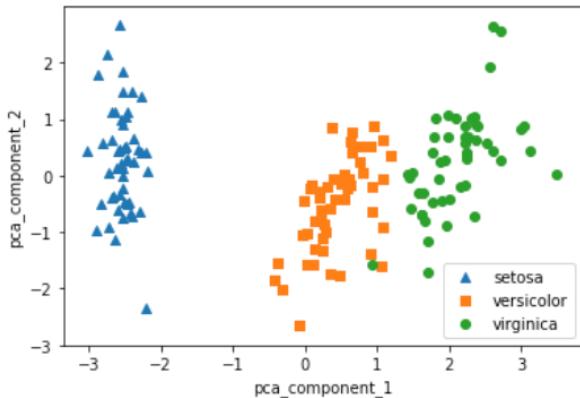
PCA 수행(n_components=2)

PCA 차원 축소된 피처들로 데이터 산포도 시각화

```
#setosa를 세모, versicolor를 네모, virginica를 동그라미로 표시
markers=['^', 's', 'o']

#pca_component_1 을 x축, pc_component_2를 y축으로 scatter plot 수행.
for i, marker in enumerate(markers):
    PCA가 x_axis_data = irisDF_pca[irisDF_pca['target']==i]['pca_component_1']
    수행된 y_axis_data = irisDF_pca[irisDF_pca['target']==i]['pca_component_2']
    데이터 plt.scatter(x_axis_data, y_axis_data, marker=marker, label=iris.target_names[i])

plt.legend()
plt.xlabel('pca_component_1')
plt.ylabel('pca_component_2')
plt.show()
```



PCA(차원 축소) 이전에 비해 데이터들이 명확히 클러스터링된 것을 확인할 수 있다

```
# 각 PCA Component 별 변동성 비율
print(pca.explained_variance_ratio_)
```

```
[0.76740358 0.18282727]
```

전체 변동성의 76%가 Component 1으로 설명될 수 있다

원본 데이터 vs PCA 데이터 간 예측성능 비교

원본 데이터와 PCA 변환된 데이터 간 랜덤포레스트 분류기 예측 성능 비교

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

rcf = RandomForestClassifier(random_state=156)

# 원본 데이터
scores = cross_val_score(rcf, iris.data, iris.target, scoring='accuracy', cv=3)
print(scores)
print(np.mean(scores))
```

```
[0.98 0.94 0.96]
0.96
```

PCA로 차원축소한 데이터

```
pca_X = irisDF_pca[['pca_component_1', 'pca_component_2']]
scores_pca = cross_val_score(rcf, pca_X, iris.target, scoring='accuracy', cv=3)
print(scores_pca)
print(np.mean(scores_pca))
```

```
[0.98 0.98 1. ]
0.9866666666666667 PCA 수행하니 예측 성능이 증가했다
```

PCA로 변환된 데이터가 원본 데이터보다 더 나은 예측 정확도를 보이는 것은 항상 그런 것은 아닙니다.

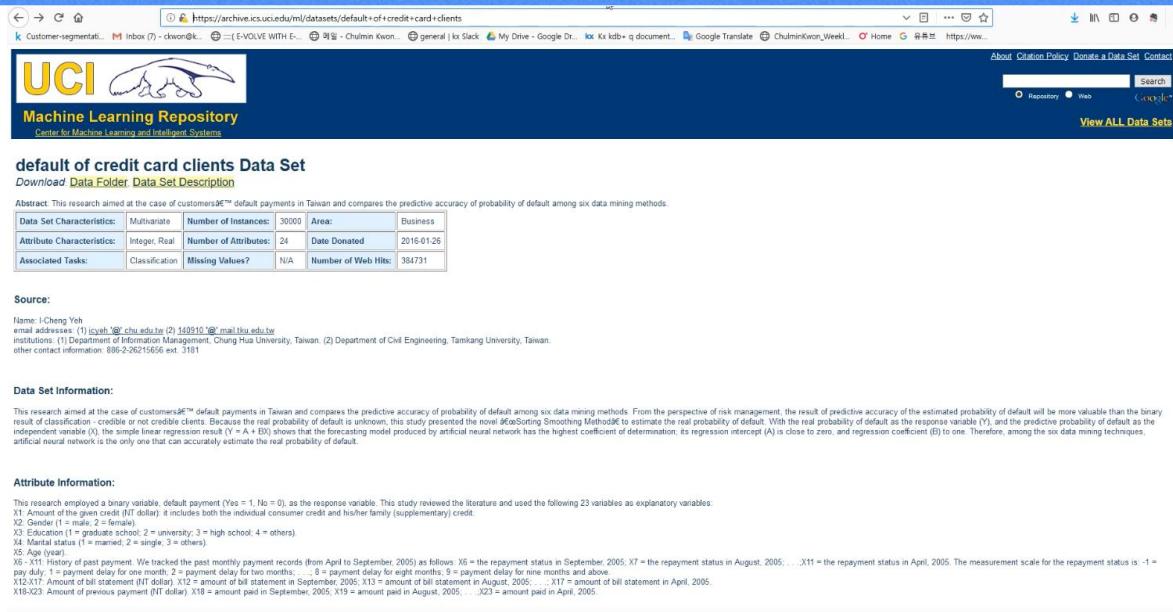
PCA 변환 차원 개수에 따라 예측 성능이 떨어지기도 합니다.

4개의 속성이 2개의 속성이 되어도 예측 성능에 영향을 받지 않을 정도로 PCA 변환이 잘 되었음을 의미합니다.

다만, 고차원 데이터를 저차원으로 변환하면 **직관적으로 이해하기 편하며**, 데이터의 **주축을 이루는 속성**이 무엇인지 파악할 수 있습니다.

실습 : 신용카드 연체 예측 데이터 PCA

신용카드 연체 예측 (UCI credit card default data)



The screenshot shows the UCI Machine Learning Repository homepage with the 'View All Data Sets' button highlighted. Below it, the 'default of credit card clients Data Set' is listed with its details. The dataset summary indicates it's about default payments in Taiwan and compares the predictive accuracy of probability of default among six data mining methods. It includes data set characteristics (Multivariate, 30000 instances, Business area), attribute characteristics (Integer, Real, 24 attributes, 2016-01-26 date), and associated tasks (Classification, Missing Values?, N/A, 304731 web hits). The 'Source' section provides contact information for the researcher, I-Cheng Yeh, and the 'Data Set Information' section details the research aim and methodology.

예제 : credit card 데이터 세트 PCA 변환

```
import pandas as pd

df = pd.read_excel('credit_card.xls', header=1, sheet_name='Data').iloc[0:,1:]

print(df.shape)
df.head(3)
```

(30000, 24)

신용카드 연체 예측

LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_5	...	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1	PAY_AMT2	PAY_AMT3	PAY_AMT4	PAY_AMT5	PAY_AMT6	default payment next month
0	20000	2	2	1	24	2	2	-1	-1	-2	...	0	0	0	0	689	0	0	0	0
1	120000	2	2	2	26	-1	2	0	0	0	...	3272	3455	3261	0	1000	1000	1000	0	2000
2	90000	2	2	2	34	0	0	0	0	0	...	14331	14948	15549	1518	1500	1000	1000	5000	0

3 rows x 24 columns

데이터 전처리 : 컬럼명 변경, 속성/클래스 분류

```
# 컬럼명 변경
df.rename(columns={'PAY_0':'PAY_1', 'default payment next month':'default'}, inplace=True)

# 속성과 클래스로 데이터 분류
y_target = df['default']
X_features = df.drop('default', axis=1)

y_target.value_counts()
```

```
0    23364
1    6636
Name: default, dtype: int64
```

```
X_features.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 23 columns):
 #   Column      Non-Null Count  Dtype  
 ---  --          --          --    
 0   LIMIT_BAL   30000 non-null   int64  
 1   SEX          30000 non-null   int64  
 2   EDUCATION    30000 non-null   int64  
 3   MARRIAGE    30000 non-null   int64  
 4   AGE          30000 non-null   int64  
 5   PAY_1        30000 non-null   int64  
 6   PAY_2        30000 non-null   int64  
 7   PAY_3        30000 non-null   int64  
 8   PAY_4        30000 non-null   int64  
 9   PAY_5        30000 non-null   int64  
 10  PAY_6        30000 non-null   int64  
 11  BILL_AMT1   30000 non-null   int64  
 12  BILL_AMT2   30000 non-null   int64  
 13  BILL_AMT3   30000 non-null   int64  
 14  BILL_AMT4   30000 non-null   int64  
 15  BILL_AMT5   30000 non-null   int64  
 16  BILL_AMT6   30000 non-null   int64  
 17  PAY_AMT1    30000 non-null   int64  
 18  PAY_AMT2    30000 non-null   int64  
 19  PAY_AMT3    30000 non-null   int64  
 20  PAY_AMT4    30000 non-null   int64  
 21  PAY_AMT5    30000 non-null   int64  
 22  PAY_AMT6    30000 non-null   int64  
dtypes: int64(23)
memory usage: 5.3 MB
```

```
y_target.value_counts()
```

```
0    23364
1    6636
Name: default, dtype: int64
```

X1: Amount of the given credit (NT dollar): it includes both the individual credit and his/her family credit.

X2: Gender (1 = male; 2 = female).

X3: Education (1 = graduate school; 2 = university; 3 = high school; 4 = others).

X4: Marital status (1 = married; 2 = single; 3 = others).

X5: Age (year).

과거
지불
금액

X6 - X11: History of past payment.

We tracked the past monthly payment records (from April to September, 2005) as follows: X6 = the repayment status in September, 2005; X7 = the repayment status in August, 2005; . . . ; X11 = the repayment status in April, 2005. The measurement scale for the repayment status is: -1 = pay duly; 1 = payment delay for one month; 2 = payment delay for two months; . . . ; 8 = payment delay for eight months; 9 = payment delay for nine months and above.

과거
정구
금액

X12-X17: Amount of bill statement (NT dollar).

X12 = amount of bill statement in September, 2005; X13 = amount of bill statement in August, 2005; . . . ; X17 = amount of bill statement in April, 2005.

X18-X23: Amount of previous payment (NT dollar).

X18 = amount paid in September, 2005; X19 = amount paid in August, 2005; . . . ; X23 = amount paid in April, 2005.

피처 간 상관관계 살펴보기

히트맵으로 피처 간 상관관계 시각화

```
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

corr = X_features.corr()
plt.figure(figsize=(14,14))
sns.heatmap(corr, annot=True, fmt='.1g')

<matplotlib.axes._subplots.AxesSubplot at 0x7fa0812424d0>
```

과거
지불
금액

과거
청구
금액



과거 지불 금액간 상관관계가 높다

과거 청구 금액간 상관관계는 더 높다

이렇게 상관도가 높은 피처들 간에는 PCA
효율이 좋다

일부 피처들 PCA 변환(n_components=2)

일부 상관도가 높은 피처들(BILL_AMT1~6)을 PCA(n_components=2) 변환 후 변동성 확인

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

#BILL_AMT1 ~ BILL_AMT6 까지 6개의 속성명 생성
cols_bill = ['BILL_AMT'+str(i) for i in range(1,7)]
print('대상 속성명:', cols_bill)

# 2개의 PCA 속성을 가진 PCA 객체 생성하고, explained_variance_ratio_ 계산 위해 fit( ) 호출
scaler = StandardScaler()
df_cols_scaled = scaler.fit_transform(X_features[cols_bill])
pca = PCA(n_components=2)
pca.fit(df_cols_scaled)

print('PCA Component별 변동성:', pca.explained_variance_ratio_)
```

대상 속성명: ['BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6']
PCA Component별 변동성: [0.90555253 0.0509867]

6개의 피처를 2개의 피처로 PCA 변환했을 때 첫번째 컴포넌트가 전체 변동성의 90%를 설명한다.

전체 피처들 PCA 변환(n_components=6)

전체 원본 데이터와 PCA 변환된 데이터 간 랜덤 포레스트 예측 성능 비교

```
# 1. 원본 데이터
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

rcf = RandomForestClassifier(n_estimators=300, random_state=156)

# 원본 데이터일 때 랜덤 포레스트 예측 성능
scores = cross_val_score(rcf, X_features, y_target, scoring='accuracy', cv=3)

print('CV=3 인 경우의 개별 Fold세트별 정확도:', scores)
print('평균 정확도:{0:.4f}'.format(np.mean(scores)))
```

CV=3 인 경우의 개별 Fold세트별 정확도: [0.8083 0.8196 0.8232]
평균 정확도:0.8170

```
# 2. PCA 변환된 데이터
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# 원본 데이터셋에 먼저 StandardScaler 적용
scaler = StandardScaler()
df_scaled = scaler.fit_transform(X_features)

# PCA 변환을 수행하고 랜덤 포레스트 예측 성능
pca = PCA(n_components=7)
df_pca = pca.fit_transform(df_scaled)
scores_pca = cross_val_score(rcf, df_pca, y_target, scoring='accuracy', cv=3)

print('CV=3 인 경우의 PCA 변환된 개별 Fold세트별 정확도:', scores_pca)
print('PCA 변환 데이터 셋 평균 정확도:{0:.4f}'.format(np.mean(scores_pca)))
```

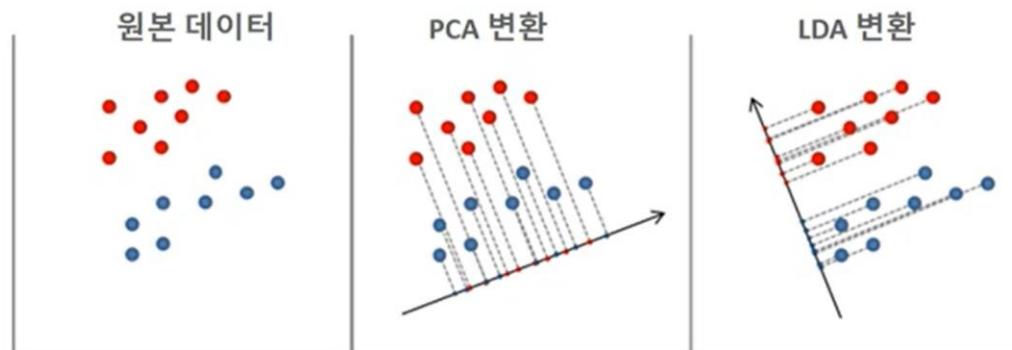
CV=3 인 경우의 PCA 변환된 개별 Fold세트별 정확도: [0.7924 0.8002 0.8024]
PCA 변환 데이터 셋 평균 정확도:0.7983

<n_components 수에 따른 예측 정확도 추이>

7:	0.7983
6:	0.7967
5:	0.7977
4:	0.7913

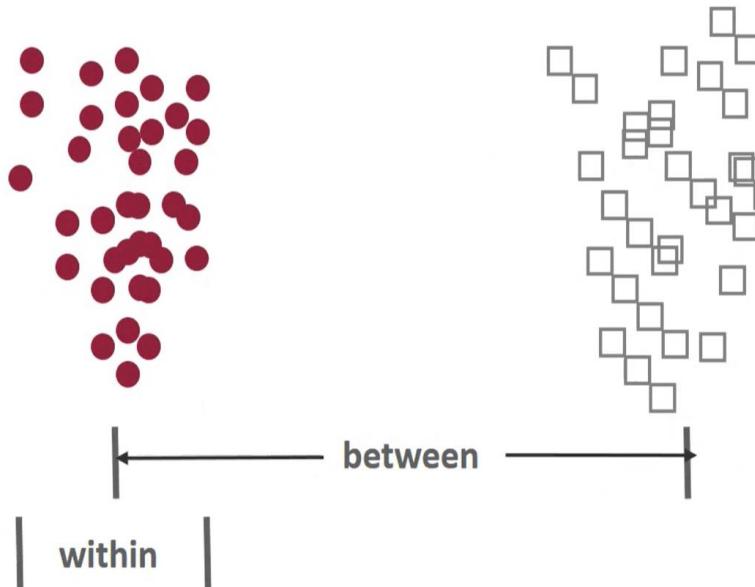
LDA(Linear Discriminant Analysis)

- LDA(Linear Discriminant Analysis)는 선형 판별 분석법으로 불리며, PCA와 매우 유사합니다.
- LDA는 PCA와 유사하게 입력 데이터 세트를 저차원 공간에 투영해 차원을 축소하는 기법이지만, 중요한 차이는 LDA는 지도학습의 분류(Classification)에서 사용하기 쉽도록 개별 클래스를 분별할 수 있는 기준을 최대한 유지하면서 차원을 축소합니다. PCA는 입력 데이터의 변동성의 가장 큰 축을 찾았지만, LDA는 입력 데이터의 결정 값 클래스를 최대한으로 분리할 수 있는 축을 찾습니다.
- LDA는 같은 클래스의 데이터는 최대한 근접해서, 다른 클래스의 데이터는 최대한 떨어뜨리는 축 매핑을 합니다.



LDA(Linear Discriminant Analysis) 차원 축소 방식

LDA는 특정 공간상에서 클래스 분리를 최대화하는 축을 찾기 위해 클래스 간 분산(between-class scatter)과 클래스 내부 분산(within-class scatter)의 비율을 최대화하는 방식으로 차원을 축소합니다. 즉, 클래스 간 분산은 최대한 크게 가져가고, 클래스 내부의 분산은 최대한 작게 가져가는 방식입니다



LDA 절차

일반적으로 LDA를 구하는 스텝은 PCA와 유사하나 가장 큰 차이점은 공분산 행렬이 아니라 앞에서 설명한 클래스 간 분산과 클래스 내부 분산 행렬을 생성한 뒤, 이 행렬에 기반해 고유벡터를 구하고 입력 데이터를 투영한다는 점입니다.

1. 클래스 내부와 클래스 간 분산 행렬을 구합니다. 이 두 개의 행렬은 입력 데이터의 결정 값 클래스별로 개별 피처의 평균 벡터(mean vector)를 기반으로 구합니다.
2. 클래스 내부 분산 행렬을 S_W , 클래스 간 분산 행렬을 S_B 라고 하면 다음 식으로 두 행렬을 고유벡터로 분해할 수 있습니다.

$$S_W^T S_B = [e_1 \ \dots \ e_n] \begin{bmatrix} \lambda_1 & \dots & 0 \\ \dots & \dots & \dots \\ 0 & \dots & \lambda_n \end{bmatrix} \begin{bmatrix} e_1^T \\ \dots \\ e_n^T \end{bmatrix}$$

3. 고유값이 가장 큰 순으로 K개(LDA변환 차수만큼) 추출합니다.
4. 고유값이 가장 큰 순으로 K개(LDA변환 차수만큼) 추출합니다. 고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터를 변환합니다.

실습 LDA

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris

iris = load_iris()
iris_scaled = StandardScaler().fit_transform(iris.data)
```

```
lda = LinearDiscriminantAnalysis(n_components=2)
#fit 훈련시 target값 입력
lda.fit(iris_scaled, iris.target)
iris_lda = lda.transform(iris_scaled)
print(iris_lda.shape)
```

(150, 2)

실습 LDA

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

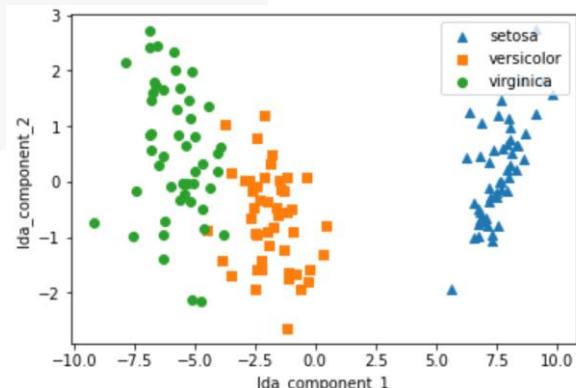
lda_columns=['lda_component_1', 'lda_component_2']
irisDF_lda = pd.DataFrame(iris_lda,columns=lda_columns)
irisDF_lda['target']=iris.target

#setosa는 세모, versicolor는 네모, virginica는 동그라미로 표현
markers=['^', 's', 'o']

#setosa의 target 값은 0, versicolor는 1, virginica는 2. 각 target 별로 다른 shape으로 scatter plot
for i, marker in enumerate(markers):
    x_axis_data = irisDF_lda[irisDF_lda['target']==i]['lda_component_1']
    y_axis_data = irisDF_lda[irisDF_lda['target']==i]['lda_component_2']

    plt.scatter(x_axis_data, y_axis_data, marker=marker, label=iris.target_names[i])

plt.legend(loc='upper right')
plt.xlabel('lda_component_1')
plt.ylabel('lda_component_2')
plt.show()
```



특이값 분해 SVD

대표적인 행렬 분해 방법

고유값 분해

Eigen-Decomposition

$$C = P \sum P^T$$

$$C = [e_1 \dots e_n] \begin{bmatrix} \lambda_1 & \dots & 0 \\ \dots & \dots & \dots \\ 0 & \dots & \lambda_n \end{bmatrix} \begin{bmatrix} e_1^T \\ \dots \\ e_n^T \end{bmatrix}$$

특이값 분해

Singular Value Decomposition

$$A = U \sum V^T$$

- 정방행렬(즉, 행과 열의 크기가 같은 행렬)만을 고유벡터로 분해
- PCA는 분해된 고유 벡터에 원본 데이터를 투영하여 차원 축소

- SVD는 정방행렬뿐만 아니라 행과 열의 크기가 다른 $m \times n$ 행렬도 분해 가능

특이값 분해 SVD

특이값 분해

Singular Value Decomposition

행렬 U 와 V 에 속한 벡터는 특이벡터(singular vector)이며, 모든 특이벡터는 서로 직교하는 성질을 가집니다

$$A = U \Sigma V^T$$

왼쪽 직교행렬
대각 행렬
오른쪽 직교행렬

$$U^T U = I$$

$$V^T V = I$$

Σ 는 대각행렬이며, 행렬의 대각에 위치한 값만 0이 아니고 나머지 위치의 값은 모두 0입니다.

Σ 이 위치한 0이 아닌 값이 바로 행렬 A 의 특이값입니다

SVD 유형

Full SVD

$$A = U \Sigma V'$$

$m \times n$ $m \times m$ $m \times n$ $n \times n$

Compact SVD

비대각 부분과 대각 원소가 0인 부분을 제거

$$A = U \Sigma V'$$

$m \times n$ $m \times p$ $p \times n$

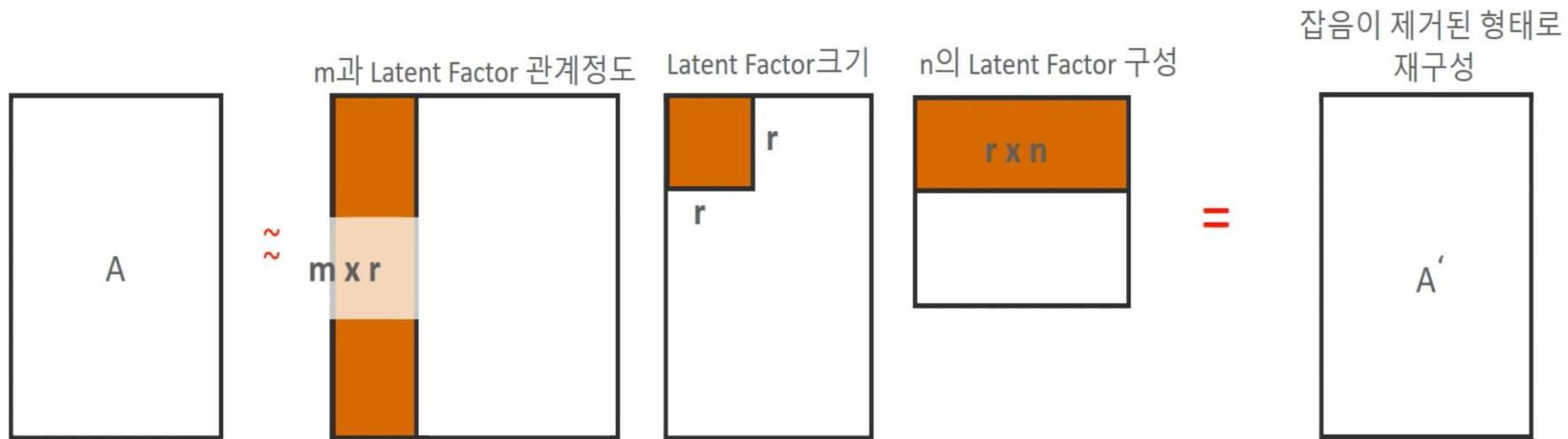
Truncated SVD

대각 원소 가운데 **상위 r 개만** 추출하여 차원 축소

$$A \approx \text{상위 } r \text{ 개만 추출하여 차원 축소}$$

$m \times n$ $r \times n$

Truncated SVD 행렬 분해 의미



SVD는 차원 축소를 위한 행렬 분해를 통해 **Latent Factor(잠재 요인)**를 찾을 수 있는데 이렇게 찾아진 Latent Factor는 많은 분야에 활용 (추천 엔진, 문서의 잠재 의미 분석 등)

SVD로 차원 축소 행렬 분해된 후 다시 분해된 행렬을 이용하여 원복된 데이터 셋은 잡음(Noise)이 제거된 형태로 재 구성될 수 있음

사이킷런에서는 Truncated SVD로 차원을 축소할 때 원본 데이터에 $U \Sigma$ 를 적용하여 차원 축소

SVD 활용

- 이미지 압축/변환
- 추천 엔진
- 문서 잠재 의미 분석
- 의사 역행렬을 통한 모델 예측

실습

```
# numpy의 svd 모듈 import
import numpy as np
from numpy.linalg import svd

# 4X4 Random 행렬 a 생성
np.random.seed(121)
a = np.random.randn(4,4)
print(np.round(a, 3))
```

```
[[-0.212 -0.285 -0.574 -0.44 ]
 [-0.33   1.184  1.615  0.367]
 [-0.014  0.63   1.71   -1.327]
 [ 0.402 -0.191  1.404 -1.969]]
```

실습

```
U, Sigma, Vt = svd(a)
print(U.shape, Sigma.shape, Vt.shape)
print('U matrix:', np.round(U, 3))
print('Sigma Value:', np.round(Sigma, 3))
print('V transpose matrix:', np.round(Vt, 3))
```

```
(4, 4) (4,) (4, 4)
U matrix:
[[ -0.079 -0.318  0.867  0.376]
 [ 0.383  0.787  0.12   0.469]
 [ 0.656  0.022  0.357 -0.664]
 [ 0.645 -0.529 -0.328  0.444]]
Sigma Value:
[3.423 2.023 0.463 0.079]
V transpose matrix:
[[ 0.041  0.224  0.786 -0.574]
 [-0.2    0.562  0.37   0.712]
 [-0.778  0.395 -0.333 -0.357]
 [-0.593 -0.692  0.366  0.189]]
```

*Sigma*를 다시 0을 포함한 대칭행렬로 변환

```
Sigma_mat = np.diag(Sigma)
a_ = np.dot(np.dot(U, Sigma_mat), Vt)
print(np.round(a_, 3))
```

```
[[ -0.212 -0.285 -0.574 -0.44 ]
 [ -0.33   1.184  1.615  0.367]
 [ -0.014  1.63   1.71   -1.327]
 [  0.402 -0.191  1.404 -1.969]]
```

데이터 의존도가 높은 원본 데이터 행렬 생성 for compact SVD

```
a[2] = a[0] + a[1]
a[3] = a[0]
print(np.round(a,3))
```

```
[[ -0.212 -0.285 -0.574 -0.44 ]
 [ -0.33   1.184  1.615  0.367]
 [ -0.542  0.899  1.041 -0.073]
 [ -0.212 -0.285 -0.574 -0.44 ]]
```

다시 SVD를 수행하여 Sigma 값 확인

```
U, Sigma, Vt = svd(a)
print(U.shape, Sigma.shape, Vt.shape)
print('Sigma Value:', np.round(Sigma,3))
```

(4, 4) (4,) (4, 4)
Sigma Value:
[2.663 0.807 0. 0.]

```

# U 행렬의 경우는 Sigma와 내적을 수행하므로 Sigma의 앞 2행에 대응되는 앞 2열만 추출
U_ = U[:, :2]
Sigma_ = np.diag(Sigma[:2])
# V 전치 행렬의 경우는 앞 2행만 추출
Vt_ = Vt[:2]
print(U_.shape, Sigma_.shape, Vt_.shape)
# U, Sigma, Vt의 내적을 수행하며, 다시 원본 행렬 복원
a_ = np.dot(np.dot(U_, Sigma_), Vt_)
print(np.round(a_, 3))

```

```

(4, 2) (2, 2) (2, 4)
[[ -0.212 -0.285 -0.574 -0.44 ]
 [ -0.33   1.184  1.615  0.367]
 [ -0.542  0.899  1.041 -0.073]
 [ -0.212 -0.285 -0.574 -0.44 ]]

```

Truncated SVD 를 이용한 행렬 분해

```
import numpy as np
from scipy.sparse.linalg import svds
from scipy.linalg import svd

# 원본 행렬을 출력하고, SVD를 적용할 경우 U, Sigma, Vt 의 차원 확인
np.random.seed(121)
matrix = np.random.random((6, 6))
print('원본 행렬:', matrix)
U, Sigma, Vt = svd(matrix, full_matrices=False)
print('분해 행렬 차원:', U.shape, Sigma.shape, Vt.shape)
print('Sigma값 행렬:', Sigma)

# Truncated SVD로 Sigma 행렬의 특이값을 4개로 하여 Truncated SVD 수행.
num_components = 4
U_tr, Sigma_tr, Vt_tr = svds(matrix, k=num_components)
print('Truncated SVD 분해 행렬 차원:', U_tr.shape, Sigma_tr.shape, Vt_tr.shape)
print('Truncated SVD Sigma값 행렬:', Sigma_tr)
matrix_tr = np.dot(np.dot(U_tr, np.diag(Sigma_tr)), Vt_tr) # output of TruncatedSVD

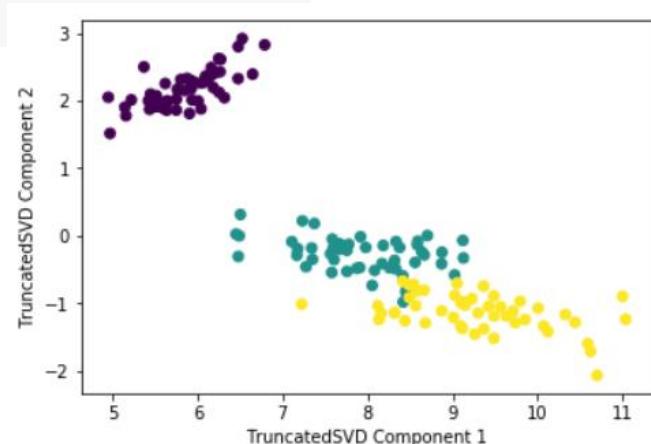
print('Truncated SVD로 분해 후 복원 행렬:', matrix_tr)
```

사이킷런 TruncatedSVD 클래스를 이용한 변환

```
from sklearn.decomposition import TruncatedSVD, PCA
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
%matplotlib inline

iris = load_iris()
iris_ftrs = iris.data
# 2개의 주요 component로 TruncatedSVD 변환
tsvd = TruncatedSVD(n_components=2)
tsvd.fit(iris_ftrs)
iris_tsvd = tsvd.transform(iris_ftrs)

# Scatter plot 2차원으로 TruncatedSVD 변환 된 데이터 표현. 품종은 색깔로 구분
plt.scatter(x=iris_tsvd[:,0], y=iris_tsvd[:,1], c=iris.target)
plt.xlabel('TruncatedSVD Component 1')
plt.ylabel('TruncatedSVD Component 2')
```



```

from sklearn.preprocessing import StandardScaler

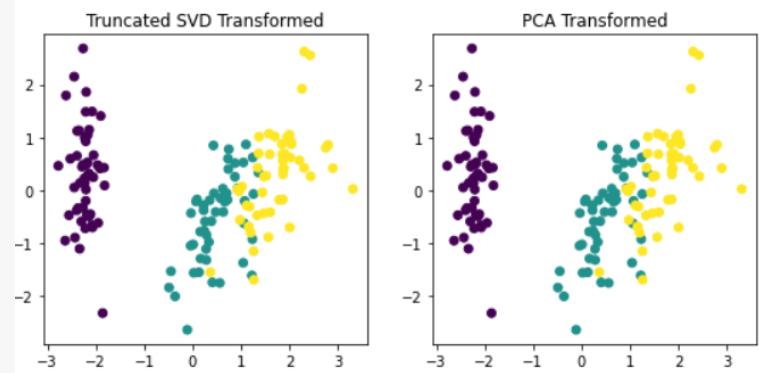
# iris 데이터를 StandardScaler로 변환
scaler = StandardScaler()
iris_scaled = scaler.fit_transform(iris_ftrs)

# 스케일링된 데이터를 기반으로 TruncatedSVD 변환 수행
tsvd = TruncatedSVD(n_components=2)
tsvd.fit(iris_scaled)
iris_tsvd = tsvd.transform(iris_scaled)

# 스케일링된 데이터를 기반으로 PCA 변환 수행
pca = PCA(n_components=2)
pca.fit(iris_scaled)
iris_pca = pca.transform(iris_scaled)

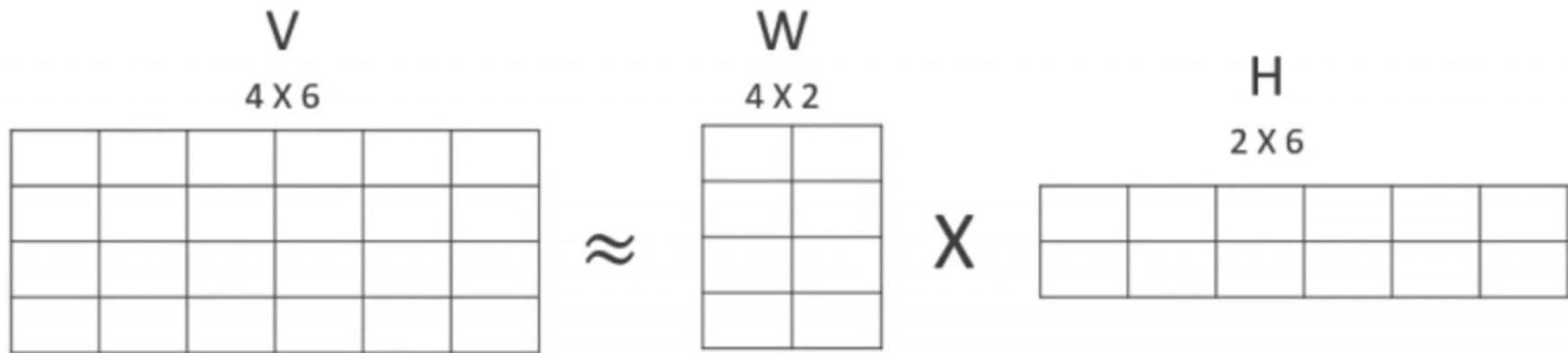
# TruncatedSVD 변환 데이터를 왼쪽에, PCA변환 데이터를 오른쪽에 표현
fig, (ax1, ax2) = plt.subplots(figsize=(9,4), ncols=2)
ax1.scatter(x=iris_tsvd[:,0], y=iris_tsvd[:,1], c=iris.target)
ax2.scatter(x=iris_pca[:,0], y=iris_pca[:,1], c=iris.target)
ax1.set_title('Truncated SVD Transformed')
ax2.set_title('PCA Transformed')

```



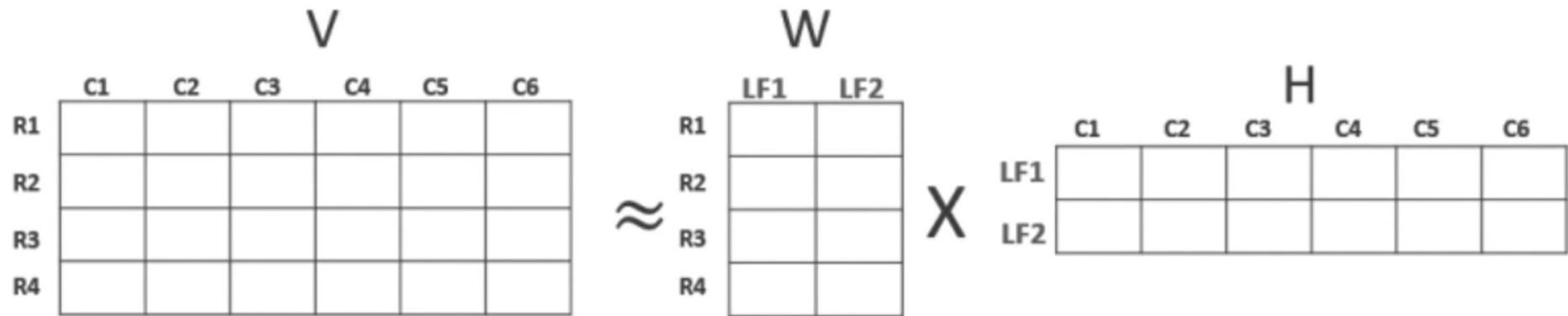
NMF(Non Negative Matric Factorization)

NMF는 원본 행렬 내의 모든 원소 값이 모두 양수(0 이상)라는 게 보장되면 다음과 같이 좀 더 간단하게 두 개의 기반 양수 행렬로 분해될 수 있는 기법을 지칭합니다.



행렬분해 (Matrix Factorization)

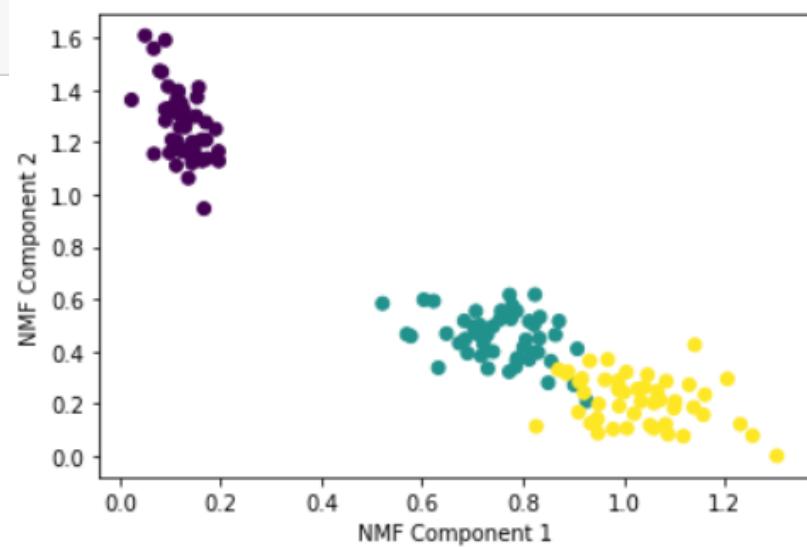
행렬 분해(Matrix Factorization)는 일반적으로 SVD와 같은 행렬 분해 기법을 통칭하는 것입니다. 이처럼 행렬 분해를 하게 되면 W 행렬과 H 행렬은 일반적으로 길고 가는 행렬 W(즉, 원본 행렬의 행 크기와 같고 열 크기보다 작은 행렬)와 작고 넓은 행렬 H(원본 행렬의 행 크기보다 작고 열 크기와 같은 행렬)로 분해됩니다. 이렇게 분해된 행렬은 Latent Factor(잠재 요소)를 특성으로 가지게 됩니다. 분해 행렬 W는 원본 행에 대해서 이 잠재 요소의 값이 얼마나 되는지에 대응하며, 분해 행렬 H는 이 잠재 요소가 원본 열(즉, 원본 속성)로 어떻게 구성됐는지를 나타내는 행렬입니다.



실습

```
from sklearn.decomposition import NMF
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
%matplotlib inline

iris = load_iris()
iris_ftrs = iris.data
nmf = NMF(n_components=2)
nmf.fit(iris_ftrs)
iris_nmf = nmf.transform(iris_ftrs)
plt.scatter(x=iris_nmf[:,0], y=iris_nmf[:,1], c=iris.target)
plt.xlabel('NMF Component 1')
plt.ylabel('NMF Component 2')
```



Chapter 07

Unsupervised

Learning

(군집화)

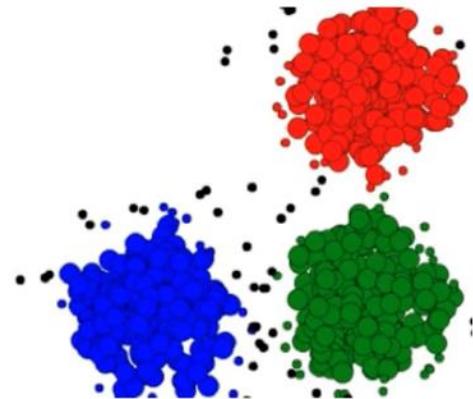


군집화란

군집화(clustering)

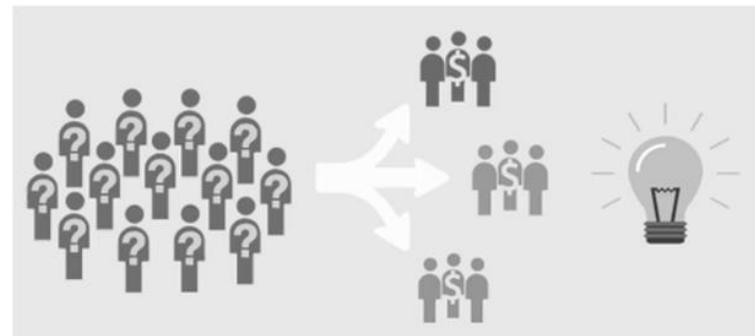
데이터 포인트들을 별개의 군집으로 그룹화 하는것을 의미합니다.

유사성이 높은 데이터들을 동일한 그룹으로 분류하고 서로 다른 군집들이
상이성을 가지도록 그룹화 합니다.



군집화 활용 분야

- 고객, 마켓, 브랜드, 사회 경제 활동 세분화(Segmentation)
- Image 검출, 세분화, 트랙킹
- 이상 검출(Abnormality detection)



어떻게 유사성을 정의할 것인가?

군집화 알고리즘 종류

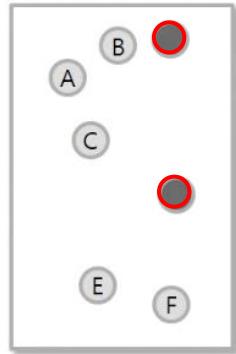
- K-Means centroid(군집 중심점) 기반
- Mean Shift centroid(군집 중심점) 기반
- Gaussian Mixture Model 데이터 정규분포 기반
- DBSCAN 데이터 밀도 기반

K-Means clustering

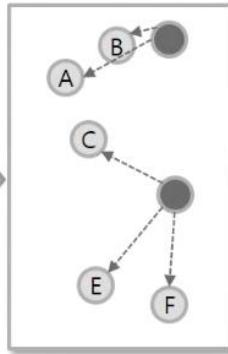
K-Means clustering

군집 중심점(Centroid) 기반 클러스터링

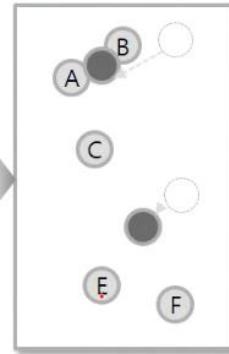
2개의 군집 중심점을 설정



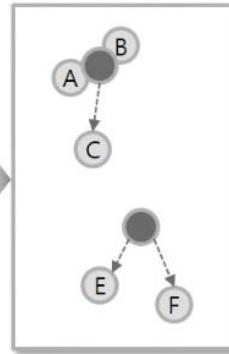
각 데이터들은 가장 가까운 중심점에 소속.



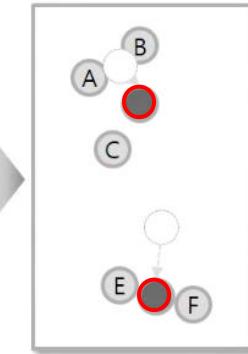
중심점에 할당된 데이터들의 평균 중심으로 중심점 이동



각 데이터들은 이동된 중심점 기준으로 가장 가까운 중심점에 소속



다시 중심점에 할당된 데이터들의 평균 중심으로 중심점 이동



중심점을 이동하였지만 데이터들의 중심점 소속 변경이 없으면 군집화 완료

A,B,C,D,E 는 데이터 포인트이고 ● 군집 중심점

K-Means의 장점/단점

장점

- 일반적인 군집화에서 가장 많이 활용되는 알고리즘입니다.
- 알고리즘이 쉽고 간결합니다.
- 대용량 데이터에도 활용이 가능합니다.

단점

- 거리 기반 알고리즘으로 속성의 개수가 매우 많을 경우 군집화 정확도가 떨어집니다(이를 위해 PCA로 차원 축소를 적용해야 할 수도 있습니다). 차원의 저주가 생길 수도 있다
- 반복을 수행하는데, 반복 횟수가 많을 경우 수행 시간이 느려집니다
- 이상치(Outlier) 데이터에 취약합니다.

사이킷런 K-Means 클래스

사이킷런 패키지는 K-평균을 구현하기 위해 KMeans 클래스를 제공합니다. KMeans 클래스는 다음과 같은 초기화 파라미터를 가지고 있습니다.

```
class sklearn.cluster.KMeans(n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001, precompute_distances='auto',  
    verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto')
```

주요 파라미터

- KMeans 초기화 파라미터 중 가장 중요한 파라미터는 `n_clusters`이며, 이는 군집화할 개수, 즉 군집 중심점의 개수를 의미합니다.
- `init`는 초기에 군집 중심점의 좌표를 설정할 방식을 말하며 보통은 임의로 중심을 설정하지 않고 일반적으로 k-means++방식으로 최초 설정합니다.
- `max_iter`는 최대 반복 횟수이며, 이 횟수 이전에 모든 데이터의 중심점 이동이 없으면 종료합니다.

주요 속성

- `labels_`: 각 데이터 포인트가 속한 군집 중심점 레이블입니다.
- `cluster_centers_`: 각 군집 중심점 좌표(Shape는 [군집 개수, 피쳐 개수]). 이를 이용하면 군집 중심점 좌표가 어디인지 시각화할 수 있습니다.

실습 : 붓꽃 데이터 K-Means clustering

```
from sklearn.preprocessing import scale
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
%matplotlib inline

iris = load_iris()

# 피처 데이터만 별도로 저장
irisDF = pd.DataFrame(data=iris.data, columns=['sepal_length', 'sepal_width', 'petal_length', 'petal_width'])

print(irisDF.shape)
irisDF.head(3)

(150, 4)
```

	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2

붓꽃 데이터에 Kmeans 군집화 수행

피처 데이터에 Kmeans 군집화 수행

```
# kmeans 객체 생성
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=300, random_state=0)

# 붓꽃 데이터에 군집화 수행
kmeans.fit(irisDF)

KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=3, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=0, tol=0.0001, verbose=0)
```

```
# 각 데이터들마다 centroid(군집 중심점) 할당됨  
print(kmeans.labels_)

irisDF[ 'cluster' ]=kmeans.labels
```

```
# 타겟 별 군집 중심점 확인
irisDF['target'] = iris.target

iris_result = irisDF.groupby(['target', 'cluster'])['sepal_length'].count()
print(iris_result)
```

```
target  cluster
0        1          50
1        0           2
                 2          48
2        0          36
                 2          14
Name: sepal length, dtype: int64
```

군집화 결과 시각화(PCA 2차원 변환)

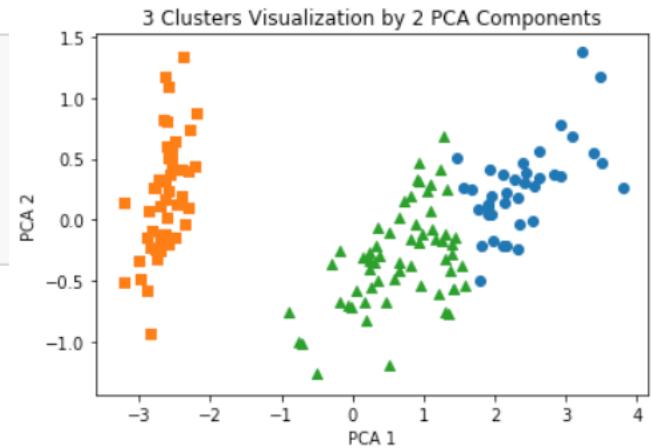
2차원 평면에 데이터 군집화된 결과 나타내기 위해 2차원 PCA로 차원 축소

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
pca_transformed = pca.fit_transform(iris.data)

irisDF[ 'pca_x' ] = pca_transformed[:,0]
irisDF[ 'pca_y' ] = pca_transformed[:,1]
irisDF.head(3)
```

	sepal_length	sepal_width	petal_length	petal_width	cluster	target	pca_x	pca_y
0	5.1	3.5	1.4	0.2	1	0	-2.684126	0.319397
1	4.9	3.0	1.4	0.2	1	0	-2.714142	-0.177001
2	4.7	3.2	1.3	0.2	1	0	-2.888991	-0.144949



```
# cluster 값 0, 1, 2 인 경우마다 별도의 Index로 추출
marker0_ind = irisDF[irisDF['cluster']==0].index
marker1_ind = irisDF[irisDF['cluster']==1].index
marker2_ind = irisDF[irisDF['cluster']==2].index

# cluster 값 0, 1, 2에 해당하는 Index로 각 cluster 레벨의 pca_x, pca_y 값 추출. o, s, ^ 로 marker 표시
plt.scatter(x=irisDF.loc[marker0_ind,'pca_x'], y=irisDF.loc[marker0_ind,'pca_y'], marker='o')
plt.scatter(x=irisDF.loc[marker1_ind,'pca_x'], y=irisDF.loc[marker1_ind,'pca_y'], marker='s')
plt.scatter(x=irisDF.loc[marker2_ind,'pca_x'], y=irisDF.loc[marker2_ind,'pca_y'], marker='^')

plt.xlabel('PCA 1')
plt.ylabel('PCA 2')
plt.title('3 Clusters Visualization by 2 PCA Components')
plt.show()
```

K-means 수행 후 개별 클러스터의 군집 중심 시각화

Clustering 알고리즘 테스트를 위한 데이터 생성

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
%matplotlib inline

# 클러스터링할 데이터 생성 - make_blobs (생성할 데이터 200개, 데이터 피쳐 갯수 2개, 군집 개수 3개, 데이터 표준편차 0.8)
X, y = make_blobs(n_samples=200, n_features=2, centers=3, cluster_std=0.8, random_state=0)
print(X.shape, y.shape)

(200, 2) (200,)
```

```
# y target 값의 분포를 확인
unique, counts = np.unique(y, return_counts=True)
print(unique, counts)
```

```
[0 1 2] [67 67 66] 총 200개 데이터  
타겟값: 0, 1, 2
```

```
import pandas as pd

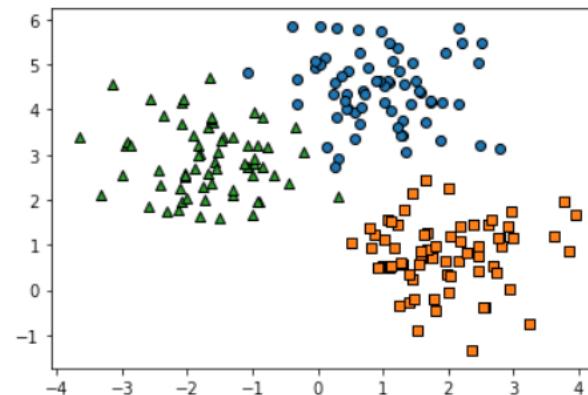
clusterDF = pd.DataFrame(data=X, columns=['ftr1', 'ftr2'])
clusterDF['target'] = y

print(clusterDF.shape)
clusterDF.head(3)
```

```
(200, 3)
```

	ftr1	ftr2	target
0	-1.692427	3.622025	2
1	0.697940	4.428867	0
2	1.100228	4.606317	0

```
# make_blobs로 만들어진 데이터 시각화
target_list = np.unique(y)
# 각 target별 scatter plot 의 marker 값들.
markers=['o', 's', '^', 'p', 'D', 'h', 'x']
# 3개의 cluster 영역으로 구분한 데이터 셋을 생성했으므로 target_list는 [0,1,2]
# target==0, target==1, target==2 로 scatter plot을 marker별로 생성.
for target in target_list:
    target_cluster = clusterDF[clusterDF['target']==target]
    plt.scatter(x=target_cluster['ftr1'], y=target_cluster['ftr2'],
                edgecolor='k', marker=markers[target])
plt.show()
```



K-means 수행 후 개별 클러스터의 군집 중심 시각화

```
# K-means 군집화 수행하고 개별 클러스터의 군집 중심 시각화
```

```
# KMeans 객체를 이용하여 X 데이터를 K-Means 클러스터링 수행
```

```
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=200, random_state=0)
```

```
cluster_labels = kmeans.fit_predict(X)
```

```
clusterDF['kmeans_label'] = cluster_labels
```

```
#cluster centers 는 개별 클러스터의 중심 위치 좌표 시각화를 위해 추출
```

```
centers = kmeans.cluster_centers_
```

```
unique_labels = np.unique(cluster_labels)
```

```
markers=['o', 's', '^', 'P', 'D', 'H', 'x']
```

```
# 군집된 label 유형별로 iteration 하면서 marker 별로 scatter plot 수행.
```

```
for label in unique_labels:
```

```
    label_cluster = clusterDF[clusterDF['kmeans_label']==label]
```

```
    center_x_y = centers[label]
```

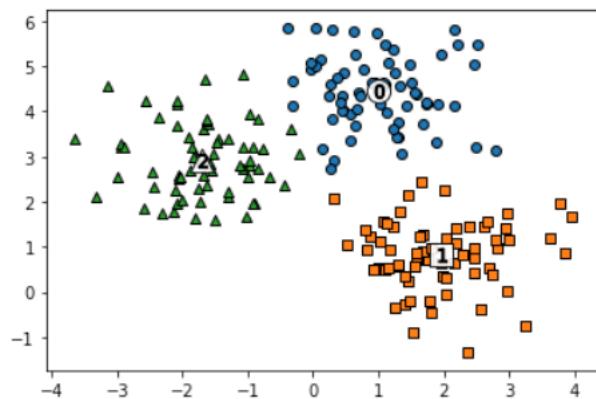
```
    plt.scatter(x=label_cluster['ftr1'], y=label_cluster['ftr2'], edgecolor='k',  
                marker=markers[label] )
```

```
# 군집별 중심 위치 좌표 시각화
```

```
plt.scatter(x=center_x_y[0], y=center_x_y[1], s=200, color='white',  
            alpha=0.9, edgecolor='k', marker=markers[label])
```

```
plt.scatter(x=center_x_y[0], y=center_x_y[1], s=70, color='k', edgecolor='k',  
            marker='$_d$' % label)
```

```
plt.show()
```



kmeans 군집화 수행

군집 중심 위치

kmeans.cluster_centers_

```
array([[ 0.990103 ,  4.44666506],  
       [ 1.95763312,  0.81041752],  
       [-1.70636483,  2.92759224]])
```

```
print(clusterDF.groupby('target')['kmeans_label'].value_counts())
```

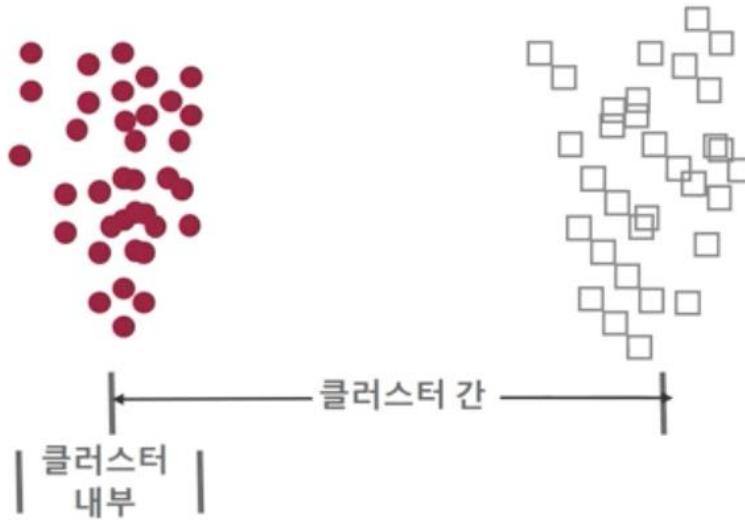
target	kmeans_label	value_counts
0	0	66
0	2	1
1	1	67
2	2	65
2	1	1

Name: kmeans_label, dtype: int64

클러스터 별로 잘 분류가 된 것을 확인할 수 있다

군집 평가 - 실루엣 분석

군집 평가 – 실루엣 분석



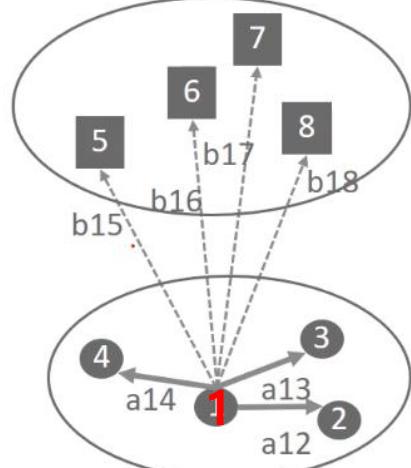
다른 군집과의 거리는 떨어져 있고 동일
군집끼리의 데이터는 서로 가깝게

- 실루엣 분석은 각 군집 간의 거리가 얼마나 효율적으로 분리돼 있는지를 나타냅니다.
- 실루엣 분석은 개별 데이터가 가지는 군집화 지표인 실루엣 계수(silhouette coefficient)를 기반으로 합니다.
- 개별 데이터가 가지는 실루엣 계수는 해당 데이터가 같은 군집 내의 데이터와 얼마나 가깝게 군집화돼 있고, 다른 군집에 있는 데이터와는 얼마나 멀리 분리돼 있는지를 나타내는 지표입니다.

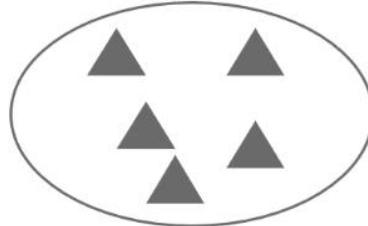
실루엣 계수

실루엣 계수(silhouette coefficient) : 개별 데이터가 가지는 군집화 지표

Cluster B
(Cluster A의 1번 데이터에서 가장 가까운 타 클러스터)



Cluster C



실루엣 계수

$$s(i) = \frac{(b(i) - a(i))}{\max(a(i), b(i))}$$

- a_{ij} 는 i 번째 데이터에서 자신이 속한 클러스터내의 다른 데이터 포인트 까지의 거리.
즉 a_{12} 는 1번 데이터에서 2번 데이터 까지의 거리
- a_i 는 i 번째 데이터에서 자신이 속한 클러스터내의 다른 데이터 포인트들의 거리 평균. 즉 $a_i = \text{평균}(a_{12}, a_{13}, a_{14})$
- b_i 는 i 번째 데이터에서 가장 가까운 타 클러스터내의 다른 데이터 포인트들의 거리 평균. 즉 $b_i = \text{평균}(b_{15}, b_{16}, b_{17}, b_{18})$
- 두 군집 간의 거리가 얼마나 떨어져 있는가의 값은 $b(i) - a(i)$ 이며 이 값을 정규화하기 위해 $\text{MAX}(a(i), b(i))$ 값으로 나눕니다.
- 실루엣 계수는 -1에서 1 사이의 값을 가지며, 1로 가까워질수록 근처의 군집과 더 멀리 떨어져 있다는것이고 0에 가까울수록 근처의 군집과 가까워진다는 것입니다. - 값은 아예 다른 군집에 데이터 포인트가 할당됐음을 뜻합니다

실루엣 계수는 1에 가까울수록 군집화가 잘 되었다는 뜻

실루엣 계수가 -값이라는 것은 해당 데이터의 군집화가 잘못 되었다는 뜻

사이킷런 실루엣분석 API

개별 데이터 군집 거리 계산법

사이킷런 실루엣 분석 API

- `sklearn.metrics.silhouette_samples(X, labels, metric='euclidean', **kwds)`: 인자로 X feature 데이터 세트와 각 피처 데이터 세트가 속한 군집 레이블 값인 labels 데이터를 입력해주면 각 데이터 포인트의 실루엣 계수를 계산해 반환합니다.
- `sklearn.metrics.silhouette_score(X, labels, metric='euclidean', sample_size=None, **kwds)`: 인자로 X feature 데이터 세트와 각 피처 데이터 세트가 속한 군집 레이블 값인 labels 데이터를 입력해주면 전체 데이터의 실루엣계수 값을 평균해 반환합니다. 즉, `np.mean(silhouette_samples())`입니다. 일반적으로 이 값이 높을수록 군집화가 어느정도 잘 됐다고 판단할 수 있습니다. 하지만 무조건 이 값이 높다고 해서 군집화가 잘 됐다고 판단할 수는 없습니다.

실루엣 분석에 기반한 좋은 군집 기준

- 전체 실루엣 계수의 평균값, 즉 사이킷런의 `silhouette_score()` 값은 0 ~ 1사이의 값을 가지며, 1에 가까울수록 좋습니다.
- 하지만 전체 실루엣 계수의 평균값과 더불어 **개별 군집의 평균값의 편차가 크지 않아야 합니다**. 즉, 개별 군집의 실루엣 계수 평균값이 전체 실루엣 계수의 평균값에서 크게 벗어나지 않는 것이 중요합니다. 만약 전체 실루엣 계수의 평균값은 높지만, 특정 군집의 실루엣 계수 평균값만 유난히 높고 다른 군집들의 실루엣 계수 평균값은 낮으면 좋은 군집화 조건이 아닙니다.

실습 : 붓꽃 데이터에서 실루엣 계수 계산

```
from sklearn.preprocessing import scale
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans

# 실루엣 분석 metric 값을 구하기 위한 API 추가
from sklearn.metrics import silhouette_samples, silhouette_score
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

%matplotlib inline

iris = load_iris()

feature_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
irisDF = pd.DataFrame(data=iris.data, columns=feature_names)

# kmeans 군집화 수행
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=300, random_state=0).fit(irisDF)
# 데이터당 클러스터값 할당
irisDF['cluster'] = kmeans.labels_

print(irisDF.shape)
irisDF.head()
```

(150, 5)

	sepal_length	sepal_width	petal_length	petal_width	cluster
0	5.1	3.5	1.4	0.2	1
1	4.9	3.0	1.4	0.2	1
2	4.7	3.2	1.3	0.2	1
3	4.6	3.1	1.5	0.2	1
4	5.0	3.6	1.4	0.2	1

데이터 별 cluster가 할당된 상태

실습 : 붓꽃 데이터에서 실루엣 계수 계산

```
# iris 의 모든 개별 데이터에 실루엣 계수값을 구함.
score_samples = silhouette_samples(iris.data, irisDF['cluster'])
print('silhouette_samples( ) return 값의 shape', score_samples.shape)

# irisDF에 실루엣 계수 컬럼 추가
irisDF['silhouette_coeff'] = score_samples

# 모든 데이터의 평균 실루엣 계수값을 구함.
average_score = silhouette_score(iris.data, irisDF['cluster'])
print('붓꽃 데이터셋 Silhouette Analysis Score:{0:.3f}'.format(average_score))

irisDF.head(15)

silhouette_samples( ) return 값의 shape (150,)
붓꽃 데이터셋 Silhouette Analysis Score:0.553
```

	sepal_length	sepal_width	petal_length	petal_width	cluster	silhouette_coeff
0	5.1	3.5	1.4	0.2	1	0.852955
1	4.9	3.0	1.4	0.2	1	0.815495
2	4.7	3.2	1.3	0.2	1	0.829315
3	4.6	3.1	1.5	0.2	1	0.805014
4	5.0	3.6	1.4	0.2	1	0.849302
5	5.4	3.9	1.7	0.4	1	0.748280
6	4.6	3.4	1.4	0.3	1	0.821651
7	5.0	3.4	1.5	0.2	1	0.853905
8	4.4	2.9	1.4	0.2	1	0.752150
9	4.9	3.1	1.5	0.1	1	0.825294
10	5.4	3.7	1.5	0.2	1	0.803103
11	4.8	3.4	1.6	0.2	1	0.835913
12	4.8	3.0	1.4	0.1	1	0.810564
13	4.3	3.0	1.1	0.1	1	0.746150
14	5.8	4.0	1.2	0.2	1	0.702594

데이터 별 실루엣 계수

클러스터가 1인 데이터들은 0.8 정도의 실루엣 계수를 가지므로
군집화가 어느정도 잘 된 듯하다.
하지만 실루엣 계수 평균 값이 0.553인 이유는
다른 클러스터에 할당된 데이터들의 실루엣 계수값이 작아서이다.

```
irisDF.groupby('cluster')[ 'silhouette_coeff' ].mean()

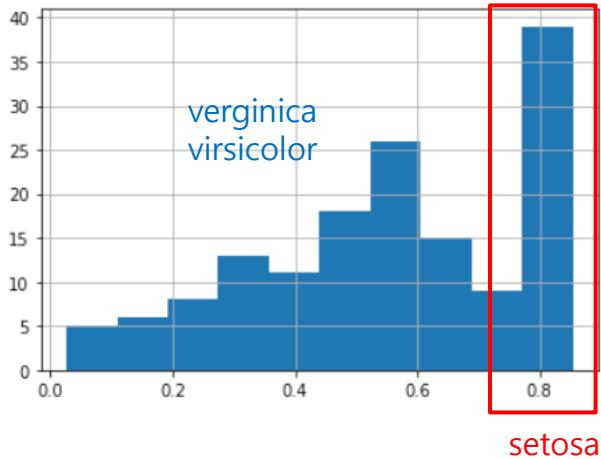
cluster
0    0.451105
1    0.798140
2    0.417320
Name: silhouette_coeff, dtype: float64
```

실습 : 붓꽃 데이터에서 실루엣 계수 계산

setosa는 군집화가 잘 되었지만, virginica와 virsicolor는 잘 되지 않았다.

```
irisDF['silhouette_coeff'].hist()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc5a22fcfd0>
```



실루엣 계수 시각화 통해 최적의 클러스터 수 찾기

데이터들의 실루엣 계수를 계산해서 시각화 해주는 함수

```
### 여러개의 클러스터링 갯수를 List로 입력 받아 각각의 실루엣 계수를 면적으로 시각화한 함수 작성
def visualize_silhouette(cluster_lists, X_features):

    from sklearn.datasets import make_blobs
    from sklearn.cluster import KMeans
    from sklearn.metrics import silhouette_samples, silhouette_score

    import matplotlib.pyplot as plt
    import matplotlib.cm as cm
    import math

    # 입력값으로 클러스터링 갯수들을 리스트로 받아서, 각 갯수별로 클러스터링을 적용하고 실루엣 개수를 구함
    n_cols = len(cluster_lists)

    # plt.subplots()으로 리스트에 기재된 클러스터링 수만큼의 sub figures를 가지는 axs 생성
    fig, axs = plt.subplots(figsize=(4*n_cols, 4), nrows=1, ncols=n_cols)

    # 리스트에 기재된 클러스터링 갯수들을 차례로 iteration 수행하면서 실루엣 개수 시각화
    for ind, n_cluster in enumerate(cluster_lists):

        # KMeans 클러스터링 수행하고, 실루엣 스코어와 개별 데이터의 실루엣 값 계산.
        clusterer = KMeans(n_clusters = n_cluster, max_iter=500, random_state=0)
        cluster_labels = clusterer.fit_predict(X_features)

        sil_avg = silhouette_score(X_features, cluster_labels)
        sil_values = silhouette_samples(X_features, cluster_labels)

        y_lower = 10
        axs[ind].set_title('Number of Cluster : ' + str(n_cluster) + '\n' \
                           'Silhouette Score : ' + str(round(sil_avg, 3)) )
        axs[ind].set_xlabel("The silhouette coefficient values")
        axs[ind].set_ylabel("Cluster label")
        axs[ind].set_xlim([-0.1, 1])
        axs[ind].set_ylim([0, len(X_features) + (n_cluster + 1) * 10])
        axs[ind].set_yticks([])
        # Clear the yaxis labels / ticks
        axs[ind].set_xticks([0, 0.2, 0.4, 0.6, 0.8, 1])

        # 클러스터링 갯수별로 fill_betweenx( ) 형태의 막대 그래프 표현.
        for i in range(n_cluster):
            ith_cluster_sil_values = sil_values[cluster_labels==i]
            ith_cluster_sil_values.sort()

            size_cluster_i = ith_cluster_sil_values.shape[0]
            y_upper = y_lower + size_cluster_i

            color = cm.nipy_spectral(float(i) / n_cluster)
            axs[ind].fill_betweenx(np.arange(y_lower, y_upper), 0, ith_cluster_sil_values, \
                                  facecolor=color, edgecolor=color, alpha=0.7)
            axs[ind].text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))
            y_lower = y_upper + 10

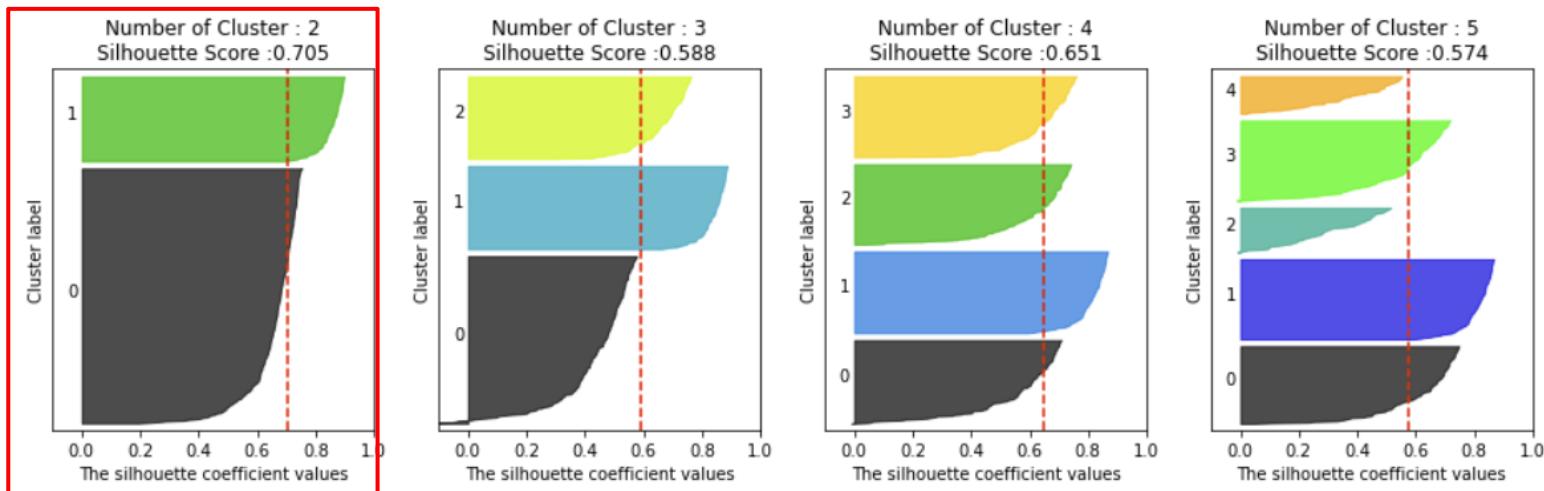
        axs[ind].axvline(x=sil_avg, color="red", linestyle="--")
```

실루엣 계수 시각화 통해 최적의 클러스터 수 찾기

클러스터 수 변화시키면서 random 데이터 실루엣 계수 분포 시각화

```
# make_blobs 를 통해 clustering 을 위한 4개의 클러스터 중심의 500개 2차원 데이터 셋 생성
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=500, n_features=2, centers=4, cluster_std=1, # 
                  center_box=(-10.0, 10.0), shuffle=True, random_state=1)

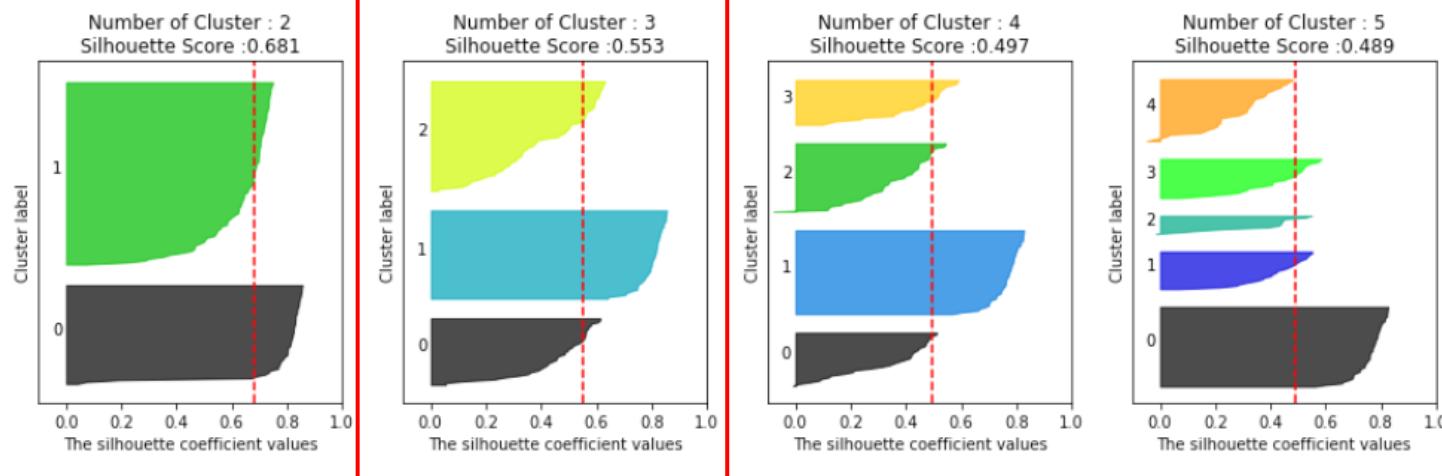
# cluster 개수를 2개, 3개, 4개, 5개 일때의 클러스터별 실루엣 계수 평균값을 시각화
visualize_silhouette([2, 3, 4, 5], X)
```



실루엣 계수 시각화 통해 최적의 클러스터 수 찾기

클러스터 수 변화시키면서 붓꽃 데이터 실루엣 계수 분포 시각화

```
from sklearn.datasets import load_iris  
  
iris=load_iris()  
visualize_silhouette([ 2, 3, 4,5 ], iris.data)
```

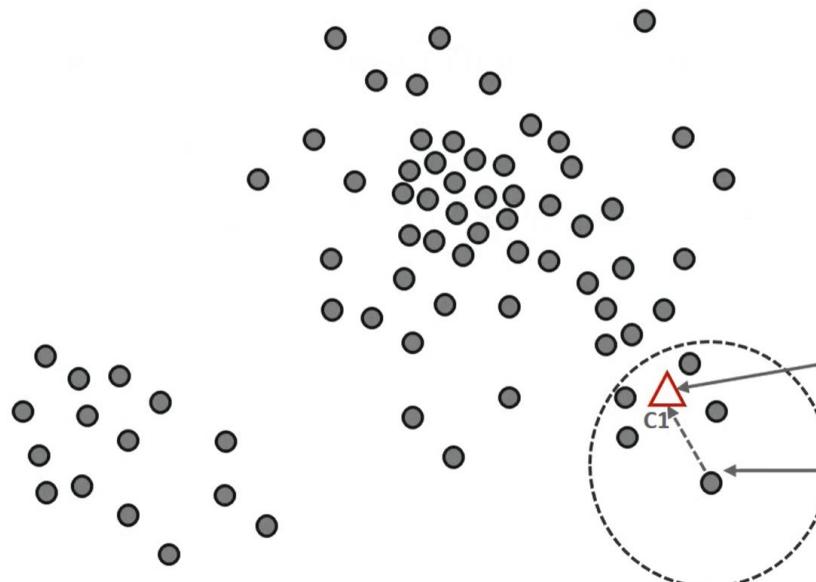


실루엣 계수 평균값(0.553)도 어느정도 높고,
클러스터간 골고루 군집화 되어 있다.

Mean Shift 군집화

- Mean Shift는 KDE(Kernel Density Estimation)를 이용하여 데이터 포인트들이 데이터 분포가 높은 곳으로 이동하면서 군집화를 수행
- 별도의 군집화 개수를 지정하지 않으면 Mean Shift는 데이터 분포도에 기반하여 자동으로 군집화 개수를 정함.

Mean Shift 수행 절차



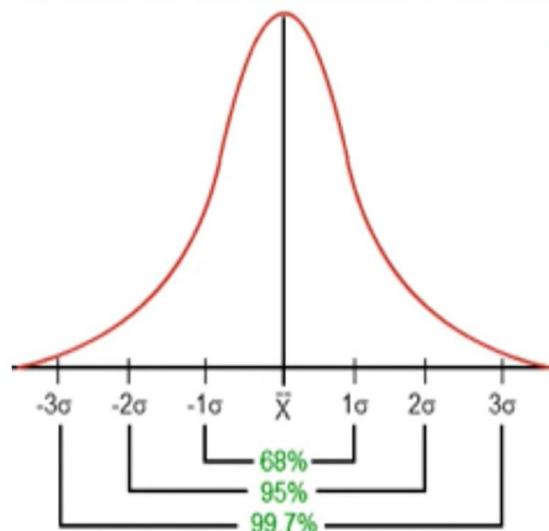
- ① 개별 데이터의 특정 반경 내에 주변 데이터를 포함한 데이터 분포도 계산
- ② 데이터 분포도가 높은 방향으로 중심점 이동
- ③ 중심점을 따라 해당 데이터 이동
- ④ 이동된 데이터의 특정 반경내에 다시 데이터 분포 계산 후 2, 3 스텝을 반복
- ⑤ 가장 분포도가 높은 곳으로 이동하면 더 이상 해당 데이터는 움직이지 않고 수렴
- ⑥ 모든 데이터를 1~5까지 수행하면서 군집 중심점을 찾음

특정 데이터가 반경내의 데이터 분포 확률 밀도가 가장 높은 곳으로 이동 할 때 주변 데이터들과의 거리값을 Kernel 함수 값으로 입력 한 뒤 그 반환값을 현재 위치에서 Update하면서 이동

KDE(Kernel Density Estimation)

KDE는 커널(Kernel)함수를 통해 어떤 변수의 확률밀도 함수를 추정하는 방식. 관측된 데이터 각각에 커널 함수를 적용한 값을 모두 더한 뒤 데이터 건수로 나누어서 확률 밀도 함수를 추정.

정규 분포



- 확률밀도 함수(PDF:Probability Density Function) : 확률 변수의 분포를 나타내는 함수. 대표적으로 정규 분포, 감마 분포, t-분포등이 있음.
- 확률밀도 함수를 알게 되면 특정 변수가 어떤 값을 갖게 될지의 확률을 알게 됨을 의미. 즉 확률밀도 함수를 통해 변수의 특성(예를 들어 정규 분포의 경우 평균, 분산), 확률 분포등 변수의 많은 요소를 알 수 있게 됨.

확률 밀도 추정 방법

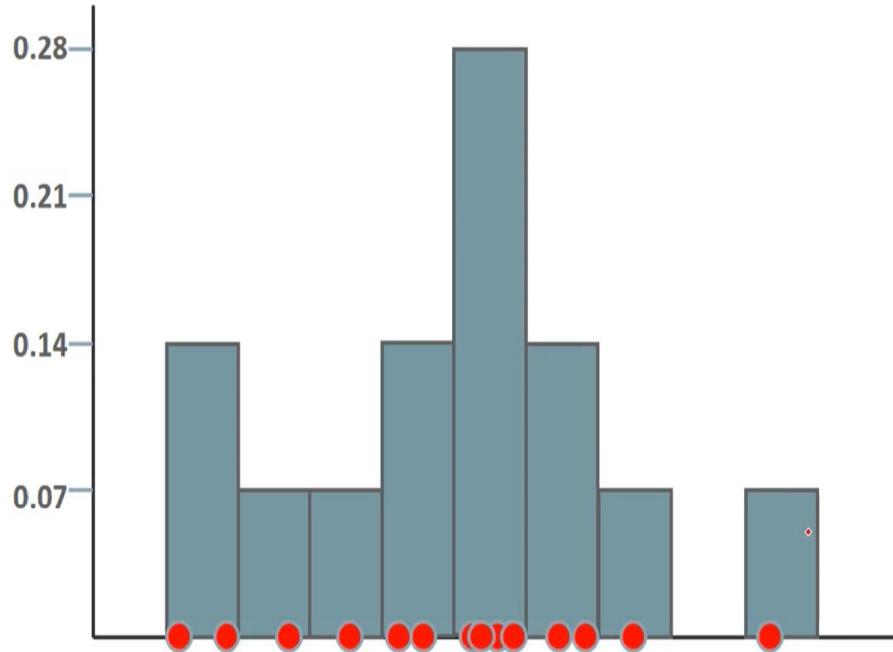
모수적(Parametric) 추정

데이터가 특정 데이터 분포(예를 들어 가우시안 분포)를 따른다는 가정 하에 데이터 분포를 찾는 방법. Gaussian Mixture 등이 있음.

비모수적(Non-Parametric) 추정

데이터가 특정 분포를 따르지 않는다는 가정 하에서 밀도를 추정. 관측된 데이터만으로 확률 밀도를 찾는 방법으로서 대표적으로 KDE가 있음.

비모수적 밀도 추정 – 히스토그램 (Histogram)



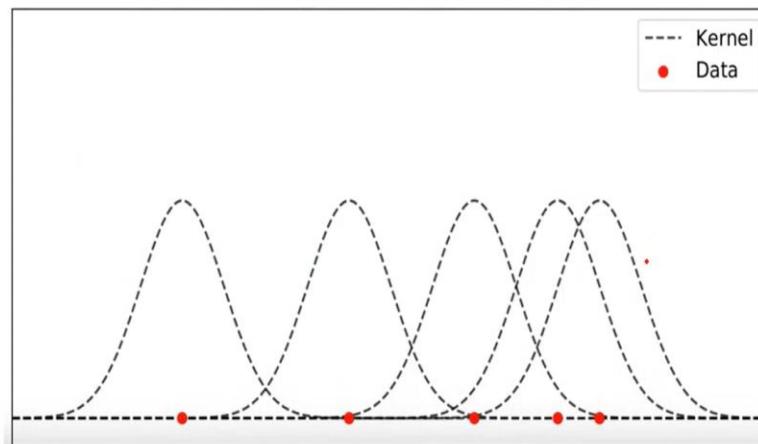
히스토그램 밀도 추정의 문제점.

- Bin의 경계에서 불연속성이 나타남
- Bin의 크기에 따라 히스토그램이 달라짐.

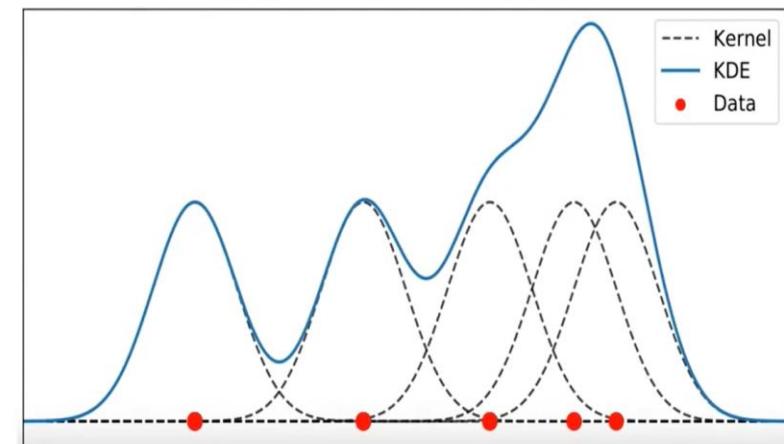
비 모수적 밀도 추정 -KDE

KDE는 개별 관측 데이터들에 커널함수를 적용한 뒤, 커널함수들의 적용값을 모두 합한 뒤에 개별 관측 데이터의 건수로 나누어서 확률밀도 함수를 추정하는 방식임. 커널함수로는 대표적으로 가우시안 분포함수가 사용됨.

개별 관측 데이터에 가우시안 커널 함수 적용



가우시안 커널 함수 적용 후 합산



KDE와 가우시안 커널 함수

KDE는 아래와 같은 커널함수 식으로 표현됨. 이때 K 는 커널함수, x 는 random variable, x_i 는 관측값, h 는 bandwidth

$$\text{KDE} = \frac{1}{n} \sum_{i=1}^n K_h(x-x_i) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x-x_i}{h}\right)$$

대표적인 커널함수는 가우시안 분포임. $f(x | \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

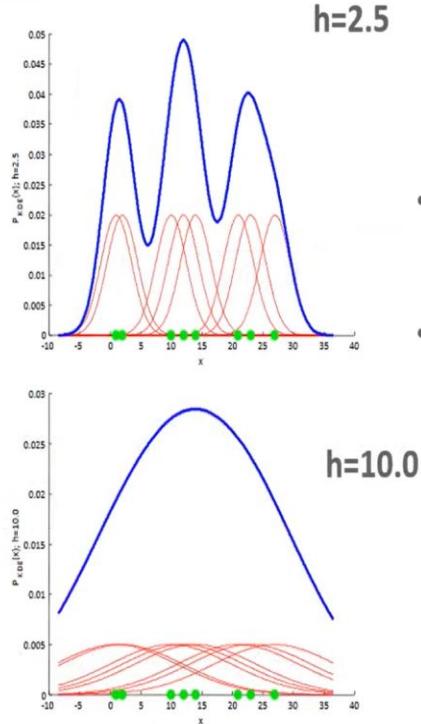
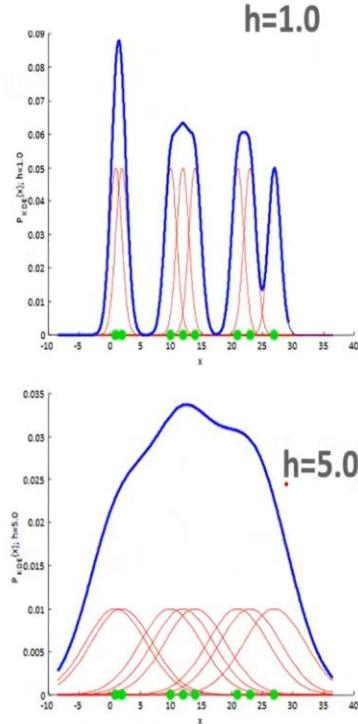
가우시안 커널함수를 적용한 KDE는 아래와 같음. 이 경우 관측값 x_i 는 평균, bandwidth h 는 표준편차와 동일.

$$\text{KDE} = \frac{1}{nh} \sum_{i=1}^n \frac{1}{\sqrt{2\pi} \sigma} e^{\left(-\frac{1}{2} \left(\frac{x-x_i}{\sigma}\right)^2\right)}$$

가우시안 커널함수를 적용할 경우 최적의 bandwidth는 아래와 같습니다.

$$h = \left(\frac{4\sigma^5}{3n}\right)^{\frac{1}{5}} \approx 1.06\sigma n^{-1/5} \quad \text{단, } n \text{은 샘플 데이터의 개수, } \sigma \text{는 샘플 데이터의 표준편차}$$

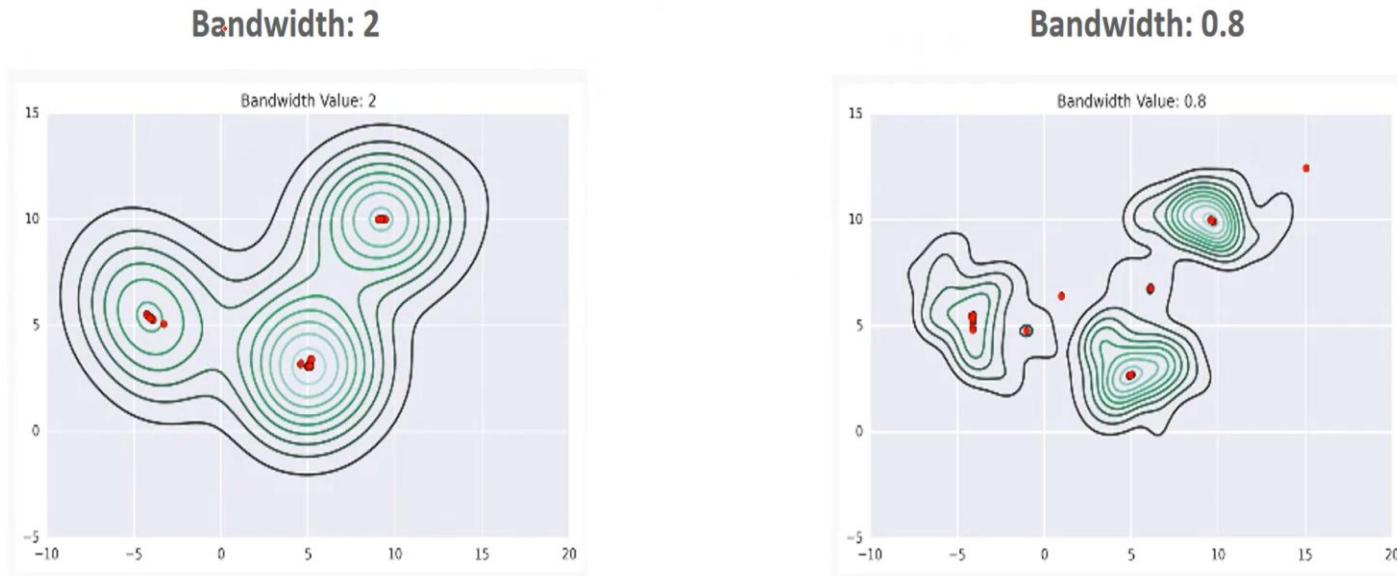
Bandwidth에 따른 KDE의 변화



- 작은 h 값은 좁고 Spike한 KDE로 변동성이 큰 확률밀도함수를 추정(오버피팅)
- 큰 h 값은 과도하게 Smoothing된 KDE로 단순화된 확률밀도함수를 추정(언더피팅)

Bandwidth에 따른 KDE의 변화

Mean Shift는 Bandwidth가 클수록 적은 수의 클러스터링 중심점을, Bandwidth가 작을수록 많은 수의 클러스터링 중심점을 가지게 됨. 또한 Mean Shift는 군집의 개수를 지정하지 않으며, 오직 Bandwidth의 크기에 따라 군집화를 수행.



사이킷런 Mean Shift

- 사이킷런은 Mean Shift 군집화를 위해 MeanShift 클래스를 제공
- MeanShift 클래스의 가장 중요한 초기화 파라미터는 bandwidth이며 해당 파라미터는 밀도 중심으로 이동 할때 사용되는 커널 함수의 bandwidth임. 이 bandwidth를 어떻게 설정하느냐에 따라 군집화 성능이 달라짐.
- 최적의 bandwidth 계산을 위해 사이킷런은 estimate_bandwidth() 함수를 제공

사이킷런을 이용한 Mean Shift

make_blobs()를 이용하여 2개의 feature와 3개의 군집 중심점을 가지는 임의의 데이터 200개를 생성하고 MeanShift를 이용하여 군집화 수행

```
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.cluster import MeanShift

X, y = make_blobs(n_samples=200, n_features=2, centers=3,
                   cluster_std=0.7, random_state=0)
```

```
meanshift = MeanShift(bandwidth=0.8)
cluster_labels = meanshift.fit_predict(X)
print('cluster labels 유형:', np.unique(cluster_labels))
```

```
cluster labels 유형: [0 1 2 3 4 5]
```

커널 함수의 **bandwidth** 크기를 1로 약간 증가 후에 Mean Shift 군집화 재수행

```
meanshift = MeanShift(bandwidth=1)
cluster_labels = meanshift.fit_predict(X)
print('cluster labels 유형:', np.unique(cluster_labels))
```

cluster labels 유형: [0 1 2]

최적의 **bandwidth**값을 **estimate_bandwidth()**로 계산 한 뒤에 다시 군집화 수행

```
from sklearn.cluster import estimate_bandwidth

bandwidth = estimate_bandwidth(X)
print('bandwidth 값:', round(bandwidth, 3))
```

bandwidth 값: 1.816

```
import pandas as pd

clusterDF = pd.DataFrame(data=X, columns=['ftr1', 'ftr2'])
clusterDF['target'] = y

# estimate_bandwidth()로 최적의 bandwidth 계산
best_bandwidth = estimate_bandwidth(X)

meanshift = MeanShift(bandwidth=best_bandwidth)
cluster_labels = meanshift.fit_predict(X)
print('cluster labels 유형:', np.unique(cluster_labels))
```

cluster labels 유형: [0 1 2]

시각화

```
import matplotlib.pyplot as plt
%matplotlib inline

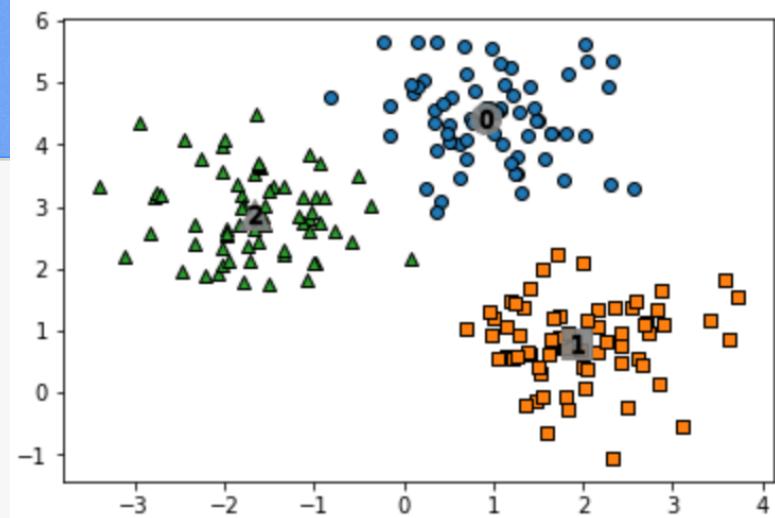
clusterDF['meanshift_label'] = cluster_labels
centers = meanshift.cluster_centers_
unique_labels = np.unique(cluster_labels)
markers=['o', 's', '^', 'x', '*']

for label in unique_labels:
    label_cluster = clusterDF[clusterDF['meanshift_label']==label]
    center_x_y = centers[label]
    # 군집별로 다른 마커로 산점도 적용
    plt.scatter(x=label_cluster['ftr1'], y=label_cluster['ftr2'], edgecolor='k', marker=markers[label])

    # 군집별 중심 표현
    plt.scatter(x=center_x_y[0], y=center_x_y[1], s=200, color='gray', alpha=0.9, marker=markers[label])
    plt.scatter(x=center_x_y[0], y=center_x_y[1], s=70, color='k', edgecolor='k', marker='%' % label)

plt.show()
```

Confirm



```
print(clusterDF.groupby('target')[['meanshift_label']].value_counts())
```

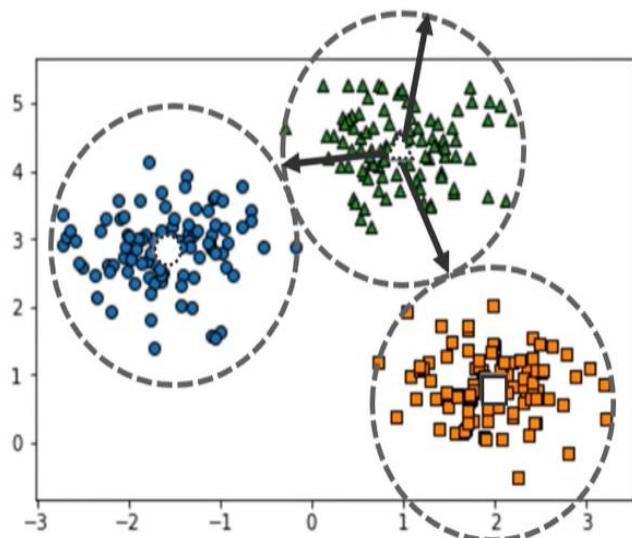
target	meanshift_label	value_counts()
0	0	67
1	1	67
2	2	66

Name: meanshift_label, dtype: int64

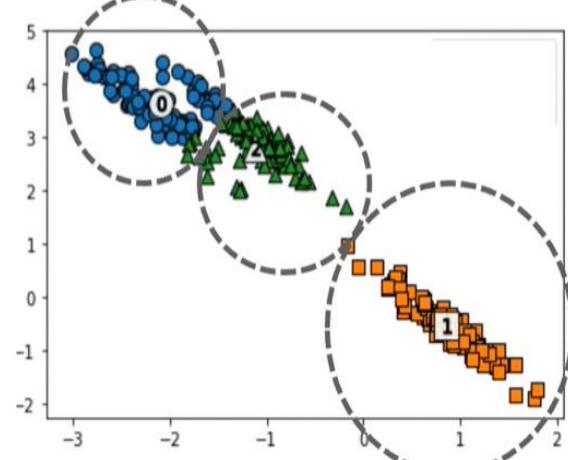
GMM-거리기반 K-Means의 문제점

K-Means는 특정 중심점을 기반으로 거리적으로 퍼져있는 데이터 세트에 군집화를 적용하면 효율적.
하지만 K-Means이러한 데이터 분포를 가지지 않는 데이터 세트에 대해서는 효율적인 군집화가 어려움

Kmeans로 효율적인 군집화 가능



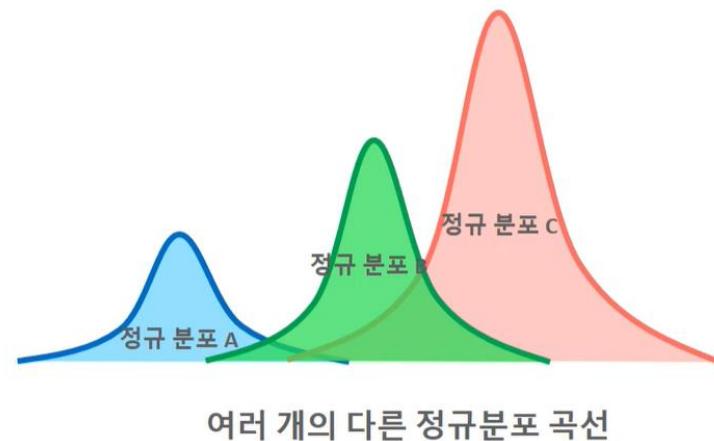
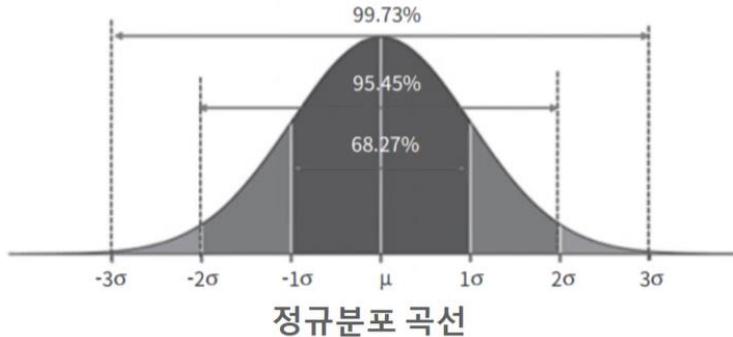
Kmeans로 군집화가 어려운 데이터



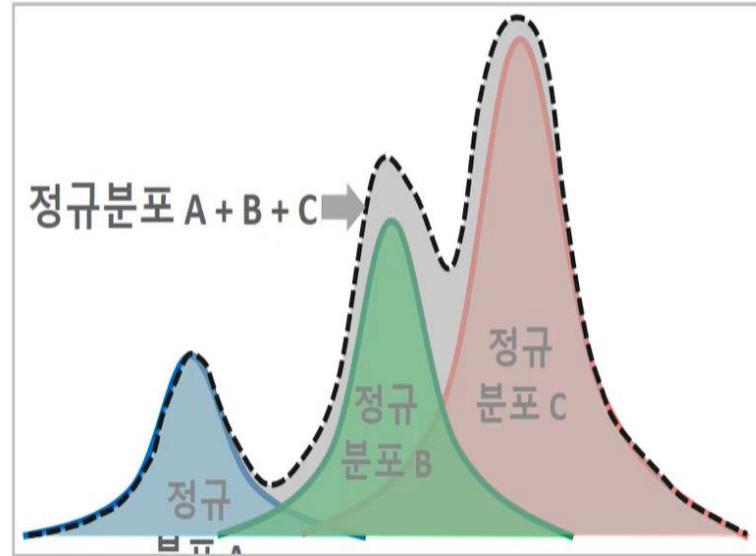
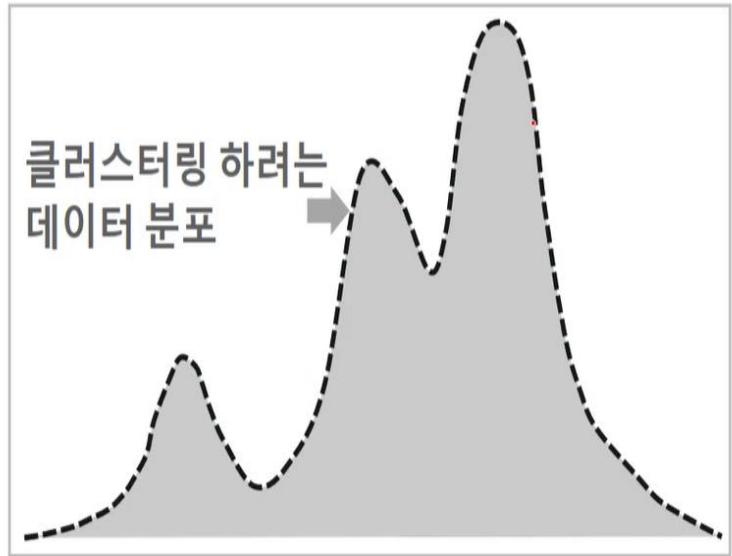
GMM(Gaussian Mixture Model)개요

GMM 군집화는 군집화를 적용하고자 하는 데이터가 여러 개의 다른 가우시안 분포(Gaussian Distribution)를 가지는 모델로 가정하고 군집화를 수행합니다.

가령 1000개의 데이터 세트가 있다면 이를 구성하는 여러 개의 정규 분포 곡선을 추출하고, 개별 데이터가 이 중 어떤 정규 분포에 속하는지 결정하는 방식입니다.

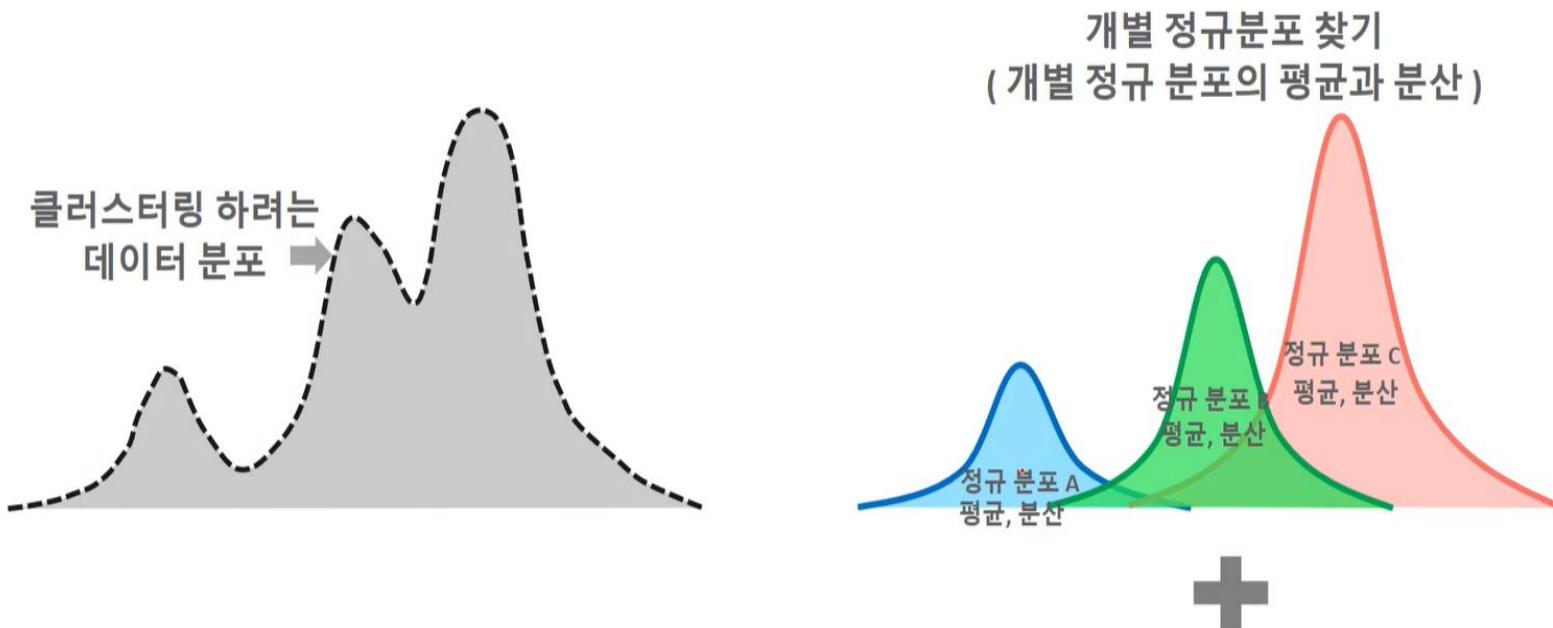


서로 다른 정규 분포로 결합된 원본 데이터 분포



GMM 모구 (Parameter) 추정

GMM 모수 추정은 개별 정규 분포들의 평균과 분산, 그리고 데이터가 특정 정규 분포에 해당 될 확률을 추정하는 것입니다



데이터가 특정 정규 분포에 해당 될 확률 구하기

데이터 $x = \text{정규분포 A}(30\%) + \text{정규분포 B}(30\%) + \text{정규분포 C}(30\%)$

GMM모수 추정을 위한 EM(Expectation and Maximazation)

Expectation

개별 데이터 각각에 대해서 특정 정규 분포에 소속될 확률을 구하고 가장 높은 확률을 가진 정규 분포에 소속
(최초시에는 데이터들을 임의로 특정 정규 분포로 소속)

Maximization

데이터들이 특정 정규분포로 소속되면 다시 해당 정규분포의 평균과 분산을 구함.
해당 데이터가 발견될 수 있는 가능도를 최대화(Maximum likelihood) 할 수 있도록 평균과 분산(모수)를 구함

개별 정규분포의 모수인 평균과 분산이 더 이상 변경되지 않고 각 개별 데이터들이 이전 정규 분포 소속이 더 이상 변경되지 않으면
그것으로 최종 군집화를 결정하고 그렇지 않으면 계속 EM 반복을 수행.

사이킷런 Gaussian Mixture

- 사이킷런은 GMM 군집화를 위해 GaussianMixture 클래스를 제공.
- GaussianMixture 클래스의 주요 생성자 파라미터는 `n_components`이며 이는 Mixture Model의 개수, 즉 군집화 개수를 의미

GMM을 이용한 붓꽃 데이터 셋 클러스터링

```
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
%matplotlib inline

iris = load_iris()
feature_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']

# 보다 편리한 데이터 Handling을 위해 DataFrame으로 변환
irisDF = pd.DataFrame(data=iris.data, columns=feature_names)
irisDF['target'] = iris.target
```

GaussianMixture를 이용한 불꽃 데이터 군집화

```
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=3, random_state=0).fit(iris.data)
gmm_cluster_labels = gmm.predict(iris.data)

# 클러스터링 결과를 irisDF 의 'gmm_cluster' 컬럼명으로 저장
irisDF['gmm_cluster'] = gmm_cluster_labels
irisDF['target'] = iris.target

# target 값에 따라서 gmm_cluster 값이 어떻게 매핑되었는지 확인.
iris_result = irisDF.groupby(['target'])['gmm_cluster'].value_counts()
print(iris_result)
```

```
target  gmm_cluster
0        0            50
1        1            45
        2             5
2        2            50
Name: gmm_cluster, dtype: int64
```

붓꽃 데이터 K-means 군집화 결과

```
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=300, random_state=0).fit(iris.data)
kmeans_cluster_labels = kmeans.predict(iris.data)
irisDF['kmeans_cluster'] = kmeans_cluster_labels
iris_result = irisDF.groupby(['target'])['kmeans_cluster'].value_counts()
print(iris_result)
```

```
target  kmeans_cluster
0        1                  50
1        2                  48
2        0                  2
2        0                  36
2        2                  14
Name: kmeans_cluster, dtype: int64
```

Comparison !

Which one better ? K-Means VS GMM

Toy Examples –for comprehensive understanding

```
## 클러스터 결과를 담은 DataFrame과 사이킷런의 Cluster 객체등을 인자로 받아 클러스터링 결과를 시각화하는 함수
def visualize_cluster_plot(clusterobj, datafram, label_name, iscenter=True):
    if iscenter:
        centers = clusterobj.cluster_centers_
    unique_labels = np.unique(datafram[label_name].values)
    markers=['o', 's', '^', 'x', '*']
    isNoise=False

    for label in unique_labels:
        label_cluster = datafram[datafram[label_name]==label]
        if label == -1:
            cluster_legend = 'Noise'
            isNoise=True
        else:
            cluster_legend = 'Cluster ' +str(label)

        plt.scatter(x=label_cluster['ftr1'], y=label_cluster['ftr2'], s=70, w
                    edgecolor='k', marker=markers[label], label=cluster_legend)

    if iscenter:
        center_x_y = centers[label]
        plt.scatter(x=center_x_y[0], y=center_x_y[1], s=250, color='white',
                    alpha=0.9, edgecolor='k', marker=markers[label])
        plt.scatter(x=center_x_y[0], y=center_x_y[1], s=70, color='k', w
                    edgecolor='k', marker='$%d$' % label)

    if isNoise:
        legend_loc='upper center'
    else: legend_loc='upper right'

    plt.legend(loc=legend_loc)
    plt.show()
```

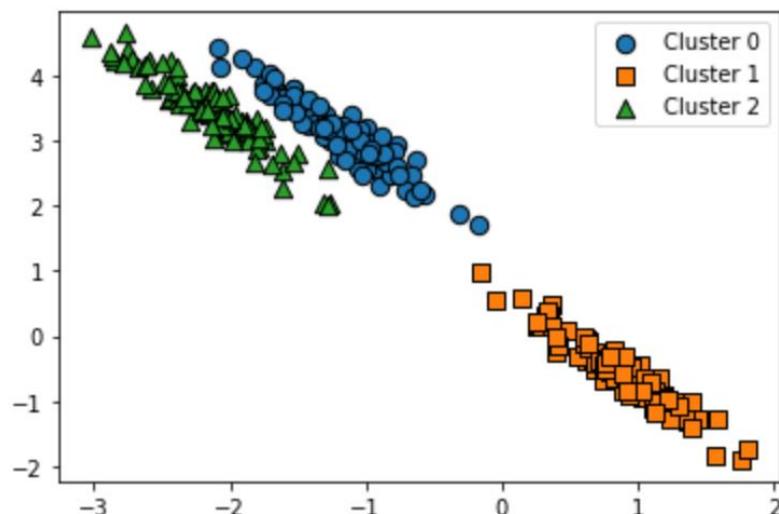
```

from sklearn.datasets import make_blobs

# make_blobs()로 300개의 데이터 셋, 3개의 cluster 셋, cluster_std=0.5을 만듬.
X, y = make_blobs(n_samples=300, n_features=2, centers=3, cluster_std=0.5, random_state=0)

# 길게 늘어난 타원형의 데이터 셋을 생성하기 위해 변환함.
transformation = [[0.60834549, -0.63667341], [-0.40887718, 0.85253229]]
X_aniso = np.dot(X, transformation)
# feature 데이터 셋과 make_blobs()의 y 결과값을 DataFrame으로 저장
clusterDF = pd.DataFrame(data=X_aniso, columns=['ftr1', 'ftr2'])
clusterDF['target'] = y
# 생성된 데이터 셋을 target 별로 다른 marker로 표시하여 시각화 함.
visualize_cluster_plot(None, clusterDF, 'target', iscenter=False)

```

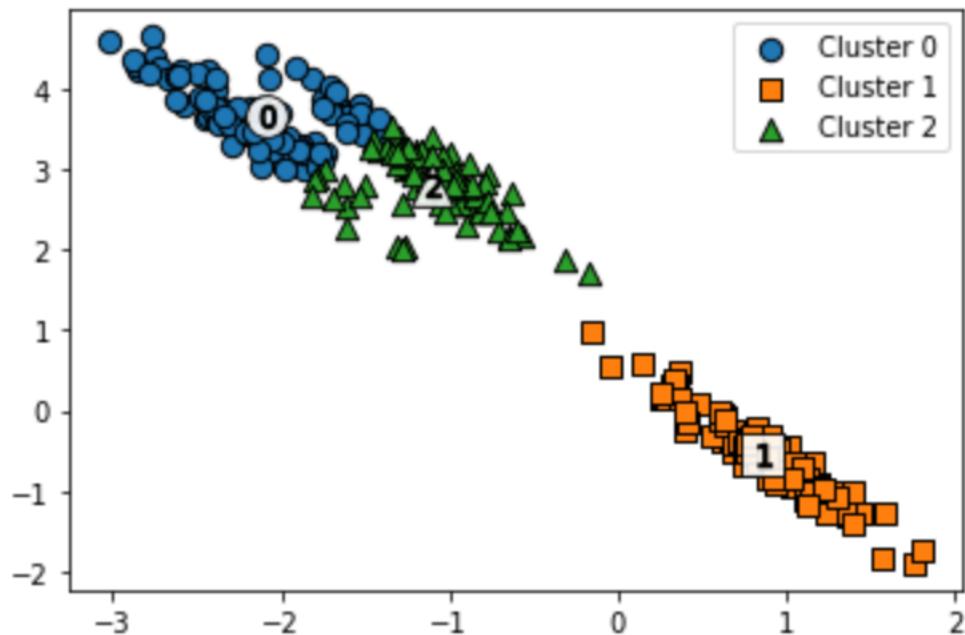


K-means

```
# 3개의 Cluster 기반 Kmeans 를 X_aniso 데이터 셋에 적용
```

```
kmeans = KMeans(3, random_state=0)  
kmeans_label = kmeans.fit_predict(X_aniso)  
clusterDF['kmeans_label'] = kmeans_label
```

```
visualize_cluster_plot(kmeans, clusterDF, 'kmeans_label', iscenter=True)
```



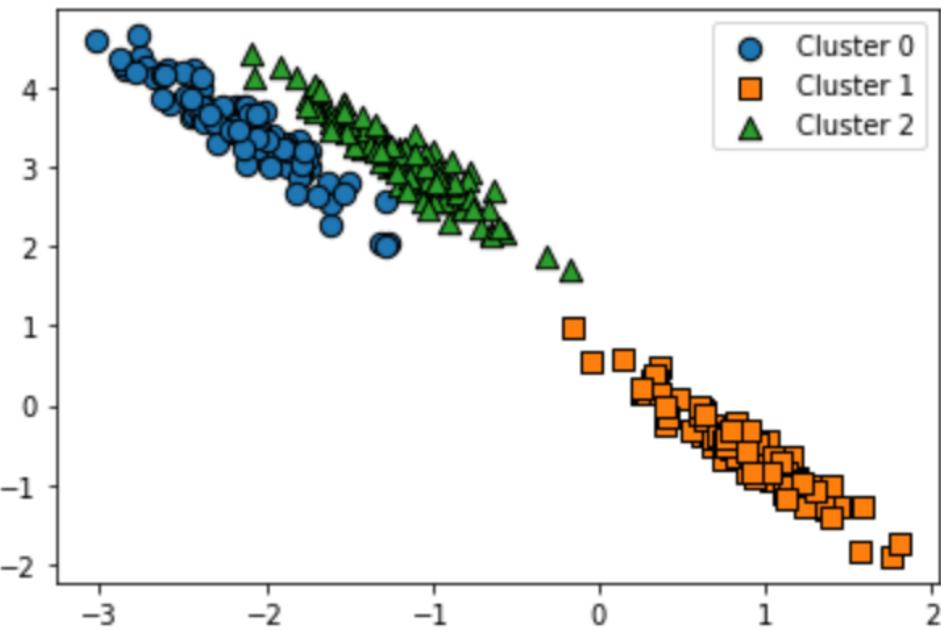
GMM Model

3개의 *n_components*기반 GMM을 *X_aniso* 데이터 셋에 적용

```
gmm = GaussianMixture(n_components=3, random_state=0)
gmm_label = gmm.fit(X_aniso).predict(X_aniso)
clusterDF['gmm_label'] = gmm_label
```

GaussianMixture는 *cluster_centers_* 속성이 없으므로 *iscenter*를 *False*로 설정.

```
visualize_cluster_plot(gmm, clusterDF, 'gmm_label', iscenter=False)
```



Results

```
print('### KMeans Clustering ###')
print(clusterDF.groupby('target')['kmeans_label'].value_counts())
print('### Gaussian Mixture Clustering ###')
print(clusterDF.groupby('target')['gmm_label'].value_counts())
```

```
### KMeans Clustering ###
```

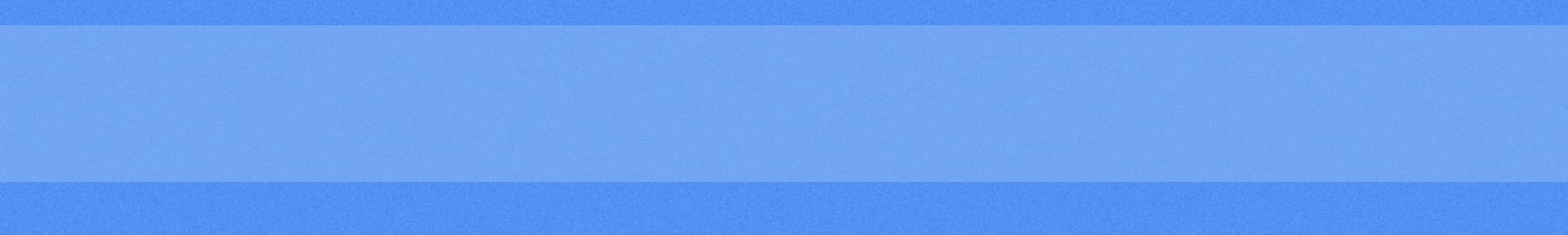
```
target  kmeans_label
0        2            73
        0            27
1        1            100
2        0            86
        2            14
```

```
Name: kmeans_label, dtype: int64
```

```
### Gaussian Mixture Clustering ###
```

```
target  gmm_label
0        2            100
1        1            100
2        0            100
```

```
Name: gmm_label, dtype: int64
```



Thank You