Classification

- Classification
  - Overview
  - Methods
    - Logistic Regression
    - Linear Discriminant Analysis
    - Naïve Bayes
    - Point Bayes
- Decision Trees
  - Overview
- Support Vector Machines

ISL Chapter 4

ISL Chapter 8

R Files:

- *DA2_Classification_LogReg_intro*
- *DA2_Classification_RegReview*
- *DA2_Decision_Tree_intro*
- *DA2_Intuition*
- *DA2_LDA*
- *DA2_LogReg_Multinomial*
- *DA2_LogReg_Exercise*
- *DA2_Naive_Bayes*

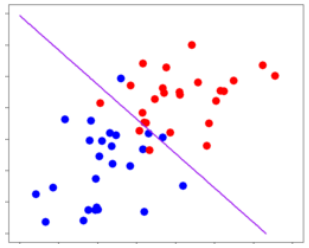
BAUER COLLEGE OF BUSINESS
UNIVERSITY of HOUSTON

Classification Methods

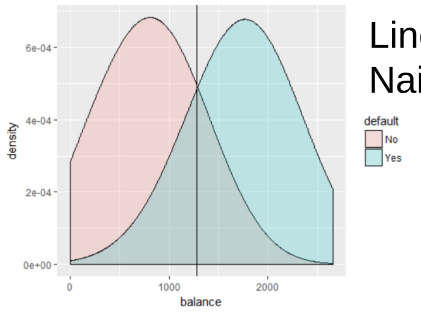
Classification is the problem of identifying to which of a set of categories *(sub-populations)* an observation belongs. Formally, given training set (x_i, y_i) for i=1…n, we want to create a classification model f that can determine the label y for x.

We'll survey a range of parametric and non-parametric algorithms:

Parametric

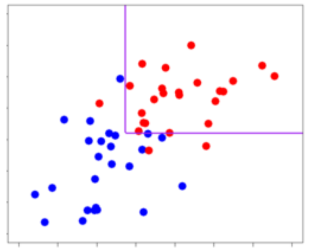


Logistic Regression

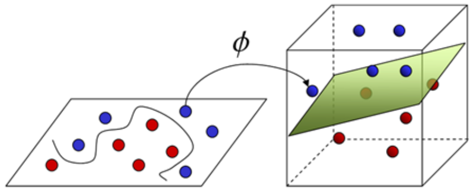


Linear Discriminant Analysis
Naive Bayes

Non-Parametric



Decision Trees



Support Vector Machines

Logistic Regression

The logistic model starts with a linear model:

$y = \beta_0 + \beta_1 X$ *where* P(y=1,0 | X)

Since we now want to model P(y = 1 | X), and we know that probability must be 0 > P(y) > 1. So, we transform the equation to exponential form *(so it's always > 0)* and to a reciprocal *(so it's always < 1):*

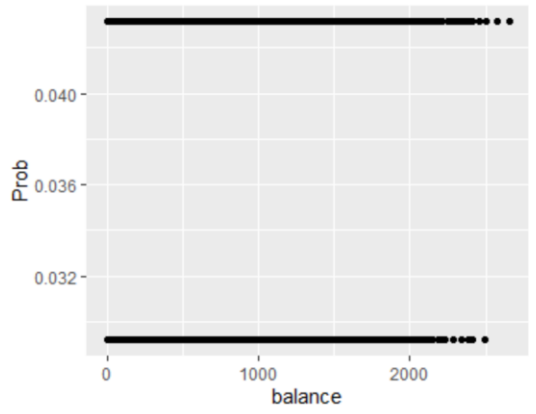
$$P(y) = \frac{e^{\beta_0+\beta_1 X}}{1+ e^{\beta_0+\beta_1 X}} \propto \log\left(\frac{P(x)}{1-P(x)}\right)$$

```
dfDefault <- Default
glm.fit <- glm(default ~ student, data = dfDefault, family = binomial)
summary(glm.fit)
```

```
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -3.50413    0.07071  -49.55  < 2e-16 ***
studentYes   0.40489    0.11502    3.52 0.000431 ***
```

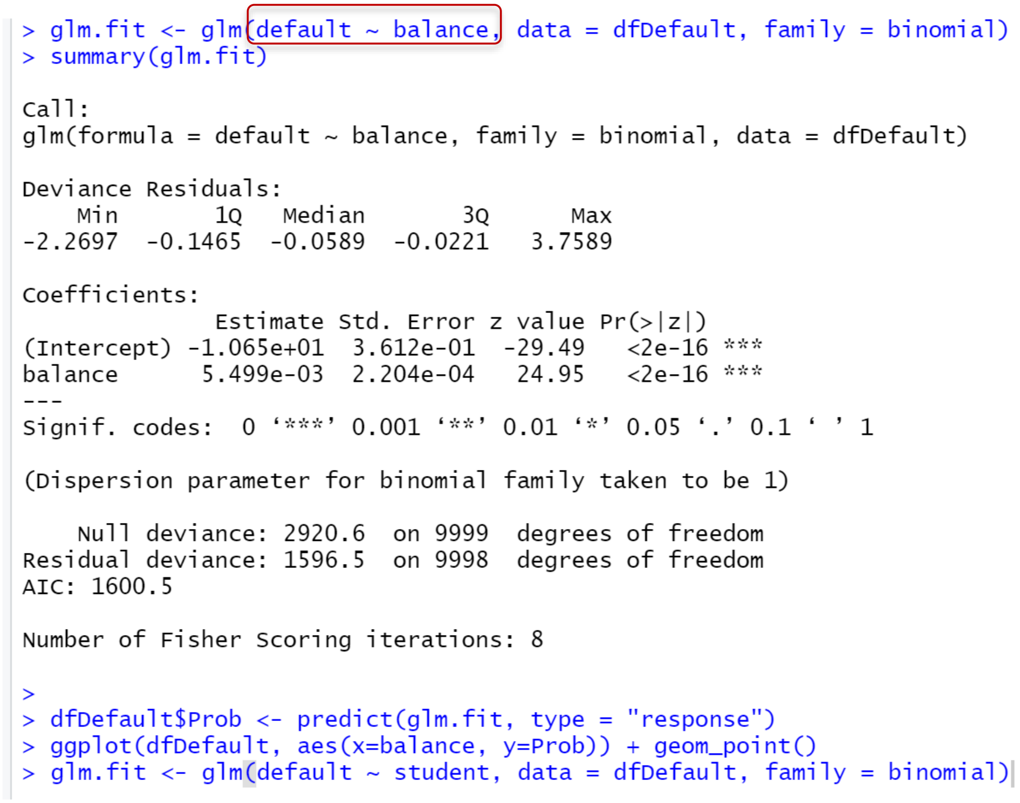
$$P(default = yes\,|student = \mathbf{yes}) = \frac{e^{-3.5041+0.4049\,*\mathbf{1}}}{1+ e^{-3.5041+0.4049\,*\mathbf{1}}} = .0431$$

$$P(default = yes\,|student = \mathbf{no}) = \frac{e^{-3.5041+0.4049\,*\mathbf{0}}}{1+ e^{-3.5041+0.4049\,*\mathbf{0}}} = .0292$$

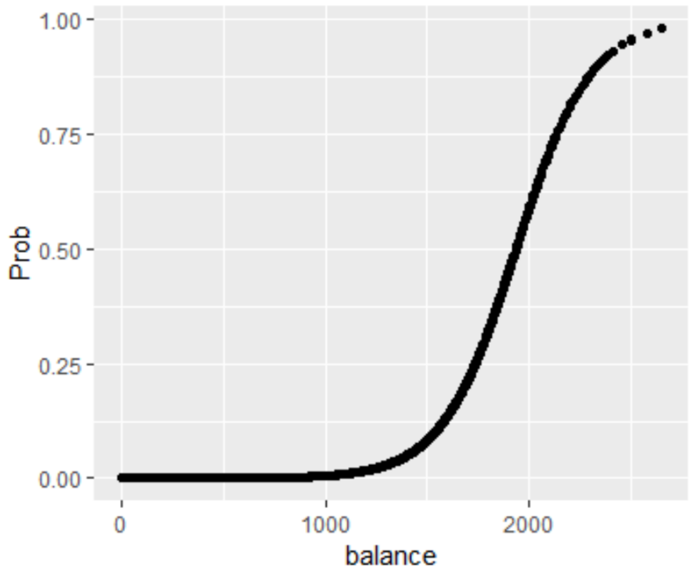


```
> (exp(-3.5041+ (0.4049 *1)))/ (1 + exp(-3.5041+ (0.4049 *1)))
[1] 0.04314027
> (exp(-3.5041+ (0.4049 * 0)))/ (1 + exp(-3.5041+ (0.4049 * 0)))
[1] 0.0291958
```

BAUER
COLLEGE OF BUSINESS
UNIVERSITY of HOUSTON

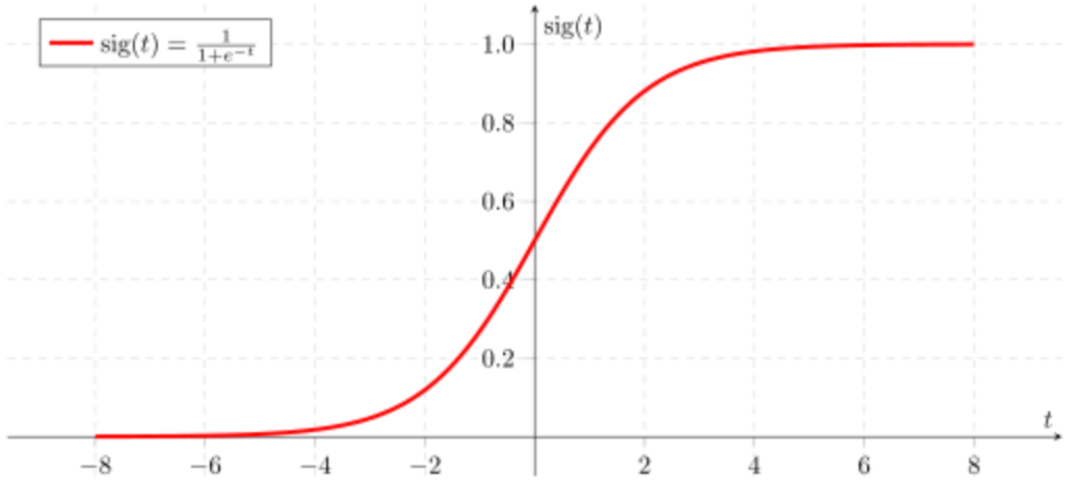


```
> glm.fit <- glm(default ~ balance, data = dfDefault, family = binomial)
> summary(glm.fit)

Call:
glm(formula = default ~ balance, family = binomial, data = dfDefault)

Deviance Residuals: 
    Min       1Q   Median       3Q      Max  
-2.2697  -0.1465  -0.0589  -0.0221   3.7589  

Coefficients:
              Estimate Std. Error z value Pr(>|z|)    
(Intercept) -1.065e+01  3.612e-01  -29.49   <2e-16 ***
balance      5.499e-03  2.204e-04   24.95   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 2920.6  on 9999  degrees of freedom
Residual deviance: 1596.5  on 9998  degrees of freedom
AIC: 1600.5

Number of Fisher Scoring iterations: 8

> 
> dfDefault$Prob <- predict(glm.fit, type = "response")
> ggplot(dfDefault, aes(x=balance, y=Prob)) + geom_point()
> glm.fit <- glm(default ~ student, data = dfDefault, family = binomial)
```



What is the Difference Between Logit and Logistic Regression?

POSTED ON NOVEMBER 5, 2018 BY ALEX



Logit and logistic regression are the same thing. However, they actually relate to generalized linear models. In a generalized linear model, you have some features x, parameters β, response y, and link function g. that relates $E(y)$ to x and β. The relationship is as follows:

$$g(E(y)) = \beta^T x \tag{1}$$

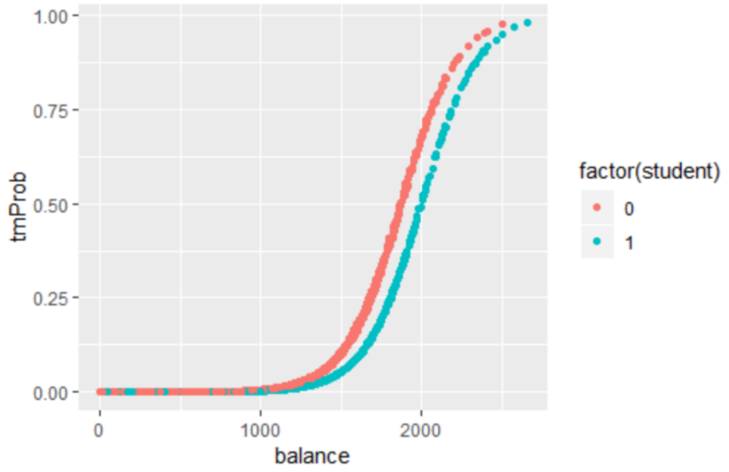
One choice of g is the logit function $\log \frac{x}{1-x}$. Its inverse, which is an activation function, is the logistic function $\frac{1}{1+\exp(-x)}$. Thus logit regression is simply the GLM when describing it in terms of its link function, and logistic regression describes the GLM in terms of its activation function.

BAUER COLLEGE OF BUSINESS
UNIVERSITY of HOUSTON

Multiple Logistic Regression

```
mglm.fit <- glm(default ~ student + balance + income, data = dfDefault, family = binomial)
summary(mglm.fit)
```

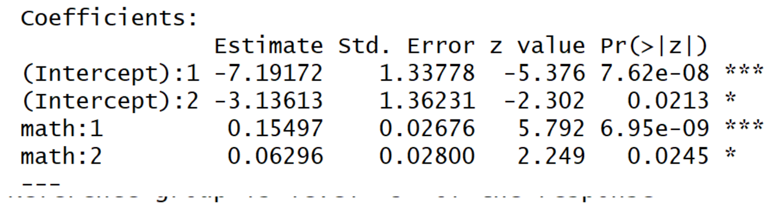
```
Coefficients:
              Estimate Std. Error z value Pr(>|z|)    
(Intercept) -1.087e+01  4.923e-01 -22.080  < 2e-16 ***
studentYes  -6.468e-01  2.363e-01  -2.738  0.00619 ** 
balance      5.737e-03  2.319e-04  24.738  < 2e-16 ***
income       3.033e-06  8.203e-06   0.370  0.71152    
```



```
alpha <- mglm.fit$coefficients[1]
beta <- mglm.fit$coefficients[2:4]

test <- dfDefault
test$student <- as.integer(dfDefault$student)-1

test$tProb <-  (exp(alpha[1]+(beta[1]*test[,2]+ beta[2]*test[,3]+beta[3]*test[,4])))/
  (1+(exp(alpha[1]+(beta[1]*test[,2]+ beta[2]*test[,3]+beta[3]*test[,4]))))
# or using matrix algebra to make this easier:
tst1 <- data.matrix(test[,2:4])
bet1 <- as.numeric(beta)
test$tmProb <- exp(alpha[1] + t(bet1%*%t(tst1)))/(1+exp(alpha[1] + t(bet1%*%t(tst1))))
# looks like just as much effort, but it's not when you're working!!
ggplot(test, aes(x=balance, y=tmProb, color = factor(student))) + geom_point()
```

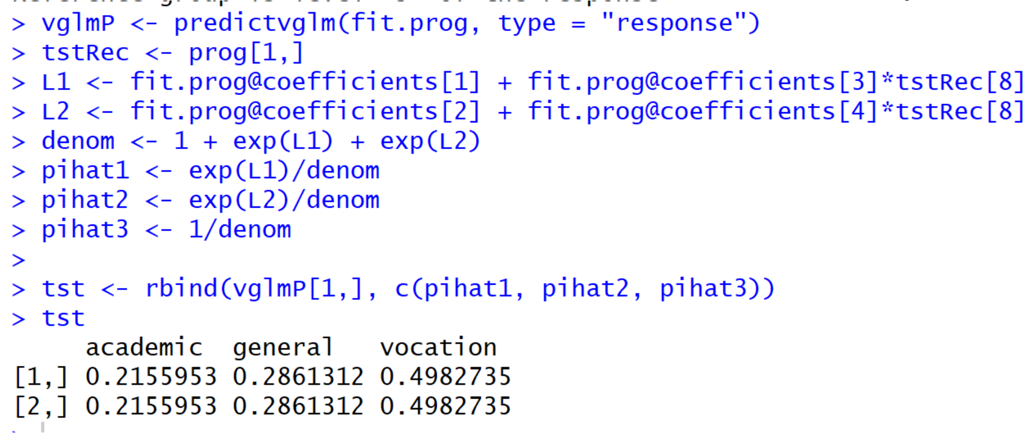
Multinomial Logistic Regression

```
setwd("C:/Users/ellen/Documents/Spring 2019/DA2/Section 1/Classification/Data")
prog <- read.csv("programs.csv")
prog$prog2 <- relevel(prog$prog, ref = "academic")
fit.prog <- vglm(prog ~ math, family = multinomial, data = prog)
coef(fit.prog, matrix = TRUE)
```



```
Coefficients:
               Estimate Std. Error z value Pr(>|z|)    
(Intercept):1 -7.19172    1.33778  -5.376 7.62e-08 ***
(Intercept):2 -3.13613    1.36231  -2.302   0.0213 *  
math:1         0.15497    0.02676   5.792 6.95e-09 ***
math:2         0.06296    0.02800   2.249   0.0245 *  
---
```

A multinomial logit model generalizes LogReg to a multiclass model. In simple models, we create a **reference *(or pivot)*** outcome, and all the rest of the nominal probabilities are independently regressed against that reference.



```
> vglmP <- predictvglm(fit.prog, type = "response")
> tstRec <- prog[1,]
> L1 <- fit.prog@coefficients[1] + fit.prog@coefficients[3]*tstRec[8]
> L2 <- fit.prog@coefficients[2] + fit.prog@coefficients[4]*tstRec[8]
> denom <- 1 + exp(L1) + exp(L2)
> pihat1 <- exp(L1)/denom
> pihat2 <- exp(L2)/denom
> pihat3 <- 1/denom
> 
> tst <- rbind(vglmP[1,], c(pihat1, pihat2, pihat3))
> tst
      academic   general  vocation
[1,] 0.2155953 0.2861312 0.4982735
[2,] 0.2155953 0.2861312 0.4982735
```

$$P_1 = \frac{e^{L1}}{1+e^{L1}+e^{L2}}$$

$$P_2 = \frac{e^{L2}}{1+e^{L1}+e^{L2}}$$

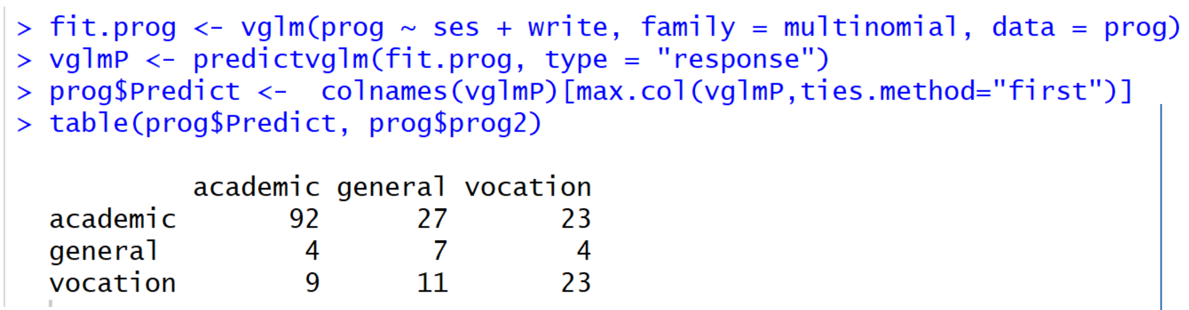
$$P_3 = \frac{1}{1+e^{L1}+e^{L2}}$$

$$P(program = academic\ |math = 41) = \frac{e^{-7.19172+0.15497*\mathbf{41}}}{1+e^{-7.19172+0.15497*\mathbf{41}}+e^{-3.13613+0.06296*\mathbf{41}}} = 0.2155953$$

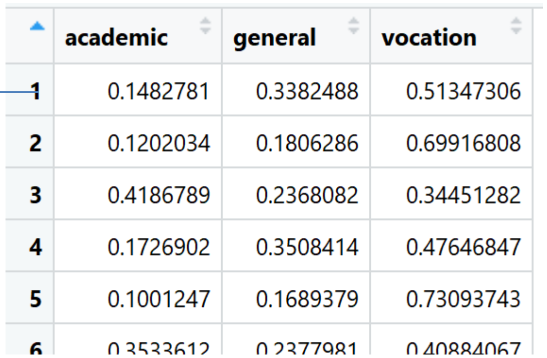
$$P(program = general\ |math = 41) = \frac{e^{-3.13613+0.6296*\mathbf{41}}}{1+e^{-7.19172+0.15497*\mathbf{41}}+e^{-3.13613+0.06296*\mathbf{41}}} = 0.2861312$$

$$P(program = vocation\ |math = 41) = \frac{1}{1+e^{-7.19172+0.15497*\mathbf{41}}+e^{-3.13613+0.06296*\mathbf{41}}} = 0.4982735$$

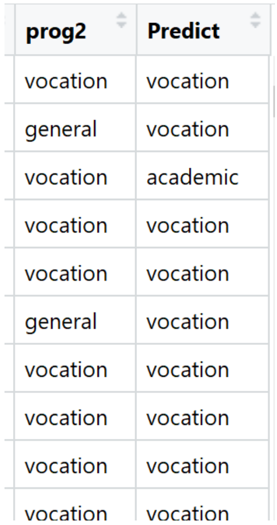
Expanding this model to multiple predictors, the model produces probabilities for each line, L, for each nominal outcome



```
> fit.prog <- vglm(prog ~ ses + write, family = multinomial, data = prog)
> vglmP <- predictvglm(fit.prog, type = "response")
> prog$Predict <-  colnames(vglmP)[max.col(vglmP,ties.method="first")]
> table(prog$Predict, prog$prog2)

          academic general vocation
 academic       92      27       23
 general         4       7        4
 vocation        9      11       23
```



	academic	general	vocation
1	0.1482781	0.3382488	0.51347306
2	0.1202034	0.1806286	0.69916808
3	0.4186789	0.2368082	0.34451282
4	0.1726902	0.3508414	0.47646847
5	0.1001247	0.1689379	0.73093743
6	0.3533612	0.2377981	0.40884067



prog2	Predict
vocation	vocation
general	vocation
vocation	academic
vocation	vocation
vocation	vocation
general	vocation
vocation	vocation
vocation	vocation
vocation	vocation
vocation	vocation

We're using vglm from the VGAM package here because it has a multinomial version of glm. This is not the most flexible approach to multinomial *(or multiclass)* analysis, and non-parametric algorithms will usually produce a lower error *(which doesn't mean it's better – remember the interpretability/flexibility tradeoff)*. It's almost always good baseline and extends conceptually into Bayesian multinomial modeling.

Just reviewing: we studied a GAM last week, which is a type of GLM that uses different functions within knots to fit data. It also uses a link function, which is the basis of the GLM:

Link Functions and the GLM:

log $\left(\frac{p(x)}{1-p(x)}\right)$ can be exprssed as $g(E(y)) = \beta_0 + \beta_1 X$ where g is a special case of a ***link function (a logit function in this section).*** This is used to transform a linear equation to nonlinear - like we did with log transforms in linear regression *(but the independent variables)*. The GLM depends on link functions, and creates a **model that does not assume a linear relationship, homoskedasticity, or normally distributed errors** *(so, it uses MLE instead of LS to solve - giving it a greater range (at increased complexity and decreased interpretability– central theme of the course).*

Family	**Default Link Function**
binomial	(link = "logit")
gaussian	(link = "identity")
Gamma	(link = "inverse")
inverse.gaussian	(link = "1/mu^2")
poisson	(link = "log")
quasi	(link = "identity", variance = "constant")
quasibinomial	(link = "logit")
quasipoisson	(link = "log")

The core of the GLM is expressing the combined influence of predictors as their weighted sum. As you can see, the GLM uses a range of link functions, so for example, the expected value of the predicted variable might be expressed as:

g(E(y)) = pdf(k, λ)

Recall the shape parameters from probability review

So we can use GLM for modeling distributions in which lm will fail, and still get parameters.

Logistic Regression Exercise

Using the quote history data, build a logistic regression model to predict whether an opportunity will result in a Win or Loss based on data about price, competition, RFP, ATP and customer requirements

```
quoteData <- dbGetQuery(con2,"
Select
([dbo].[Quote].[Competitor_Quote] - [dbo].[Quote].[Quote]) AS QuoteDiff
,[dbo].[Customer].[RSF]
,[dbo].[Quote].[Result]
,DATEDIFF(d, [dbo].[Quote].[Date_Submitted], [dbo].[Quote].[Date_Due]) AS RFPDiff
,DATEDIFF(d, [dbo].[Quote].[ATP], [dbo].[Quote].[Date_Required] ) AS ATPDiff
FROM [dbo].[Quote]
INNER JOIN
[dbo].[Customer] ON [dbo].[Quote].[Customer_ID] = [dbo].[Customer].[Customer_ID]
")

quoteData <- filter(quoteData, Result %in% c("W", "L"))
quoteData$Result <- as.integer(factor(quoteData$Result))-1
quoteData <- quoteData %>% rownames_to_column("SampleID")
quoteData$SampleID  <- as.numeric(quoteData$SampleID)
quoteData$QuoteDiff  <- quoteData$QuoteDiff/1000
quoteData$RSF <- as.integer(quoteData$RSF) # its really ordinal, but to make easier
train <- sample_n(quoteData, nrow(quoteData)-100)
test <- quoteData %>% anti_join(train, by = "SampleID")
```

Here, we're pulling the quote data from the server *(which you're familiar with)* and applying some more advanced *(from your perspective)* SQL functions to make our job easier.

A few transformations in R and splitting train and test *(100 records in test)**

Converting dates to integers that algorithms will understand ***(further discussion on this shortly)***

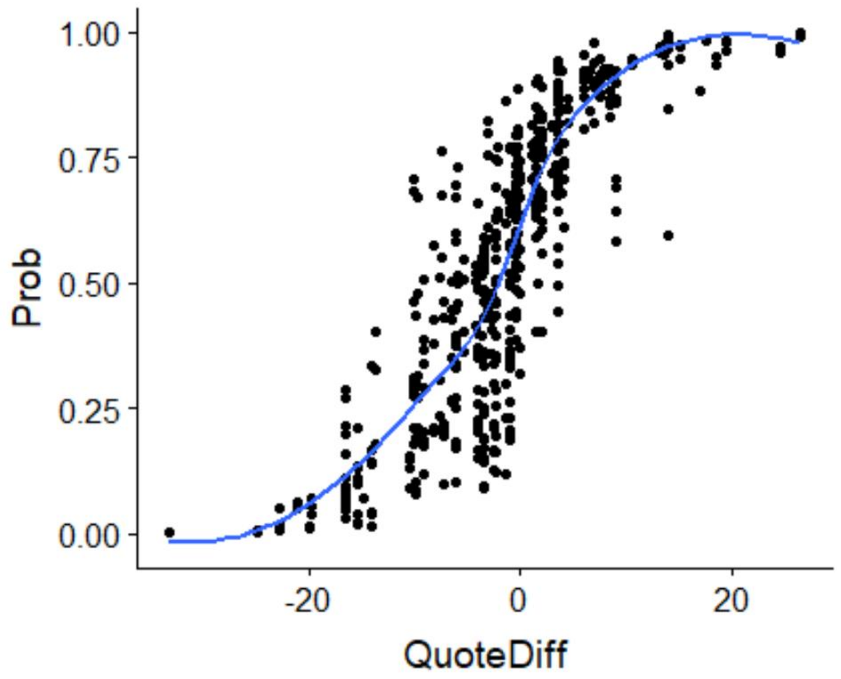
Convert and scale quote vs competitor quote to difference in quotes (which is what matters here)

RSF is an ordinal value and will work within the LogReg equation. Why?

Remove invalid data

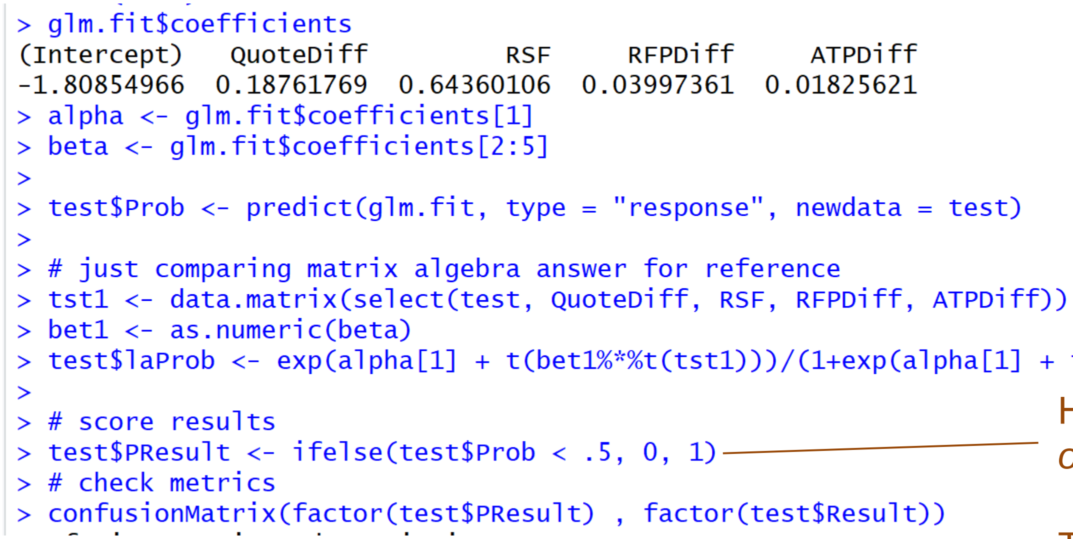
Convert variables in LogReg models to 0 and 1. Depending on the algorithm, you sometimes don't have to, but it always works.

```
glm.fit <- glm(Result ~ QuoteDiff + RSF + RFPDiff + ATPDiff, data = train, family = binomial)
summary(glm.fit)
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.808550   0.374556  -4.829 1.38e-06 ***
QuoteDiff    0.187618   0.017278  10.859  < 2e-16 ***
RSF          0.643601   0.107163   6.006 1.90e-09 ***
RFPDiff      0.039974   0.015097   2.648   0.0081 **
ATPDiff      0.018256   0.004095   4.458 8.28e-06 ***
train$Prob <- predict(glm.fit, type = "response")
ggplot(train, aes(x=QuoteDiff, y=Prob)) + geom_point() + geom_smooth(se = F)
```

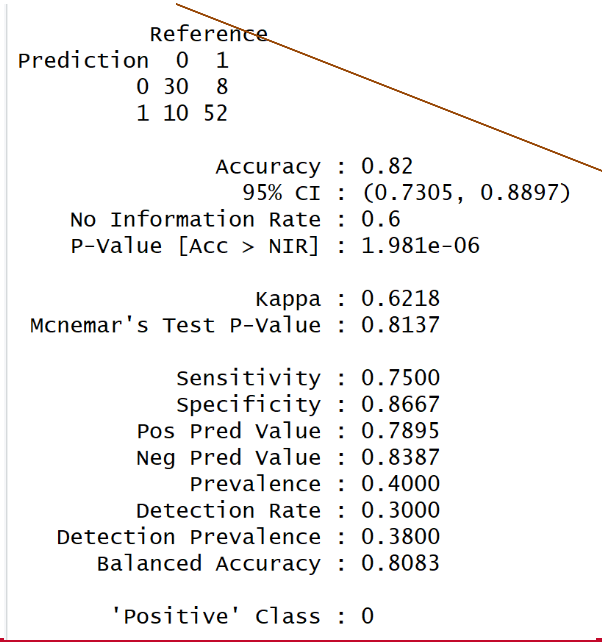


Here, we’re showing the probability of a win vs the different in quote vs competitor price (a negative means the competitor price is less than our price)

Note how this still follows a “log” shape, which is common on continuous variables



```
> glm.fit$coefficients
(Intercept)   QuoteDiff         RSF     RFPDiff     ATPDiff 
-1.80854966  0.18761769  0.64360106  0.03997361  0.01825621 
> alpha <- glm.fit$coefficients[1]
> beta <- glm.fit$coefficients[2:5]
> 
> test$Prob <- predict(glm.fit, type = "response", newdata = test)
> 
> # just comparing matrix algebra answer for reference
> tst1 <- data.matrix(select(test, QuoteDiff, RSF, RFPDiff, ATPDiff))
> bet1 <- as.numeric(beta)
> test$laProb <- exp(alpha[1] + t(bet1%*%t(tst1)))/(1+exp(alpha[1] + t(bet1%*%t(tst1))))
> 
> # score results
> test$PResult <- ifelse(test$Prob < .5, 0, 1)
> # check metrics
> confusionMatrix(factor(test$PResult) , factor(test$Result))
```



```
          Reference
Prediction  0  1
         0 30  8
         1 10 52
                                          
               Accuracy : 0.82            
                 95% CI : (0.7305, 0.8897)
    No Information Rate : 0.6             
    P-Value [Acc > NIR] : 1.981e-06       
                                          
                  Kappa : 0.6218          
 Mcnemar's Test P-Value : 0.8137          
                                          
            Sensitivity : 0.7500          
            Specificity : 0.8667          
         Pos Pred Value : 0.7895          
         Neg Pred Value : 0.8387          
             Prevalence : 0.4000          
         Detection Rate : 0.3000          
   Detection Prevalence : 0.3800          
      Balanced Accuracy : 0.8083          
                                          
       'Positive' Class : 0               
```

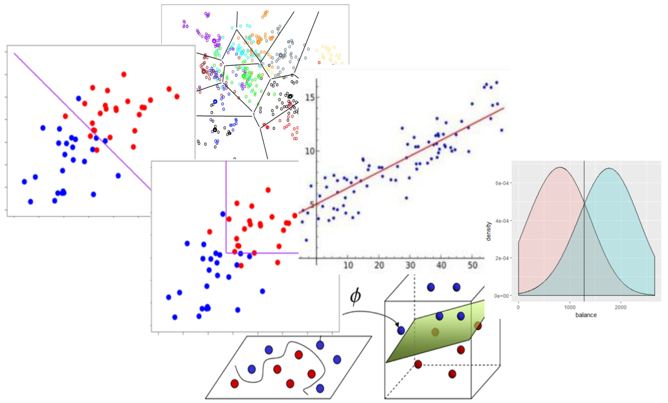
Here, we're converting probabilities *(the outcome of the equation)* to categories *(0, 1)*.

This is an important point – we can decide the level of probability breaks *(.5 is common in binomial models, but it doesn't have to be that way – as we'll soon see)*

I'm putting the output in a confusion matrix which gives us a number of metrics – we will study these shortly, but the LogReg model is performing well *(if you're getting 80% in a complex classifier, you're on the right track). There are many things we can do with tuning and resampling, which we'll study in the next couple of sections*

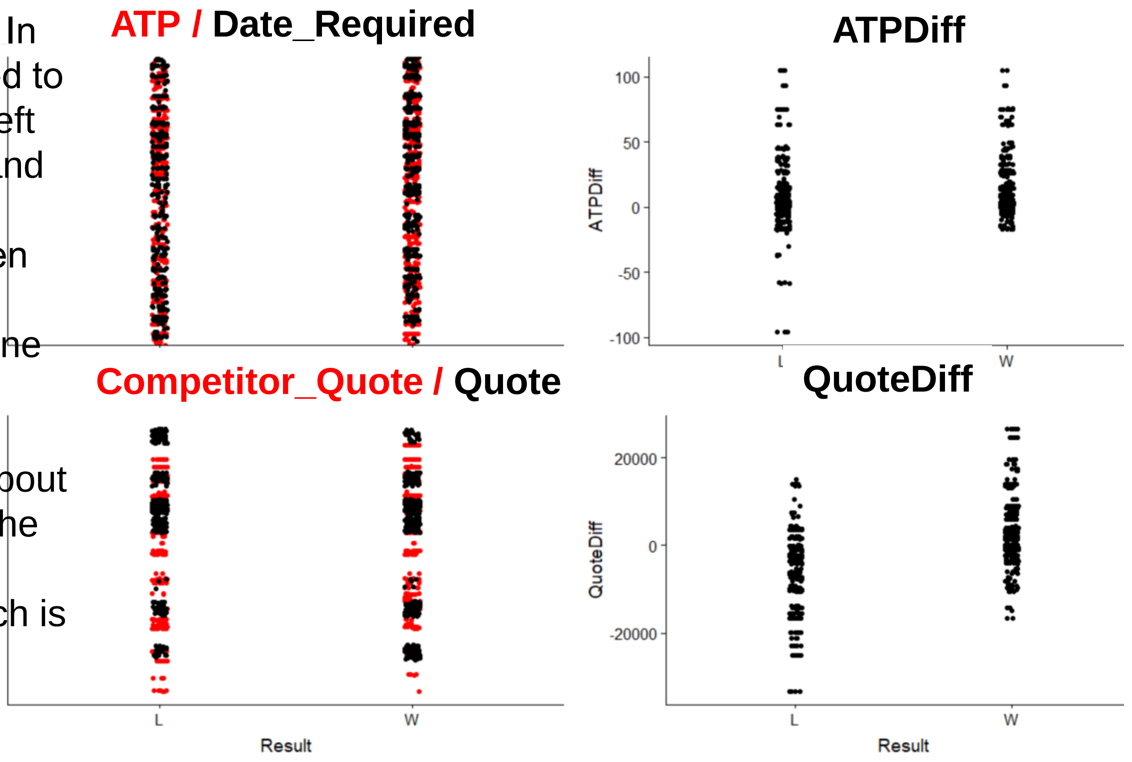
Care and Feeding of your Algorithms

There are many ingenious ways to define relationships and separate data, but they are all based on a common set of theories and rely on Euclidian distance and deltas – i.e., they have very sensitive digestive systems… a delicate constitution ☺



So, we have to be careful what we feed them. Pretend that you're an algorithm. In the frames to the left, you've been asked to differentiate between L and W. On the left side, you're given the ATP dates *(red)* and the Date Required *(black)*. On the right side, you're given the difference between the dates. Same with the quote data. Which one would you find easier to define and differentiate?

Also, remember what we said in DA1 about dates: dates are rarely a predictor. On the upper right, we're not measuring dates, we're measuring ability to deliver – which is the relevant issue



Linear Discriminant Analysis

In Logistic Regression, we directly model P (Y = k | X = x) using a logistic function.

In linear discriminant analysis, we model X given Y and then ***invert it using Bayes theorem***. LDA will produce a result comparable to Logistic Regression, but is more flexible in multi-class analysis.

$$P\,(Y = k \,/\, X = x) = \frac{\pi_k f_k(x)}{\sum_l^k \pi_l f_l(x)}$$

We can estimate prior π_k using the sample *(the prior probability that Y belongs to k class)*. The classifier assigns an observation to the class for which the *log likelihood* is largest.

Marginal Populations

$\mathbb{P}(\text{Default} = \ .0333)$

xmax	GrpProb
250	0.000
750	0.006
1250	0.072
1750	0.390
2250	0.477
2750	0.054

$\mathbb{P}(\text{Default} = \ .9667)$

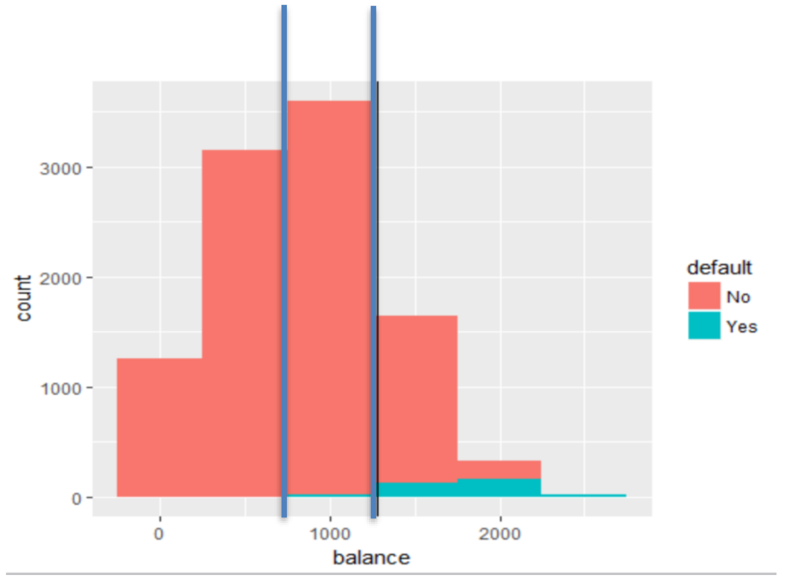
xmax	GrpProb
250	0.130
750	0.325
1250	0.369
1750	0.157
2250	0.018
2750	0.000

$$\mathbb{P}(\text{Default} = \text{Yes} \mid \text{Balance} = 1000)$$
$$= \frac{\mathbb{P}(Balance=1000 \mid \text{Default} = \text{Yes}) \ * \mathbb{P}(Default = Yes)}{\mathbb{P}(Balance=1000\,)}$$

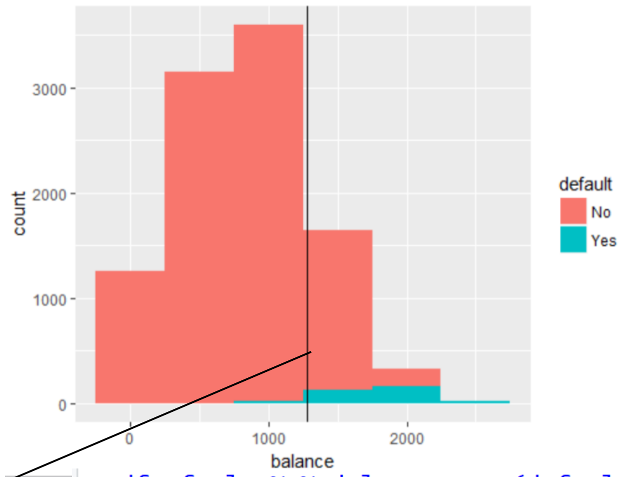
$$\frac{..072 \ * .033}{.359} = .007$$

$$\mathbb{P}(\text{Default} = \text{No} \mid \text{Balance} = 1000)$$
$$= \frac{\mathbb{P}(Balance=1000 \mid \text{Default} = \text{No}) \ * \mathbb{P}(Default = No)}{\mathbb{P}(Balance=1000\,)}$$

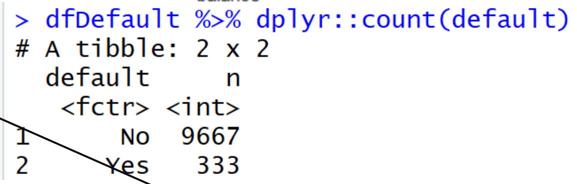
$$\frac{.369 \ * .967}{.359} = .993$$



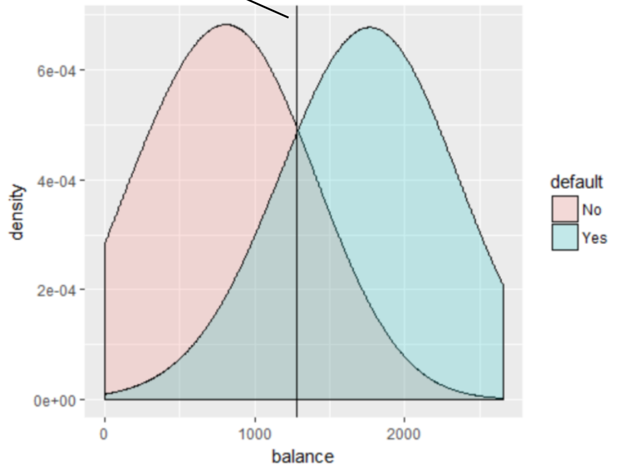
```
library(tidyverse)
library(MASS)
library(ISLR)

dfDefault <- Default

p <- ggplot(dfDefault, aes(balance, fill = default)) +
  geom_histogram(binwidth = 500)
p

pl1 <- ggplot(dfDefault, aes(balance, fill = default))
pl1 <- pl1 + geom_density(alpha = 0.2, adjust = 5 )
pl1

lda.fit <- lda(default ~ balance, data = dfDefault)
lda.fit

lda.pred <- predict(lda.fit)

pl1 <- pl1 + geom_vline(xintercept = mean(lda.fit$means) )
pl1
p <- p + geom_vline(xintercept = mean(lda.fit$means) )
p
```



$$\frac{u_1 + u_2}{2}$$



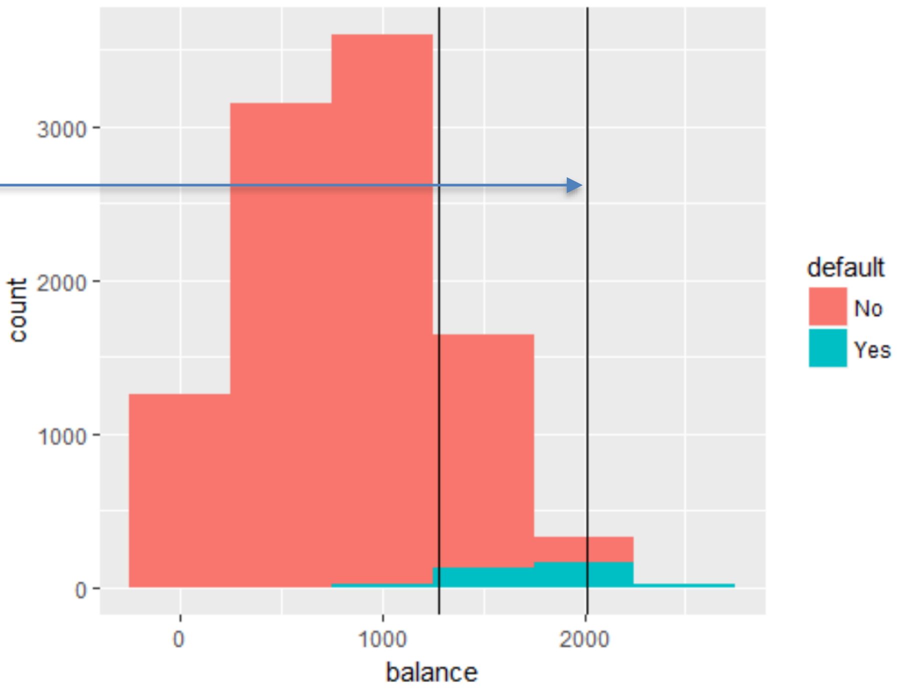
```
> dfDefault %>% dplyr::count(default)
# A tibble: 2 x 2
  default     n
   <fctr> <int>
1      No  9667
2     Yes   333
```



```
# get **decision rule**
A <- A <- mean(lda.fit$means)
B <- log(lda.fit$prior[2]) - log(lda.fit$prior[1])
s2.k <- t(tapply(dfDefault$balance, dfDefault$default, var)) %*%
lda.fit$prior
C <- s2.k/(lda.fit$means[1] - lda.fit$means[2])
dr <- A + B * C
dr

p <- p + geom_vline(xintercept = dr )
p
```

```
> dr <- A + B * C
> dr
         [,1]
[1,] 2008.554
```

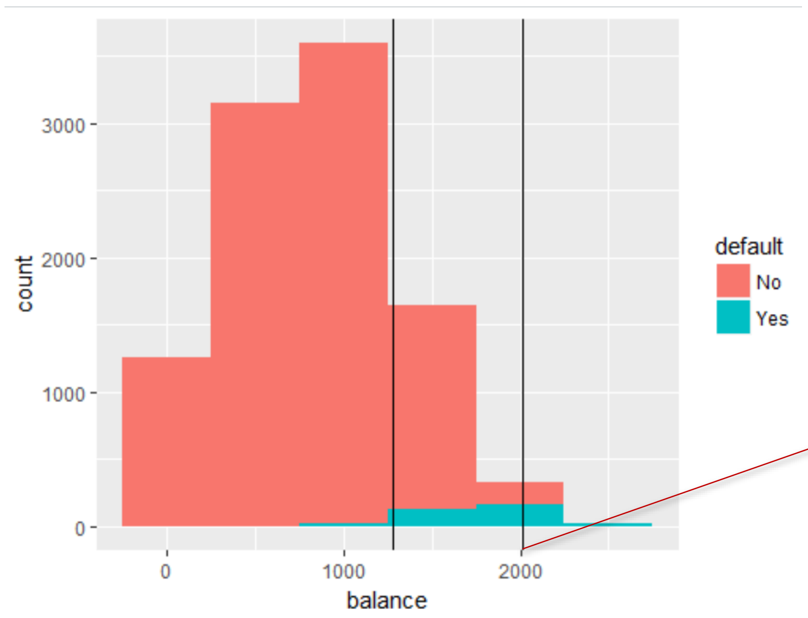
The classification boundary (decision rule) is not the average of the means.The decision rule is computed above (we're not going to cover the formula for the boundary)

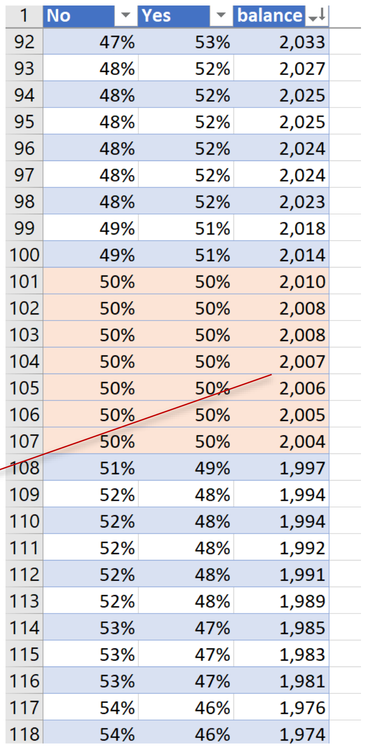


BAUER
COLLEGE OF BUSINESS
UNIVERSITY of HOUSTON

```
# get back original and look at it:
finalAnalysis <- as_tibble(cbind(as.character(lda.pred$class),
as.character(xTest$default), lda.pred$posterior))
write_csv(finalAnalysis, "finalAnalysis.csv")
```

Notice how the decision rule is implemented (2009 was the amt calculated by the dr function)





1	No	Yes	balance
92	47%	53%	2,033
93	48%	52%	2,027
94	48%	52%	2,025
95	48%	52%	2,025
96	48%	52%	2,024
97	48%	52%	2,024
98	48%	52%	2,023
99	49%	51%	2,018
100	49%	51%	2,014
101	50%	50%	2,010
102	50%	50%	2,008
103	50%	50%	2,008
104	50%	50%	2,007
105	50%	50%	2,006
106	50%	50%	2,005
107	50%	50%	2,004
108	51%	49%	1,997
109	52%	48%	1,994
110	52%	48%	1,994
111	52%	48%	1,992
112	52%	48%	1,991
113	52%	48%	1,989
114	53%	47%	1,985
115	53%	47%	1,983
116	53%	47%	1,981
117	54%	46%	1,976
118	54%	46%	1,974

Initial Performance Metrics

Confusion Matrix

```
tab <- table(lda.pred$class, dfDefault$default,
         dnn = c('Predicted', 'Actual'))

addmargins(tab)

sens <- tab[2,2]/(tab[1,2] + tab[2,2]); sens
spec <- tab[1,1]/(tab[1,1] + tab[2,1]); spec
er <- mean(lda.pred$class != dfDefault$default); er
```

	Actual		
		Negative	Positive
Predicted	Negative	True Negative	False Negative
	Positive	False Positive	True Positive

```
> tab <- table(lda.pred$class, dfDefault$default,
+              dnn = c('Predicted', 'Actual'))
>
> addmargins(tab)
          Actual
Predicted    No   Yes   Sum
      No   9643   257  9900
      Yes    24    76   100
      Sum  9667   333 10000
>
> sens <- tab[2,2]/(tab[1,2] + tab[2,2]); sens
[1] 0.2282282
> spec <- tab[1,1]/(tab[1,1] + tab[2,1]); spec
[1] 0.9975173
> er <- mean(lda.pred$class != dfDefault$default); er
[1] 0.0281
```

Sensitivity (also called the **true positive rate**, the **recall**) measures the proportion of positives that are correctly identified. 76/(76+257) = **.23**
Specificity (also called the **true negative rate**) measures the proportion of negatives that are correctly identified. 9643/(9643+24) = **.99**

TP	76	**Precision**	**0.76**
FP	24	**Recall**	**0.23**
TN	9643	**F1**	**0.35**
FN	257		

Precision	**TP/(TP+FP)**
Recall	**TP/(TP+FN)**
F1	**2*(Precision*Recall)/(Precision + Recall)**

So, out of 333 defaults, we correctly predicted 76 (true positive), missing 257 (false negative). Note that the Pos / Neg follows the prediction – what we're interested in getting right! And we also incorrectly predicted 24 false positives. Not too hot!

Building Understanding:

A precision score of 1.0 for a class C means that every item labeled (predicted) as belonging to class C does indeed belong to class C – a True Positive, but says nothing about the number of items from class C that were not labeled correctly *(False Negatives - FNs)*.

A recall of 1.0 means that every item from class C was labeled as belonging to class C *(TPs)*, but says nothing about how many other items were incorrectly also labeled as belonging to class C *(False Positives – FPs)*.

Often, there is an inverse relationship between precision and recall, where it is possible to increase one at the cost of reducing the other.

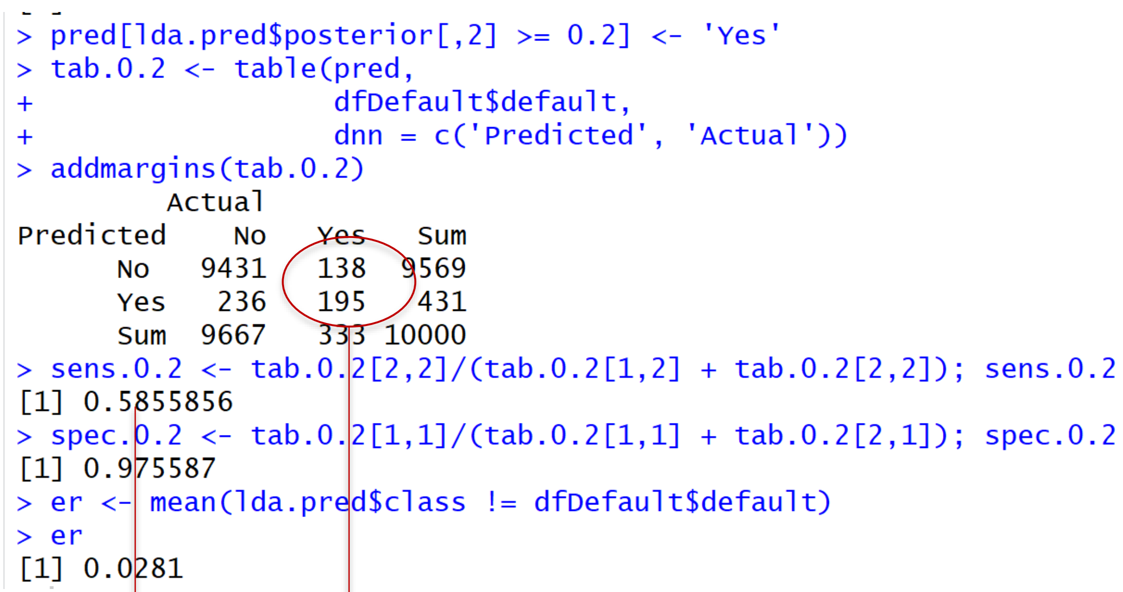
Often, both are combined into a single measure. The F-measure *(the weighted harmonic mean of precision and recall)*

TP	76	**Precision**	**0.76**
FP	24	**Recall**	**0.23**
TN	9643	**F1**	**0.35**
FN	257		

Precision	**TP/(TP+FP)**
Recall	**TP/(TP+FN)**
F1	**2*(Precision*Recall)/(Precision + Recall)**

Lowering the Threshold

```
pred[lda.pred$posterior[,2] >= 0.2] <- 'Yes'
tab.0.2 <- table(pred,
               dfDefault$default,
               dnn = c('Predicted', 'Actual'))
addmargins(tab.0.2)

sens.0.2 <- tab.0.2[2,2]/(tab.0.2[1,2] +
tab.0.2[2,2]); sens.0.2
spec.0.2 <- tab.0.2[1,1]/(tab.0.2[1,1] +
tab.0.2[2,1]); spec.0.2
er <- mean(lda.pred$class !=
dfDefault$default)
er
```



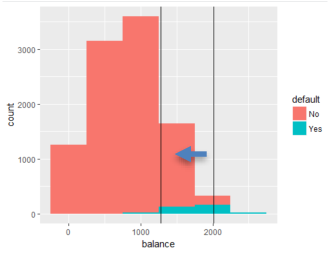
```
> pred[lda.pred$posterior[,2] >= 0.2] <- 'Yes'
> tab.0.2 <- table(pred,
+                   dfDefault$default,
+                   dnn = c('Predicted', 'Actual'))
> addmargins(tab.0.2)
          Actual
Predicted    No   Yes   Sum
      No   9431   138  9569
      Yes   236   195   431
      Sum  9667   333 10000
> sens.0.2 <- tab.0.2[2,2]/(tab.0.2[1,2] + tab.0.2[2,2]); sens.0.2
[1] 0.5855856
> spec.0.2 <- tab.0.2[1,1]/(tab.0.2[1,1] + tab.0.2[2,1]); spec.0.2
[1] 0.975587
> er <- mean(lda.pred$class != dfDefault$default)
> er
[1] 0.0281
```

TP	195	**Precision**	**0.45**
FP	236	**Recall**	**0.59**
TN	9431	**F1**	**0.51**
FN	138		

So, while Precision decreases, recall increases and F1 increases as the threshold for labeling default is lowered.

In simpler terms, the ratio of true positives to total positives from 23% to 58% (sensitivity)

So now, out of 333 defaults, we correctly predicted 195 true positives (76 before), missing false negatives 138 (257 before). So better. But we also incorrectly predicted 236 false positives (24 before) so that's worse. Think about how the decision boundary "shifted" to include more defaults



More Real World now

testSplit <- .4
totalSampleSize <- nrow(dfDefault)
testSampleSize <- round(totalSampleSize*testSplit)
trainSampleSize <- totalSampleSize - testSampleSize
tindexes <- sample(1:nrow(dfDefault), testSampleSize)
indexes <- sample(1:nrow(dfDefault[-tindexes,]), trainSampleSize)
xTrain <- dfDefault[indexes,]
xTest <- dfDefault[tindexes,]

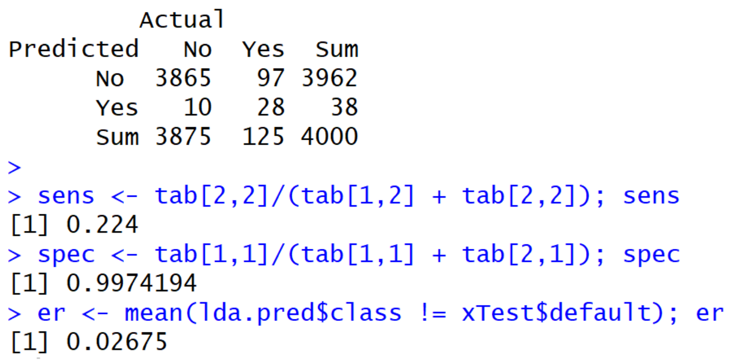
lda.fit <- lda(default ~ balance, xTrain)
lda.fit
lda.pred <- predict(lda.fit, xTest)

pl1 <- pl1 + geom_vline(xintercept = mean(lda.fit$means))
pl1
p <- p + geom_vline(xintercept = mean(lda.fit$means))
p

tab <- table(lda.pred$class, xTest$default,
            dnn = c('Predicted', 'Actual'))

addmargins(tab)

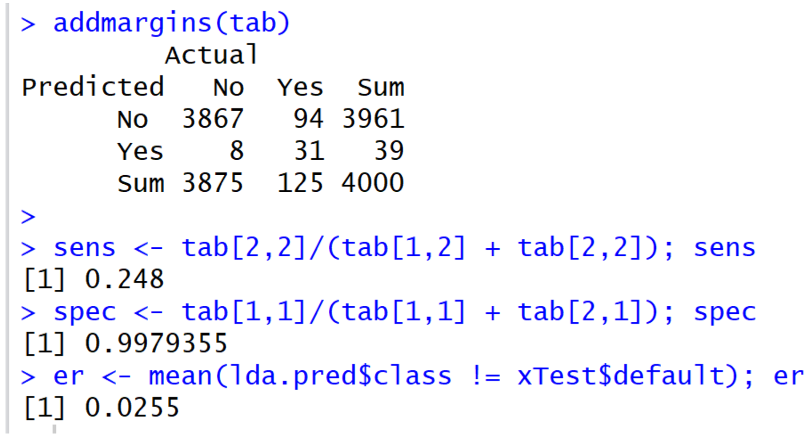
sens <- tab[2,2]/(tab[1,2] + tab[2,2]); sens
spec <- tab[1,1]/(tab[1,1] + tab[2,1]); spec
er <- mean(lda.pred$class != xTest$default); er



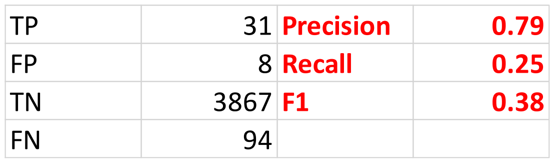
```
         Actual
Predicted   No  Yes  Sum
      No  3865   97 3962
      Yes   10   28   38
      Sum 3875  125 4000
>
> sens <- tab[2,2]/(tab[1,2] + tab[2,2]); sens
[1] 0.224
> spec <- tab[1,1]/(tab[1,1] + tab[2,1]); spec
[1] 0.9974194
> er <- mean(lda.pred$class != xTest$default); er
[1] 0.02675
```

TP	28	**Precision**	**0.74**
FP	10	**Recall**	**0.22**
TN	3865	**F1**	**0.34**
FN	97		

```
# add more predictors (p >1)

lda.fit <- lda(default ~ ., xTrain)
lda.fit

lda.pred <- predict(lda.fit, xTest)

pl1 <- pl1 + geom_vline(xintercept =
mean(lda.fit$means[,2]) )
pl1
p <- p + geom_vline(xintercept = mean(lda.fit$means[,2]) )
p

tab <- table(lda.pred$class, xTest$default,
        dnn = c('Predicted', 'Actual'))

addmargins(tab)

sens <- tab[2,2]/(tab[1,2] + tab[2,2]); sens
spec <- tab[1,1]/(tab[1,1] + tab[2,1]); spec
er <- mean(lda.pred$class != xTest$default); er
```



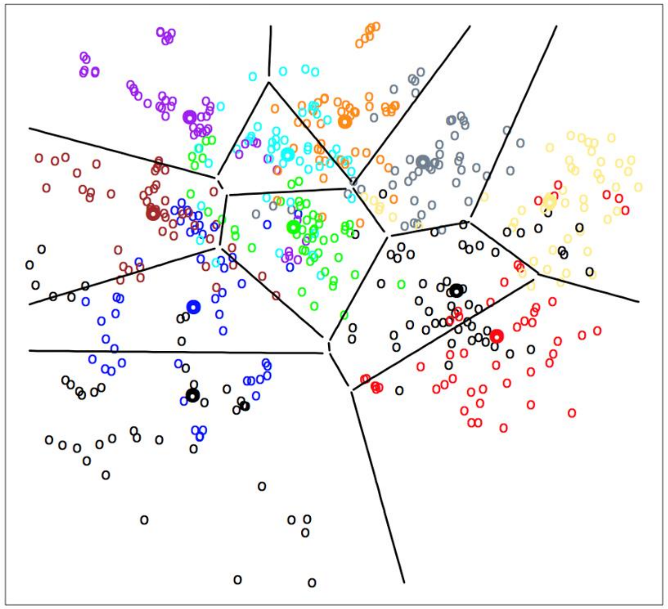
```
> addmargins(tab)
         Actual
Predicted   No  Yes  Sum
      No  3867   94 3961
      Yes    8   31   39
      Sum 3875  125 4000
>
> sens <- tab[2,2]/(tab[1,2] + tab[2,2]); sens
[1] 0.248
> spec <- tab[1,1]/(tab[1,1] + tab[2,1]); spec
[1] 0.9979355
> er <- mean(lda.pred$class != xTest$default); er
[1] 0.0255
```



TP	31	**Precision**	**0.79**
FP	8	**Recall**	**0.25**
TN	3867	**F1**	**0.38**
FN	94		

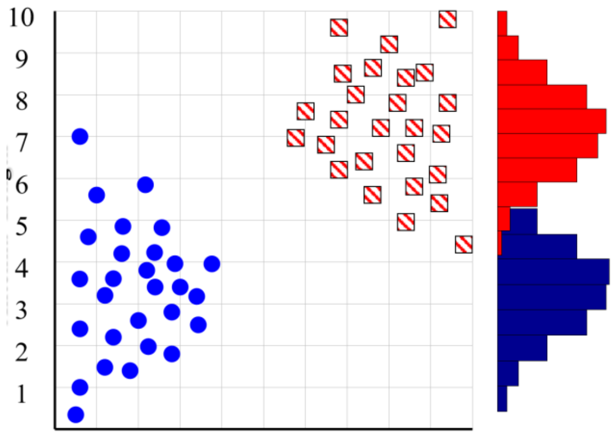
We have a range of sampling tools to improve this *(we'll cover in the resampling section)*, but more complex approaches are often used *(e.g., bayesian networks)* with imbalanced data if prediction accuracy is critical.

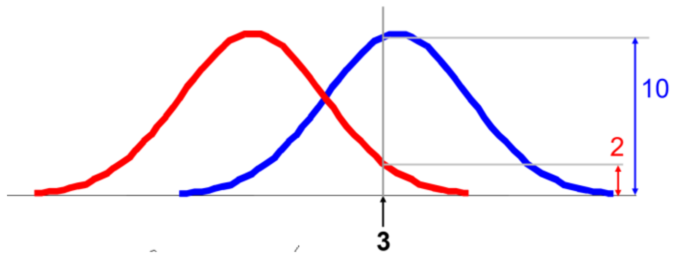
Multiclass LDA



LDA is very common in complex multiclass analysis – you have more control and extensibility, but with some loss of flexibility to non-parametric clustering

Naïve Bayes

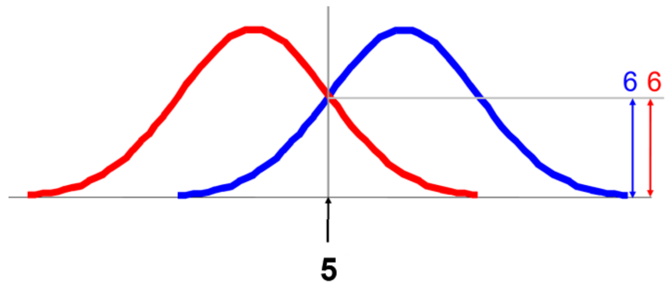




$P(A \mid 3) = \frac{10}{(10+2)} = .833$

$P(B \mid 3) = \frac{2}{(10+2)} = .166$

LDA is closely related to NB in that both classifiers assume Gaussian within-class distributions. However, NB relies on a less flexible distributional model in that it assumes zero off-diagonal covariance *(no correlations between variables within a class – i.e. NB assumes variables are independent, which is why it's naïve).*



LDA generally operates better on continuous, parametric features, whereas Naïve Bayes operates better on categorical, non-parametric features.

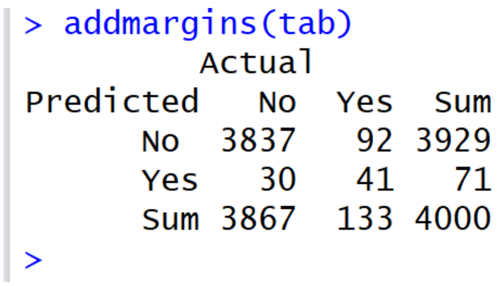
$P(A \mid 5) = \frac{6}{(6+6)} = .5$

$P(B \mid 5) = \frac{6}{(6+6)} = .5$

Caveat: just because the population is not Gaussian distributed does not mean the model is invalid (remember George Box)

Naïve Bayes

```
model <- naiveBayes(default ~ student + balance + income,
data = xTrain)
xTest$pred <- predict(model, xTest[,-1], prob = TRUE)

tab <- table(xTest$pred, xTest$default,
            dnn = c('Predicted', 'Actual'))
addmargins(tab)

# just checking
nrow(filter(xTest, default == "Yes"))

CM <- as_tibble(tab)

sens <- tab[2,2]/(tab[1,2] + tab[2,2]); sens
spec <- tab[1,1]/(tab[1,1] + tab[2,1]); spec
er <- mean(xTest$pred != xTest$default); er
```



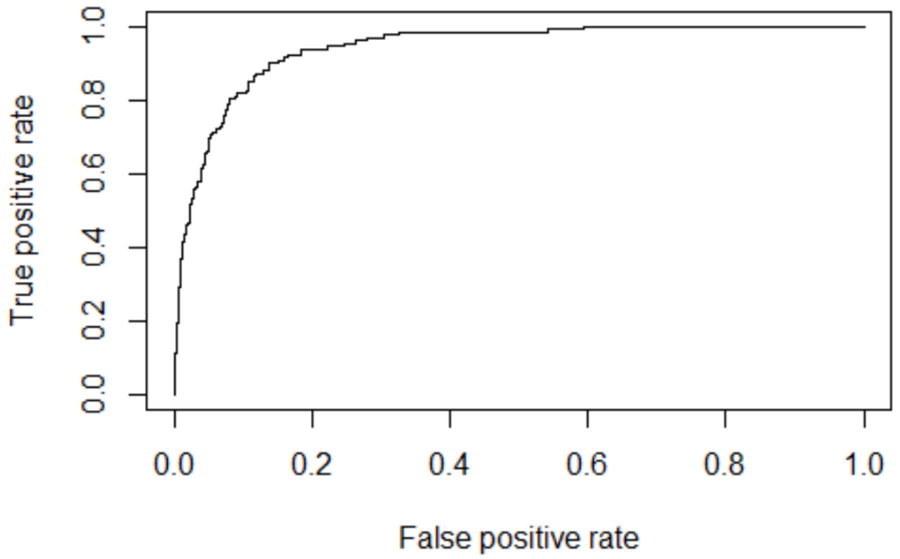
```
> addmargins(tab)
          Actual
Predicted    No  Yes  Sum
      No   3837   92 3929
      Yes    30   41   71
      Sum  3867  133 4000
>
```

TP	41	**Precision**	**0.58**
FP	30	**Recall**	**0.31**
TN	3837	**F1**	**0.40**
FN	92		

```
library(ROCR)
```

# ROC Curve

```
yTest <- xTest$default
pred <- prediction(probs[, "Yes"], xTest$default)
perf_nb <- performance(pred, measure = 'tpr', x.measure
= 'fpr')
plot(perf_nb)
auc<- performance(pred,"auc")
auc
```

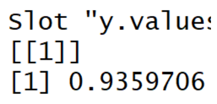


The ingredients of a ROC curve are true positive rate = TP/P (# positives correctly classified / total positives in dataset) and false positive rate FP/N (# negatives incorrectly classified / total negatives).

ROC curves are insensitive to changes in class distribution and balance. if you down-sample (covered soon) by cutting N in half, TP/P doesn't change at all. FP/N might not change much either,

This applies to AUC too. If TP (and True positive rate – TPR) includes a minority, or unbalanced class, then the rate (and AUC) will not be significantly affected by misclassification of minority class items.

The solution to these issues will be covered in Resampling.



```
Slot "y.value
[[1]]
[1] 0.9359706
```

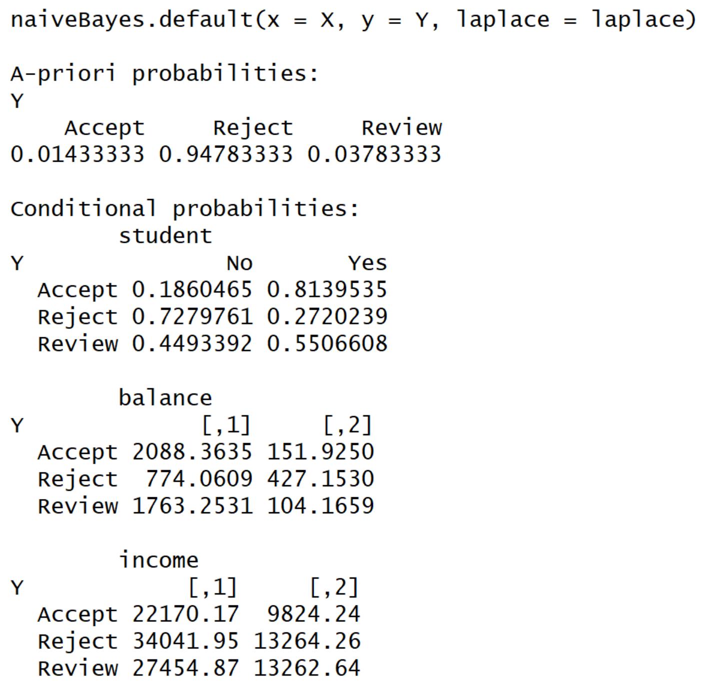
$$A = \int_{\infty}^{-\infty} \mathrm{TPR}(T)\left(-\mathrm{FPR}'(T)\right)$$

The area under the curve (often referred to as simply the AUC) is equal to the probability that a classifier will rank a randomly true positive instance higher than a false negative one

Multiclass Classification

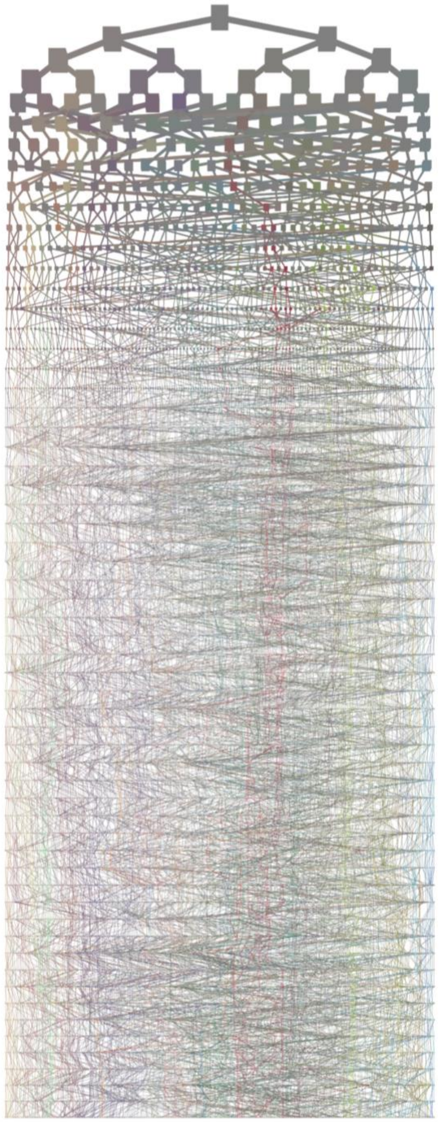
```
# -----------------------  multiclass ----------------------#

dfDefault <- Default

model <- naiveBayes(default ~ student + balance + income, data = dfDefault)
probs <- predict(model, dfDefault[,-1], type = 'raw')
dfDefault <- cbind(dfDefault, probs)
multiAnalysis <- mutate(dfDefault, route = ifelse((Yes < 0.2), "Accept", ifelse((Yes >= .2 & Yes <= .5), "Review", "Reject")))
multiAnalysis  %>% group_by(route) %>% summarize(count = n())
dfRouting <- dplyr::select(multiAnalysis, route, student, balance, income)
dfRouting$route <- as.factor(dfRouting$route)

testSplit <- .4
totalSampleSize <- nrow(dfRouting)
testSampleSize <- round(totalSampleSize*testSplit)
trainSampleSize <- totalSampleSize - testSampleSize
tindexes <- sample(1:nrow(dfRouting), testSampleSize)
indexes <- sample(1:nrow(dfRouting[-tindexes,]), trainSampleSize)
xTrain <- dfRouting[indexes, ]
xTest <- dfRouting[tindexes,]

model <- naiveBayes(route ~ student + balance + income,  data = xTrain)
model
```

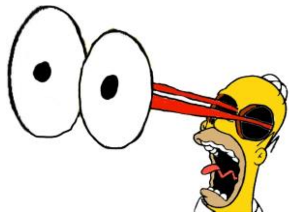
Here, we just arbitrarily assigned routing classes to observations based on the posterior probabilities of the 2 class problem – this is really cheesy, but just trying to set up a quick multiclass problem.



```
naiveBayes.default(x = X, y = Y, laplace = laplace)

A-priori probabilities:
Y
    Accept     Reject     Review 
0.01433333 0.94783333 0.03783333 

Conditional probabilities:
        student
Y               No       Yes
  Accept 0.1860465 0.8139535
  Reject 0.7279761 0.2720239
  Review 0.4493392 0.5506608

        balance
Y             [,1]     [,2]
  Accept 2088.3635 151.9250
  Reject  774.0609 427.1530
  Review 1763.2531 104.1659

        income
Y            [,1]     [,2]
  Accept 22170.17  9824.24
  Reject 34041.95 13264.26
  Review 27454.87 13262.64
```

Decision Trees





We've looked at LDA and Naïve Bayes classifiers, which work by determining probability based on a likelihood of a distribution, and applying a decision rule to separate the classes.

Trees work in a very different way

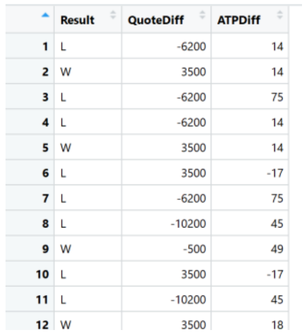
Decision Tree Algorithms

CART (Classification and Regression Trees) Algorithm
C4.5 (Quinlan 1993) and others

CART works by ***Recursive Binary Splitting*** which is a top-down, greedy approach. It is top-down because it begins at the top of the tree *(at which point all observations belong to a single region)* and then successively splits the predictor space; each split is indicated via **two new branches** further down on the tree, until we can't divide it any longer – in which case we add a leaf *(the number of leaves can be limited)*.

It is greedy because at each step of the tree-building process, **the best split is made at that particular step, rather than looking ahead** and picking a split that will lead to a better tree in some future step. Then, we;

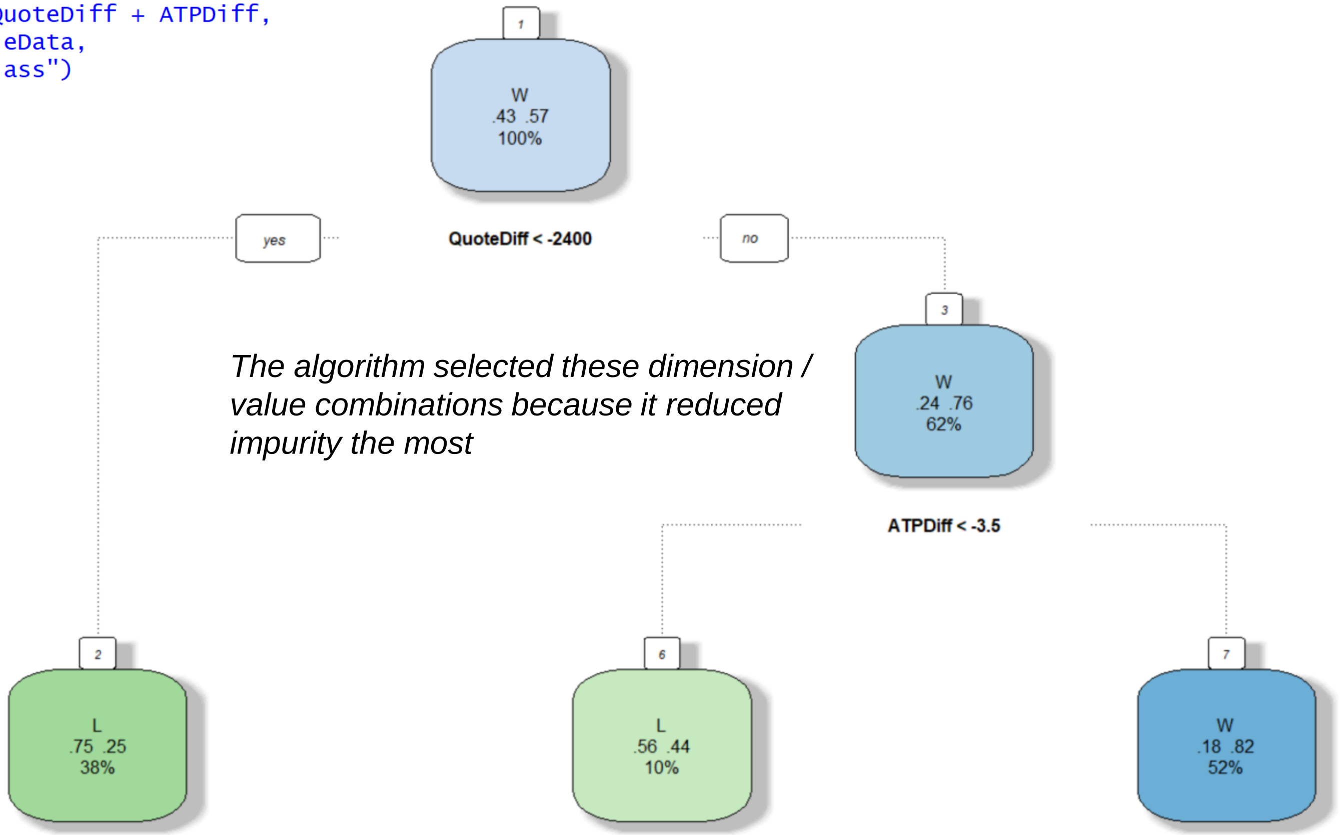
1. Apply cost complexity **pruning** to the large tree in order to obtain a sequence of best subtrees, as a function of α *(a cost function)*
2. Use K-fold ***cross-validation*** to choose α *(usually a parameter)*. That is, divide the training observations into K folds. For each k =1,...,K:.
3. Return the subtree from Step 2 that corresponds to the chosen value of α, which minimizes error.



	Result	QuoteDiff	ATPDiff
1	L	-6200	14
2	W	3500	14
3	L	-6200	75
4	L	-6200	14
5	W	3500	14
6	L	3500	-17
7	L	-6200	75
8	L	-10200	45
9	W	-500	49
10	L	3500	-17
11	L	-10200	45
12	W	3500	18

The algorithm iterates over the dimensions and selects a dimension / value combination based on a cost function, and creates a question. It then partitions the data into two groups based on that question (true or false). The best question is the one that that reduced uncertainty the most (the metric for this is called Gini impurity)

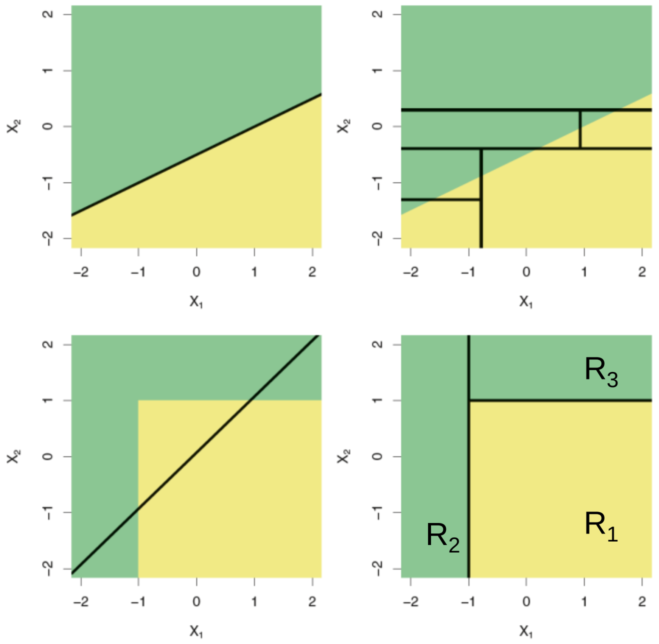
```
> fit <- rpart(Result ~ QuoteDiff + ATPDiff,
+              data=SampleData,
+              method="class")
>
> fancyRpartPlot(fit)
```



The algorithm selected these dimension / value combinations because it reduced impurity the most

Trees vs. Linear Models

Regions *(leaves)*

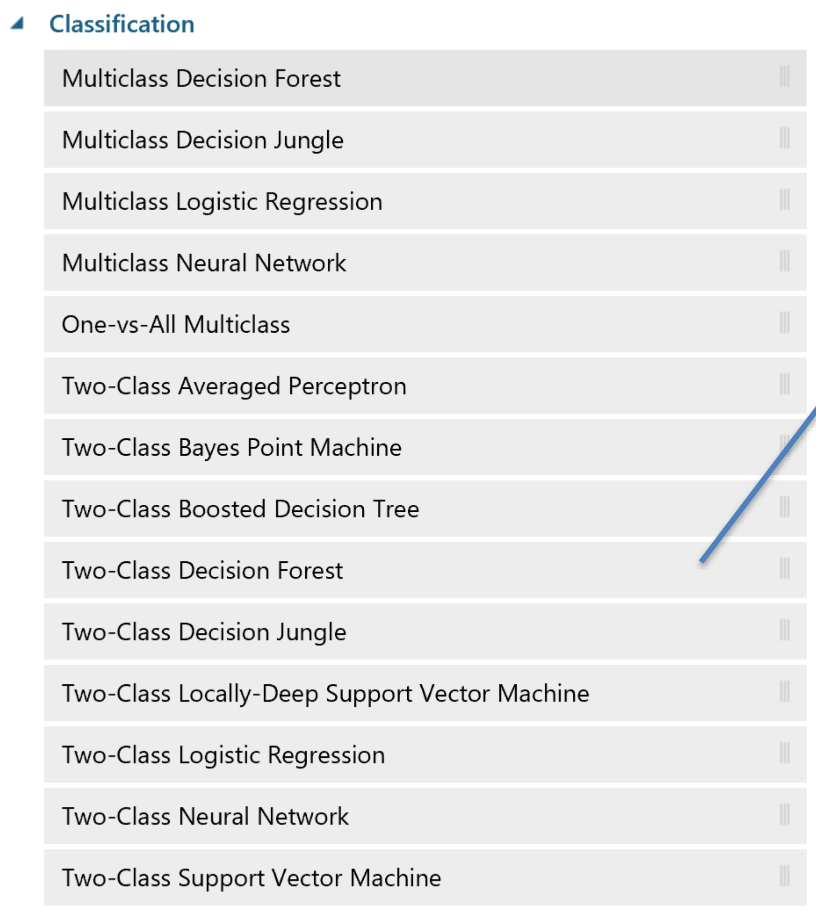


From the Book *(pg 315)*.

Top Row: A two-dimensional classification example in which the true decision boundary is linear, and is indicated by the shaded regions. A classical approach that assumes a linear boundary (left) will outperform a decision tree that performs splits parallel to the axes (right).

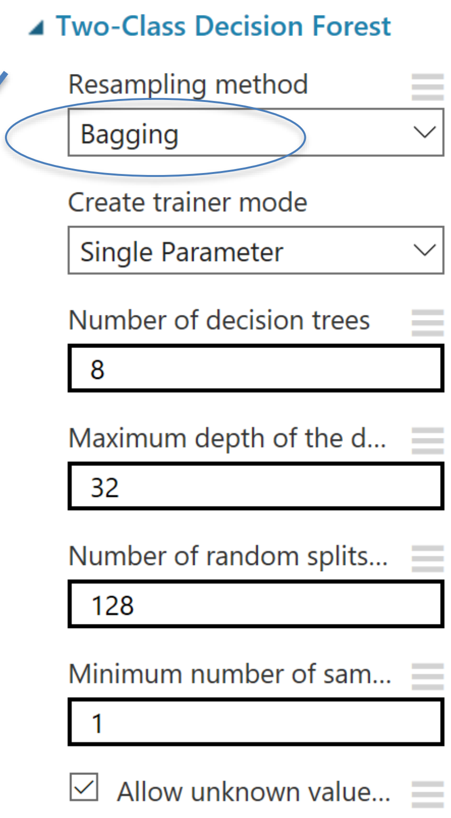
Bottom Row: Here the true decision boundary is non-linear. Here a linear model is unable to capture the true decision boundary (left), whereas a decision tree is successful (right).

Decision Tree Models



Classification

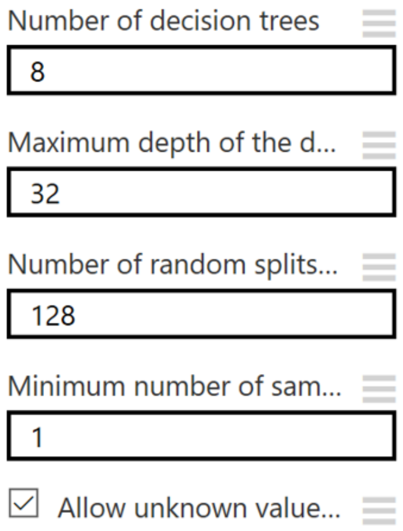
- Multiclass Decision Forest
- Multiclass Decision Jungle
- Multiclass Logistic Regression
- Multiclass Neural Network
- One-vs-All Multiclass
- Two-Class Averaged Perceptron
- Two-Class Bayes Point Machine
- Two-Class Boosted Decision Tree
- Two-Class Decision Forest
- Two-Class Decision Jungle
- Two-Class Locally-Deep Support Vector Machine
- Two-Class Logistic Regression
- Two-Class Neural Network
- Two-Class Support Vector Machine



Two-Class Decision Forest

Resampling method	[✓ Bagging]
Create trainer mode	Single Parameter
Number of decision trees	8
Maximum depth of the d...	32
Number of random splits...	128
Minimum number of sam...	1
☑ Allow unknown value...	

Every tree algorithm behaves differently with different datasets. You will have to read the manual. Here's an description with an Azure tree parameters:



Number of decision trees: 8

Maximum depth of the d...: 32

Number of random splits...: 128

Minimum number of sam...: 1

☑ Allow unknown value...

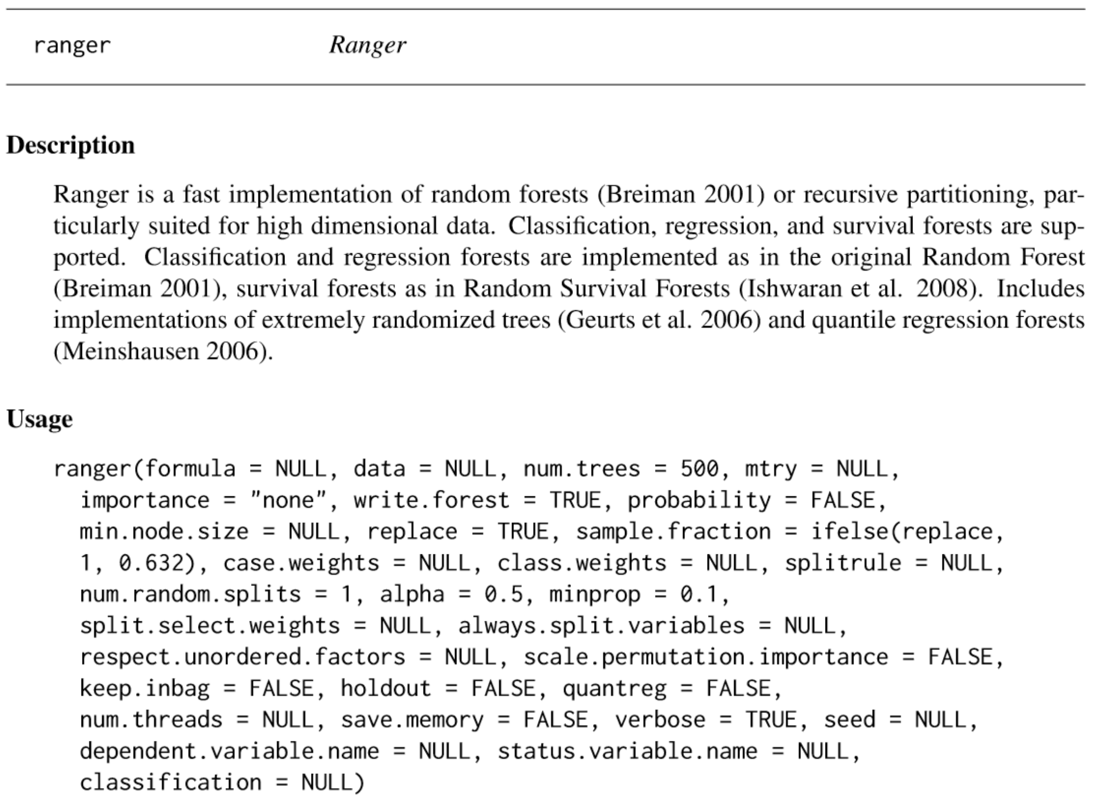
Many algorithms can create new trees as needed

Increasing the levels increases precision, but overfitting becomes a problem *(you have to watch variance with trees – error in actual data)*

type the number of splits to use when building each node of the tree.

minimum number of cases that are required to create any terminal node (leaf) in a tree - the threshold for creating new rules.

https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/



ranger *Ranger*

Description

Ranger is a fast implementation of random forests (Breiman 2001) or recursive partitioning, particularly suited for high dimensional data. Classification, regression, and survival forests are supported. Classification and regression forests are implemented as in the original Random Forest (Breiman 2001), survival forests as in Random Survival Forests (Ishwaran et al. 2008). Includes implementations of extremely randomized trees (Geurts et al. 2006) and quantile regression forests (Meinshausen 2006).

Usage

```
ranger(formula = NULL, data = NULL, num.trees = 500, mtry = NULL,
  importance = "none", write.forest = TRUE, probability = FALSE,
  min.node.size = NULL, replace = TRUE, sample.fraction = ifelse(replace,
  1, 0.632), case.weights = NULL, class.weights = NULL, splitrule = NULL,
  num.random.splits = 1, alpha = 0.5, minprop = 0.1,
  split.select.weights = NULL, always.split.variables = NULL,
  respect.unordered.factors = NULL, scale.permutation.importance = FALSE,
  keep.inbag = FALSE, holdout = FALSE, quantreg = FALSE,
  num.threads = NULL, save.memory = FALSE, verbose = TRUE, seed = NULL,
  dependent.variable.name = NULL, status.variable.name = NULL,
  classification = NULL)
```

R tree documentation is unique to each package. Here’s ranger:
https://cran.r-project.org/web/packages/ranger/ranger.pdf

R trees are the same way.

There are some consistent concepts, and we will focus on those concepts in ISL for the exam.

Bagging

(Bootstrap Aggregation)*

B training sets

model

model

model

Average *Models*

No need for cross validation – on average, each bagged tree uses 2/3 of observations. We then use the remaining 1/3 (called out-of-bag – OOB) is used to validate

Random Forests

Starts with bagging but restricts predictors (p) to a random sample of m predictors. This ***eliminates over-influence by strong predictors*** (i.e., the difference between random forests and bagging is the predictor subset size)

Boosting

Does not use bootstrapping, it uses a modified sampling that ***fits the residuals rather than the predicted value***. More complex, slower learning algorithm *(slower learners are often more accurate, but with higher processing costs)*

**In simple terms: Bootstrapping refers to resampling with replacement*

Recall the LDA homework.

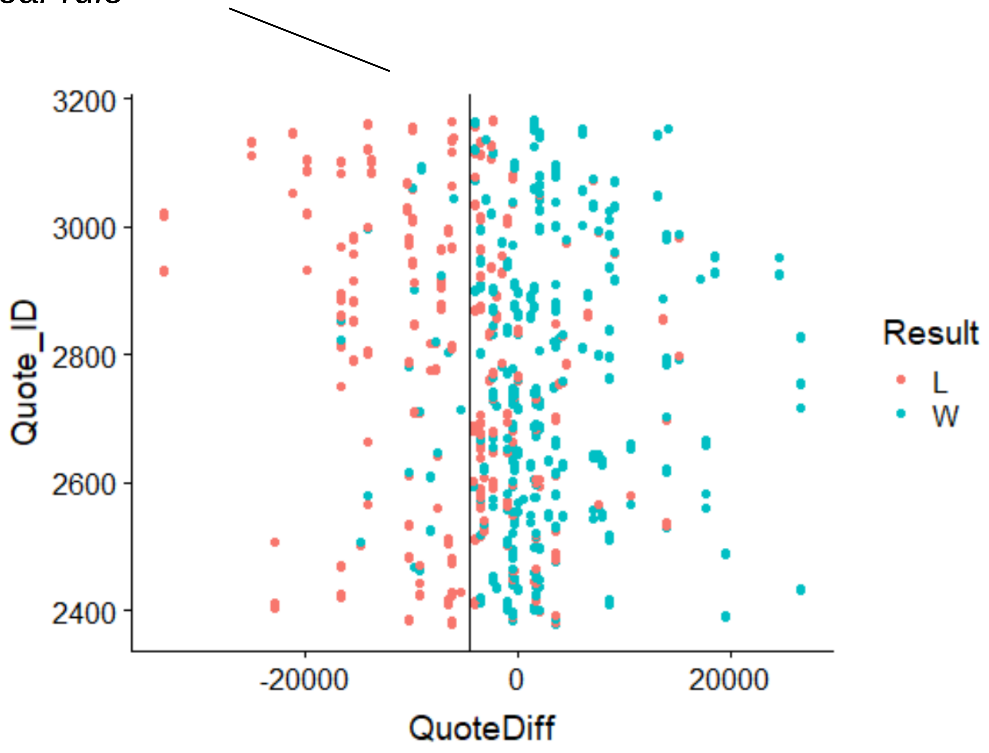
How do we separate the classes?

With LDA, we used a decision rule which is a linear rule

So, we have some strong assumptions:

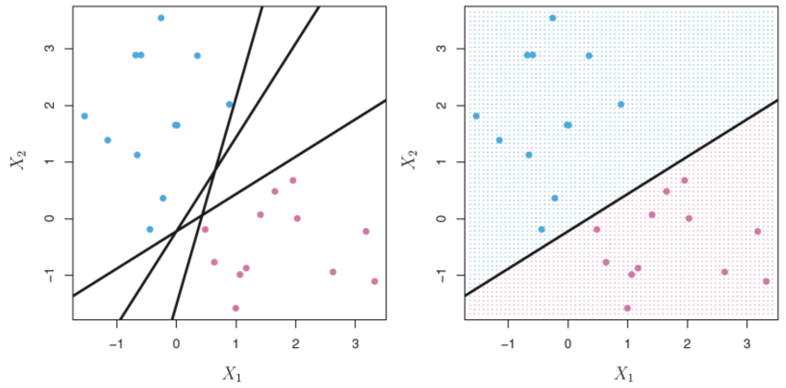
- *Normal distribution*
- *Linear margins*

Trees are not restricted by these assumptions, ***but*** *trees often struggle with non-discrete spaces, and are less robust with dynamic data (which is a big issue in transaction land)*



Chapter 9: Support Vector Machines

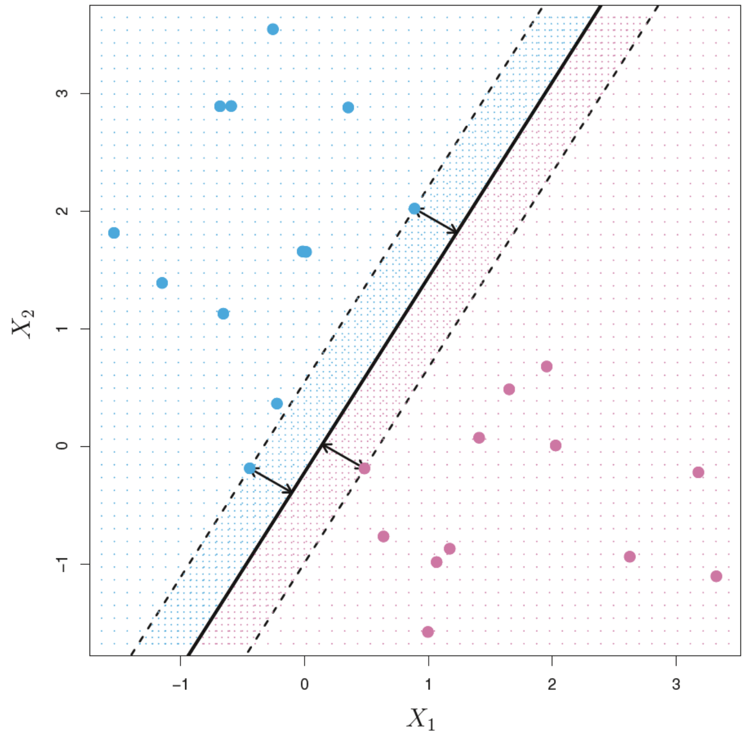
The Separating Hyperplane



The blue and purple grid indicates the decision rule *(recall the dr in LDA)* made by a classifier based on this separating hyperplane.

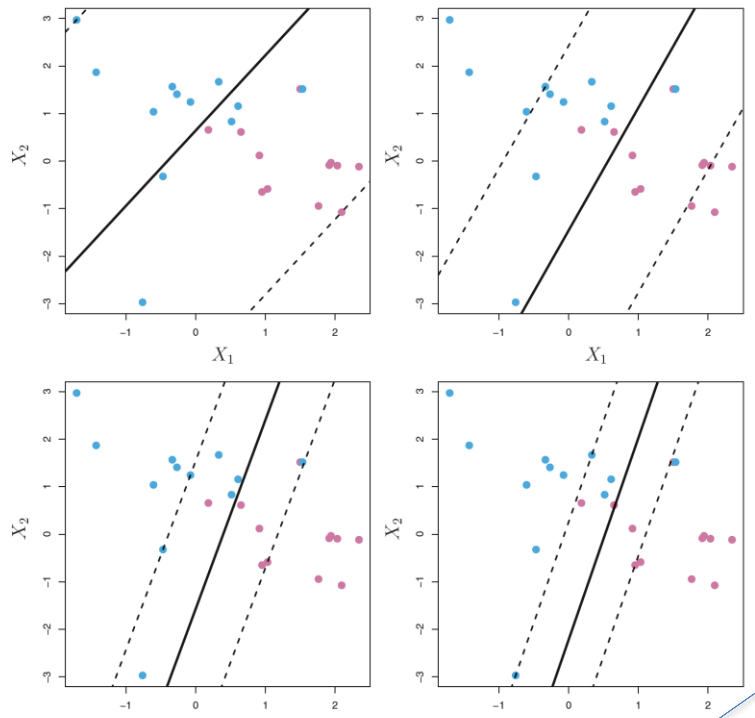
The right-hand panel shows an example of such a classifier. That is, we classify the test observation *x* based on the sign of $f(x) = \beta_0+\beta_1 x*1+\beta_2 x*2+\ldots$ If $f(x)$ is positive, then we assign the test observation to class 1, and if $f(x)$ is negative, then we assign it to class −1.

We can also make use of the *magnitude* of $f(x)$. If $f(x)$ is far from zero, then this means that *x* lies far from the hyperplane, and so we can be confident about our class assignment for *x*. On the other hand, if $f(x)$ is close to zero, then *x* is located near the hyperplane, and so we are less certain about the class assignment for *x*. Not surprisingly, and as we see in Figure 9.2, a classifier that is based on a separating hyperplane leads to a linear decision boundary.



Maximal margin hyperplane (also known as the *optimal separating hyperplane*), is the separating hyperplane that is farthest from the training observations. That is, we can compute the perpendicular ***(orthogonal)*** distance from each training observation to a given separating hyperplane; the smallest such distance is the minimal distance from the observations to the hyperplane, and is known as the ***margin***.

(and the points on the margins are called support vectors)



$$\underset{\beta_0,\beta_1,\ldots,\beta_p,\epsilon_1,\ldots,\epsilon_n,M}{\text{maximize}} \quad M \tag{9.12}$$

$$\text{subject to } \sum_{j=1}^{p}\beta_j^2 = 1, \tag{9.13}$$

$$y_i(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2}+\ldots+\beta_p x_{ip}) \geq M(1-\epsilon_i), \tag{9.14}$$

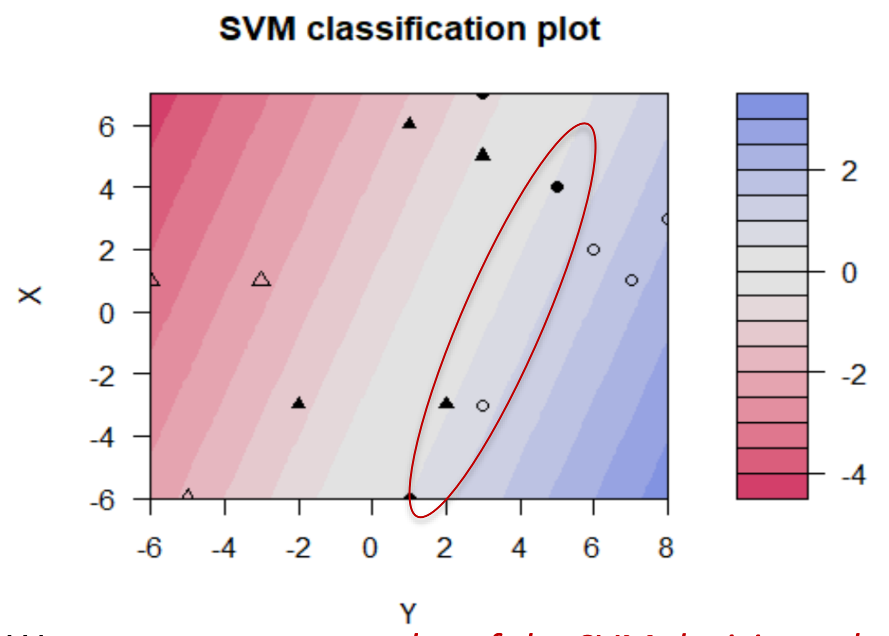
$$\epsilon_i \geq 0, \quad \sum_{i=1}^{n}\epsilon_i \leq C, \tag{9.15}$$

Soft margin classifier. Rather than seeking the largest possible margin so that every observation is not only on the correct side of the hyperplane but also on the correct side of the margin, we instead allow some observations to be on the incorrect side of the margin, or even the incorrect side of the hyperplane *(The margin is soft because it can be violated by some of the training observations.)*

C is a nonnegative tuning parameter. *M* is the width of the margin; we seek to make M as large as possible.

We can create *slack variables* that allow individual observations to be on the wrong side of the margin or the hyperplane

These are the equations from the book – we're going to revise these a bit



We can see a *contour plot of the SVM decision rules (compare this to the decision rules of LDA ☺)* Note how different *weight* values are grouped in the same contour:

$$f(x) = w^\top \Phi(x)$$

Just testing a crude result *(% correct predictions)*:

```
result <- data.frame(predict(t1, mTrain))
result <- cbind(result, tst$O)
result$diff <- result[,1]-result[,2]
CrudeResult <-  round(nrow(result[result$diff == 0, ])/nrow(result),2)
CrudeResult
```

```
> CrudeResult
[1] 0.86
```

This looks OK, but we're just running the training dataset through again – not what we want to see.

Also, notice that the contours are lines…

ksvm(mTst, yTst, type="**C-svc**", **C=1000**, kernel=vanilladot(),scaled=c())

9 settings for different types of classification and regression Depending on whether y is a factor or not, the default setting for type is C-svc or eps-svr, respectively, but can be overwritten by setting an explicit value.

The cost parameter penalizes large residuals. So a larger cost will result in a more flexible model with fewer misclassifications. In effect the cost parameter allows you to adjust the bias/variance trade-off. **The greater the cost parameter, the more variance in the model** and the less bias. Note how this is the opposite of regularization which penalizes large coefficients, resulting in higher bias and lower variance. Here we penalize the residuals resulting in higher variance and lower bias.

So, this is set for a high error tolerance, forcing the algorithm to allow slack variables on the wrong side of the margin or the hyperplane. In this simple case, it doesn't make any difference…

$$\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C,$$

9 class kernels OOB, but you can write a custom kernel

A logical vector indicating the variables to be scaled. If scaled is of length 1, the value is recycled as many times as needed and all non-binary variables are scaled. Per default, data are scaled internally (both x and y variables) to zero mean and unit variance. The center and scale values are returned and used for later predictions.

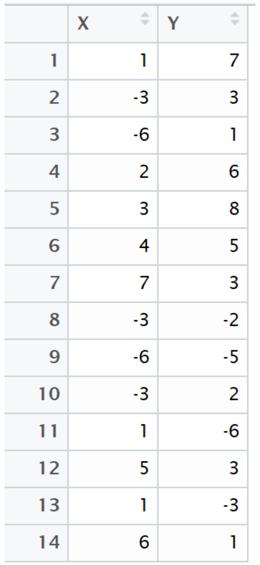
scaled=c(F,T,T,T) would scale variables 2:4 in the dataset.

Now we're going to focus on the second part of the model – the kernel. A kernel function is used to transform *(~like we did with log transforms and scaling)* the data to a form that the SVM can process. In this case, we used a linear transformation:

$$f(x) = w^\top \Phi(x)$$

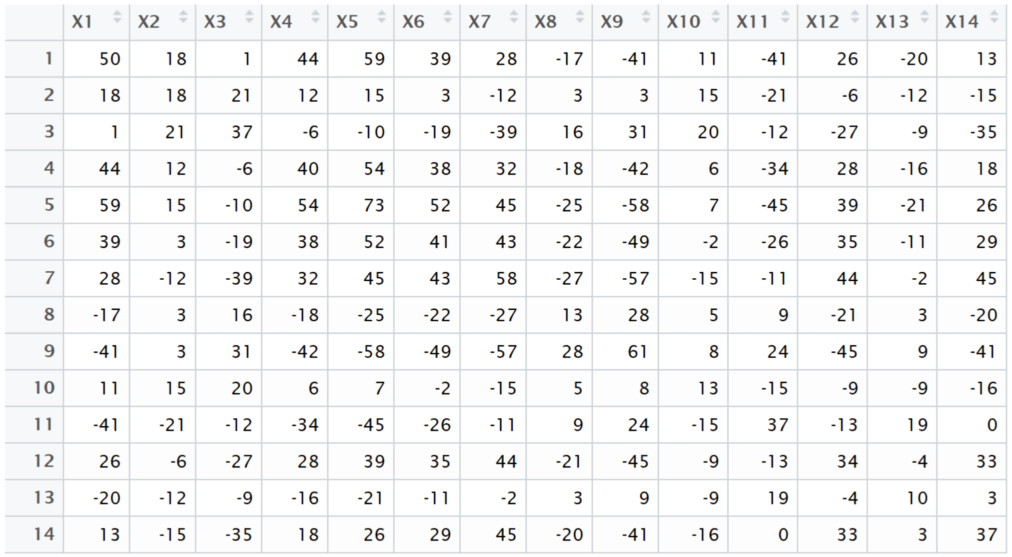
btw, we also use phi to signify a distribution later, so pay attention to context

We can see what the model did by generating a linear kernel matrix *(this uses dot products)*:



	X	Y
1	1	7
2	-3	3
3	-6	1
4	2	6
5	3	8
6	4	5
7	7	3
8	-3	-2
9	-6	-5
10	-3	2
11	1	-6
12	5	3
13	1	-3
14	6	1

```
kl <- kernelMatrix(vanilladot(),
mTrain)
dim(kl)
kl
dfKl <- data.frame(kl)
```

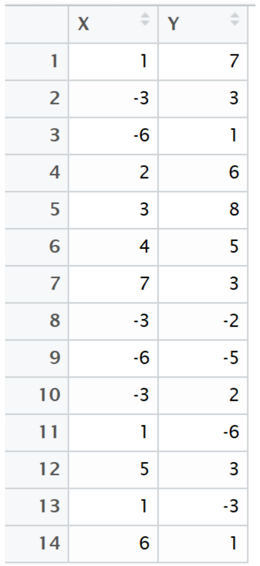


	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14
1	50	18	1	44	59	39	28	-17	-41	11	-41	26	-20	13
2	18	18	21	12	15	3	-12	3	3	15	-21	-6	-12	-15
3	1	21	37	-6	-10	-19	-39	16	31	20	-12	-27	-9	-35
4	44	12	-6	40	54	38	32	-18	-42	6	-34	28	-16	18
5	59	15	-10	54	73	52	45	-25	-58	7	-45	39	-21	26
6	39	3	-19	38	52	41	43	-22	-49	-2	-26	35	-11	29
7	28	-12	-39	32	45	43	58	-27	-57	-15	-11	44	-2	45
8	-17	3	16	-18	-25	-22	-27	13	28	5	9	-21	3	-20
9	-41	3	31	-42	-58	-49	-57	28	61	8	24	-45	9	-41
10	11	15	20	6	7	-2	-15	5	8	13	-15	-9	-9	-16
11	-41	-21	-12	-34	-45	-26	-11	9	24	-15	37	-13	19	0
12	26	-6	-27	28	39	35	44	-21	-45	-9	-13	34	-4	33
13	-20	-12	-9	-16	-21	-11	-2	3	9	-9	19	-4	10	3
14	13	-15	-35	18	26	29	45	-20	-41	-16	0	33	3	37

Now we’re going to use a different kernel – a radial basis function *(which is a type of gaussian function)*:

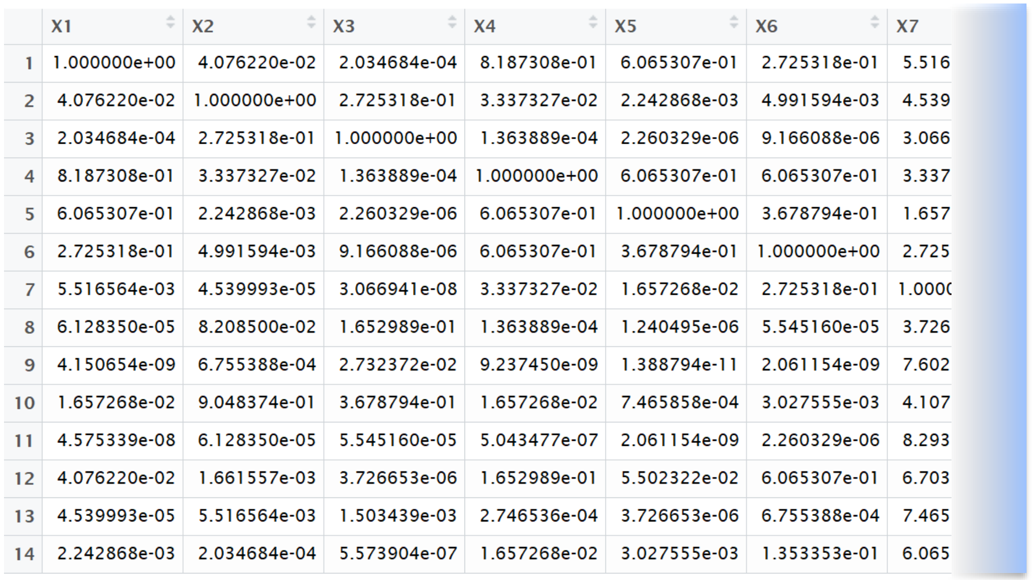
$$f(x) = w^\top \Phi(x)$$

Notice how the transformation is yields completely different values *(also notice that the dimensions are not n x n)*



	X	Y
1	1	7
2	-3	3
3	-6	1
4	2	6
5	3	8
6	4	5
7	7	3
8	-3	-2
9	-6	-5
10	-3	2
11	1	-6
12	5	3
13	1	-3
14	6	1

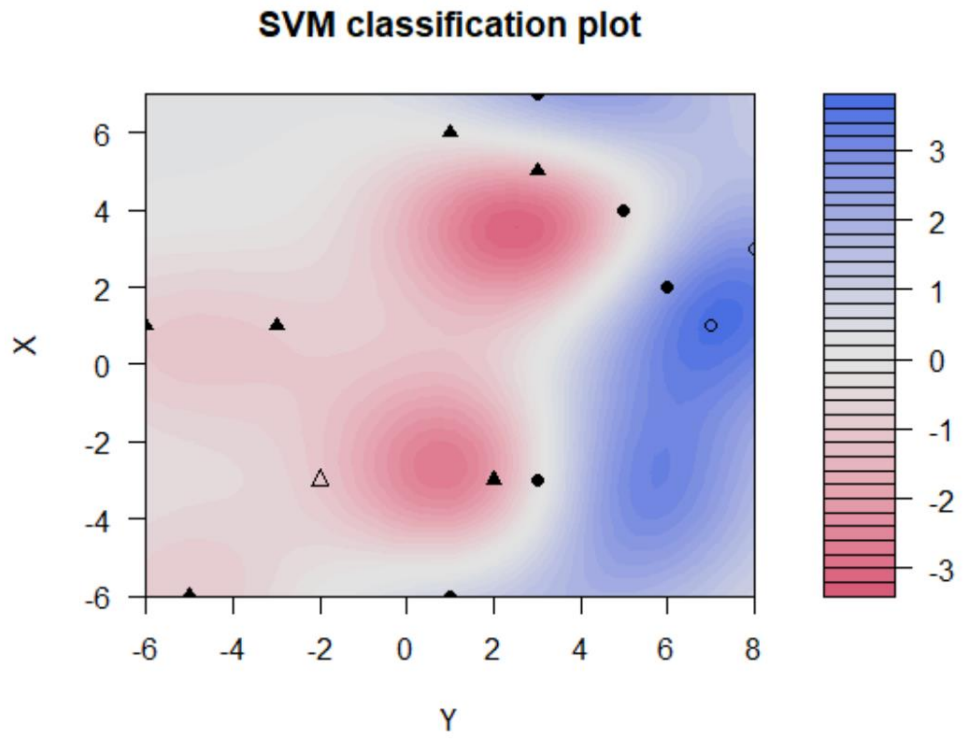
```
rbf <- function(x,y) exp(-0.1 *
sum((x-y)^2))
class(rbf) <- "kernel"
mTst <- as.matrix(tst[,1:2])
yTst <- as.matrix(tst[,3])

k2 <- kernelMatrix(rbf, mTst)

dim(k2)
dfK2 <- data.frame(k2)
```



	X1	X2	X3	X4	X5	X6	X7
1	1.000000e+00	4.076220e-02	2.034684e-04	8.187308e-01	6.065307e-01	2.725318e-01	5.516
2	4.076220e-02	1.000000e+00	2.725318e-01	3.337327e-02	2.242868e-03	4.991594e-03	4.539
3	2.034684e-04	2.725318e-01	1.000000e+00	1.363889e-04	2.260329e-06	9.166088e-06	3.066
4	8.187308e-01	3.337327e-02	1.363889e-04	1.000000e+00	6.065307e-01	6.065307e-01	3.337
5	6.065307e-01	2.242868e-03	2.260329e-06	6.065307e-01	1.000000e+00	3.678794e-01	1.657
6	2.725318e-01	4.991594e-03	9.166088e-06	6.065307e-01	3.678794e-01	1.000000e+00	2.725
7	5.516564e-03	4.539993e-05	3.066941e-08	3.337327e-02	1.657268e-02	2.725318e-01	1.000
8	6.128350e-05	8.208500e-02	1.652989e-01	1.363889e-04	1.240495e-06	5.545160e-05	3.726
9	4.150654e-09	6.755388e-04	2.732372e-02	9.237450e-09	1.388794e-11	2.061154e-09	7.602
10	1.657268e-02	9.048374e-01	3.678794e-01	1.657268e-02	7.465858e-04	3.027555e-03	4.107
11	4.575339e-08	6.128350e-05	5.545160e-05	5.043477e-07	2.061154e-09	2.260329e-06	8.293
12	4.076220e-02	1.661557e-03	3.726653e-06	1.652989e-01	5.502322e-02	6.065307e-01	6.703
13	4.539993e-05	5.516564e-03	1.503439e-03	2.746536e-04	3.726653e-06	6.755388e-04	7.465
14	2.242868e-03	2.034684e-04	5.573904e-07	1.657268e-02	3.027555e-03	1.353353e-01	6.065

So now we have a completely different plot. Notice how the vector values line up with contours.

```
t2 <- ksvm(mTst, yTst, type="C-svc", C=100, kernel=rbf, scale=c())
```



```
result2 <- data.frame(predict(t2, mTrain))
result2 <- cbind(result2, tst$O)
result2$diff <- result2[,1]-result2[,2]
CrudeResult2 <-  round(nrow(result[result2$diff == 0, ])/nrow(result2),2)
CrudeResult2
```

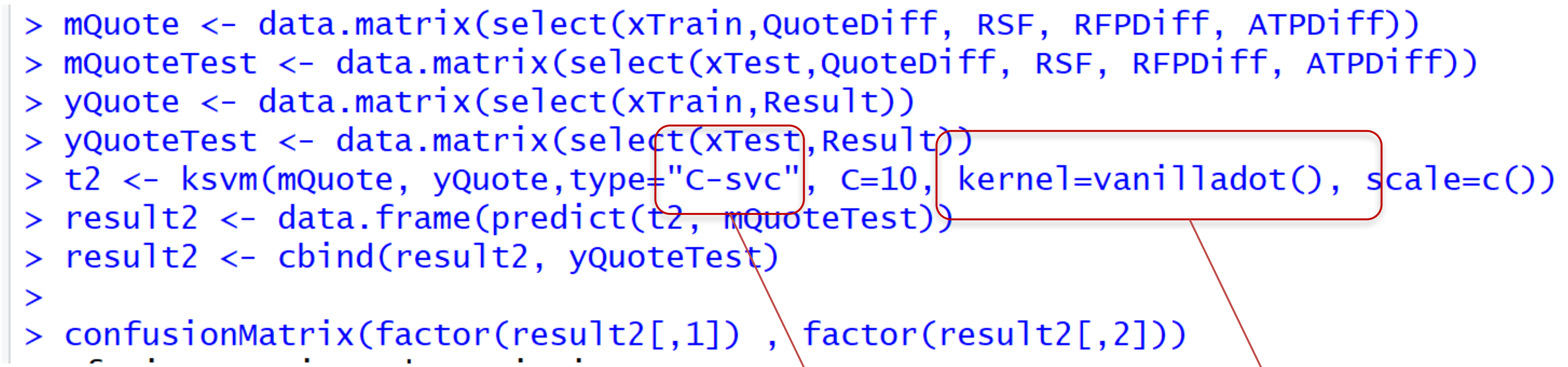
```
> CrudeResult2
[1] 1
```

And the results are 100% accurate *(although this is testing against a training set – kinda stupid, but using this as an illustration)*

BAUER COLLEGE OF BUSINESS
UNIVERSITY of HOUSTON

SVM Exercise

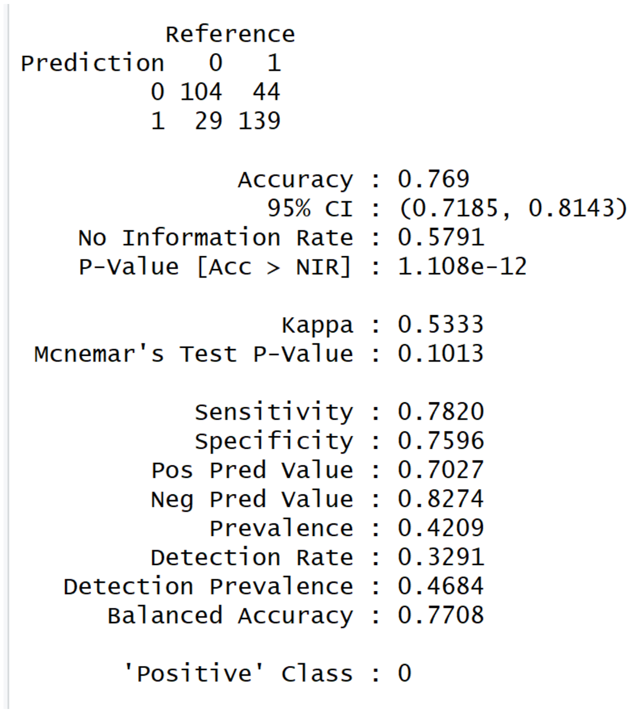
Dot Product Kernel



```
> mQuote <- data.matrix(select(xTrain,QuoteDiff, RSF, RFPDiff, ATPDiff))
> mQuoteTest <- data.matrix(select(xTest,QuoteDiff, RSF, RFPDiff, ATPDiff))
> yQuote <- data.matrix(select(xTrain,Result))
> yQuoteTest <- data.matrix(select(xTest,Result))
> t2 <- ksvm(mQuote, yQuote,type="C-svc", C=10, kernel=vanilladot(), scale=c())
> result2 <- data.frame(predict(t2, mQuoteTest))
> result2 <- cbind(result2, yQuoteTest)
>
> confusionMatrix(factor(result2[,1]) , factor(result2[,2]))
```

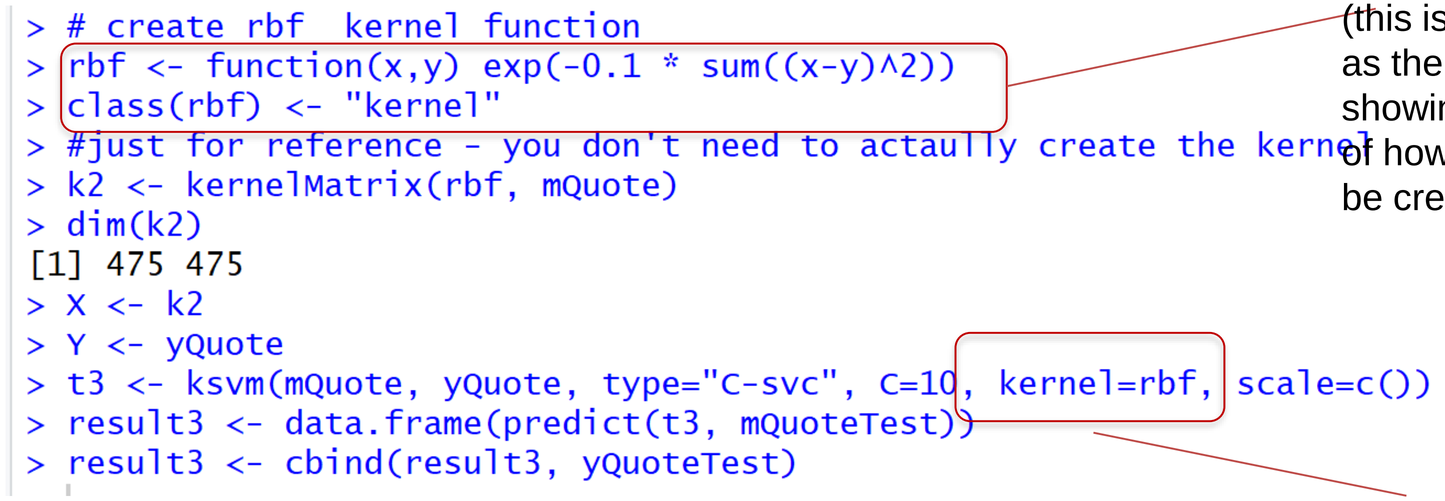
Classification (see manual)

dot product kernel



```
          Reference
Prediction   0   1
         0 104  44
         1  29 139

               Accuracy : 0.769
                 95% CI : (0.7185, 0.8143)
    No Information Rate : 0.5791
    P-Value [Acc > NIR] : 1.108e-12

                  Kappa : 0.5333
 Mcnemar's Test P-Value : 0.1013

            Sensitivity : 0.7820
            Specificity : 0.7596
         Pos Pred Value : 0.7027
         Neg Pred Value : 0.8274
             Prevalence : 0.4209
         Detection Rate : 0.3291
   Detection Prevalence : 0.4684
      Balanced Accuracy : 0.7708

       'Positive' Class : 0
```

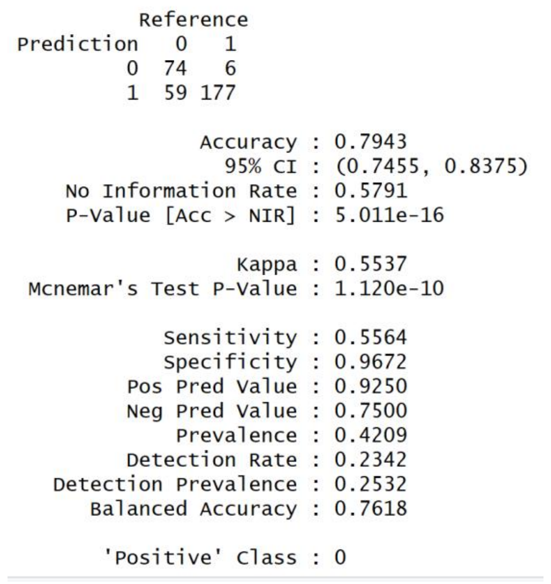
Radial Basis Function Kernel *(custom)*



```
> # create rbf  kernel function
> rbf <- function(x,y) exp(-0.1 * sum((x-y)^2))
> class(rbf) <- "kernel"
> #just for reference - you don't need to actually create the kernel
> k2 <- kernelMatrix(rbf, mQuote)
> dim(k2)
[1] 475 475
> X <- k2
> Y <- yQuote
> t3 <- ksvm(mQuote, yQuote, type="C-svc", C=10, kernel=rbf, scale=c())
> result3 <- data.frame(predict(t3, mQuoteTest))
> result3 <- cbind(result3, yQuoteTest)
```

Creating custom kernel (this is basically the same as the ootb kernel – just showing so you get an idea of how custom kernels can be created

rbf kernel

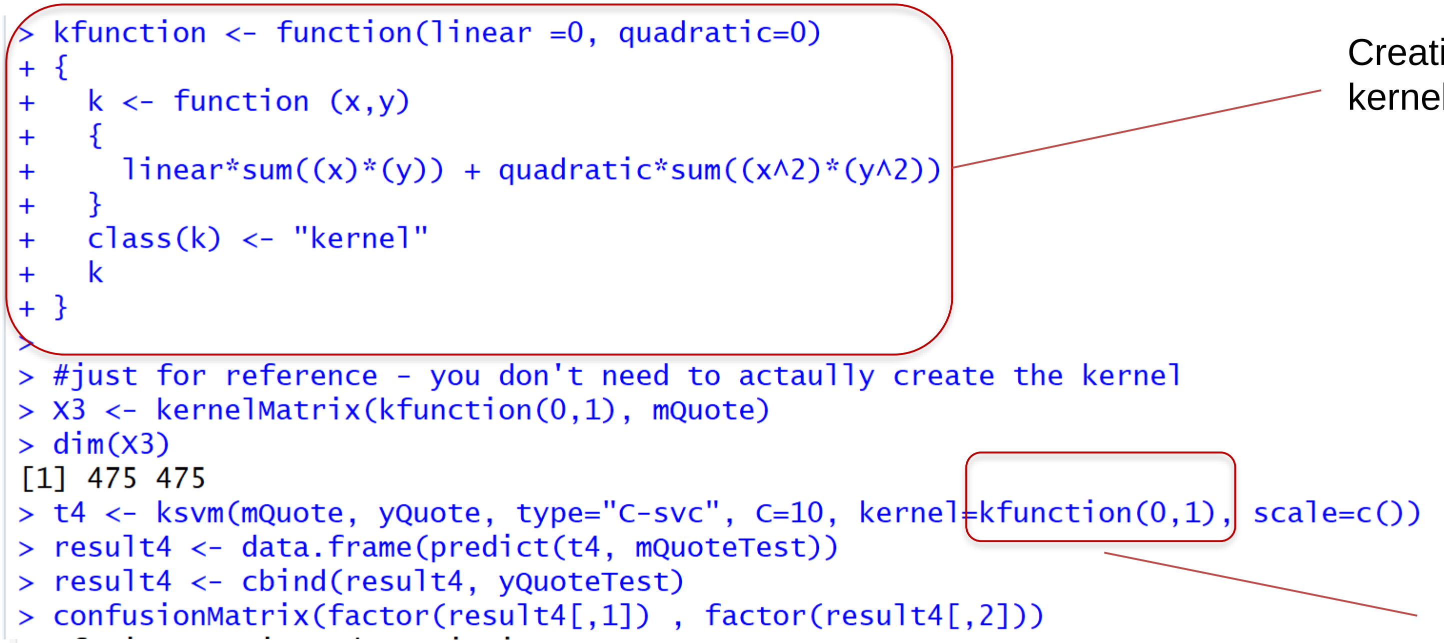


```
          Reference
Prediction   0   1
         0  74   6
         1  59 177
                                          
               Accuracy : 0.7943          
                 95% CI : (0.7455, 0.8375)
    No Information Rate : 0.5791          
    P-Value [Acc > NIR] : 5.011e-16       
                                          
                  Kappa : 0.5537          
 Mcnemar's Test P-Value : 1.120e-10       
                                          
            Sensitivity : 0.5564          
            Specificity : 0.9672          
         Pos Pred Value : 0.9250          
         Neg Pred Value : 0.7500          
             Prevalence : 0.4209          
         Detection Rate : 0.2342          
   Detection Prevalence : 0.2532          
      Balanced Accuracy : 0.7618          
                                          
       'Positive' Class : 0               
```

Out of the Box (OOTB) kernels available with ksvm:

polydot	Polynomial kernel function
vanilladot	Linear kernel function
tanhdot	Hyperbolic tangent kernel function
laplacedot	Laplacian kernel function
besseldot	Bessel kernel function
anovadot	ANOVA RBF kernel function
splinedot	Spline kernel
stringdot	String kernel
rbfdot	Radial Basis kernel function "Gaussian“

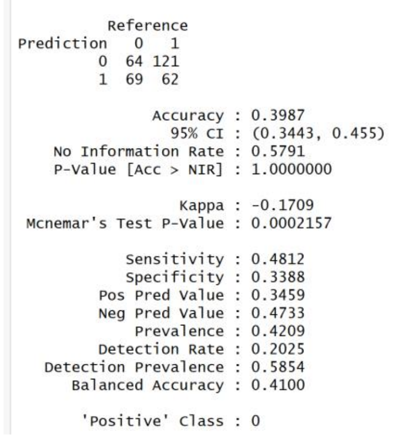
Quadratic Kernel *(custom)*



```
> kfunction <- function(linear =0, quadratic=0)
+ {
+   k <- function (x,y)
+   {
+     linear*sum((x)*(y)) + quadratic*sum((x^2)*(y^2))
+   }
+   class(k) <- "kernel"
+   k
+ }
>
> #just for reference - you don't need to actually create the kernel
> X3 <- kernelMatrix(kfunction(0,1), mQuote)
> dim(X3)
[1] 475 475
> t4 <- ksvm(mQuote, yQuote, type="C-svc", C=10, kernel=kfunction(0,1), scale=c())
> result4 <- data.frame(predict(t4, mQuoteTest))
> result4 <- cbind(result4, yQuoteTest)
> confusionMatrix(factor(result4[,1]) , factor(result4[,2]))
```

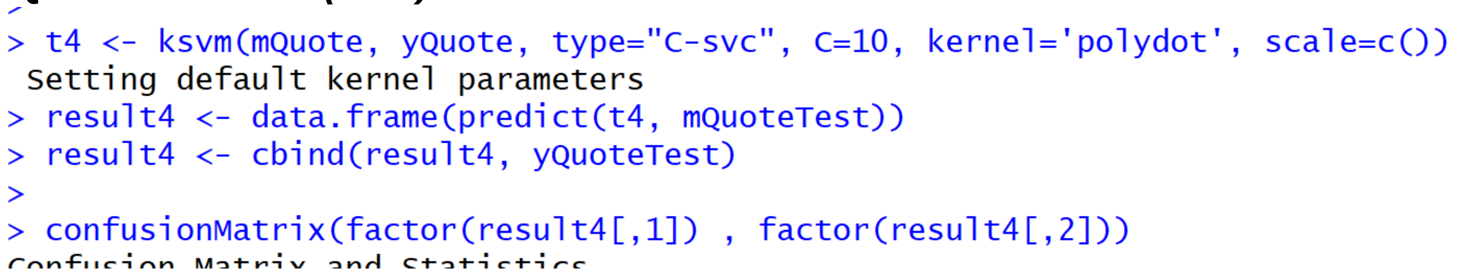
Creating custom quadratic kernel

rbf kernel

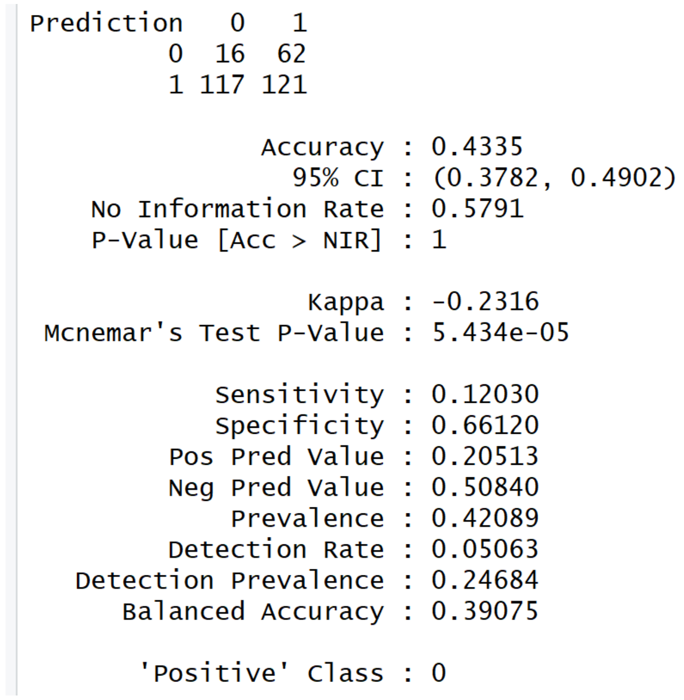


```
          Reference
Prediction   0   1
         0  64 121
         1  69  62

               Accuracy : 0.3987
                 95% CI : (0.3443, 0.455)
    No Information Rate : 0.5791
    P-Value [Acc > NIR] : 1.0000000

                  Kappa : -0.1709
 Mcnemar's Test P-Value : 0.0002157

            Sensitivity : 0.4812
            Specificity : 0.3388
         Pos Pred Value : 0.3459
         Neg Pred Value : 0.4733
             Prevalence : 0.4209
         Detection Rate : 0.2025
   Detection Prevalence : 0.5854
      Balanced Accuracy : 0.4100

       'Positive' Class : 0
```

Quadratic Kernel *(ootb)*



```
> t4 <- ksvm(mQuote, yQuote, type="C-svc", C=10, kernel='polydot', scale=c())
 Setting default kernel parameters
> result4 <- data.frame(predict(t4, mQuoteTest))
> result4 <- cbind(result4, yQuoteTest)
>
> confusionMatrix(factor(result4[,1]) , factor(result4[,2]))
Confusion Matrix and Statistics
```



```
Prediction   0   1
         0  16  62
         1 117 121

               Accuracy : 0.4335
                 95% CI : (0.3782, 0.4902)
    No Information Rate : 0.5791
    P-Value [Acc > NIR] : 1

                  Kappa : -0.2316
 Mcnemar's Test P-Value : 5.434e-05

            Sensitivity : 0.12030
            Specificity : 0.66120
         Pos Pred Value : 0.20513
         Neg Pred Value : 0.50840
             Prevalence : 0.42089
         Detection Rate : 0.05063
   Detection Prevalence : 0.24684
      Balanced Accuracy : 0.39075

       'Positive' Class : 0
```

So, we've used 4 different kernels with 4 very different results. Remember, we're not changing the model, we're transforming the data

Now,, we'll consider basic tuning:

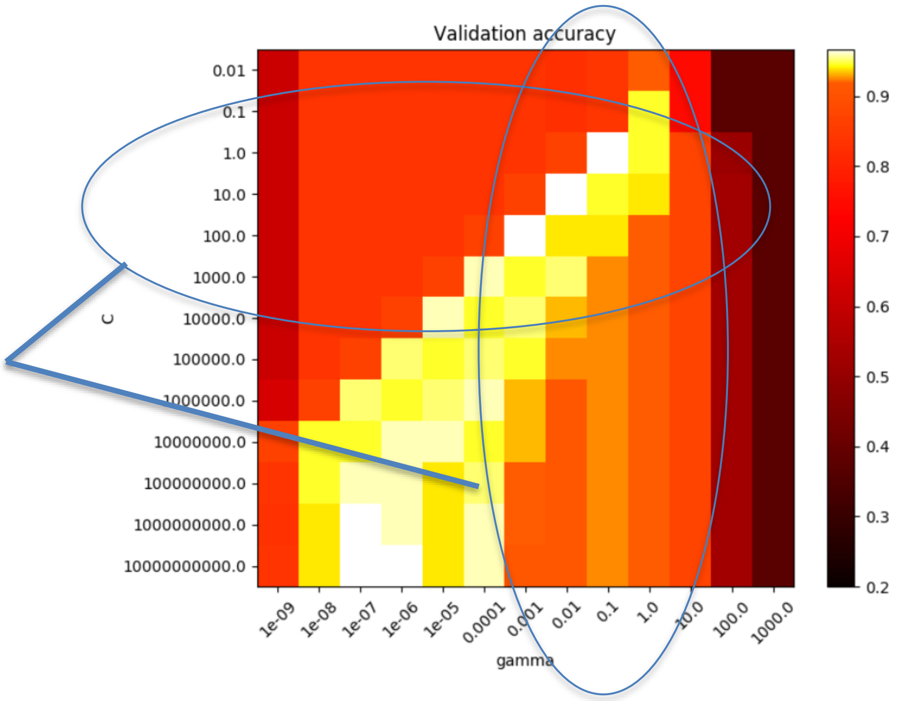
Model Tuning

The **gamma** parameter is used with an RBF kernel to define how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. The gamma parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors.

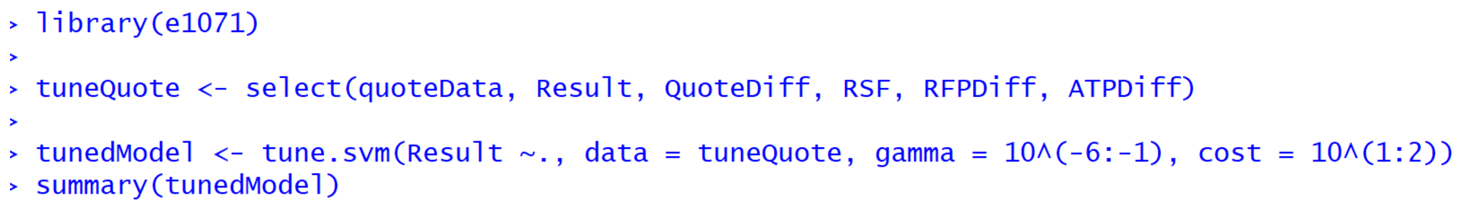
The **C** parameter trades off misclassification of training examples against simplicity of the decision surface. A low C makes the decision surface smooth, while a high C aims at classifying all training examples correctly by giving the model freedom to select more samples as support vectors.

This is a grid search result that is fairly typical of SVMs.

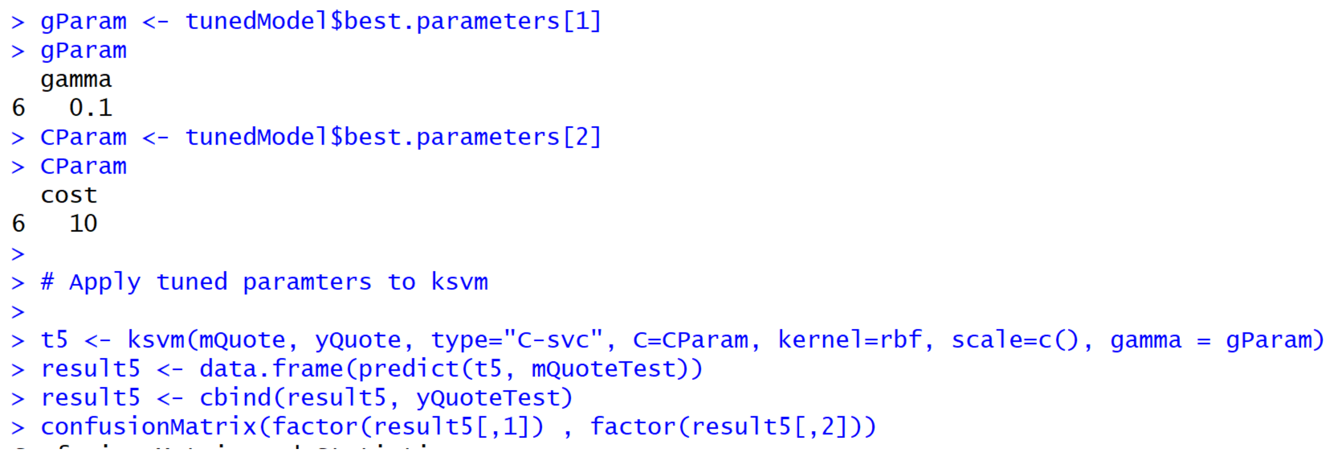
I find the "sweet spot' generally falls in C = 1-1000, and gamma = .01 - 1



Your goal with validation is to mininmize the training error. So, you're are solving an optimization problem which is usually be something like minimize training error plus a regularization term.
It can be very expensive to solve optimization for each combination of values of parameters. But in simple cases, we can use **grid search**: pick a bunch of values and for each pair of values, evaluate the validation error function and pick the pair that gives the minimum value of the validation error function.
You can also do a random grid search, which has been shown to be comparable to

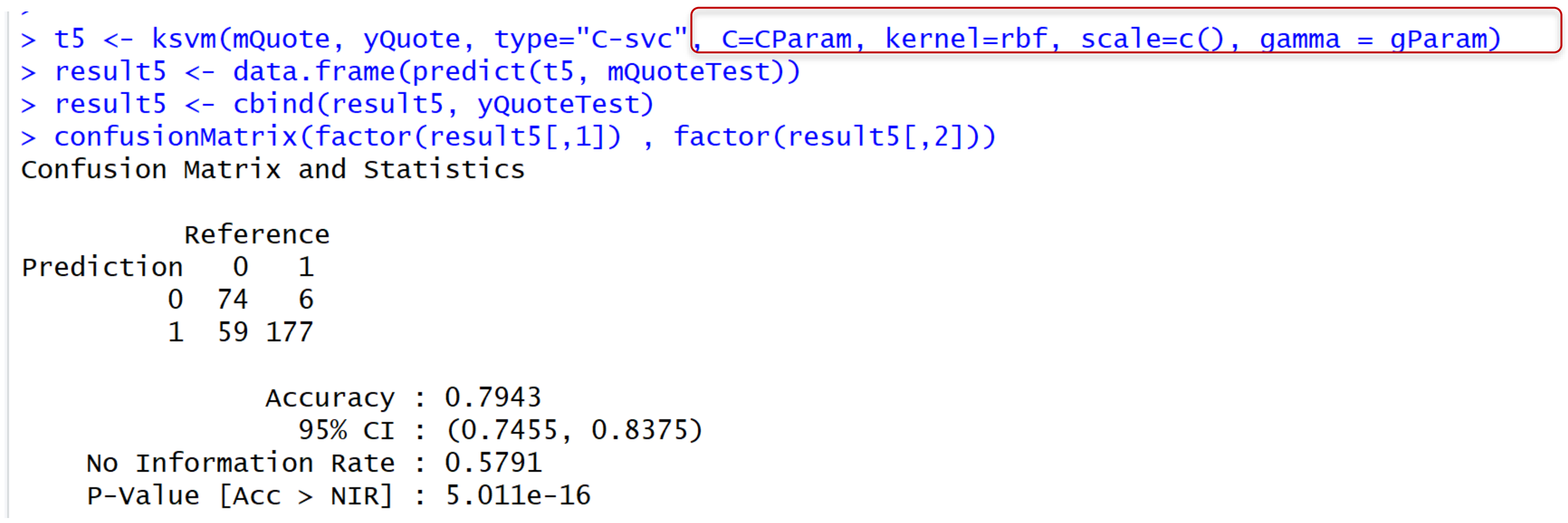


```
> library(e1071)
>
> tuneQuote <- select(quoteData, Result, QuoteDiff, RSF, RFPDiff, ATPDiff)
>
> tunedModel <- tune.svm(Result ~., data = tuneQuote, gamma = 10^(-6:-1), cost = 10^(1:2))
> summary(tunedModel)
```



```
> gParam <- tunedModel$best.parameters[1]
> gParam
  gamma
6   0.1
> CParam <- tunedModel$best.parameters[2]
> CParam
  cost
6   10
>
> # Apply tuned paramters to ksvm
>
> t5 <- ksvm(mQuote, yQuote, type="C-svc", C=CParam, kernel=rbf, scale=c(), gamma = gParam)
> result5 <- data.frame(predict(t5, mQuoteTest))
> result5 <- cbind(result5, yQuoteTest)
> confusionMatrix(factor(result5[,1]) , factor(result5[,2]))
```

Then we apply the tuned parameters to the model

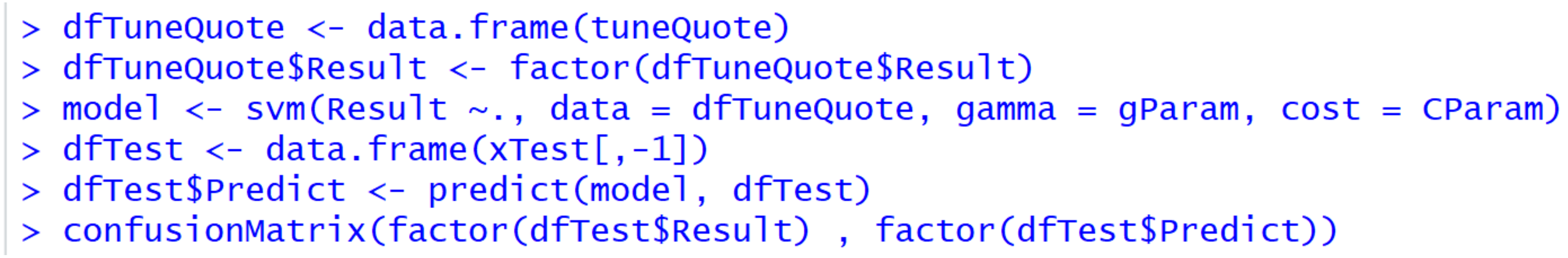


```
> t5 <- ksvm(mQuote, yQuote, type="C-svc", C=CParam, kernel=rbf, scale=c(), gamma = gParam)
> result5 <- data.frame(predict(t5, mQuoteTest))
> result5 <- cbind(result5, yQuoteTest)
> confusionMatrix(factor(result5[,1]) , factor(result5[,2]))
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0  74   6
         1  59 177

               Accuracy : 0.7943
                 95% CI : (0.7455, 0.8375)
    No Information Rate : 0.5791
    P-Value [Acc > NIR] : 5.011e-16
```

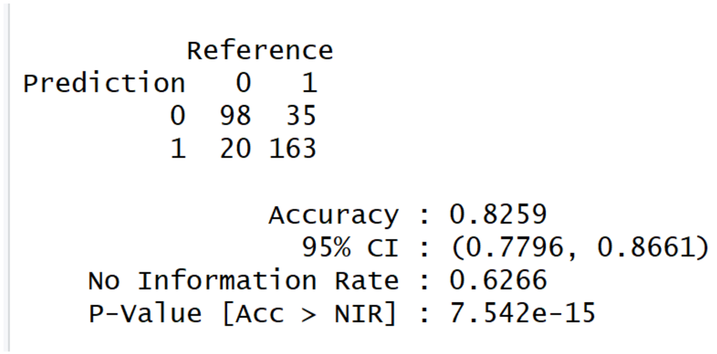
Here, the results didn't improve significantly, but this is crude tuning without resampling

e1071 package svm

If you don’t need to control the kernels, the e1071 package is usually easier to work with – the default kernel is rbf. It also works better with dataframes and character data.



```
> dfTuneQuote <- data.frame(tuneQuote)
> dfTuneQuote$Result <- factor(dfTuneQuote$Result)
> model <- svm(Result ~., data = dfTuneQuote, gamma = gParam, cost = CParam)
> dfTest <- data.frame(xTest[,-1])
> dfTest$Predict <- predict(model, dfTest)
> confusionMatrix(factor(dfTest$Result) , factor(dfTest$Predict))
```



```
          Reference
Prediction   0   1
         0  98  35
         1  20 163

               Accuracy : 0.8259
                 95% CI : (0.7796, 0.8661)
    No Information Rate : 0.6266
    P-Value [Acc > NIR] : 7.542e-15
```

BAUER
COLLEGE OF BUSINESS
UNIVERSITY of HOUSTON

Classification Wrap-up

Classification process

- Understand data relationships
- Select features
- Select metrics
- Create model
- Evaluate model
- Improve model
- Cross Validate model

Steps to improve models

- Start by understanding errors
- Filter or transform the data
- Better feature engineering
- Improve feature selection
- Use a different type of model
- Choice of model parameters

Start by understanding errors
Filter or transform the data
Better feature engineering
Improve feature selection
Use a different type of model
Choice of model parameters