Boston Housing Modeling

Release 2016.02.21

Russell Nakamura

March 04, 2016

Contents

The repository for the source of this document is here.

This is Project One from Udacity's Machine Learning Nanodegree program. It uses the UCI Boston Housing Dataset to build a model to predict prices for homes in the suburbs of Boston. The project begins with an exploration of the data to understand the feature and target variables, this is followed by the selection of a performance metric for the model, an analysis of how the model performed, and finally the model is applied to a hypothetical client's house to predict a value for the house.

1 Statistical Analysis and Data Exploration

This section is an exploratory analysis of the Boston Housing data which will introduce the data and some changes that I made, summarize the median-value data, then look at the features to make an initial hypothesis about the value of the client's home.

1.1 The Data

The data was taken from the sklearn.load_boston function (*sklearn* cites the UCI Machine Learning Repository as their source for the data). The data gives values for various features of different suburbs of Boston as well as the

median-value for homes in each suburb. The features were chosen to reflect various aspects believed to influence the price of houses including the structure of the house (age and spaciousness), the quality of the neighborhood, transportation access to employment centers and highways, and pollution.

There are 14 variables in the data set (13 features and the median-value target). Here is the description of the data variables provided by sklearn.

Table 1: Attribute Information (in order)

Variable Name	Description
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 centers
RAD	index of accessibility to radial highways
TAX	full-value property-tax rate per $10,000
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

Note: The data comes from the 1970 U.S. Census and the *median-values* have not been inflation-adjusted.

Cleaning the Data

There are no missing data points but the odd variable names are sometimes confusing so I'm going to expand them to full variable names.

Table 2: Variable Aliases

Original Variable	New Variable
CRIM	crime_rate
ZN	large_lots
INDUS	industrial
CHAS	charles_river
NOX	nitric_oxide
RM	rooms
AGE	old_houses
DIS	distances
RAD	highway_access
TAX	property_taxes
PTRATIO	pupil_teacher_ratio
B	proportion_blacks
LSTAT	lower_status

1.2 Median Value

The target variable for this data-set is the *median-value* of houses within a given suburb. After presenting some summary statistics for the *median-value* I'll make some plots to get a sense of the shape of the data.

Table 3: Boston Housing median-value statistics (in $1000's)

Item	Value
count	506
mean	22.53
std	9.20
min	5.00
25%	17.02
50%	21.20
75%	25.00
max	50.00
IQR	7.975

Outlier Check

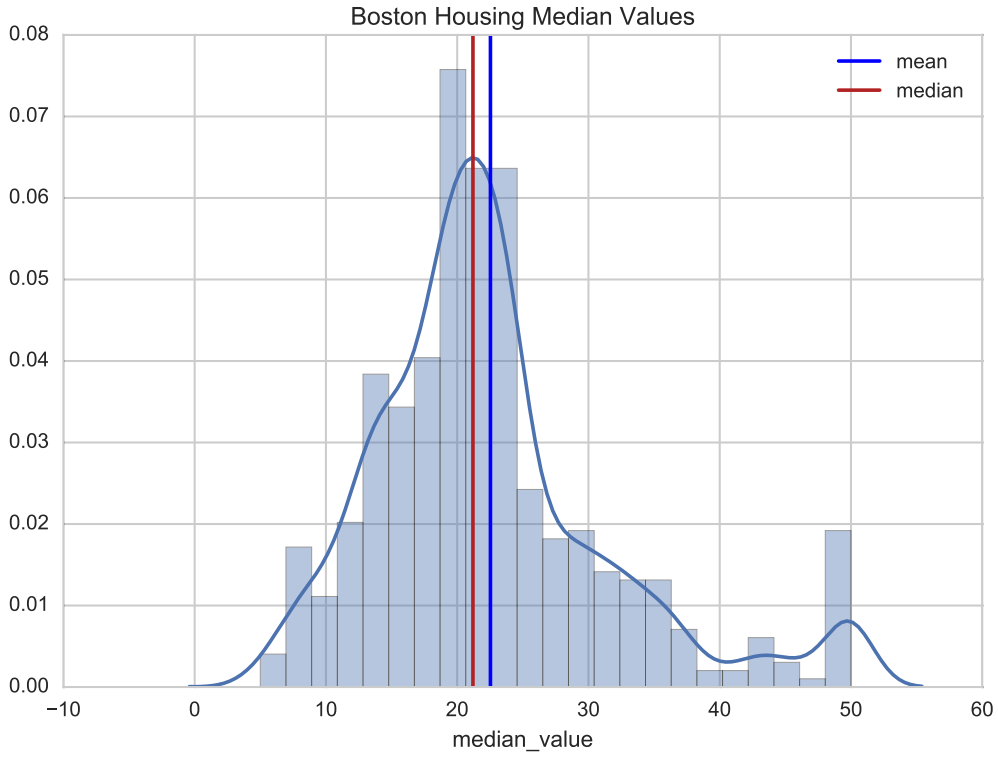
Comparing the mean (22.53) and the median (21.2) it looks like the distribution might be right-skewed. This is more obvious looking at distribution plots below, but I'll also do an outlier check here using the traditional $Q1—1.5\times IQR$ for low outliers and $Q3 + 1.5 \times IQR$ for the higher outliers to see how many there might be.

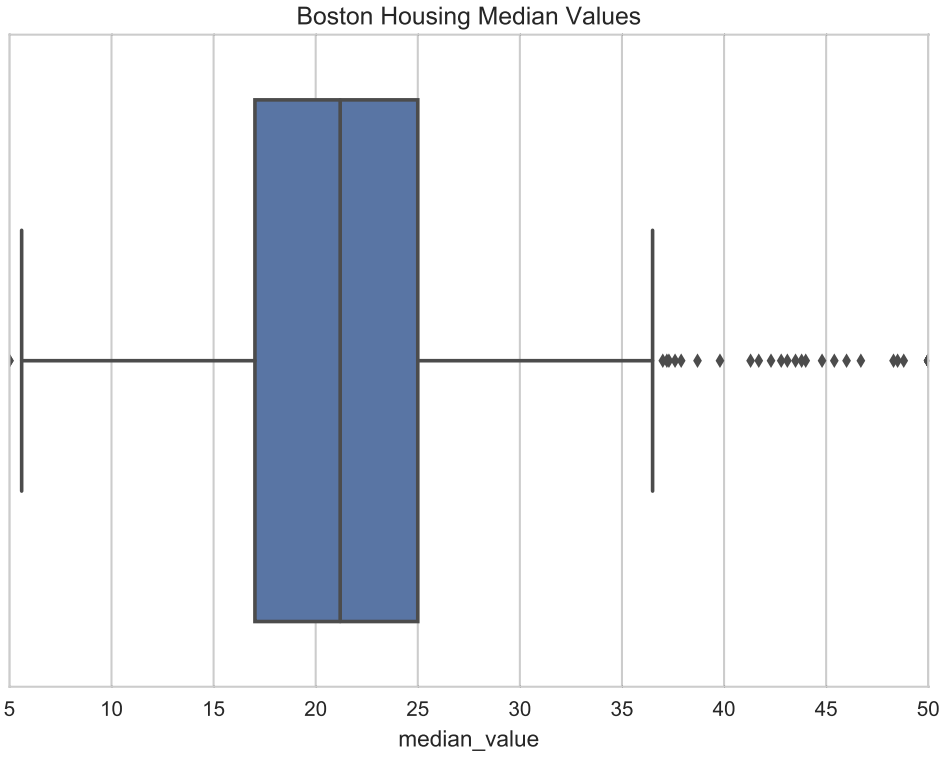
Table 4: Outlier Count

Description	Value
Low Outlier Limit (LOL)	5.06
LOL - min	0.06
Upper Outlier Limit (UOL)	36.96
max - UOL	13.04
Low Outlier Count	2
High Outlier Count	38

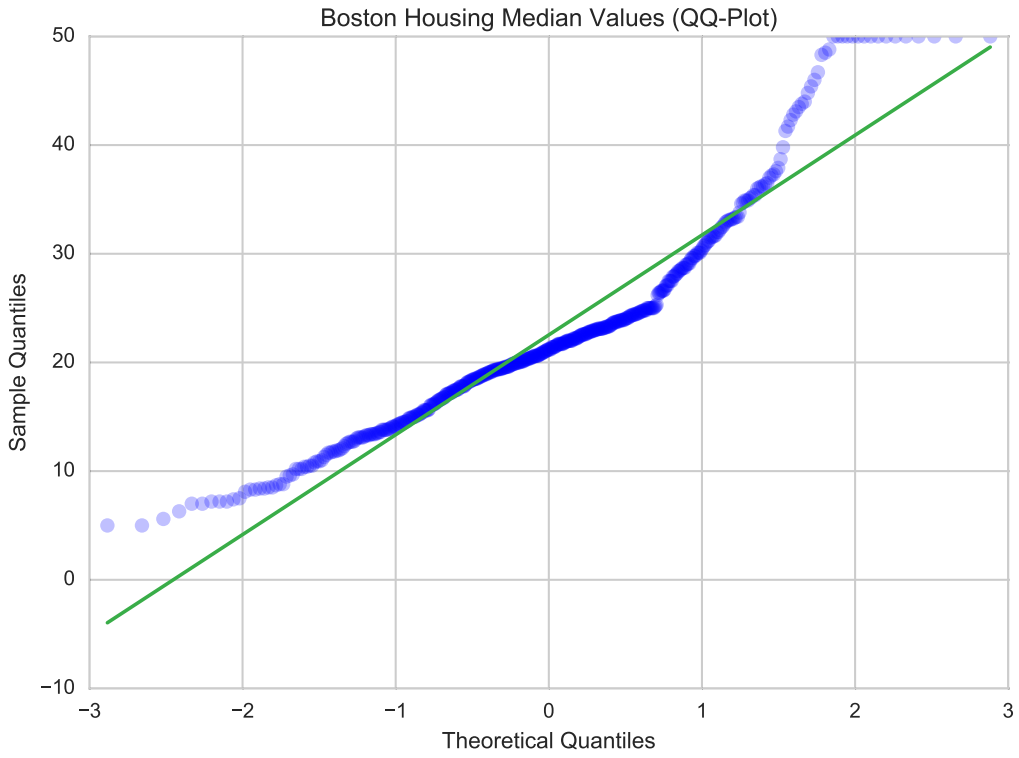
There aren't an excessive number of outliers - about 8% of the median-values are above the upper outlier limit (UOL) and less than 1% below the lower-outlier limit. The difference between the maximum value of 50 and the UOL is 13.04, however, which is almost as large as the difference between the UOL and the median (15.76) so there might be an undue influence from the upper values if parametric statistics are used.

Plots

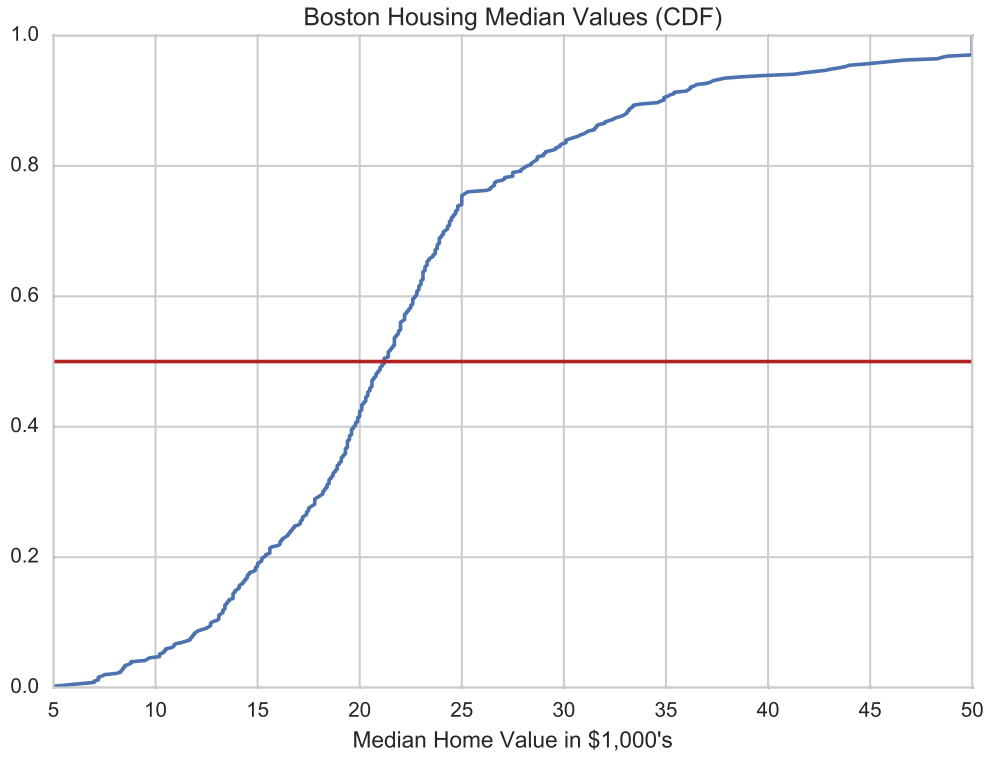

Boston Housing Median Values
mean
median
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
−10
0
10
20
30
40
50
60
median_value


Boston Housing Median Values
5
10
15
20
25
30
35
40
45
50
median_value

The KDE/histogram and box-plot seem to confirm what was shown in the section on outliers, which is that there are some unusually high median-values in the data.



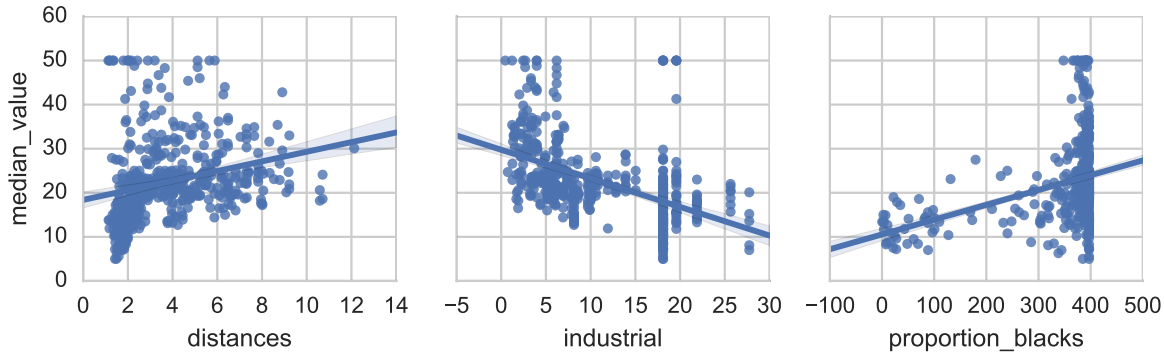
The QQ-Plot shows that the distribution is initially fairly normal but the upper-third seems to come from a different distribution than the lower two-thirds.

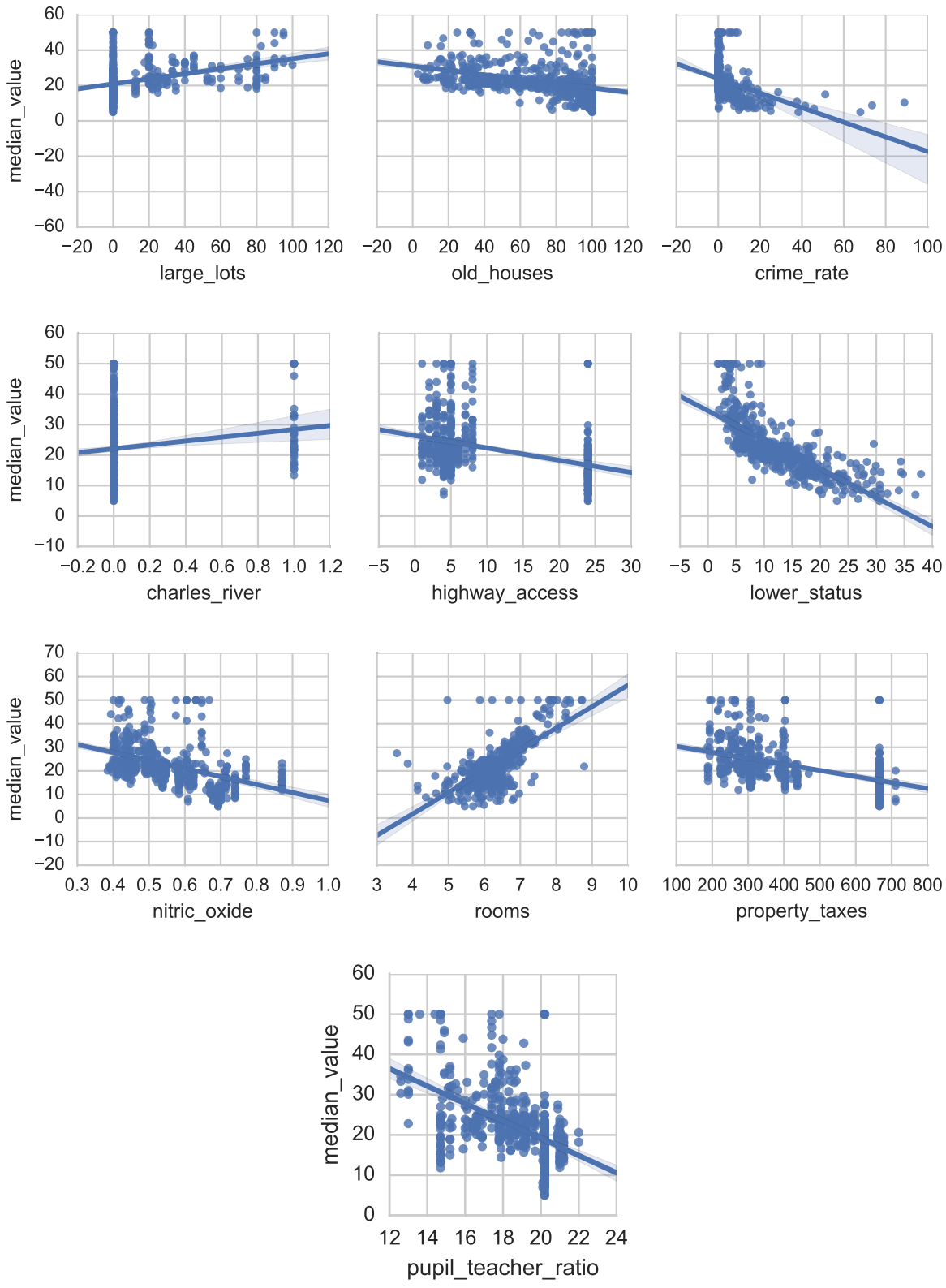


Looking at the distribution (histogram and KDE plot) and box-plot the median-values for the homes appear to be right-skewed. The CDF shows that about 90% of the homes are $35,000 or less (the 90th percentile for median-value is 34.8) and that there's a change in the spread of the data around $25,000. The qq-plot and the other plots show that the median-values aren't normally distributed.

1.3 Possibly Significant Features

To get an idea of how the features are related to the median-value, I'll plot some linear-regressions.





Looking at the plots, the three features that I think are the most significant are *lower_status (LSTAT)*, *nitric_oxide (NOX)*, and *rooms (RM)*. The *lower_status* variable is the percent of the population of the town that is of 'lower status' which is defined in this case as being an adult with less than a ninth-grade education or a male worker that is classified

as a laborer. The *nitric_oxide* variable represents the annual average parts per million of nitric-oxide measured in the air and is thus a stand-in for pollution. *rooms* is the average number of rooms per dwelling, representing the spaciousness of houses in the suburb (Harrison and Rubinfeld, 1978).

1.4 The Client

As I mentioned previously, the main goal of this project is to create a model to predict the house price for a client. Here are the client's values.

Table 5: Client Values

Feature	Value
crime_rate	11.95
large_lots	0.0
industrial	18.1
charles_river	0
nitric_oxide	0.659
rooms	5.609
old_houses	90.0
distances	1.385
highway_access	24
property_taxes	680.0
pupil_teacher_ratio	20.2
proportion_blacks	332.09
lower_status	12.13

The Client's Significant Features

Now a comparison of the client's values for the three features that I hypothesized might be the most significant along with the values from the data-set.

Table 6: Client Significant Features

Variable	Client Value	Boston Q1	Boston Median	Boston Q3
lower_status	12.13	6.95	11.36	16.96
nitric_oxide	0.66	0.45	0.54	0.62
rooms	5.61	5.89	6.21	6.62

Comparing the values I guessed would be significant for the client to the median-values for the data set as a whole shows that the client has a higher ratio of lower-status adults, more pollution and fewer rooms than the median suburbs so I would expect that the predicted value will be lower than the median.

2 Evaluating Model Performance

Here I'll discuss splitting the data for training and testing, the performance metric I chose, the algorithm used for the modeling and how the hyper-parameters for the model were chosen.

2.1 Splitting the Data

First a function named *shuffle_split_data* was created that acts as an alias for the *train_test_split* function from *sklearn*. The main difference is that the ordering of the data-sets is changed from both x's followed by both y's to both training sets followed by both testing sets. In this case a 70% training data, 30% test data split was used.

We split the data into training and testing subsets so that we can assess the model using a different data-set than what it was trained on, thus reducing the likelihood of overfitting the model to the training data and increasing the likelihood that it will generalize to other data.

2.2 Choosing a Performance Metric

There are several possible regression metrics to use, but I chose *Mean Squared Error* as the most appropriate performance metric for predicting housing prices because we are predicting a numeric value (a regression problem) and while *Mean Absolute Error*, *Median Absolute Error*, *Explained Variance Score*, or *r2_score* could also be used, I wanted a metric that would be based on the errors in the model and the MSE emphasizes larger errors more and so I felt it would be preferable.

The *Mean Squared Error* is an average of the squared differences between predicted values and the actual values.

$$MSE(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2$$

2.3 DecisionTreeRegressor

The model was built using sklearn's DecisionTreeRegressor, a non-parametric, tree-based algorithm (using the Classification and Regression Trees (CART) tree algorithm).

2.4 Grid Search

A grid search was used to find the optimal parameters (tree depth) for the DecisionTreeRegressor. The GridSearchCV algorithm exhaustively works through the parameters it is given to find the parameters that create the best model using cross-validation. Because it is exhaustive it is appropriate when the model-creation is not excessively computationally intensive, otherwise its run-time might be infeasible.

Cross-Validation

As mentioned, *GridSearchCV* uses *cross-validation* to find the optimal parameters for a model. Cross-validation is a method of testing a model by partitioning the data into subsets, with each subset taking a turn as the test set while the data not being used as a test-set is used as the training set. This allows the model to be tested against all the data-points, rather than having some data reserved exclusively as training data and the remainder exclusively as testing data.

Because grid-search attempts to find the optimal parameters for a model, it's advantageous to use the same training and testing data in each case (case meaning a particular permutation of the parameters) so that the comparisons are equitable. One could simply perform an initial train-validation-test split and use this throughout the grid search, but this then risks the possibility that there was something in the initial split that will bias the outcome. By using all the partitions of the data as both test and training data, as cross-validation does, the chance of a bias in the splitting is reduced and at the same time all the parameter permutations are given the same data to be tested against.

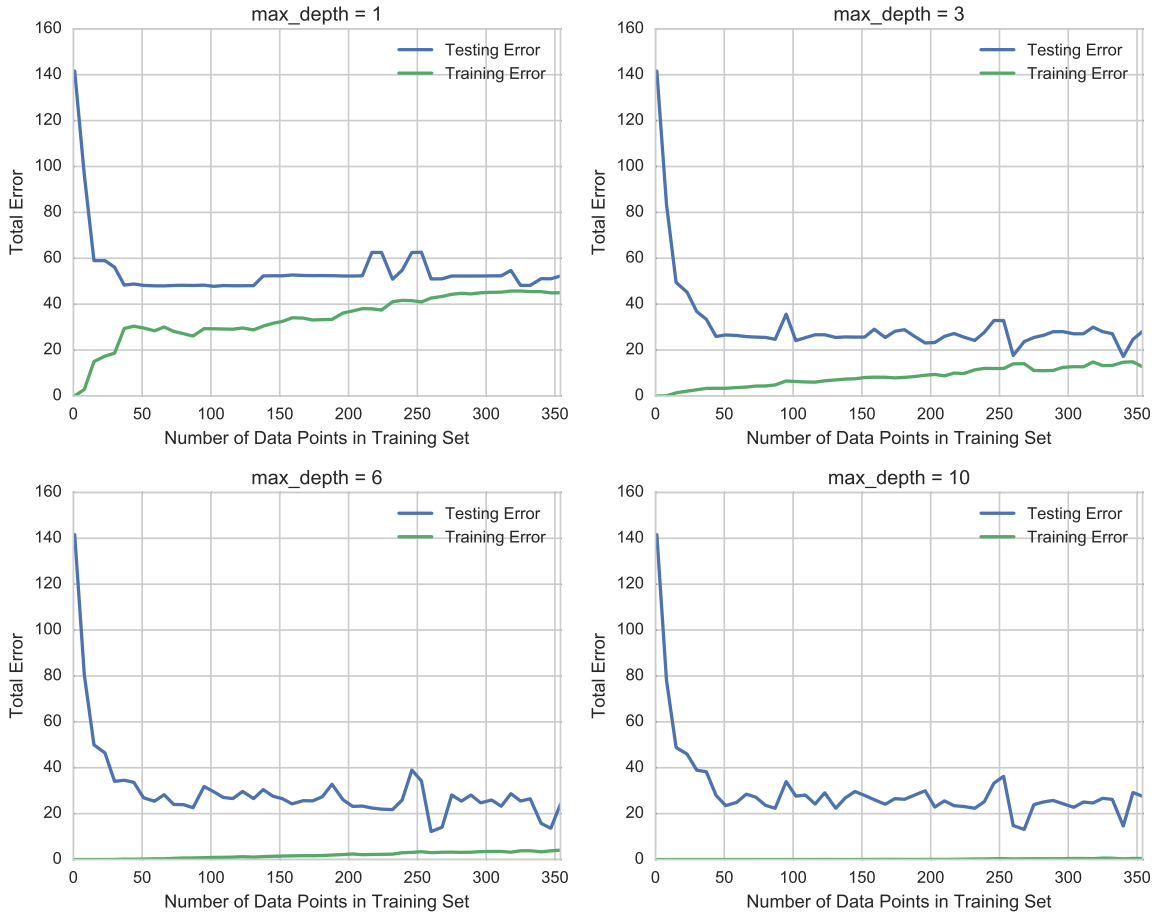
In this case I used *k=10* for the k-fold cross validation that the *GridSearchCV* uses.

3 Analyzing Model Performance

The two methods used here for analyzing how the model is performing with the data are *Learning Curves* and a *Model Complexity* plot.

3.1 Learning Curves

The *Learning Curves* show how a model's performance changes as it is given more data. In this case four *max_depth* sizes were chosen for comparison.



Looking at the model with max-depth of 3, as the size of the training set increases, the training error gradually increases. The testing error initially decreases, then seems to more or less stabilize.

The training and testing plots for the model with max-depth 1 move toward convergence with an error near 50, indicating a high bias (the model is too simple, and the additional data isn't improving the generalization of the model).

For the model with max-depth 10, the curves haven't converged, and the training error remains near 0, indicating that it suffers from high variance, and should be improved with more data.

3.2 Model Complexity

The *Model Complexity* plot allows us to see how the model's performance changes as the max-depth is increased.

As max-depth increases the training error improves, while the testing error decreases up until a depth of 5 and then begins a slight increase as the depth is increased. Based on this I would say that the max-depth of 5 created the model that best generalized the data set, as it minimized the testing error, while the models with greater max-depth parameters likely overfitted the training data.

4 Model Prediction

To find the 'best' model I ran the *fit_model* function 1,000 times and took the *best_params_* (max-depth) and *best_score_* (negative MSE) for each trial.

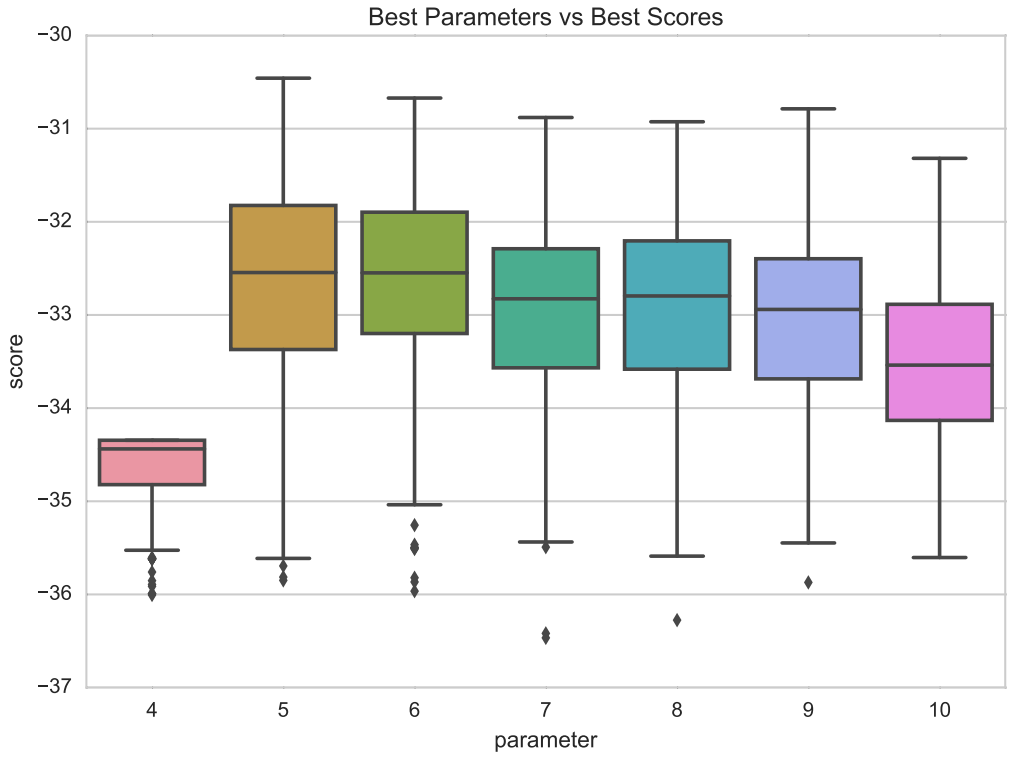


Table 7: Parameter Counts

Max-Depth	Count
4	315
5	190
7	166
6	136
8	111
9	82

Table 8: Median Scores

Max-Depth	Median Score
4	-34.44
5	-32.54
6	-32.55
7	-32.83
8	-32.80
9	-32.94
10	-33.54

Table 9: Max Scores

Max-Depth	Max Score
4	-34.35
5	-30.46
6	-30.67
7	-30.88
8	-30.93
9	-30.79
10	-31.32

Note: Since the *GridSearchCV* normally tries to maximize the output of the scoring-function, but the goal in this case was to minimize it, the scores are negations of the MSE, thus the higher the score, the lower the MSE.

While a max-depth of 4 was the most common best-parameter, the max-depth of 5 was the median max-depth, had the highest median score, and had the highest overall score, so I will say that the optimal *max_depth* parameter is 5. This is in line with what I had guessed, based on the Complexity Performance plot.

4.1 Predicting the Client's Price

Using the model that had the lowest MSE (30.46) out of the 1,000 generated, I then made a prediction for the price of the client's house.

Table 10: Predicted Price

Predicted value of client's home	$20,967.76
Difference between median and predicted	$232.24

My three chosen features (*lower_status*, *nitric_oxide*, and *rooms*) seemed to indicate that the client's house might be a lower-valued house, and the predicted value was about $232 less than the median median-value, so it appears that our model predicts that the client has a below-median-value house.

Confidence Interval

Although this isn't an inferential analysis, I calculated the 95% Confidence Interval for the median-value so that I would have a range to compare the prediction to. Since the data isn't symmetric I used a bootstrapped confidence interval (bias-corrected and accelerated (BCA)) of the median instead of one based on the standard error and found 95% CI [20.40, 21.75].

Our prediction for the client's house falls within a 95% confidence interval for the median, so although I predicted that it would be below the median, there's insufficient evidence to conclude that it differs from the median house price.

4.2 Assessing the Model

I think that this model seems reasonable for the given data (Boston Suburbs in 1970), but I think that I might be hesitant to predict the value for a specific house using it, given that we are using aggregate-values for entire suburbs, not values for individual houses. I would also think that separating out the upper-class houses would give a better model for certain clients, given the right-skew of the data. Also, the median MSE for the best model was ~ 32 so

taking the square root of this gives an 'average' error of about $5,700, which seems fairly high, given the low median-values for the houses. I think that the model gives a useful ball-park-figure estimate, but I think I'd have to qualify the certainty of prediction for future clients, noting also the age of the data and not extrapolating much beyond 1970.

5 References

Harrison, D. and Rubinfeld, D.L. 'Hedonic prices and the demand for clean air', J. Environ. Economics & Management, vol.5, 81-102, 1978.

Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

6 Software

Here are some auto-generated descriptions of some of the software used. *train_test_split, mean_squared_error, DecisionTreeRegressor,* and *GridSearchCV* are from *sklearn*, the rest was part of this project.

6.1 Evaluating Model Performance

Splitting the Data

`shuffle_split_data`(X, y[, test_size, ...])	Shuffles and splits data into training and testing subsets

boston_housing.evaluating_model_performance.shuffle_split_data

`boston_housing.evaluating_model_performance.shuffle_split_data`(*X*, *y*, *test_size=0.3*, *random_state=0*)

Shuffles and splits data into training and testing subsets

Param

- *X*: feature array
- *y*: target array
- *test_size*: fraction of data to use for testing
- *random_state*: seed for the random number generator

Returns x-train, y-train, x-test, y-test

`train_test_split`(*arrays, **options)	Split arrays or matrices into random train and test subsets

sklearn.cross_validation.train_test_split

`sklearn.cross_validation.train_test_split`(**arrays*, ***options*)

Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and `next(iter(ShuffleSplit(n_samples)))` and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.

Read more in the User Guide.

*arrays : sequence of indexables with same length / shape[0]

allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.

New in version 0.16: preserves input type instead of always casting to numpy array.

test_size [float, int, or None (default is None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is automatically set to the complement of the train size. If train size is also None, test size is set to 0.25.

train_size [float, int, or None (default is None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state [int or RandomState] Pseudo-random number generator state used for random sampling.

stratify [array-like or None (default is None)] If not None, data is split in a stratified fashion, using this as the labels array.

New in version 0.17: *stratify* splitting

splitting [list, length = 2 * len(arrays),] List containing train-test split of inputs.

New in version 0.16: Output type is the same as the input type.

```
>>> import numpy as np
>>> from sklearn.cross_validation import train_test_split
>>> X, y = np.arange(10).reshape((5, 2)), range(5)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> list(y)
[0, 1, 2, 3, 4]
```

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.33, random_state=42)
...
>>> X_train
array([[4, 5],
       [0, 1],
       [6, 7]])
>>> y_train
[2, 0, 3]
>>> X_test
array([[2, 3],
       [8, 9]])
>>> y_test
[1, 4]
```

performance_metric(y_true, y_predict)	Calculates total error between true and predicted values

boston_housing.evaluating_model_performance.performance_metric

boston_housing.evaluating_model_performance.performance_metric(*y_true*, *y_predict*)

Calculates total error between true and predicted values

Param

- *y_true*: array of target values
- *y_predict*: array of values the model predicted

Returns mean_squared_error for the prediction

mean_squared_error(y_true, y_pred[, ...])	Mean squared error regression loss

sklearn.metrics.mean_squared_error

sklearn.metrics.mean_squared_error(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

Mean squared error regression loss

Read more in the User Guide.

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average']] or array-like of shape (n_outputs) Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors in case of multioutput input.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

loss [float or ndarray of floats] A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.

```
>>> from sklearn.metrics import mean_squared_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred)
0.375
>>> y_true = [[0.5, 1],[-1, 1],[7, -6]]
>>> y_pred = [[0, 2],[-1, 2],[8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.708...
>>> mean_squared_error(y_true, y_pred, multioutput='raw_values')
...
array([ 0.416...,  1.        ])
>>> mean_squared_error(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.824...
```

Decision Tree Regressor

fit_model(X, y[, k, n_jobs])	Tunes a decision tree regressor model using GridSearchCV

boston_housing.evaluating_model_performance.fit_model

boston_housing.evaluating_model_performance.fit_model(*X*, *y*, *k=10*, *n_jobs=1*)

Tunes a decision tree regressor model using GridSearchCV

Param

- *X*: the input data
- *y*: target labels y
- *k*: number of cross-validation folds
- *n_jobs*: number of parallel jobs to run

Returns the optimal model

DecisionTreeRegressor([criterion, splitter, ...])	A decision tree regressor.

sklearn.tree.DecisionTreeRegressor

class sklearn.tree.DecisionTreeRegressor(*criterion='mse'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *presort=False*)

A decision tree regressor.

Read more in the User Guide.

criterion [string, optional (default="mse")] The function to measure the quality of a split. The only supported criterion is "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion.

splitter [string, optional (default="best")] The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_features [int, float, string or None, optional (default=None)]

The number of features to consider when looking for the best split:

- If int, then consider *max_features* features at each split.
- If float, then *max_features* is a percentage and *int(max_features * n_features)* features are considered at each split.
- If "auto", then *max_features=n_features*.
- If "sqrt", then *max_features=sqrt(n_features)*.
- If "log2", then *max_features=log2(n_features)*.
- If None, then *max_features=n_features*.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

max_depth [int or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if `max_leaf_nodes` is not None.

min_samples_split [int, optional (default=2)] The minimum number of samples required to split an internal node.

min_samples_leaf [int, optional (default=1)] The minimum number of samples required to be at a leaf node.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the input samples required to be at a leaf node.

max_leaf_nodes [int or None, optional (default=None)] Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then `max_depth` will be ignored.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

presort [bool, optional (default=False)] Whether to presort the data to speed up the finding of best splits in fitting. For the default settings of a decision tree on large datasets, setting this to true may slow down the training process. When using either a smaller dataset or a restricted depth, this may speed up the training.

feature_importances_ [array of shape = [n_features]] The feature importances. The higher, the more important the feature. The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance [4].

max_features_ [int,] The inferred value of max_features.

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

tree_ [Tree object] The underlying Tree object.

DecisionTreeClassifier

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.tree import DecisionTreeRegressor
>>> boston = load_boston()
>>> regressor = DecisionTreeRegressor(random_state=0)
>>> cross_val_score(regressor, boston.data, boston.target, cv=10)
...
...
array([ 0.61..., 0.57..., -0.34..., 0.41..., 0.75...,
        0.07..., 0.29..., 0.33..., -1.42..., -1.77...])
```

`__init__`(*criterion='mse', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, presort=False*)

Methods

[4] L. Breiman, and A. Cutler, "Random Forests", http://www.stat.berkeley.edu/ ~ breiman/RandomForests/cc_home.htm

`__init__`([criterion, splitter, max_depth, ...])	
`apply`(X[, check_input])	Returns the index of the leaf that each sample is predicted as.
`fit`(X, y[, sample_weight, check_input, ...])	Build a decision tree from the training set (X, y).
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X[, check_input])	Predict class or regression value for X.
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(*args, **kwargs)	DEPRECATED: Support to use estimators as feature selectors will be removed in

Attributes

`feature_importances_`	Return the feature importances.

Grid Search

`GridSearchCV`(estimator, param_grid[, ...])	Exhaustive search over specified parameter values for an estimator.

sklearn.grid_search.GridSearchCV

class `sklearn.grid_search.GridSearchCV`(*estimator*, *param_grid*, *scoring=None*, *fit_params=None*, *n_jobs=1*, *iid=True*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score='raise'*)

Exhaustive search over specified parameter values for an estimator.

Important members are fit, predict.

GridSearchCV implements a "fit" and a "score" method. It also implements "predict", "predict_proba", "decision_function", "transform" and "inverse_transform" if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

Read more in the User Guide.

estimator [estimator object.] A object of that type is instantiated for each grid point. This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

param_grid [dict or list of dictionaries] Dictionary with parameters names (string) as keys and lists of parameter settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored. This enables searching over any sequence of parameter settings.

scoring [string, callable or None, default=None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`. If `None`, the `score` method of the estimator is used.

fit_params [dict, optional] Parameters to pass to the fit method.

n_jobs [int, default=1] Number of jobs to run in parallel.

Changed in version 0.17: Upgraded to joblib 0.9.3.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

iid [boolean, default=True] If True, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample, and not the mean loss across the folds.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

For integer/None inputs, if y is binary or multiclass, `StratifiedKFold` used. If the estimator is a classifier or if y is neither binary nor multiclass, `KFold` is used.

Refer User Guide for the various cross-validation strategies that can be used here.

refit [boolean, default=True] Refit the best estimator with the entire dataset. If "False", it is impossible to make predictions using this GridSearchCV instance after fitting.

verbose [integer] Controls the verbosity: the higher, the more messages.

error_score ['raise' (default) or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error.

```
>>> from sklearn import svm, grid_search, datasets
>>> iris = datasets.load_iris()
>>> parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
>>> svr = svm.SVC()
>>> clf = grid_search.GridSearchCV(svr, parameters)
>>> clf.fit(iris.data, iris.target)
...
GridSearchCV(cv=None, error_score=...,
       estimator=SVC(C=1.0, cache_size=..., class_weight=..., coef0=...,
                     decision_function_shape=None, degree=..., gamma=...,
                     kernel='rbf', max_iter=-1, probability=False,
                     random_state=None, shrinking=True, tol=...,
                     verbose=False),
       fit_params={}, iid=..., n_jobs=1,
       param_grid=..., pre_dispatch=..., refit=...,
       scoring=..., verbose=...)
```

grid_scores_ [list of named tuples] Contains scores for all parameter combinations in param_grid. Each entry corresponds to one parameter setting. Each named tuple has the attributes:

- `parameters`, a dict of parameter settings
- `mean_validation_score`, the mean score over the cross-validation folds
- `cv_validation_scores`, the list of scores for each fold

best_estimator_ [estimator] Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if refit=False.

best_score_ [float] Score of best_estimator on the left out data.

best_params_ [dict] Parameter setting that gave the best results on the hold out data.

scorer_ [function] Scorer function used on the held out data to choose the best parameters for the model.

The parameters selected are those that maximize the score of the left out data, unless an explicit score is passed in which case it is used instead.

If *n_jobs* was set to a value higher than one, the data is copied for each point in the grid (and not *n_jobs* times). This is done for efficiency reasons if individual jobs take very little time, but may raise errors if the dataset is large and not enough memory is available. A workaround in this case is to set *pre_dispatch*. Then, the memory is copied only *pre_dispatch* many times. A reasonable value for *pre_dispatch* is *2 * n_jobs*.

`ParameterGrid`: generates all the combinations of a an hyperparameter grid.

`sklearn.cross_validation.train_test_split()`: utility function to split the data into a development set usable for fitting a GridSearchCV instance and an evaluation set for its final evaluation.

`sklearn.metrics.make_scorer()`: Make a scorer from a performance metric or loss function.

`__init__`(*estimator*, *param_grid*, *scoring=None*, *fit_params=None*, *n_jobs=1*, *iid=True*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score='raise'*)

Methods

`__init__`(estimator, param_grid[, scoring, ...])	
`decision_function`(*args, **kwargs)	Call decision_function on the estimator with the best found parameters.
`fit`(X[, y])	Run fit with all sets of parameters.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(*args, **kwargs)	Call inverse_transform on the estimator with the best found parameters.
`predict`(*args, **kwargs)	Call predict on the estimator with the best found parameters.
`predict_log_proba`(*args, **kwargs)	Call predict_log_proba on the estimator with the best found parameters.
`predict_proba`(*args, **kwargs)	Call predict_proba on the estimator with the best found parameters.
`score`(X[, y])	Returns the score on the given data, if the estimator has been refit.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(*args, **kwargs)	Call transform on the estimator with the best found parameters.

6.2 Analyzing Model Performance

`learning_curves`(X_train, y_train, X_test, y_test)	Calculates performance of several models with varying training data sizes
`model_complexity`(X_train, y_train, X_test, ...)	Calculates the performance of the model as model complexity increases.

boston_housing.analyzing_model_performance.learning_curves

`boston_housing.analyzing_model_performance.learning_curves`(*X_train*, *y_train*, *X_test*, *y_test*)

Calculates performance of several models with varying training data sizes Then plots learning and testing error rates for each model

boston_housing.analyzing_model_performance.model_complexity

boston_housing.analyzing_model_performance.model_complexity(*X_train*, *y_train*, *X_test*, *y_test*)

Calculates the performance of the model as model complexity increases. Then plots the learning and testing errors rates

Index

Symbols

D

F

G

L

M

P

S

T