# Lecture 17: Fit linear regression model using scikit-learn library functions & regression evaluation

Textbook: chapter 10

Dr. Huiping Cao

# Fit linear regression model

- Steps to fit a linear regression model
  - Preprocessing data
  - Scikit-learn function LinearRegression
  - Plot the results
  - Convert the target variable to the original scale
- Robust linear regression using RANSAC
- Regularization
  - Ridge Regression
  - LASSO
  - Elastic Net

# Preprocessing – feature scaling

```
X = df[['MedInc']].values
y = df[['MEDV']].values
print("X:", type(X), X.shape)
print("y:", type(y),y.shape)

sc_x = StandardScaler()
sc_x.fit(X)
X_std = sc_x.transform(X)

sc_y = StandardScaler()
sc_y.fit(y)
y_std = sc_y.transform(y).flatten()
print("y_std flattened: ", type(y_std),y_std.shape)
```

```
X: <class 'numpy.ndarray'> (1000, 1)
y: <class 'numpy.ndarray'> (1000, 1)
y_std flattened:  <class 'numpy.ndarray'> (1000,)
```

- df[[MedInc’]].values, df[[’MEDV’]].values gets the values of ‘MedInc’ and ‘MEDV’ in a two-dimensional array format.
- Most transformers in scikit-learn expect data to be stored in 2-dimensional arrays.

# Scikit-learn function LinearRegression

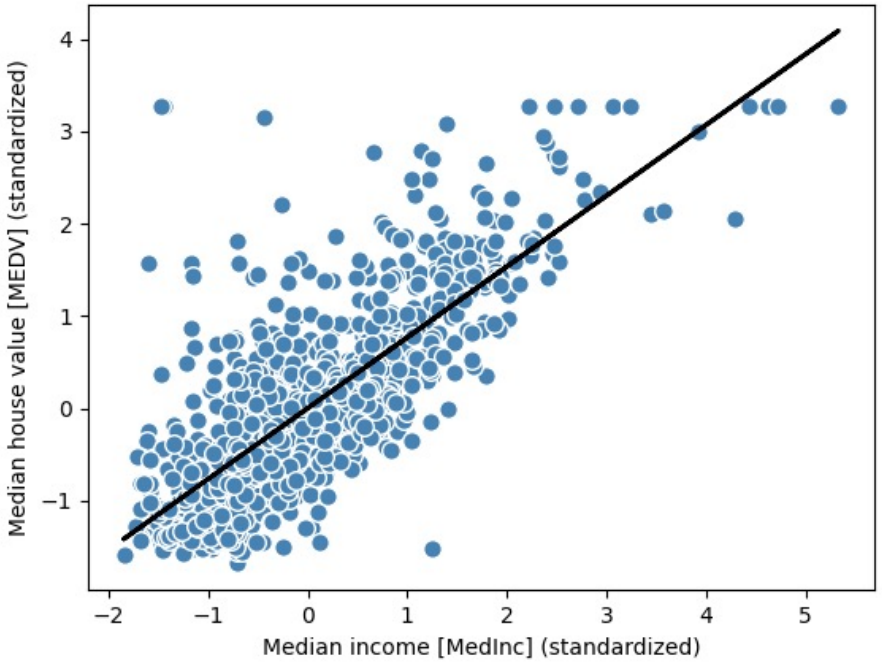
- **LinearRegression** model, results:
  - coef_[0]
  - Intercept_
- The **intercept** of the model that work with standardized variables is always **zero**.

```
from sklearn.linear_model import LinearRegression

lrmodel_original = LinearRegression()
lrmodel_original.fit(X,y)
print("Fitting original data, slope = %.3f" %lrmodel_original.coef_[0])
print("Fitting original data, intercept = %.3f" %lrmodel_original.intercept_)

lrmodel_std = LinearRegression()
lrmodel_std.fit(X_std,y_std)
print("Fitting standardized data, slope = %.3f " %lrmodel_std.coef_[0])
print("Fitting standardized data, intercept = %.3f " %lrmodel_std.intercept_)
```

```
Fitting original data, slope = 0.377
Fitting original data, intercept = 0.647
Fitting standardized data, slope = 0.767
Fitting standardized data, intercept = -0.000
```

# Plot the results

```
def linear_regression_plot(X, y, model):
    plt.scatter(X, y, c='steelblue', edgecolor = 'white', s=70)
    plt.plot(X,model.predict(X),color='black', lw=2)
    return None

import matplotlib.pyplot as plt
linear_regression_plot(X_std,y_std, lrmodel_std)
plt.xlabel('Median income [MedInc] (standardized)')
plt.ylabel('Median house value [MEDV] (standardized)')
plt.show()
```



# Convert the target variable to the original scale

```
test_data = [[8.0]]
MedInc_std = sc_x.transform(test_data)
price_std = lrmodel_std.predict(MedInc_std)

price = sc_y.inverse_transform([price_std])
print("price_std=%0.3f" % price_std, " price=%0.3f" % price)
```

```
price_std=1.760
price=3.662
```

# Robust linear regression using RANSAC

- Linear regression is sensitive to outliers.
- A robust method of regressing using **RANdom SAmple, Consensus (RANSAC)** algorithm. This algorithm fits a regression model to a subset of the data, which is called **inliers**.

# Algorithms of RANSAC

- Select **a random number of samples** to be inliers and fit the model.
- **Test** all other data points against the fitted model and add those points that fall within a user-given tolerance to the inliers.
- **Refit** the model using all inliers.
- **Estimate the error** of the fitted model versus the inliers.
- **Terminate** the algorithm if the performance meets a certain user-defined threshold or if a fixed number of iterations were reached; go back to step 1 otherwise.

# Skicit learn RANSACRegressor

- **min_sample=50**: the minimum number of the randomly chosen samples is at least 50.
- **loss=** ***'absolute_error'***: an argument for the residual_metric parameter. It calculates the absolute vertical distances between the fitted line and the sample points.
    - The loss 'absolute_loss' was deprecated in v1.0 and will be removed in version 1.2.

```
from sklearn.linear_model import RANSACRegressor

ransac = RANSACRegressor(LinearRegression(), max_trials=100,
            min_samples=50, loss='absolute_error',
            residual_threshold=5.0, random_state=1)

ransac.fit(X_std, y_std)
print('Slope: %.3f' %ransac.estimator_.coef_[0])
print ('Intercept : %.3f' %ransac.estimator_ . intercept_ )

#make predictions
```

```
Slope: 0.767
Intercept : -0.000
```

# Regularization

- **Regularization** is one approach to tackle the problem of overfitting by adding additional information, which will shrink the parameter values of the model to induce a penalty against complexity.
- Most popular regularized linear regression approaches are
  - (1) Ridge Regression,
  - (2) Least Absolute Shrinkage and Selection Operator (LASSO)
  - (3) Elastic Net

# Ridge Regression

- This is an **L2 penalized model**. It adds the squared sum of the weights to the lest-squared cost function.
- Increase the value of $\lambda$, increase the regularization strength and shrink the weights of the model.
- Please note that it does not regularize the intercept term $w_0$

$$J(\mathbf{w})_{Ridge} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda\|w\|_2^2 = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda\sum_{i=1}^{m} w_j^2$$

# LASSO

- L1 penalty to generate sparse models. Sparse models mean that many $w_j$s are zero.
- Reducing L2 norm: it is more effective to reduce the larger $w_j$s than reducing smaller $w_j$s.
  - E.g., change a weight value from 10 to 9 reduces the L2 norm by 19 (100-81), while reduce from 2 to 1 only reduces the L2 norm by 3 (4-1).

$$J(\mathbf{w})_{LASSO} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda\|w\|_1 = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda\sum_{i=1}^{m}\left|w_j\right|$$

# LASSO

- Reducing L1 norm: it is equally effective to reduce a larger $w_j$ or a smaller $w_j$.
  - Consider the above example, reducing 10 to 9 has the same effect as reducing 2 to 1. The optimization tends to reduces the smaller weights to be smaller (even to zero).
- The sparse model makes LASSO useful to select features.

# Elastic Net

- It is a compromise between Ridge regression and LASSO. It has an L1 penalty to generate sparsity and L2 penalty to overcome some of the limitations of LASSO.

$$J(\mathbf{w})_{ElasticNet} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_1 \sum_{i=1}^{m} w_j^2 + \lambda_2 \sum_{i=1}^{m} \left|w_j\right|$$

# Scikit-learn library functions

- In ElasticNet function, if we set *l1_ratio* to be 1.0, it is equal to the LASSO regression.

```
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet

ridge = Ridge(alpha=1.0)
lasso = Lasso(alpha=1.0)
elasticnet = ElasticNet(alpha=1.0, l1_ratio=0.5)

# fit the models
# make predictions
```

# Regression evaluation

- Prediction
- Evaluate linear regression models
  - Mean squared error (MSE)
  - Coefficient of determination
  - Residual plot

# Prediction

- We separate the full dataset to a training set and a test set.
- Then, we train a linear regression model, and calculates the prediction from the training data and the testing data.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

X = df.iloc[:,:-1].values
y = df['MEDV'].values
print (X.shape)

X_train, X_test, y_train, y_test =
   train_test_split (X, y, test_size=0.3, random_state=0)

lr = LinearRegression()
lr.fit(X_train, y_train)
y_train_pred = lr.predict(X_train)
y_test_pred = lr .predict(X_test)
print(y_train_pred.shape)
print(y_test_pred.shape)
```

```
(1000, 8)
(700,)
(300,)
```

# Mean squared error (MSE)

- MSE is useful to compare different regression models or for tuning their parameters via grid search and cross-validation.
- It is defined as

$$MSE = \frac{1}{n} \sum_{(i=1)}^{n} \left( y^{(i)} - \hat{y}^{(i)} \right)^2$$

# Calculate MSE

```
from sklearn.metrics import mean_squared_error

error_train = mean_squared_error(y_train, y_train_pred)
error_test = mean_squared_error(y_test, y_test_pred)
print('MSE train: %.3f, test:%.3f' % (error_train, error_test))
```

```
MSE train: 0.266, test:0.337
```

- The testing error is higher than the training error, which may indicate that the model is overfitting.

# Coefficient of determination $R^2$

$$R^2 = 1 - \frac{SSE}{SST}$$

- Where *SSE* is the sum of squared error. *SST* is the total sum of squares (or the variance of the response), SST = $\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2$
- $R^2$ is a rescaled version of MSE.
- $R^2 = 1 - \frac{SSE}{SST} = 1 - \frac{\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2} = 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2} = 1 - \frac{MSE}{Var(y)}$
- For the training dataset, $R^2 \in [0,1]$. For the testing data, it can be negative.
- When $R^2 = 1$, the model fits the data perfectly with a corresponding MSE=0.

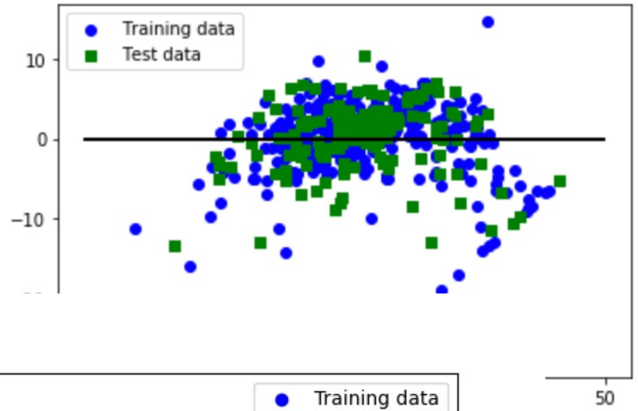
# Calculate $R^2$

```
from sklearn.metrics import r2_score

r2_train = r2_score (y_train, y_train_pred )
r2_test = r2_score (y_test, y_test_pred )
print('R^2 train: %.3f, test:%.3f' % (r2_train, r2_test))
```

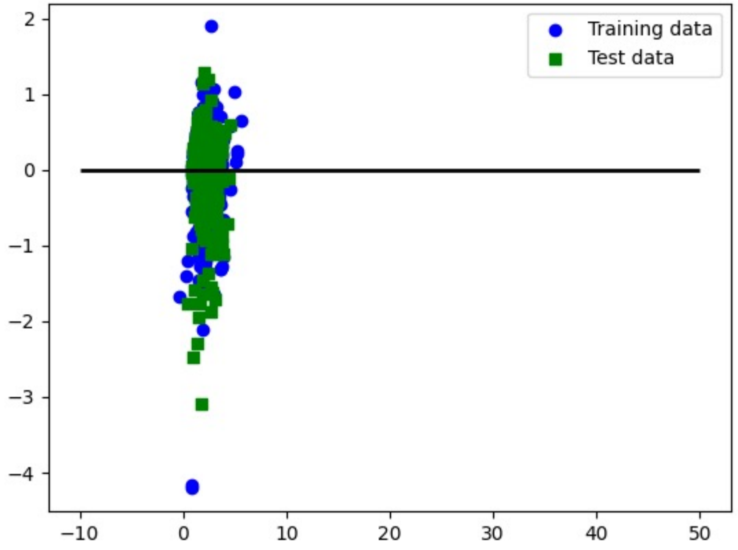
```
R^2 train: 0.640, test:0.628
```

# Residual plot

- In case of a perfect prediction, the residuals would be exactly zero. In real applications, this will not happen.




```
import matplotlib.pyplot as plt

plt.scatter(y_train_pred,y_train_pred - y_train,
            c='blue', marker='o', label='Training data')
plt.scatter(y_test_pred, y_test_pred - y_test,
            c='green', marker='s', label='Test data')

plt.hlines(y=0, xmin=-10, xmax=50, color='black', lw=2)
plt.legend(loc='best')
plt.show()
```



# Residual plot

- For a good regression model, we would expect that the **errors are randomly distributed** and the **residuals should be randomly scattered** around the centerline.
- If the residual plot shows some pattern, it means that our model is not able to capture some explanatory information.
- The residual plot can also help detect **outliers**, which are presented by the points with a large deviation from the centerline.