05

회귀

회귀

회귀는 여러 개의 독립변수와 한 개의 종속변수 간의 상관관계를 모델링하는 기법을 통칭합니다. 예를들어 아파트의 방 개수, 방 크기, 주변 학군 등 여러 개의 독립변수에 따라 아파트 가격이라는 종속변수가 어떤 관계를 나타내는지를 모델링하고 예측하는 것입니다.

$$Y = W_1 \times X_1 + W_2 \times X_2 + W_3 \times X_3 + \cdots + W_n \times X_n$$

(Y: 종속변수, X: 독립변수, W: 회귀 계수)

machine learning 관점에서 보면 독립변수는 feature 에 해당되며, 종속변수는 결정 값입니다. 이때 핵심은 주어진 feature 과 결정 값 데이터 기반 학습을 통해 최적의 회귀 계수를 찾아내는 것입니다.

지도학습은 두 가지 유형으로 나뉘는데, 바로 분류와 회귀입니다. 이들의 가장 큰 차이는 분류는 예측값이 카테고리와 같은 이산형 클래스 값이고, 회귀는 연속형 숫자 값이라는 것입니다.

분류	회귀
category 이산값	숫자값(연속값)

회귀는 회귀 계수의 선형/비선형 여부, 독립변수의 개수, 종속변수의 개수에 따라 여러 가지 유형으로 나눌 수 있습니다.

독립변수 개수	회귀 계수의 결합
1 개: 단일 회귀	선형: 선형 회귀

여러 개: 다중 회귀 비선형: 비선형 회귀

여러가지 회귀 중에서 선형 회귀가 가장 많이 사용됩니다. 선형 회귀는 실제 값과 예측값의 차이(오류의 제곱 값)를 최소화하는 직선형 회귀선을 최적화하는 방식입니다.

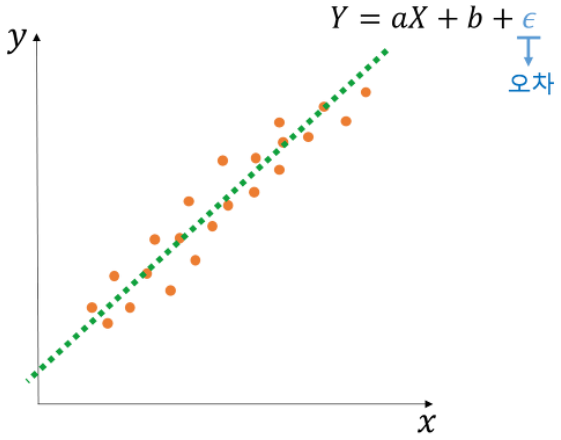
선형 회귀

선형 회귀 모델은 regularization 방법에 따라 다시 별도의 유형으로 나뉩니다. regularization 은 선형 회귀의 overfitting 문제를 해결하기 위해 회귀 계수에 페널티 값을 적용하는 것을 말합니다.

- 선형 회귀 모델 종류
  - 일반 선형 회귀: regularization 을 적용하지 않은 모델
  - Ridge: L2 regularization 을 추가한 모델, L2 regularization 은 상대적으로 큰 회귀 계수 값의 예측 영향도를 감소시키기 위해서 회귀 계수값을 더 작게 만듦.
  - Lasso: L1 regularization 을 적용한 모델. 예측 영향력이 작은 회귀 계수를 0 으로 만들어 회귀 예측 시 feature 가 선택되지 않게 하는 것.
  - ElasticNet: L2, L1 regularization 을 함께 결합한 모델. 주로 feature 개수가 많은 데이터셋에 적용
  - Logistic Regression: 강력한 분류 알고리즘. 일반적을 이진 분류 뿐만 아니라 텍스트 분류 영역에서도 뛰어난 예측 성능을 보임.

단순 선형 회귀를 통한 회귀 이해

단순 선형 회귀는 독립변수도 하나, 종속변수도 하나인 선형회귀입니다. 예를 들어, 주택 가격이 주택의 크기로만 결정된다고 합시다.



이때 기울기 w1 과 절편 w0 을 회귀 계수로 지칭하고, 회귀 모델을 Y = w0 + w1*X 로 모델링할 수 있습니다. 이때 실제 주택 가격은 이러한 회귀 모델 값에서 실제 값만큼의 오류를 더한 값이 됩니다.(w0+w1*X+오류 값)

이렇게 실제 값과 회귀 모델의 차이에 따른 오류 값을 잔차라고 부릅니다. 최적의 회귀 모델을 만든다는 것은 바로 전체 데이터의 잔차 합이 최소가 되는 모델을 만든다는 의미이며, 동시에 오류 값 합이 최소가 될 수 있는 최적의 회귀 계수를 찾는다는 의미도 됩니다.

오류 값은 +나 -가 될 수 있습니다. 그래서 전체 데이터의 오류 합을 구하기 위해서 단순히 더하는 것은 좋은 선택이 될 수 없습니다. 따라서 보통 오류 합을 계산할 때에는 절댓값을 취해서 더하거나(Mean Absolute Error), 오류 값의 제곱을 구해서 더하는 방식(RSS, Residual Sum of Square)을 취합니다. 일반적으로 RSS 으로 오류 합을 구합니다. RSS 를 최소로하는 회귀 계수 w_0와 w_1, 즉 회귀 계수를 학습을 통해서 찾는 것이 머신러닝 기반 회귀의 핵심 사항입니다.

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

회귀에서 이 RSS 는 비용(cost)이며, w 변수(회귀 계수)로 구성되는 RSS 를 비용 함수 혹은 손실함수라고 합니다. machine learning 회귀 알고리즘은 데이터를 계속 학습하면서 이 비용 함수가 반환하는 값(오류값)을 지속해서 감소시키고 최종적으로는 더 이상 감소하지 않는 최소의 오류 값을 구하는 것입니다.

비용 최소화 – 경사하강법 (Gradient Descent)

그렇다면 어떻게 비용함수가 최소가 되는 w parameter 를 구할 수 있을까요? 경사하강법은 RSS 를 최소화하는 방법을 직관적으로 제공하는 방식입니다. 점진적인(gradient) 하강 이라는 뜻에서도 알 수 있듯이, 점진적으로 w parameter 를 업데이트하면서 오류 값이 최소가 되도록 합니다.

경사하강법은 반복적으로 비용함수의 반환 값, 즉 예측 값과 실제 값의 차이가 작아지는 방향성을 가지고 w parameter 를 지속해서 보정해 나갑니다. 경사하강법의 핵심은 '어떻게 하면 오류가 작아지는 방향을 찾을 수 있을까'입니다. 앞서 언급한 $RSS(w_0, w_1)$를 편의상 R(w)라고 칭하겠습니다. 이때 R(w)를 미분해서 미분 함수의 최솟값을 구해야하는데, R(w)는 두 개의 parameter 를 가지고 있기 때문에 w_1, w_0 으로 순차적으로 편미분을 수행해야합니다. 각각 편미분을 수행한 결과값인 $\left(\frac{R(w)}{w_1}\right)' = -\frac{2}{N}\sum_{i=1}^{N} x_i * (\textit{실제값} - \textit{예측값})$, $\left(\frac{R(w)}{w_0}\right)' = -\frac{2}{N}\sum_{i=1}^{N}(\textit{실제값} - \textit{예측값})$ 을 반복적으로 보정하며 w_0, w_1 값을 업데이트하면 R(w)가 최소가 되는 w parameter 를 구할 수 있습니다. 실제로 위 편미분값이 너무 커지는 문제점을 보정하기 위해 학습률 η 을 반영하여 비용함수가 최소가 되는 값을 찾습니다. 따라서 $\textit{새로운 } w_1 = \textit{이전 } w_1 - \eta\frac{2}{N}\sum_{i=1}^{N} x_i * (\textit{실제값} - \textit{예측값})$, $\textit{새로운 } w_0 = \textit{이전 } w_0 - \eta\frac{2}{N}\sum_{i=1}^{N}(\textit{실제값} - \textit{예측값})$ 입니다. 정리하자면, 경사하강법의 일반적인 프로세스는 아래와 같습니다.

1) w_0, w_1를 임의로 설정하고, 첫 비용함수의 값을 계산합니다.
2) $\textit{새로운 } w_1 = \textit{이전 } w_1 - \eta\frac{2}{N}\sum_{i=1}^{N} x_i * (\textit{실제값} - \textit{예측값})$, $\textit{새로운 } w_0 = \textit{이전 } w_0 - \eta\frac{2}{N}\sum_{i=1}^{N}(\textit{실제값} - \textit{예측값})$으로 업데이트한 후 다시 비용함수를 계산합니다.
3) 비용함수의 값이 감소했으면 다시 2)를 반복하고, 더 이상 감소하지 않으면 반복을 중지합니다.

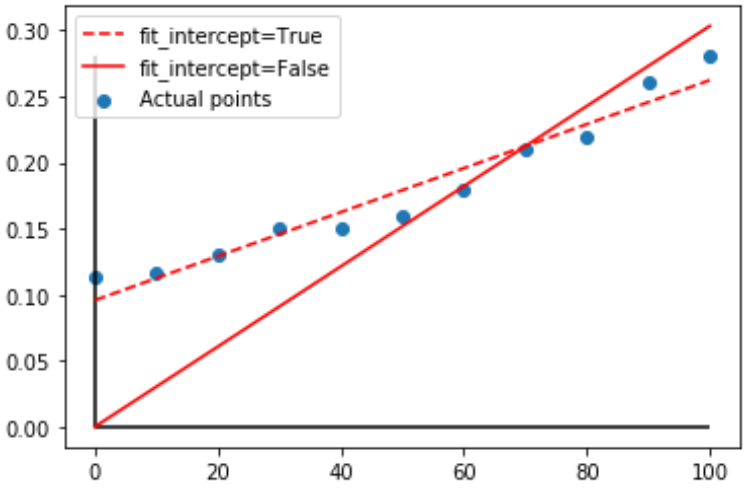
일반적으로 경사하강법은 모든 학습 데이터에 대해 반복적으로 비용함수 최소화를 위한 값을 업데이트하기 때문에 수행 시간이 매우 오래 걸린다는 단점이 있습니다. 그 때문에 실전에서는 대부분 확률적 경사 하강법(Stochastic Gradient Descent)를 이용합니다. 확률적 경사 하강법은 전체 입력 데이터가 아니라 일부 데이터만 이용해 (mini batch) w 가 업데이트되는 값을 계산하므로 빠른 속도를 보장합니다.

Scikit-learn LinearRegression

scikit-learn 은 regularization 이 적용되지 않은 일반 선형 회귀를 LinearRegression 클래스를 통해 구현하였습니다. LinearRegression 클래스는 아래와 같이 정의됩니다.

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

- 입력 parameter
  - fit_intercept: boolean (deafult= True). 절편(intercept) 값을 계산할 것인지 말지를 지정.



  - normalize: boolean (deafult= False). 입력셋을 정규화 할 것인지를 결정.

LinearRegression 클래스는 fit( ) method 로 X, y 배열을 입력받으면 회귀 계수(coefficient)인 w 를 coef_ 속성에 저장합니다. 이때 coef_의 shape 은 (target 값 개수, feature 개수)입니다. 회귀 계수 계산은 feature 사이의 상관관계가 매우 높은 경우 분산이 커져서 오류에 매우 민감해집니다. 이러한 현상을 multi-collinearity 문제라고 하고, 독립적인 중요한 feature 만 남기고 제거하거나 regularization 을 적용합니다. 또한 PCA 를 통해 차원 축소를 수행하는 것도 고려해볼 수 있습니다.

회귀 평가 지표

일반적으로 회귀의 성능을 평가하는 지표는 아래와 같습니다.

- MAE: Mean Absolute Error (MAE), 실제 값과 예측 값의 차이를 절댓값화 평균.
- MSE: Mean Squared Error (MSE), 실제 값과 예측 값의 차이를 제곱해 평균.
- RMSE: MSE 에 루트를 씌운 것.
- R^2: $\frac{예측값\ Variance}{실제값 Variance}$

LinearRegression 을 이용한 회귀 구현

scikit-learn 에 내장된 데이터셋인 보스턴 주택 가격 데이터를 이용하여 주택 가격을 예측하는 선형 회귀 모델을 만들어보겠습니다. 해당 데이터셋을 로드하고 DataFrame 으로 변경하겠습니다.

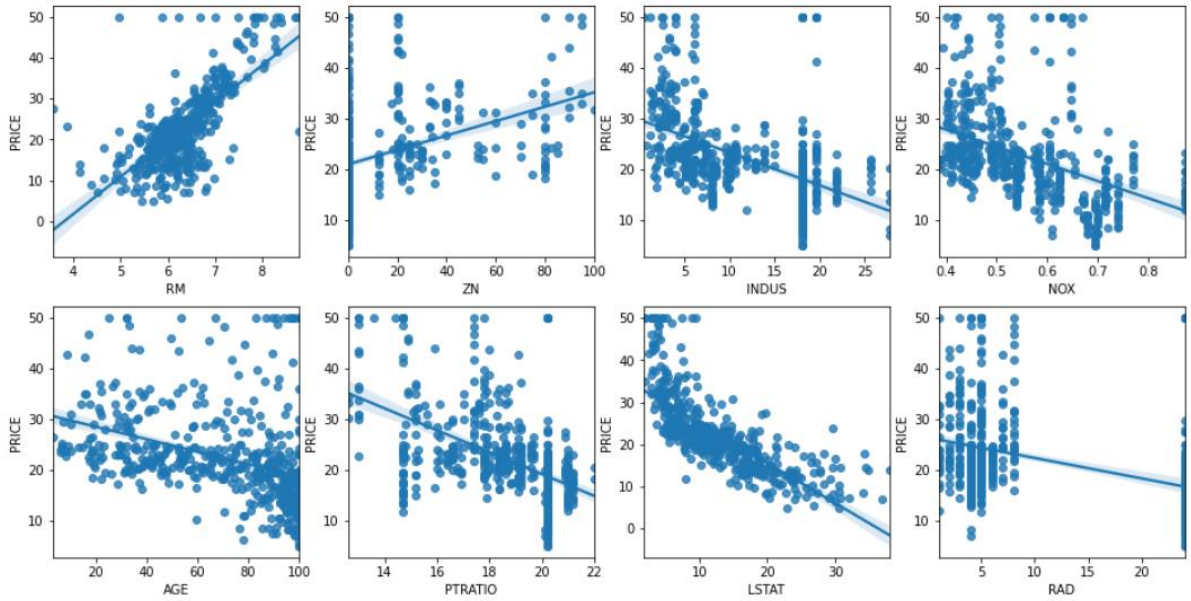
```
from sklearn.datasets import load_boston
%matplotlib inline

# boston 데이타셋 로드
boston = load_boston()

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

# boston dataset의 target array는 주택 가격임. 이를 PRICE 컬럼으로 DataFrame에 추가함.
bostonDF['PRICE'] = boston.target
print('Boston 데이타셋 크기 :',bostonDF.shape)
bostonDF.head()
```

Boston 데이타셋 크기 : (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

다음으로 각 column 이 회귀 결과에 미치는 영향이 어느 정도인지 시각화해서 알아보겠습니다.

```
# 2개의 행과 4개의 열을 가진 subplots를 이용. axs는 4x2개의 ax를 가짐.
fig, axs = 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
    # 시본의 regplot을 이용해 산점도와 선형 회귀 직선을 함께 표현
    sns.regplot(x=feature , y='PRICE',data=bostonDF , ax=axs[row][col])
```



다른 column 보다 RM 과 LSTAT 의 영향도가 가장 크게 두드러지게 나타납니다. 이제, LinearRegression 클래스를 이용해 보스턴 주택 가격의 회귀 모델을 만들어보겠습니다. train_test_split( )을 이용해 학습과 테스트 데이터셋을 분리해 학습과 예측을 수행합니다. 그리고 metics 모듈의 mean_squared_error( )와 r2_score( )를 이용해 MSE 와 R2 Score 를 측정합니다.

```
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 ,t
est_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 : 17.306 , RMSE : 4.160
Variance score : 0.757
```

LinearRegression 으로 생성한 주택 가격모델의 intercept_(절편)과 coef_(회귀 계수)를 보겠습니다.

```
절편 값: 40.995595172164755
```

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

다음으로 최적화 작업을 진행해보도록 하겠습니다. 본래 scikit-learn 의 지표 평가 기준은 높은 지표 값일수록 좋은 모델인 데 비해, 회귀는 MSE 값이 낮을수록 좋은 회귀 모델입니다. scikit-

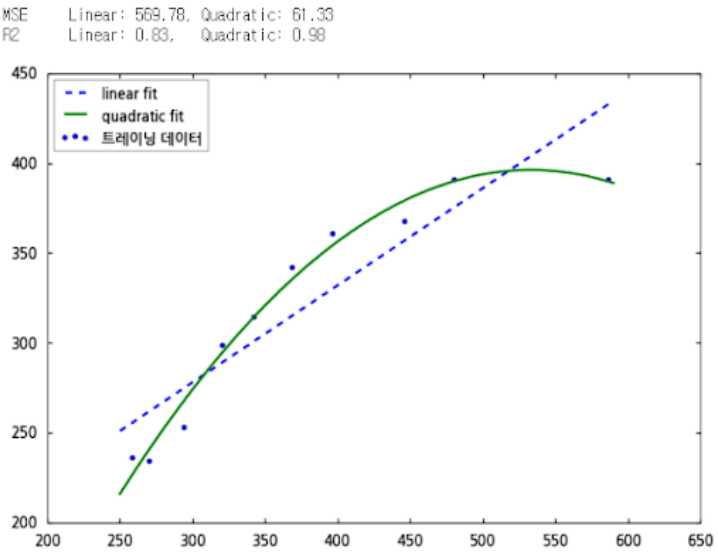
learn 에 이를 반영하기 위해 scoring = 'neg_mean_squred_error'로 모델에서 계산된 MSE 값에 -1 을 곱해서 반환합니다.

```
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)
```

5 개 fold 세트에 대해서 교차검증을 수행한 결과, 평균 RMSE 는 약 5.836 으로 나왔습니다.

다항 회귀

선형회귀는 $Y = W_1 \times X_1 + W_2 \times X_2 + W_3 \times X_3 + \cdots + W_n \times X_n$ 과 같이 독립변수(feature)와 종속변수(target)의 관계가 일차 방정식 형태로 표현된 회귀였습니다. 회귀가 독립변수의 단항식이 아닌 2 차 이상의 다항식으로 표현되어있는 것을 다항(polynomial) 회귀라고 합니다. 다항 회귀는 $Y = W_1 \times X_1 + W_2 \times X_2 + W_3 \times X_1 \times X_2 + W_4 \times X_1 \times X_1 + W_5 \times X_2 \times X_2$ 과 같이 표현될 수 있습니다. 한 가지 주의할 것은 다항 회귀를 비선형 회귀로 혼동하기 쉽지만, 다항 회귀는 선형회귀라는 점입니다. 회귀에서 선형/비선형을 나누는 기준은 회귀 계수(W)가 선형/비선형인지에 따른 것이기 때문입니다. $Y = W_1 \times X_1 + W_2 \times X_2 + W_3 \times X_1 \times X_2 + W_4 \times X_1 \times X_1 + W_5 \times X_2 \times X_2$ 에서 $Z = [X_1, X_2, X_1 * X_2, {X_1}^2, {X_2}^2]$ 로 한다면 $Y = W_1 \times Z_1 + W_2 \times Z_2 + W_3 \times Z_3 + \cdots + W_n \times Z_n$ 로 표현할 수 있기에 여전히 선형회귀입니다. 아래 그림을 보면 단순 선형 회귀 직선형으로 표현한 것보다 다항 회귀 곡선형으로 표현한 것이 더 예측 성능이 높습니다.



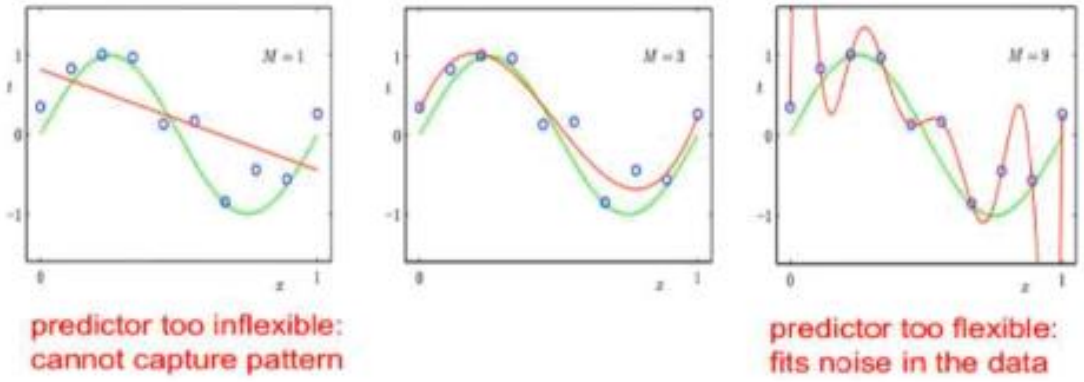
아쉽지만 scikit-learn 은 다항회귀를 위한 클래스를 명시적으로 제공하지는 않습니다. 하지만 PolynomialFeatures 클래스를 통해 feature 를 polynomial feature 로 변환하여 선형회귀로 접근할 수 있습니다. PolynomialFeatures 클래스는 degree parameter 를 통해 입력 받은 단항식 feature 를 degree 에 해당하는 다항식 feature 로 변환합니다. 일차 단항식 계수를 삼차 다항식 계수로 변환하고, 이를 선형회귀에 적용하면 다항 회귀로 구현됩니다.

```
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

X = np.arange(4).reshape((2,2))
y = polynomial_func(X)
poly_ftr = PolynomialFeatures(degree=3).fit_transform(X)
model.fit(poly_ftr, y)
```

Underfitting 과 Overfitting 의 이해

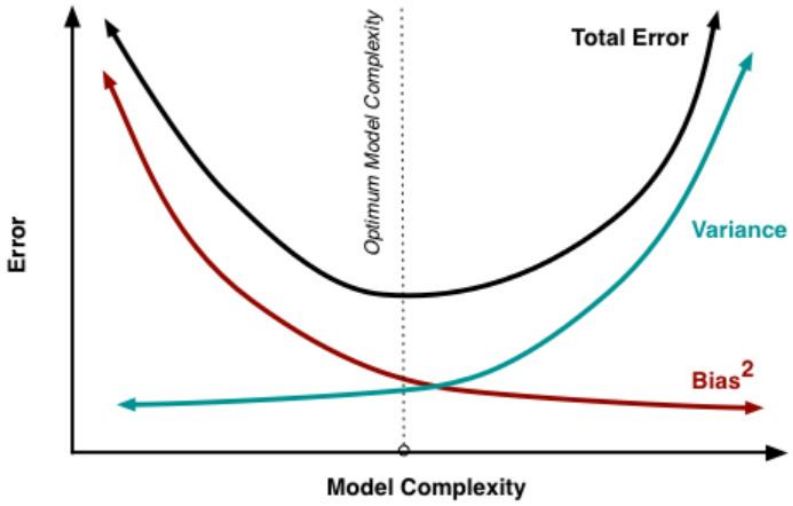
다항 회귀는 feature 의 직선적 관계가 아닌 복잡한 다항 관계를 모델링할 수 있습니다. 다항식의 차수가 높아질수록 매우 복잡한 feature 과의 관계까지도 모델링이 가능하나, 차수를

높일수록 학습데이터에만 너무 맞춘 학습이 이뤄져서 테스트셋에 대한 예측 정확도가 오히려 떨어집니다. 즉, 차수가 높아질수록 overfitting 의 문제가 크게 발생합니다.



Bias-Variance Trade off

앞의 degree = 1 과 같은 모델은 매우 단순화된 모델로서 high bias 를 가졌다고 표현합니다. 만대로 degree=9 와 같은 모델은 매우 복잡하게 표현되었고 high variance 를 가졌다고 표현합니다. 일반적으로 bias 와 variance 는 한 쪽이 높으면 한 쪽이 낮아지는 경향이 있습니다. 즉, bias 가 높으면 variance 가 낮아지고(underfitting) 반대로 variance 가 높으면 편향이 낮아집니다(overfitting). bias 와 variance 가 서로 tradeoff 를 이루면서 오류 cost 값이 최대로 낮아지는 모델을 구축하는 것이 필요합니다.



Regularization 선형 모델 – Ridge, Lasso, ElasticNet

좋은 machine learning 회귀 모델이 되기 위해서는 적절히 데이터에 적합하면서도 회귀 계수가 기하급수적으로 커지는 것을 제어할 수 있어야합니다. 이전까지 선형 모델 비용함수는 RSS 를 최소화하는 것만 고려했습니다. 그러다 보니 학습 데이터에 지나치게 맞추어지게 되고, 회귀 계수가 쉽게 커졌습니다. 이를 반영해 비용 함수의 목표가 RSS(W) + alpha * $||W||_2^2$ 를 최소화하는 것으로 변경될 수 있습니다. 여기서 alpha는 학습데이터 적합 정도와 회귀 계수 값의 크기 제어를 수행하는 tuning parmeter 입니다. alpha 가 매우 작은 값이라면 비용함수 식은 기존과 동일한 Min(RSS(W)))이 될 것입니다. 반면 alpha 가 매우 큰 값이라면 비용함수 식은 alpha * $||W||_2^2$가 너무 커지게 되므로 W 를 매우 작게 만들어야 cost 가 최소화되는 비용함수 목표를 달성할 수 있습니다. 즉, alpha가 커지면 회귀 계수 W를 작게 하여 overfitting을 개선할 수 있으며, alpha 가 작으면 회귀계수 W 가 커져도 어느정도 상쇄가 가능하므로 학습데이터 적합을 더 개선할 수 있습니다.

이처럼 비용 함수에 alpha 값으로 페털티를 부여해 회귀 계수의 크기를 감소시켜 overfitting 을 개선하는 방식을 regularization 이라고 부릅니다. 이는 크게 L2 방식과 L1 방식으로 분류됩니다. L2 regularization 은 $||W||_2^2$에 대해 페널티를 부여하는 방식을 말하며, 이를 적용한 회귀를 Ridge라고 합니다. L1 regularization은 $||W||_1$, 즉 W의 절댓값에 대해 페널티를 부여합니다. L1 regularization 을 이용하면 영향력이 크지 않은 회귀 계수의 값을 0 으로 변환하고, 이를 적용한 회귀는 Lasso 입니다.

Ridge 회귀

scikit-learn 은 Ridge 클래스를 통해 Ridge 회귀를 구현합니다. Ridge 클래스의 주요 생성 parameter 는 allpha 이며, 이는 ridge 회귀의 L2 regularization 규제 계수에 해당합니다. 아래 예제에서는 alpha를 10으로 하여 ridge 회귀를 진행하였습니다.

```
# 앞의 LinearRegression예제에서 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

ridge = Ridge(alpha = 10)
```

이때 cross validation 결과는 평균 RMSE 5.524 로, regularization 이 없는 일반 선형 회귀 모델의 5.836 보다 뛰어난 수치에 해당합니다.

이번에는 ridge 의 alpha 값을 0, 0.1, 1, 10, 100 으로 변화시키면서 RMSE 와 회귀 계수의 값을 비교해보도록 하겠습니다.

```
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	3.804752	3.813177	3.849256	3.698132	2.331966
CHAS	2.688561	2.671849	2.554221	1.953452	0.638647
RAD	0.305655	0.303105	0.289650	0.279016	0.314915
ZN	0.046395	0.046546	0.047414	0.049547	0.054470
INDUS	0.020860	0.016293	-0.008547	-0.042745	-0.052626
B	0.009393	0.009449	0.009754	0.010117	0.009471
AGE	0.000751	-0.000212	-0.005368	-0.010674	0.001230
TAX	-0.012329	-0.012415	-0.012907	-0.013989	-0.015852
CRIM	-0.107171	-0.106612	-0.103622	-0.100352	-0.101451
LSTAT	-0.525467	-0.526678	-0.534072	-0.560097	-0.661312
PTRATIO	-0.953464	-0.941449	-0.876633	-0.798335	-0.829503
DIS	-1.475759	-1.459773	-1.372570	-1.248455	-1.153157
NOX	-17.795759	-16.711712	-10.793436	-2.374959	-0.263245

alpha 값이 증가하면서 회귀 계수가 지속적으로 작아지고 있음을 알 수 있습니다.

Lasso 회귀

W 의 절댓값에 페널티를 부여하는 L1 규제를 선형회귀에 적용한 것이 lasso 회귀입니다. L2 regularization 이 회귀 계수의 크기를 감소시키는 데 반해, L1 regularization 은 불필요한 회귀 계수를 0 으로 만들고 제거합니다.

scikit-learn 은 Lasso 클래스를 통해 lasso 회귀를 구현합니다. Lasso 모델을 생성하고, alpha 값을 조정하여 변화를 비교해보겠습니다.

```
from sklearn.linear_model import Lasso, ElasticNet

# alpha값에 따른 회귀 모델의 폴드 평균 RMSE를 출력하고 회귀 계수값들을 DataFrame으로 반환
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
```

```
# 라쏘에 사용될 alpha 파라미터의 값들을 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)
```

```
#######  Lasso #######
alpha 0.07일 때 5 폴드 세트의 평균 RMSE: 5.618
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.621
alpha 0.5일 때 5 폴드 세트의 평균 RMSE: 5.672
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.776
alpha 3일 때 5 폴드 세트의 평균 RMSE: 6.189
```

```
# 반환된 coeff_lasso_df를 첫번째 컬럼순으로 내림차순 정렬하여 회귀계수 DataFrame출력
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	3.785460	3.698943	2.494509	0.946786	0.000000
CHAS	1.436287	0.957097	0.000000	0.000000	0.000000
RAD	0.270327	0.274112	0.277118	0.264175	0.061867
ZN	0.049026	0.049179	0.049528	0.049169	0.037231
B	0.010326	0.010327	0.009532	0.008291	0.006510
NOX	-0.000000	-0.000000	-0.000000	-0.000000	0.000000
AGE	-0.011675	-0.010006	0.003630	0.020927	0.042495
TAX	-0.014287	-0.014567	-0.015440	-0.015209	-0.008602
INDUS	-0.041924	-0.036425	-0.005109	-0.000000	-0.000000
CRIM	-0.097061	-0.096788	-0.082662	-0.063423	-0.000000
LSTAT	-0.561179	-0.569509	-0.656853	-0.761433	-0.807679
PTRATIO	-0.765456	-0.771003	-0.759070	-0.723199	-0.265072
DIS	-1.176150	-1.160121	-0.936447	-0.669009	-0.000000

alpha 가 0.07 일 때 가장 좋은 평균 RMSE 를 보여줍니다. alpha 의 크기가 증가함에 따라 일부 feature 의 회귀 계수는 아예 0 으로 바뀌고 있습니다.

ElasticNet 회귀

ElasticNet 회귀는 L2 regularization 과 L1 regularization 을 결합한 회귀입니다. ElasticNet 은 중요 feature 만 셀렉션하고 나머지 feature 들의 회귀 계수를 모두 0 으로 만드는 성향이 강합니다. 이러한 성향으로 인해 alpha 값에 따라 회귀 계수의 값이 급격히 변동할 수 있는데, elasticNet 은 이를 완화하기 위해 L2 regularization 를 추가한 것입니다.

scikit-learn 은 ElasticNet 클래스를 통해 elasticNet 을 구현합니다. ElasticNet 의 주요 생성 parameter 는 alpha 와 l1_ratio 입니다. ElasticNet 의 alpha 는 이전과는 조금 다릅니다. ElasticNet 의 regularization 은 a*L1 + b*L2 로 정의될 수있고, 따라서 alpha parameter 값은 a+b 입니다. l1_ratio parameter 값은 a / (a + b)입니다. l1_ratio 가 0 이면 a 가 0 이므로 L2 regularization 과 동일하며 1 이면 b 가 0 이므로 L1 regularization 과 동일합니다.

학습을 시키게 되면,

```
model = ElasticNet(alpha=param, l1_ratio=0.7)

model.fit(X_data , y_target)
```

	alpha:0.07	alpha:0.1	alpha:0.5	alpha:1	alpha:3
RM	3.570126	3.410274	1.915894	0.937179	0.000000
CHAS	1.332117	0.980900	0.000000	0.000000	0.000000
RAD	0.278304	0.282881	0.300449	0.289167	0.147089
ZN	0.050074	0.050586	0.052860	0.052126	0.038281
B	0.010200	0.010145	0.009182	0.008373	0.007029
AGE	-0.010084	-0.008248	0.007777	0.020360	0.043445
TAX	-0.014519	-0.014810	-0.016044	-0.016213	-0.011417
INDUS	-0.044641	-0.042520	-0.023093	-0.000000	-0.000000
CRIM	-0.098392	-0.098179	-0.088550	-0.073471	-0.019596
NOX	-0.178164	-0.000000	-0.000000	-0.000000	-0.000000
LSTAT	-0.575540	-0.588404	-0.694327	-0.760714	-0.800276
PTRATIO	-0.779878	-0.785066	-0.791232	-0.738850	-0.423093
DIS	-1.189080	-1.173268	-0.975771	-0.725297	-0.031389

alpha 값에 따른 feature 들의 회귀 계수 값이 lasso 보다는 상대적으로 0 으로 되는 값이 적음을 볼 수 있습니다.

지금까지 regularization 선형 회귀의 가장 대표적인 기법인 ridge, lasso, elasticNet 회귀에 대해 살펴보았습니다. 이들 중 어떤 것이 가장 좋은지는 상황에 따라 다릅니다. 각각의 알고리즘에서 hyper-parameter 를 변경해가면서 최적의 예측 성능을 찾아내야합니다. 하지만 선형 회귀의 경우 먼저 데이터 분포도의 정규화와 인코딩 방법이 매우 중요합니다.

선형 회귀 모델을 위한 데이터 변환

scikit-learn을 이용해 데이터셋에 적용하는 변환 작업은 다음과 같은 방법이 있을 수 있습니다.

1) StandardScaler 클래스를 이용해 평균이 0, 분산이 1 인 표준 정규 분포를 가진 데이터셋으로 변환하거나, MinMaxScaler 클래스를 이용해 최솟값이 0 이고 최댓값이 1 인 값으로 정규화를 수행합니다.

2) 1)을 통해 예측 성능 향상이 없을 경우 다시 다항 특성을 적용하여 변환합니다.

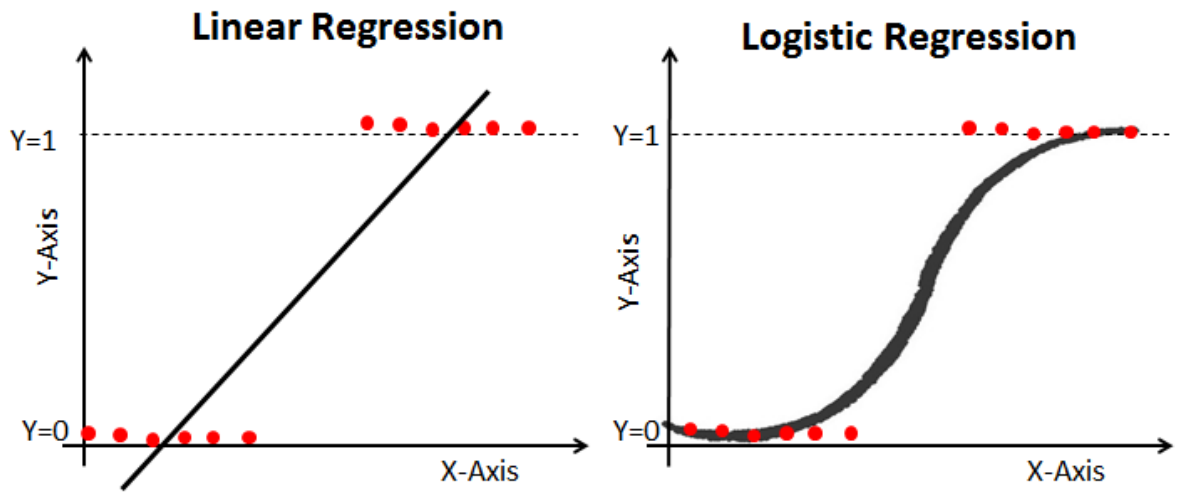
3) 원래 값에 log 함수를 적용하면 보다 정규분포에 가까운 형태로 값이 분포됩니다. 이러한 변환을 log transform(로그변환)이라고 부릅니다. 보통 1),2)보다 훨씬 많이 사용되는 변환 방법입니다.

```
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
```

```
## 변환 유형:None, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.796
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.659
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.524
alpha 100일 때 5 폴드 세트의 평균 RMSE: 5.332

## 변환 유형:Standard, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.834
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.810
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.643
alpha 100일 때 5 폴드 세트의 평균 RMSE: 5.424

## 변환 유형:Standard, Polynomial Degree:2
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 8.776
alpha 1일 때 5 폴드 세트의 평균 RMSE: 6.849
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.487
alpha 100일 때 5 폴드 세트의 평균 RMSE: 4.631

## 변환 유형:MinMax, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.770
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.468
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.755
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: 5.294
alpha 1일 때 5 폴드 세트의 평균 RMSE: 4.320
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.186
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: 4.772
alpha 1일 때 5 폴드 세트의 평균 RMSE: 4.676
alpha 10일 때 5 폴드 세트의 평균 RMSE: 4.835
alpha 100일 때 5 폴드 세트의 평균 RMSE: 6.244
```

로지스틱 회귀

로지스틱 회귀는 선형 회귀 방식을 분류에 적용한 알고리즘입니다. 선형회귀와 다른 점은 학습을 통해 선형 함수의 회귀 최적선을 찾는 것이 아니라 sigmoid 함수 최적선을 찾고 이 sigmoid 함수의 반환 값을 확률로 간주해 확률에 따라 분류를 결정한다는 것입니다.



많은 자연, 사회 현상에서 특정 변수의 확률 값은 선형이 아니라 sigmoid 와 같이 s 자 커브 형태를 가집니다. 위의 그림에서 왼쪽 직선 회귀 라인은 0 과 1 을 제대로 분류해내지 못하고 있습니다. 하지만 s 자 커브 형태의 sigmoid 함수를 이용하면 좀 더 정확하게 분류를 할 수 있게 됩니다.

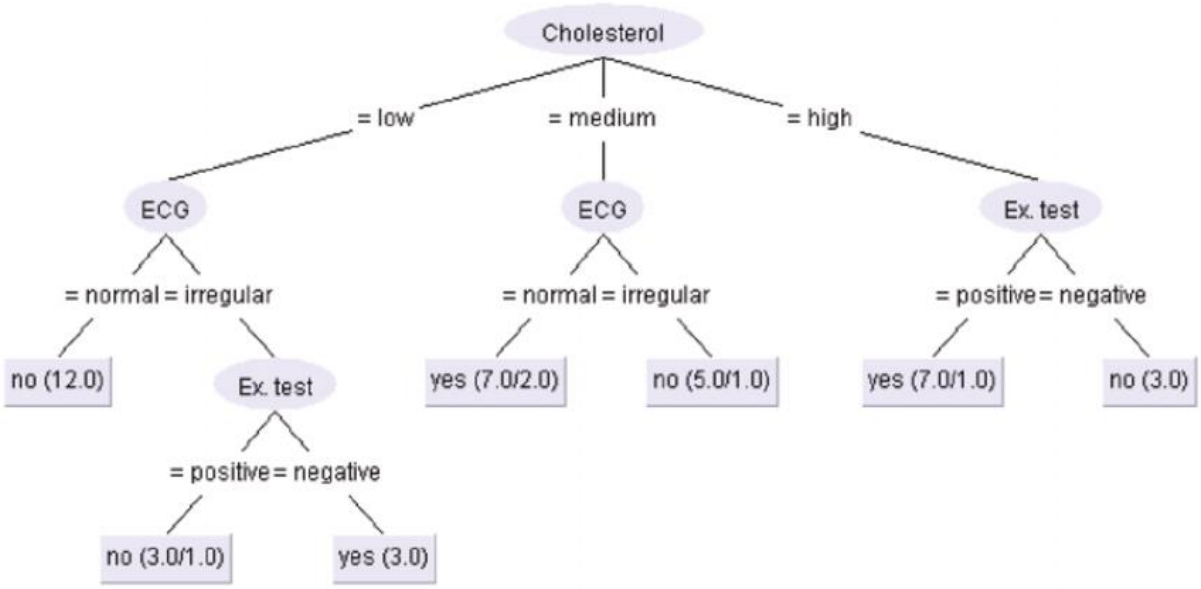
scikit-learn 은 로지스틱 회귀를 위해 LogisticRegression 클래스를 제공하고 있습니다. 주요 hyper-parameter 로는 penalty 와 C 가 있습니다. Penalty 는 regularization 의 유형(l1, l2)을 설정하고, C 는 regularization 강도를 조절하는 alpha 값의 역수로, C 가 작을수록 regularizion 의 강도가 큽니다.

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, roc_auc_score

# 로지스틱 회귀를 이용하여 학습 및 예측 수행.
lr_clf = LogisticRegression(penalty='l2', C=1)
lr_clf.fit(X_train, y_train)
lr_preds = lr_clf.predict(X_test)

# accuracy와 roc_auc 측정
print('accuracy: {:0.3f}'.format(accuracy_score(y_test, lr_preds)))
print('roc_auc: {:0.3f}'.format(roc_auc_score(y_test , lr_preds)))
```

```
accuracy: 0.977
roc_auc: 0.972
```

회귀 tree

machine learning 기반 회귀는 회귀 계수를 기반으로 최적 회귀 함수를 도출하는 것이 주요 목표입니다. 회귀 tree 는 회귀 함수를 기반으로 하지 않고, decision tree 와 같이 tree 를 기반으로 하는 회귀 방식을 뜻합니다. decision tree 와 다른 점은 decision tree 가 특정 클래스 label 을 결정하는 것과는 달리 회귀 tree 는 leaf node 가 속한 데이터 값의 평균을 구해 회귀 예측 값을 계산합니다.



위와 같은 방식으로, leaf node 생성 기준에 부합하는 tree 분할이 완료됐다면 leaf node 에 소속된 데이터 값의 평균값을 구해서 최종적으로 leaf node 에 결정 값으로 할당합니다.

decition tree, random forest, GBM, XGBoost, LightGBM 등 앞서 소개한 모든 tree 기반의 알고리즘은 분류 뿐만 아니라 회귀도 가능합니다. scikit-learn 은 tree 기반 알고리즘에서 회귀 수행을 할 수 있는 Estimator 클래스를 제공합니다.

회귀 Estimator 클래스	분류 Estimator 클래스
DecisionTreeRegressor	DecisionTreeClassifier
GradientBoostingRegressor	GradientBoostingClassifier
XGBRegressor	XGBClassifier

LGBMRegressor	LGBMClassifier
RandomForestRegressor	RandomForestClassifier

예를 들어, RandomForestRegressor 를 이용해 주택 가격 예측을 수행하는 과정은 아래와 같습니다.

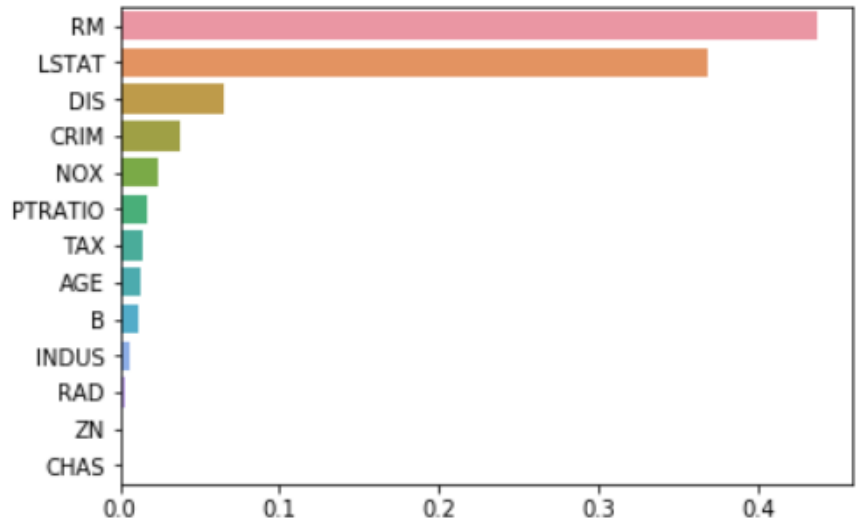
```
from sklearn.datasets import load_boston
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor
import pandas as pd
import numpy as np

# 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 : {0:.3f} '.format(avg_rmse))
```

```
 5 교차 검증의 개별 Negative MSE scores:  [ -7.92 -13.03 -20.51 -46.35 -18.95]
 5 교차 검증의 개별 RMSE scores :  [2.81 3.61 4.53 6.81 4.35]
 5 교차 검증의 평균 RMSE : 4.423
```

또한 feature_importances_를 이용해 feature 별 중요도를 알 수 있습니다.

```
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.c
olumns )
feature_series = feature_series.sort_values(ascending=False)
sns.barplot(x= feature_series, y=feature_series.index)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x27d4b724780>
```



이번에는 회귀 tree regressor 가 어떻게 예측값을 판단하는지를 선형 회귀와 비교해 설명해 보겠습니다. 다음은 DecisionTreeRegressor 과 LinearRegression 을 테스트한 결과입니다.

선형 회귀는 직선으로 예측 회귀선을 표현하는 데 반해, 회귀 tree 의 경우 분할되는 데이터 지점에 따라 브랜치를 만들면서 계단 형태로 회귀선을 만듭니다. DecisionTreeRegressor 의

max_depth=7 인 경우에는 이상치(outlier)도 학습하면서 복잡한 계단 형태의 회귀선을 만들어 overfitting 되기 쉬운 모델이 되었음을 알 수 있습니다.

회귀 실습 - 자전거 대여 수요 예측

https://www.kaggle.com/c/bike-sharing-demand/data

회귀 실습 - 캐글 주택 가격: 고급 회귀 기법

http://www.kaggle.com/c/house-prices-advanced-regression-techniques/data