I. 모델링(Modling) 코드

파이썬라이브러리를 활용한 머신러닝

기본적인 훈련데이터, 테스트데이터 나누는 scikit-learn 코드

- 반드시 X_train, X_test, y_train, y_test 순서로 지정할 것. 대문자는 배열(2차원), 소문자는 벡터를의미. 즉, 훈련세트(set=행렬)인 2차원 배열 X, 타깃인 1차원 벡터 y를 의미

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = \
train_test_split(df,target, test_size=0.2)
```

💡 **매개변수**

`train_test_split(` [훈련데이터의 훈련세트] `,` [훈련데이트의 타깃] `, test_size =` 0.2, stratify = target `)`

`test_size` : 0.2 로 지정시 훈련데이터의 80%를 훈련세트로, 20%를 테스트 세트로 분리
해당 옵션 지정 안할시 기본 25%를 테스트 세트로 분리

(필수지정)
`stratify =` target : 이 매개변수를 지정하면 타깃의 비율에 따라 데이터 분할가능
※ 돌릴때 마다 샘플이 랜덤이 되지 않도록 stratify 대신 `random_state =` n 대체 가능

기본적인 scikit-learn 모델링 순서

1. `from` `sklearn.` [모듈명] `import` [클래스명]
2. [개체명] `=` [클래스명] `.(` 매개변수1, 매개변수2, ... `)`
3. [개체명] `.fit(` [훈련데이터의 훈련세트] `,` [훈련데이터의 타깃] `)`
4. [개체명] `.score(` [검증데이터의 훈련세트] `,` [검증데이터의 타깃] `)`
5. [개체명] `.score(` [테스트데이터의 훈련세트] `,` [테스트데이터의 타깃] `)`
6. y_pred `=` [개체명] `.predict(` [테스트 데이터의 훈련세트] , [테스트 데이터의 타깃] `)`

   np.mean(y_pred == y_test) # 정확도 계산 가능

- [ ] 부분은 사용자 지정부분, 개체명은 모두 동일

k-최근접 이웃 알고리즘[분류, 회귀]

- 기본적으로 여러 여러 환경에서 동작하는 유클리디안 거리방식 사용
- 이해하기 매우 쉬운 모델이라 더 복잡한 알고리즘을 적용해보기 전에 시도해볼 수 있는 좋은 시작점.
- 장점
  1. 결과 설명이 쉬움
  2. 예측 정확도가 높음
- 단점
  1. 훈련 세트가 너무 크거나(특성의 수나 샘플의 수가 클경우) 예측이 느려짐. 사실 데이터 훈련이 아닌 데이터 저장만 하기 때문
  2. k 값 결정이 어려움.(통상 훈련데이터의 제곱근으로 정함)

```
#분류
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors= 1, p = 2)

#회귀
from sklearn.neighbors import KNeighborsRegressor
reg = KNeighborsRegressor(n_neighbors = 1, p = 2)
```

💡 **중요 매개변수**

`n_neighbors =` : 최근접 이웃의 개수를 지정

`p =` : 1이 맨허튼 거리, 2가 유클리디안 거리

I. 선형 모델

- 학습 속도가 빠르고 예측도 빠르며 매우 큰 데이터셋과 희소한 데이터셋에도 잘 작동
- 회귀와 분류에서 본 공식을 사용해 예측이 어떻게 만들어지는지 비교적 쉽게 이해 가능
  단, 데이터셋의 특성들이 서로 깊게 연관되어 있다면 계수 값 설명이 어려울 수 있음.
- 샘플에 비해 특성이 많을 때 잘 작동함.
  단, 저차원의 데이터셋에서는 다른 모델들의 일반화 성능이 더 좋음.

단순회귀분석

- 예측과 훈련 세트에 있는 타깃 y 사이의 평균제곱오차(mean squared error)를 최소화하는 파라미터 w, b를 찾음
- 매개변수가 없는 것이 장점이나 그래서 모델의 복잡도를 제어할 방법이 없음.

```
from sklearn.linear_model import LinearRegression

# 선형 모델 적합
lr = LinearRegression().fit(X_train, y_train)

# 절편(intercept)은 intercept_ 속성에 저장, 가중치(weight)는 coef_ 속성에 저장
print(lr.intercept_ , lr.coef_)
```

리지(Ridge) 회귀

- 리지도 최소적합법과 사용하는것 같은 예측함수 사용
- 리지 회귀에서의 가중치(w) 선택은 훈련 데이터를 잘 예측하기 위하면서도 추가 규제(regularization) 조건을 만족시키기 위한 목적도 있음.
- 추가 규제 조건이란 w의 모든 원소가 0에 가깝게 되길 원함. (L2 규제)
- alpha 값으로 매개변수 조정가능. alpha 값을 높이면 그만큼 제약이 더 가해져 가중치를 0에 더 가깝게 만들어 훈련 세트의 성능은 나빠지나, 일반화에는 도움을 줄 수 있음.

```
from sklearn.linear_model import Ridge

ridge = Ridge().fit(X_train, y_train)
```

💡 **매개변수**

`alpha =` : 아주 작은 alpha 값은 계수를 거의 제한하지 않으므로 LinearRegrssion으로 만든 모델과 거의 같아짐. 기본값은 1

라소(Lasso) 회귀

- 라소도 리지와 마찬가지로 가중치를 0에 가깝게 만들려고 함. 방식이 달라 L1 규제라 함.
- 어떤 가중치(계수)는 정말 0이 되어 이모델에서 완전히 제외되는 특성이 생기게 됨. 특성 선택(feature selection)이 자동으로 이루어 짐.
- max_iter로 반복 시행하는 최대횟수를 지정할 수 있으며 한 특성씩 좌표축을 따라 최적화하는 좌표하강법(coordiniate descent) 방식을 사용하여 학습 과정이 반복적으로 여러번 진행되어 최적값 찾음.
- 실제로는 리지를 더 선호하나 특성이 많고 그중 일부분만 중요하다면 라소 선호. 또한 입력 특성 중 일부만 사용하므로 쉽게 해석할 수 있는 모델 만듦.

```
from sklearn.linear_model import Lasso

Lasso = Lasso(alpha=0.01, max_iter = 100000).fit(X_train, y_train)
```

💡 **매개변수**

`alpha =` : 계수를 얼마나 강하게 0으로 보낼지를 조절하는 alpha 매개변수 지원

`max_iter=`: 반복 실행하는 최대 횟수 (n_iter_속성에서 확인가능)

엘라스틱 넷(ElasticNet) 회귀

- L1와 L2의 규제를 조정함.
- 이 회귀는 최고의 성능을 내나 두개의 매개변수를 조절해야되는 단점 존재.

회귀용 선형모델와 분류용 선형 모델의 차이

- 회귀용 선형모델에서는 직선, 평면, 초평면이 선형 함수
- 분류용 선형모델에서는 결정경계가 입력의 선형 함수. 선, 평면, 초평면을 사용해 클래스를 구분하는 분류기임.
- 분류용 선형모델은 잘못된 분류의 수를 최소화하도록 w, b를 조정하는 것은 불가능. 손실함수에 대한 차이는 크게 중요하지 않음
- 회귀 모델에서는 alpha가 주요 매개변수며 LinearSVC와 LogisticRegression에서는 C가 주요 매개변수. alpha 값이 클수록, C 값이 작을수록 모델이 단순해짐.
- 보통 C와 alpha는 로그 스케일로 최적치를 정함.(자릿수가 바뀌도록 10배씩 변경함. 0.01,0.1..)
- 중요한 특성이 많지 않다고 생각하면 L1 규제를 사용, 그렇지 않으면 기본적으로 L2 규제 사용
  L1 규제는 모델의 해석이 중요한 요소일 때도 사용할 수 있음. L2 규제는 몇가지 특성만 사용하므로 해당 모델에 중요한 특성이 무엇이고 그 효과가 어느 정도인지 설명하기 쉬움.

로지스틱 회귀(logistic regression) - 분류용 선형 모델

- 이진 분류에서는 로지스틱(logistic) 손실함수를 사용하고 다중 분류에서는 cross-entropy 손실함수를 사용
- 매개변수 C의 값이 낮아지면 데이터 포인트 중 다수에 맞추려는 반면, C의 값을 높이면 개개의 데이터 포인트를 정확히 분류하려고 노력함. 하지만 C값을 너무 높이면 과대적합 문제 발생
- L2 규제 사용

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=100, max_iter = 5000, solver='sag').fit(X_train, y_train)
```

💡 **매개변수**

`C=` : 값이 높을수록 과대적합 위험이 상승

`max_iter=`: 반복 실행하는 최대 횟수

`solver='sag'` : 수십만에서 수백만 개의 샘플로 이뤄진 대용량 데이터셋과 희소한 데이터셋에서 잘 작동하기 위해 solver = 'sag' 옵션을 줌. 다른 대안으로는 선형 모델의 대용량 처리 버전으로 SGDClassifier, SGDRegressor 사용 가능.

서포트 벡터 머신(support vector machine) - 분류용 선형 모델

- 기본 손실함수는 힌지(squared hinge)손실함수를 사용

```
from sklearn.svm import LinearSVC
linear_svm = LinearSVC(C=100, max_iter = 5000).fit(X_train, y_train)
```

💡 **매개변수**

`C=` : 값이 높을수록 과대적합 위험이 상승

`max_iter=` : 반복 실행하는 최대 횟수

로지스틱 회귀 vs 그 밖의 많은 선형 분류 모델

- 로지스틱을 제외한 그 밖의 선형분류모델은 태생적으로 이진분류만을 지원
- 이진 분류 알고리즘을 다중 클래스 분류 알고리즘으로 확장하는 보편적인 기법은 일대다(one-vs, -rest)방법. 일대다 방식은 각 클래스를 다른 모든 클래스와 구분하도록 이진 분류 모델을 학습
- 결국 클래스의 수만큼 이진 분류 모델이 만들어짐. 예측 할 때 이렇게 만들어진 모든 이진 분류기가 작동하여 가장 높은 점수를 내는 분류기의 클래스를 예측값으로 선택

II. 나이브 베이즈 분류기(naive bayes)

- LinearSVC 같은 선형 분류기보다 훈련 속도가 빠름, 그 대신 일반화 성능이 조금 뒤짐
- 나이브 베이즈 분류기가 효과적인 이유는 각 특성을 개별로 취급해 파라미터를 학습하고 각 특성에서 클래스별 통계를 단순 취합하기 때문.
- 희소한 고차원 데이터에서 잘 작동하며 비교적 매개변수에 민감하지 않음.
- 선형 모델로는 학습 시간이 너무 오래 걸리는 매우 큰 데이터셋에서는 나이브 베이즈 모델 시도해볼만하며 종종 사용 됨.

나이브 베이즈 분류기 비교

Aa 분류기 명	고유속성
제목 없음	
GaussianNB	1. 연속적인 어떤 데이터에도 적용 가능 2. 대부분 매우 고차원인 데이터셋에 사용 3. 데이터를 분류할 때 사용 (클래스별로 특성의 표준편차와 평균을 저장)
MultinormialNB	1. 카운트 데이터에 적용 2. 비교적 많은 데이터셋(큰 문서들)인 경우 BernoulliNB보다 성능이 좋음. 1. 텍스트 데이터를 분류할 때 사용 (클래스별로 특성의 평균을 계산)
BernoulliNB	1. 이진 데이터에 적용 2. 각 클래스의 특성 중 0이 아닌 것이 몇 개인지 셈

MultinormialNB와 BernoulliNB 공통점

Aa 공통점
1. 예측 공식은 선형 모델과 형태가 같음(단, 기울기 w가 아니라 선형모델과 의미가 다름)
2. 모델의 복잡도 조절하는 alpha 매개변수 가짐
3. 텍스트와 같은 희소한 데이터 카운트시 사용

III. 결정트리(decision tree)

- 분류와 회귀에 널리 사용하는 모델
- 결정 트리를 학습한다는 것은 정답에 가장 빨리 도달하는 예/아니요 질문 목록을 학습한다는 뜻
- 머신러닝에서는 이런 질문들을 테스트라고 부름.

- 각 테스트는 하나의 축을 따라 데이터를 둘로 나누는 것이며 이는 계층적으로 영역을 분할해가는 알고리즘이라 부름.
- 뒤에 나오는 랜덤포레스트 대신에 단일 트리를 사용해야 할 수 있음. 만약 의사 결정을 간소하게 표현해야 한다면 단일 트리를 사용 함.
  수십, 수백개의 트리를 자세히 분석하기 어렵고 랜덤 포레스트의 트리는(특성의 일부만 사용하므로) 결정 트리보다 더 깊어지는 경향도 있기 때문.
  따라서 비전문가에게 예측 과정을 시각적으로 보여주기 위해서 하나의 결정 트리가 좋은선택

결정트리의 복잡도 제어

- 일반적으로 트리 만들기를 모든 리프 노드가 순수 노드가 될 때까지 진행되며, 모델이 매우 복잡해지고 훈련 데이터에 과대 적합됨.
- 순수 노드로 이루어진 트리는 훈련 세트에 100% 정확하게 맞는다는 의미
- 과대 적합을 막는 전략은 크게 두가지 **사전가지치기**(pre-pruning), **사후가지치기**(또는 가지치기 post-pruning(prunging)) 임.
- scikit - learn은 사전 가지치기만 지원

사전 가지치기(pre - pruning)

- 트리 생성을 일찍 중단하는 전략
- 트리의 최대 깊이나 리프의 개수를 제한하거나 또는 노드가 분할하기 위한 포인트의 최소 개수를 지정하는 것
- 결정 트리의 깊이를 제한하지 않으면 트리는 무한정 깊어지고 복잡해질 수 있음.
- 따라서 가지치기하지 않은 트리는 과대적합되기 쉽고 새로운 데이터에 잘 일반화되지 않음.

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth = 4,
          max_leaf_nodes = 4, min_samples_leaf = 4, min_samples_split = 40,
          min_impurity_decrease = 4, n_jobs = -1)
tree.fit(X_train, y_train)
```

💡 **매개변수**

`max_depth =` : 연속된 질문을 최대 4개로 제한함으로서 과대적합을 줄임.(사전가지치기)
`max_leaf_nodes =` : 리프 노드의 최대 개수를 지정(사전가지치기)
`min_samples_leaf =` : 리프 노드가 되기 위한 최소한의 샘플 개수 지정(사전가지치기)
`min_samples_split =` : 노드가 분기할 수 있는 최소 샘플 개수 지정(사전가지치기)
`min_impurity_decrease =` : 불순도(impurity) 감소 최솟값을 지정(사전가지치기)
* (사전가지치기) 방법 중 `max_depth`, `max_leaf_nodes`, `min_samples_leaf` 중 하나만 지정해도 과대 적합을 막는데 충분함.
`n_jobs = -1` : 컴퓨터의 모든코어 사용(-1 지정시)

결정 트리 시각화

- sklearn.tree 로 의사결정나무 시각화
- 알고리즘의 예측이 어떻게 이뤄지는지 잘 이해할 수 있으며 비전문가에게 머신러닝 알고리즘 설명하기 좋음.

```
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree
plt.figure(figsize=(10,7))
plot_tree(tree, filled = True)
```

💡 **매개변수**

`filled =` : 시각화한 상자에 색 채워넣기

트리의 특성 중요도

- 전체 트리를 살펴보는 것이 어려울 때, 트리가 어떻게 작동하는지 요약하는 속성들을 사용 가능
- 가장 널리 사용되는 속성은 트리를 만드는 결정에 각 특성이 얼마나 중요한지 평가하는 것:
  특성 중요도(feature importance)이라 함.
- 특성 중요도는 0 ~ 1사이의 숫자로 각 특성에 대해 0은 전혀 사용되지 않음을, 1은 완벽하게 타깃 클래스를 예측했다는 의미
- **선형 모델의 계수와는 달리 특성 중요도 값이 낮다고 해서 이 특성이 유용하지 않다는 뜻은 아님.**
  단지 트리가 그 특성을 선택하지 않았으며 다른 특성이 동일한 정보를 지니고 있기 때문일 수 있음.
- **선형 모델의 계수와는 달리, 특성 중요도는 항상 양수이며 특성이 어떤 클래스를 지지하는지는 알 수 없음.** 즉 특성 중요도가 값이 가장 높다고 이를 어느 한 쪽 클래스로 단정지을 수 없음.
  사실 특성과 클래스 사이에는 간단하지 않은 관계가 있을 수 있기 때문
- 위 사항들은 DecisionTreeRegressor로 구현된 회귀 결정트리에서도 비슷하게 적용 됨.

```
tree.feature_importance_
```

트리 기반 회귀 모델(DecisionTreeRegressor)

- 모든 다른 트리 기반 회귀 모델은 외삽(extrapolation), 즉 훈련 데이터의 범위 밖의 포인트에 대해 예측할 수 없음.
- 트리 모델은 데이터 범위 밖으로 나가면 단순히 마지막 포인트를 이용해 예측하는게 전부이기 때문에 일정하게 값이 유지됨.
- 따라서 범위 밖의 포인트에 대해 예측하기 위해서는 선형 모델을 사용해야 함.

IV. 결정 트리의 앙상블

- 앙상블(ensemble)은 여러 머신러닝 모델을 연결하여 더 강력한 모델을 만드는 기법
- 앙상블은 머신러닝의 여러 종류의 모델 중 분류와 회귀 문제의 다양한 데이터셋에서 효과적이라 입증됨.
- **랜덤포레스트(random forest)**와 **그레이디언트 부스팅(gradient boosting) 결정 트리**는 둘 다 모델을 구성하는 기본 요소로 결정 트리를 사용 함.

랜덤 포레스트(random forest)

- 결정 트리의 단점은 훈련 데이터에 과대적합되는 경향이 있다는 것.
- 랜덤 포레스트는 기본적으로 조금씩 다른 여러 결정 트리의 묶음
- 각 트리는 비교적 예측을 잘 할 수 있지만 데이터의 일부에 과대적합하는 경향을 가진다는 데 기초 함.
- 잘 작동하되 서로 다른 방향으로 과대적합된 트리를 많이 만들면 그 결과를 평균냄으로써 과대적합된 양을 줄일 수 있음. 이로써 트리 모델의 예측 성능이 유지되면서 과대적합이 줄어드는 것이 수학적으로 증명됨.
- 이런 전략 구현을 위해 결증 트리를 많이 만들어야 함. 각각의 트리는 타깃 예측을 잘해야 하며 다른 트리와는 구별되어야 함.
- 랜덤 포레스트에서 트리를 랜덤하게 만드는방법은 두 가지. **데이터 포인트를 무작위로 선택하는 방법**과 **분할 테스트에서 특성을 무작위로 선택하는 방법**
- 무작위성은 알고리즘이 가능성 있는 많은 경우를 고려할 수 있도록 하므로, 그 결과 랜덤 포레스트가 단일 트리보다 더 넓은 시각으로 데이터를 바라볼 수 있음.
- **데이터의 스케일을 맞출 필요 없음**
- 랜덤 포레스트는 텍스트 데이터 같이 매우 차원이 높고 희소한 데이터에는 잘 작동하지 않음.
  선형 모델이 더 적합
- n_estimators 값이 크면 클 수록 좋음. 더 많은 트리를 평균하면 과대적합을 줄여 더 안정적인 모델을 만들기 때문. 다만 더 많은 메모리와 긴 훈련 시간이 필요

랜덤 포레스트 구축순서

1. 생성할 트리의 개수를 정한다. (n_estimators 매개변수, 기본값은 10)
2. 트리들이 독립적으로 만들기 위해 데이터의 **부트스트랩 샘플(boostrap sample)**생성
   n_samples 개의 데이터 포인트 중에서 무작위로 데이터를 중복하여 반복 추출
   * 이 데이터셋(**부스트랩 샘플**)은 원래 데이터셋 크기와 같지만 대략 1/3 누락되어 있음.
3. 이렇게 만든 데이터셋으로 결정트리 만듬.

- 1) 알고리즘이 각 노드에서 후보 특성을 무작위로 선택한 후 이 후보들 중 최선의 테스트를 찾음.
  * 결정 트리 알고리즘은 전체 특성을 대상으로 최선의 테스트를 하는것과 비교 됨.
  * 몇 개의 특성을 고를지는 max_features 매개변수로 조정할 수 있음.
- 2) 후보 특성을 고르는 것은 매 노드마다 반복되므로 트리의 각 노드는 다른 후보 특성들을 사용하여 테스트 만듬.

[정리] 부트스트랩 샘플링은 랜덤 포레스트의 트리가 조금씩 다른 데이터셋을 이용해 만들어지도록 함. 만들어진 랜덤 포레스트의 트리는 각 노드에서 특성의 일부만 사용하여 각 분기는 각기 다른 특성 부분 집합을 사용. 이 두 메커니즘이 합쳐져 랜덤 포레스트의 모든 트리가 서로 달라짐

핵심 매개변수 max_features

- 이를 n_features로 설정하면 트리의 각 분기에서 모든 특성을 고려하므로 특성 선택에 무작위성이 들어가지 않음. (단, 부트스트랩 샘플링의 무작위성은 그대로)
- max_features=1 이면 트리의 분기는 테스트할 특성을 고를 필요가 없게 되며 그냥 무작위로 선택한 특성의 임계값을찾기만 하면 됨.
- max_features를 크게 하면 랜덤 포레스트의 트리들은 매우 비슷해지며 가장 두드러진 특성을 이용해 데이터에 잘 맞춰짐.
- 반대로 낮추면, 랜덤 포레스트 트리들은 많이 달라지고 각 트리는 데이터에 맞추기 위해 깊이가 깊어짐. 따라서 과대적합을 줄여 줌.
- 일반적으로 기본값을 쓰는 것이 좋은 방법
  분류 : max_features = sqrt(n_features)
  회귀 : max_features = n_features

랜덤 포레스트의 예측 순서

1. 먼저 알고리즘이 모델에 있는 모든 트리의 예측을 만듦.
   (**회귀** : 예측들의 평균하여 최종예측 산출 / **분류** : 약한 투표 전략 사용)
2. 각 알고리즘이 가능성 있는 출력 레이블의 확률을 제공함으로써 간접적인 예측을 함.
3. 트리들이 예측한 확률을 평균내어 가장 높은 확률을 가진 클래스가 예측값이 됨.

랜덤 포레스트 분석

```
forest = RandomForestClassifier(n_estimators = 5, random_state = 42, n_jobs = -1)
forest.fit(X_train, y_train)

# 랜덤 포레스트안에 만들어진 트리 정보
forest.estimators_
```

💡 **매개변수**

`n_estmators =` : 생성할 트리개수 지정(기본값은 10)

`random_state =` : 42

* 랜덤이기 때문에 같은 결과를 위해서는 random_state 값을 고정해야 함.

그레이디언트 부스팅 회귀 트리

- 여러 개의 결정트리를 묶어 강력한 모델을 만드는 또 다른 앙상블 방법
- 이름은 회귀이나 이 모델은 회귀, 분류 모두 사용 가능

- 랜덤 포레스트와는 달리 그레이디언트 부스팅 회귀 트리에는 무작위성이 없음. 대신 강력한 사전 가지치기가 사용됨.
- 보통 하나에서 다섯 정도의 깊지 않은 트리를 사용하므로 메모리를 적게 사용하고 예측도 빠름
  이런 얕은 트리 같은 간단한 모델(약한 학습기 weak learner)을 많이 연결하는것.
- 각각의 트리는 데이터의 일부에 대에서만 예측을 잘 수행하므로 트리가 많이 추가될수록 성능이 좋아짐.
- 랜덤 포레스트보다 매개변수 설정에 조금 더 민감
- 대규모 머신러닝 문제에 그레이디언트 부스팅 적용하려면 xgboost 패키지와 파이썬 인터페이스를 검토해보는것이 좋음. (scikeit-learn의 그레이디언트 부스팅보다 빨랐음)

핵심 매개변수 learning_rate

- 앙상블 방식에 있는 사전 가지치기나 트리 개수 외에도 그레이디언트 부스팅에서 중요한 매개변수는 이전 트리의 오차를얼마나 강하게 보정할 것인지를 제어하는 learning_rate 임.
- 학습률이 크면 트리는 더 많이 추가되어 모델의 복잡도가 커지고 훈련 세트에서의 실수를 바로잡을 기회가 더 많아짐.
- 이전에 만든 트리의 예측과 타깃값 사이의 오차를 줄이는 방향으로 새로운 트리를 추가하는 알고리즘
  1) 손실함수 정의 후
  2) 경사하강법을 사용하여 다음에 추가될 트리가 예측해야 할 값을 보정해 감.

그레이디언트 부스팅 회귀 구축

```
gbrt = GradientBoostingClassifier(random_state =0, max_depth = 1, learning_rate = 0.01)
gbrt.fit(X_train, y_train)
```

💡 **매개변수**

`max_depth =` : 연속된 질문을 최대 4개로 제한함으로서 과대적합을 줄임.(사전가지치기)

`learning_rate = 0.01` :

랜덤포레스트 vs 그레이디어트 부스팅 회귀

- 비슷한 종류의 데이터에서 그레이디언트 부스팅과 랜덤 포레스트 둘 다 잘 작동하지만, 보통 더 안정적인 랜덤 포레스트를 먼저 적용 함.
- 랜덤 포레스트가 잘 작동하더라도 예측 시간이 중요하거나 머신러닝 모델에서 마지막 성능까지 쥐어짜야 할 때 그레이디언트 부스팅을 사용함.

배깅(Bagging)

- Bootstrap aggregating의 줄임말로 중복을 허용한 랜덤 샘플링으로 만든 훈련 세트를 사용하여 분류기를 각기 다르게 학습 시킴.
- 부트스트랩 샘플을 만드는 것은 앞서 랜덤포레스트의 특징과 같음.
- 분류기가 predict_proba() 메서드를 지원하는 경우 확률값을 평균하여 예측을 수행 함. 그렇지 않은 경우 빈도가 높은 클래스 레이블이 예측 결과가 됨.

배깅을 활용한 100개의 로지스틱 회귀모델 훈련하여 앙상블

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import BaggingClassifier

bagging = BaggingClassifier(LogisticRegression(), n_estimators = 100, oob_score =True,
                            n_jobs = -1 , random_state= 42)
bagging.fit(Xc_train, yc_train)
```

💡 **매개변수**

`n_estimators =` : LogisticRegression 객체를 기반 분류기로 전달하고 훈련할 분류기의 개수 100개로 지정

`oob_score = True :` oob_score 값을 통해 테스트 세트의 성능 짐작 가능. Out Of Bag 오차라 부르며 부트스트래핑에 포함되지 않은 샘플을 기반으로 훈련된 모델을 평가함.

배깅을 활용한 100개의 결정 트리 모델 훈련하여 앙상블 vs 랜덤포레스트

- 결정 트리에 배깅을 수행해보면 랜덤 포레스트와 매우 비슷한 결과가 산출 됨.
  (결정트리에 배깅을 수행하는 것보다 랜덤포레스트를 수행하는 것이 일반적)
- 샘플의 크기를 지정 못하는 랜덤 포레스트와는 달리 max_samples 매개변수에서 부트스트랩 샘플의 크기 지정 가능
- splitter = 'random' 이라 설정하면

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier

bagging = BaggingClassifier(DecisionTreeClassifier(splitter = 'random', max_samples = 40), n_estimators = 100,
                            oob_score = True, n_jobs = -1, random_state = 42)
bagging.fit(Xc_train, yc_train)
```

💡 **매개변수**

`splitter = 'random'` **:** 무작위로 분할한 후보 노드 중에서 최선의 분할을 찾음.
(random : 엑스트라트리, best : 랜덤포레스트)

`max_samples = 40` : 부트스트랩 샘플의 크기 지정

`n_estimators =` : 결정트리가 기반 분류기 이므로 훈련할 분류기의 개수 100개로 지정

`oob_score = True :` oob_score 값을 통해 테스트 세트의 성능 짐작 가능. Out Of Bag 오차라 부르며 부트스트래핑에 포함되지 않은 샘플을 기반으로 훈련된 모델을 평가함.

엑스트라 트리(extra tree)

- 랜덤 포레스트와 비슷하나 후보 특성을 무작위로 분할한 다음 최적의 분할을 찾는 다는점이 랜덤포레스트와 다름.
- 랜덤 포레스트는 splitter = 'best' 를 사용하나 엑스트라 트리는 splitter='random'을 사용하고 부트스트랩 샘플링은 적용하지 않음.
- 무작위성을 증가시키면 일반적으로 모델의 편향이 늘어나지만 분산이 감소함. 즉, 엑스트라 트리와 랜덤 포레스트는 다른 방식으로 모델에 무작위성을 주입함.
- 예측 방식은 랜덤포레스트와 동일하게 각 트리가 만든 확률값을 평균 함.

엑스트라 트리를 활용한 100개의 결정 트리 모델 훈련하여 앙상블

```
from sklearn.ensemble import ExtraTreesClassifier
xtree = ExtraTreesClassifier(n_estimators = 100, n_jobs = -1, random_state = 0)
xtree.fit(Xc_train, yc_train)
```

💡 **매개변수**

`n_estimators =` : 결정트리가 기반 분류기 이므로 훈련할 분류기의 개수 100개로 지정

에이다 부스트(AdaBoost)

- Adaptive Boosting의 줄임말
- 에이다부스트는 그레이디언트 부스팅처럼 약한 학습기를 사용
- 그레이디언트 부스팅과 마찬가지로 순차적으로 학습해야 하기 때문에 n_jobs 매개변수는 지원하지 않음.

- scikit-learn의 AdaBoostClassifier는 기본적으로 DecisionTreeClassifier(max_depth = 1)을 사용하고 AdaBoostClassifier는 기본적으로 DecisionTreeRegressor(max_depth = 3)을 사용하나 base_estimator 매개변수에서 다른 모델을 지정할 수 있음.
- 아주 얕은 트리를 앙상블하여 일반화 성능이 조금 더 향상 됨.

에이다 부스트 과정

1. 그레이디언트 부스팅과는 달리 이전의 모델이 잘못 분류한 샘플에 가중치를 높여 다음 모델을 훈련시킴
2. 훈련된 각 모델은 성능에 따라 가중치가 부여됨.
3. 예측을 만들 때는 모델이 예측한 레이블을 기준으로 모델의 가중치를 합산하여 가장 높은 값을 가진 레이블을 선택

에이다 부스트를 활용한 100개의 결정 트리 모델 훈련하여 앙상블

```
from sklearn.ensemble import AdaBoostClassifier
ada = AdaBoostClassifier(n_estimators=100, random_state = 42)
ada.fit(X_train, y_train)
```

💡 **매개변수**

`n_estimators =` : 결정트리가 기반 분류기 이므로 훈련할 분류기의 개수 100개로 지정

V. 커널 서포트 벡터 머신(Kernelized Support Vector Machines)

- 분류와 회귀문제 모두 같이 적용됨
- 직선과 초평면은 유연하지 못하여 저차원 데이터셋에는 선형 모델이 매우 제한적. 이때 더 복잡한 모델을 만들 수 있도록 확장한 것임.
  (물론 선형 모델을 유연하게 만드는 방법은특성끼리 곱하거나 특성을 거듭제곱하는 방법있음)
- 많은 경우 어떤 특성을 추가해야 할지 모르고 특성을 많이 추가하면(100개의 특성에서 가능한 모든조합)연산 비용이 커짐.
- 이때 수학적 기교를 사용해 새로운 특성을많이 만들지 않고 고차원에서 분류기로 학습가능
  : 이를 **커널기법(kernel trick)**이라고 함.
- 커널 기법은 실제로 데이터를 확장하지 않고 확장된 특성에 대한 데이터 포인트들의 거리(더 정확히는 스칼라 곱)을 계산 함.
- **SVM은 매개변수 설정과 데이터 스케일에 매우 민감**함. 따라서 입력 특성의 범위가 비슷한 데이터 전처리를 반드시 진행해야 함.
- 모든 특성이 비슷한 단위(ex. 모든 값이 픽셀의 컬러 강도) 스케일이 비슷하면 해볼만 함.
- 매개변수 설정에 민감하고 예측이 어떻게 결정되었는지 이해하기 어려울 뿐더러 비 전공자에게 모델을 설명하기 난해함.
- 10,000개 샘플 정도면 SVM이 잘 작동하나 100,000개 이상의 데이터셋에는 속도와 메모리 관점에서 도전적 과제

서포트 벡터 머신에서 데이터를 고차원 공간에 매핑법

- 1) 원래 특성의 가능한 조합을 지정된 차수까지 모두계산 : **다항식 커널**
- 2) 차원이 무한한 특성 공간에 매핑하는 : **가우시안 커널 또는 RBF(Fradial basis function)커널**
  모든 차수의 모든 다항식을 고려하나, 특성의 중요도는 고차항이 될수록 줄어듬
  (테일러 급수 전개로인해)

서포트 벡터 학습 절차(두 클래스 일 경우)

1. 훈련 데이터의 일부인 두 클래스 사이의 경계에 위치한 데이터 포인트들만이 결정 경계를 만드는데 영향을 줌. 이런 데이터 포인트를 '**서포트 벡터(support vector)**' 라 부름
2. 새로운 데이터 포인트에 대해 예측하려면 각 서포트 벡터와의 거리를 측정함. 분류 결정은 서포트 벡터까지의 거리에 기반하여 서포트 벡터의 중요도는 훈련 과정에서 학습.
   (SVC 객체의 dual_coef_ 속성에 저장됨)

이 데이터 포인트 사이의 거리는 **가우시안 커널에 의해 계산**

$k_{(rbf)}(x_1, x_2) = exp(-\gamma||x_1 - x_2||^2)$

x_1와 x_2는 데이터 포인트며 $||x_1 - x_2||^2$는 유클리디안 거리이고 γ(감마)는 가우시안 커널의폭을 제어하는 매개변수

핵심 서포트 벡터 매개변수 gamma, C

- 감마(γ) 매개변수는 가우시안 커널 폭의 역수에 해당하며 이는 하나의 훈련 샘플이 미치는 영향의 범위를 결정함. 즉, 1 ~ 0 사이인 감마가 0에 가까워 질수록(가우시안 커널의 반경이 커질수록) 훈련샘플의 영향 범위도 커짐.
- C 매개변수는 선형 모델에서 사용한 것과 비슷한 규제 매개변수. 이 매개변수는 각 포인트의 중요도(dual_coef_ 값)을 계산함. (자릿수가 바뀌도록 10배씩 변경함. 0.01,0.1..)
- gamma와 C 모두 모델의 복잡도를 조정하며 둘 다 큰 값이더 복잡한 모델을 만들기 때문에 C와 gamma를 함께 조정해야 함.

서포트 벡터 모델링

```
from sklearn.svm import SVC

# 훈련 세트와 테스트 세트가 전처리 되었다고 가정
svc = SVC()
svc.fit(X_train_scaled, y_train, log_C=-1, log_gamma=-1)
```

💡 **매개변수**
`log_C=-1`: 10^{-1} 의 의미. log를 옆으로 넘기면 이해하기 쉬움.
`log_gamma=-1` : 10^{-1} 의 의미. log를 옆으로 넘기면 이해하기 쉬움.

VI. 신경망(딥러닝)

- 신경망이라 알려진 알고리즘들은 최근 딥러닝(deep learning)이란 이름으로 다시 주목을 받음
- 복잡한 딥러닝 알고리즘의 출발점이며 비교적 간단하게 분류와 회귀에 쓸 수 있는 다층 퍼셉트론(multilayer perceptrons, MLP)을 보겠음.
- skitit-learn에서는 합성곱 신경망, 순환 신경망이 구현되어 있지 않음.
  > **핸즈온 머신러닝 참고. 여기서는 깊게 다루지 않음.**

분류를 위한 다층 퍼셉트론

- 매끄러운 결정 경계를 얻기 위해서는 은닉 유닛이나 은닉층을 추가한다거나(hidden_layer), 또는 tanh 함수를 사용할 수 있음.(activation = 'tanh')
- 리지회귀와 선형 분류기에서 한것처럼 L2 페널티를 사용해 가중치를 0에 가깝게 감소시켜 모델의 복잡도 제어 가능(alpha =)
- random_state= 값 지정해줘야 하는 이유는 신경망에서 학습 시작하기전 가중치를 무작위로 설정하며 이 무작위한 초기화가 모델의 학습에 영향을주기 때문. 신경망이 크고 복잡도도 적절하면 이런 점이 정확도에 미치는 양향은 크지 않으나 항상 기억하고 있어야 함.
- 훈련 데이터의 전처리는 반드시 필수. SVM과 마찬가지로 스케일에 영향을 받음.

```
from sklearn.neural_network import MLPClassifier


mlp = MLPClassifier(solver = 'lbfgs', activation = 'tanh', random_state = 0, hidden_layer_sizes = [10, 10],
                    max_iter=1000, alpha = 0.0001)
# 훈련데이터가 전처리되었다고 가정
mlp.fit(X_train_scaled, y_train)
mlp.score(X_train_scaled, y_train)
mlp.score(X_test_scaled, y_test)
```

💡 **매개변수**

`solver = 'lbfgs' :` 최적화 알고리즘 설정. (LBFGS 알고리즘 설정됨.)

`hidden_layer_sizes = [10,10]` : 은닉 유닛 10개를 가진 두 개의 은닉층 생성

`max_iter=` : 반복 횟수 지정

`activation =` 'tanh' : tanh 활성화 함수 지정

`alpha =` : L2 패널티 지정

`random_state =` : 신경망 학습 전 가중치 무작위 지정되나 매개변수의 변화에 따른 신경망 모델을 살펴보기 위해서 무작위값 임의로 지정

신경망의 장점과 단점

장점

- 대량의 데이터에 내재된 정보를 잡아내고 매우 복잡한 모델을 만들 수 있음
- 충분한 연산 시간과 데이터를 주고 매개변수를 세심하게 조정하면 신경망은 분류와 회귀에서 모두 종종 다른 머신러닝 알고리즘을 뛰어넘는 성능을 냄

단점

- 신경망은 크고 강력한 모델일수록 종종 학습이 오래 걸리며 데이터 전처리에 주의해야 함.
- SVM과 비슷하게 모든 특성이 같은 의미를 가진 동질의 데이터에서 잘 작동
- 다른 종류의 특성을 가진 데이터라면 트리 기반 모델이 더 잘 작동할 수 있음.
- 신경망 매개변수 튜닝은 예술에 가까운 일

IIV. 알고리즘 총평

- **최근접 이웃** : 작은 데이터셋일 경우, 기본모델로서 좋고 설명하기 쉬움
- **선형 모델** : 첫 번쨰로 시도할 알고리즘, 대용량 데이터셋 가능, 고차원 데이터에 가능
- **나이브 베이즈** : 분류만 가능. 선형 모델보다 훨씬 빠름, 대용량 데이터셋과 고차원 데이터에 가능, 선형 모델보다 덜 정확
- **결정 트리** : 매우 빠름, 데이터 스케일 조정 필요 없음. 시각화하기 좋고 설명하기 쉬움
- **랜덤 포레스트** : 결정 트리 하나보다 거의 항상 좋은 성능을 냄. 매우 안정적이고 강력함. 데이터 스케일 조정 필요 없음. 고차원 희소 데이터에는 잘 안맞음.
- **그레이디언트 부스팅 결정 트리** : 랜덤 포레스트보다 조금 더 성능이 좋음. 랜덤 포레스트보다 학습은 느리나 예측은 빠르고 메모리를 조금 사용. 랜덤 포레스트보다 매개변수 튜닝이 많이 필요
- **서포트 벡터 머신** : 비슷한 의미의 특성으로 이뤄진 중간 규모 데이터셋에 잘 맞음. 데이터 스케일 조정 필요. 매개변수에 민감
- **신경망** : 특별히 대용량 데이터셋에서 매우 복잡한 모델을 만들 수 있음. 매개변수 선택과 데이터 스케일에 민감. 큰 모델은 학습이 오래 걸림.

IX. 새로운 데이터로 모델링할 때

1. 선형 모델이나 나이브 베이즈 또는 최근접 이웃 분류기 같은 간단한 모델로 시작해서 성능이 얼마나 나오는지 가늠해보기
2. 데이터를 충분히 이해한 다음 랜덤 포레스트나 그레이디언트 부스팅 결정트리, SVM, 신경망 같은 복잡한 모델을 만들 수 있는 알고리즘 고려