PennState World Campus

DAAN862: Analytics Programming in Python

Lesson 9: Supervised Learning with SciKit-Learn II: Regression

---

Lesson 9: Objectives and Overview (1 of 5)

Lesson 9: Supervised Learning with SciKit-Learn II: Regression

---

The mechanism of the three regression models introduced in this lesson have been introduced in the course IE575. Here we only provide a quick walk-through of how to use regression algorithms with Scikit-learn package in Python.

At the end of this lesson, using scikit-learn, the student will be able to:

Build and evaluate Linear Regression models

Build and evaluate Decision Tree models

Build and evaluate Nerual networks models

By the end of this lesson, please complete all readings and assignments found in the Lesson 9 Course Schedule.

Lesson 9.1: Linear Regression (2 of 5)

Lesson 9.1: Regression Model

Linear regression is the most popular regression model since you can build the model quickly and the model is easy to interpret.

First, look at Figure 9.1 where we import the necessary packages into Python:

```
In [1]: import os
   ...: import pandas as pd
   ...: from sklearn.model_selection import train_test_split
   ...: import sklearn.metrics as metrics
   ...: import matplotlib.pyplot as plt
```

Fig 9.1 (click to enlarge)

The mtcars dataset is used to build all regression models as shown in Figure 9.2. Since you have used this dataset for assignments, we are not going to explore the data again.

```
In [2]: path = "E:\GoogleDrive\PSU\DAAN862\Course contents\Lesson 9"

In [3]: os.chdir(path)

In [4]: mtcars = pd.read_csv("mtcars.csv")

In [5]: mtcars.describe()
Out[5]:
              mpg        cyl        disp   ...            am       gear     carb
count  32.000000  32.000000   32.000000   ...     32.000000  32.000000  32.0000
mean   20.090625   6.187500  230.721875   ...      0.406250   3.687500   2.8125
std     6.026948   1.785922  123.938694   ...      0.498991   0.737804   1.6152
min    10.400000   4.000000   71.100000   ...      0.000000   3.000000   1.0000
25%    15.425000   4.000000  120.825000   ...      0.000000   3.000000   2.0000
50%    19.200000   6.000000  196.300000   ...      0.000000   4.000000   2.0000
75%    22.800000   8.000000  326.000000   ...      1.000000   4.000000   4.0000
max    33.900000   8.000000  472.000000   ...      1.000000   5.000000   8.0000

[8 rows x 11 columns]
```

Fig 9.2 (click to enlarge)

Data Preparation, Model Generation, and Residual Analysis:

(click the tabs to learn more)

Data Preparation:

We will build a simple linear regression model with the disp column as X and mpg column as y. This simple model allows us to visualize the data and the model. For complicated model, you can assign more columns to X. The expand_dims is used to convert the one-dimensional array to two-dimensional array as shown in Figure 9.3:

```
In [6]: X = np.expand_dims(mtcars.disp, axis = 1)

In [7]: y = mtcars.mpg

In [8]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
   ...:                                                     random_state = 123)
```

Fig 9.3 (click to enlarge)

Model Generation:

Creating a linear regression object to fit the model using the training data and make predictions using the test set is shown in Figure 9.4:

```
In [9]: from sklearn import linear_model

In [10]: lreg = linear_model.LinearRegression()

In [11]: lreg.fit(X_train, y_train)
Out[11]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=False)
```

Fig 9.4 (click to enlarge)

The coefficients can be viewed from Figure 9.5:

```
In [12]: lreg.coef_
Out[12]: array([-0.03920055])

In [13]: lreg.intercept_
Out[13]: 29.347055987487952
```

Fig 9.5 (click to enlarge)

Model Evaluation:

The performance of the model can be evaluated by R-square and mean squared error as shown in Figure 9.6:

```
In [14]: lreg_train_pred = lreg.predict(X_train)

In [15]: lreg_test_pred = lreg.predict(X_test)

In [16]: metrics.r2_score(y_train, lreg_train_pred)
Out[16]: 0.6937060764344621

In [17]: metrics.mean_squared_error(y_train, lreg_train_pred)
Out[17]: 12.024245106958194

In [18]: metrics.r2_score(y_test, lreg_test_pred)
Out[18]: 0.7839519517623064

In [19]: metrics.mean_squared_error(y_test, lreg_test_pred)
Out[19]: 5.599619733443839
```

Fig 9.6 (click to enlarge)

Residual Analysis:

1. Plot the original data points and the fitted line as shown in Figure 9.7 A & B:

```
In [20]: plt.figure()
    ...: plt.scatter(X, y, color = 'black', label = 'True values')
    ...: plt.plot(X, lreg.predict(X), color = 'blue', linewidth = 3,
    ...:          label = 'Linear regression model')
    ...: plt.xlabel('Displacement')
    ...: plt.ylabel('MPG')
    ...: plt.legend()
Out[20]: <matplotlib.legend.Legend at 0x2dce9edb7b8>
```

Fig 9.7 (A) (click to enlarge)

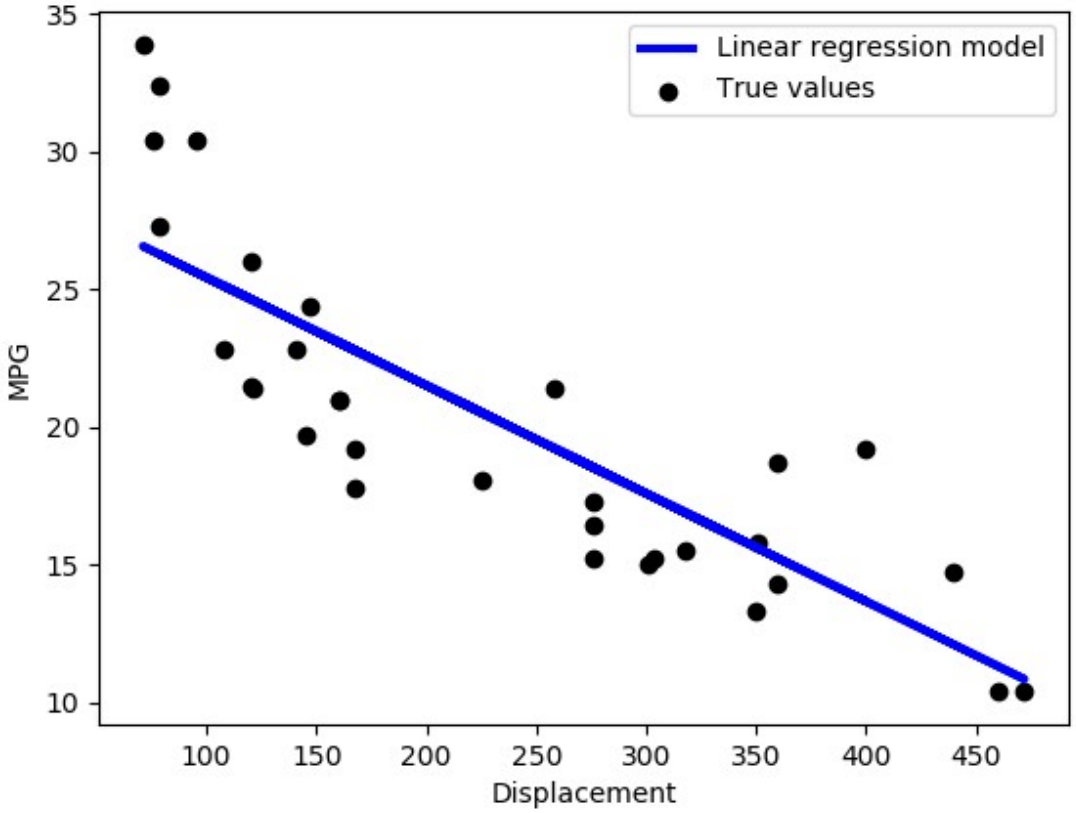

Fig 9.7 (B) (click to enlarge)

From this plot, we can tell that Mpg has a linear relationship with Displacement.

2. Figure 9.8 A & B shows the scatterplot of fitted values against residuals:

```
In [21]: y_pred = lreg.predict(X)
    ...: residual = y - y_pred
    ...: plt.figure()
    ...: plt.scatter(y_pred, residual)
    ...: plt.hlines(0, xmin = 10, xmax = 30)
    ...: plt.xlim([10, 30])
    ...: plt.xlabel('Predicted values')
    ...: plt.ylabel('Residual')
Out[21]: Text(0,0.5,'Residual')
```

Fig 9.8 (A) (click to enlarge)

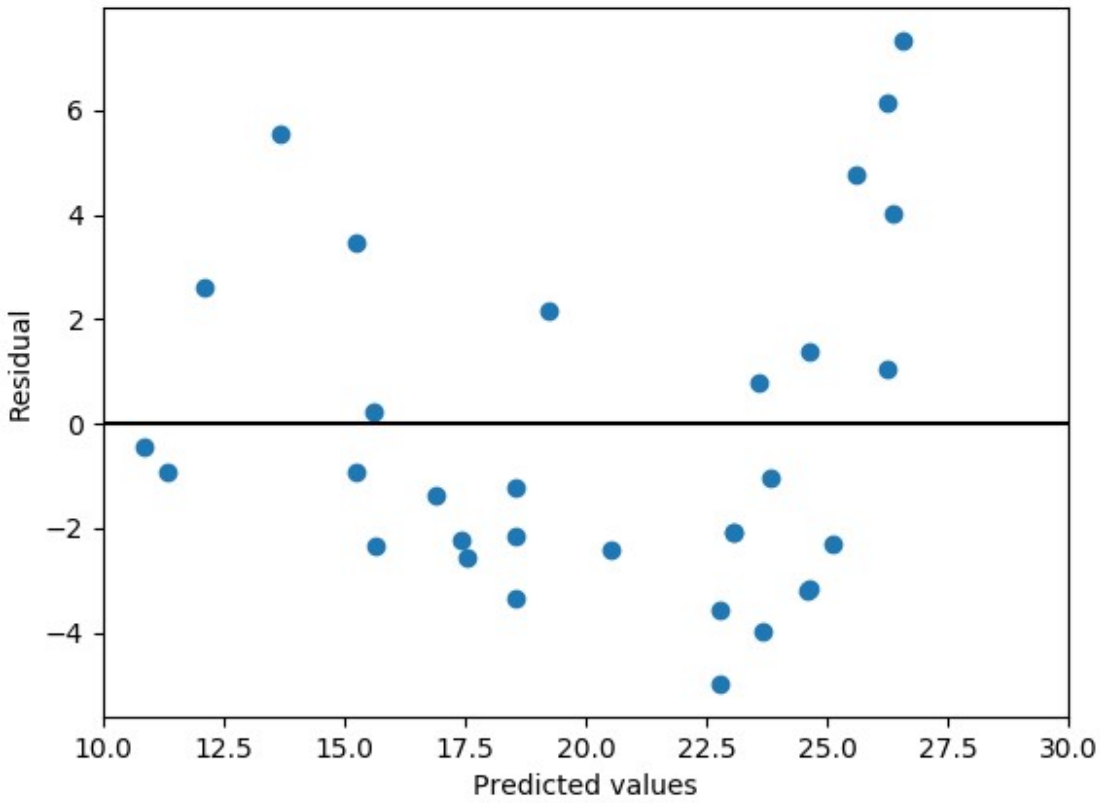

Fig 9.8 (B) (click to enlarge)

It looks like the residual is equally distributed around y=0 line and there is no obvious pattern in the residual.

3. QQ plot:

QQ plot shows how well the distribution of residuals fit the normal distribution. In Python, we will use the probplot (shown in Figure 9.9 A & B) in *Scipy* package to realize it:

```
In [22]: from scipy.stats import probplot
    ...: plt.figure()
    ...: probplot(residual, plot = plt)
```

Fig 9.9 (A) (click to enlarge)

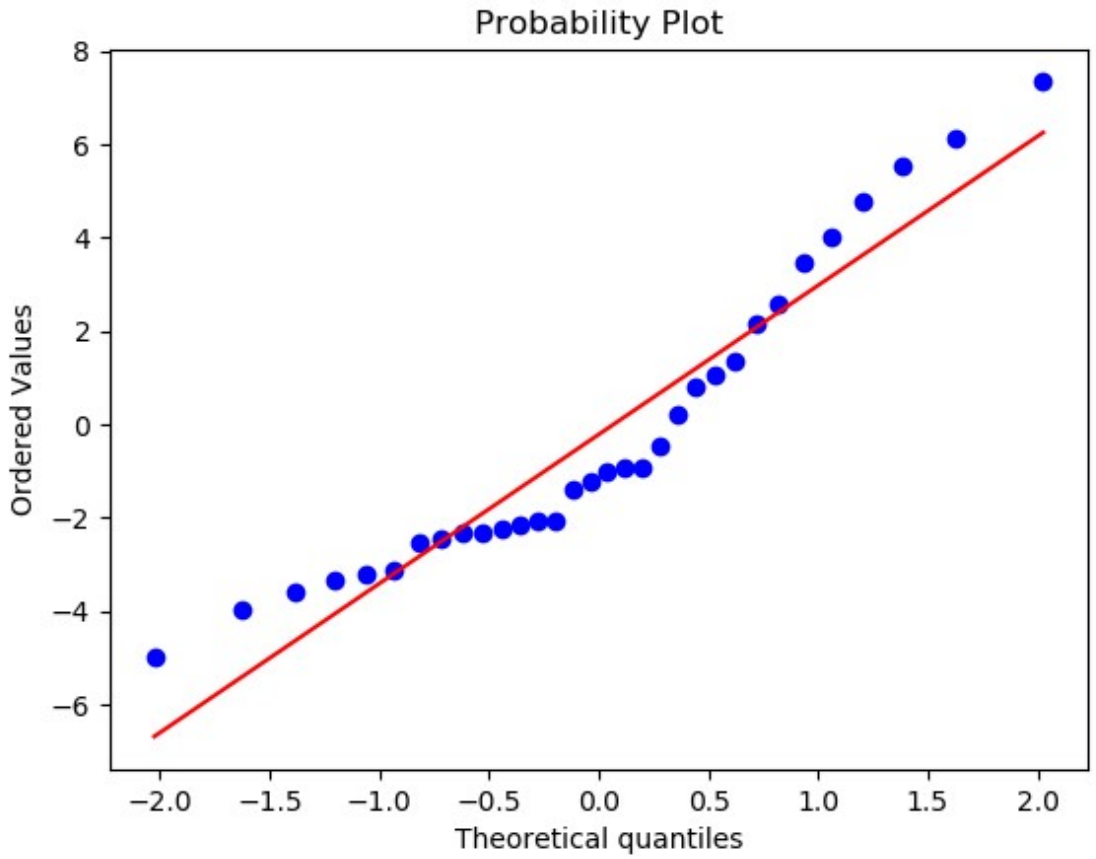

Fig 9.9 (B) (click to enlarge)

Transcript

In this video, I will introduce how to use a linear regression model.

First, let's import the required packages: os, pandas, trian_test_split, metrics, matplotlip.pyplot and numpy.

For all regression model introduced in this lesson, we will use mtcars data as an example. I will not explore this data since you have used it before.

Let's change the working directory and load mtcars into Python.

For the linear regression model, we only choose one variable, disp, as X. We need to use expand_dims function to adjust its dimension from (32,) to (32, 1). The reason to pick one variable here is for easy visulation.

y is mpg variable.

Now you can split X, y into train and test data, with test_size = 0.3, and random_state = 123.

From sklean import linear_model, and you can find linear regression is under linear_model.

First, we initiate the linear regression model and use default parameters.

Now, we fit the model with X_train, y_train.

Again, you can use coef underscore and intercept underscore to check coefficient and intercept of the

linear model.

For model evaluation, we predict mpg value for both train and test sets. And use r squared and mean squared error to evaluate it.

From the results, it shows that it is not a bad model. Since R square is 0.69 for train data and 0.78 for the test set.

Now, we can plot the data and the regression model. We use the scatter plot for data points, and line plot for the regression model. It looks like that mpg do have a linear relation with displacement.

Now we can analyize residual. The first plot is Residual verse predicted values, and it looks like residuals are randomly distributed around y = 0.

Second, we use qqplot to check if the residual is normally distributed. We need to import probplot from scipy.stats.

I added a semicolon here such that it won't show probability results in the console. From qqplot, we can see that the residual has a relative good normal distribution.

You can try to use more variable to build a linear regression model.

Lesson 9.2: Regression Tree

We have introduced Classification tree in the previous lesson, here we show how to use tree model (DecisionTreeRegressor) for regression tasks. *DecisionTreeClassifier* has similar parameters, attribute and methods with *DecisionTreeClassifer.* Please refer to lesson 8.3 for the details.

Data Preparation, Model Generation, Model Evaluation, and Model Improvement:

(click the tabs to learn more)

Data Preparation:

For CART regression tree model, all variables of the mtcars dataset will be used. First, we split the data into training and test sets as shown in Figure 9.10:

```
In [23]: X = mtcars.loc[:, 'cyl':]

In [24]: y = mtcars.mpg

In [25]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
    ...:                                                     random_state = 123)
```

Fig 9.10 (click to enlarge)

Model Generation:

Next, we import tree models into Python, create a regression tree object and fit the model with training data. The *min_samples_leaf* is set to 5 as shown in Figure 9.11:

```
In [27]: from sklearn import tree

In [28]: tree_model = tree.DecisionTreeRegressor(min_samples_leaf = 5,
    ...:                                         random_state = 39)

In [29]: tree_model.fit(X_train, y_train)
Out[29]:
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=5,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=39, splitter='best')
```

Fig 9.11 (click to enlarge)

Figure 9.12 A & B shows how you can plot the tree structure by using the graphviz package:

```
In [30]: from graphviz import Source

In [31]: Source(tree.export_graphviz(tree_model, out_file=None,
    ...:                             feature_names=X.columns))
```

Fig 9.12 (A) (click to enlarge)

Fig 9.12 (B)

Model Evaluation:

The performance of the regression tree on both train and test sites are shown in Figure 9.13:

```
In [32]: tree_train_pred = tree_model.predict(X_train)

In [33]: metrics.mean_squared_error(y_train, tree_train_pred)
Out[33]: 5.591662337662337

In [34]: metrics.r2_score(y_train, tree_train_pred)
Out[34]: 0.8575634327625987

In [35]: tree_test_pred = tree_model.predict(X_test)

In [36]: metrics.mean_squared_error(y_test, tree_test_pred)
Out[36]: 19.596158367346938

In [37]: metrics.r2_score(y_test, tree_test_pred)
Out[37]: 0.2439287005622669
```

Fig 9.13 (click to enlarge)

It looks the performance on training data is much better than the test data, which is a strong indication of overfitting. Overfitting is a common issue for tree models.

Model Improvement:

Let's increase the *min samples size* to see if the overfitting can be solved. Figure 9.14 shows how we set the min sample size to 8:

```
In [38]: tree_model1 = tree.DecisionTreeRegressor(min_samples_leaf = 8,
    ...:                                           random_state = 39)
    ...: tree_model1.fit(X_train, y_train)
    ...:
    ...: Source(tree.export_graphviz(tree_model1, out_file=None,
    ...:                             feature_names=X.columns))
Out[38]:
```

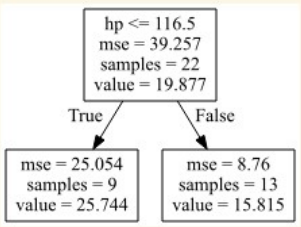

Fig 9.14 (click to enlarge)

Figure 9.15 shows that we got a very simple tree with only two branches:

```
In [39]: tree_train_pred1 = tree_model1.predict(X_train)

In [40]: metrics.mean_squared_error(y_train, tree_train_pred1)
Out[40]: 15.425415695415687

In [41]: metrics.r2_score(y_train, tree_train_pred1)
Out[41]: 0.6070679652692539

In [42]: tree_test_pred1 = tree_model1.predict(X_test)

In [43]: metrics.mean_squared_error(y_test, tree_test_pred1)
Out[43]: 11.850100372561913

In [44]: metrics.r2_score(y_test, tree_test_pred1)
Out[44]: 0.5427919789585037
```

Fig 9.15 (click to enlarge)

The overfitting are largely reduced by using a simple tree. In addtion, the tree model is not appropriate for this data since it predicts y to be either 25.744 or 15.815.

Transcript

In this lesson, I will introduce how to use regression tree model. For this model, we will use all variable except for mpg and model as x variable, and mpg as y variable.

We split X, y into X train, y train, X test and y test.

Similarly, from sklean import tree.

The regression tree is called DecisionTreeRegressor, you can initiate with minimum sample leaf equals to 5 and random state equals to 39.

Now, you can fit the model with X_train, y_train,

For visualize the tree, you need import Source function from graphviz package

And use Source to plot the dot file created from tree dot export underscore graphviz.

For model evaluation, we use mean squared error and r squared.

We get a poor performance on test set and good performance on train set, which is the indication of overfitting. You can play with the parameters to improve the model.

Lesson 9.3: Nerual Network

Class *MLPRegressor* is the neural network model for regression tasks. It uses a multi-layer perception to train the data with the backpropagation method and the square error as the loss function.

Data Preparation, Model Generation, Model Evaluation, and Model Optimization:

(click the tabs to learn more)

Data Preparation:

For Neural Network, preprocessing data is recommended. First, let's convert gear and carb variables to dummy variables as shown in Figure 9.16:

```
In [41]: dummies_gear = pd.get_dummies(mtcars['gear'], prefix ='gear')

In [42]: dummies_gear[:3]
Out[42]:
   gear_3  gear_4  gear_5
0       0       1       0
1       0       1       0
2       0       1       0

In [43]: dummies_carb = pd.get_dummies(mtcars['carb'], prefix = 'carb')

In [44]: dummies_carb[:3]
Out[44]:
   carb_1  carb_2  carb_3  carb_4  carb_6  carb_8
0       0       0       0       1       0       0
1       0       0       0       1       0       0
2       1       0       0       0       0       0
```

Fig 9.16 (click to enlarge)

Next, as shown in Figure 9.17 we need to join *dummies_gear* and *dummies_carb* to the data (after remove the original gear and carg variable):

```
In [45]: mtcars_dummies = mtcars.iloc[:, 1:10].join(dummies_gear)

In [46]: mtcars_dummies = mtcars_dummies.join(dummies_carb)

In [47]: mtcars_dummies.columns
Out[47]:
Index(['mpg', 'cyl', 'disp', 'hp', 'drat', 'wt', 'qsec', 'vs', 'am', 'gear_3',
       'gear_4', 'gear_5', 'carb_1', 'carb_2', 'carb_3', 'carb_4', 'carb_6',
       'carb_8'],
      dtype='object')
```

Fig 9.17 (click to enlarge)

Then, we need to rescale all variables to the range [0, 1]. In Figure 9.18 we use a slightly different approach compared to the lesson 8. First, we fit the *scaler* with training data instead of using *fit_transform* function and then transform the train and test dataset:

```
In [48]: from sklearn.preprocessing import MinMaxScaler

In [49]: train, test = train_test_split(mtcars_dummies, test_size = 0.3,
    ...:                                random_state = 123)

In [50]: scaler = MinMaxScaler()

In [51]: scaler.fit(train)
Out[51]: MinMaxScaler(copy=True, feature_range=(0, 1))

In [52]: train = scaler.transform(train)

In [53]: test = scaler.transform(test)
```

Fig 9.18 (click to enlarge)

After rescaling the train and test sets, they are divided into X and y (shown in Figure 9.19):

```
In [54]: X_train_scaled = train[:, 1:]

In [55]: X_test_scaled = test[:, 1:]

In [56]: y_train_scaled = train[:, 0]

In [57]: y_test_scaled = test[:, 0]
```

Fig 9.19 (click to enlarge)

Model Generation:

Now the data is ready for model generation. Let's import the model and create an MLPRegressor object with 50 hidden nodes and using logistic as the activation function shown in Figure 9.20:

```
In [58]: from sklearn import neural_network

In [59]: nn_model = neural_network.MLPRegressor(10, activation = 'logistic',
    ...:                                        max_iter = 10000, random_state =
21)

In [60]: nn_model.fit(X_train_scaled, y_train_scaled)
Out[60]:
MLPRegressor(activation='logistic', alpha=0.0001, batch_size='auto',
       beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=10, learning_rate='constant',
       learning_rate_init=0.001, max_iter=10000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=21, shuffle=True,
       solver='adam', tol=0.0001, validation_fraction=0.1, verbose=False,
       warm_start=False)
```

Fig 9.20 (click to enlarge)

The coefficients and intercepts are shown in Figure 9.21:

```
In [61]: nn_model.coefs_
    ...: nn_model.intercepts_
Out[61]:
[array([ 0.13751866,  0.05974603,  0.2206123 , -0.27563786,  0.41171846,
        -0.35085505,  0.08822939,  0.11984628, -0.22345124,  0.27746675]),
 array([0.02845864])]
```

Fig 9.21 (click to enlarge)

Model Evaluation:

The model performance is shown in Figure 9.22:

```
In [62]: nn_train_pred = nn_model.predict(X_train_scaled)

In [63]: metrics.r2_score(y_train_scaled, nn_train_pred)
Out[63]: 0.8605519867793164

In [64]: metrics.mean_squared_error(y_train_scaled, nn_train_pred)
Out[64]: 0.1394480132206836

In [65]: nn_test_pred = nn_model.predict(X_test_scaled)

In [66]: metrics.r2_score(y_test_scaled, nn_test_pred)
Out[66]: 0.5694820706459445

In [67]: metrics.mean_squared_error(y_test_scaled, nn_test_pred)
Out[67]: 0.28423659472373364
```

Fig 9.22 (click to enlarge)

Model Optimization:

Let's try to optimize the parameters of the neural network model by using grid search shown in Figure 9.23.

- The parameters to be optimized are hidden-layer sizes and activation functions
- The mean square error is used as the metrics.

First, we need to import the *GridSearchCV* class and create a dict with the parameters as the keys and potential values you want to try as the values:

```
In [68]: from sklearn.model_selection import GridSearchCV

In [69]: tuned_parameters = {'hidden_layer_sizes':[10, 15, 20, 25, 30, 35, 40, 45,
50],
    ...:                     'activation': ['logistic', 'tanh','relu']}
```

Fig 9.23 (click to enlarge)

Next, you need to create the neural network models and create and train the optimizer shown in Figure 9.24:

```
In [72]: nn_optimizer.fit(X_train_scaled, y_train_scaled)
Out[72]:
GridSearchCV(cv=10, error_score='raise',
       estimator=MLPRegressor(activation='relu', alpha=0.0001, batch_size='auto',
beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(100,), learning_rate='constant',
       learning_rate_init=0.001, max_iter=10000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=21, shuffle=True,
       solver='adam', tol=0.0001, validation_fraction=0.1, verbose=False,
       warm_start=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'hidden_layer_sizes': [10, 15, 20, 25, 30, 35, 40, 45, 50],
'activation': ['logistic', 'tanh', 'relu']},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
       scoring='neg_mean_squared_error', verbose=0)
```

Fig 9.24 (click to enlarge)

Please note that the chosen values for *max_iter* and *tol* parameters are only for a quick calculation. Please choose larger *max_iter* or smaller *tol* to make the model fully converge.

The best parameters and models are available from the *best_params_* and *best_estimator_* attributes shown in Figure 9.25:

```
In [73]: nn_optimizer.best_params_
Out[73]: {'activation': 'logistic', 'hidden_layer_sizes': 40}

In [74]: nn_optimizer.best_estimator_
Out[74]:
MLPRegressor(activation='logistic', alpha=0.0001, batch_size='auto',
       beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=40, learning_rate='constant',
       learning_rate_init=0.001, max_iter=10000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=21, shuffle=True,
       solver='adam', tol=0.0001, validation_fraction=0.1, verbose=False,
       warm_start=False)
```

Fig 9.25 (click to enlarge)

You can convert all results into a DataFrame and double click “results” in the Variable explorer window to view it (shown in Figure 9.26 A & B):

```
In [75]: results = pd.DataFrame(nn_optimizer.cv_results_)[['param_activation',
    ...:                         'param_hidden_layer_sizes', 'mean_test_score',
    ...:                         'std_test_score', 'rank_test_score']].round(2)
```

Now, you can use the best model to fit with train data and evaluate the model with both train and test datasets:

```
In [76]: nn_best_model = nn_optimizer.best_estimator_

In [77]: nn_best_model.fit(X_train_scaled, y_train_scaled)
Out[77]:
MLPRegressor(activation='logistic', alpha=0.0001, batch_size='auto',
       beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=40, learning_rate='constant',
       learning_rate_init=0.001, max_iter=10000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=21, shuffle=True,
       solver='adam', tol=0.0001, validation_fraction=0.1, verbose=False,
       warm_start=False)

In [78]: nn_train_pred_best = nn_model.predict(X_train_scaled)
    ...: metrics.r2_score(y_train_scaled, nn_train_pred_best)
Out[78]: 0.8605519867793164

In [79]: metrics.mean_squared_error(y_train_scaled, nn_train_pred_best)
Out[79]: 0.1394480132206836

In [80]: nn_test_pred_best = nn_best_model.predict(X_test_scaled)
    ...: metrics.r2_score(y_test_scaled, nn_test_pred_best)
Out[80]: 0.577361601867875

In [81]: metrics.mean_squared_error(nn_test_pred_best , nn_test_pred)
Out[81]: 0.008595038690324335
```

Fig 9.26 (A) (click to enlarge)

Transcript

In this video, I will introduce how to use neural networks for regression tasks.

For Neural networks, preprocessing data is often needed. First, you need to convert categorical variable to dummy variables. These have been introduced in Lesson 4. You can create dummy variables for each categorical column and join them to the original data.

Or you can convert the these variable to categorical first by using astype('category'), then you can apply get_dummies function on the entire data sets and it will transfer all categorical variable at once, as shown in Line 131 to 133.

Next, you need to use Minmaxscaler to rescale all variables to the range [0, 1]. This has been introduced in the previous lesson. We split the data into train, test first, rescale it, then separated into X, y such that we only need to fit the data once.

MLPRegressor is the regression model. It is under sklearn dot neural network, you need to import neural_network first.

Here we initiate the MLPregressor with 10 hidden layer nodes, activation function is logistic. Max interaction is 10000, and random state is 21.

Next, you need to fit the model with X_train, y_train.

You can view the coefficient and intercept by its attribute coef_ and intercepts_

Now, you can evaluate the train and test data. You can predict mpg value for train and test sets, then use R squared and mean squared error to compare them. R square is 0.77 for train data and 0.59 for test data, which is an indication of overfitting.

Next, we show how to use gridSearchCV function to optimize parameters with cross-validation.

First, you need to import this function from sklearn.model_selection.

Next, you need to use dictionary to present the parameters you want to optimize. The parameter names should be the keys and the values you want to try for each parameter will be the values for the dictionary.

Then, you need to initiate the MLPRegression. You can specify other parameters here. Here we use max iteration equals 10000, and random state is 21.

You also need to initiate the gridsearchcv object, in the function, you need to show model name, turned parameters, and specify which scoring method you use, here we use negative mean square error. Cv equals 10, 10 fold cross-validation, return _train_score is False. And verbose = 0 means that I don't want to see the report of the process. Chang to a positive integer, it will show the process.

Now, you can fit the model with your X_train_scaled and y_train_scaled.

You can use attributes to get access to best parameter, best estimator.

Here, we only use the train data for model optimization. Once you select the best model, you can fit with all train data and evaluate the train and test data.

From the results, we can see that the performance on test sets has been improved. However, the best hidden_layer_sizes is 50 which is the largest number I tried. You should increase the hidden layer size in the turned parameters and rerun everything to see if the model can be further improved.

Lesson 9 References

- http://scikit-learn.org/stable/supervised_learning.html#supervised-learning (http://scikit-learn.org/stable/supervised_learning.html#supervised-learning)
- Python Data Science Handbook, Jake VanderPlas, https://jakevdp.github.io/PythonDataScienceHandbook/index.html (https://jakevdp.github.io/PythonDataScienceHandbook/index.html)