빅데이터분석 실습

데이터 표현과 피처 공학

데이터 사이언스 전공

담당교수: 곽철완

강의 내용

- 범주형 변수
- OneHotEncoder와 ColumnTransformer
- make_column_transformer
- 구간 분할, 이산화 그리고 선형 모델, 트리 모델
- 상호작용과 다항식
- 피처 자동 선택

▪ 데이터 표현과 피처 공학 필요성

- 일반적인 피처의 형태
  - 범주형 categorical feature or 이산형 discrete feature
  - 범주형 피처와 연속적인 피처의 차이는 데이터 입력에 있다
    - 범주형 피처 예: 제품 브랜드, 색상, 제품 이름
    - 연속적인 피처 예: 픽셀의 밝기, iris 꽃 크기
- 머신러닝에 영향
  - 데이터 스케일에 따라 머신러닝 성능에 큰 영향을 미친다
  - 피처 공학(특정 알고리즘에 가장 적합한 데이터 표현을 찾는 것)을 통해 알고리즘 성능 향상을 위한 피처 찾기가 필요하다

1. 범주형 변수

▪ one-hot-encoding

- one-out-of-N encoding 혹은 가변수(dummy variable)라고도 한다
- 범주형 변수를 0 혹은 1 값을 가진 하나 이상의 새로운 피처로 바꾼 것이다
- 예) 직업 범주에 '공무원', '회사원', '자영업', '소규모기업인'이 있을 때, 코딩을 위해 이중 하나의 피처는 1이 되고 나머지는 0이 된다
- pandas나 scikit-learn을 이용하여 범주형 변수를 원-핫-인코딩으로 바꿀 수 있다

```
import os

data = pd.read_csv(
    os.path.join(mglearn.datasets.DATA_PATH, "adult.data"), header=None, index_col=False,
    names=['age', 'workclass', 'fnlwgt', 'education',  'education-num',  'marital-status', 'occupation', 'relationship',
           'race', 'gender', 'capital-gain', 'capital-loss', 'hours-per-week', 'native-country', 'income'])

data = data[['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation', 'income']]

display(data.head())
```

	age	workclass	education	gender	hours-per-week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K

- pandas 예
  - 예제를 위해 7개 열만 선택
  - Ipython.display 함수는 포맷된 출력을 display 한다

- 범주형 데이터 문자열(gender 문자열) 확인

```
print(data.gender.value_counts())
```

```
Male      21790
Female    10771
Name: gender, dtype: int64
```

  - 열 내용 확인을 위해, pandas 의 Series에 있는 value_counts 메서드를 사용하여 고유한 값이 몇 번 출현하는지 파악한다
  - gender 에 Male과 Female 두가지로 구분되어 원-핫-인코딩 하기에 좋은 형태이다
  - pandas에서는 get_dummies 함수를 사용해 데이터를 쉽게 인코딩할 수 있다
    - 문자열이나 범주형을 가진 열을 자동으로 변환해준다

```
print("원본 특성:\n", list(data.columns), "\n")
data_dummies = pd.get_dummies(data)
print("get_dummies 후의 특성:\n", list(data_dummies.columns))
```

```
원본 특성:
 ['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation', 'income']

get_dummies 후의 특성:
 ['age', 'hours-per-week', 'workclass_ ?', 'workclass_ Federal-gov', 'workclass_ Local-go
e', 'workclass_ Self-emp-inc', 'workclass_ Self-emp-not-inc', 'workclass_ State-gov', 'wo
ion_ 11th', 'education_ 12th', 'education_ 1st-4th', 'education_ 5th-6th', 'education_ 7t
m', 'education_ Assoc-voc', 'education_ Bachelors', 'education_ Doctorate', 'education_ H
ool', 'education_ Prof-school', 'education_ Some-college', 'gender_ Female', 'gender_ Mal
'occupation_ Armed-Forces', 'occupation_ Craft-repair', 'occupation_ Exec-managerial', 'o
```

- age와 hours-per-week를 제외하고는 범주형 피처 값이 새로운 피처로 확장되었다
  - 예, workclass → ‘workclass_?’, ‘workclass_Federal-gov’, …

```
display(data_dummies.head())
```

	age	hours-per-week	workclass_?	workclass_Federal-gov	workclass_Local-gov	workclass_Never-worked	workclass_Private	workclass_Self-emp-inc	workclass_Self-emp-not-inc	workclass_State-gov	...
0	39	40	0	0	0	0	0	0	0	1	...
1	50	13	0	0	0	0	0	0	1	0	...
2	38	40	0	0	0	0	1	0	0	0	...
3	53	40	0	0	0	0	1	0	0	0	...
4	28	40	0	0	0	0	1	0	0	0	...

5 rows × 46 columns

occupation_Machine-op-inspct	occupation_Other-service	occupation_Priv-house-serv	occupation_Prof-specialty	occupation_Protective-serv	occupation_Sales	occupation_Tech-support	occupation_Transport-moving	income_<=50K	income_>50K
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	1	0

- data_dummies의 values 속성을 이용해 데이터 프레임을 Numpy 배열로 바꿀 수 있으며, 이를 이용해 머신러닝 모델을 학습시킨다
- 모델을 학습시키지 전에 데이터로부터 타킷값(이 데이터 세트에서는 income_<=50K, income_>50K)을 분리해야 한다

```
features = data_dummies.loc[:, 'age':'occupation_ Transport-moving']
X = features.values
y = data_dummies['income_ >50K'].values
print("X.shape: {}  y.shape: {}".format(X.shape, y.shape))
```

```
X.shape: (32561, 44)  y.shape: (32561,)
```

- 피처를 age 부터 occupation_Transport-moving까지 추출했다

- 로지스틱 회귀분석 사용
  - 새로 인코딩한 데이터 세트를 로지스틱 회귀분석으로 테스트

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("테스트 점수: {:.2f}".format(logreg.score(X_test, y_test)))
```

```
테스트 점수: 0.81
```

■ 숫자로 표현된 범주형 피처

- adult 데이터 세트는 범주형 변수가 문자열로 코딩 되어 있다
  - 일반적으로 입력 오류 방지 혹은 공간 절약을 위해 사용되는 경우가 많다

```
demo_df = pd.DataFrame({'숫자 피처': [0, 1, 2, 1], '범주형 피처': ['양말', '여우', '양말', '상자']})
display(demo_df)
```

	숫자 피처	범주형 피처
0	0	양말
1	1	여우
2	2	양말
3	1	상자

◦ get_dummies 함수를 사용하면 문자열 피처만 인코딩 되며 숫자 피처는 바뀌지 않는다

```
display(pd.get_dummies(demo_df))
```

	숫자 피처	범주형 피처_상자	범주형 피처_양말	범주형 피처_여우
0	0	0	1	0
1	1	0	0	1
2	2	0	1	0
3	1	1	0	0

- 숫자 피처도 원-핫-인코딩(가변수)으로 만들고 싶다면 열 매개변수에 인코딩하고 싶은 열을 명시해야 한다
- 그러면 두 피처를 모두 범주형으로 간주한다

```
demo_df['숫자 피처'] = demo_df['숫자 피처'].astype(str)
display(pd.get_dummies(demo_df, columns=['숫자 피처', '범주형 피처']))
```

	숫자 피처_0	숫자 피처_1	숫자 피처_2	범주형 피처_상자	범주형 피처_양말	범주형 피처_여우
0	1	0	0	0	1	0
1	0	1	0	0	0	1
2	0	0	1	0	1	0
3	0	1	0	1	0	0

2. OneHotEncoder 와 ColumnTransformer

- OneHotEncoder는 모든 열을 인코딩한다

```
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder(sparse=False)
print(ohe.fit_transform(demo_df))
```

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

  - sparse=False 로 설정하면 OneHotEncoder가 희소 행렬이 아닌 Numpy 배열로 반환한다
- scikit-learn의 출력은 데이터 프레임이 아니기 때문에 열 이름이 없다
- 변환된 피처에 해당하는 원본 범주형 피처 이름을 얻기 위해 get_feature_names 메서드를 사용한다

```
print(ohe.get_feature_names())
```

```
['x0_0' 'x0_1' 'x0_2' 'x1_상자' 'x1_양말' 'x1_여우']
```

- 처음 3개 열은 첫번째 원본의 피처이며, 뒤의 3개는 두번째 원본의 피처이다

- ColumnTransformer
  - OneHotEncoder는 모든 피처를 범주형이라 가정하기 때문에 바로 적용할 수 없다
  - ColumnTransformer를 이용하여 입력 데이터에 있는 열마다 다른 변환을 적용할 수 있다
  - 연속형 피처와 범주형 피처는 매우 다른 종류의 전처리 과정이 필요하기 때문에 ColumnTransformer가 매우 유용하다

```
display(data.head())
```

	age	workclass	education	gender	hours-per-week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K

- 이 데이터 세트로 선형회귀 모델을 통해 소득(income)을 예측하려면 범주형 변수에 원-핫-인코딩 적용 외에, 연속형 변수인 age와 hours-per-week의 스케일도 조정해야 한다
- 이때 ColumnTransformer 가 필요하다

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler

ct = ColumnTransformer(
    [("scaling", StandardScaler(), ['age', 'hours-per-week']),
     ("onehot", OneHotEncoder(sparse=False), ['workclass', 'education', 'gender', 'occupation'])])
```

◦ 각 열의 변환은 이름, 변환기 객체, 이 변환이 적용될 열을 지정한다

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
data_features = data.drop("income", axis=1)
X_train, X_test, y_train, y_test = train_test_split(data_features, data.income, random_state=0)

ct.fit(X_train)
X_train_trans = ct.transform(X_train)
print(X_train_trans.shape)
```

```
(24420, 44)
```

◦ income을 제외한 모든 열 추출→ data_features

◦ 로지스틱 회귀 실행

```
logreg = LogisticRegression()
logreg.fit(X_train_trans, y_train)

X_test_trans = ct.transform(X_test)
print("테스트 점수: {:.2f}".format(logreg.score(X_test_trans, y_test)))
```

```
테스트 점수: 0.81
```

◦ 데이터 스케일을 바꾸었지만, 테스트 점수는 변화가 없다
◦ 하지만, 하나의 변환기 ColumnTransformer 하나로 변환시킨 장점이 있다

3. make_column_transformer

- make_column_transformer로 간편하게ColumnTransformer
  - ColumnTransformer는 각 단계 이름을 일일이 지정해야 한다
  - 클래스 이름을 기반으로 각 단계에 이름을 자동으로 붙여주는 함수가 make_column_transformer 이다

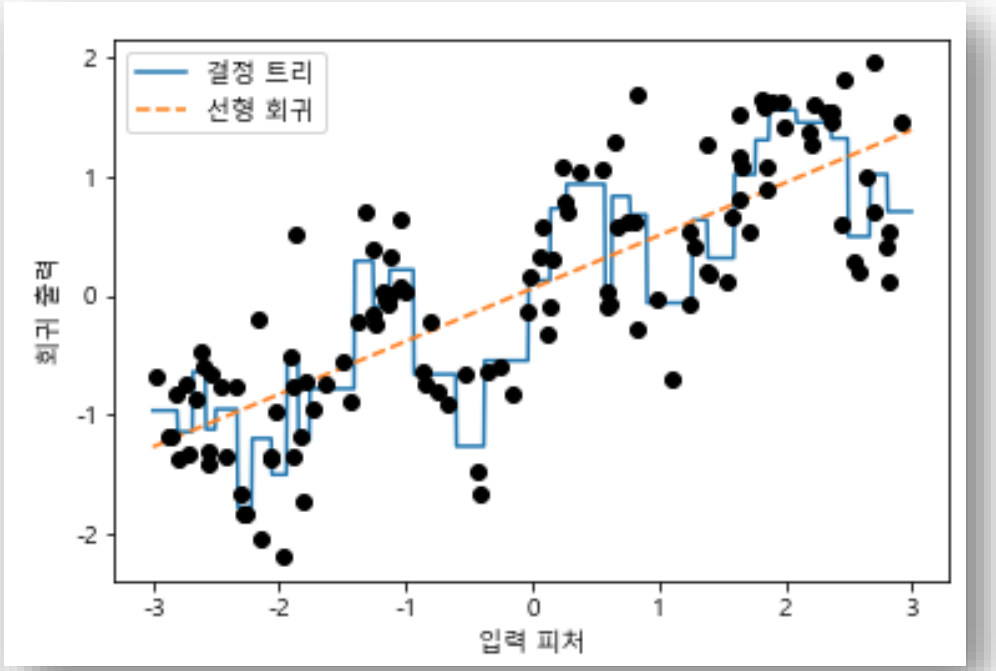
```
from sklearn.compose import make_column_transformer
ct = make_column_transformer(
    (StandardScaler(), ['age', 'hours-per-week']),
    (OneHotEncoder(sparse=False), ['workclass', 'education', 'gender', 'occupation']))
```

  - scaling, onehot 을 사용하지 않았다

4. 구간 분할, 이산화 그리고 선형 모델, 트리 모델

- wave 데이터 세트를 이용하여 피처의 표현 형식에 따라 선형 회귀 모델과 트리 기반 모델의 차이점을 비교한다
- wave 데이터 세트는 하나의 입력 피처를 가지고 있다

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
X, y = mglearn.datasets.make_wave(n_samples=120)
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)
reg = DecisionTreeRegressor(min_samples_leaf=3).fit(X, y)
plt.plot(line, reg.predict(line), label="결정 트리")
reg = LinearRegression().fit(X, y)
plt.plot(line, reg.predict(line), '--', label="선형 회귀")
plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("회귀 출력")
plt.xlabel("입력 피처")
plt.legend(loc="best")
```



- 연속형 데이터에 아주 강력한 선형 회귀 모델을 만드는 방법 중 하나는 한 피처를 여러 피처로 나누는 구간분할 bining(이산화)이다
- KBinsDiscretizer 함수를 이용하여 다양한 구간분할이 가능하다

```
from sklearn.preprocessing import KBinsDiscretizer
```

```
kb = KBinsDiscretizer(n_bins=10, strategy='uniform')
kb.fit(X)
print("bin edges: \n", kb.bin_edges_)
```

```
bin edges:
 [array([-2.9668673 , -2.37804841, -1.78922951, -1.20041062, -0.61159173,
        -0.02277284,  0.56604605,  1.15486494,  1.74368384,  2.33250273,
         2.92132162])]
```

  - 첫번째 구간은 -2.967 에서 -2.378, 두번째 구간은 -2.378 에서 -1.789 순으로 모든 데이터 포인트를 포함한다
  - n_bins = 10: 10개 구간을 만듦

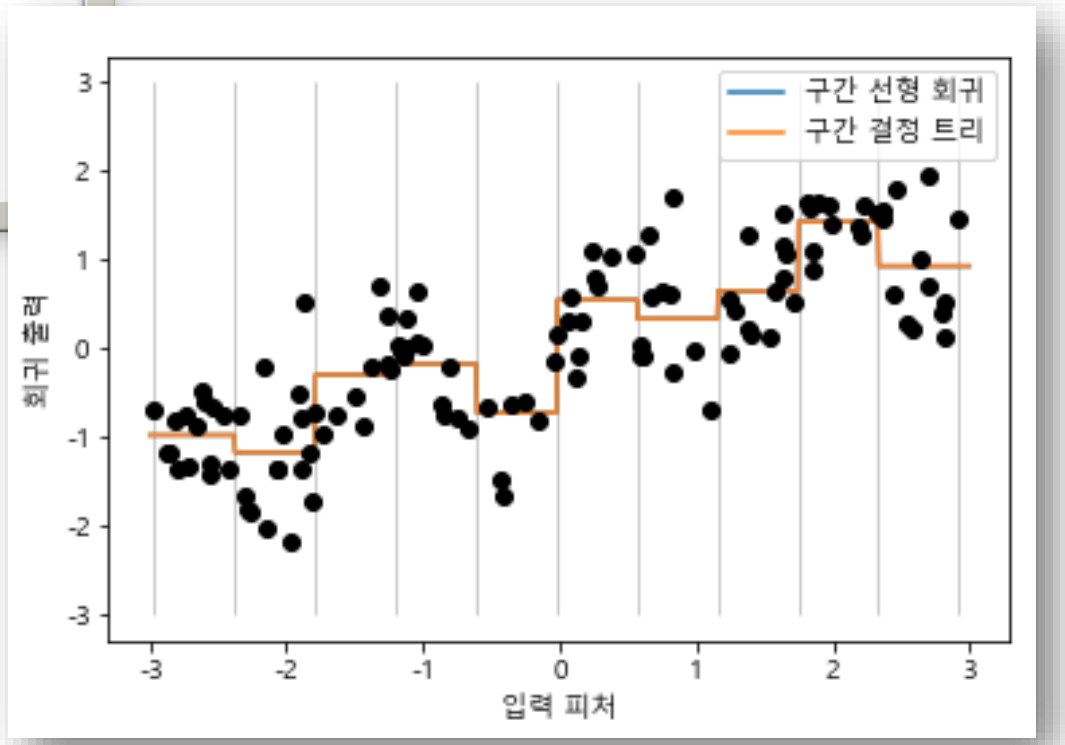
- wave 데이터 세트에 있는 연속형 피처를 각 데이터 포인트가 어느 구간에 속했는지 원-핫-인코딩한 범주형 피처로 변환한다
- encode = 'onehot-dense'로 지정하여 변환한다

```
kb = KBinsDiscretizer(n_bins=10, strategy='uniform', encode='onehot-dense')
kb.fit(X)
X_binned = kb.transform(X)
```

```
line_binned = kb.transform(line)

reg = LinearRegression().fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='구간 선형 회귀')

reg = DecisionTreeRegressor(min_samples_split=3).fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='구간 결정 트리')
plt.plot(X[:, 0], y, 'o', c='k')
plt.vlines(kb.bin_edges_[0], -3, 3, linewidth=1, alpha=.2)
plt.legend(loc="best")
plt.ylabel("회귀 출력")
plt.xlabel("입력 피처")
```



- 구간 선형 회귀 선과 구간 결정 트리 선이 겹쳐있다
- 구간으로 나누기 전과 비교하면 선형 회귀 선이 훨씬 유연해졌다
- 반면 결정 트리는 덜 유연해졌다

5. 상호작용과 다항식

- 상호작용과 다항식 추가
  - 피처를 풍부하게 나타내는 또 다른 방법으로 사용된다
  - 선형 회귀 모델에서 절편 외에도 기울기를 학습할 수 있다
    - 구간으로 분할된 데이터에 원래 피처를 다시 추가하는 방법

```
X_combined = np.hstack([X, X_binned])
print(X_combined.shape)
```

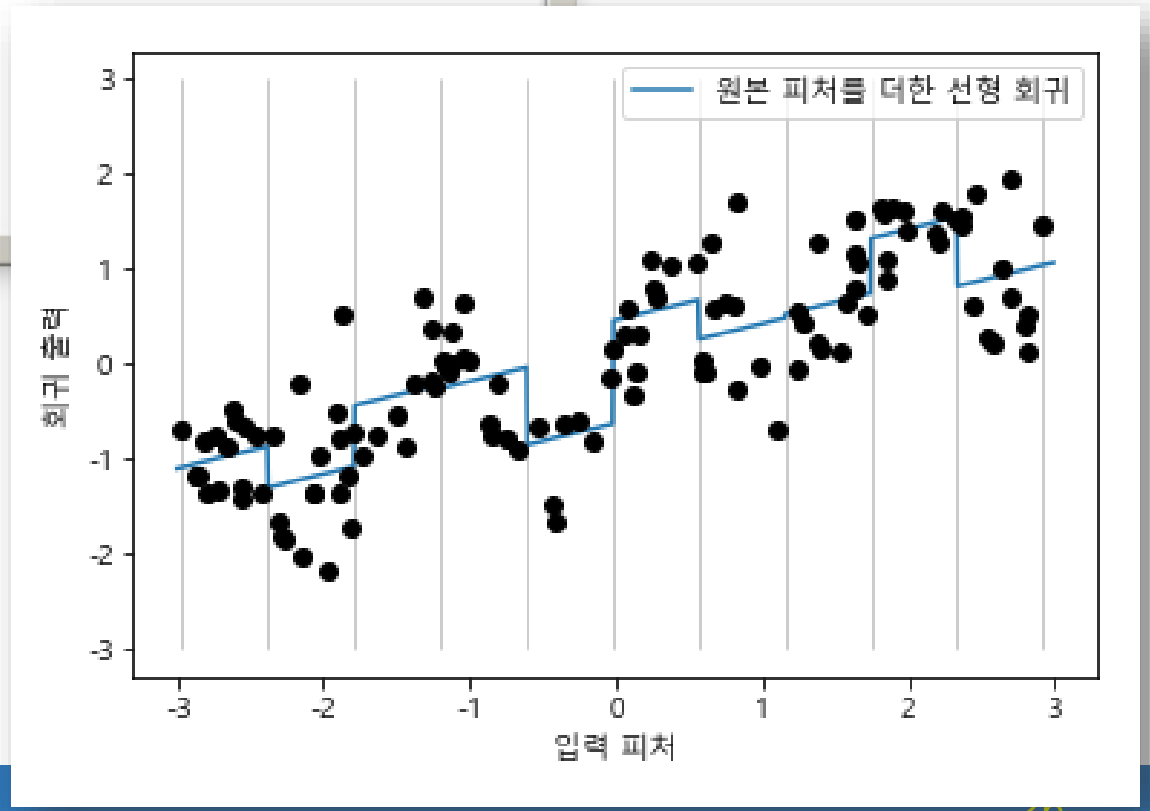
```
(120, 11)
```

    - 앞에서 구간분할을 통해 피처를 10개로 나누었는데 여기에서 하나를 추가하여 11개를 만들었다

```
reg = LinearRegression().fit(X_combined, y)

line_combined = np.hstack([line, line_binned])
plt.plot(line, reg.predict(line_combined), label='원본 피처를 더한 선형 회귀')

plt.vlines(kb.bin_edges_[0], -3, 3, linewidth=1, alpha=.2)
plt.legend(loc="best")
plt.ylabel("회귀 출력")
plt.xlabel("입력 피처")
plt.plot(X[:, 0], y, 'o', c='k')
```



- 피처가 하나이기 때문에 기울기도 하나이다 → 효과가 없어 보임
- x 축 사이의 상호작용을 추가할 수 있다(구간분할 피처와 원본 피처의 곱)

```
X_product = np.hstack([X_binned, X * X_binned])
print(X_product.shape)
```

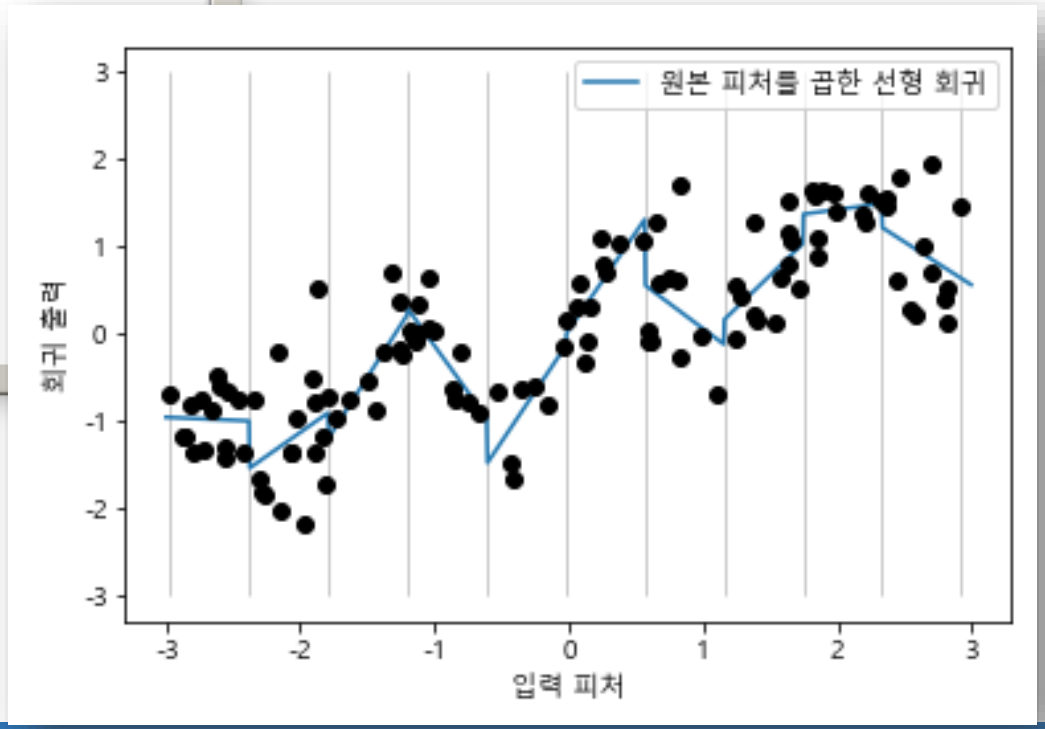
```
(120, 20)
```

```
reg = LinearRegression().fit(X_product, y)

line_product = np.hstack([line_binned, line * line_binned])
plt.plot(line, reg.predict(line_product), label='원본 피처를 곱한 선형 회귀')

plt.vlines(kb.bin_edges_[0], -3, 3, linewidth=1, alpha=.2)

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("회귀 출력")
plt.xlabel("입력 피처")
plt.legend(loc="best")
```



- 피처가 20개로 증가
- 곱셈 피처는 각 구간에 대한 x 축 피처의 복사본으로 생각(각 구간 안에서 원본 피처이고 다른 곳에서는 0)

- 원본 피처에 다항식 추가 방법
  - 피처 x 가 주어지면 x**2, x**3, x**4 등을 시도해 볼 수 있다
  - 이 방식은 preprocessing 모듈의 PolynomialFeatures에 구현되어 있다

```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=10, include_bias=False)
poly.fit(X)
X_poly = poly.transform(X)
```

```
print("X_poly.shape:", X_poly.shape)
```

```
X_poly.shape: (120, 10)
```

  - x**10까지 고차항을 추가한다
  - 10개 피처가 만들어진다

◦ X 와 X_ploy 값 비교

```
print("X 원소:\n", X[:5])
print("X_poly 원소:\n", X_poly[:5])
```

```
X 원소:
 [[-0.75275929]
 [ 2.70428584]
 [ 1.39196365]
 [ 0.59195091]
 [-2.06388816]]
```

```
X_poly 원소:
 [[-7.52759287e-01  5.66646544e-01 -4.26548448e-01  3.21088306e-01
  -2.41702204e-01  1.81943579e-01 -1.36959719e-01  1.03097700e-01
  -7.76077513e-02  5.84199555e-02]
 [ 2.70428584e+00  7.31316190e+00  1.97768801e+01  5.34823369e+01
   1.44631526e+02  3.91124988e+02  1.05771377e+03  2.86036036e+03
   7.73523202e+03  2.09182784e+04]
 [ 1.39196365e+00  1.93756281e+00  2.69701700e+00  3.75414962e+00
   5.22563982e+00  7.27390068e+00  1.01250053e+01  1.40936394e+01
   1.96178338e+01  2.73073115e+01]
 [ 5.91950905e-01  3.50405874e-01  2.07423074e-01  1.22784277e-01
   7.26822637e-02  4.30243318e-02  2.54682921e-02  1.50759786e-02
   8.92423917e-03  5.28271146e-03]
 [-2.06388816e+00  4.25963433e+00 -8.79140884e+00  1.81444846e+01
  -3.74481869e+01  7.72888694e+01 -1.59515582e+02  3.29222321e+02
  -6.79478050e+02  1.40236670e+03]]
```

◦ 각 피처의 차수를 알려주는 get_feature_names 메서드 사용

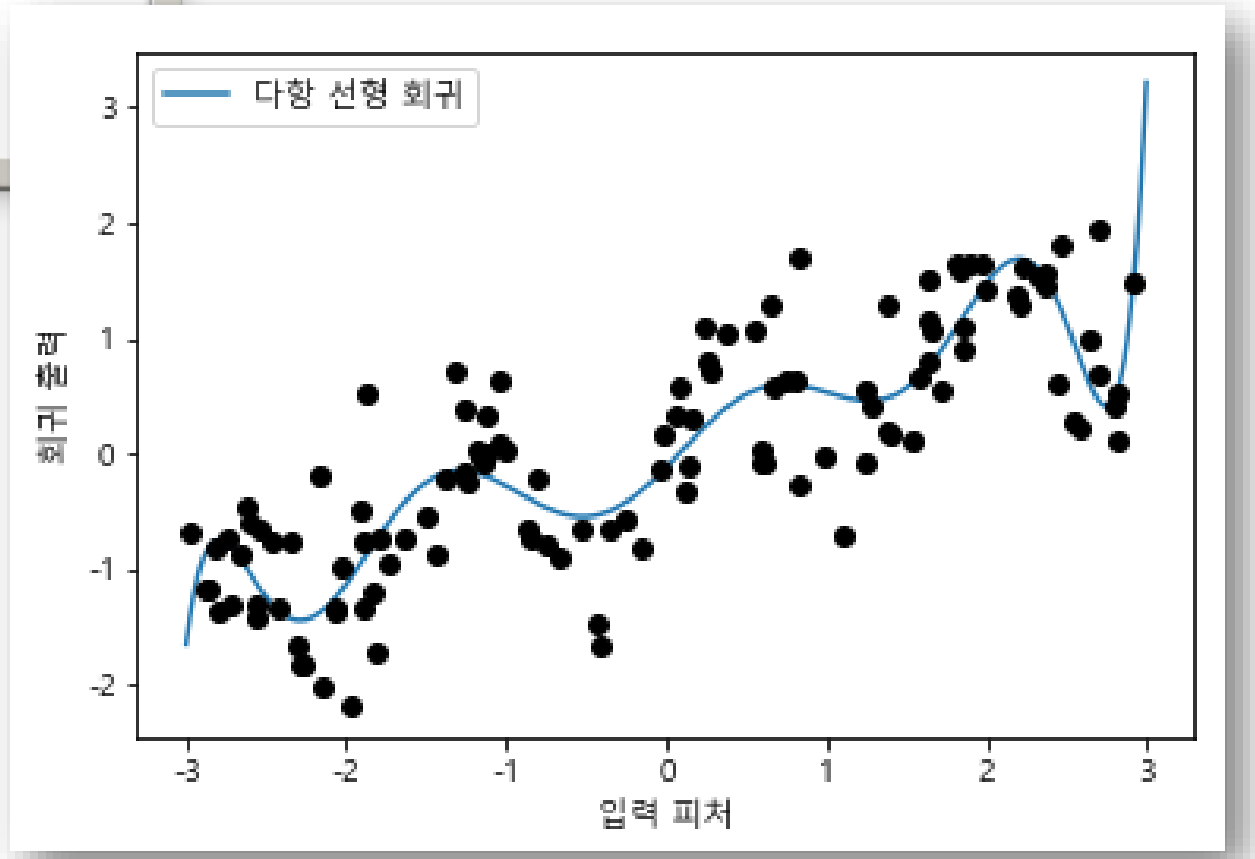
```
print("항 이름:\n", poly.get_feature_names())
```

```
항 이름:
 ['x0', 'x0^2', 'x0^3', 'x0^4', 'x0^5', 'x0^6', 'x0^7', 'x0^8', 'x0^9', 'x0^10']
```

◦ 다항식 피처를 선형 회귀 모델에 사용하면, 다항 회귀 모델이 된다

```
reg = LinearRegression().fit(X_poly, y)

line_poly = poly.transform(line)
plt.plot(line, reg.predict(line_poly), label='다항 선형 회귀')
plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("회귀 출력")
plt.xlabel("입력 피처")
plt.legend(loc="best")
```



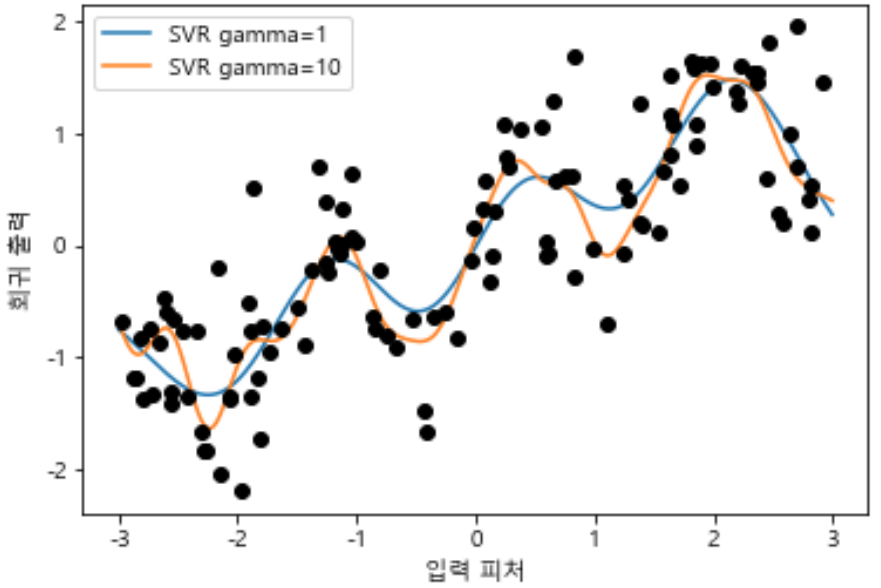
- 다항식 피처는 1차원 데이터 세트이지만 매우 부드러운 곡선을 만든다
- 고차원 다항식은 데이터가 부족한 경우 세밀하게 동작한다

◦ 원본 데이터에 커널 SVM 모델을 학습시켰다

```
from sklearn.svm import SVR

for gamma in [1, 10]:
    svr = SVR(gamma=gamma).fit(X, y)
    plt.plot(line, svr.predict(line), label='SVR gamma={}'.format(gamma))

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("회귀 출력")
plt.xlabel("입력 피처")
plt.legend(loc="best")
```



◦ SVM은 다항식과 유사한 곡선

◦ 보스톤 주택 가격 데이터 세트를 적용하여 다항식 특성 비교
  ◦ MinMaxScaler를 사용해 스케일을 0에서 1 사이로 조정한다

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target, random_state=0)

scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
poly = PolynomialFeatures(degree=2).fit(X_train_scaled)
X_train_poly = poly.transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)
print("X_train.shape:", X_train.shape)
print("X_train_poly.shape:", X_train_poly.shape)
```

```
X_train.shape: (379, 13)
X_train_poly.shape: (379, 105)
```

- 이 데이터는 원래 피처가 13개인데 105개 교차 피처로 확장되었다
- 새로운 피처는 원래 피처의 제곱과 가능한 두 피처의 조합을 모두 포함한다(degree=2 사용)
- 어떤 원본 피처가 곱해져 새 피처가 만들어졌는지 확인하기 위해서는 get_feature_names 메서드를 사용한다

◦ 상호작용 피처가 있는 데이터와 없는 데이터를 Ridge를 사용해 비교했다

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train_scaled, y_train)
print("상호작용 피처가 없을 때 점수: {:.3f}".format(ridge.score(X_test_scaled, y_test)))
ridge = Ridge().fit(X_train_poly, y_train)
print("상호작용 피처가 있을 때 점수: {:.3f}".format(ridge.score(X_test_poly, y_test)))
```

```
상호작용 피처가 없을 때 점수: 0.621
상호작용 피처가 있을 때 점수: 0.753
```

◦ 상호작용과 다항식 피처가 Ridge 성능을 높혔다

◦ 랜덤 포레스트를 사용한 경우

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=100, random_state=0).fit(X_train_scaled, y_train)
print("상호작용 피처가 없을 때 점수: {:.3f}".format(rf.score(X_test_scaled, y_test)))
rf = RandomForestRegressor(n_estimators=100, random_state=0).fit(X_train_poly, y_train)
print("상호작용 피처가 있을 때 점수: {:.3f}".format(rf.score(X_test_poly, y_test)))
```

```
상호작용 피처가 없을 때 점수: 0.795
상호작용 피처가 있을 때 점수: 0.774
```

◦ 상호작용을 추가하면, 성능이 다소 떨어진다. 이유는 모델이 복잡하기 때문이다

6. 피처 자동 선택

▪ 일변량 통계

- 지도 학습 방법이므로 최적값을 찾기 위해서는 타깃값이 필요하다
- 개개의 피처와 타깃 사이에 중요한 통계적 관계가 있는지 조사한다
- 그 후, 깊게 관련되어 있다고 판단되는 피처를 선택한다(분류에서는 분산분석 ANOVA 라고도 한다)
- 이 방법의 핵심 요소는 일변량, 즉 각 피처가 독립적으로 평가된다는 점이다
  - 따라서 다른 피처와 깊게 연관된 피처는 선택되지 않는다
- 이 방법은 계산이 빠르며, 평가를 위한 모델을 만들 필요가 없다

- scikit-learn에서 일변량 분석으로 피처를 선택하려면, 분류에서는 f_classif(기본값)를, 회귀에서는 f_regression을 선택하여 테스트하고, 계산한 p-값에 기초하여 피처를 제외하는 방식을 선택한다
- 이 방식은 매우 높은 p-값을 가진(타깃값과 연관성이 적은) 피처를 제외할 수 있도록 임계값을 조정하는 매개변수를 사용한다
  - SelectPercentile: 지정된 비율만큼 피처 선택
  - SelectKBest: 고정된 k 개의 피처 선택
- cancer 데이터 세트를 이용하 피처 선택 적용한다
  - 문제를 복잡하게 하기 위해 노이즈 피처를 데이터 추가하고
  - 피처 선택이 이 노이즈 피처를 식별해서 제거하는지 확인한다

```python
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectPercentile, f_classif
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()

rng = np.random.RandomState(42)
noise = rng.normal(size=(len(cancer.data), 50))

X_w_noise = np.hstack([cancer.data, noise])

X_train, X_test, y_train, y_test = train_test_split(
    X_w_noise, cancer.target, random_state=0, test_size=.5)

select = SelectPercentile(score_func=f_classif, percentile=50)
select.fit(X_train, y_train)

X_train_selected = select.transform(X_train)

print("X_train.shape:", X_train.shape)
print("X_train_selected.shape:", X_train_selected.shape)
```

임의의 수 발생

데이터에 노이즈 피처 추가

f_classif(기본값)와 SelectPercentile을 사용하여 피처의 50%를 선택한다

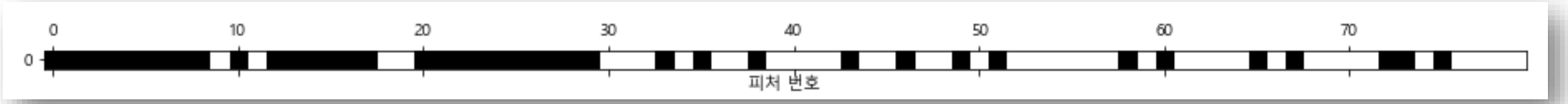
```
X_train.shape: (284, 80)
X_train_selected.shape: (284, 40)
```

- get_support 메서드는 선택된 피처를 불리언 값으로 표시하여 어떤 피처가 선택되었는지 확인할 수 있다

```
mask = select.get_support()
print(mask)
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("피처 번호")
plt.yticks([0])
```

```
[ True  True  True  True  True  True  True  True  True False  True False
  True  True  True  True  True  True False False  True  True  True  True
  True  True  True  True  True  True False False False  True False  True
 False False  True False False False False  True False False  True False
 False  True False  True False False False False False False  True False
  True False False False False  True False  True False False False False
  True  True False  True False False False False]
```

  - 검은색 True, 흰색 False



  - 대부분 30번 미만이 검은색으로 원본 피처가 선택되었다
  - 하지만, 완벽하게 복원되지 않았다

◦ 전체 피처와 선택된 피처만 사용했을 때, 로지스틱 회귀 성능 비교

```
from sklearn.linear_model import LogisticRegression

X_test_selected = select.transform(X_test)

lr = LogisticRegression()
lr.fit(X_train, y_train)
print("전체 피처를 사용한 점수: {:.3f}".format(lr.score(X_test, y_test)))
lr.fit(X_train_selected, y_train)
print("선택된 일부 피처를 사용한 점수: {:.3f}".format(
      lr.score(X_test_selected, y_test)))
```

```
전체 피처를 사용한 점수: 0.930
선택된 일부 피처를 사용한 점수: 0.940
```

◦ 노이즈를 제거한 쪽 성능이 높다
◦ 하지만 실제 사회에서는 다를 수도 있다

■ 모델 기반 피처 선택

- 지도 학습 머신러닝 모델을 사용하여 피처의 중요도를 평가해서 가장 중요한 피처들만 선택한다
  - 피처 선택에 사용되는 학습 모델이 최종적으로 사용할 지도 학습 모델과 같을 필요가 없다
  - 피처 선택을 위한 모델은 각 피처의 중요도를 측정하여 순서를 매길 수 있어야 한다
    - feature_importances_ 속성: 결정 트리 모델에서 사용
- 한번에 모든 피처를 고려하므로 상호작용 부분을 반영할 수 있다
- 피처 선택은 SelectFromModel 에 구현되어 있다

- SelectFromModel
  - 중요도가 지정한 임계치보다 큰 모든 피처를 선택한다
  - 열변량 분석으로 선택한 피처와 결과를 비교하기 위해 절반 가량의 피처가 선택될 수 있도록 중간값을 임계치로 사용한다

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42), threshold="median")
```

```
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
print("X_train.shape:", X_train.shape)
print("X_train_l1.shape:", X_train_l1.shape)
```

```
X_train.shape: (284, 80)
X_train_l1.shape: (284, 40)
```

```
mask = select.get_support()

plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("피처 번호")
plt.yticks([0])
```

- 이번에는 원본에서 2개를 제외하고 모든 원본 피처가 선택되었다

```
X_test_l1 = select.transform(X_test)
score = LogisticRegression().fit(X_train_l1, y_train).score(X_test_l1, y_test)
print("테스트 점수: {:.3f}".format(score))
```

```
테스트 점수: 0.951
```

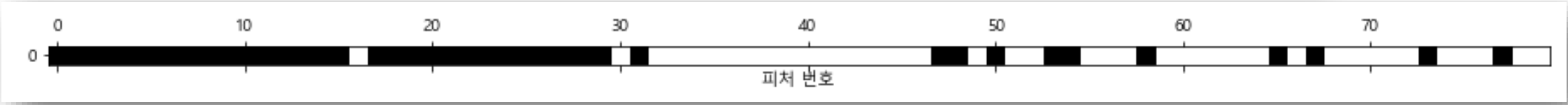
▪ 반복적 피처 선택

- ‘반복적 피처 선택’ Iterative Feature Selection은 피처의 수가 각기 다른 일련의 모델이 만들어진다
  - 피처를 하나도 선택하지 않은 상태로 시작해서 어떤 종료 조건에 도달할 때까지 하나씩 추가하는 방법
  - 모든 피처를 가지고 시작해서 어떤 종료 조건이 될 때까지 피처를 하나씩 제거해가는 방법
- ‘재귀적 피처 제거’ Recursive Feature Elimination
  - 모든 피처를 가지고 시작해서 피처 중요도가 가장 낮은 피처를 제거
  - 다음 나머지 피처 전체로 새로운 모델을 만들어 미리 정의한 피처 개수가 남을 때 까지 지속

- 피처 중요도를 결정하는 방법 제공이 필요하다
- 다음은 랜덤 포레스트 모델을 사용한 결과이다

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42), n_features_to_select=40)

select.fit(X_train, y_train)
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("피처 번호")
plt.yticks([0])
```



- 여전히 피처 1개를 놓쳤다

◦ RFE를 사용해서 피처를 선택했을 때, 로지스틱 회귀 모델의 정확도

```
X_train_rfe = select.transform(X_train)
X_test_rfe = select.transform(X_test)

score = LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe, y_test)
print("테스트 점수: {:.3f}".format(score))
```

```
테스트 점수: 0.951
```

◦ RFE에 사용된 모델을 이용한 예측(이 경우 선택된 피처만 사용)

```
print("테스트 점수: {:.3f}".format(select.score(X_test, y_test)))
```

```
테스트 점수: 0.951
```

◦ 피처 선택이 잘되면 선형 회귀 모델 성능도 랜덤 포레스트와 견줄만 하다

요약

- 범주형 변수
- OneHotEncoder와 ColumnTransformer
- make_column_transformer
- 구간 분할, 이산화 그리고 선형 모델, 트리 모델
- 상호작용과 다항식
- 피처 자동 선택