Interpreting decision-tree models

Ralph Haygood

January 17, 2017

Outline

- Decision-tree models
  - Individual decision trees
  - Random forests
  - Gradient-boosted tree models
- Feature importances
- Partial dependence functions and plots
  - 1-dimensional
  - 2-dimensional
- Feature interactions
  - Decision-tree depth
  - *H* statistics

Data set for examples

```
>>> from sklearn.datasets.california_housing import fetch_california_housing

>>> housing = fetch_california_housing()

>>> print(housing.DESCR)
California housing dataset.

The original database is available from StatLib

    http://lib.stat.cmu.edu/

The data contains 20,640 observations on 9 variables.

This dataset contains the average house value as target variable
and the following input variables (features): average income,
housing average age, average rooms, average bedrooms, population,
average occupation, latitude, and longitude in that order.
```

(The target is continuous, so this is a regression situation. We'll use language appropriate to regression. However, analogous considerations apply to classification and ranking.)

Decision-tree models — Individual decision trees

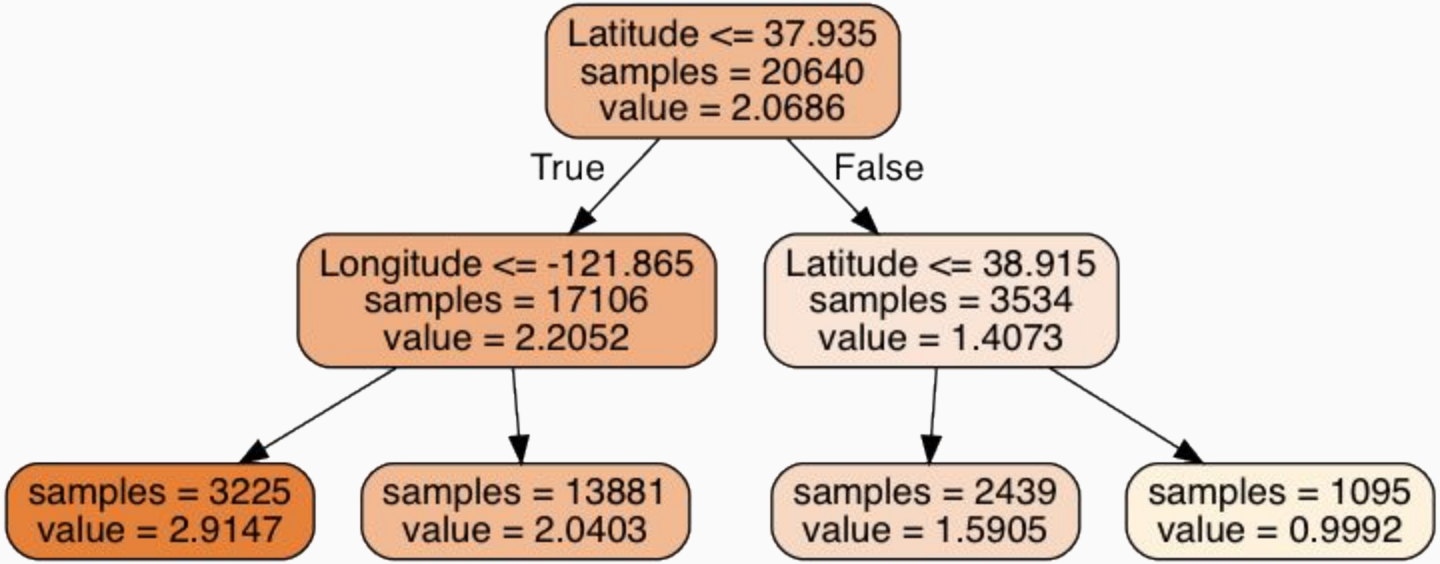
A *decision tree* is a partition of the feature space into a set of boxes.

Given any observation, its feature values lie in some box.

Given a new observation, its predicted target value is the mean of the target values of the training observations whose feature values lie in the same box.

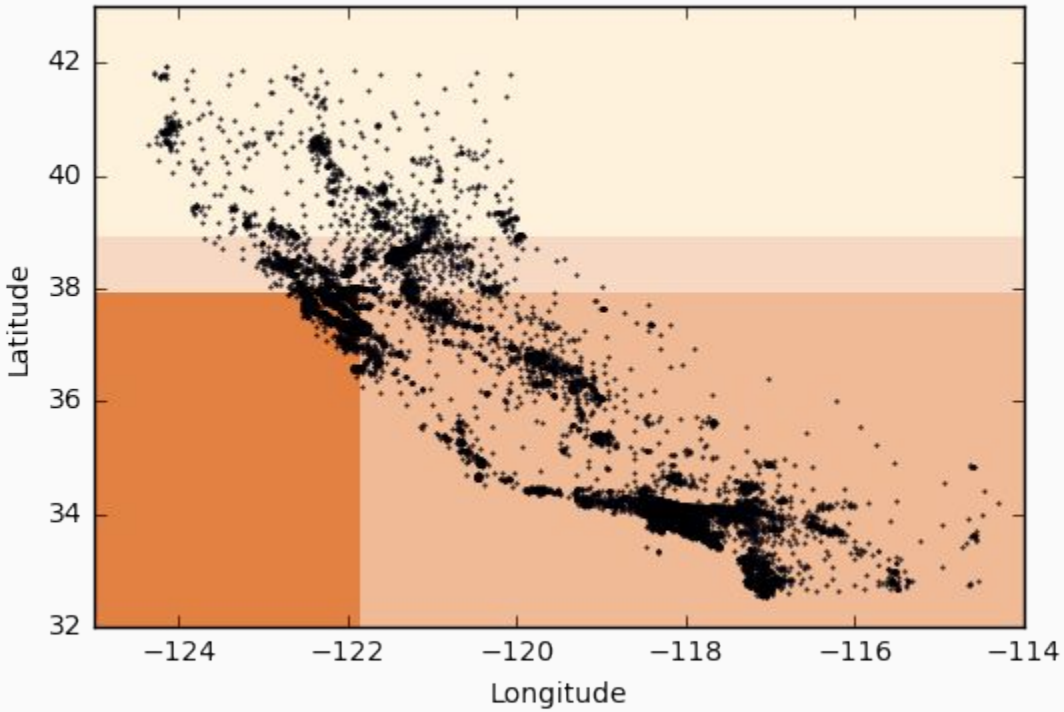
Decision-tree models — Individual decision trees

Example: California housing data with features restricted to latitude and longitude.



Decision-tree models — Individual decision trees

Example: California housing data with features restricted to latitude and longitude.



Decision-tree models — Individual decision trees

Advantages:

- Quick to build.
- **Easy to interpret.**
- Automatically capture interactions (if deep enough).

Disadvantages:

- Prone to overfitting — pruning is essential.
- **Not very accurate — other methods are often more accurate.**
- Not very robust — small changes in data yield large changes in tree.

Decision-tree models — Individual decision trees

```
>>> from sklearn.model_selection import train_test_split as tts

>>> data_train, data_test, target_train, target_test = \
...     tts(housing.data, housing.target, test_size = 0.1, random_state = 42)

>>> from sklearn import tree

>>> dtr = tree.DecisionTreeRegressor(random_state = 42)

>>> dtr.fit(data_train, target_train)
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
        max_leaf_nodes=None, min_impurity_split=1e-07,
        min_samples_leaf=1, min_samples_split=2,
        min_weight_fraction_leaf=0.0, presort=False, random_state=42,
        splitter='best')

>>> dtr.score(data_test, target_test)
0.637318351331017
```

Decision-tree models — Random forests

A *random forest* is an ensemble of decision trees.

> Each tree is built using a *bootstrap replicate* — a random sample with replacement — of the training observations. This is called *bagging*.
>
> Each tree is built using a random sample (without replacement) of the features. If there are d features, then $\log_2(d)$ features might be used.
>
> Given a new observation, its predicted target value is its mean predicted target value over the ensemble.

The trees aren't pruned (although a maximum depth may be specified), so typically each tree is overfitted. However, due to bagging and feature sampling, the ensemble is less so, typically much less so.

Decision-tree models — Random forests

```
>>> from sklearn.ensemble import RandomForestRegressor

>>> rfr = RandomForestRegressor(max_features = 'log2', random_state = 42)

>>> rfr.fit(data_train, target_train)
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
          max_features='log2', max_leaf_nodes=None,
          min_impurity_split=1e-07, min_samples_leaf=1,
          min_samples_split=2, min_weight_fraction_leaf=0.0,
          n_estimators=10, n_jobs=1, oob_score=False, random_state=42,
          verbose=0, warm_start=False)

>>> rfr.score(data_test, target_test)
0.79622623663286562
```

Decision-tree models — Gradient-boosted tree models

A *gradient-boosted tree model* is also an ensemble of decision trees.

The ensemble is built sequentially. After m steps, the predicted target value associated with any given feature values $\boldsymbol{x}$ is

$$T_m(\boldsymbol{x}) = \lambda \left[t_1(\boldsymbol{x}) + t_2(\boldsymbol{x}) + \ldots + t_m(\boldsymbol{x}) \right]$$

where $t_k(\boldsymbol{x})$ is the output of tree k and λ is a *shrinkage parameter* between 0 and 1.

Tree m+1 is fitted to the residuals $y - T_m(\boldsymbol{x})$ for training observations $(\boldsymbol{x}, y)$. This is called *gradient boosting*.

As with random forests, bagging and feature sampling may be used.

A maximum depth is specified, typically a small one (say, 3), so each tree involves just a few features. This limit, shrinkage, and bagging and feature sampling (if used) inhibit overfitting.

Decision-tree models — Gradient-boosted tree models

```
>>> from sklearn.ensemble import GradientBoostingRegressor

>>> gbr = GradientBoostingRegressor(random_state = 42)

>>> gbr.fit(data_train, target_train)
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
             learning_rate=0.1, loss='ls', max_depth=3, max_features=None,
             max_leaf_nodes=None, min_impurity_split=1e-07,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=100,
             presort='auto', random_state=42, subsample=1.0, verbose=0,
             warm_start=False)

>>> gbr.score(data_test, target_test)
0.84732901469177024
```

So random forests and gradient-boosted tree models are more accurate than individual decision trees.

But how do we interpret an ensemble of decision trees?

Feature importances

For any given feature:

For any given tree in the ensemble:

For each node of the tree involving a split on the feature:

Take the decrease in the residual sum of squares (RSS) of the training observations due to the node.

Sum over nodes of the tree (if more than one is relevant).

Average over trees in the ensemble.

Normalize so that the sum over the features is 1.

These are called *feature importances*.

Feature importances

```
>>> pd.Series(gbr.feature_importances_, index = housing.feature_names) \
...       .sort_values(ascending = False)
Longitude     0.214620
Latitude      0.197368
MedInc        0.172621
AveOccup      0.149446
AveBedrms     0.081797
HouseAge      0.072276
AveRooms      0.061962
Population    0.049909
dtype: float64
```

Partial dependence functions and plots — 1-dimensional

Let $T(\boldsymbol{x}) = T(x_1, x_2, \ldots, x_d)$ be the mapping from feature values to predicted target value via the ensemble.

The *partial dependence function* of T on x_k, denoted $T_k(x_k)$, is the average of T with x_k fixed and $x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_d$ varying over the training observations.

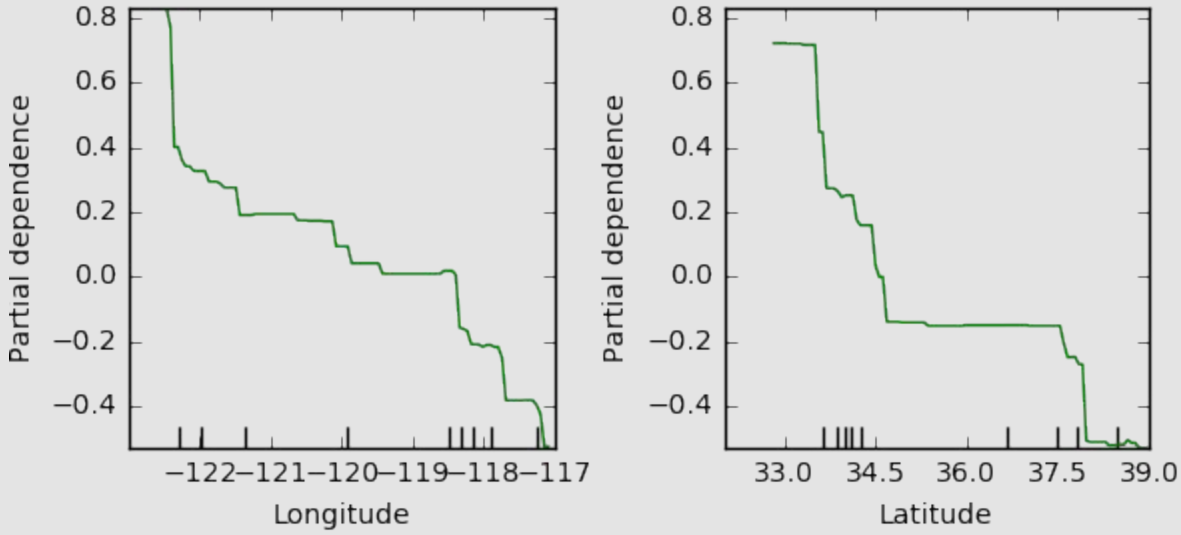
That is, for any given value of one feature, let the values of the other features vary over the training observations, and take the average predicted target value.

T is a function of d variables, which may be hard to understand if d is even 3, whereas $T_k(x_k)$ is a function of one variable, which is easier to understand, especially by plotting it.

Partial dependence functions and plots — 1-dimensional

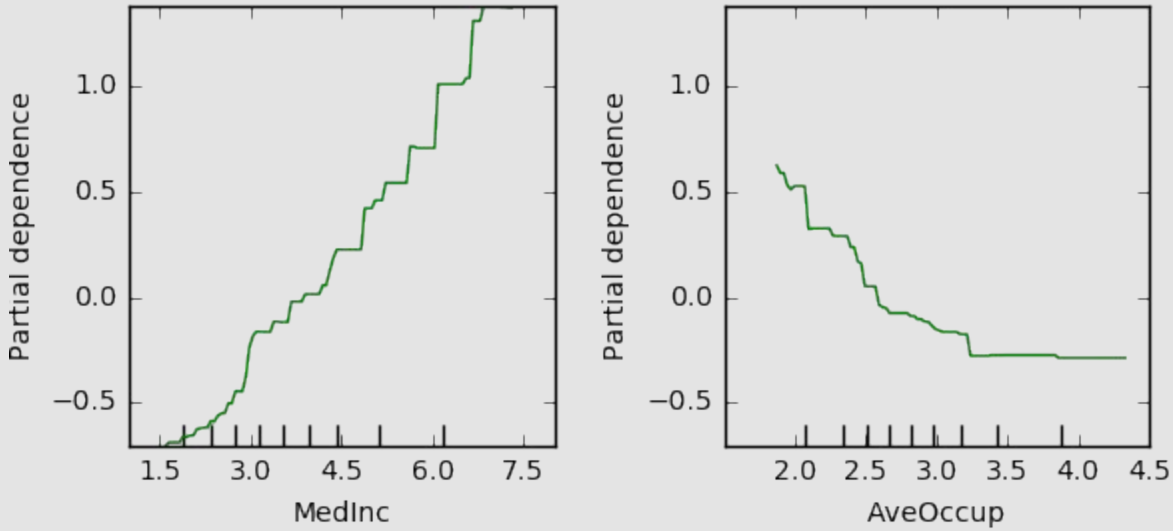
```
>>> from sklearn.ensemble.partial_dependence import plot_partial_dependence

>>> # Features 7 and 6 are longitude and latitude.
... plot_partial_dependence(gbr, housing.data, [7, 6],
...                         feature_names = housing.feature_names)
```



Partial dependence functions and plots — 1-dimensional

```
>>> # Features 0 and 5 are median income and average occupancy (people/household).
... plot_partial_dependence(gbr, housing.data, [0, 5],
...                         feature_names = housing.feature_names)
```



Partial dependence functions and plots — 2-dimensional

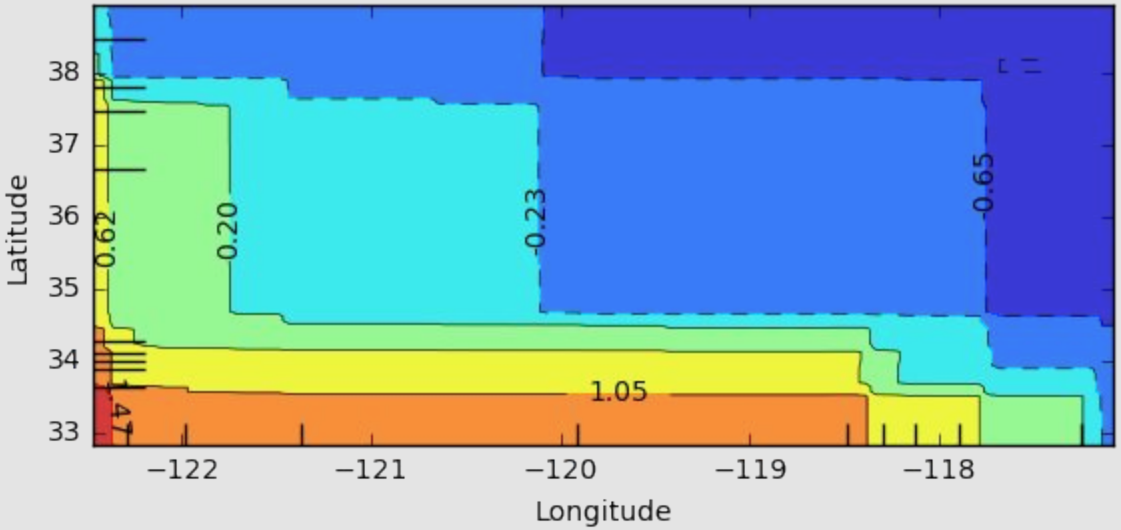
Likewise, the partial dependence function of T on x_j and x_k, denoted $T_{jk}(x_j, x_k)$, is the average of T with x_j and x_k fixed and $x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_{k-1}, x_{k+1}, \ldots, x_d$ varying over the training observations.

That is, for any given values of two features, let the values of the other features vary over the training observations, and take the average predicted target value.

$T_{jk}(x_j, x_k)$ can help with understanding the interaction, if any, between two variables.

Partial dependence functions and plots — 2-dimensional

```
>>> # Features 7 and 6 are longitude and latitude.
... plot_partial_dependence(gbr, housing.data, [(7, 6)],
...                         feature_names = housing.feature_names)
```



Feature interactions

Two features *interact* if they combine non-additively to affect the target.

Equivalently, two features x_j and x_k **don't** interact if T is the sum of two functions, one of which doesn't depend on x_j and the other of which doesn't depend on x_k.

Similarly, three features don't interact if T is the sum of three functions, each of which doesn't depend on one of the features.

Etc., although higher-order interactions tend to be weaker.

Feature interactions — Decision-tree depth

A decision tree can automatically capture interactions (unlike traditional regression, where each interaction requires an explicit term in the model formula).

A decision tree of depth D can capture interactions of order up to D.

One way to see whether interactions of order D are important is to fit two gradient-boosted tree models, one with maximum depth D and the other with maximum depth $D-1$, and see whether the first model is appreciably more accurate.

Feature interactions — Decision-tree depth

```
>>> gbr_2 = GradientBoostingRegressor(max_depth = 2, random_state = 42)

>>> gbr_2.fit(data_train, target_train)
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
...

>>> gbr_2.score(data_test, target_test)
0.78301713864372346

>>> gbr_1 = GradientBoostingRegressor(max_depth = 1, random_state = 42)

>>> gbr_1.fit(data_train, target_train)
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
...

>>> gbr_1.score(data_test, target_test)
0.64537811440179915
```

Feature interactions — *H* statistics

A useful topic from the research literature.

Let $T^c_k(x_k)$ be the *centered partial dependence function* of T on x_k, $T^c_k(x_k) = T_k(x_k) - \text{ave}[\, T_k(x_k) \,]$, where the average is over the training observations. Define $T^c_{jk}(x_j, x_k)$ analogously.

If two features x_j and x_k don't interact, then it's easy to show that $T^c_{jk}(x_j, x_k) = T^c_j(x_j) + T^c_k(x_k)$.

So to measure the strength of an interaction between x_j and x_k, Friedman and Popescu defined

$$H_{jk}^2 = \text{sum}[\, (T^c_{jk}(x_j, x_k) - T^c_j(x_j) - T^c_k(x_k))^2 \,] \,/\, \text{sum}[\, T^c_{jk}(x_j, x_k)^2 \,]$$

where the sums are over the training observations. They defined higher-order *H* statistics too. An *H* statistic varies from 0 to 1, and the larger it is, the stronger the evidence for an interaction.

(See Jerome H. Friedman and Bogdan E. Popescu, 2008, "Predictive learning via rule ensembles", *Ann. Appl. Stat.* **2**:916–954, http://projecteuclid.org/download/pdfview_1/euclid.aoas/1223908046, s. 8.1.)

Feature interactions — *H* statistics

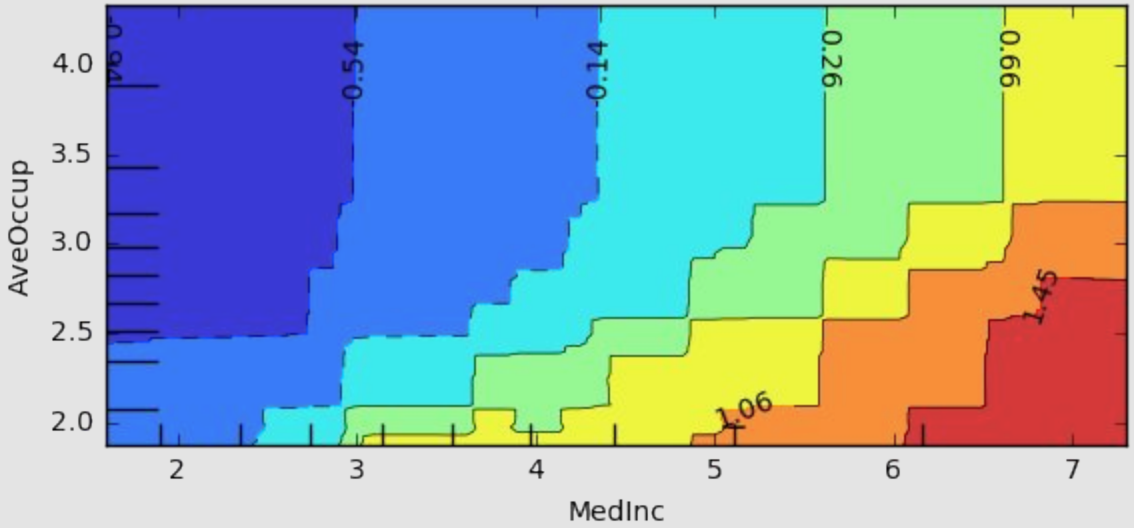
```
>>> # You may need to do "pip install sklearn-gbmi".
... from sklearn_gbmi import *

>>> # Features 7, 6, 0, and 5 are longitude, latitude, med. inc., and ave. occup.
... h_vals = h_all_pairs(gbr, data_train, [7, 6, 0, 5])

>>> rows = []; cols = ['feature 1', 'feature 2', 'h']
>>> for (f1, f2), h_val in h_vals.iteritems():
...     rows.append([housing.feature_names[f1], housing.feature_names[f2], h_val])
>>> pd.DataFrame(rows, columns = cols).set_index(['feature 1', 'feature 2']).h \
...     .sort_values(ascending = False)
feature 1  feature 2
Longitude  Latitude   0.596719
MedInc     AveOccup   0.142883
Latitude   AveOccup   0.104409
Longitude  AveOccup   0.061841
Latitude   MedInc     0.061684
Longitude  MedInc     0.033325
Name: h, dtype: float64
```

Feature interactions — *H* statistics

```
>>> # Features 0 and 5 are median income and average occupancy (people/household).
... plot_partial_dependence(gbr, housing.data, [(0, 5)],
...                         feature_names = housing.feature_names)
```



So an ensemble of decision trees is still interpretable, using feature importances, partial dependence functions and plots, and *H* statistics.