Regression with *sklearn* Machine Learning

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

Preliminaries

Misc

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

    Analysis on 2021-07-12 at 13:49

import os
os.getcwd()

    '/content'
```

Import Standard Data Analysis Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Access Solution Algorithm

The `sklearn` package provides many different solution algorithms to accommodate many different types of machine learning models, each in its own module called a class. The `LinearRegression` module provides the functions for doing linear regression. Access an algorithm by creating a specific instance of the algorithm, refered to by a specific name in the analysis. This process is called *instantiation*.

Here instantiate `LinearRegression()` with the name *reg_model*, 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 name *reg_model*.

```
from sklearn.linear_model import LinearRegression
reg_model = LinearRegression()
```

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

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

d.shape

    (506, 15)

d.head()
```

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

Do not need the first column, so drop.

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

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lst
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.

Check for missing data to determine if any action such as row or column deletion or any data imputation is needed.

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

    crim       0
    zn         0
    indus      0
    chas       0
    nox        0
    rm         0
    age        0
    dis        0
    rad        0
    tax        0
    ptratio    0
    black      0
    lstat      0
    medv       0
    dtype: int64

    Total Missing: 0
```

No missing data.

Check out the distribution of the target with seaborn `displot()`. Set parameter `kde` to `True` to show the smoothed summary, called a *density plot*. If going to run a model focused on forecasting the target, one should also understand the nature of the target variable. The main purpose is to understand the distribution, not necessarily to show normality per se. Look for skewness, outliers, etc.

Data Exploration

```
plt.figure(figsize=(6,6))
sns.displot(d.medv, kde=True, color='steelblue')

    <seaborn.axisgrid.FacetGrid at 0x7f8385b12e50>
    <Figure size 432x432 with 0 Axes>
```

More or less normal, with some large values beyond normality. It appears prices more than 50,000 USD are truncated to 50,000 USD for some reason.

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 function `sort_values()` to sort from smallest to largest. Correlations of large magnitude, regardless of sign, indicate relevance.

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.

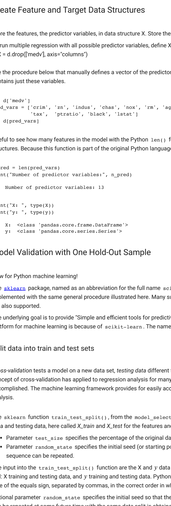
The most relevant features are *lstat* and *rm*.

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

    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
```

> *Data leakage*: Feedback from data analysis from which the model is trained is used to evaluate the model used for forecasting.

We need to avoid data leakage. 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.

```
plt.figure(figsize=(12,10))
sns.heatmap(d.corr().round(2), linewidths=2.0, annot=True,
            cmap=sns.diverging_palette(h_neg=240, h_pos=0, as_cmap=True))

    <matplotlib.axes._subplots.AxesSubplot at 0x7f8381d45fd0>
```



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, define X as the entire data frame with *medv* dropped, as in

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.

```
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 in the model with the Python `len()` function for length, and observe the data type of the X and y data structures. Because this function is part of the original Python language, no package prefix is needed, just the function name.

```
n_pred = len(pred_vars)
print("Number of predictor variables:", n_pred)

    Number of predictor variables: 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!

The `sklearn` package, named as an abbreviation for the full name `scikit-learn`, provides many machine learning algorithms, all implemented with the same general procedure illustrated here. Many support functions such as dividing data into training and testing partitions are also supported.

The underlying goal is to provide "Simple and efficient tools for predictive data analysis". A primary reason Python has become the leading platform for machine learning is because of `scikit-learn`. The name `scikit-learn` is a scientific toolkit for machine learning.

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*. The concept of cross-validation has applied to regression analysis for many decades, though perhaps often recommended more than actually accomplished. The machine learning framework provides for easily accessible cross-validation methods, and is a necessary component of the analysis.

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.

Optional parameter `random_state` specifies the initial seed so that the same random process, more properly called a pseudo-random process, can be repeated at some future time with the same data split is obtained from `train_test_split()`. Re-run with the same value, here 7, obtain the same random split.

```
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_state=7)
```

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.

```
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 the model parameters

All `sklearn` solution algorithms fit the model, that is, estimate the model parameters, with the `fit()` function, presumably first applied only to the training data. Expressed yet another way, the machine (i.e., algorithm implemented on the machine) learns from the training data. Only use the training data at this point in the analysis.

Here apply the `fit()` function for linear regression by applying our *reg_model* instantiation of `LinearRegression`.

```
reg_model.fit(X_train, y_train)
```

```
    LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

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 name, in this example, with the model's name and a period. The coefficients of the final, validated model, estimated with all of the data, are needed to apply the model to other situations.

The machine learning implementation of regression is typically not primarily directed towards understanding and interpreting the model coefficients. Instead, focus on evaluating the extent of forecasting 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.

The corresponding 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 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.

```
print("intercept: %.3f" % (reg_model.intercept_), "\n")

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

```
    intercept: 23.957

              Coefficients
    crim         -0.129373
    zn            0.029590
    indus         0.022293
    chas          2.837446
    nox         -15.395420
    rm            5.275573
    age          -0.010538
    dis          -1.301708
    rad           0.266393
    tax          -0.010969
    ptratio      -0.964830
    black         0.010860
    lstat        -0.378363
```

Calculate $\hat{y}$

Given the estimated model, generate forecasts. The standard `sklearn` function to calculate a fitted value from the estimated model is `predict()`.

Here 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.

```
y_fit = reg_model.predict(X_train)
y_pred = reg_model.predict(X_test)
```

> Evaluate the descriptive analysis of fit by comparing y to $\hat{y}$ for the training data.

> Evaluate true forecasting fit by comparing y to $\hat{y}$ for the testing data.

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.

```
if n_pred == 1:
  plt.scatter(X_train, y_train, color='gray')
  plt.plot(X_train, y_fit, color='black', linewidth=2)
  plt.xlabel("Prices: $X_i$")
  plt.title("Y and Fitted $\hat{Y}_i$ Plotted Against X")
```

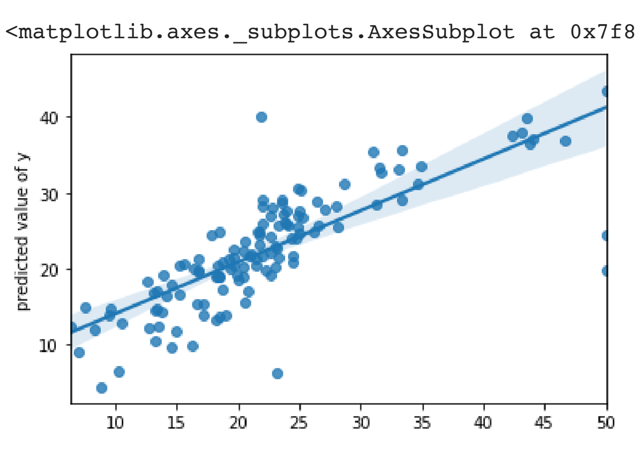
The basis of the assessment of the model is the comparison of the actual data values of y in the testing data, *y_test*, to the values of y calculated from the model, *y_pred*.

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. By default, the horizontal axis started numbering at 10, which was explicitly overridden to start at 0 with the `xlim()` function so that both axes begin at 0.

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 requires the `pandas` function `Series()` to name the associated series. In my opinion, it's a bit of contortion just to label the axes, but it works.

```
y_test = pd.Series(y_test, name="y from testing data")
y_pred = pd.Series(y_pred, name="predicted value of y")
sns.regplot(x=y_test, y=y_pred)
```

```
    <matplotlib.axes._subplots.AxesSubplot at 0x7f83762b7790>
```



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

Fit metrics

This first application is not always done. It evaluates the fit of the model to the training data, comparing the actual data values, *y_train*, to the corresponding values computed by the model, *y_fit*. This is *not* the official evaluation of model fit and performance. It is useful, however, to compare the fit indices for the training data to the testing data. A large drop indicates *overfitting* the model to the training data.

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.

```
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 *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.

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

```
    stdev of residuals: 4.502
```

Here we do 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.

```
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 an alternative to the one hold-out cross-validation in the previous section, here evaluate model fit with cross-validation on *multiple* samples. The `sklearn KFold` module performs the cross-validation in which the model is estimated using $k - 1$ folds and then tested on the remaining fold. The process automatically repeats for each fold.

Here pass specific parameter values to `Kfold`.

- *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 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. Larger data sets support a larger number of splits. Usually, shuffle the data first to keep the entire process entirely random.

Here instantiate the `KFold` module with *kf*, invoking the desired parameter values.

```
from sklearn.model_selection import KFold, cross_validate
kf = KFold(n_splits=5, shuffle=True, random_state=1)
```

The `cross_validate()` conveniently provides for multiple evaluation scores from the same cross-validation folds without manually repeating the computations for each score. Plus, computation times are also provided.

To estimate the model for each fold, here five different estimates from five different samples, specify the estimation algorithm. We have already instantiated the `LinearRegression()` estimator earlier as *reg_model*, but repeat here for clarity. 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.

Weirdly, MSE is reported in the negative. The reason is that the best score is always the largest across all scoring procedures and all estimation algorithms and is thus consistent with other model-tuning algorithms that expect this behavior and consistent with the internal code of the related `sklearn` functions. So here, the least negative is the largest value, the most desirable value. In reality, MSE must be a non-negative number, so the sign of the real MSE is just flipped to go negative.

The training data evaluations are not needed for the evaluation per se, which occurs on the testing data, but sometimes helpful to compare to the corresponding testing scores. Training scores much larger than the related testing scores indicates overfitting. Obtain the training information with the parameter *return_train_score*.

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

```
scores = cross_validate(reg_model, X, y, cv=kf,
                        scoring=('r2', 'neg_mean_squared_error'),
                        return_train_score=True)
```

Our *scores* array contains much information regarding the fit of each model over the five different analyses, but 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.

The parameter `inplace` set to `True` means making the change in the specified data frame and saving the data frame with those changes. This parameter removes the need to copy to a new data frame.

```
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	0.0048	0.0047	0.7634	0.7294	23.3808	21.8628
1	0.0020	0.0014	0.6468	0.7582	28.6143	20.5029
2	0.0019	0.0015	0.7921	0.7262	15.1606	23.7937
3	0.0019	0.0013	0.6508	0.7580	27.2082	20.8185
4	0.0022	0.0014	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.

```
print('Mean of test R-squared scores: %.3f' % ds['test_r2'].mean())
print('\n')
print('Mean of test MSE scores: %.3f' % ds['test_MSE'].mean())

se = np.sqrt(ds['test_MSE'].mean())
print('Standard deviation of mean test MSE scores: %.3f' % se)
```

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


    Mean of test MSE scores: 23.547
    Standard deviation of mean test MSE scores: 4.853
```

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.

> All that is needed for model validation if computation time permits is the k-fold cross-validation with multiple-scores.

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 for 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.

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.