Regression with *sklearn* Machine Learning

<div id="author""> David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

Preliminaries

Misc

In [1]:
```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2023-09-25 at 17:59
```

In [2]:
```
import os
```

In [2]:
```
os.getcwd()
```

Out[2]:
```
'/Users/davidgerbing/Documents/000/575/0Templates'
```

Import Standard Data Analysis Libraries

In [3]:
```
import pandas as pd
import numpy as np
import seaborn as sns
```

Data

Boston Housing Data Set

- *crim*: per capita crime rate by town
- *zn*: proportion of residential land zoned for lots over 25,000 sq.ft.
- *indus*: proportion of non-retail business acres per town.
- *chas*: Charles River dummy variable (1 if tract bounds river; 0 otherwise)
- *nox*: nitric oxides concentration (parts per 10 million)
- *rm*: average number of rooms per dwelling
- *age*: proportion of owner-occupied units built prior to 1940
- *dis*: weighted distances to five Boston employment centres
- *rad*: index of accessibility to radial highways
- *tax*: full-value property-tax rate per 10,000 USD
- *ptratio*: pupil-teacher ratio by town
- *b*: 1000(Bk*: 0.63)^2 where Bk is the proportion of blacks by town
- *lstat*: % lower status of the population
- *medv*: Median value of owner-occupied homes in 1000's USD

In [4]:
```
d = pd.read_csv('data/Boston.csv')
#d = pd.read_csv('http://web.pdx.edu/~gerbing/data/Boston.csv')
```

In [5]:
```
d.shape
```

Out[5]:
```
(506, 15)
```

In [6]:
```
d.head()
```

Out[6]:

	Unnamed: 0	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	bl
0	1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	39(
1	2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	39(
2	3	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	39:
3	4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	39.
4	5	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	39(

Do not need the first column, so drop

Do not need the first column, so drop.

In [7]:
```
d = d.drop(['Unnamed: 0'], axis="columns")
d.head()
```

Out[7]:

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33

Check for missing data to determine if any action such as row or column deletion or any data imputation is needed. The `\n` instructs to insert a new line in the output.

In [ ]:
```
print(d.isna().sum())
print('\nTotal Missing:', d.isna().sum().sum())
```

No missing data.

Data Exploration

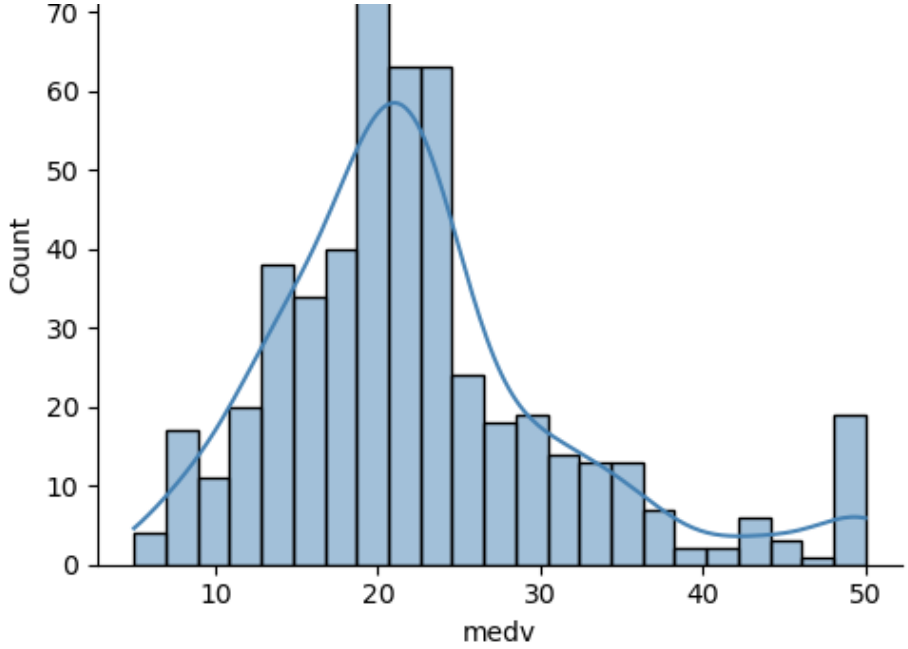
Distribution of the Target Variable

Before constructing and analyzing a model that forecasts the target value from the values of the predictor variables, the features, gain some understanding of the target variable. The primary purpose is to view the distribution of the target variable, not necessarily to show normality per se. Look for skewness, outliers, and data anomalies in general.

Check out the distribution of the target with seaborn `displot()`, what `seaborn` calls a *figure-level* function. Set parameter `kde` to `True` to show the smoothed summary of the distribution's shape, called a *density plot*. Set the figure size with the `height` and `aspect` parameters. The `aspect` is the ratio of the width to the height of the plot.

In [8]:
```
sns.displot(d.medv, kde=True, color='steelblue', height=4, aspect=1.25)
```

Out[8]:
```
<seaborn.axisgrid.FacetGrid at 0x13eb83b10>
```



The distribution of the target variable *medv* is more or less normal, except some large values beyond normality. It appears that all prices more than 50,000 USD are truncated to 50,000 USD for some reason.

Feature Relevance

Examine the *relevance* of each feature according to its correlation with the target. Use `pandas` function `corr()` to calculate just the correlations of the variables with `medv`. Use `pandas` function `sort_values()` to sort from smallest to largest. Correlations of large magnitude, regardless of sign, indicate relevance, a strong relation between a feature (predictor) and the variable to be predicted, the target.

Feature *chas* appears the least relevant with a correlation of the target of only 0.18. Even so, not 0, so with the small data set, will retain for the initial model analysis.

The analysis reveals that the most relevant features are *lstat* and *rm*.

In [9]:
```
(d
 .corr()['medv']
 .sort_values()
 .round(2)
)
```

Out[9]:
```
lstat     -0.74
ptratio   -0.51
indus     -0.48
tax       -0.47
nox       -0.43
crim      -0.39
rad       -0.38
age       -0.38
```

```
chas       0.18
dis        0.25
black      0.33
zn         0.36
rm         0.70
medv       1.00
Name: medv, dtype: float64
```

The data and the corresponding analysis upon which a model is trained must have no influence on the model's performance regarding the training data.

> *Data leakage*: Information regarding the data used to train the model becomes available to the testing data used to assess the model's predictive efficiency.

When applying the model to real world prediction, the extra information used to train the model would no longer be available to the testing data.

Avoid data leakage. Leaking information across training and testing data sets artificially increases model fit beyond what would be obtained in real world prediction. For example, first standardizing *all* of the data and then selecting the training and testing data sets follows from information gathered on *all* the data, so the testing data are not completely isolated from the data on which the model was estimated. Always test the final proposed forecasting model on data that has not in any way been used to estimate the model.

In practice, however, you might need to reduce computation time if you have a huge data set and a model with many predictors, particularly with a more complicated model and solution algorithm than for linear regression. In that situation, without doing any model estimation, perhaps eliminate some features that violate the two properties of *relevance* and *uniqueness* before model estimation.

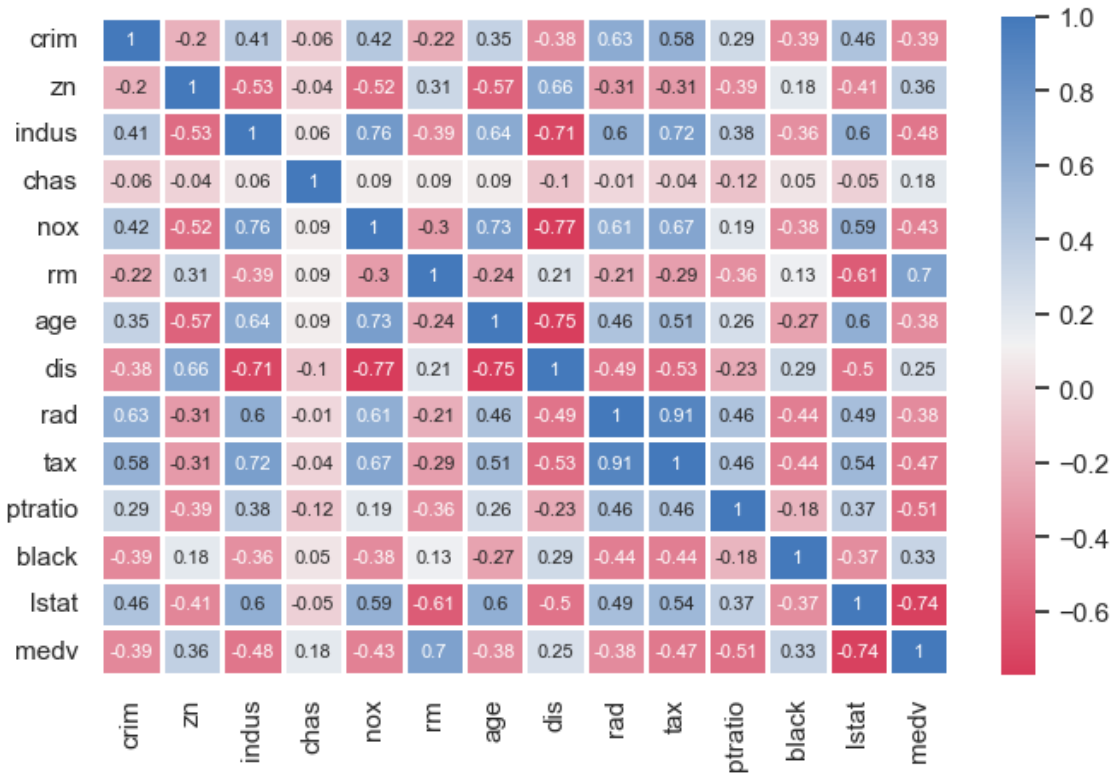
Feature Uniqueness

Features, predictor variables, are valuable for increasing prediction that are both *relevant* and *unique*. To explore the relations among all the variables in the model so as to help identify redundant features, obtain an illustrated correlation matrix that shows all pair-wise correlations called a *heat map*. Obtain the heat map with the `seaborn` function `heatmap()`. By default, the darker the blue shading, the stronger the positive correlation. The darker the red shading, the stronger the negative correlation. To specify the size of the displayed correlation coefficients, use the `annot_kws()` function, as below, with `heatmap()`. The `cmap` parameter specifies the displayed palette to diverge across two colors with the state intensities.

Note: Some `seaborn` functions specify the dimensions of a plot with parameters `height` and `aspect`, as with the previous `displot()` function. Other plotting functions require a separate line of code with the `seaborn` function `set()` and a corresponding verbose expression for the `rc` parameter. This distinction involves what are called axis-level functions versus the generally preferred figure-level functions, a distinction that adds yet another level of complexity. Figure-level functions are generally preferred but not always available, so `seaborn` users generally access both types. The following `heatmap()` function is an axis-level function, the only available function for this task. All of this is more confusing and awkward than it should be.

In [10]:
```
sns.set(rc={"figure.figsize":(8, 5)})
sns.heatmap(d.corr().round(2), linewidths=2.0,
            annot=True, annot_kws={"size": 8},
            cmap=sns.diverging_palette(5, 250, as_cmap=True))
```

Out[10]:
```
<Axes: >
```



Many features are redundant with other features, so the final model would likely not contain all the features in the initially specified model. For example, *nox* correlates 0.76 with *indus*. The information inherent in one variable pretty much overlaps with the information in the other variable regarding the prediction of the target, *medv*.

Create Feature and Target Data Structures

Store the features, the predictor variables, in data structure X. Store the target variable in data structure y.

To run multiple regression with all possible predictor variables in this data set, define X as the entire data frame with the target variable *medv* dropped:

```
X = d.drop(['medv'], axis="columns")
```

Or, use the procedure below that manually defines a vector of the predictor variables (features) names, and then define X as the subset of *d* that contains just these variables.

In [11]:
```
y = d['medv']
pred_vars = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad'
             'tax',  'ptratio', 'black', 'lstat']
X = d[pred_vars]
```

Useful to see how many features define the model with the Python `len()` function for length. Also observe the data type of the X and y data structures with the Python `type()` function. Because these functions are part of the original Python language, no package prefix is needed, just the respective function name.

In [12]:
```
n_pred = len(pred_vars)
print("Number of predictor variables:", n_pred)
```

```
Number of predictor variables: 13
```

In [13]:
```
print("X: ", type(X))
print("y: ", type(y))
```

```
X:  <class 'pandas.core.frame.DataFrame'>
y:  <class 'pandas.core.series.Series'>
```

Model Validation with One Hold-Out Sample

Now for Python machine learning!

Access Solution Algorithm

The `sklearn` package abbreviates the full name `scikit-learn`, which, in turn, abbreviates `scientific toolkit for machine learning`. `sklearn` provides "Simple and efficient tools for predictive data analysis", a primary reason Python has become the leading platform for machine learning.

The `sklearn` package provides many different solution algorithms to accommodate many different types of machine learning models. Each solution type is presented in its own module called a *class*. To the huge advantage of `sklearn`, all solution methods follow the same programming form. Moving from one machine learning technique to another is straightforward with relatively small changes in the code.

The `sklearn` class `LinearRegression` provides the functions for doing linear regression. Access the computer code for an algorithm by creating a specific instance of the algorithm, referred to by a specific name in the analysis. This process is called *instantiation*.

Instantiate a module with any valid Python expression. In this example, instantiate `LinearRegression()` with the name *reg*, accepting all default parameters, not passing any parameter values between the parentheses. All subsequent references to the linear regression algorithm below are then implemented via this assigned name *reg*.

In [14]:
```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
```

Split Data into Train and Test Sets

Cross-validation tests a model on a new data set, *testing data* different from the data on which the model was estimated, *training data*. Usually the data sets are random samples obtained from splitting the original data table into training and testing data.

The concept of cross-validation has applied to regression analysis for many decades, though, due to smaller historical data sets, perhaps often recommended more than actually accomplished. The machine learning framework provides for easily accessible cross-validation methods, considered a necessary component of the analysis.

Cross-validation can take place for one split of the data or over multiple, different splits.

> If computational time is available, accomplish cross-validation with multiple splits of the original data, usually three to five splits.

However, this discussion begins with a single split of the data.

The `sklearn` function `train_test_split()`, from the `model_selection` module, randomly shuffles the original data into two sets, training data and testing data, here called *X_train* and *X_test* for the features and *y_train* and *y_test* for the target.

- Parameter `test_size` specifies the percentage of the original data set allocated

to the test split.
- Parameter `random_state` specifies the initial seed (or starting point) from which the process of number generation begins so that the sequence can be repeated.

The input into the `train_test_split()` function are the X and y data structures. The function provides four outputs from a single function call: X training and testing data, and y training and testing data. Python has the convention of listing the names for multiple outputs on the left side of the equals sign, separated by commas, in the correct order in which the function lists the output.

The generated "random" numbers that result in each row of data either assigned to the training data or the testing data are more properly called `pseudo-random numbers`. Optional parameter `random_state` specifies the initial seed that initiates the generation of each set of pseudo-random numbers. Specifying the same seed for each time the function is run repeats the same pseudo-random process. That way, the same analysis can be repeated at some future time with the same data split is obtained from `train_test_split()`. If no seed is specified then each time the `train_test_split()` function is run the data will be split differently.

In this example, `random_state` is set at 7. Each time the `train_test_split()` function is run with the value of 7, the same sets of training and testing data will be obtained.

In [15]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=.25, random_st
```

The `shape` method displays the dimensions of each of the resulting two data sets, *X_train* and *X_test*. The first number is the number of rows in the corresponding data structure. Here with the size of the testing data set at 25% of all the data, there are 379 rows of data in the two training data structures and 127 rows of data in the two testing data structures. The y data structures have only one column. The y structures are not data frames, so their number of columns is not specified.

In [16]:
```
print("Size of X data structures: ", X_train.shape, X_test.shape)
print("Size of y data structures: ", y_train.shape, y_test.shape)
```

```
Size of X data structures:  (379, 13) (127, 13)
Size of y data structures:  (379,) (127,)
```

Estimate Model Parameters

The primary Python package for machine learning is `sklearn` (more formally, scikit-learn). The `sklearn` machine learning function to estimate the values of the specified model's parameters from the data is `fit()`.

> What the machine learns is the numerical value of each weight that is applied to each predictor variable or feature, plus the y-intercept.

The `sklearn` function for any machine learning algorithm is `fit()`. Apply the

`fit()` function for linear regression by applying the function to our *reg* instantiation of `LinearRegression`.

In [17]:
```
reg.fit(X_train, y_train)
```

Out[17]:
```
▾ LinearRegression
LinearRegression()
```

The `fit()` function creates several different data structures as output, each structure stored with a pre-defined name. The name of a data structure whose values that the analysis procedure creates ends in an underline. The estimated model coefficients are stored in the `intercept_` and `coef_` structures. To reference, precede each structure name with the model's name and a period, such as `reg.coef_`.

This machine learning implementation of regression is typically not primarily directed towards understanding and interpreting the model coefficients. Instead, the focus is on evaluating the extent of prediction error. The estimated coefficients are not even displayed by default. The analysis does not provide the usual regression model output with the coefficients listed along with their corresponding t-tests of the null hypothesis of 0, and the associated confidence interval, such as obtained from the `OLS()` function in the `statsmodels` package. Still, the estimated weights that the machine learned are in the corresponding `coef_` output.

The corresponding `fit()` output structures are not `pandas` data frames, but rather `numpy` arrays, which do not display as nicely. To make the output more readable, convert the `numpy` array output format to a `pandas` data frame.

In [18]:
```
reg.coef_
```

Out[18]:
```
array([-1.29372986e-01,  2.95904870e-02,  2.22928425e-02,  2.83744579e+00,
       -1.53954203e+01,  5.27557273e+00, -1.05383841e-02, -1.30170765e+00,
        2.66392896e-01, -1.09686702e-02, -9.64830193e-01,  1.08603361e-02,
       -3.78363465e-01])
```

In [19]:
```
X.columns
```

Out[19]:
```
Index(['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'ta
x',
       'ptratio', 'black', 'lstat'],
      dtype='object')
```

The following listing of the estimated coefficients shows what the machine learned. In the `print()` function, the `%.3f` is a format that indicates to display a floating-point number, that is, one with decimal digits, and to display three decimal digits. The `\n` specifies to skip an output line.

In [20]:
```
print(f'Intercept: {reg.intercept_:.3f}', '\n')

cdf = pd.DataFrame(reg.coef_, X.columns, columns=['Coefficients'])
print(cdf.round(3))
```

```
Intercept: 23.957
```

```
         Coefficients
crim           -0.129
zn              0.030
indus           0.022
chas            2.837
nox           -15.395
rm              5.276
age            -0.011
dis            -1.302
rad             0.266
tax            -0.011
ptratio        -0.965
black           0.011
lstat          -0.378
```

Calculate y^

Given the estimated model, generate predictions from given values of the X variables, the features. The standard `sklearn` function to calculate a fitted value from the estimated model is `predict()`, again applicable to any machine learning method.

Compute two sets of $\hat{y}$ values: *y_fit* when the model is applied (fitted) to the data on which it trained, and, for model evaluation, *y_pred* when the model is applied to the test data.

In [21]:
```
y_fit = reg.predict(X_train)
y_pred = reg.predict(X_test)
```

> Evaluate the descriptive analysis of fit by comparing y to $\hat{y}$ for the training data, y compared to y_fit.
>
> Evaluate true predictive fit by comparing y to $\hat{y}$ for the testing data, y compared to y_pred.

$\hat{y}$ is always the value of y fitted by the model but this is a prediction only when the value of y is unknown by the model, which is only true for the testing data.

As always, the assessment of fit is *not* based on the training data in which the actual value of *y* is already known, but on **new** data for which the model is not aware of the actual value of *y*. Have the model generate its predicted value, $\hat{y}$, then evaluate how close the values of *y* tend to be to corresponding values of $\hat{y}$, $y_i - \hat{y}_i$.

Assess Fit

Visual Assessment of Fit

If there is only one predictor variable, plot the scatter plot of X and y and the least-squares regression line through the scatterplot. If this multiple regression, then this code

is not run.

The Python syntax for an `if` statement uses the double equal sign, ==, to evaluate the equality, and a single equal sign, =, to create equality by assigning the value on the right to the variable on the left. Indicate the end of the conditional statement, here n_pred==1, with a colon, : . Indent two spaces for the statements that are run if the conditional statement is true.

In [22]:
```
if n_pred == 1:
    sns.regplot(x=X_train, y=y_train, color='steelblue')
```

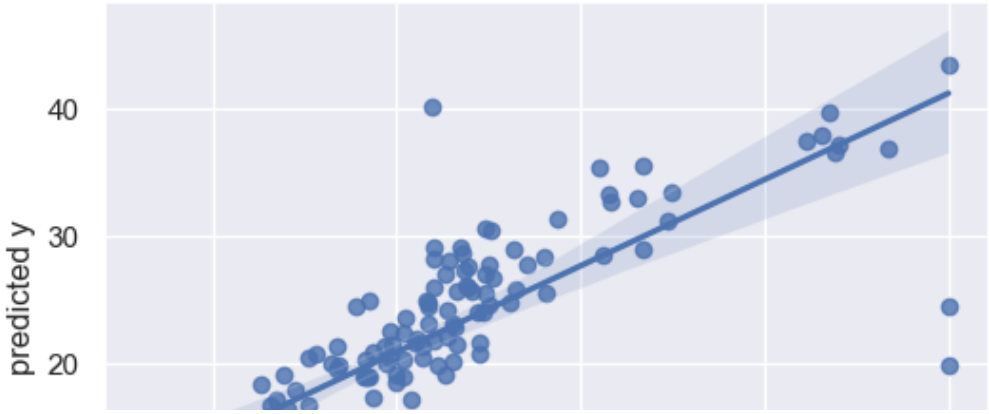
Assess the model by comparing the actual data values of y in the testing data, *y_test*, to the values of y calculated from the estimated model, *y_pred*.

What to plot if there is more than a single feature, the usual case? Visualize the overall fit by plotting the actual values of y in the test data, *y_test*, with the corresponding values of the forecasted y's, $\hat{y}$, or *y_pred*. If the forecasting is perfect, then $y = \hat{y}$, and all points lie on the 45-degree line through the origin.

To obtain a scatter plot with the regression line and associated confidence interval, use the `seaborn` function `regplot()`. The variables to be plotted are not in a data frame, so there is no `data` parameter. To label the axes, use the `set()` function with the name applied from the output of `regplot()`. In this example, the output of `regplot()` was set `ax` for axis.

In [23]:
```
sns.set(rc={"figure.figsize":(6, 4)})
ax = sns.regplot(x=y_test, y=y_pred)
ax.set(xlabel='y from testing data', ylabel='predicted y')
```

Out[23]:
```
[Text(0.5, 0, 'y from testing data'), Text(0, 0.5, 'predicted y')]
```



We can see that the predicted values closely match with the actual data values from the testing data.

Fit Metrics

A large drop of fit from training to testing model indicates *overfitting* the model to the training data. To evaluate the fit of the model to the training data, compare the actual data values, *y_train*, to the corresponding values computed by the model, *y_fit*.

The `metrics` module in the `sklearn` package provides the computations for the fit indices. The module provides the mean squared error, MSE, and R^2 fit indices with the functions `mean_squared_error()` and `r2_score()`. To get the standard deviation of the residuals, manually take the square root of the variance MSE with the `numpy` function `sqrt()`.

The %.3f formatting code instructs the Python `print()` function to print a floating-point number (numeric with decimal digits) with three decimal digits.

In [24]:
```
from sklearn.metrics import mean_squared_error, r2_score
mse = mean_squared_error(y_train, y_fit)
rsq = r2_score(y_train, y_fit)
print("MSE: %.3f" % mse)
se = np.sqrt(mse)
range95 = 4 * se
print("Stdev of residuals: %.3f " % se)
print("Approximate 95 per cent range of residuals: %.3f " % range95)
print("R-squared: %.3f" % rsq)
```

```
MSE: 20.266
Stdev of residuals: 4.502
Approximate 95 per cent range of residuals: 18.007
R-squared: 0.767
```

For pedagogy, here compute the standard deviation of the residuals from the data. Define the residuals as the variable *e*. Note that the mean squared residual, both here and from the previous cell, is calculated with the full sample size, not the technically correct degrees of freedom.

In [25]:
```
e = y_train - y_fit
print("stdev of residuals: %.3f " % np.sqrt(np.mean(e**2)))
```

```
stdev of residuals: 4.502
```

Now perform the actual evaluation of model performance. Evaluate how well the actual data values for y, y_test, match the forecasted or predicted values of y, $\hat{y}$. From this split of data, the value of R^2 typically drops from that obtained from the training data.

Sometimes, however, by chance, the testing data may outperform the training data, again due to chance.

In [26]:
```
mse_f = mean_squared_error(y_test, y_pred)
rsq_f = r2_score(y_test, y_pred)
print('Forecasting Mean squared error: %.3f' % mse_f)
print('Forecasting Standard deviation of residuals: %.3f' % np.sqrt(mse_f))
print('Forecasting R-squared: %.3f' % rsq_f)
```

```
Forecasting Mean squared error: 29.515
Forecasting Standard deviation of residuals: 5.433
Forecasting R-squared: 0.617
```

We see that when applied to new data, the standard deviation of residuals, s_e, increased from 4.502 to 5.433, still a small number. R^2 decreased from 0.767 from the training data to 0.617 applying the model to the testing data. Regardless, good fit is obtained even with the forecasting model.

Model Validation with Multiple Hold-Out Samples

As a generalization to the one hold-out cross-validation described in the previous section, pursue the usually more desirable procedure that assesses model fit with cross-validation on *multiple* samples. The `sklearn` `model_selection` module provides the functions for the cross-validation in which the model for each fold is estimated using the remaining $k - 1$ folds and then tested on that one remaining fold. The process automatically repeats for each fold.

The `Kfold()` class is instantiated, just as is the `LinearRegression()` class. Also needed is the `cross-validate()` function.

In [27]:
```
from sklearn.model_selection import KFold, cross_validate
```

Evaluate Fit on Testing Data

Generate the folds with `KFold()` given the following parameters.

- *n_splits*: Number of splits (folds) of the training data.
- *shuffle*: Randomly shuffle the data before splitting into the folds.
- *random_state*: Set the seed to recover the same "random" data set in a future analysis.

The number of splits, folds, can vary from 2 to $n - 1$, where n is the total number of rows in the training data. Values of 3 and 5 are the most common number of folds. Larger data sets support a larger number of splits. Usually, shuffle the data before forming the folds to keep the entire process entirely random.

In this example, instantiate the `KFold` class with the chosen name *kf*, invoking the desired parameter values. Choose whatever valid name you wish.

In [28]:
```
from sklearn.model_selection import KFold, cross_validate
kf = KFold(n_splits=5, shuffle=True, random_state=1)
```

The `cross_validate()` function performs the cross-validation, randomly generating the specified number of folds and providing multiple evaluation scores from each fold. The function also provides computation times.

Estimate five different models from five different samples. We have already instantiated the `LinearRegression()` estimator earlier as *reg*, also referenced for the cross-validation. The *scoring* parameter specifies to obtain R^2 and MSE scores for each of the true forecasts of applying the model, for each split, from the k-1 folds data to the hold-out fold.

Assess model fit two ways.

1. *Primary assessment of fit*: Fit of the model estimated from training data to test data.
2. *Overfitting*: Comparison of the fit of the training model to the fit of the model on the test data.

Training scores much larger than the related testing scores indicate overfitting, in which the model does well for the data in which it trained but fails to generalize to new data. Obtain the training fit information with the parameter *return_train_score*.

Here, name the output of `cross_validate()` as *scores*, a `numpy` array.

In [29]:
```
scores = cross_validate(reg, X, y, cv=kf,
                        scoring=('r2', 'neg_mean_squared_error'),
                        return_train_score=True)
```

The smaller is MSE, the smaller the residuals, the better the fit. For example, a 0.5 MSE indicates better fit than a 0.7 MSE. However, to maintain consistency with other fit indices, the best fit score should be the largest across all scoring procedures and estimation algorithms, which is the general rule recognized by the related `sklearn` functions. Accordingly, although the MSE statistic is necessarily a positive value, it is reported as a negative number. For example, -0.5 > -0.7, so the largest of the two negative values is the most desirable value, here -0.5. Of course, as the mean squared error, MSE must be a non-negative number, so the sign of the actual MSE is just flipped to negative for the `sklearn` assessment of fit.

Assess Fit

Our *scores* array contains much information regarding the fit of each model over the five different analyses, but is not so directly readable. To make it more readable, convert *scores* to a data frame, rename the long column names to more compact versions, convert the MSE scores to positive numbers, and average the results. The display includes the time to fit the training data for each fold and the time to calculate the evaluation scores, which includes getting the predicted values.

Setting the parameter `inplace` to `True` changes the specified data frame and saves the data frame with those changes. This parameter setting removes the need to copy to a new data frame.

In [30]:
```
ds = pd.DataFrame(scores)
ds.rename(columns = {'test_neg_mean_squared_error': 'test_MSE',
                     'train_neg_mean_squared_error': 'train_MSE'},
          inplace=True)

ds['test_MSE'] = -ds['test_MSE']
ds['train_MSE'] = -ds['train_MSE']
print(ds.round(4))
```

```
   fit_time  score_time  test_r2  train_r2  test_MSE  train_MSE
0    0.0021      0.0012   0.7634    0.7294   23.3808    21.8628
1    0.0013      0.0007   0.6468    0.7582   28.6143    20.5029
2    0.0009      0.0005   0.7921    0.7262   15.1606    23.7937
3    0.0012      0.0005   0.6508    0.7580   27.2082    20.8185
4    0.0009      0.0005   0.7353    0.7409   23.3712    21.6071
```

A fit index averaged over all the folds is the best summary of how well the model fits, either to the training data, or more interestingly, to the testing data.

In [31]:
```
m_r2 = ds['test_r2'].mean()
print('Mean of test R-squared scores: ' +  f'{m_r2:.3f}')

m_MSE = ds['test_MSE'].mean()
print('\nMean of test MSE scores: ' + f'{m_MSE:.3f}')

m_se = np.sqrt(ds['test_MSE'].mean())
print('Mean standard deviation of test MSE scores: '  + f'{m_se:.3f}')
```

```
Mean of test R-squared scores: 0.718

Mean of test MSE scores: 23.547
Mean standard deviation of test MSE scores: 4.853
```

The mean of the R^2 scores across the test samples (folds) is the summary indicator of model fit. This 13-predictor model fits well, with an average R^2 across the five folds of 0.72. (Note that we never see the actual estimated model from each fold.) The average MSE and s_e is also low in terms of the more interpretable standard deviation of the residuals. Once the model is validated, fit it to the entire, full data set.

Strategy to Obtain the Final Model

Begin data preparation by deleting any unnecessary features, removing any obvious

univariate outliers, and converting any categorical variables to indicator/dummy variables if included in the model as features. Also check for missing data as machine learning solution algorithms do not run if missing data are present.

If CPU time is an issue, cross-validate with only one hold-out sample. Otherwise, cross-validate with 3 or 5 or more hold-out samples, depending on CPU time and the size of the original data set.

> If computation time permits, evaluate the model with the mean chose fit index over the the k-fold cross-validations.

The only advantage of the one train-test split approach is that the model coefficients can be obtained, but they are not of primary interest because the final model has not yet been estimated on all of the data. Cross-validation with k-fold does what the one train-test split approach does, but now k times. The train-test one split approach almost becomes pedagogical as a way to learn how the k-fold procedure works.

The initial model is usually pared down to a more parsimonious model, retaining a smaller set of relevant features that each provide unique information. Obvious candidates for features to delete can be deleted before model validation begins, that is, those with low correlations with the target and/or high correlations with other features.

More sophisticated feature deletion can occur after the model if the model is validated. Then use the `statsmodels` regression function `OLS()` for ordinary least squares to estimate the model on all of the data to get the estimated model on the largest sample possible. Do a more sophisticated feature selection procedure using your own judgement, based on p-values for individual features and VIF values to assess the collinearity of individual features. Also, use Cook's distance to investigate and possibly eliminate any rows of data that are outliers with respect to the regression model.

The coefficients of the final, validated model, estimated with all of the data, are needed to apply the model to other situations. Once a final model is selected, re-run the cross-validation on the smaller number of features to make sure the reduced model still evaluates well. Ideally, this analysis would be done on a completely new data set, but that may not be practical.

When completed, with the final `statsmodels` run you have the b coefficients -- b_0, b_1, b_2, etc. -- that define the model that you now, in another context, put into production.

In [ ]: