**[ 지도 학습 및 비지도 학습 ]**

머신러닝은 크게 지도 학습과 비지도 학습으로 구분

**지도 학습 ( supervised learning )**

* 레이블(label)이 달린 데이터로 학습 모델을 만든다
  + 나이와 학력, 거주지와 같은 다양한 매개변수를 기반으로 개인의 소득을 예측하는 시스템을 구축하려면, 먼저 여러 사람에 대한 구체적인 정보를 모은 후 항목마다 레이블(각 사람의 소득 정보)을 달아서 데이터베이스를 구축
* 여러가지 매개변수와 소득의 관계가 정의된 데이터베이스를 학습 알고리즘에게 전달하면, 특정한 사람에 대한 매개변수가 주어졌을 때 그 사람의 소득을 계산하는 방법을 학습

**비지도 학습 ( unsupervised learning )**

* 레이블이 달리지 않은 데이터로 학습 모델을 만든다.
* 레이블이 없기 때문에 데이터의 내용만 보고 의미 있는 정보를 추출해야 한다.
  + 특정 데이터 집합에 속한 개별 데이터들을 여러 개의 그룹으로 나눌 때 비지도 학습 방식을 적용할 수 있다.
* 그룹을 나누는 기준이 명확하지 않기 때문에 가능한 최선의 방법을 동원해 데이터를 나눠야 한다.

**[ 분류 ] – 지도학습**

분류 ( classification ) : 데이터를 지정한 수만큼의 클래스(범주)로 나누는 기법  
데이터를 가장 효과적이면서 효율적으로 활용하도록 일정한 수의 그룹으로 데이터를 분류한다.

머신 러닝에서는 주어진 data 항목이 속하는 클래스를 결정하는 문제를 다룰 때 분류 기법 사용

* Ex ) 주어진 이미지에 사람 얼굴이 있는지 검사하고 싶다면, 먼저 얼굴 이미지와 얼굴이 아닌 이미지라는 두가지 클래스로 구분된 학습 데이터를 구축한 후 , 이 데이터로 모델을 학습시킨다. 이 후 , 학습이 완료된 모델로 추론(이미지 검사) 한다.

분류를 위해 샘플(학습용 데이터)의 수는 다양한 상황을 반영할 수 있도록 충분해야 함.  
샘플이 부족하면 알고리즘이 학습 데이터에 필요 이상으로 최적화되는 오버피팅(overfitting ) 현상이 발생할 수 있다.

**[ 데이터 전처리 ]**

머신 러닝을 구현 할 때 현실 세계에서 추출한 방대한 양의 미가공 데이터를 다룬다.  
따라서, 학습 시키기 전에 먼저 데이터의 포맷을 머신 러닝 알고리즘의 처리할 수 있는 형태로 변환하는 전처리(preprocessing) 작업부터 해야한다.

|  |
| --- |
| Import numpy as np  From sklearn import preprocessing  Input\_data = np.array([[5.1,-2.9,3.3],  [-1.2,7.8,-6.1],  [3.9,0.4,2.1],  [7.3,-9.9,-4.5]]) |

전처리 기법 4가지

1. 이진화
2. 평균 제거
3. 크기 조정
4. 정규화

**[ 이진화 ]**

이진화 ( binariaztion ) 란 숫자를 불리언 ( Boolean ) 수로 변환하는 기법 ( 이진수로 변환 )

사이킷런에서 제공하는 메소드로 입력 데이터를 이진화하는 방법  
기준점 2.1

|  |
| --- |
| Data\_binarized = preprocessing.Binarizer(threshold=2.1).transform(input\_data)  Print(“\nBinarized data:\n”, data\_binarized) |

결과를 보면 2.1 보다 큰 값을 1로, 나머지는 0으로 변환됨을 볼 수 있음

**[ 평균 제거 ]**

머신 러닝에서 흔히 사용하는 전처리 기법, 특히 **특징 벡터( feature vector ) 의 값들이 0을 중심으로 분포하게 만들 때** 평균 제거 기법을 많이 사용  
특징 벡터 : 물체의 특징점들을 나열한 것.

*앞에서 작성코드에 이어서 작성*

|  |
| --- |
| #평균과 표준편차를 출력  Print(“\nbefore: “)  Print(“Mean =”, input\_data.mean(axis=0))  Print(“std deviation =”, input\_data.std(axis=0))  #평균 제거  Data\_scaled = preprocessing.scale(input\_data)  Print(“\nAfter: “)  Print(“Mean =”, input\_data.mean(axis=0))  Print(“std deviation =”, input\_data.std(axis=0)) |

결과를 보면 평균 값은 0에 표준편차는 1에 가깝게 조정된 것을 알 수 있다

**[ 크기 조정 ]**

특징 벡터의 각 요소에 대한 값의 범위는 일정하지 않을 수 있다.  
머신 러닝 알고리즘을 학습 시키려면, 모든 특징을 동일 선상에서 비교할 수 있도록 각각의 특징에 대한 값의 범위를 일정한 기준으로 조정하는 크기 조정(scaling) 과정을 거쳐야 한다.

|  |
| --- |
| #최솟값/최댓값 조정  Data\_scaler\_minmax = preprocessing.MinMaxScaler(feature\_range=(0,1))  Data\_scaled\_minmax = data\_scaler\_minmax.fit\_transform(input\_data)  Print(“\nMin Max Scaled data: \n”, data\_scaled\_minmax) |

각 행마다 최댓값은 1로, 나머지는 1을 기준으로 상대적인 값으로 표현됨을 확인

**[ 정규화 ]**

특징 벡터의 값을 일정한 기준으로 측정하려면 정규화( normalization ) 과정을 거친다.  
정규화 기법

1. L1 정규화 ( 최소 절대 편차 Least Absolute Deviation ) : 각 행의 절대 값의 합이 1이 되도록 조정
2. L2 정규화 ( 최소 제곱 Least Square ) 제곱의 합이 1이 되도록 조정

L2보다 L1 기법이 더 안정적!! L1이 데이터의 이상치(Outlier) 에 영향을 덜 받기 때문에.  
반면 이상치가 문제 해결에 중요한 역할을 한다면 L2 정규화를 사용해야함

*Outlier : 잘못 평가된 값, 결과적으로 잘못된 분석 결과를 초래 할 수 있는 값들. 간단하게 말해서 샘플의 전체적인 패턴에서 벗어나게 관측되는 값*

|  |
| --- |
| #데이터 정규화  Data\_normalized\_l1 = preprocessing.normalize(input\_data, norm=’l1’)  Data\_normalized\_l2 = preprocessing.normalize(input\_data, norm=’l2’)  Print(“\n L1 normalized data: \n” , data\_normalized\_l1)  Print(“\n L2 normalized data: \n” , data\_normalized\_l2) |

**[ 레이블 인코딩 ]**

분류 작업 시, 수많은 레이블을 처리한다. 레이블은 문자나 숫자 뿐만 아니라 다양한 형태로 표현  
사이킷런에서 제공하는 머신 러닝 함수는 숫자로 된 레이블만 처리.

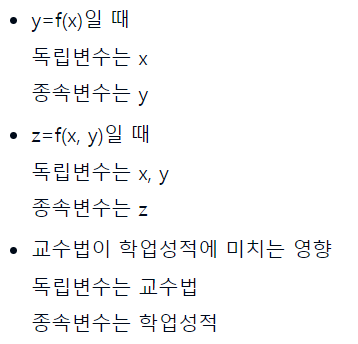
레이블은 대체로 사람이 읽기 좋도록 문자로 표현하는 경우가 많다. 문자로 된 레이블을 숫자로 변환하려면 레이블 인코더를 거쳐야 한다.

레이블 인코딩 ( label encoding ) 이란 문자로 표현된 레이블을 숫자 형태로 바꾸는 과정

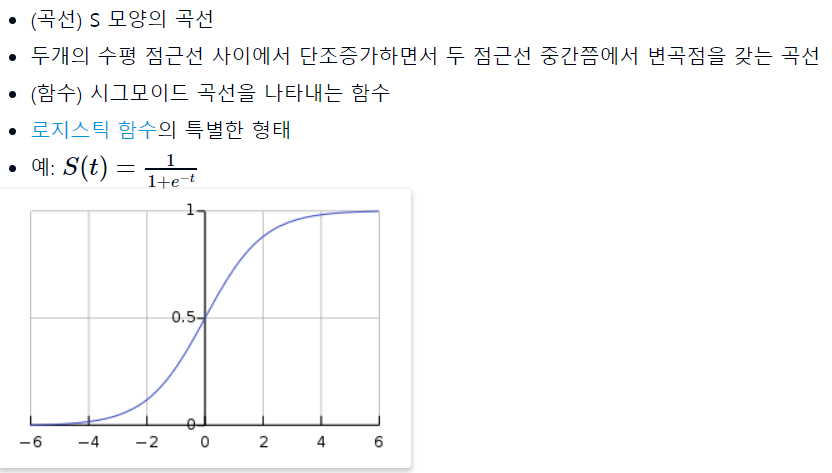
|  |
| --- |
| Import numpy as np  From sklearn import preprocessing  #샘플 입력 레이블  Input\_labels = [ ‘red’ , ‘black’ , ‘red’, ‘green’, ‘black’, ‘yellow’, ‘white’ ]  #레이블 인코더 생성 후 앞에서 정의한 레이블로 학습시키기  Encoder = preprocessing.LabelEncoder()  Encoder.fit(input\_labels)  #매핑 관계 출력  Print(\n Label mapping:\n”)  For I, item in enumerate(encoder.classes\_) :  Print(item, ‘🡪’ , i)  #레이블 인코딩  Test\_labels = [‘green’ , ‘red’, ‘black’ ]  Encoded\_values = encoder.transform(test\_labels)  Print(“\n Lables = “ , test\_labels)  Print(“\n Encoded values= “ , list(encoded\_values))  #숫자 값 디코딩  encoded\_values = [3,0,4,1]  decoded\_list = encoder.inverse\_transform(encoded\_values)  Print(“\n Encoded values = “ , encoded\_values)  Print(“\n decoded values= “ , list(decoded\_list)) |

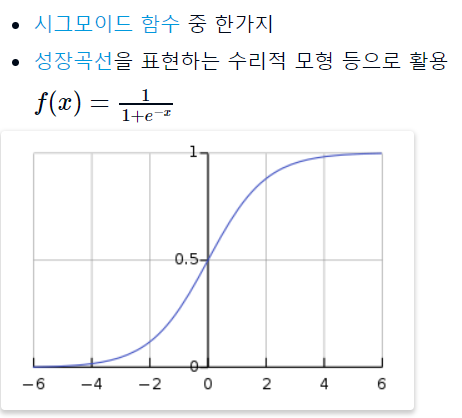
**[ 로지스틱 회귀 분류기 ]**

로지스틱 회귀 분석 ( logistic regression ) 이란 입력 변수와 출력 변수의 관계를 표현하는 기법 중 하나다. 여기서 입력은 독립 변수고 출력은 종속 변수다. 종속 변수의 수는 고정된다. 분류 문제를 다룰 때는 종속 변수로 분류할 클래스(그룹)를 표현한다.



**로지스틱 회귀 분석은** **독립 변수와 종속 변수의 관계를 로지스틱 함수를 통해 계산된 확률로 표현한다**. 이때 로지스틱 함수는 시그모이드 곡선( sigmoid curve) 으로 표현한다.

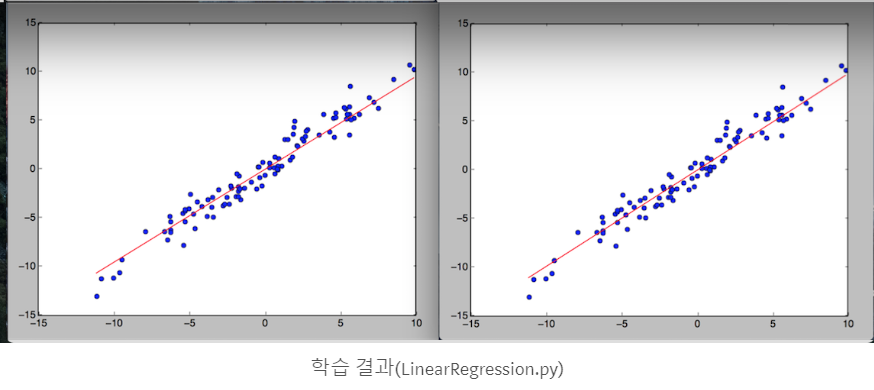
**시그모이드 곡선**

**로지스틱 함수**

여러가지 **매개변수로 구성된 함수를 만들 때 주로 사용**

로지스틱 함수는 데이터의 분포를 표현하는 직선 중에 오차가 가장 적은 직선을 구하는 일반 선형 모델 ( generalized linear model 선형 회귀 분석 ) 과 밀접한 관계가 있다.





엄밀히 말하면, 로지스틱 회귀 분석이 분류 기법은 아니지만, 분류 문제를 다루는 데 효과적이고 간결하기 때문에 머신러닝에서 굉장히 많이 사용한다.

로지스틱 회귀 분석을 이용한 분류기 만들기.

|  |
| --- |
| #패키지를 불러오기  import numpy as np *//numpy 란 과학 계산을 위한 라이브러리(다차원 배열 처리)*  from sklearn import linear\_model *//sklearn은 다양한 머신러닝 모형, 알고리즘을 하나의 패키지에서 모두 제공*  import matplotlib.pyplot as plt *//자료를 차트나 플롯으로 시각화하는 패키지*  *// import 모듈이름 ( 모듈안의 함수 사용할 때, plt.~~ 이렇게 사용해야 함)*  from utilities import visualize\_classifier  *// from 모듈이름 import 모듈함수 ( 모듈함수 바로 쓰고 싶을 때 사용 )*  #샘플 입력 데이터 정의 및 레이블도 정의  X = np.array([[3.1, 7.2], [4, 6.7], [2.9, 8], [5.1, 4.5], [6, 5], [5.6, 5],  [3.3, 0.4], [3.9, 0.9], [2.8, 1], [0.5, 3.4], [1, 4], [0.6, 4.9]])  y = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3])  #로지스틱 회귀 분류기 오브젝트 생성  classifier = linear\_model.LogisticRegression(solver='liblinear',C=1) *// sklearn 안의 linear\_model 함수 사용 . 이곳의 로지스틱 사용하여 오브젝트 생성.*  #앞서 정의한 데이터로 분류기 학습  classifier.fit(X,y) *//학습 시킬 때 fit 사용*  #분류기 성능 시각화  visualize\_classifier(classifier,X,y) |

visualize\_classifier 를 사용하기 위해 별도의 utilities.py로 정의

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  #visiualize\_classifier 함수는 분류기 오브젝트와 입력 데이터, 입력 데이터 레이블을 인수로 받도록 정의  def visualize\_classifier(classifier, X, y):  #메시 그리드에서 사용할 X와 y에 대한 최댓값과 최솟값을 정의한다.  min\_x, max\_x = X[:, 0].min() - 1.0, X[:, 0].max() + 1.0  min\_y, max\_y = X[:, 1].min() - 1.0, X[:, 1].max() + 1.0  # 메시 그리드 mesh grid : 함수를 평가하는 데 사용할 값들의 집합, 각각의 클래스의 경계 값을 시각적으로 표현하는 데 활용. 앞에서 정의한 최댓값과 최솟값을 이용해 그리드의 간격을 정의  #메시 그리드를 그릴 때 적용할 간격 정의  mesh\_step\_size = 0.01  #메시 그리드의 x값과 y값 정의  x\_vals, y\_vals = np.meshgrid(np.arange(min\_x, max\_x, mesh\_step\_size), np.arange(min\_y, max\_y, mesh\_step\_size))  #메시 그리드에 대해 분류기 실행  output = classifier.predict(np.c\_[x\_vals.ravel(), y\_vals.ravel()])  #결과로 나온 배열 정리  output = output.reshape(x\_vals.shape)  #그래프 생성  plt.figure()  #그래프 색상 지정  plt.pcolormesh(x\_vals, y\_vals, output, cmap=plt.cm.gray)  # 그래프에 학습용 점 그리기  plt.scatter(X[:, 0], X[:, 1], c=y, s=75, edgecolors='black', linewidth=1, cmap=plt.cm.Paired)  # 그래프 경계 지정  plt.xlim(x\_vals.min(), x\_vals.max())  plt.ylim(y\_vals.min(), y\_vals.max())  # x축과 y축에 대한 눈금 표시  plt.xticks((np.arange(int(X[:, 0].min() - 1), int(X[:, 0].max() + 1), 1.0)))  plt.yticks((np.arange(int(X[:, 1].min() - 1), int(X[:, 1].max() + 1), 1.0)))  plt.show() |

앞의 c 값을 100으로 변경하면, 다른 클래스 사이의 경계를 좀더 정확하게 표현할 수 있다.  
c 값을 높이면 알고리즘이 학습 데이터에 좀 더 최적화된다. 단, 이 값을 너무 크게 지정하면 학습 데이터에 치우치는 오버피팅이 발생해 제대로 분류할 수 없다.

*오버피팅 : 샘플 데이터에 너무 정확하게 학습이 되었기 때문에, 샘플 데이터를 가지고 판단을 하면 100%에 가까운 정확도를 보이지만 다른 데이터를 넣게 되면 정확도가 급격히 떨어지는 문제*

**[ 나이브 베이즈 분류기 ]**

나이브 베이즈 (Naïve Bayes) (단순 베이즈, 독립 베이즈) 분류 기법은 베이즈 정리를 기반으로 분류기 만든다. 베이즈 정리(Bayes Theorem) 는 사건이 발생할 확률을 그 사건에 관련된 여러가지 조건을 기반으로 표현한다.

문제의 인스턴스에 대해 클래스 레이블을 할당하는 방식으로 나이브 베이즈 분류기를 만든다.  
 *사과'는 클래스이구요, '내가 엊저녁에 먹은 사과 다섯 개 중에 두 번째 것'이라고 콕 찍어서 말해주면 실체(instance)로 봐줄만합니다*. **즉, 인스턴스는 실제 실체, 클래스는 부류**

문제의 인스턴스는 특징 값으로 구성된 벡터로 표현한다. 이때 주어진 특징 값들은 서로 독립적이라 가정.   
서로 다른 특징들 사이의 관계는 고려하지 않고 주어진 클래스 변수에 대해 개별 특징이 미치는 효과만 본다.

**즉, 나이브 베이즈 분류기는 결과에 영향을 미치는 특징이 서로 영향을 받지 않고 각각 결과에 독립적으로 영향을 미친다고 간주한다.**

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as pyplot  from sklearn.naive\_bayes import GaussianNB  from sklearn import cross\_validation  from utilities import visualize\_classifier  input\_file = 'data\_multivar\_nb.txt'  #입력 파일에서 데이터 가져오기  data = np.loadtxt(input\_file,delimiter=',')  X,y = data[:, :-1],data[:, -1]  #나이브 베이즈 분류기 생성 ( 가우시안 나이브 베이즈 분륙 생성 )  classifier = GaussianNB()  #분류기 학습시키기  classifier.fit(X,y)  #학습한 분류기로 예측한 결과 구하기  y\_pred = classifier.predict(X);  #정확도 계산  accuracy = 100.0 \* (y==y\_pred).sum() / X.shape[0]  print("Accuracy of naive\_bayes classifier =", round(accuracy,2),"%")  #분류기 성능 시각화  visualize\_classifier(classifier,X,y) |

*numpy에서 배열은 동일한 타입의 값들을 가지며, 배열의 차원을 rank 라 하고, 각 차원의 크기를 튜플로 표시하는 것을 shape 라 한다. 예를 들어, 행이 2이고 열이 3인 2차원 배열에서 rank는 2 이고, shape는 (2, 3) 이 된다.*

위의 방식대로 정확도를 측정하는 방식은 완벽하지 않다. 따라서 학습할 때 사용한 데이터로 검증하지 않도록 교차검증(cross validation) 기법을 적용한다. ( ex) 데이터의 80%는 학습용으로 사용하고 나머지 20%는 테스트에 사용하도록 test\_size 매개변수의 값을 0.2로 지정

|  |
| --- |
| #데이터를 학습용과 테스트용으로 나누기  X\_train,X\_test,y\_train,y\_test = cross\_validation.train\_test\_split(X,y,test\_size=0.2,random\_state=3)  classifier\_new = GaussianNB()  classifier\_new.fit(X\_train,y\_train)  y\_test\_pred = classifier\_new.predict(X\_test)  #분류기 정확도 계산  accuracy = 100.0 \* (y\_test==y\_test\_pred).sum() / X\_test.shape[0]  print("Accuracy of naive\_bayes new classifier =", round(accuracy,2),"%")  #분류기 성능 시각화  visualize\_classifier(classifier\_new,X\_test,y\_test) |

분류기의 정확도(accuracy), 정확률(precision)(정밀도),재현율(recall)을 측정, 패키지에서 제공하는 교차 검증 함수로 계산하며 3중 교차 검증 적용

|  |
| --- |
| num\_folds = 3  accuracy\_values = cross\_validation.cross\_val\_score(classifier,  X, y, scoring='accuracy', cv=num\_folds)  print("Accuracy: " + str(round(100\*accuracy\_values.mean(), 2)) + "%")  precision\_values = cross\_validation.cross\_val\_score(classifier,  X, y, scoring='precision\_weighted', cv=num\_folds)  print("Precision: " + str(round(100\*precision\_values.mean(), 2)) + "%")  recall\_values = cross\_validation.cross\_val\_score(classifier,  X, y, scoring='recall\_weighted', cv=num\_folds)  print("Recall: " + str(round(100\*recall\_values.mean(), 2)) + "%")  f1\_values = cross\_validation.cross\_val\_score(classifier,  X, y, scoring='f1\_weighted', cv=num\_folds)  print("F1: " + str(round(100\*f1\_values.mean(), 2)) + "%")  //round 는 반올림, str은 user가 보기 쉽게 보이게끔 |

**[ 오차 행렬 ]**

오차행렬(Confusion matrix) 이란 분류기의 성능을 표현한 그림 또는 표  
이 행렬은 주로 실제 참인 값(GT ground truth 값)이 주어진 테스트 데이터에서 구한다.  
이 행렬을 통해 분류된 결과를 서로 비교해서 잘못 분류된 샘플의 개수를 파악한다.  
이 테이블을 생성할 때 다음과 같이 머신 러닝에서 굉장히 중요하게 다루는 측정 지표 등장

* 참 양성 true positive : 예측 결과가 1이고 , GT 값도 1인 샘플
* 참 음성 true negative : 예측 결과가 0이고, GT 값도 0인 샘플
* 거짓 양성 false positive : 예측 결과가 1이지만 , GT 값은 0인 샘플, 1종 오류 Type1 error라고 부름
* 거짓 음성 false negative : 예측 결과는 0이지만, GT값은 1인 샘플, 2종 오류 Type2 error라고 부름

즉, 주어진 문제에 대해 거짓 양성이나 거짓 음성 비율을 최소화하도록 알고리즘을 최적화해야한다.

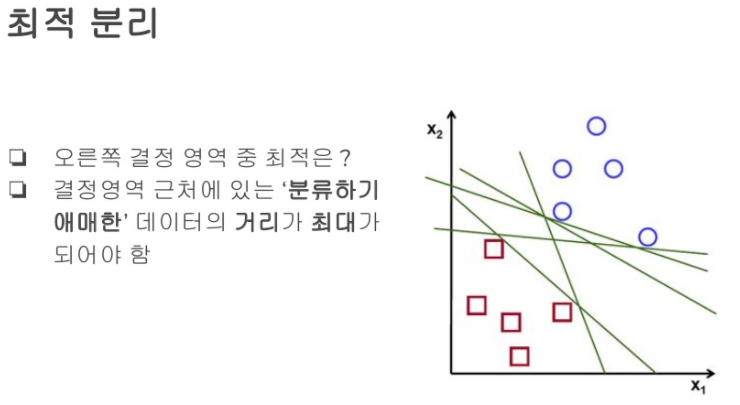
|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  from sklearn.metrics import confusion\_matrix  from sklearn.metrics import classification\_report  #샘플 레이블 정의  true\_labels = [2,0,0,2,4,4,1,0,3,3,3]  pred\_labels = [2,1,0,2,4,3,1,0,1,3,3]  #오차 행렬 생성  confusion\_mat = confusion\_matrix(true\_labels,pred\_labels)  #오차 행렬 시각화  plt.imshow(confusion\_mat,interpolation='nearest',cmap=plt.cm.gray)  plt.title('Confusion matrix')  plt.colorbar()  ticks = np.arange(5)  plt.xticks(ticks,ticks)  plt.yticks(ticks,ticks)  plt.ylabel('True labels')  plt.xlabel('Predicted labels')  plt.show()  #ticks 변수는 클래스의 수 표현  #분류 결과 출력  targets = ['Class-0','Class-1','Class-2','Class-3','Class-4']  print('\n',classification\_report(true\_labels,pred\_labels,target\_names=targets)) |

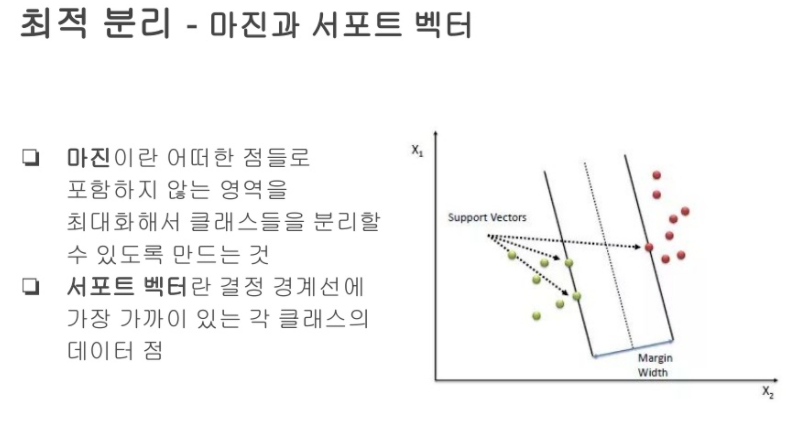
**[ 서포트 벡터 머신 ]**

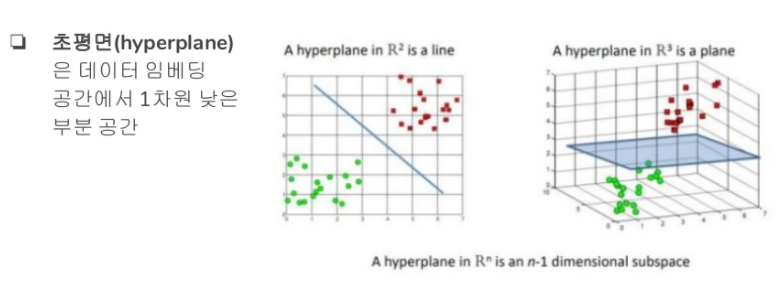
서포트 벡터 머신(SVM,Support Vector Machine) 은 클래스를 구분하는 경계선을 직선이 아닌 초평면으로 표현한다.

초평면 : 직선을 N차원으로 표현 한 것

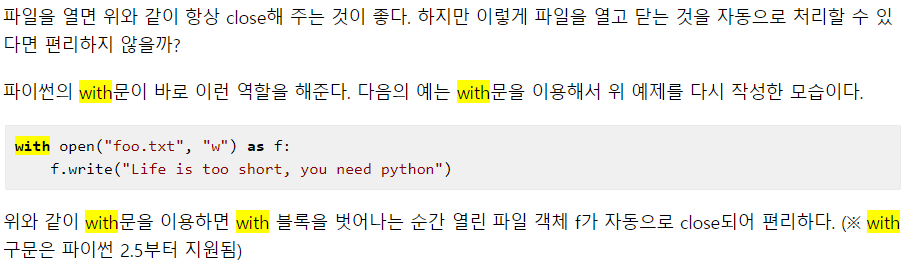
레이블이 달린 학습 데이터가 주어진 이진 분류 문제에 SVM을 적용하면, 학습 데이터를 두 클래스로 가장 잘 나눌 수 있는 초평면을 찾는다. 이 이진 분류 기법은 N개의 클래스로 분류하는 문제로 쉽게 확장 가능

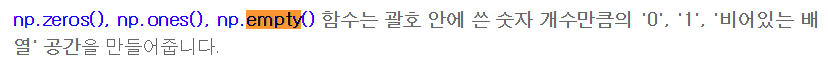






**[ SVM으로 소득 계측 분류하기 ]**





**[ ]**