Chapter 13

Classification and Regression Trees

[[TODO: Notes taken from another course; integrate]]

Having built up increasingly complicated models for regression, I'll now switch gears and introduce a class of nonlinear predictive model which at first seems too simple to possible work, namely **prediction trees**. These have two varieties, **regression trees** and **classification trees**.

[[TODO: New opening]]

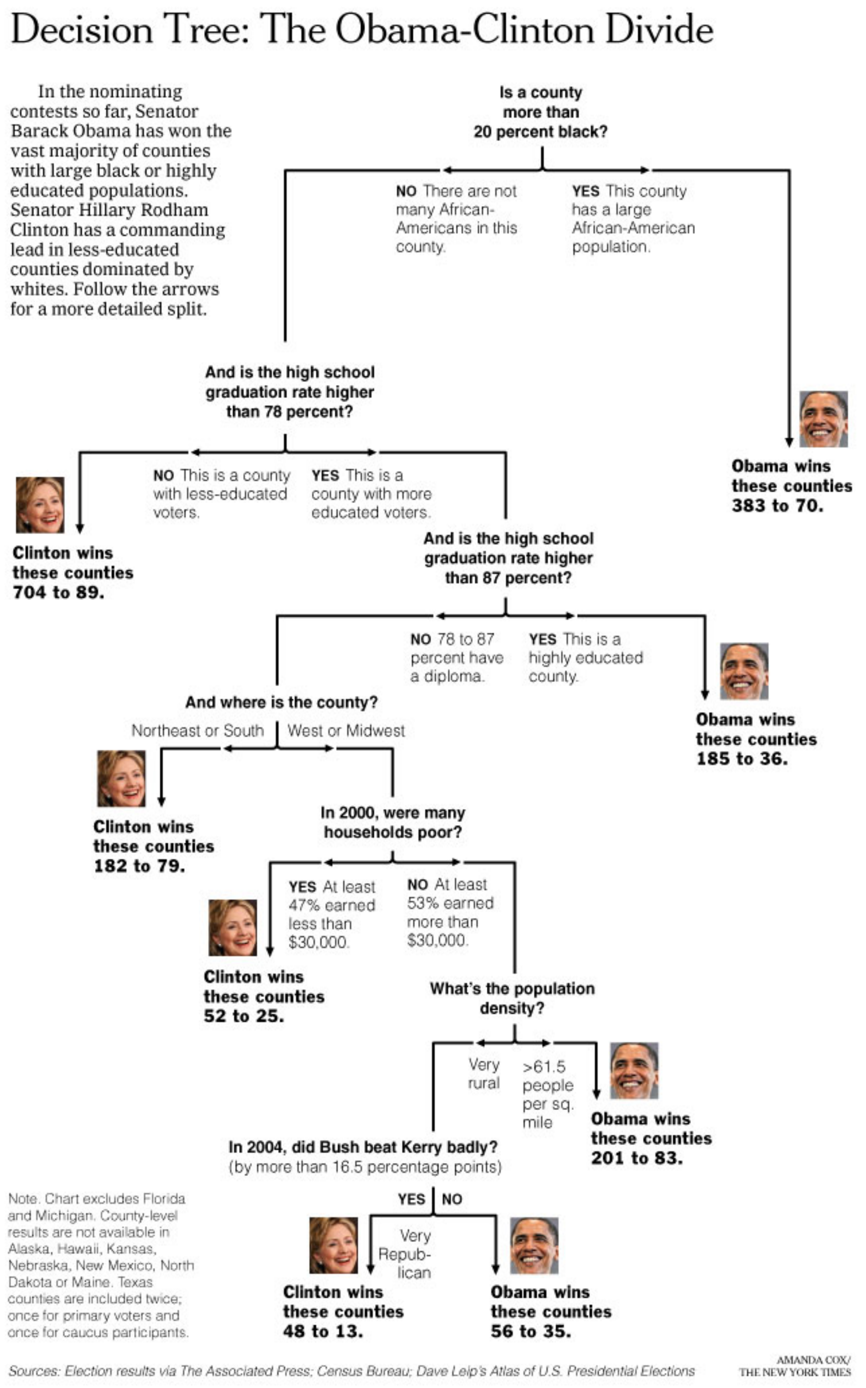
13.1 Prediction Trees

The basic idea is very simple. We want to predict a response or class Y from inputs $X_1, X_2, \ldots X_p$. We do this by growing a binary tree. At each internal node in the tree, we apply a test to one of the inputs, say X_i. Depending on the outcome of the test, we go to either the left or the right sub-branch of the tree. Eventually we come to a leaf node, where we make a prediction. This prediction aggregates or averages all the training data points which reach that leaf. Figure 13.1 should help clarify this.

Why do this? Predictors like linear or polynomial regression are **global models**, where a single predictive formula is supposed to hold over the entire data space. When the data has lots of features which interact in complicated, nonlinear ways, assembling a single global model can be very difficult, and hopelessly confusing when you do succeed. Some of the non-parametric smoothers try to fit models **locally** and then paste them together, but again they can be hard to interpret. (Additive models are at least pretty easy to grasp.)

An alternative approach to nonlinear regression is to sub-divide, or **partition**, the space into smaller regions, where the interactions are more manageable. We then partition the sub-divisions again — this is **recursive partitioning** (as in hierarchical clustering) — until finally we get to chunks of the space which are so tame that we can fit simple models to them. The global model thus has two parts: one is just the recursive partition, the other is a simple model for each cell of the partition.

Now look back at Figure 13.1 and the description which came before it. Prediction trees use the tree to represent the recursive partition. Each of the **terminal**


Decision Tree: The Obama-Clinton Divide

In the nominating contests so far, Senator Barack Obama has won the vast majority of counties with large black or highly educated populations. Senator Hillary Rodham Clinton has a commanding lead in less-educated counties dominated by whites. Follow the arrows for a more detailed split.

Note. Chart excludes Florida and Michigan. County-level results are not available in Alaska, Hawaii, Kansas, Nebraska, New Mexico, North Dakota or Maine. Texas counties are included twice; once for primary voters and once for caucus participants.

Sources: Election results via The Associated Press; Census Bureau; Dave Leip's Atlas of U.S. Presidential Elections

Is a county more than 20 percent black?
NO There are not many African-Americans in this county.
YES This county has a large African-American population.
Obama wins these counties 383 to 70.
And is the high school graduation rate higher than 78 percent?
NO This is a county with less-educated voters.
YES This is a county with more educated voters.
Clinton wins these counties 704 to 89.
And is the high school graduation rate higher than 87 percent?
NO 78 to 87 percent have a diploma.
YES This is a highly educated county.
Obama wins these counties 185 to 36.
And where is the county?
Northeast or South
West or Midwest
Clinton wins these counties 182 to 79.
In 2000, were many households poor?
YES At least 47% earned less than $30,000.
NO At least 53% earned more than $30,000.
Clinton wins these counties 52 to 25.
What's the population density?
Very rural
>61.5 people per sq. mile
Obama wins these counties 201 to 83.
In 2004, did Bush beat Kerry badly? (by more than 16.5 percentage points)
YES
NO
Very Repub-lican
Clinton wins these counties 48 to 13.
Obama wins these counties 56 to 35.
AMANDA COX/ THE NEW YORK TIMES

FIGURE 13.1: *Classification tree for county-level outcomes in the 2008 Democratic Party primary (as of April 16), by Amanada Cox for the New York* Times.

nodes, or **leaves**, of the tree represents a cell of the partition, and has attached to it a simple model which applies in that cell only. A point x **belongs** to a leaf if x falls in the corresponding cell of the partition. To figure out which cell we are in, we start at the **root node** of the tree, and ask a sequence of questions about the features. The interior nodes are labeled with questions, and the edges or branches between them labeled by the answers. Which question we ask next depends on the answers to previous questions. In the classic version, each question refers to only a single attribute, and has a yes or no answer, e.g., "Is `HSGrad` $>$ `0.78`?" or "Is `Region` $==$ MIDWEST?" The variables can be of any combination of types (continuous, discrete but ordered, categorical, etc.). You could do more-than-binary questions, but that can always be accommodated as a larger binary tree. Asking questions about multiple variables at once is, again, equivalent to asking multiple questions about single variables.

That's the recursive partition part; what about the simple local models? For classic regression trees, the model in each cell is just a *constant* estimate of Y. That is, suppose the points $(x_i, y_i), (x_2, y_2), \ldots (x_c, y_c)$ are all the samples belonging to the leaf-node l. Then our model for l is just $\hat{y} = \frac{1}{c}\sum_{i=1}^{c} y_i$, the sample mean of the response variable in that cell. This is a piecewise-constant model.[1] There are several advantages to this:

- Making predictions is fast (no complicated calculations, just looking up constants in the tree)
- It's easy to understand what variables are important in making the prediction (look at the tree)
- If some data is missing, we might not be able to go all the way down the tree to a leaf, but we can still make a prediction by averaging all the leaves in the sub-tree we do reach
- The model gives a jagged response, so it can work when the true regression surface is not smooth. If it is smooth, though, the piecewise-constant surface can approximate it arbitrarily closely (with enough leaves)
- There are fast, reliable algorithms to learn these trees

A last analogy before we go into some of the mechanics. One of the most comprehensible non-parametric methods is k-nearest-neighbors: find the points which are most similar to you, and do what, on average, they do. There are two big drawbacks to it: first, you're defining "similar" entirely in terms of the inputs, not the response; second, k is constant everywhere, when some points just might have more very-similar neighbors than others. Trees get around both problems: leaves correspond to regions of the input space (a neighborhood), but one where the *responses* are similar, as well as the inputs being nearby; and their size can vary arbitrarily. Prediction trees are *adaptive* nearest-neighbor methods.

[1]We could instead fit, say, a different linear regression for the response in each leaf node, using only the data points in that leaf (and using dummy variables for non-quantitative features). This would give a piecewise-linear model, rather than a piecewise-constant one. If we've built the tree well, however, all the points in each leaf are pretty similar, so the regression surface would be nearly constant anyway.

13.2 Regression Trees

[[TODO: Update to more modern California data]]

Let's start with an example.

13.2.1 Example: California Real Estate Again

We'll revisit the Califonia house-price data from Chapter 9, where we try to predict the median house price in each census tract of California from the attributes of the houses and of the inhabitants of the tract. We'll try growing a regression tree for this.

[[TODO: Add citation]]

There are several R packages for regression trees; the easiest one is called, simply, `tree`.

```
calif = read.table("http://www.stat.cmu.edu/~cshalizi/350/hw/06/cadata.dat",
    header = TRUE)
require(tree)
treefit = tree(log(MedianHouseValue) ~ Longitude + Latitude, data = calif)
```

This does a tree regression of the log price on longitude and latitude. What does this look like? Figure 13.2 shows the tree itself; Figure 13.3 shows the partition, overlaid on the actual prices in the state. (The ability to show the partition is why I picked only two input variables.)

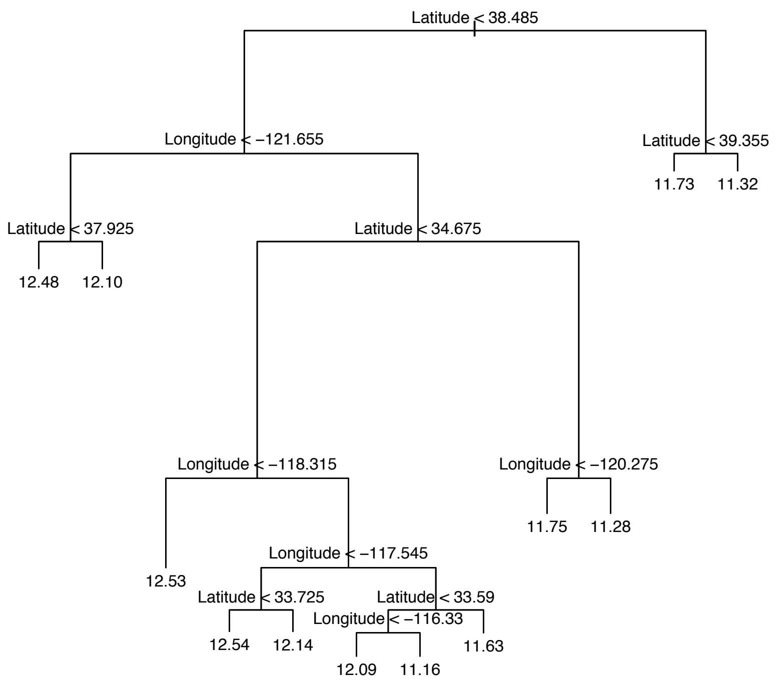
Qualitatively, this looks like it does a fair job of capturing the interaction between longitude and latitude, and the way prices are higher around the coasts and the big cities. Quantitatively, the error isn't bad:

```
summary(treefit)
##
## Regression tree:
## tree(formula = log(MedianHouseValue) ~ Longitude + Latitude,
##     data = calif)
## Number of terminal nodes:  12
## Residual mean deviance:  0.1662 = 3429 / 20630
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.75900 -0.26080 -0.01359  0.00000  0.26310  1.84100
```

[[TODO: replace hard-coding with R]]

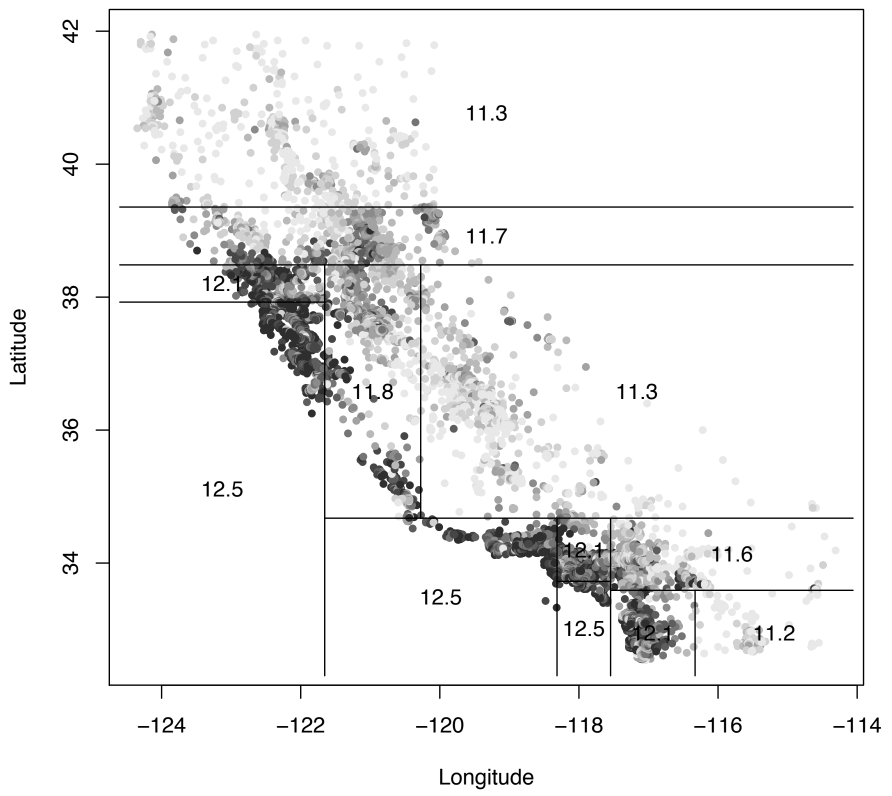
Here "deviance" is just mean squared error; this gives us an RMS error of 0.41 , which is higher than the smooth non-linear models in Chapter 9, but not shocking since we're using only two variables, and have only twelve nodes.

The flexibility of a tree is basically controlled by how many leaves they have, since that's how many cells they partition things into. The `tree` fitting function has a number of controls settings which limit how much it will grow — each node has to contain a certain number of points, and adding a node has to reduce the error by at least a certain amount. The default for the latter, `mindev`, is 0.01; let's turn it down and see what happens.



```
plot(treefit)
text(treefit, cex = 0.75)
```

FIGURE 13.2: *Regression tree for predicting California housing prices from geographic coordinates. At each internal node, we ask the associated question, and go to the left child if the answer is "yes", to the right child if the answer is "no". Note that leaves are labeled with* log *prices; the plotting function isn't flexible enough, unfortunately, to apply transformations to the labels.*



```
price.deciles = quantile(calif$MedianHouseValue, 0:10/10)
cut.prices = cut(calif$MedianHouseValue, price.deciles, include.lowest = TRUE)
plot(calif$Longitude, calif$Latitude, col = grey(10:2/11)[cut.prices], pch = 20,
    xlab = "Longitude", ylab = "Latitude")
partition.tree(treefit, ordvars = c("Longitude", "Latitude"), add = TRUE)
```

FIGURE 13.3: *Map of actual median house prices (color-coded by decile, darker being more expensive), and the partition of the* `treefit` *tree.*

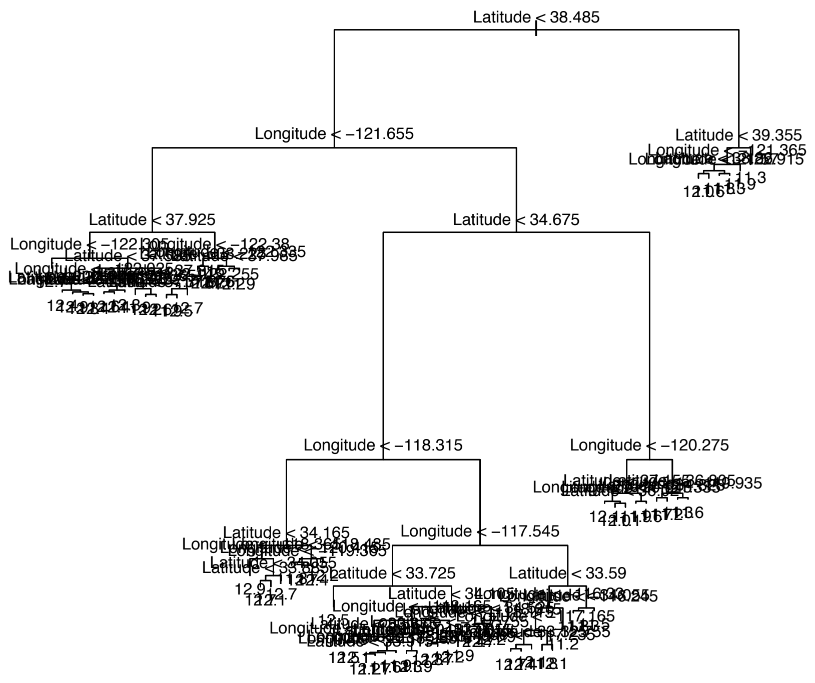
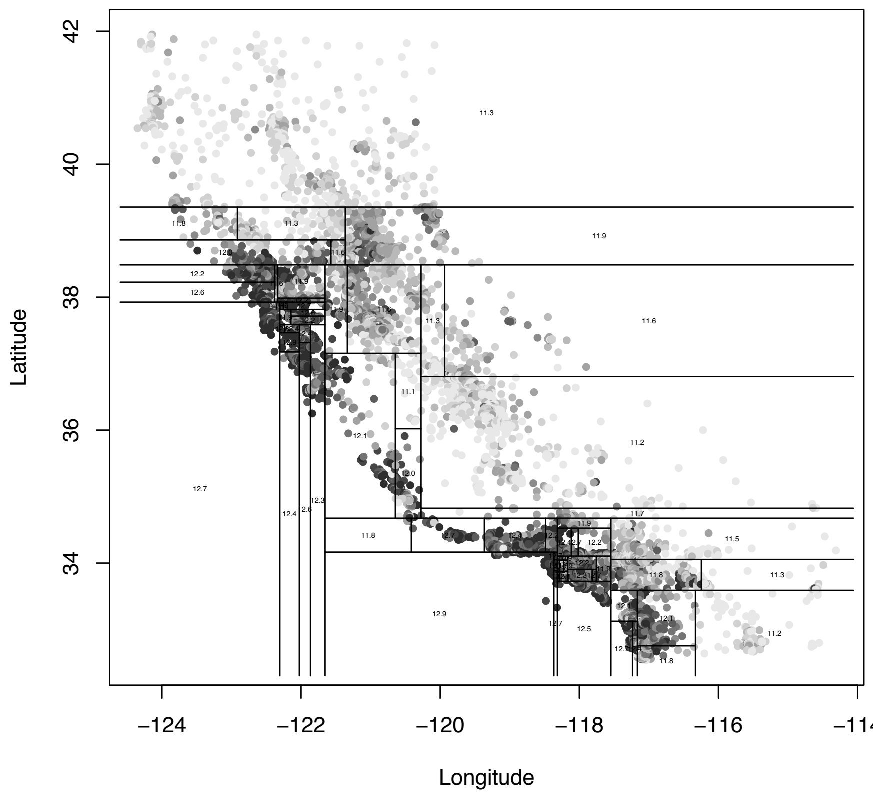


FIGURE 13.4: *As Figure 13.2, but allowing splits for smaller reductions in error (*`mindev=0.001` *rather than the default* `mindev=0.01`*).*

```
treefit2 = tree(log(MedianHouseValue) ~ Longitude + Latitude, data = calif,
    mindev = 0.001)
```

Figure 13.4 shows the tree itself; with 68 nodes, the plot is fairly hard to read, but by zooming in on any part of it, you can check what it's doing. Figure 13.5 shows the corresponding partition. It's obviously much finer-grained than that in Figure 13.3, and does a better job of matching the actual prices (RMS error 0.32). More interestingly, it doesn't just uniformly divide up the big cells from the first partition; some of the new cells are very small, others quite large. The metropolitan areas get a lot more detail than the Mojave.

Of course there's nothing magic about the geographic coordinates, except that they make for pretty plots. We can include all the input features in our model:



```
plot(calif$Longitude, calif$Latitude, col = grey(10:2/11)[cut.prices], pch = 20,
    xlab = "Longitude", ylab = "Latitude")
partition.tree(treefit2, ordvars = c("Longitude", "Latitude"), add = TRUE, cex = 0.3)
```

FIGURE 13.5: *Partition for* `treefit2`. *Note the high level of detail around the cities, as compared to the much coarser cells covering rural areas where variations in prices are less extreme.*

```
treefit3 <- tree(log(MedianHouseValue) ~ ., data = calif)
```

with the result shown in Figure 13.6. This model has fifteen leaves, as opposed to sixty-eight for `treefit2`, but the RMS error is almost as good (0.36). This is highly interactive: latitude and longitude are only used if the income level is sufficiently low. (Unfortunately, this does mean that we don't have a spatial partition to compare to the previous ones, but we *can* map the predictions; Figure 13.7.) Many of the features, while they were available to the tree fit, aren't used at all.

Now let's turn to how we actually grow these trees.

13.2.2 Regression Tree Fitting

Once we fix the tree, the local models are completely determined, and easy to find (we just average), so all the effort should go into finding a good tree, which is to say into finding a good partitioning of the data.

Ideally, we would like to maximize the information the partition gives us about the response variable. Since we are doing regression, what we would really like is for the conditional mean $\mathbf{E}[Y|X=x]$ to be nearly constant in x over each cell of the partition, and for adjoining cells to have distinct expected values. (It's OK if two cells of the partition far apart have similar average values.) We can't do this directly, so we do a greedy search. We start by finding the one binary question we can ask about the predictors which maximizes the information we get about the average value of Y; this gives us our root node and two daughter nodes.[2] At each daughter node, we repeat our initial procedure, asking which question would give us the maximum information about the average value of Y, given where we already are in the tree. We repeat this recursively.

Every recursive algorithm needs to know when it's done, a **stopping criterion**. Here this means when to stop trying to split nodes. Obviously nodes which contain only one data point cannot be split, but giving each observations its own leaf is unlikely to generalize well. A more typical criterion is something like: halt when each child would contain less than five data points, or when splitting increases the information by less than some threshold. Picking the criterion is important to get a good tree, so we'll come back to it presently.

To really make this operational, we need to be more precise about what we mean by "information about the average value of Y". This can be measured straightforwardly by the mean squared error. The MSE for a tree T is

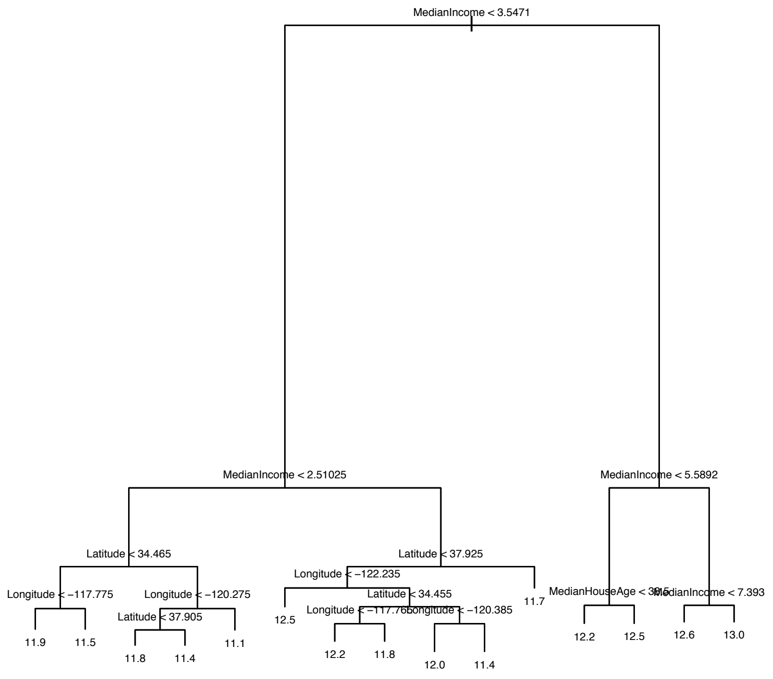
$$MSE(T) = \frac{1}{n} \sum_{c \in \mathrm{leaves}(T)} \sum_{i \in c} (y_i - m_c)^2$$

where $m_c = \frac{1}{n_c}\sum_{i \in c} y_i$, the prediction for leaf c.

The basic regression-tree-growing algorithm then is as follows:

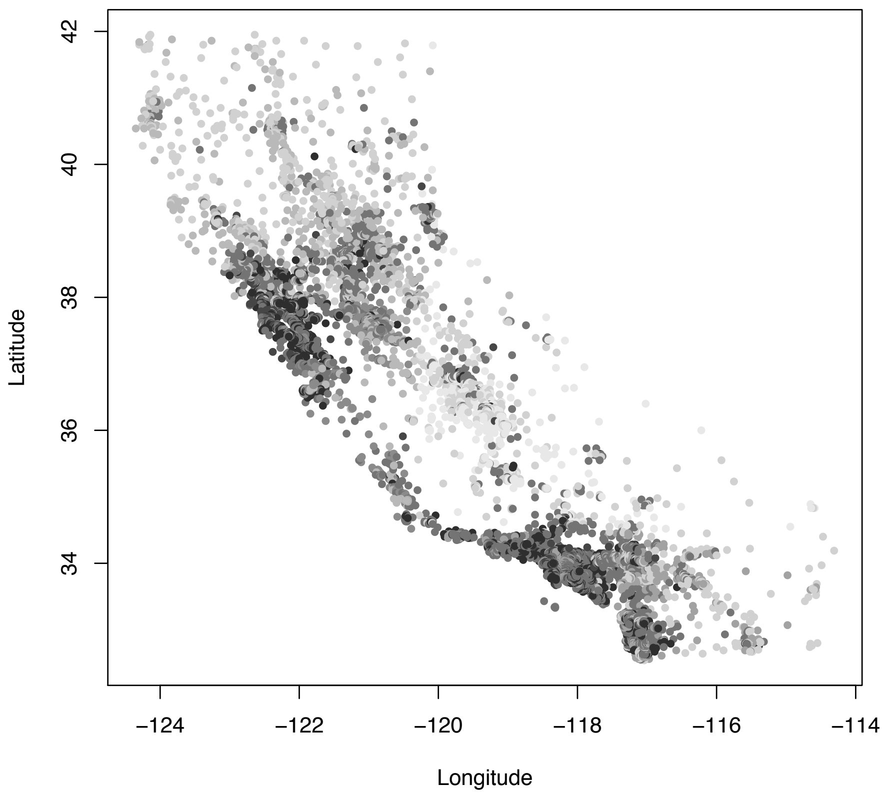
1. Start with a single node containing all points. Calculate m_c and MSE.

[2]Mixing botanical and genealogical metaphors for trees is ugly, but I can't find a way around it.



```
plot(treefit3)
text(treefit3, cex = 0.5, digits = 3)
```

FIGURE 13.6: *Regression tree for log price when all other features are included as (potential) inputs. Note that many of the features are not used by the tree.*



```
cut.predictions = cut(predict(treefit3), log(price.deciles), include.lowest = TRUE)
plot(calif$Longitude, calif$Latitude, col = grey(10:2/11)[cut.predictions],
    pch = 20, xlab = "Longitude", ylab = "Latitude")
```

FIGURE 13.7: *Predicted prices for the* `treefit3` *model. Same color scale as in previous plots (where dots indicated actual prices).*

2. If all the points in the node have the same value for all the input variables, stop. Otherwise, search over all binary splits of all variables for the one which will reduce MSE as much as possible. If the largest decrease in MSE would be less than some threshold δ, or one of the resulting nodes would contain less than q points, stop. Otherwise, take that split, creating two new nodes.

3. In each new node, go back to step 1.

Trees use only one feature (input variable) at each step. If multiple features are equally good, which one is chosen is a matter of chance, or arbitrary programming decisions.

One problem with the straight-forward algorithm I've just given is that it can stop too early, in the following sense. There can be variables which are not very informative themselves, but which lead to very informative subsequent splits. This suggests a problem with stopping when the decrease in S becomes less than some δ. Similar problems can arise from arbitrarily setting a minimum number of points q per node.

A more successful approach to finding regression trees uses the idea of cross-validation (Chapter 3), especially k-fold cross-validation. We initially grow a large tree, looking only at the error on the training data. (We might even set $q = 1$ and $\delta = 0$ to get the largest tree we can.) This tree is generally too large and will over-fit the data.

The issue is basically about the number of leaves in the tree. For a given number of leaves, there is a unique best tree. As we add more leaves, we can only lower the bias, but we also increase the variance, since we have to estimate more. At any finite sample size, then, there is a tree with a certain number of leaves which will generalize better than any other. We would like to find this optimal number of leaves.

The reason we start with a big (lush? exuberant? spreading?) tree is to make sure that we've got an upper bound on the optimal number of leaves. Thereafter, we consider simpler trees, which we obtain by **pruning** the large tree. At each pair of leaves with a common parent, we evaluate the error of the tree on the testing data, and also of the sub-tree, which removes those two leaves and puts a leaf at the common parent. We then prune that branch of the tree, and so forth until we come back to the root. Starting the pruning from different leaves may give multiple pruned trees with the same number of leaves; we'll look at which sub-tree does best on the testing set. The reason this is superior to arbitrary stopping criteria, or to rewarding parsimony as such, is that it directly checks whether the extra capacity (nodes in the tree) pays for itself by improving generalization error. If it does, great; if not, get rid of the complexity.

There are lots of other cross-validation tricks for trees. One cute one is to alternate growing and pruning. We divide the data into two parts, as before, and first grow and then prune the tree. We then exchange the role of the training and testing sets, and try to grow our pruned tree to fit the second half. We then prune again, on the first half. We keep alternating in this manner until the size of the tree doesn't change.

13.2.2.1 Cross-Validation and Pruning in R

The `tree` package contains functions `prune.tree` and `cv.tree` for pruning trees by cross-validation.

The function `prune.tree` takes a tree you fit by `tree` (see R advice for last homework), and evaluates the error of the tree and various prunings of the tree, all the way down to the stump. The evaluation can be done either on new data, if supplied, or on the training data (the default). If you ask it for a particular size of tree, it gives you the best pruning of that size[3]. If you don't ask it for the best tree, it gives an object which shows the number of leaves in the pruned trees, and the error of each one. This object can be plotted.

```
my.tree = tree(y ~ x1 + x2, data = my.data)
prune.tree(my.tree, best = 5)
prune.tree(my.tree, best = 5, newdata = test.set)
my.tree.seq = prune.tree(my.tree)
plot(my.tree.seq)
my.tree.seq$dev
opt.trees = which(my.tree.seq$dev == min(my.tree.seq$dev))
min(my.tree.seq$size[opt.trees])
```

Finally, `prune.tree` has an optional `method` argument. The default is `method="deviance"`, which fits by minimizing the mean squared error (for continuous responses) or the negative log likelihood (for discrete responses; see below).[4]

The function `cv.tree` does k-fold cross-validation (default is $k = 10$). It requires as an argument a fitted tree, and a function which will take that tree and new data. By default, this function is `prune.tree`.

```
my.tree.cv = cv.tree(my.tree)
```

The type of output of `cv.tree` is the same as the function it's called on. If I do

```
cv.tree(my.tree, best = 19)
```

I get the best tree (per cross-validation) of no more than 19 leaves. If I do

```
cv.tree(my.tree)
```

I get information about the cross-validated performance of the whole *sequence* of pruned trees, e.g., `plot(cv.tree(my.tree))`. Optional arguments to `cv.tree` can include the number of folds, and any additional arguments for the function it applies (e.g., any arguments taken by `prune`).

To illustrate, think back to `treefit2`, which predicted predicted California house prices based on geographic coordinates, but had a very large number of nodes because

[3]Or, if there is no tree with that many leaves, the smallest number of leaves $\geq$ the requested size.

[4]With discrete responses, you may get better results by saying `method="misclass"`, which looks at the misclassification rate.

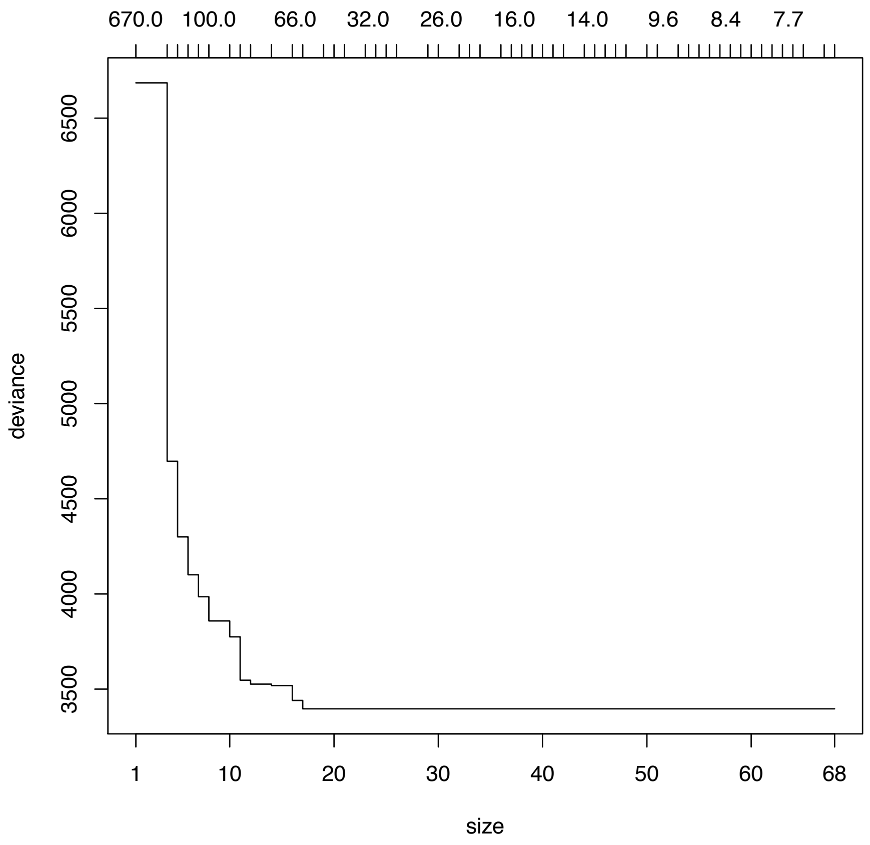
the tree-growing algorithm was told to split at the least provcation. Figure 13.8 shows the size/performance trade-off. Figures 13.9 and 13.10 show the result of pruning to the smallest size compatible with minimum cross-validated error.

13.2.3 Uncertainty in Regression Trees

Even when we are making point predictions, we have some uncertainty, because we've only seen a finite amount of data, and this is not an *entirely* representative sample of the underlying probability distribution. With a regression tree, we can separate the uncertainty in our predictions into two parts. First, we have some uncertainty in what our predictions should be, *assuming* the tree is correct. Second, we may of course be wrong about the tree.

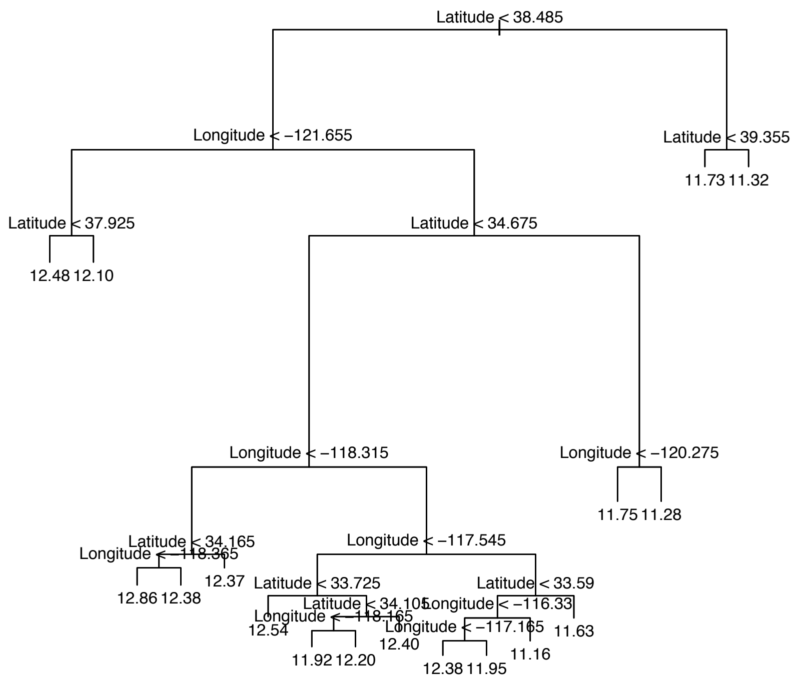
The first source of uncertainty — imprecise estimates of the conditional means within a given partition — is fairly easily dealt with. We can consistently estimate the standard error of the mean for leaf c just like we would for any other mean of IID samples. The second source is more troublesome; as the response values shift, the tree itself changes, and discontinuously so, tree shape being a discrete variable. What we want is some estimate of how different the tree could have been, had we just drawn a different sample from the same source distribution.

One way to estimate this, from the data at hand, is to use bootstrapping (ch. 6). It is important that we apply the bootstrap to the predicted values, which can change smoothly if we make a tiny perturbation to the distribution, and not to the shape of the tree itself (which can only change abruptly).



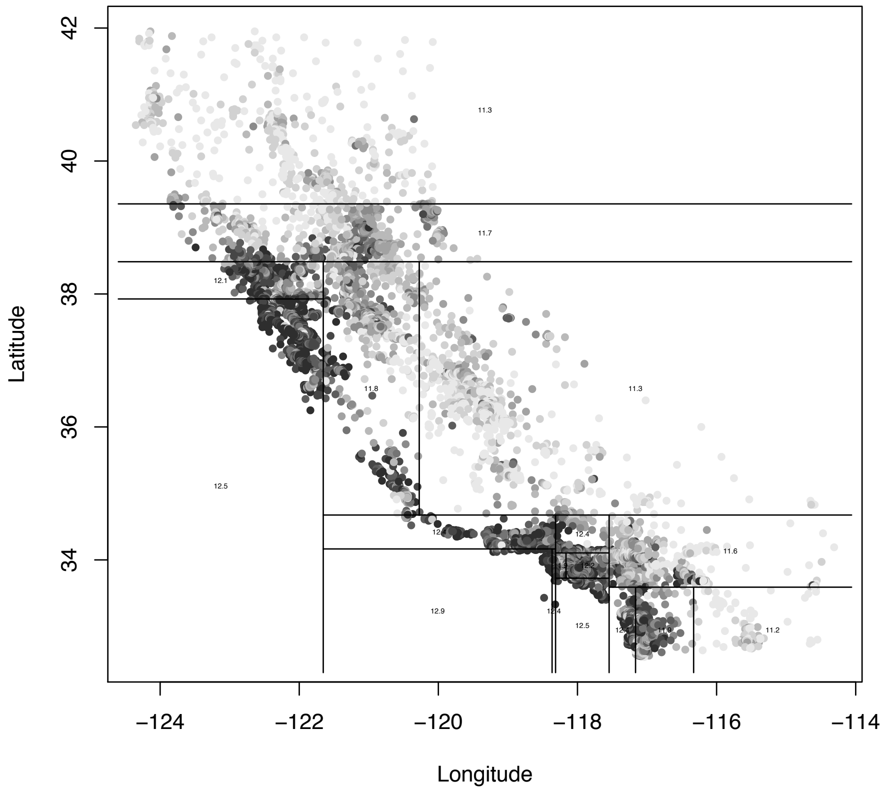
```
treefit2.cv <- cv.tree(treefit2)
plot(treefit2.cv)
```

FIGURE 13.8: *Size (horizontal axis) versus cross-validated sum of squared errors (vertical axis) for successive prunings of the* `treefit2` *model. (The upper scale on the horizontal axis refers to the "cost/complexity" penalty. The idea is that the pruning minimizes* (total error) + λ(complexity) *for a certain value of* λ*, which is what's shown on that scale. Here* complexity *is taken to just be the number of leaves in the tree, i.e., its size (though sometimes other measures of complexity are used).* λ *then acts as a Lagrange multiplier (§H.5.2) which enforces a constraint on the complexity of the tree. See Ripley (1996, §7.2, pp. 221–226) for details.*



```
opt.trees = which(treefit2.cv$dev == min(treefit2.cv$dev))
best.leaves = min(treefit2.cv$size[opt.trees])
treefit2.pruned = prune.tree(treefit2, best = best.leaves)
plot(treefit2.pruned)
text(treefit2.pruned, cex = 0.75)
```

FIGURE 13.9: `treefit2`, *after being pruned by ten-fold cross-validation.*



```
plot(calif$Longitude, calif$Latitude, col = grey(10:2/11)[cut.prices], pch = 20,
    xlab = "Longitude", ylab = "Latitude")
partition.tree(treefit2.pruned, ordvars = c("Longitude", "Latitude"), add = TRUE,
    cex = 0.3)
```

FIGURE 13.10: `treefit2.pruned`*'s partition of California. Compare to Figure 13.5.*

13.3 Classification Trees

[[TODO: Make classification task related to CA data]]

Classification trees work just like regression trees, only they try to predict a discrete category (the class), rather than a numerical value. The variables which go into the classification — the inputs — can be numerical or categorical themselves, the same way they can with a regression tree. They are useful for the same reasons regression trees are — they provide fairly comprehensible predictors in situations where there are many variables which interact in complicated, nonlinear ways.

We find classification trees in almost the same way we found regression trees: we start with a single node, and then look for the binary distinction which gives us the most information about the class. We then take each of the resulting new nodes and repeat the process there, continuing the recursion until we reach some stopping criterion. The resulting tree will often be too large (i.e., over-fit), so we prune it back using (say) cross-validation. The differences from regression-tree growing have to do with (1) how we measure information, (2) what kind of predictions the tree makes, and (3) how we measure predictive error.

13.3.1 Measuring Information

The response variable Y is categorical, so we can use information theory to measure how much we learn about it from knowing the value of another discrete variable A:

$$I[Y;A] \equiv \sum_{a} \Pr(A=a) I[Y;A=a] \qquad (13.1)$$

where

$$I[Y;A=a] \equiv H[Y] - H[Y|A=a] \qquad (13.2)$$

and you remember the definitions of entropy $H[Y]$ and conditional entropy $H[Y|A=a]$,

$$H[Y] \equiv \sum_{y} -\Pr(Y=y)\log_2 \Pr(Y=y) \qquad (13.3)$$

and

$$H[Y|A=a] \equiv sum_y - \Pr(Y=y|A=a)\log_2 \Pr(Y=y|A=a) \qquad (13.4)$$

$I[Y;A=a]$ is how much our uncertainty about Y decreases from knowing that $A=a$. (Less subjectively: how much less variable Y becomes when we go from the full population to the sub-population where $A=a$.) $I[Y;A]$ is how much our uncertainty about Y shrinks, on average, from knowing the value of A.

For classification trees, A isn't (necessarily) one of the predictors, but rather the answer to some question, generally binary, about one of the predictors X, i.e., $A = \mathbf{1}_{\mathscr{A}}(X)$ for some set $\mathscr{A}$. This doesn't change any of the math above, however. So we chose the question in the first, root node of the tree so as to maximize $I[Y;A]$, which we calculate from the formula above, using the relative frequencies in our data to get the probabilities.

When we want to get good questions at subsequent nodes, we have to take into account what we know already at each stage. Computationally, we do this by computing the probabilities and informations using only the cases in that node, rather than the complete data set. (Remember that we're doing *recursive* partitioning, so at each stage the sub-problem looks just like a smaller version of the original problem.) Mathematically, what this means is that if we reach the node when $A = a$ and $B = b$, we look for the question C which maximizes $I[Y; C|A = a, B = b]$, the information *conditional* on $A = a$, $B = b$. Algebraically,

$$I[Y; C|A = a, B = b] = H[Y|A = a, B = b] - H[Y|A = a, B = b, C] \tag{13.5}$$

Computationally, rather than looking at all the cases in our data set, we just look at the ones where $A = a$ and $B = b$, and calculate as though that were all the data. Also, notice that the first term on the right-hand side, $H[Y|A = a, B = b]$, does not depend on the next question C. So rather than maximizing $I[Y; C|A = a, B = b]$, we can just minimize $H[Y|A = a, B = b, C]$.

13.3.2 Making Predictions

There are two kinds of predictions which a classification tree can make. One is a **point** prediction, a single guess as to the class or category: to say "this is a flower" or "this is a tiger" and nothing more. The other, a **distributional prediction**, gives a *probability* for each class. This is slightly more general, because if we need to extract a point prediction from a probability forecast we can always do so, but we can't go in the other direction.

For probability forecasts, each terminal node in the tree gives us a distribution over the classes. If the terminal node corresponds to the sequence of answers $A = a$, $B = b$, $\ldots Q = q$, then ideally this would give us $\Pr(Y = y|A = a, B = b, \ldots Q = q)$ for each possible value y of the response. A simple way to get close to this is to use the empirical relative frequencies of the classes in that node. E.g., if there are 33 cases at a certain leaf, 22 of which are tigers and 11 of which are flowers, the leaf should predict "tiger with probability 2/3, flower with probability 1/3". This is the **maximum likelihood** estimate of the true probability distribution, and we'll write it $\widehat{\Pr}(\cdot)$.

Incidentally, while the empirical relative frequencies are consistent estimates of the true probabilities under many circumstances, nothing particularly *compells* us to use them. When the number of classes is large relative to the sample size, we may easily fail to see any samples at all of a particular class. The empirical relative frequency of that class is then zero. This is good if the actual probability is zero, not so good otherwise. (In fact, under the negative log-likelihood error discussed below, it's infinitely bad, because we will eventually see that class, but our model will say it's impossible.) The empirical relative frequency estimator is in a sense too reckless in following the data, without allowing for the possibility that it the data are wrong; it may under-smooth. Other probability estimators "shrink away" or "back off" from the empirical relative frequencies; Exercise 1 involves one such estimator.

For point forecasts, the best strategy depends on the loss function. If it is just the mis-classification rate, then the best prediction at each leaf is the class with the highest conditional probability in that leaf. With other loss functions, we should make the guess which minimizes the expected loss. But this leads us to the topic of measuring error.

13.3.3 Measuring Error

There are three common ways of measuring error for classification trees, or indeed other classification algorithms: misclassification rate, expected loss, and normalized negative log-likelihood, a.k.a. **cross-entropy**.

13.3.3.1 Misclassification Rate

We've already seen this: it's the fraction of cases assigned to the wrong class.

13.3.3.2 Average Loss

The idea of the average loss is that some errors are more costly than others. For example, we might try classifying cells into "cancerous" or "not cancerous" based on their gene expression profiles[5]. If we think a healthy cell from someone's biopsy is cancerous, we refer them for further tests, which are frightening and unpleasant, but not, as the saying goes, the end of the world. If we think a cancer cell is healthy, th consequences are much more serious! There will be a different cost for each combination of the real class and the guessed class; write L_{ij} for the cost ("loss") we incur by saying that the class is j when it's really i.

For an observation x, the classifier gives class probabilities $\Pr(Y = i|X = x)$. Then the expected cost of predicting j is:

$$\text{Loss}(Y = j|X = x) = \sum_i L_{ij} \Pr(Y = i|X = x)$$

A cost matrix might look as follows

	prediction	
truth	"cancer"	"healthy"
"cancer"	0	100
"healthy"	1	0

We run an observation through the tree and wind up with class probabilities $(0.4, 0.6)$. The most likely class is "healthy", but it is not the most cost-effective decision. The expected cost of predicting "cancer" is $0.4 * 0 + 0.6 * 1 = 0.6$, while the expected cost of predicting "healthy" is $0.4 * 100 + 0.6 * 0 = 40$. The probability of $Y =$ "healthy" must be 100 times higher than that of $Y =$ "cancer" before "cancer" is a cost-effective prediction.

Notice that if our estimate of the class probabilities is very bad, we can go through the math above correctly, but still come out with the wrong answer. If our estimates were exact, however, we'd always be doing as well as we could, given the data.

You can show (and will, in the homework!) that if the costs are symmetric, we get the mis-classification rate back as our error function, and should always predict the most likely class.

[5]Think back to Homework 4, only there *all* the cells were cancerous, and the question was just "which cancer?"

13.3.3.3 Likelihood and Cross-Entropy

The normalized negative log-likelihood is a way of looking not just at whether the model made the wrong call, but whether it made the wrong call with confidence or tentatively. ("Often wrong, never in doubt" is *not* a good way to go through life.) More precisely, this loss function for a model Q is

$$L(\text{data}, Q) = -\frac{1}{n}\sum_{i=1}^{n} \log Q(Y = y_i | X = x_i)$$

where $Q(Y = y|X = x)$ is the conditional probability the model predicts. If perfect classification were possible, i.e., if Y were a function of X, then the best classifier would give the actual value of Y a probability of 1, and $L = 0$. If there is some irreducible uncertainty in the classification, then the best possible classifier would give $L = H[Y|X]$, the conditional entropy of Y given the inputs X. Less-than-ideal predictors have $L > H[Y|X]$. To see this, try re-write L so we sum over values rather than data-points:

$$\begin{aligned} L &= -\frac{1}{n}\sum_{x,y} N(Y = y, X = x) \log Q(Y = y|X = x) \\ &= -\sum_{x,y} \widehat{\Pr}(Y = y, X = x) \log Q(Y = y|X = x) \\ &= -\sum_{x,y} \widehat{\Pr}(X = x)\widehat{\Pr}(Y = y|X = x) \log Q(Y = y|X = x) \\ &= -\sum_{x} \widehat{\Pr}(X = x)\sum_{y} \widehat{\Pr}(Y = y|X = x) \log Q(Y = y|X = x) \end{aligned}$$

If the quantity in the log was $\Pr(Y = y|X = x)$, this would be $H[Y|X]$. Since it's the model's estimated probability, rather than the real probability, it turns out that this is always *larger* than the conditional entropy. L is also called the **cross-entropy** for this reason.

There is a slightly subtle issue here about the difference between the in-sample loss, and the expected generalization error or risk. $N(Y = y, X = x)/n = \widehat{\Pr}(Y = y, X = x)$, the empirical relative frequency or empirical probability. The law of large numbers says that this converges to the true probability, $N(Y = y, X = x)/n \rightarrow \Pr(Y = y, X = x)$ as $n \rightarrow \infty$. Consequently, the model which minimizes the cross-entropy in sample may not be the one which minimizes it on future data, though the two ought to converge. Generally, the in-sample cross-entropy is lower than its expected value.

Notice that to compare two models, or the same model on two different data sets, etc., we do not need to know the true conditional entropy $H[Y|X]$. All we need to know is that L is smaller the closer we get to the true class probabilities. If we could get L down to the cross-entropy, we would be exactly reproducing all the class probabilities, and then we could use our model to minimize any loss function we liked (as we saw above).[6]

[6]Technically, if our model gets the class probabilities right, then the model's predictions are just as

13.3.3.4 Neyman-Pearson Approach

Using a loss function which assigns different weights to different error types has two noticeable drawbacks. First of all, we have to *pick* the weights, and this is often quite hard to do. Second, whether our classifier will do well in the future depends on getting the same *proportion* of cases in the future. Suppose that we're developing a tree to classify cells as cancerous or not from their gene expression profiles. We will probably want to include lots of cancer cells in our training data, so that we can get a good idea of what cancers look like, biochemically. But, fortunately, most cells are *not* cancerous, so if doctors start applying our test to their patients, they're going to find that it massively over-diagnoses cancer — it's been calibrated to a sample where the proportion (cancer):(healthy) is, say, 1:1, rather than, say, 1:20.[7]

There is an alternative to weighting which deals with both of these issues, and deserves to be better known and more widely-used than it is. This was introduced by Scott and Nowak (2005), under the name of the "Neyman-Pearson approach" to statistical learning. The reasoning goes as follows.

When we do a binary classification problem, we're really doing a hypothesis test, and the central issue in hypothesis testing, as first recognized by Neyman and Pearson, is to distinguish between the rates of different *kinds* of errors: false positives and false negatives, false alarms and misses, type I and type II. The Neyman-Pearson approach to designing a hypothesis test is to first fix a limit on the false positive probability, the **size** of the test, canonically α. Then, among all tests of size α, we want to minimize the false negative rate, or equivalently maximize the power, β.

In the traditional theory of testing, we know the distribution of the data under the null and alternative hypotheses, and so can (in principle) calculate α and β for any given test. This is *not* the case in data mining, but we do generally have very large samples generated under both distributions (depending on the class of the data point). If we fix α, we can ask, for any classifier — say, a tree — whether its false alarm rate is $\leq \alpha$. If so, we keep it for further consideration; if not, we discard it. Among those with acceptable false alarm rates, then, we ask "which classifier has the lowest false negative rate, the highest β?" This is the one we select.

Notice that this solves both problems with weighting. We don't have to pick a weight for the two errors; we just have to say what *rate* of false positives α we're willing to accept. There are many situations where this will be easier to do than to fix on a relative cost. Second, the rates α and β are properties of the *conditional* distributions of the features, $\Pr(X|Y)$. If those conditional distributions stay they same but the proportions of the classes change, then the error rates are unaffected. Thus, training the classifier with a different mix of cases than we'll encounter in the future is not an issue.

informative as the original data. We then say that the predictions are a **sufficient statistic** for forecasting the class. In fact, if the model gets the exact probabilities wrong, but has the correct partition of the feature space, then its prediction is still a sufficient statistic. Under any loss function, the optimal strategy can be implemented using *only* a sufficient statistic, rather than needing the full, original data. This is an interesting but much more advanced topic; see, e.g., Blackwell and Girshick (1954) for details.

[7] Cancer is rarer than that, but realistically doctors aren't going to run a test like this unless they have some reason to suspect cancer might be present.

Unfortunately, I don't know of any R implementation of Neyman-Pearson learning; it wouldn't be hard, I think, but goes beyond one problem set at this level.

13.4 Further Reading

The classic book on prediction trees, which basically introduced them into statistics and data mining, is Breiman *et al.* (1984). Chapter three in Berk (2008) is clear, easy to follow, and draws heavily on Breiman et al. Another very good chapter is the one on trees in Ripley (1996), which is especially useful for us because Ripley wrote the `tree` package. (The whole book is strongly recommended.) There is another tradition of trying to learn tree-structured models which comes out of artificial intelligence and inductive logic; see Mitchell (1997).

The clearest explanation of the Neyman-Pearson approach to hypothesis testing I have ever read is that in Reid (1982), which is one of the books which made me decide to learn statistics.

13.5 Exercises

1. Repeat the analysis of the California house-price data with the Pennsylvania data.

2. Explain why, for a *fixed* partition, a regression tree is a linear smoother.

3. Suppose that we see each of k classes n_i times, with $\sum_{i=1}^{k} n_i = n$. The maximum likelihood estimate of the probability of the i^{th} class is $\widehat{p}_i = n_i/n$. Suppose that instead we use the estimates

$$\tilde{p}_i = \frac{n_i + 1}{\sum_{j=1}^{k} n_j + 1} \qquad (13.6)$$

   This estimator goes back to Laplace, who called it the "rule of succession".

   (a) Show that $\sum_i^k \tilde{p}_i = 1$, no matter what the sample is.

   (b) Show that if $\widehat{p} \rightarrow p$ as $n \rightarrow \infty$, then $\tilde{p} \rightarrow p$ as well.

   (c) Using the result of the previous part, show that if we observe an IID sample, that $\tilde{p} \rightarrow p$, i.e., that $\tilde{p}$ is a consistent estimator of the true distribution.

   (d) Does $\tilde{p} \rightarrow p$ imply $\widehat{p} \rightarrow p$?

   (e) Which of these properties still hold if the +1s in the numerator and denominator are replaced by $+d$ for an arbitrary $d > 0$?

4. *Fun with Laplace's rule of succession: will the Sun rise tomorrow?* One illustration Laplace gave of this probability estimator was the following. Suppose we

know, from written records, that the Sun has risen in the east every day for the last 4000 years.[8]

(a) Calculate the probability of the event "the Sun will rise in the east tomorrow", using Eq. 13.6. You may take the year as containing 365.256 days.

(b) Calculate the probability that the Sun will rise in the east *every day for the next four thousand years*, assuming this is an IID event. Is this a reasonable assumption?

(c) Calculate the probability of the event "the Sun will rise in the east every day for four thousand years" directly from Eq. 13.6. Does your answer agree with part (b)? Should it?

Laplace did not, of course, base his belief that the Sun will rise in the morning on such calculations; besides everything else, he was the world's expert in celestial mechanics! But this shows the problem with the

5. Show that, when all the off-diagonal elements of L_{ij} (from §13.3.3.2) are equal (and positive!), the best class to predict is always the most probable class.

[8]Laplace was thus ignoring people who live above the Artic circle, or below the Antarctic circle. The latter seems particularly unfair, because so many of them are scientists.