

PML_3-1 Predicting with Trees

Key Ideas:

- Iteratively split variables into groups
- Evaluate "homogeneity" within each group
- Split again if necessary

Pros:

- Easy to interpret
- Better performance in nonlinear settings

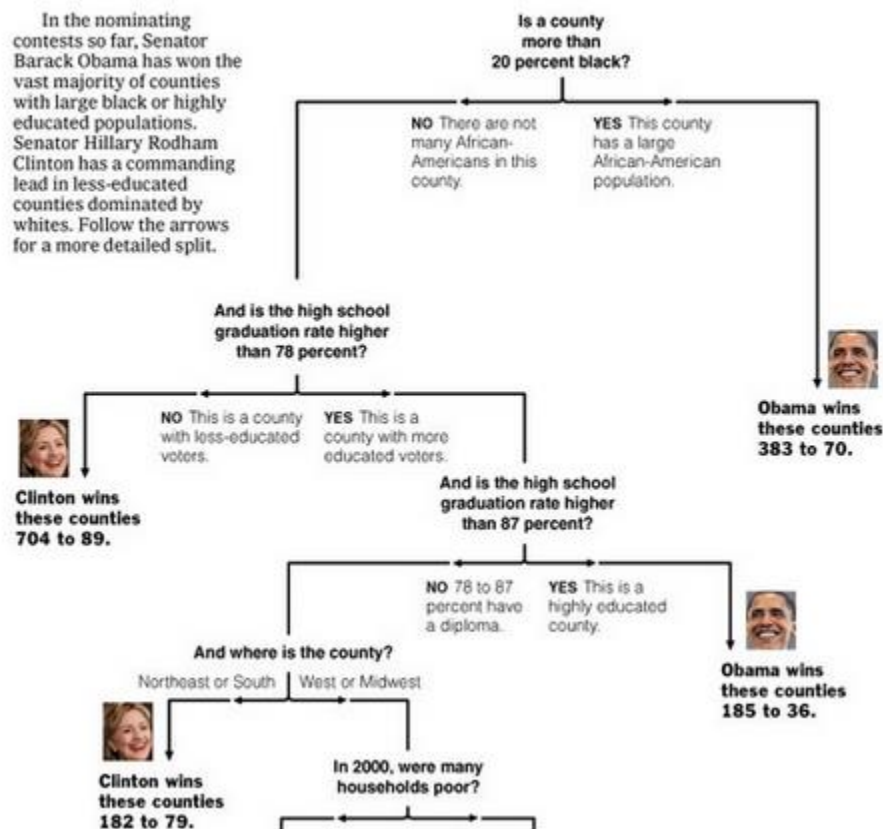
Cons:

- Without pruning/cross-validation can lead to overfitting
- Harder to estimate uncertainty
- Results may be variable

Example Tree:

<http://graphics8.nytimes.com/images/2008/04/16/us/0416-nat-subOBAMA.jpg>

Decision Tree: The Obama-Clinton Divide



Basic algorithm

1. Start with all variables in one group
2. Find the variable/split that best separates the outcomes
3. Divide the data into two groups ("leaves") on that split ("node")
4. Within each split, find the best variable/split that separates the outcomes
5. Continue until the groups are too small or sufficiently "pure"

Measures of Impurity

http://en.wikipedia.org/wiki/Decision_tree_learning

probability: number of times k occurs in leaf m

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \text{ in Leaf } m} 1(y_i = k)$$

Misclassification Error:

$$1 - \hat{p}_{mk(m)}; k(m) = \text{most; common; } k$$

- 0 = perfect purity
- 0.5 = no purity

Gini index:

$$\sum_{k \neq k'} \hat{p}_{mk} \times \hat{p}_{mk'} = \sum_{k=1}^K \hat{p}_{mk} (1 - \hat{p}_{mk}) = 1 - \sum_{k=1}^K \hat{p}_{mk}^2$$

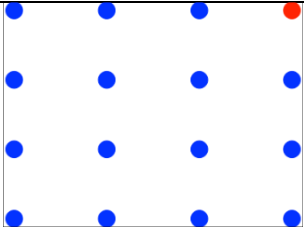
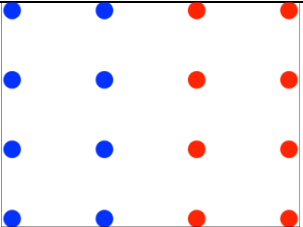
- 0 = perfect purity
- 0.5 = no purity

Deviance/information gain:

$$-\sum_{k=1}^K \hat{p}_{mk} \log_2 \hat{p}_{mk}$$

probability: $\log_e \rightarrow$ deviance; $\log_2 \rightarrow$ information

- 0 = perfect purity
- 1 = no purity

	
<p>Misclassification: $1/16=0.06$ Gini: $1 - [(1/16)2 + (15/16)2] = 0.12$ Information: $-[1/16 \times \log_2(1/16) + 15/16 \times \log_2(15/16)] = 0.34$</p>	<p>Misclassification: $8/16=0.5$ Gini: $1 - [(8/16)2 + (8/16)2] = 0.5$ Information: $-[1/16 \times \log_2(1/16) + 15/16 \times \log_2(15/16)] = 1$</p>

Example: Iris Data

```
data(iris); library(ggplot2)
names(iris)
```

```
[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"
```

```
table(iris$Species)
```

```
setosa versicolor virginica
    50         50         50
```

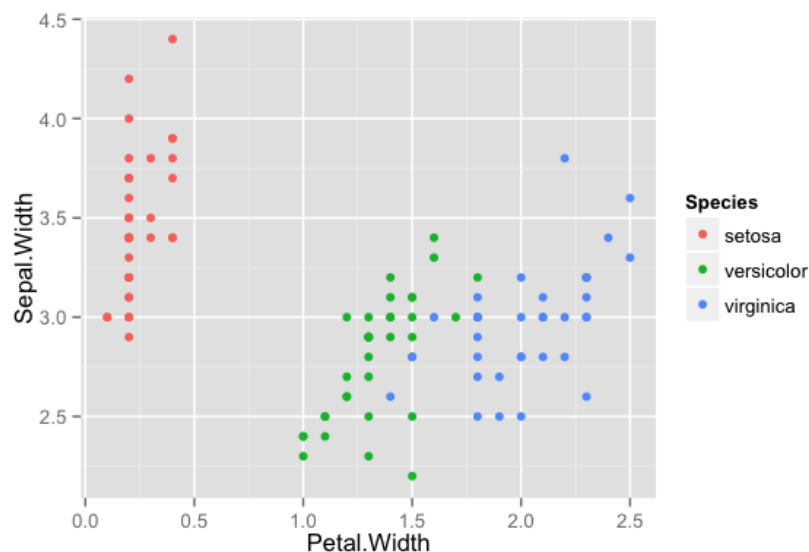
Create training and test sets

```
inTrain <- createDataPartition(y=iris$Species,
                                p=0.7, list=FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
dim(training); dim(testing)
```

```
[1] 45  5
```

Iris petal width/sepal width

```
qplot(Petal.Width, Sepal.Width, colour=Species, data=training)
```



```
# three obvious clusters
```

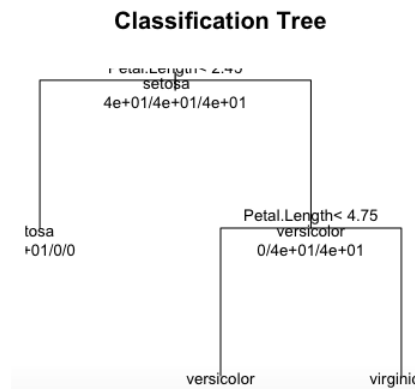
```
library(caret)
modFit <- train(Species ~ ., method="rpart", data=training)
print(modFit$finalModel)
# note "rpart" method for splitting
```

```
n= 105
```

```
node), split, n, loss, yval, (yprob)
  * denotes terminal node
```

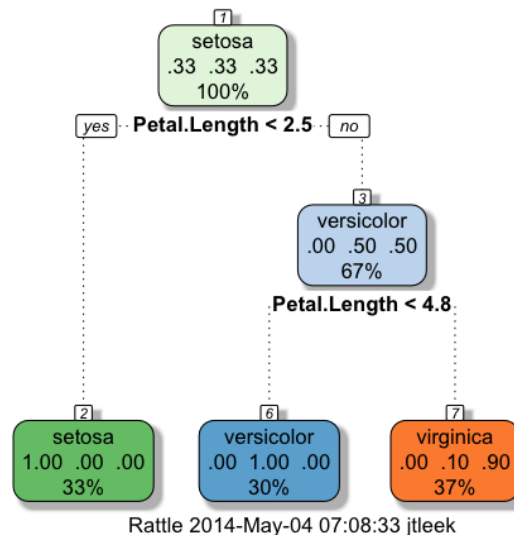
```
1) root 105 70 setosa (0.3333 0.3333 0.3333)
  2) Petal.Length< 2.45 35 0 setosa (1.0000 0.0000 0.0000) *
  3) Petal.Length>=2.45 70 35 versicolor (0.0000 0.5000 0.5000)
    6) Petal.Length< 4.75 31 0 versicolor (0.0000 1.0000 0.0000) *
    7) Petal.Length>=4.75 39 4 virginica (0.0000 0.1026 0.8974) *
```

```
plot(modFit$finalModel, uniform=TRUE,
     main="Classification Tree")
text(modFit$finalModel, use.n=TRUE, all=TRUE, cex=.8)
# first split is at Petal.Length < 2.45
```



Prettier plots

```
library(rattle) # rattle package
fancyRpartPlot(modFit$finalModel)
```



Predicting new values

```
predict(modFit,newdata=testing) # predicting class in this case
```

```
[1] setosa setosa setosa setosa setosa setosa setosa setosa
[9] setosa setosa setosa setosa setosa setosa setosa versicolor
[17] versicolor versicolor versicolor versicolor versicolor versicolor versicolor versicolor
[25] virginica versicolor virginica versicolor versicolor versicolor virginica virginica
[33] virginica versicolor virginica virginica virginica virginica virginica virginica
[41] virginica virginica virginica virginica virginica
Levels: setosa versicolor virginica
```

Notes and further resources

- Classification trees are non-linear models
 - They use interactions between variables # typically use multiple variables
 - Data transformations may be less important (monotone transformations)
 - Trees can also be used for regression problems (continuous outcome)
- Note that there are multiple tree building options in R both in the caret package - [party](#), [rpart](#), and out of the caret package - [tree](#) # tree not in caret package
- [Introduction to statistical learning](#)
- [Elements of Statistical Learning](#)
- [Classification and regression trees](#)

PML_3-2 Bagging

Bootstrap aggregating (bagging)

Basic idea:

1. Resample cases and recalculate predictions
2. Average or majority vote of recalculated predictions

Notes:

- Similar bias
- Reduced variance
- More useful for non-linear functions

Ozone data

```
library(ElemStatLearn); data(ozone,package="ElemStatLearn")
ozone <- ozone[order(ozone$ozone),] # ordered by ozone variable
head(ozone)
```

```
      ozone radiation temperature wind
17         1         8          59  9.7
19         4        25          61  9.7
14         6        78          57 18.4
45         7        48          80 14.3
106        7        49          69 10.3
7          8        19          61 20.1
```

http://en.wikipedia.org/wiki/Bootstrap_aggregating

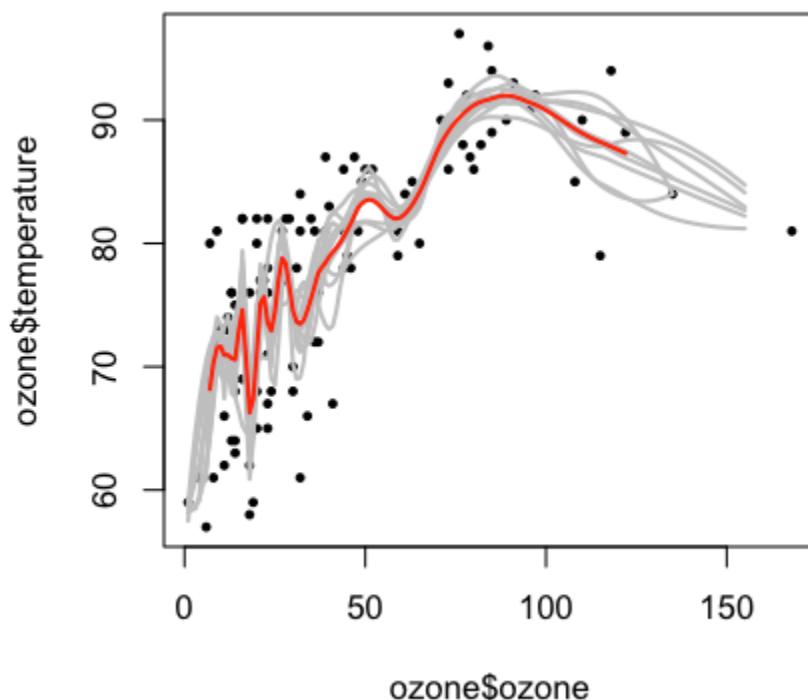
Bagged loess

```
# predict temperature from ozone
ll <- matrix(NA,nrow=10,ncol=155)
for(i in 1:10){
  ss <- sample(1:dim(ozone)[1],replace=T)
  ozone0 <- ozone[ss,]; ozone0 <- ozone0[order(ozone0$ozone),]
  loess0 <- loess(temperature ~ ozone,data=ozone0,span=0.2) # loess curve
  ll[i,] <- predict(loess0,newdata=data.frame(ozone=1:155))
}
```

Use ozone data in the ElemStatLearn package. Load the ozone data set. Order by outcome (shows how this works), the ozone variable - four variables, ozone, radiation, temperature, and wind.

Predict temperature ~ ozone. Create a matrix (10 rows and 155 columns). Resample the data set ten different times (loop over ten different samples of the data set). Sample with replacement from the entire data set. Create a new data set, ozone0, which is the resample data set for that particular element of the loop. That's just the subset of the data set corresponding to our random sample. Reorder the data set every time by the ozone variable. Fit a loess curve (a smooth curve) each time. Basic idea is fitting a smooth curve relating temperature (outcome) to the ozone variable (predictor). Each time - use the resample data set as the data set to build that predictor on. Use a common span for each time, the span being a measure of how smooth that fit will be. Predict for every single loess curve the outcome for a new data set for the exact same values. Always predict for ozone values 1 to 155. So the i^{th} row of this ll object is now the prediction from the loess curve, from the i^{th} resample of the data ozone. Result: resampled the data set ten different times, fit a smooth curve through it those ten different times, and then averaged those values.

```
plot(ozone$ozone,ozone$temperature,pch=19,cex=0.5)
for(i in 1:10){lines(1:155,ll[i,],col="grey",lwd=2)}
lines(1:155,apply(ll,2,mean),col="red",lwd=2)
```



gray lines are individual fits. Note greater variability. Red is average of gray lines and better fits the data. Bagged loess curve. There's a proof that shows that the bagging estimate will always have lower variability but similar bias to the individual model fits that you do.

Bagging in caret

- Some models perform bagging for you, in train function consider method options
 - bagEarth
 - treebag
 - bagFDA
- Alternatively you can bag any model you choose using the bag function

More bagging in caret

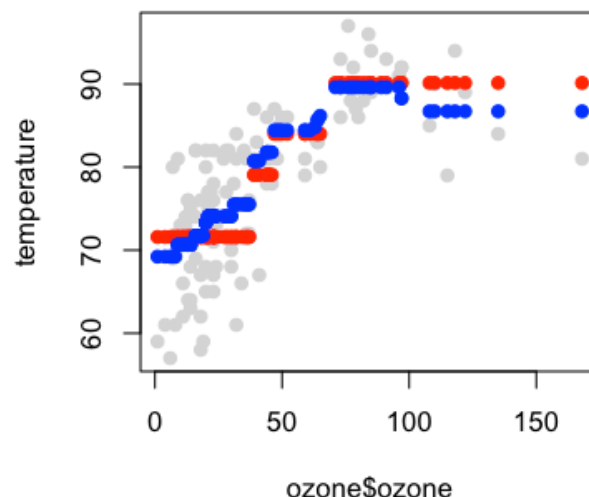
```
predictors = data.frame(ozone=ozone$ozone)
temperature = ozone$temperature
treebag <- bag(predictors, temperature, B = 10,
               bagControl = bagControl(fit = ctreeBag$fit,
                                       predict = ctreeBag$pred,
                                       aggregate = ctreeBag$aggregate))
```

Build your own bagging function in caret. Advanced use - read the documentation carefully. Idea basically to take your predictor variable and put it into one data frame - a data frame that contains the ozone data. Outcome variable – here the temperature variable from the data set. Pass to the bag function in caret package. Use the predictors from that data frame, this is my outcome, this is the number of replications with the number of sub samples I'd like to take from the data set. And then bagControl tells me about how I'm going to fit the model. fit is the function that's going to be applied to fit the model every time. This could be a call to the train function in the caret package. predict is a the way that given a particular model fit, that we'll be able to predict new values. So this could be, for example, a call to the predict function from a trained model. And then aggregate is the way that we will put the predictions together. So, for example, it could average the predictions across all the different replicated samples.

<http://www.inside-r.org/packages/cran/caret/docs/nbBag>

Example of custom bagging (continued)

```
plot(ozone$ozone, temperature, col='lightgrey', pch=19)
points(ozone$ozone, predict(treebag$fits[[1]]$fit, predictors), pch=19, col="red")
points(ozone$ozone, predict(treebag, predictors), pch=19, col="blue")
```



Plotting ozone again on the x-axis versus temperature on the y-axis. The little grey dots represent actual observed values. The red dots represent the fit from a single conditional regression tree. And so you can see that for example, it captures, it doesn't capture the trend that's going on down here very well, the red line is just flat. Even though there appears to be a trend upward in the data points here. But when I average over ten different bagged model fits with these conditional regression trees. I see that there's an increase here in the values in the blue fit, which is the fit from the bagged regression.

Parts of bagging

```
ctreeBag$fit
function (x, y, ...)
{
  library(party)
  data <- as.data.frame(x)
  data$y <- y
  ctree(y ~ ., data = data)
}
<environment: namespace:caret>
```

So we're going to look a little bit at those different parts of the bagging function. So in this particular case I'm using the `ctreeBag` function, which you can look at in, if you've loaded the `caret` package in R. So, for the `fit` part it takes the data frame that we've passed and the `predict`, and the outcome that we've passed, and it basically uses the `ctree` function to train a tree, conditional regression tree on the data set. This is the last command that's called the `ctree` command. So it returns this model fit from the `ctree` function.

```
ctreeBag$pred
function (object, x)
{
  obsLevels <- levels(object@data@get("response")[, 1])
  if (!is.null(obsLevels)) {
    rawProbs <- treeresponse(object, x)
    probMatrix <- matrix(unlist(rawProbs), ncol = length(obsLevels),
      byrow = TRUE)
    out <- data.frame(probMatrix)
    colnames(out) <- obsLevels
    rownames(out) <- NULL
  }
  else out <- unlist(treeresponse(object, x))
  out
}
<environment: namespace:caret>
```

The prediction takes in the `object`. So this is going to be an object from the `ctree` model fit. And a new data set `x`, and it's going to get a new prediction. So what you can see here is it basically calculates each time the tree response or the outcome from the object and the new data. It then calculates this probability matrix and returns either the actually observed levels

that it predicts or it actually re, just returns the response, the predicted response from the variable.

```
ctreeBag$aggregate
function (x, type = "class")
{
  if (is.matrix(x[[1]]) | is.data.frame(x[[1]])) {
    pooled <- x[[1]] & NA
    classes <- colnames(pooled)
    for (i in 1:ncol(pooled)) {
      tmp <- lapply(x, function(y, col) y[, col], col = i)
      tmp <- do.call("rbind", tmp)
      pooled[, i] <- apply(tmp, 2, median)
    }
    if (type == "class") {
      out <- factor(classes[apply(pooled, 1, which.max)],
        levels = classes)
    }
    else out <- as.data.frame(pooled)
  }
}
```

The aggregation then takes those values and averages them together or puts them together in some way. So here what this is doing is it's basically getting the prediction from every single one of these model fits, so that's across a large number of observations. And then it binds them together into one data matrix by with each row being equal to the prediction from one of the model predictions. And then it takes the median at every value. So in other words it takes the median prediction from each of the different model fits across all the bootstrap samples.

Notes and further resources

Notes:

- Bagging is most useful for nonlinear models
- Often used with trees - an extension is random forests
- Several models use bagging in caret's train function

Further resources:

- [Bagging](#)
- [Bagging and boosting](#)
- [Elements of Statistical Learning](#)

So bagging is very useful for nonlinear models, and it's widely used. It's often used with trees. And you can think of an extension to this as being random forest, which we'll talk about in a future lecture. Several models use bagging and caret's main train function, like I told you about in previous slide. And you can also build your own specific bagging functions, for any classification or prediction algorithm that you'd like to take a look at. For further resources, I've linked to a couple of different tutorials on bagging and boosting, as well as the Elements of Statistical Learning which has a lot more details about how bagging works. But remember that

the basic idea is to basically resample your data, refit your nonlinear model, then average those model fits together over resamples to get a smoother model fit, than you would've got from any individual fit on its own.

PML_3-3 Random forests

Random Forests

1. Bootstrap samples
2. At each split, bootstrap variables
3. Grow multiple trees and vote

Pros:

- Accuracy

Cons:

- Speed
- Interpretability
- Overfitting

Basically an extension to bagging for classification and regression trees. The idea is very similar to bagging in the sense that we bootstrap samples, so we take a resample of our observed data, and our training data set. And then we rebuild classification or regression trees on each of those bootstrap samples. The one difference is that at each split, when we split the data each time in a classification tree, we also bootstrap the variables. In other words, only a subset of the variables is considered at each potential split. This makes for a diverse set of potential trees that can be built. And so the idea is we grow a large number of trees. And then we either vote or average those trees in order to get the prediction for a new outcome.

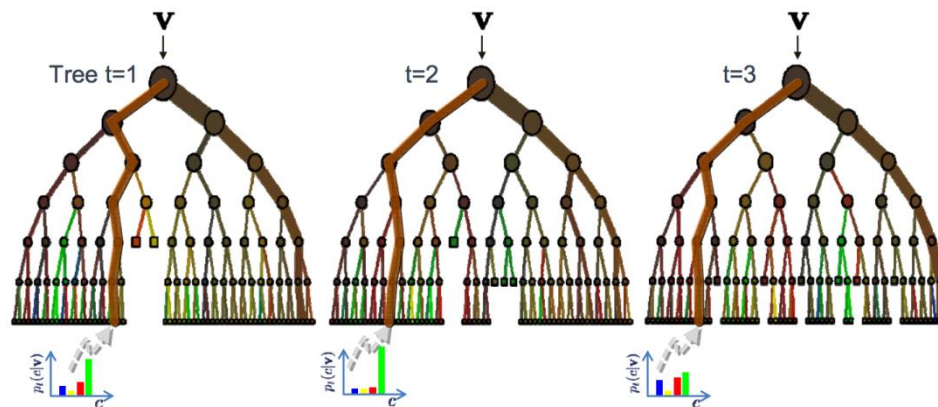
The pros for this approach are that it's quite accurate. And along with boosting, it's one of the most, widely used and highly accurate methods for prediction in competitions like Kaggle.

The cons are that it's, it can be quite slow. It has to build a large number of trees. And it can be hard to interpret, in the sense that you might have a large number of trees that are averaged together. And those trees represent bootstrap samples with bootstrap nodes that can be a little bit complicated to understand. It can also lead to a little bit of overfitting which can be complicated by the fact that it's very hard to understand which trees are leading to that overfitting, and so it's very important to use cross validation when building random forests.

Here's an example of how this works in practice. So the idea is that you build a large number of trees where each tree is based on a bootstrap sample. So, for example, this tree is built on a random subsample of your data and this is a separate random subsample of the data and this is a separate random subsample of the data. And then at each node we allow a different subset of the variables to potentially contribute to the splits. Then if we get a new observation, say this V observation here, we run that observation through tree one, and it ends up at this leaf down here at the bottom of that, tree. And so it gets a particular prediction here. Then the

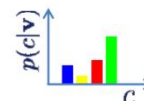
next, we take that same observation, we run it through the next tree, and it goes down a slightly different leaf, and it gets a slightly different set of predictions here. And finally we go down the third tree, and we get an even different set of predictions. Then what we do is we basically average those predictions together in order to get the predictive probabilities of each class across all the different trees.

<http://www.robots.ox.ac.uk/~az/lectures/ml/lect5.pdf>



The ensemble model

$$\text{Forest output probability } p(c|\mathbf{v}) = \frac{1}{T} \sum_t^T p_t(c|\mathbf{v})$$



An example of how this works.

Iris data

```
data(iris); library(ggplot2)
inTrain <- createDataPartition(y=iris$Species,
                                p=0.7, list=FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
```

Random forests

In the caret package, I use the train function just like I've used for the other model building.

```
library(caret)
modFit <- train(Species~ ., data=training, method="rf", prox=TRUE)
modFit
```

I send it the training data set and I tell it method equals rf, which is the random forest method. I'm also telling it to fit the outcome to be Species and to use any of the other predictive variables as potential predictors. I'm setting prox equals true because it produces a little bit of extra information I could use when I'm building these model fits. So modelFit tells me that I built the model and I've done bootstrap re-sampling and then, I tried a bunch of different tuning parameters.

```
105 samples
 4 predictors
 3 classes: 'setosa', 'versicolor', 'virginica'
```

```
No pre-processing
Resampling: Bootstrap (25 reps)
```

```
Summary of sample sizes: 105, 105, 105, 105, 105, ...
```

```
Resampling results across tuning parameters:
```

mtry	Accuracy	Kappa	Accuracy SD	Kappa SD
2	0.9	0.9	0.03	0.04
3	0.9	0.9	0.03	0.05
4	0.9	0.9	0.03	0.05

And so the tuning parameter in particular is the number of basically tries, or number of repeated trees that it's going to build. I can look at a specific tree in our final model fit using the get tree function.

Getting a single tree

```
getTree(modFit$finalModel,k=2)
```

	left daughter	right daughter	split var	split point	status	prediction
1	2	3	4	0.70	1	0
2	0	0	0	0.00	-1	1
3	4	5	4	1.70	1	0
4	6	7	3	4.95	1	0
5	8	9	3	4.85	1	0
6	0	0	0	0.00	-1	2
7	10	11	4	1.55	1	0
8	12	13	1	5.95	1	0
9	0	0	0	0.00	-1	3
10	0	0	0	0.00	-1	3
11	0	0	0	0.00	-1	2
12	0	0	0	0.00	-1	2
13	0	0	0	0.00	-1	3

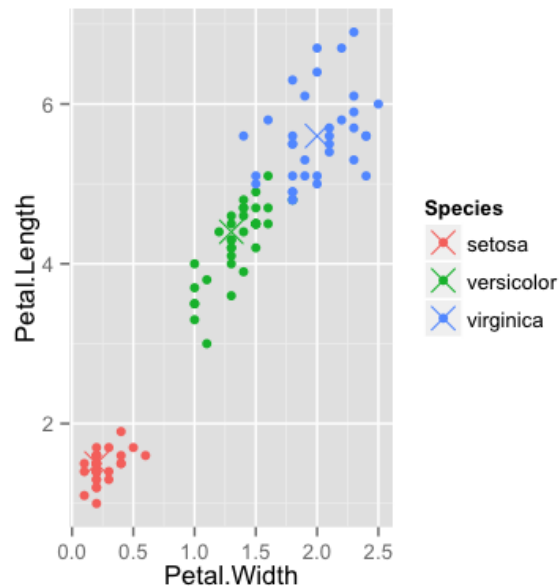
So I applied get tree here to our final model and I say I want the second tree out. And this is what the tree looks like. So each of these columns, or each of these rows corresponds to a particular split. And so, you can see what the left daughter of the tree is, the right daughter of the tree, which variable we're splitting on, what's the value where that variable is split and then what the prediction is going to be out of that particular split.

You can use this centers information as well to see what the predictions would be, or the center of the class predictions.

Class "centers"

```
irisP <- classCenter(training[,c(3,4)], training$Species,  
modFit$finalModel$prox)  
irisP <- as.data.frame(irisP); irisP$Species <- rownames(irisP)  
p <- qplot(Petal.Width, Petal.Length, col=Species, data=training)  
p + geom_point(aes(x=Petal.Width, y=Petal.Length, col=Species), size=5, shape=4, data=irisP)
```

So what I've done here is, I'm looking at two particular variables, the petal length and the petal width. So I plotted petal width on the X axis and petal length on the Y axis. I then get the class centers. So these are going to be the centers for the predicted values. So I'm going to send in the model fit, and I'm going to give it this prox variable which we asked for in the previous fitting. And when I'm going to tell it we're looking at the training data set. And so that gives us the class centers. Those class centers will then, we can then plot those to see where they fall in the data.



So now, I've created the centers data set, as well as the species data set. And what I'm going to do is plot petal width versus petal length. And I'm going to color it by species in the training data. That's what I did with this qplot command. Then I'm going to add points on top of that. That are the petal width and petal length, corresponding to the color being the species, and now I am using it from the irisP which is the centers of the data set. So what you can see is each dot here represents an observation. And the x's show the color center, or the observation centers for each of the different predictions. So you can see that we predict the, each species has a prediction for these two variables that's right in the center of the cloud of points corresponding to that particular species.

You can then predict new values using the predict functions.

Predicting new values

```
pred <- predict(modFit, testing); testing$predRight <- pred==testing$Species
```

So you pass to predict our model fit and the testing data set. And here, I'm also setting a variable, testing predict right, which is that we got the prediction right in the data set. In other

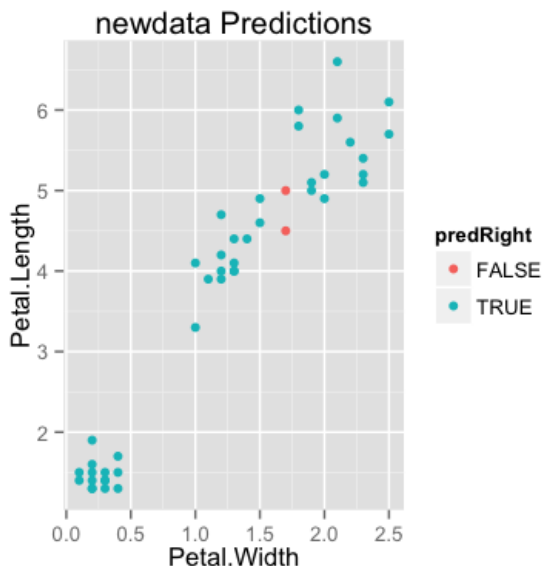
words, our prediction is equal to the testing data set species. I can then make a table of our predictions versus the species to see what that variable would look like.

```
table(pred,testing$Species)
```

pred	setosa	versicolor	virginica
setosa	15	0	0
versicolor	0	14	1
virginica	0	1	14

So I can see for example I missed two values here with my random forest model. But overall it was highly accurate in the prediction. I can then look and see which of the two that I missed.

```
qplot(Petal.Width,Petal.Length,colour=predRight,data=testing,main="newdata Predictions")
```



And perhaps unsurprisingly you can see the two that I missed, marked in red here, are the two that, in-between, two separate classes.

So remember there was one class up in this right corner, and one class right here in the middle, and this cloud, and the two points that lie right on the border we were misclassified.

So you can kind of use this to explore, and see where your prediction is doing well and where your prediction is doing poorly.

Notes and further resources

Notes:

- Random forests are usually one of the two top performing algorithms along with boosting in prediction contests.
- Random forests are difficult to interpret but often very accurate.
- Care should be taken to avoid overfitting (see `rfcv` function)

Further resources:

- [Random forests](#)
- [Random forest Wikipedia](#)
- [Elements of Statistical Learning](#)

Random forests are usually one of the top performing algorithms along with boosting in any prediction contests. They're often difficult to interpret because of these multiple trees that

we're fitting but they can be very accurate for a wide range of problems. You can check out the `rfcv` function to make sure that cross validation is being performed, but the `train` function in `caret` also handles that for you. For more information you can read about random forests directly from the inventor here. The Wikipedia page for random forests is also quite good, and the elements of statistical learning covers it as well.

PML_3-4 Boosting

Basic idea

- Take lots of (possibly) weak predictors
- Weight them and add them up
- Get a stronger predictor

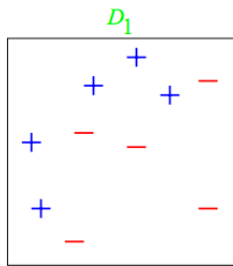
Boosting, along with random forest, is one of the most accurate out of the box classifiers that you can use. The basic idea here is, take a large number of possibly weak predictors, and we're going to take those possibly weak predictors, and weight them in a way, that takes advantage of their strengths, and add them up. When we weight them and add them up, we're sort of doing the same kind of idea that we did with bagging for regression trees. Or that we did with random forest, where we're talking a large number of classifiers and sort of averaging them. And then, by averaging them together, we get a stronger predictor.

Basic idea behind boosting

1. Start with a set of classifiers h_1, \dots, h_k
 - Examples: All possible trees, all possible regression models, all possible cutoffs.
2. Create a classifier that combines classification functions:
$$f(x) = \text{sgn}(\sum_{t=1}^T \alpha_t h_t(x)).$$
 - Goal is to minimize error (on training set)
 - Iterative, select one h at each step
 - Calculate weights based on errors
 - Upweight missed classifications and select next h

So the basic idea here is to take k classifiers. These come from, usually, from the same kind of class of classifiers. And so, some ideas might be using all possible classification trees, or all possible regression models, or all possible cutoffs, where you just divide the data into different points. You then create a classifier that combines these classification functions together, and weights them together. So, α_t here is a weight times the classifier $h_t(x)$, and so this weighted set of classifiers, gives you a prediction for the new point, that's our $f(x)$. The goal here is to minimize error on the training set. And so this is iterative at each step we take, exactly one h . We calculate weights for the next step, based on the errors that we get from that original h . And then we upweight missed classifications, and select the next stage and move on. The most famous boosting algorithm is probably [Adaboost](http://webee.technion.ac.il/people/rmeir/BoostingTutorial.pdf). A very nice tutorial on boosting: <http://webee.technion.ac.il/people/rmeir/BoostingTutorial.pdf>

Simple example

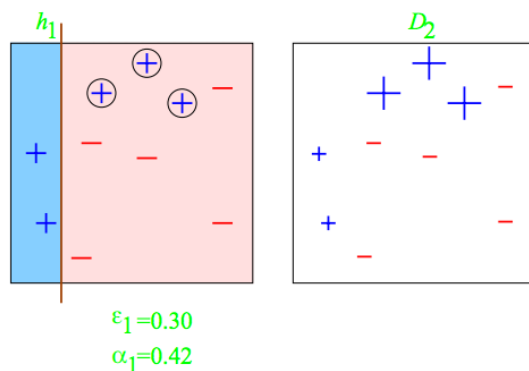


Suppose we're trying to separate the blue plus signs, from the red minus signs, and we have two variables to predict with. So here I plotted variable one on the x-axis, and variable two on the y-axis. We haven't named them because this is just a very simple example. And so the idea is, can we build a classifier that separates these variables from each other?

<http://webee.technion.ac.il/people/rmeir/BoostingTutorial.pdf>

Round 1: adaboost

Round 1



We could start off with a really simple classifier. We could say just draw a line a vertical line, and say, which vertical line separates these points well? Here's a, a classifier that says anything to the left of this vertical line is a blue plus, and anything to the right is a red minus, and so you can see we've misclassified these three points up here in the top right. So the thing that we would do is build that classifier, calculate the error rate, in this case we're missing about 30% of the points. And then we would

upweight those points that we missed. Here I've shown that upweighting, by drawing them in a larger scale. So those pluses are now upweighted, for building the next classifier. We would then build the next classifier.

Round 2 & 3

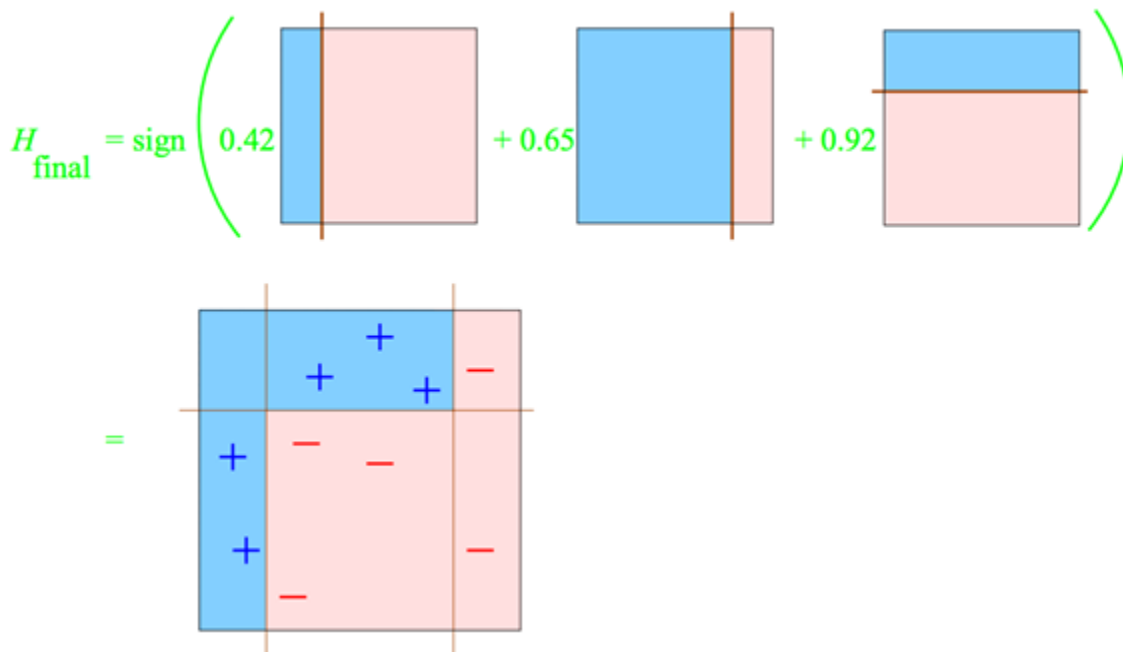
Round 2	Round 3
<p>$J_2 = 0.21, \epsilon_2 = 0.65$</p>	<p>$J_3 = 0.14, \epsilon_3 = 0.92$</p>

And in this case, our second classifier would be, one that drew a vertical line h_2 . And so, that vertical line would classify everything to the right of that line as a red minus, and everything to the left, as a blue plus. And so here we again misclassified three points (circled red minus). And

so, those three points are now upweighted, and they are also drawn larger for the next iteration. So we can again calculate the error rate, and use that to calculate the weights for the next step. Then the third classifier, will intentionally try to classify those points that we misclassified in the last couple of rounds. So, for example, these pluses, and these minuses need to be correctly classified. To do that we now draw a horizontal line h_3 , and we say anything below that horizontal line is a red minus, anything above is a blue plus, and now we misclassify this point, and these two points (circled blue plus).

Completed classifier

Final Hypothesis



So then what we can do, is we can take that, those classifiers, and we can weight them, and add them up, and so what we do is we say, we're going to classify a weighted combination of 0.42 times the first vertical line; plus 0.65 times the second vertical line, plus 0.92 times the classification given by this horizontal line. So, what that ends up doing is, once you add these classification rules up together, you can see that our classifier works much better now. We get, to a much better classification when adding them up, that gets all of the blue pluses, and all of the red minuses together. So in each case, each of our classifiers was quite simple, was just a line across the plane, and so, it's actually quite a naive classifier. But when you weight them and add them up, you can get quite a strong classifier at the end of the day.

Boosting can be done with any subset of classifiers. In other words, you can start with any sort of weak set of classifiers. In the previous example, we just used straight lines to separate the plane. A very large class is what's called gradient boosting.

Boosting in R

- Boosting can be used with any subset of classifiers
- One large subclass is [gradient boosting](#)
- R has multiple boosting libraries. Differences include the choice of basic classification functions and combination rules.
 - [gbm](#) - boosting with trees.
 - [mboost](#) - model based boosting
 - [ada](#) - statistical boosting based on [additive logistic regression](#)
 - [gamBoost](#) for boosting generalized additive models
- Most of these are available in the caret package

And you can read more about that here. R has multiple boosting libraries. They basically depend on the different kind of classification functions, and combination rules. [Gbm](#) package does boosting with trees. [Mboost](#) does model based boost, boosting. [Ada](#) does additive logistic regression boosting. And [gamBoost](#) does boosting for generalized additive models. Most of these are available in the caret package, so you can use them directly by using the `train` function with caret.

Wage example

```
library(ISLR); data(Wage); library(ggplot2); library(caret);  
Wage <- subset(Wage, select=-c(logwage))  
inTrain <- createDataPartition(y=Wage$wage,  
                                p=0.7, list=FALSE)  
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
```

So here we're going to use the wage example, to illustrate how you can apply a boosting algorithm. So here we're going to load the ISLR library, and the wage data. The ggplot2 library, and the caret library, and then we're going to, to create a wage data set that removes the variable that we're trying to predict, the log wage variable. And we create a training set and a testing set.

Fit the model

```
modFit <- train(wage ~ ., method="gbm", data=training, verbose=FALSE)
```

We can then model the wage variable, as a function of all the remaining variables. That's why there's this dot here. And we can use gbm, which does boosting with trees, and we use `verbose=FALSE` here, because this produces a lot of output when `method="gbm"` if you don't set `verbose=FALSE`. `method` equals gbm. So when we print the model fit, you can see that there's a different numbers of trees that are used, and different interaction depths, and they're basically used together, to build a boosted version of regression trees.

```
print(modFit)
```

```
2102 samples
 10 predictors
```

```
No pre-processing
```

```
Resampling: Bootstrap (25 reps)
```

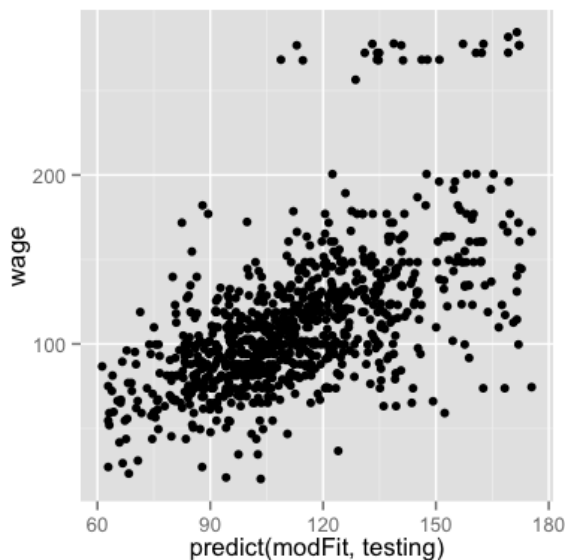
```
Summary of sample sizes: 2102, 2102, 2102, 2102, 2102, 2102, ...
```

```
Resampling results across tuning parameters:
```

interaction.depth	n.trees	RMSE	Rsquared	RMSE SD	Rsquared SD
1	50	30	0.3	1	0.02
1	100	30	0.3	1	0.02
1	200	30	0.3	1	0.02
2	50	30	0.3	1	0.02
2	100	30	0.3	1	0.02

Plot the results

```
qplot(predict(modFit,testing),wage,data=testing)
```



So, here I'm plotting the predicted results from the testing sets. So this is the using R model fit (modFit). We're predicting on the test set, versus the wage variable in the test set. And we get a real, a reasonably good prediction, although there still seems to be a lot of variability there.

But the basic idea for fitting a boosting tree a boosted algorithm in general, is to be, we take these weak classifiers, and average them together with weights, in order to get a better classifier.

Notes and further reading

- A couple of nice tutorials for boosting
 - Ron Meir- <http://webee.technion.ac.il/people/rmeir/BoostingTutorial.pdf>
 - Freund and Shapire - <http://www.cc.gatech.edu/~thad/6601-gradAI-fall2013/boosting.pdf>
- Boosting, random forests, and model ensembling are the most common tools that win Kaggle and other prediction contests.
 - http://www.netflixprize.com/assets/GrandPrize2009_BPC_BigChaos.pdf

- <https://kaggle2.blob.core.windows.net/wiki-files/327/09ccf652-8c1c-4a3d-b979-ce2369c985e4/Willem%20Mestrom%20-%20Milestone%201%20Description%20V2%202.pdf>

A lot of the information that I have given in this lecture, can be found in the Ron Meir tutorial, which also has even more in-depth information if you are interested. Here's a little bit more technical introduction to boosting from Freund and Shapire.

And then, here are several more write-ups about boosting and random forests. These are actually write-ups from different prizes that have been won, using a combination of random forests and boosting blended together, in order to achieve maximal prediction accuracy.

PML_3-5 Model Based Prediction

Basic idea

1. Assume the data follow a probabilistic model
2. Use Bayes' theorem to identify optimal classifiers

Pros:

- Can take advantage of structure of the data
- May be computationally convenient
- Are reasonably accurate on real problems

Cons:

- Make additional assumptions about the data
- When the model is incorrect you may get reduced accuracy

The basic idea here is that we're going to assume the data follow a specific probabilistic model. Then we're going to use Bayes' theorem to identify optimal classifiers based on that probabilistic model. The advantage is that this approach can take advantage of some structure that might appear in the data; for example, they follow a specific distribution and that may lead to some computational conveniences. There may also be reasonably accurate on real problems, particularly the real problems that appear to follow the data distribution that underlies our probabilistic model. The cons are that they do make these additional assumptions about the data, and they don't have to be exactly satisfied in order for the prediction algorithms to work very well. But if they're very far off, the algorithms may fail. And when the model is incorrect, you do get reduced accuracy.

Model based approach

1. Our goal is to build parametric model for conditional distribution $P(Y = k|X = x)$

2. A typical approach is to apply [Bayes theorem](#):

$$Pr(Y = k|X = x) = \frac{Pr(X = x|Y = k)Pr(Y = k)}{\sum_{\ell=1}^K Pr(X = x|Y = \ell)Pr(Y = \ell)}$$

$$Pr(Y = k|X = x) = \frac{f_k(x)\pi_k}{\sum_{\ell=1}^K f_{\ell}(x)\pi_{\ell}}$$

3. Typically prior probabilities π_k are set in advance.

4. A common choice for $f_k(x) = \frac{1}{\sigma_k \sqrt{2\pi}} e^{-\frac{(x-\mu_k)^2}{\sigma_k^2}}$ a Gaussian distribution

5. Estimate the parameters (μ_k, σ_k^2) from the data.

6. Classify to the class with the highest value of $P(Y = k|X = x)$

Here's the model based approach. Our goal is to build a parametric model, or a model based on probability distributions. For the conditional distribution the probability that y our outcome equals some specific class k given a particular set of predictor variables so that our x variables equal the little value of x, $P(Y = k|X = x)$. A typical approach is to apply a Bayes theorem. In other words we want to know something about the probability y equals k, that y comes from class k, given the variables that we've observed, $P(Y = k|X = x)$. And we write that down using Bay's theorem as $P(X = x|Y = k)$ times $P(Y = k)$ divided by the law of total probability here below.

We then assume some parametric model for the distribution of the features given the class, so that's the component $f_k(x)$ and we assume a prior that each particular element comes from a specific class denoted by π_k . So then we can basically model the distribution, the probability that y equals k given a particular set of predictor variables $P(Y = k|X = x)$ as, the fraction where we have a model for the x variables $f_k(x)$ and a model for the prior probability, π_k . The prior probabilities, π_k , are usually set in advance from the data. And then a common choice for $f_k(x)$ is a Gaussian distribution. It may be a multivariate Gaussian distribution if there are multiple x variables. And then we might estimate the parameters, (μ_k, σ_k^2) , from the data. Then once we have these parameters estimated, we can calculate the probability y belongs to any given class, as soon as we observe the predictor variables. And we classify the variables to the class having the highest probability in this sense.

Classifying using the model

A range of models use this approach <http://statweb.stanford.edu/~tibs/ElemStatLearn/>

- Linear discriminant analysis assumes $f_k(x)$ is multivariate Gaussian with same covariances
- Quadratic discriminant analysis assumes $f_k(x)$ is multivariate Gaussian with different covariances

- [Model based prediction](#) assumes more complicated versions for the covariance matrix
- Naive Bayes assumes independence between features for model building

So, a range of models use this approach. The most popular of which is probably **linear discriminant analysis**, which assumes that $f_k(x)$ is a multivariate Gaussian Distribution, so the features have a multivariate Gaussian Distribution within each class and there is the same covariance matrix for every class. This ends up drawing basically lines through the data, the covariate space. And so that's why it's called linear discriminant analysis, we'll see that in a minute.

Quadratic discriminant analysis is very much like linear discriminant analysis, although it allows different covariance matrices within each of the different classes. And so that ends up drawing quadratic curves through the data, as opposed to lines.

Model based prediction basically allows for more complicated versions of the covariance matrix when building these models. And **naive Bayes classifiers**, which we'll talk about a little bit more in this lecture, basically assume independence between the features. So, in other words in, assume independence between our predictor variables through the model building process. This may not be true. We may not believe the features are or the predictors are independent, but it still might be a useful model for prediction purposes.

Why linear discriminant analysis?

$$\begin{aligned}
 & \log \frac{\Pr(Y = k | X = x)}{\Pr(Y = j | X = x)} \\
 &= \log \frac{f_k(x)}{f_j(x)} + \log \frac{\pi_k}{\pi_j} \\
 &= \log \frac{\pi_k}{\pi_j} - \frac{1}{2} (\mu_k + \mu_j)^T \Sigma^{-1} (\mu_k + \mu_j) \\
 & \quad + x^T \Sigma^{-1} (\mu_k - \mu_j)
 \end{aligned}$$

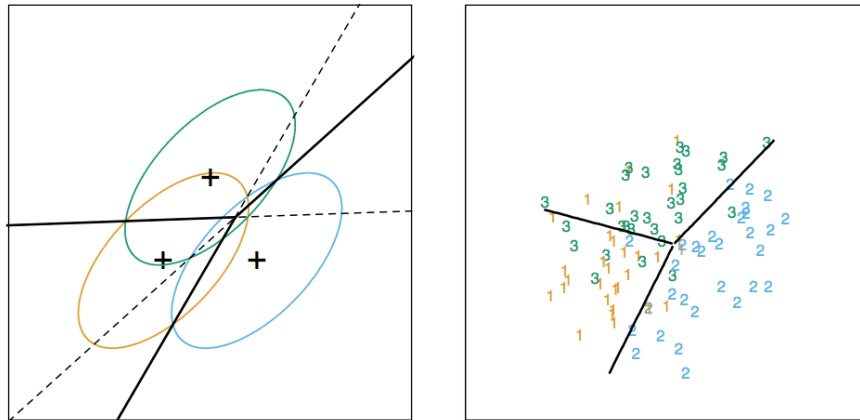
So, why is this called linear discriminant analysis? Well, if we consider the ratio of the probabilities of the two classes. So, this is the probability you're in class k, given our predictor variables ($X = x$), divided by the probability here in class j given the predictor variables. And we take the log of that quantity, so if we take the log of it, it's a monotone function which means that as this ratio increases, so will the log of that ratio. So we can look at this quantity, and we can see that that breaks down by basically writing these quantities out using base theorem.

As on the previous page, we get log of the ratio of the two Gaussian densities, $\log \frac{f_k(x)}{f_j(x)}$, plus log of the ratio of the two prior probabilities, $\log \frac{\pi_k}{\pi_j}$. Now the terms, $\log \frac{f_k(x)}{f_j(x)}$, can be written out as $-\frac{1}{2} (\mu_k + \mu_j)^T \Sigma^{-1} (\mu_k + \mu_j) + x^T (\mu_k - \mu_j)$. We get the log of the ratio of the prior probabilities, $\log \frac{\pi_k}{\pi_j}$, plus $-\frac{1}{2} (\mu_k + \mu_j)^T \Sigma^{-1} (\mu_k + \mu_j) + x^T (\mu_k - \mu_j)$ that depends on the

parameters of our Gaussian distributions, the normal distributions for each class, plus a linear term, an x variable $x^T(\mu_k - \mu_j)$ which is a linear terms so you end up getting basically lines that are drawn through the data.

A variable will have a higher probability of being of one class if it's on one side of the line and a higher probability of being in another class if it's on the other side of the line. You can go and read about it in the elements of statistical for a little bit more about the details.

Decision boundaries



This is what the decision boundaries tend to look like for these prediction models. Imagine we have three different groups of points. We're trying to classify into **class one**, **class two** or **class three**. And we have two variables that we're using to classify and they are the x and the y axis here. What we would end up doing is fitting one Gaussian distribution. Here's **one Gaussian distribution** and **another Gaussian distribution** and here's a **third Gaussian distribution**, one to each class. And then we would basically draw lines where the probability switches over from being higher in this class to higher in that class. That ends up as the black lines like the three spokes. If you're on this side of the line, you'll be **classified as a two**. If you're on this side of the line, you'll be **classified as a three**. And if you're down here, you'll be **classified as a one**. You basically fit Gaussian distributions to the data and then use those Gaussian distributions to draw lines that, assign the data (observation) points to the highest posterior probabilities.

Discriminant function

In general, the discriminate function is what gets used. So the basic idea is we have a discriminate function that looks like this where μ_k is the mean of class k for all our features.

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + \log(\mu_k)$$

- Decide on class based on $\hat{Y}(x) = \operatorname{argmax}_k \delta_k(x)$
- We usually estimate parameters with maximum likelihood

Σ^{-1} is the inverse of the co-variance matrix for that class. Actually, it's the same co-variance matrix for all classes here. And then this is the term, the linear term in x , x^T , our predictors, our co-variance matrix Σ^{-1} and the mean μ_k . And so basically what we do is we plug in our

new data value into this function. And we pick the value of k that produces the largest value of this particular discriminate function. And that's how we decide on a class. You can usually estimate these parameters by maximum likelihood estimation if you remember that from the inference class that you would've taken as part of the data science specialization.

Naive Bayes

Suppose we have many predictors, we would want to model: $P(Y = k|X_1, \dots, X_m)$

We could use Bayes Theorem to get:

$$P(Y = k|X_1, \dots, X_m) = \frac{\pi_k P(X_1, \dots, X_m|Y = k)}{\sum_{\ell=1}^K P(X_1, \dots, X_m|Y = \ell) \pi_\ell}$$

$$\propto \pi_k P(X_1, \dots, X_m|Y = k)$$

Now Naïve Bayes does something a little bit more to more to simplify the problem. Suppose we're trying to model the probability that y is in a particular class, k . And we have a bunch of variables that we want to predict with. We can use Bayes' theorem to say the probability that the class is k , given all the variables that we've observed, is the prior probability we're in class k , π_k , times the probability for all these features given you're in class k , $P(X_1, \dots, X_m|Y = k)$, divided by some constant, $\sum_{\ell=1}^K (\dots) \pi_\ell$. The way that this can be written is that this probability is proportional to the prior probability times the probability of the features, given that we're in class k , $\propto \pi_k P(X_1, \dots, X_m|Y = k)$. In other words, if you picked the largest value of this quantity, it will be the same as picking the largest probability of $P(Y = k|X_1, \dots, X_m)$ because the term in the denominator is just a constant for all the different probabilities.

This can be written:

$$P(X_1, \dots, X_m, Y = k) = \pi_k P(X_1|Y = k) P(X_2, \dots, X_m|X_1, Y = k)$$

$$= \pi_k P(X_1|Y = k) P(X_2|X_1, Y = k) P(X_3, \dots, X_m|X_1, X_2, Y = k)$$

$$= \pi_k P(X_1|Y = k) P(X_2|X_1, Y = k) \dots P(X_m|X_1, \dots, X_{m-1}, Y = k)$$

We can say the [joint] probability of the features and the class variable k is equal to this prior probability (π_k) times $P(X_1|Y = k)$ times the probability of all the other variables, except X_1 , conditional on X_1 and Y_k , $P(X_2, \dots, X_m|Y = k)$. So this is basically just a statement about probability. Continue to break down in the same way until you get one term for every feature. But those features are always conditional on all the other variables that have come before it. That's because each features, i.e., each of the predictors may be dependent on each other.

One assumption you could make to make this quite a bit easier would be to just assume that all of the predictor variables are independent of each other in which case they drop out of the conditioning argument. The result is to end up with the prior probability times the probability of each feature by itself conditional on being in each class.

$$\approx \pi_k P(X_1|Y = k) P(X_2|Y = k) \dots P(X_m|Y = k)$$

Now this is kind of a naive assumption because we're assuming that all the features are independent even though we know they're probably not. And that's why this method has the title **Naive Bayes**. And so, it still works reasonably well in a large number of applications. And it's particularly useful when you have a very large number of features that are binary or categorical variables. This very frequently get, comes up in text classification and classification of other kind of document classification.

Example: Iris Data

So just to show you briefly how these two work, I'm going to show you on the Iris Data again.

load data(iris); library(ggplot2) Note the variables, names(iris), to predict with and three different species (setosa, versicolor, virginica) to predict.

```
data(iris); library(ggplot2)
names(iris)
```

```
[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"
```

```
table(iris$Species)
```

setosa	versicolor	virginica
50	50	50

Create training and test sets

Fit them on a training set and apply them to a test set just like I do with all of our other prediction algorithms. Use the createDataPartition. Create a training set and a test set.

```
inTrain <- createDataPartition(y=iris$Species,
                                p=0.7, list=FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
dim(training); dim(testing)
```

```
[1] 45 5
```

Build predictions

Then build an lda model on the training set, in the following way. Use the train function from the caret package, apply it to the training set with method="lda". Then do similarly for naive Bayes with method="nb", a naïve Bayes classification. And I can predict the values from lda and from a naïve base on the test set and make a table of the predictions.

```
modlda = train(Species ~ ., data=training, method="lda")
modnb = train(Species ~ ., data=training, method="nb")
plda = predict(modlda, testing); pnb = predict(modnb, testing)
table(plda, pnb)
```

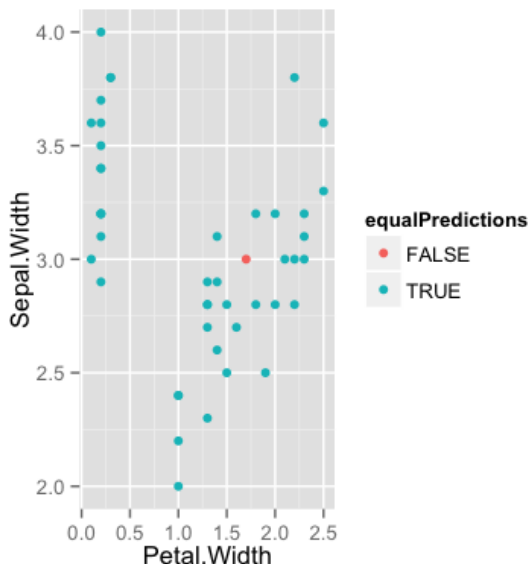
	pnb		
plda	setosa	versicolor	virginica
setosa	15	0	0
versicolor	0	13	1
virginica	0	0	16

And we can see that the predictions agree for all but one value. So even though we know that the features or the predictors are dependent here, using the naive Bayes classification means very similar prediction rules to the linear discriminate analysis classification and so we can do a comparison of the results.

Comparison of results

We see that just one value (1) between the two classes that appears to be not classified in the same way by the two outputs (plda, pnb) but overall they perform very similarly.

```
equalPredictions = (plda==pnb)
qplot(Petal.Width, Sepal.Width, colour=equalPredictions, data=testing)
```



red dot (FALSE); not classified the same.

Notes and further reading

- [Introduction to statistical learning](#)
- [Elements of Statistical Learning](#)
- [Model based clustering](#)
- [Linear Discriminant Analysis](#)
- [Quadratic Discriminant Analysis](#)

You can learn more about naïve Bayes classification and other model based classification in the Introduction to Statistical Learning and Elements of Statistical Learning books.

You can also learn about model based clustering in more detail in the academic paper that I've linked to. Find linear discriminant analysis and quadratic discriminant analysis on the Wikipedia pages.