Lecture 17: Non-linear regression

Textbook: chapter 10

Dr. Huiping Cao

Outline

- Adding polynomial terms using scikit-learn
- Transform the data (non-polynomial) before applying linear regression
- Decision tree regressor to deal with nonlinear regression
- Random forest regressor to deal with nonlinear regression

Data

```
from sklearn.datasets import fetch_california_housing
import numpy as np
import pandas as pd

CA_housing = fetch_california_housing(as_frame=True)
print('CA_housing feature names:', CA_housing.feature_names)

df_all = CA_housing.frame
df_all = df_all.rename(columns = {'MedHouseVal': 'MEDV'})
df = df_all.iloc[:1000]
print('df.info:', df.info)
```

```
CA_housing feature names: ['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup',
'Latitude', 'Longitude’]


…
[1000 rows x 9 columns]
```

Linear regression

- When explanatory and response variable does not have a linear relationship, we can account for the non-linearity using a **polynomial regression model** by adding polynomial terms.

$$y = w_0 + w_1\mathbf{x} + w_2\mathbf{x}^2 + \ \ldots + w_d\mathbf{x}^d$$

- *d* denotes the **degree of the polynomial**.
- It is still considered a multiple linear regression model because of the linear regression coefficients *w*.

Adding polynomial terms to model nonlinear relationships in the Housing dataset

- Step 1: Create polynomial features from X
- Step 2: Fit the polynomial features to a linear regression model

Create polynomial features

***class* sklearn.preprocessing.PolynomialFeatures(*degree=2*, *interaction_only=False*, *include_bias=True*, *order='C'*)**

- Generate polynomial and interaction features.
- Generate a **new feature matrix** consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, if an input sample is two dimensional and of the form [a, b], the degree-2 polynomial features are [1, a, b, a^2, ab, b^2].

Create polynomial features from X

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

X=np.array([1.0, 2.0,3.0,4.0,5.0])
X=X[: ,np.newaxis]
quadratic = PolynomialFeatures(degree=2)
X2= quadratic.fit_transform(X)
print("X2: ", X2)

cubic = PolynomialFeatures(degree=3)
X3= cubic.fit_transform(X)
print("X3: ", X3)
```

```
X2:
[[ 1. 1. 1.]
 [ 1. 2. 4.]
 [ 1. 3. 9.]
 [ 1. 4. 16.]
 [ 1. 5. 25.]]
X3:
[[ 1. 1. 1. 1.]
 [ 1. 2. 4. 8.]
 [ 1. 3. 9. 27.]
 [ 1. 4. 16. 64.]
 [ 1. 5. 25. 125.]]
```

- X has 1 feature (a) and 5 records (i.e., 5 values)
- X2 has 3 features (a^0, a^1, a^2) and 5 records. The first column's values (1, 1, 1, 1, 1) represent the values for feature a^0, the 2nd column's values (1, 2, 3, 4, 5) represent the values for feature a^1, the 3rd column's values (1, 4, 9, 16, 25) represent the values for feature a^2
- X3 has 4 features (a^0, a^1, a^2 , a^3) and 5 records

Create polynomial features from X – Example 2

```
X = np.array([[2, 3],[4,5],[6,7]])
print(X.shape,X)

quadratic = PolynomialFeatures(degree=2)
X2 = quadratic.fit_transform(X)
print(X2.shape, "X2: ", X2)

cubic = PolynomialFeatures(degree=3)
X3= cubic.fit_transform(X)
print(X3.shape,"X3: ", X3)
```

```
X.shape: (3, 2)
X: [[2 3] [4 5] [6 7]]
X2.shape (3, 6)
X2: [[ 1. 2. 3. 4. 6. 9.]
      [ 1. 4. 5. 16. 20. 25.]
      [ 1. 6. 7. 36. 42. 49.]]
X3,shape: (3, 10)
X3: [[ 1. 2. 3. 4. 6. 9. 8. 12. 18. 27.]
      [ 1. 4. 5. 16. 20. 25. 64. 80. 100. 125.]
      [ 1. 6. 7. 36. 42. 49. 216. 252. 294. 343.]]
```

- X features: a, b
- X2 features: a, b, a^2, ab, b^2
- X3 features: a, b, a^2, ab, b^2, a^3, a^2b, ab^2, b^3

Fit the polynomial features to a linear regression model - example

- Get a X and y
- Fit a **linear model using X** and y directly
- Generate degree-2 polynomial features (X-quadratic) for X and fit a **linear model using the X-quadratic** and y
- Generate degree-3 polynomial features (X-cubic) for X and fit a **linear model using the X-cubic** and y
- Compare the results of the three models

Fit the polynomial features to a linear regression model

```
X = df[['MedInc', 'AveRooms']].values
y = df['MEDV'].values

#linear
lr_linear = LinearRegression()
lr_linear.fit(X,y)
y_pred_linear = lr_linear.predict(X)
r2_linear = r2_score(y,y_pred_linear)
print("X.shape: ", X.shape)

#polynomial 2nd degree (quadratic)
quadratic = PolynomialFeatures(degree=2)
X_quadratic = quadratic.fit_transform(X)
lr_quadratic = LinearRegression()
lr_quadratic.fit(X_quadratic,y)
y_pred_quadratic=lr_quadratic.predict(X_quadratic)
r2_quadratic = r2_score(y,y_pred_quadratic)
print("X_quadratic.shape: ", X_quadratic.shape)
```

```
#polynomial 3rd degree (cubic)
cubic = PolynomialFeatures(degree=3)
X_cubic= cubic.fit_transform(X)
lr_cubic = LinearRegression()
lr_cubic.fit(X_cubic, y)
y_pred_cubic = lr_cubic.predict(X_cubic)
r2_cubic = r2_score(y, y_pred_cubic)
print("X_cubic.shape: ", X_cubic.shape)

print("R^2 linear : %.3f , quadratic %.3f , cubic : %.3f "
    %(r2_linear , r2_quadratic , r2_cubic ) )
```

```
X.shape:  (1000, 2)
X_quadratic.shape:  (1000, 6)
X_cubic.shape:  (1000, 10)
R^2 linear : 0.588 , quadratic 0.629 , cubic : 0.637
```

Fit the polynomial features to a linear regression model

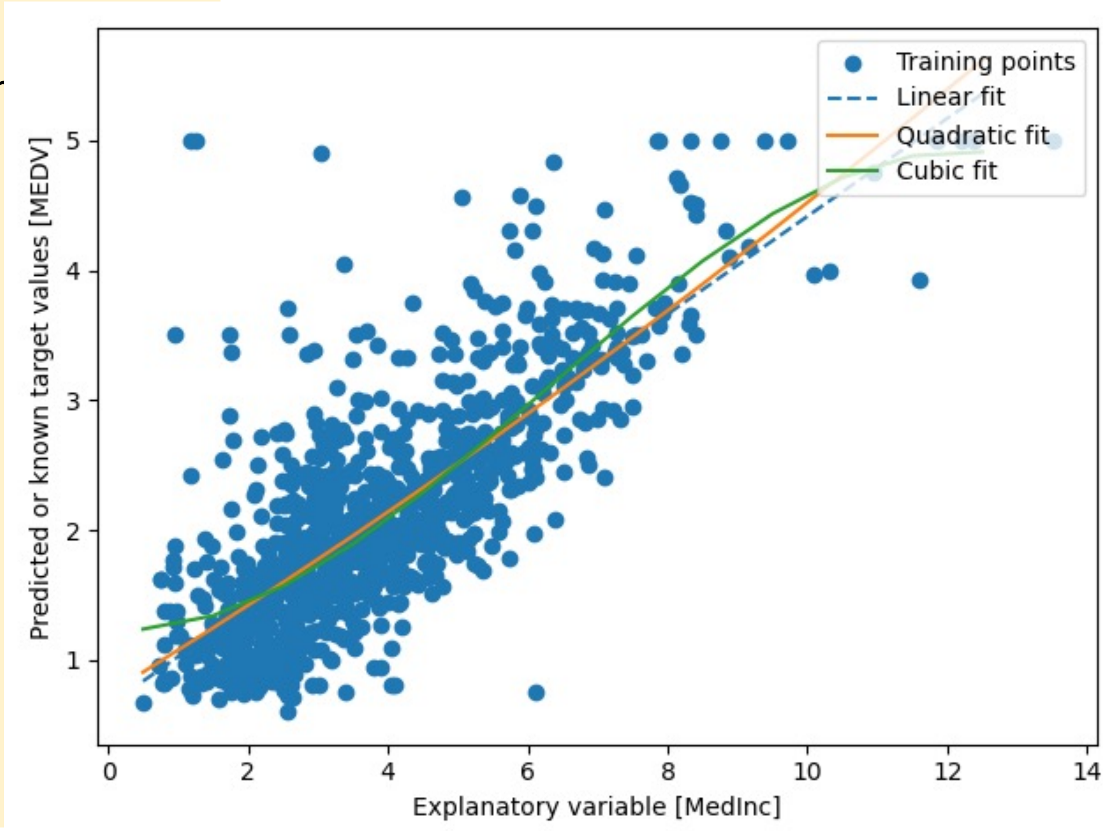
```
X = df[['MedInc']].values
# reeat code in the previous slide

X_fit = np.arange(X.min(),X.max(),1)[:, np.newaxis]
y_pred_linear2 = lr_linear.predict(X_fit)
y_pred_quadratic2 = lr_quadratic.predict(quadratic.fit_transforr
y_pred_cubic2 = lr_cubic.predict(cubic.fit_transform(X_fit))

plt.scatter(X, y, label='Training points')
plt.plot(X_fit, y_pred_linear2, label='Linear fit', linestyle='--')
plt.plot(X_fit, y_pred_quadratic2, label='Quadratic fit')
plt.plot(X_fit, y_pred_cubic2, label='Cubic fit')
plt.xlabel('Explanatory variable')
plt.ylabel('Predicted or known target values')
plt.legend(loc='upper right’)

plt.tight_layout()
plt.show()
```

- According to the results, we can see that the cubic fit is the best.
- Note: adding more polynomial features increases the complexity of a model, and thus increases the chance of overfitting.



Transform the data before applying linear regression

- **Polynomial features** are not always the best choice for modeling nonlinear relationships. Sometimes, transform the data beforehand may be more useful.

Transform the data before applying linear regression

```
X = df[['MedInc']].values
y = df['MEDV'].values
X_log = np.log(X)
y_sqrt = np.sqrt(y)

lr_linear = LinearRegression()
lr_linear.fit(X_log, y_sqrt)
y_pred = lr_linear.predict(X_log)
r2_linear = r2_score(y_sqrt,y_pred)
print("R^2 transformed data linear : %.3f " %r2_linear)

# calculate the fitted line
X_fit = np.arange(X_log.min()-1, X_log.max()+1, 1)[:, np.newaxis]
y_lin_fit = lr_linear.predict(X_fit)

# plot the fitted line
```

```
R^2 transformed data linear : 0.513
```

- Transform X to log(X) and transform y to sqrt(y)
- Create linear model between log(X) and sqrt(y)
- Make prediction
- Calculate and plot the fitted line
- This does not improve the linear R2

Decision tree regressor

- from sklearn.tree import DecisionTreeRegressor

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

  - f: the feature/attribute to perform the split.
  - D_p: dataset corresponding to the parent node.
  - D_{left}, D_{right}: datasets corresponding to the left and right child node respectively.
  - N_p, N_{left}, N_{right}: number of samples in D_p, D_{left}, D_{right} respectively.
  - $I(\cdot)$: impurity measure of a node.

Decision tree regressor

$$IG\left(D_p, f\right) = I\left(D_p\right) - \frac{N_{left}}{N_p} I\left(D_{left}\right) - \frac{N_{right}}{N_p} I\left(D_{right}\right)$$

- $I(\cdot)$: impurity measure of a node.
  - For classification, *I*(·) uses entropy or Gini index.
  - For regression, the impurity metric is defined as the MSE of the node *Dt* (or **within-node variance**).

$$I(D_t) = MSE(D_t) = \frac{1}{N_t} \sum_{i \in D_t} \left(y^{(i)} - \hat{y}_t\right)^2$$

$\hat{y}_t$: the predicted target value (sample mean) for node D_t : $\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} \hat{y}^{(i)}$

Decision tree regressor

```
from sklearn.tree import DecisionTreeRegressor

X = df[['MedInc', 'AveRooms']].values
y = df['MEDV'].values

tree = DecisionTreeRegressor(max_depth=3)
tree. fit (X,y)
y_pred = tree.predict (X)
print("R^2 decision tree regressor: %.3f" %(r2_score(y,y_pred)))
```

```
R^2 decision tree regressor: 0.629
```

- Using MedInc and AveRooms:
  - R^2 linear R^2 linear : 0.588 , quadratic 0.629 , cubic : 0.637
  - Decision tree regressor: 0.629

Support Vector Regression (SVR)

sklearn.svm.**SVR**(*, kernel='rbf', degree=3, gamma='scale', coef0=0.0, tol=0.001, C=1.0, epsilon=0.1, shrinking=True, cache_size=200, verbose=False, max_iter=- 1)

SVM classifier

sklearn.svm.SVC(*, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0,
shrinking=True, probability=False, tol=0.001,
cache_size=200, class_weight=None, verbose=False, max_iter=- 1, decision_function_shape='ovr',
break_ties=False, random_state=None)

Support Vector Regression (SVR)

score(X, y, sample_weight=None)

Return the coefficient of determination of the prediction. The coefficient of determination $\mathbf{R^2}$.

```
>>> from sklearn.svm import SVR
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import StandardScaler
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> regr = make_pipeline(StandardScaler(), SVR(C=1.0, epsilon=0.2))
>>> regr.fit(X, y)
Pipeline(steps=[('standardscaler', StandardScaler()), ('svr', SVR(epsilon=0.2))])
```