Lecture 06
Decision Trees I

08 February 2016

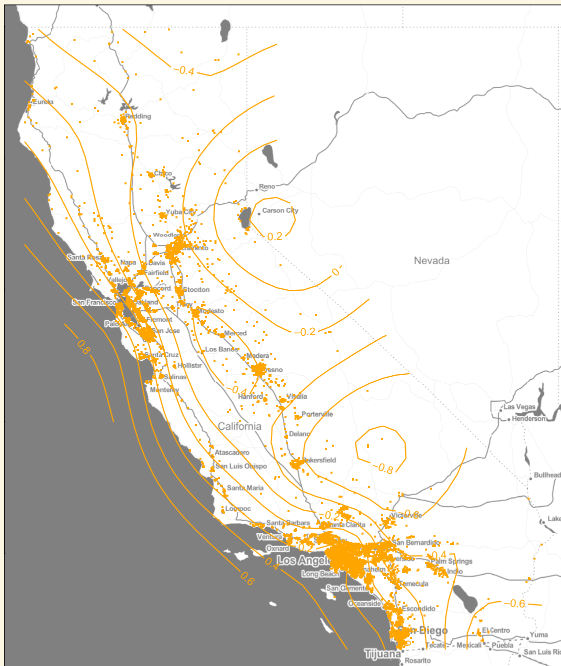
Taylor B. Arnold
Yale Statistics
STAT 365/665

- Problem Set #2 Posted – Due February 19th
- Piazza site `https://piazza.com/`

Last time we starting fitting additive models to the variables in the California pricing data.

As I alluded to, it actually makes more sense to allow an interaction between latitude and longitude. This is easy to include in `mgcv`:

```
> ca <- read.csv("ca.csv", as.is=TRUE)
> library(mgcv)
> ca.gam2 <- gam(log(median_house_value)
+   ~ s(median_household_income) + s(mean_household_income)
+   + s(population) + s(total_units) + s(vacant_units)
+   + s(owners) + s(median_rooms) + s(mean_household_size_owners)
+   + s(mean_household_size_renters)
+   + s(longitude,latitude), data=ca, subset=trainFlag)
```



Eureka
Redding
Chico
Reno
Carson City
Yuba City
Nevada
Santa Rosa
Napa
Davis
Fairfield
Vallejo
Stockton
San Francisco
Fremont
San Jose
Modesto
Merced
Los Banos
Madera
Santa Cruz
Hollister
Salinas
Monterey
Hanford
Visalia
Porterville
California
Delano
Las Vegas
Henderson
Atascadero
San Luis Obispo
Bakersfield
Santa Maria
Bullhead
Lompoc
Santa Barbara
Ventura
Oxnard
San Bernardino
Palm Springs
Indio
Long Beach
Temecula
Oceanside
Escondido
El Centro
Yuma
Tecate
Mexicali
Puebla
Tijuana
Rosarito

How well does these methods do in terms of prediction? We can predict using the `predict` function just as with linear models:

```
> y <- log(ca$median_house_value)
> ca.lm.pred <- predict(ca.lm, ca)
> ca.gam.pred <- predict(ca.gam, ca)
> ca.gam2.pred <- predict(ca.gam2, ca)
```

And then check the mean squared error on both the training set and testing set:

```
> tapply((ca.lm.pred - y)^2, trainFlag, mean)
FALSE  TRUE
0.096 0.101
> tapply((ca.gam.pred - y)^2, trainFlag, mean)
FALSE  TRUE
0.064 0.072
> tapply((ca.gam2.pred - y)^2, trainFlag, mean)
FALSE  TRUE
0.059 0.065
```

In machine learning, you'll often hear the caveat that our conclusions always depends on future values following the same underlying model. I think we say that a lot, but forget to really think about it. To illustrate, let's re-fit the model on the California data without the latitude and longitude components. We can then see how well the model trained on California data generalizes to Pennsylvania data.

Here are the two linear models fit on the two different datasets.

```
> ca.lm2 <- lm(log(median_house_value) ~ median_household_income
+   + mean_household_income + population + total_units +
+   + vacant_units + owners + median_rooms +
+   + mean_household_size_owners + mean_household_size_renters,
+   data = ca, subset=trainFlag)
>
> pa.lm3 <- lm(log(median_house_value) ~ median_household_income
+   + mean_household_income + population + total_units +
+   + vacant_units + owners + median_rooms +
+   + mean_household_size_owners + mean_household_size_renters,
+   data = pa, subset=trainFlagPa)
```

And here are the two additive models fit on the two datasets:

```
> ca.gam3 <- gam(log(median_house_value)
+    ~ s(median_household_income) + s(mean_household_income)
+    + s(population) + s(total_units) + s(vacant_units)
+    + s(owners) + s(median_rooms) + s(mean_household_size_owners)
+    + s(mean_household_size_renters), data=ca, subset=trainFlag)
>
> pa.gam4 <- gam(log(median_house_value)
+    ~ s(median_household_income) + s(mean_household_income)
+    + s(population) + s(total_units) + s(vacant_units)
+    + s(owners) + s(median_rooms) + s(mean_household_size_owners)
+    + s(mean_household_size_renters), data=pa, subset=trainFlagPa)
```

Finding the predicted values from these models all on the PA data:

```
> y.pa <- log(pa$median_house_value)
> pa.lm2.pred <- predict(ca.lm2, pa)
> pa.gam3.pred <- predict(ca.gam3, pa)
> pa.lm3.pred <- predict(pa.lm3, pa)
> pa.gam4.pred <- predict(pa.gam4, pa)
```

We see that the California ones yield very poor MSE scores for PA:

```
> tapply((pa.lm2.pred - y.pa)^2,trainFlagPa,mean)
FALSE  TRUE
 0.58  0.55
> tapply((pa.gam3.pred - y.pa)^2,trainFlagPa,mean)
FALSE  TRUE
 0.47  0.44
```

Compared to those models trained on the PA data:

```
> tapply((pa.lm3.pred - y.pa)^2,trainFlagPa,mean)
FALSE  TRUE
0.095 0.093
> tapply((pa.gam4.pred - y.pa)^2,trainFlagPa,mean)
FALSE  TRUE
0.070 0.063
```

However, if we account for the overall means being different, we see that the California models perform reasonably well on the Pennsylvania data:

```
> tapply((pa.lm2.pred - y.pa),trainFlagPa,var)
FALSE  TRUE
 0.14  0.13
> tapply((pa.gam3.pred - y.pa),trainFlagPa,var)
FALSE  TRUE
0.094 0.084
> tapply((pa.lm3.pred - y.pa),trainFlagPa,var)
FALSE  TRUE
0.095 0.093
> tapply((pa.gam4.pred - y.pa),trainFlagPa,var)
FALSE  TRUE
0.070 0.063
```

Tree-Based Models

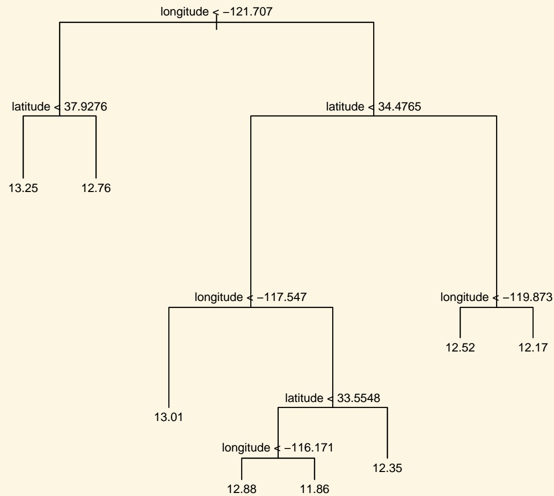
In order to deal with the combinatorial explosion of modeling every interaction in higher-dimensional non-parametric regression and classification models, we have seen that additive models assume that there are no interactions between the input variables.

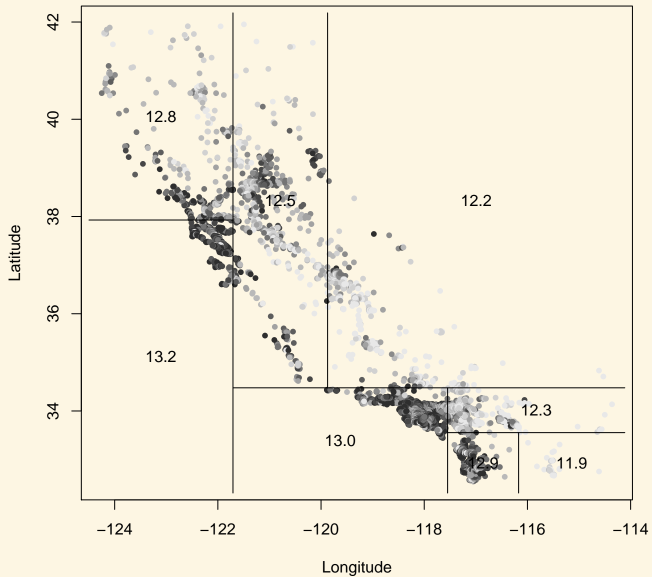
In order to deal with the combinatorial explosion of modeling every interaction in higher-dimensional non-parametric regression and classification models, we have seen that additive models assume that there are no interactions between the input variables.

Tree-based models instead allow interactions, but use the data to greedily determine which interactions to include.

I think it is easiest to understand tree models by first seeing an example before backing into the formal definition of how these models are fit.

```
> tf <- tree(log(median_house_value) ~ longitude + latitude, data = ca)
> plot(tf)
> text(tf, cex=0.75)
```


longitude < −121.707
latitude < 37.9276
latitude < 34.4765
13.25
12.76
longitude < −117.547
longitude < −119.873
12.52
12.17
latitude < 33.5548
13.01
longitude < −116.171
12.35
12.88
11.86


42
40
38
36
34
Latitude
−124
−122
−120
−118
−116
−114
Longitude
12.8
12.5
12.2
13.2
13.0
12.3
12.9
11.9

Regression trees

The fitting algorithm for learning such a tree is as follows:

1. Consider splitting on every possible unique value of every single variable in the data. Pick the 'best' split from amongst all of these options.

Regression trees

The fitting algorithm for learning such a tree is as follows:

1. Consider splitting on every possible unique value of every single variable in the data. Pick the 'best' split from amongst all of these options.
2. Partition the training data into two classes based on step 1.

Regression trees

The fitting algorithm for learning such a tree is as follows:

1. Consider splitting on every possible unique value of every single variable in the data. Pick the 'best' split from amongst all of these options.
2. Partition the training data into two classes based on step 1.
3. Now, iteratively apply step 1 separately to each partition of the data.

Regression trees

The fitting algorithm for learning such a tree is as follows:

1. Consider splitting on every possible unique value of every single variable in the data. Pick the 'best' split from amongst all of these options.
2. Partition the training data into two classes based on step 1.
3. Now, iteratively apply step 1 separately to each partition of the data.
4. Continue splitting each subset until an appropriate stopping criteria has been reached.

Regression trees

The fitting algorithm for learning such a tree is as follows:

1. Consider splitting on every possible unique value of every single variable in the data. Pick the 'best' split from amongst all of these options.
2. Partition the training data into two classes based on step 1.
3. Now, iteratively apply step 1 separately to each partition of the data.
4. Continue splitting each subset until an appropriate stopping criteria has been reached.
5. Calculate the mean of all the training data in a terminal node (i.e., 'leaf") of the learned tree structure. Use this as the predicted value for future inputs that fall within the same bucket.

Stopping criterion

There are many commonly used stopping criterion, often used simultaneously (if any is satisfied stop splitting the partition):

1. minimum number of training samples in a node
2. maximum number of splits
3. minimum improvement in the best split
4. maximum depth of the tree

In practice, particularly for larger datasets, the maximum number of splits is the mostly commonly used.

Measuring 'best' splits

A simple way of measuring how good a partition of the dataset is is to use the residual sum of squares from the predicted values that would be implied by the partition. So for the partition I, we have:

$$\sum_{i\in I}(y_i - \Sigma_{i\in I}(y_i))^2 + \sum_{i\notin I}(y_i - \Sigma_{i\notin I}(y_i))^2$$

Measuring 'best' splits, cont.

Notice that using the standard trick with variance calculations we can simplify this. Setting $\Sigma_{i \in I}(y_i) = \bar{y}_I$, $\Sigma_{i \notin I}(y_i) = \bar{y}_{I^c}$ and n_I and n_{I^c} as the sample sizes of the partitions, we have:

$$\begin{aligned}
\sum_{i \in I}(y_i - \bar{y}_I)^2 + \sum_{i \notin I}(y_i - \bar{y}_{I^c})^2 &= \sum_{i \in I}\left(y_i^2 + \bar{y}_I^2 - 2y_i\bar{y}_I\right) + \sum_{i \notin I}\left(y_i^2 + \bar{y}_{I^c}^2 - 2y_i\bar{y}_{I^c}\right) \\
&= \sum_{i \in I}\left(y_i\right)^2 - n_I \cdot \bar{y}_I^2 + \sum_{i \notin I}\left(y_i\right)^2 - n_{I^c} \cdot \bar{y}_{I^c}^2 \\
&= \sum_{i}\left(y_i\right)^2 - n_I \cdot \bar{y}_I^2 - n_{I^c} \cdot \bar{y}_{I^c}^2
\end{aligned}$$

Measuring 'best' splits, cont.

Notice that using the standard trick with variance calculations we can simplify this. Setting $\Sigma_{i\in I}(y_i) = \bar{y}_I$, $\Sigma_{i\notin I}(y_i) = \bar{y}_{I^c}$ and n_I and n_{I^c} as the sample sizes of the partitions, we have:

$$\begin{aligned}
\sum_{i\in I}(y_i - \bar{y}_I)^2 + \sum_{i\notin I}(y_i - \bar{y}_{I^c})^2 &= \sum_{i\in I}\left(y_i^2 + \bar{y}_I^2 - 2y_i\bar{y}_I\right) + \sum_{i\notin I}\left(y_i^2 + \bar{y}_{I^c}^2 - 2y_i\bar{y}_{I^c}\right) \\
&= \sum_{i\in I}\left(y_i\right)^2 - n_I\cdot\bar{y}_I^2 + \sum_{i\notin I}\left(y_i\right)^2 - n_{I^c}\cdot\bar{y}_{I^c}^2 \\
&= \sum_{i}\left(y_i\right)^2 - n_I\cdot\bar{y}_I^2 - n_{I^c}\cdot\bar{y}_{I^c}^2
\end{aligned}$$

The first term does not change, so the goal is to actually minimize the following:

$$\arg\max_{\forall I} \frac{1}{n_I}\cdot\left(\sum_{i\in I} y_i\right)^2 + \frac{1}{n_{I^c}}\cdot\left(\sum_{i\notin I} y_i\right)^2$$

Computational details

If we sort the responses y_i such that their corresponding x values are increasing for a particular variable, we can write this even more compactly as:

$$\underset{j=1,2,\ldots n-1}{\arg\max} \left\{ \frac{1}{j} \cdot \left(\sum_{i \leq j} y_i \right)^2 + \frac{1}{n-j} \cdot \left(\sum_{i > j} y_i \right)^2 \right\}$$

If we are clever, these can be calculated in only $\mathcal{O}(n)$ operations.

Computational details

If we sort the responses y_i such that their corresponding x values are increasing for a particular variable, we can write this even more compactly as:

$$\underset{j=1,2,\ldots n-1}{\arg\max} \left\{ \frac{1}{j} \cdot \left(\sum_{i \leq j} y_i \right)^2 + \frac{1}{n-j} \cdot \left(\sum_{i > j} y_i \right)^2 \right\}$$

If we are clever, these can be calculated in only $\mathcal{O}(n)$ operations.

If we have a maximum of K splits, the total computational cost of fitting a regression tree can be bounded by $\mathcal{O}(np \cdot (\log(n) + K))$, but requires us to store p copies of the n response variables.

Trees as Adaptive knn

Notice that the final predictions of a decision tree amount to simply averaging all of responses for samples that are in a given bucket. In this way, trees are like k-nearest neighbors, except that what defines 'near' is learned from the data rather than blindly using distances in the x-space...This helps to combat the curse of dimensionality but may over-fit the training data.

Note that decision trees are **not** linear smoothers because the weights are not independent functions of the x values.

Classification trees

By coding a class label as ± 1, and using the one-vs-many trick, we can directly apply the regression tree algorithm to classification problems. This actually works fine for two class problems, but is inefficient for multi-class ones. It is better to instead use a measurement of the goodness of a partition that directly includes all of the class labels. Choices include mutual entropy and multinomial deviance.

Categorical and ordinal prediction variables

If a predictor variable is an ordinal variable, such as letter grades or terms such as 'low', 'medium' and 'high', these can be mapped to numerical values in a natural way in tree based models.

Unordered categorical variables can be converted to factor levels as in a linear regression, however there is another option: when considering the split points of the categorical variable simply choose between all possible permutations of the categories. This is feasible for up to several dozen variables, depending on the amount of data and how long you are willing to wait on the results.

Why trees?

- able to interpret the important variables in a model
- handles missing values well in both training and testing data
- reasonably fast to train
- does not depend on the scale of the predictor variables
- can handle multiclass problems directly
- works natively with categorical variables
- easily modified to be robust to outliers (just replace MSE with MAD for evaluating splits; means with medians for predictions in terminal nodes)
- very fast to classify new points
- robust to tuning parameters (the stopping criteria), *for a given set of data*

Stability

Regression and classification trees are robust to the stopping criterion, but very sensitive to the input data used from training. Removing some small fraction, say 10%, of the training data can often lead to an extremely different set of predictions.

Fortunately, we can use this instability to our advantage!

Random forests

Random forests fit many decision trees to the same training data. Each decision tree is fit using the aforementioned procedure except that:

- the training set for a given tree is sampled from the original data, with or without replacement
- at each training split, only a randomly chosen subset (of size m) of the variable are considered for splitting the data

In order to predict new values from this ensemble of trees, we take the predictions from each model and average them together (for classification, the class labels can also be used as 'votes').

Random forest tuning parameters

The main tuning parameters for random forests are the total number of trees T, the maximum depth of the trees K and the number of variables m randomly chosen to be available at each split.

Conveniently, T and K typically just need to be large enough and so are relatively easy to set in practice. Values of m typically range from $\sqrt{m}$ up to $m/3$. This parameter can have a large effect on the model, but is also generally easy to find a reasonable value and not extremely sensitive in most cases.

Random forest, computational details

Random forests have three very convenient computational benefits. First of all, each tree is learned independently and so can be fit in parallel. Over a cluster this requires a local copy of the data, but on a single machine the data can be shared across workers as the raw data is never modified.

Secondly, the sorting scheme we described earlier, where the sample responses are sorted based on each variable, can be (carefully) used as-is for each tree. That is, we only need to sort the data once and can use this sorted set for each tree. We just simply apply weights to this set according the sampling scheme used on each tree.

Finally, the fact that we only need to consider m variables at each split reduces the cost of fitting a given tree.

Random forest, computational details, cont.

In total, a random forest can be fit in

$$\mathcal{O}(np \log(p) + (KT) \cdot (mn))$$

operations when running serially. Using C parallel cores, this can reduced to only:

$$\mathcal{O}(np \log(p) + (KT/C) \cdot (mn))$$

Random forest, computational details, cont.

If the data is too large to fit into memory, one can set the sub-sampling rate small enough to allow each tree to load its data into memory (though this requires re-sorting each time). Alternatively, each node in a cluster can locally run a random forest only on its individual chunk of data; this often produces reasonable results.

There are also some completely different approaches which either involve very low latency message passing at each node, or discretizing the input variables and representing the problem at each node as a two-way table (see MLLib for an example of the latter).

Out-of-bag error

Notice that for any given observation i, there should be a set of the trees in a random forest such that i was not randomly chosen to train them. Because i was not used to train these trees, if we look at only the predictions from this set and compare it to y_i, this is an unbiased view of how good the tree is fitting the data.

Using these predicted values for each i, we can efficiently calculate a variant of cross-validation. This is called the out-of-bag error.

Variable importance

We can associated each node with a score indicating how much that node decreased the overall mean squared error of the training data for that tree. Taking the sum of these scores for a given variable across all of the trees is a reasonable proxy for how important a variable is in the prediction algorithm.