

# Classification – Part 2

- Classification

- Overview

ISL Chapter 4

- Methods

- Logistic Regression
    - Linear Discriminant Analysis
    - Naïve Bayes
    - Point Bayes

- Decision Trees

ISL Chapter 8

- Overview

- Support Vector Machines

There are hundreds of classification algorithms available (*regression too*). But most of them fall into the broad categories outlined here (*and in ch 4 and 8 of ISL*).

Which one should you use? What's the best car to buy?

It depends on what you need. Model selection is art and science. The science piece is understanding how the algorithm works, and how that fits your data and analysis, how to “tune” the algorithm parameters and compare results.

The tradeoffs: Bias and Variance, and Prediction Accuracy and Interpretability, apply here. As does intuition based on case study – as we look into these algorithms, note how they treat continuous vs discrete data (or can the data be transformed?) – that will be a clue that can guide you...

<http://topepo.github.io/caret/available-models.html>

Show  entries

Search:

Model	method	Value	Type	Libraries	Tuning Parameters
AdaBoost Classification Trees	adaboost		Classification	fastAdaboost	nIter, method
AdaBoost.M1	AdaBoost.M1		Classification	adabag, plyr	mfinal, maxdepth, coeflearn
Adaptive Mixture Discriminant Analysis	amdai		Classification	adaptDA	model
Adjacent Categories Probability Model for Ordinal Data	vglmAdjCat		Classification	VGAM	parallel, link
Bagged AdaBoost	AdaBag		Classification	adabag, plyr	mfinal, maxdepth
Bagged FDA using gCV Pruning	bagFDAGCV		Classification	earth	degree
Bagged Flexible Discriminant Analysis	bagFDA		Classification	earth, mda	degree, nprune
Binary Discriminant Analysis	binda		Classification	binda	lambda.freqs

# Homework Review

default	balGrp	student	n
No	1	9667 96.7%	1261 13% No 1066 85%
			Yes 195 15%
		2	3145 33% No 2408 77%
			Yes 737 23%
		3	3569 37% No 2410 68%
			Yes 1159 32%
		4	1514 16% No 904 60%
			Yes 610 40%
		5	174 2% No 61 35%
			Yes 113 65%
		6	4 0% No 1 25%
			Yes 3 75%
Yes	1	333 3.3%	0 0% No 0 0%
			Yes 0 0%
		2	2 1% No 2 100%
			Yes 0 0%
		3	24 7% No 19 79%
			Yes 5 21%
		4	130 39% No 91 70%
			Yes 39 30%
		5	159 48% No 90 57%
			Yes 69 43%
		6	18 5% No 4 22%
			Yes 14 78%

$$\mathbb{P}(\text{default} = \text{"Yes"} \mid \text{balGrp} = 3, \text{student} = \text{"Yes"}) = \frac{.033 * .072 * .21}{(.37 * .96 * .32) + (.033 * .07 * .21)} = .004$$

So, there's a .4% chance of default, given the balance is between (750,1.25e+03) and a student

$$\frac{0.05\%}{F72 * C68} = 0.43\%$$

Notice how P<sub>level 1, level 2</sub> becomes the prior for level 3

Also notice that the prior probability is

$$P(A \cap B) = P(B) * P(A \mid B) =$$

$$.03 * .07 = .002$$

Hypothesis	Prior	Likelihood	Numerator	Posterior
Default = No	0.357	0.325	0.1159	0.996
Default = Yes	0.002	0.208	0.0005	0.004
			0.1164	1.000

# Homework Extension

Lets assume that dfDefault is month 1, and that month 2 is a little different. The change is an increase in student defaults in balGrp 3 – from 5 to 24 (maybe the recent grad job market stalls out).

So, if we go through the same process, then we'll get an increase in posterior probability from .4% to 2%.

But is that the best projection for month 3? Should we start getting tougher on student credit? Is month 2 a trend or a deviation (*randomness*)?

Hypothesis	Prior	Likelihood	Bayes Numerator	Posterior
Default = No	0.356	0.325	0.116	0.980
Default = Yes	0.004	0.558	0.002	<b>0.020</b>
			<b>0.118</b>	<b>1.000</b>

```
dfDefault1 = dfDefault %>%
  dplyr::select(default, student, balance, balGrp, income) %>%
  mutate (month = 1)

dfDefault2 = dfDefault1 %>%
  filter(default == "Yes", balGrp == 3, student == "Yes") %>%
  sample_n(20, replace = T) %>%
  mutate (balance = balance + rnorm(balance, 0, 30)) %>%
  filter(balance > 750, balance <1250) %>%
  bind_rows(dfDefault1) %>%
  mutate(month = 2)

dfDefaultNew = bind_rows(dfDefault1, dfDefault2)

dfDefaultNew %>%
  filter(student == "Yes", balGrp == 3, default == "Yes", month == 2) %>%
  summarise(Tot = n())
Tot
24

Bayes2N = dfDefault2 %>% filter(student == "Yes", balGrp == 3, default == "Yes")
%>%
  summarise(Tot = n())
Bayes2D = dfDefault2 %>% filter(student == "Yes", balGrp == 3) %>%
  summarise(Tot = n())
Bayes2N/Bayes2D
Tot
0.0202874
```

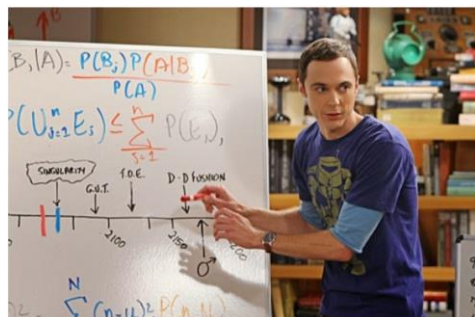
We could also analyze ALL (*month1 + month 2*) the data (*in which case, the posterior becomes 1.2%*)

A more conservative (*and Bayesian*) approach would be to set the priors based on Month 1 posteriors. This will adjust the posterior with more weight of prior probability – based on data. This is essentially a compromise between prior probability and current data. It’s also more practical because we don’t always have every period we need, and even if we do, we often don’t have time to sample ad analyze (*regardless of all the wiz-bang technology*). Also, keep in mind that we’re generally projecting imaginary data – regardless of whether we’re forecasting, planning or validating.

Hypothesis	Prior	Likelihood	Bayes Numerator	Posterior
Default = No	0.996	0.325	0.323	0.993
Default = Yes	0.004	0.558	0.002	<b>0.007</b>
			<b>0.326</b>	<b>1.000</b>

As we move into Bayesian analysis is the 2<sup>nd</sup> half, you’ll see that we can adjust and weigh priors based on evidence and credibility, also including expert knowledge and judgment.

# Linear Discriminant Analysis



*prior*  
(hypothesis)

*likelihood*  
(sample)

$$\mathbb{P}(Y | X) = \frac{\mathbb{P}(Y) \mathbb{P}(X | Y)}{\mathbb{P}(X)}$$

*data*  
(population)

```
> BayesN = dfDefault %>% filter(student == "Yes", balGrp == 3, default == "Yes") %>%
```

```
+ summarise(Tot = n())
```

```
>
```

```
> BayesD = dfDefault %>% filter(student == "Yes", balGrp == 3) %>%
```

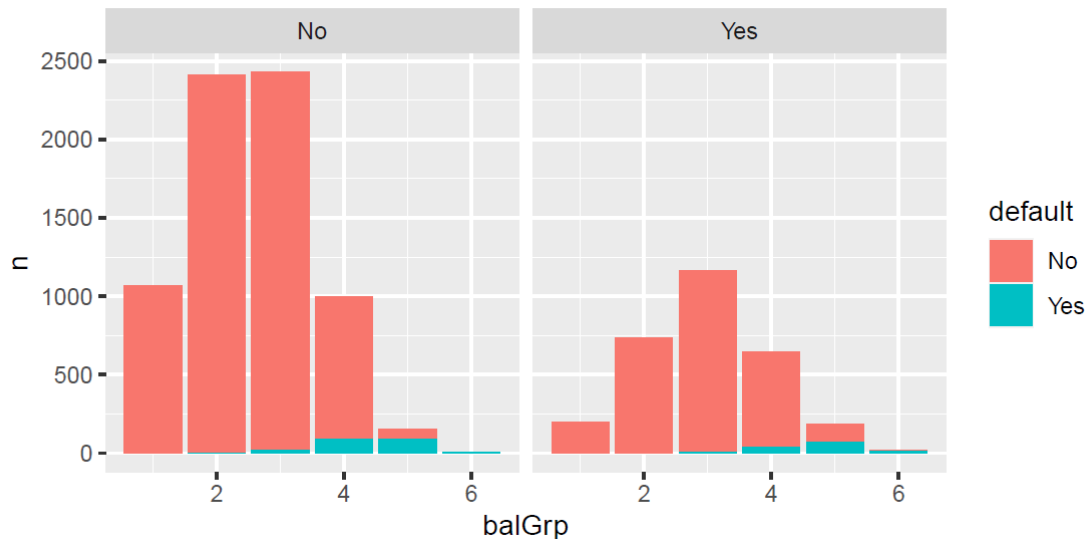
```
+ summarise(Tot = n())
```

```
>
```

```
> BayesN/BayesD
```

Tot

```
1 0.004295533
```



$$\mathbb{P}(Y | X) = \frac{\mathbb{P}(X \cap Y)}{\mathbb{P}(X)}$$

remember, in probability,  $\cap$  is an “and”, so we multiply  
“or” is additive

# comparing *rough* estimate of group probabilities

Summary from Data

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

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

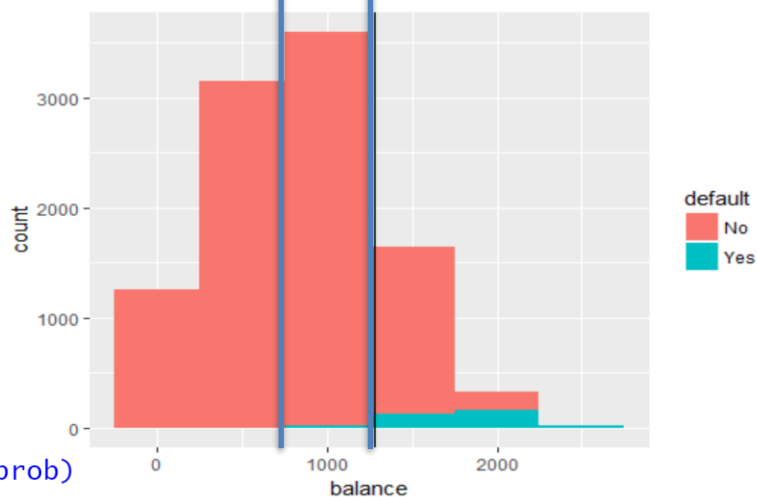
```
pgData %>% filter(xmax == 1250) %>% dplyr::select(xmin, xmax, prob)
xmin xmax  prob
750 1250 0.3593
```

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

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

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

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



Expanding dimensions to:

$P(\text{default} = \text{"Yes"} \mid \text{balGrp (discrete } X) = 3)$

$P(\text{default} = \text{"Yes"} \mid \text{balGrp} = 3, \text{student (discrete } X_s) = \text{"Yes"})$

$$\mathbb{P}(\text{default} = \text{"Yes"} \mid \text{balGrp} = 3) = \frac{.033 * .072}{(.37 * .96) + (.033 * .07)} = .007$$

So, there's a .7% chance of default, given the balance is between (750,1.25e+03)

$$\mathbb{P}(\text{default} = \text{"Yes"} \mid \text{balGrp} = 3, \text{student} = \text{"Yes"}) = \frac{.033 * .072 * .21}{(.37 * .96 * .32) + (.033 * .07 * .21)} = .004$$

So, there's a .4% chance of default, given the balance is between (750,1.25e+03) and a student

default		balGrp		student	n	
No	9667	96.7%	1	1261	13%	No
						Yes
			2	3145	33%	No
						Yes
			3	3569	37%	No
						Yes
			4	1514	16%	No
						Yes
			5	174	2%	No
						Yes
			6	4	0%	No
						Yes
Yes	333	3.3%	1	0	0%	No
						Yes
			2	2	1%	No
						Yes
			3	24	7%	No
						Yes
			4	130	39%	No
						Yes
			5	159	48%	No
						Yes
			6	18	5%	No
						Yes

$$\frac{0.05\%}{F72 * C68} = 0.43\%$$



# 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.

We estimate prior based on the sample (this approach is also called empirical Bayes and naïve Bayes). The classifier assigns an observation to the class for which the  $\log$  ***likelihood*** is largest.

*(note: our logistic regression algorithm also used likelihood, which we'll review again shortly).*

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

```
dfDefault <- Default
```

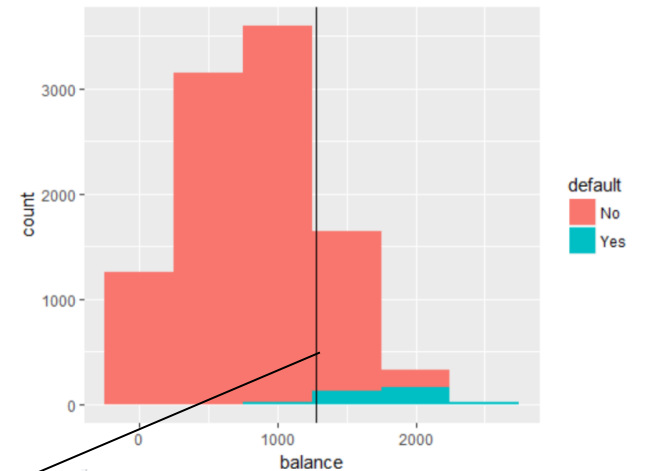
```
p <- ggplot(dfDefault, aes(balance, fill = default)) +
  geom_histogram(binwidth = 500)
p
```

```
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
```

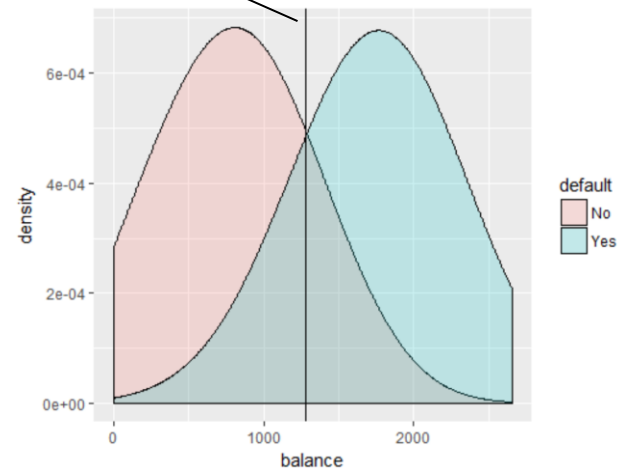
```
lda.pred <- predict(lda.fit)
```

```
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 <- 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
```

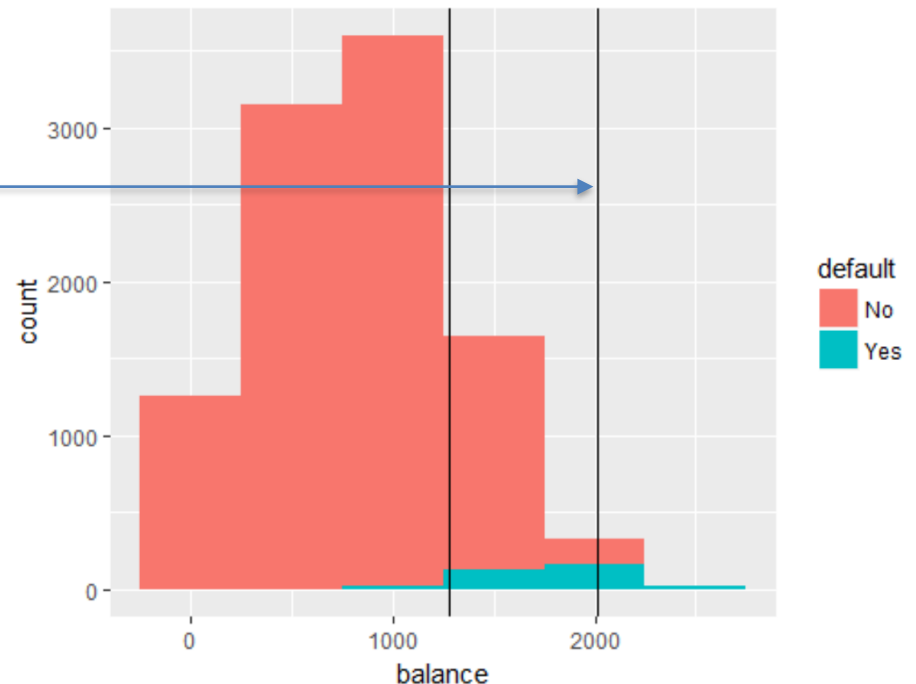
```
> dr <- A + B * C
```

```
> 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)*

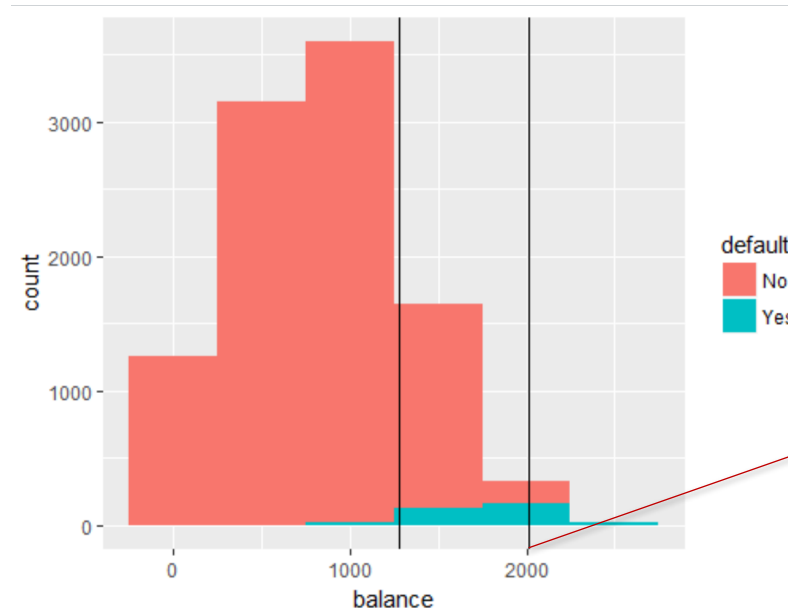


The decision boundary where two log likelihood functions have the same value  $P(\text{default} = \text{"Yes"} \mid \text{conditions}) = P(\text{default} = \text{"No"} \mid \text{conditions})$ .

You can compare results by pulling the posterior for these classes:

```
firstAnalysis <- as_tibble(cbind(as.character(lda.pred$class),  
                                as.character(dfDefault$default),  
                                lda.pred$posterior))  
  
firstAnalysis <- cbind(firstAnalysis, dplyr::select(dfDefault, student, balance, income))
```

*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

## Confusion Matrix and Statistics

	Reference	
Prediction	No	Yes
No	9643	257
Yes	24	76

Accuracy : 0.9719  
95% CI : (0.9685, 0.9751)

No Information Rate : 0.9667  
P-Value [Acc > NIR] : 0.001652

Kappa : 0.3409

Mcnemar's Test P-Value : < 2.2e-16

Sensitivity : 0.2282

Specificity : 0.9975

Pos Pred Value : 0.7600

Neg Pred Value : 0.9740

Prevalence : 0.0333

Detection Rate : 0.0076

Detection Prevalence : 0.0100

Balanced Accuracy : 0.6129

'Positive' Class : Yes

## Review from last week:

**Sensitivity** (also called the **true positive rate**) measures the proportion of positives that are correctly identified.  $76/(76+257) = .22..$

**Specificity** (also called the **true negative rate**) measures the proportion of negatives that are correctly identified.  $9643/(9643+24) = .97...$

**Prevalence** =  $(76+257)/(76+257+9643+257) = .03...$   
*Total Pos in Sample*

**Positive Pred Value** =  $(\text{sensitivity} * \text{prevalence}) / ((\text{sensitivity} * \text{prevalence}) + ((1 - \text{specificity}) * (1 - \text{prevalence}))) = .76$  (est % of predicted positives that were correctly predicted  $76/(76+24)$  for rough)

**Neg Pred Value** =  $(\text{specificity} * (1 - \text{prevalence})) / (((1 - \text{sensitivity}) * \text{prevalence}) + ((\text{specificity}) * (1 - \text{prevalence})))...$   
etc.

# lowering the threshold

```
pred[llda.pred$posterior[,2] >= 0.2] <- 'Yes'
dfPred <- data.frame(pred)
```

		Reference	
Prediction		No	Yes
No		9431	138
Yes		236	195

Accuracy : 0.9626  
 95% CI : (0.9587, 0.9662)  
 No Information Rate : 0.9667  
 P-Value [Acc > NIR] : 0.9886  
  
 Kappa : 0.4914  
  
 McNemar's Test P-Value : 5.283e-07  
  
 Sensitivity : 0.5856  
 Specificity : 0.9756  
 Pos Pred Value : 0.4524  
 Neg Pred Value : 0.9856  
 Prevalence : 0.0333  
 Detection Rate : 0.0195  
 Detection Prevalence : 0.0431  
 Balanced Accuracy : 0.7806  
  
 'Positive' Class : Yes

This doesn't change the probabilities, it just assigns a Yes to observations (*you can do this outside of the algorithm – classifiers usually give you access to the probabilities, and these values are usually updated to another analysis or process application*).

Now, you have increased Sensitivity (*proportion of positives that are correctly identified*), but decreased Pos Pred Value (% of predicted positives that were correctly identified).

This should all makes sense to you – if not, go back and think it through.

All of this depends on the goals of your project.

## Increasing dimensions and using a validation set:

```
dfDefault <- dfDefault %>% rownames_to_column("SampleID")
xTrain <- sample_n(dfDefault, round(nrow(dfDefault)*.6))
xTest <- dfDefault %>% anti_join(xTrain, by = "SampleID")
lda.fit <- lda(default ~ student + balance + income, xTrain)
lda.fit
```

	Reference	
fiction	No	Yes
No	3838	118
Yes	10	34

This is a more realistic approach.  
Remember, we never want to test an algorithm using data it's seen before  
– algorithms are cheaters.

Accuracy : 0.968  
95% CI : (0.9621, 0.9732)  
No Information Rate : 0.962  
P-Value [Acc > NIR] : 0.02375

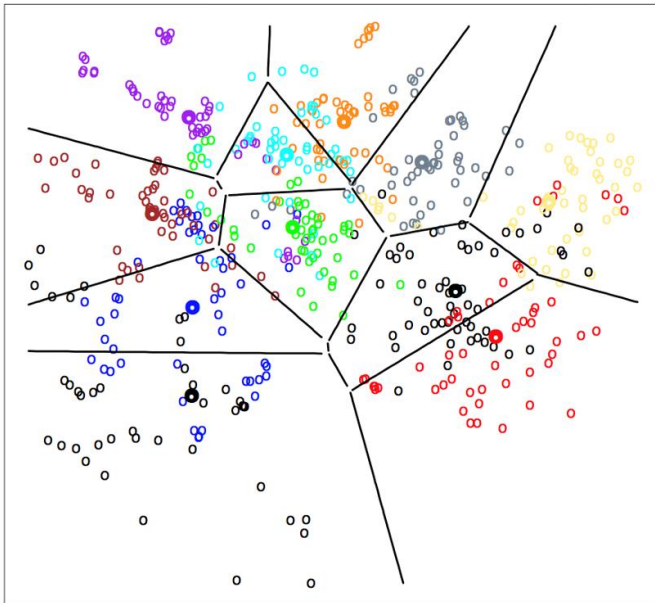
Kappa : 0.3356

Mcnemar's Test P-Value : < 2e-16

Sensitivity : 0.2237  
Specificity : 0.9974  
Pos Pred Value : 0.7727  
Neg Pred Value : 0.9702  
Prevalence : 0.0380  
Detection Rate : 0.0085  
Detection Prevalence : 0.0110  
Balanced Accuracy : 0.6105

'Positive' Class : Yes

# Multiclass LDA



LDA is very common in complex multiclass analysis – you have more control and extensibility.

We touched on multiclass problems last week, and we may revisit.

But we're going to press on with other two-class algorithms for now.



# Reviewing Maximum Likelihood

```
library(tidyverse)
```

```
# normal equations
```

```
Advertising <- dbGetQuery(con2,"  
SELECT
```

```
  [TV]  
  ,[Sales]  
FROM [dbo].[Advertising]  
")
```

```
vY = Advertising$Sales
```

```
mX <- as.matrix(cbind(1, dplyr::select(Advertising, TV))) # set up x  
values in matrix
```

```
vBeta <- solve(t(mX)%*%mX, t(mX)%*%vY) # solve using normal  
equations
```

```
vBeta
```

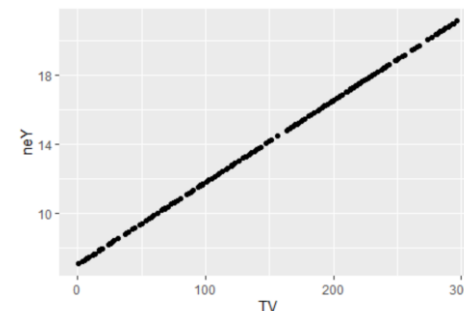
```
vBeta <- as.numeric(vBeta)
```

```
Advertising$neY <- t(vBeta)%*%t(mX) # 3 columns on left * 3 rows on  
right (after transpose)
```

```
ggplot(Advertising, aes(x = TV, y = neY)) + geom_point()
```

# Recall our search for parameters. OLS is actually pretty rare in that it has a closed form solution i.e. its tractable the answer is unique and definite.

# And we can use the same form to generate projected values (simulation).



```
> # linear likelihood function
> linear.lik <- function(theta, y, X){
+   n       <- nrow(X)
+   k       <- ncol(X)
+   beta    <- theta[1:k]
+   sigma2  <- theta[k+1]^2
+   e       <- y - X%*%beta
+   logl    <- -.5*n*log(2*pi) - .5*n*log(sigma2) - (e%*%e)/(2*sigma2)
+   return(-logl)
+ }
>
> # create some linear data
>
> y = as.numeric(Advertising$Sales)
> x = model.matrix(Sales ~ TV, data = Advertising)
>
> linear.MLE <- optim(fn=linear.lik, par=c(1,1,1), lower = c(-Inf, -Inf, 1e-8),
+   upper = c(Inf, Inf, Inf), hessian=TRUE,
+   y=y, X=x, method = "L-BFGS-B")
>
> linear.MLE$par[1]
[1] 7.032609
> vBeta[1]
[1] 7.032594
> linear.MLE$par[2]
[1] 0.04753661
> vBeta[2]
[1] 0.04753664
> |
```

```
x <- mnorm(1000, 10, 2)
df2 <- density(x)
x2 <- data.frame(x = df2$x, y = df2$y)
ggplot(x2, aes(x,y)) + geom_line()

capMu <- matrix( nrow = 0, ncol = 3)

NLL <- function(theta, data) {
  mu = theta[1]
  sigma = theta[2]
  n = length(data)
  t <- n
  NLL = -(n/2)*log(2*pi) - (n/2)*log(sigma^2)
  tmp = 0
  for (i in 1:n) {
    tmp = tmp + (data[i]-mu)**2
  }
  NLL = NLL + (1/(2*(sigma^2)))*tmp
  capMu <- rbind(capMu, c(NLL, mu, sigma))
  -NLL
}
out = optim(par=c(9,1), fn=NLL, data=x)
out$par
```

Generate normal distribution with mean of 10  
and standard deviation of 2

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Gaussian density function from your Cheatsheet

(see why we use a log ☺)

$$\log P(x) = \log \left[ \frac{1}{\sqrt{2\pi}\sigma} \right] - \frac{(x-\mu)^2}{2\sigma^2}$$

```
> out$par
[1] 9.960670 1.963221
> |
```

We can also estimate those parameters using an optimization of the likelihood function (density of the cost function) using optim, which is a derivative based method like gradient descent.

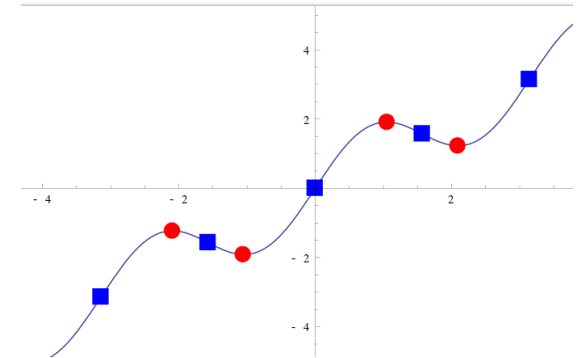
In this case, we are using BFGS.

BFGS is a quasi-Newton method to find the stationary point of a function, where the gradient is 0. Newton's method assumes that the function can be locally approximated as a quadratic in the region around the optimum, and uses the first and second derivatives to find the stationary point.

In higher dimensions, Newton's method uses the gradient and the Hessian matrix of second derivatives of the function to be minimized.

This is a fancy way of saying that we have to worry about local and global minimums (or maximums – i.e., optimizations).

There are a range of ways to find global optimizations, and none of them work “best”. Which is why we set methods for optim.



Recall that Logistic Regression does not have a closed form solution, so we must use MLE and derivative based methods to estimate parameters. The glm function does this for us.

```
<
> 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 ***

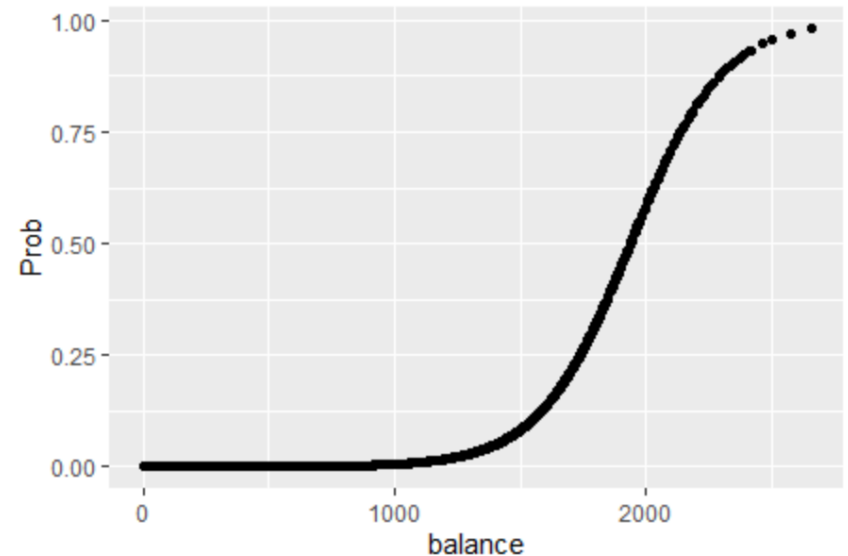
```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
(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
```

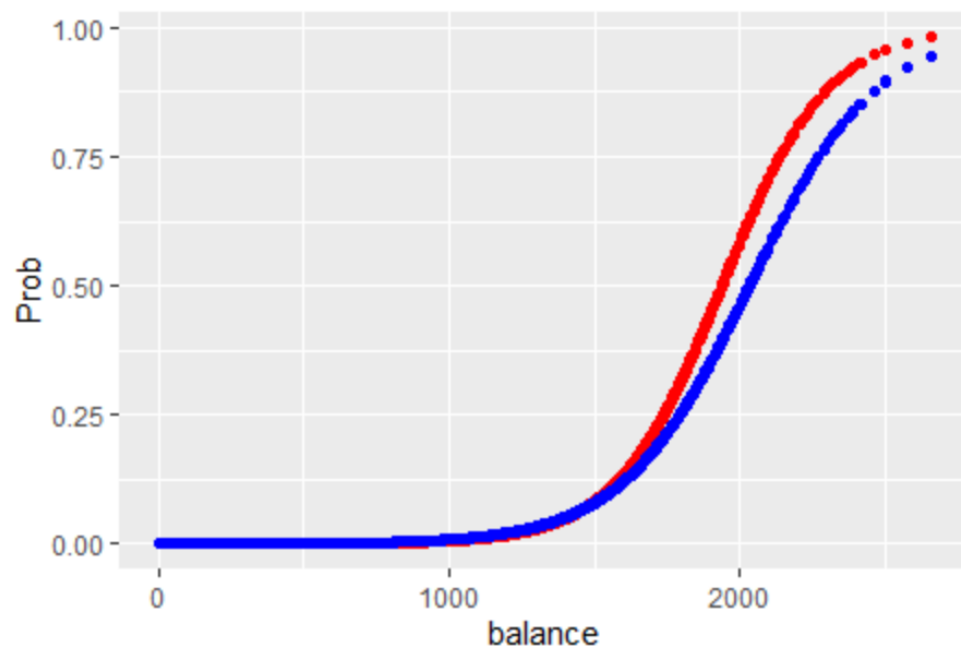
```
Number of Fisher Scoring iterations: 8
```

```
>
> dfDefault$Prob <- predict(glm.fit, type = "response")
> p = ggplot(dfDefault, aes(x=balance, y=Prob)) + geom_point()
> `
```



# We can construct our own glm using a log likelihood function (which is least squares applied to the logistic format) , so a log of the logistic least squares ☺, and as you can see, the estimate is good but not exactly the same (*glm uses MLE and derivative based optimization – just a different method*).

```
> r
> vY = as.numeric(dfDefault$default)-1
> mX = model.matrix(default ~ balance, data = dfDefault)
>
> logit = function(mX, vBeta) {
+   return(exp(mX %*% vBeta)/(1+ exp(mX %*% vBeta)) )
+ }
>
> # stable parametrisation of the log-likelihood function
> # Note: The negative of the log-likelihood is being returned, si
> # /minimising/ the function.
>
> logLikelihoodLogitStable = function(vBeta, mX, vY) {
+   return(-sum(
+     vY*(mX %*% vBeta - log(1+exp(mX %*% vBeta)))
+     + (1-vY)*(-log(1 + exp(mX %*% vBeta)))
+   )
+ )
+ }
>
>
> # initial set of parameters
> vBeta0 = c((alpha - 1), 0)
>
> # minimise the (negative) log-likelihood to get the logit fit
> optimLogit = optim(vBeta0, logLikelihoodLogitStable,
+                   mX = mX, vY = vY, method = 'BFGS',
+                   hessian=TRUE)
>
> optimLogit$par[1]
(Intercept)
-9.324605
> alpha
(Intercept)
-10.65133
> optimLogit$par[2]
```



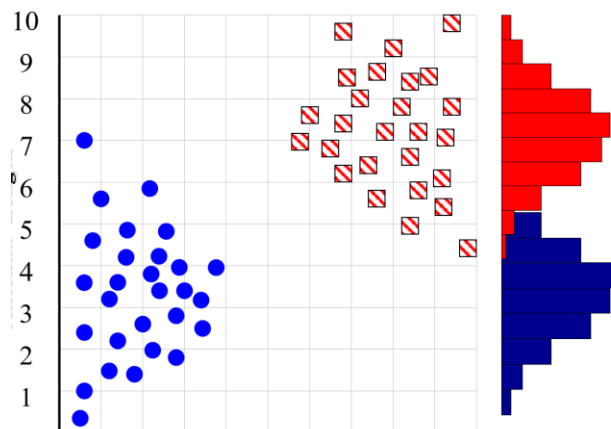
Now we're going to look at some more algorithms with different approaches from LogReg and LDA: Naïve Bayes, Trees and SVM. All have application and can be used productively.

Most of these algorithms are more complex and more flexible, but also more variable than logistic regression. But they may be easier to manage in high dimensional space. And while they get great results in training, they are more high-maintenance in operation. And people will often find themselves generalizing to get back to the same level of precision that logistic regression produces – but without interpretability.

These algorithms also serve as a good baseline when developing explanatory models. So, there's complimentary value too.

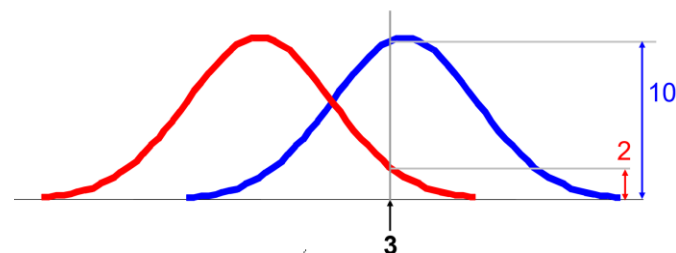
In summary, Machine Learning applications can be very accurate, but complex and sometimes difficult to manage. And perhaps most important, they don't explain relationships and effects, so they are limited in planning and assurance.

# Naïve Bayes



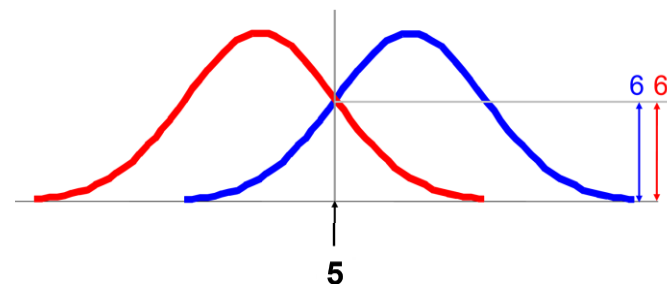
LDA is closely related to NB in that both classifiers assume Gaussian within-class distributions. However, NB relies on a less flexible distributional model in that it assumes zero off-diagonal covariance (*no correlations between variables within a class – i.e. NB assumes variables are independent,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

Surprisingly, LDA generally operates better on continuous, parametric features, while Naïve Bayes often performs better on categorical, non-parametric features (*e.g., NB is the baseline in spam identification*).



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

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



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

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

# Naïve Bayes

```
dfDefault <- dfDefault %>% rownames_to_column("SampleID")
xTrain <- sample_n(dfDefault, round(nrow(dfDefault)*.6))
xTest <- dfDefault %>% anti_join(xTrain, by = "SampleID")
```

```
model <- naiveBayes(default ~ student + balance +
xTest$pred <- predict(model, xTest[, -1], prob = TR
confusionMatrix((xTest$default), factor(xTest$pred
```

Prediction	Reference	
	No	Yes
No	3827	119
Yes	21	33

Accuracy : 0.965

95% CI : (0.9588, 0.9705)

No Information Rate : 0.962

P-Value [Acc > NIR] : 0.1711

Kappa : 0.3066

Mcnemar's Test P-Value : 2.444e-16

Sensitivity : 0.21711

Specificity : 0.99454

Pos Pred Value : 0.61111

Neg Pred Value : 0.96984

Prevalence : 0.03800

Detection Rate : 0.00825

Detection Prevalence : 0.01350

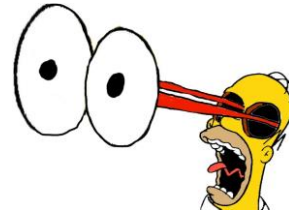
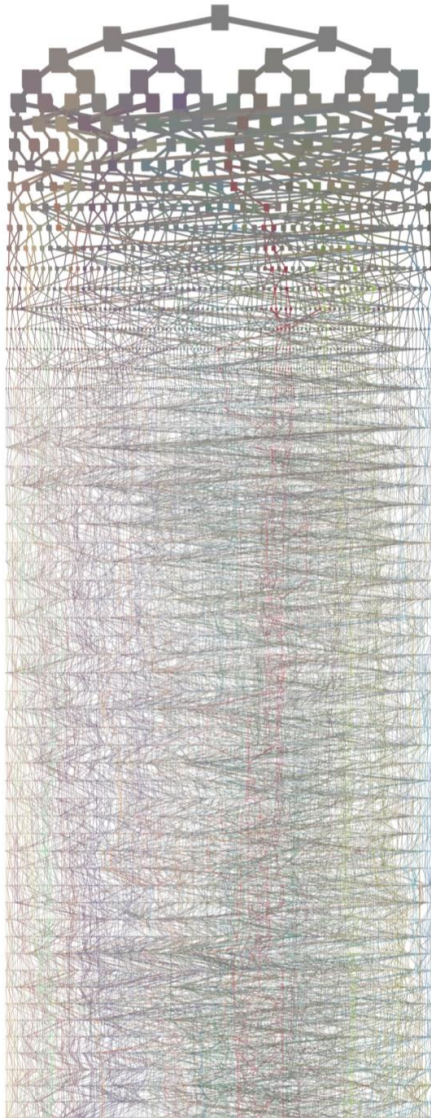
Balanced Accuracy : 0.60582

'Positive' Class : Yes

Improvement over LDA (.19)



# 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;

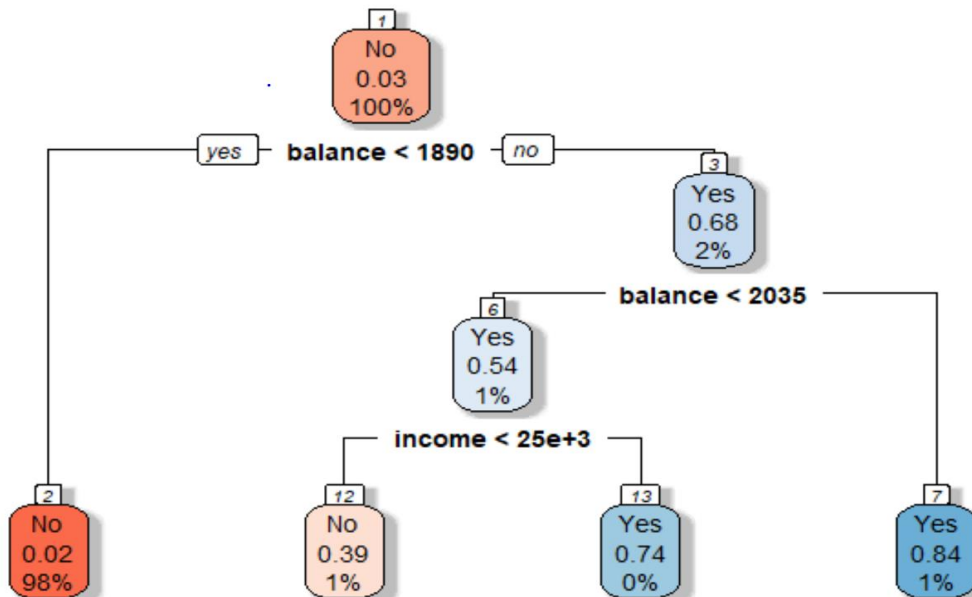
1. Apply cost complexity **pruning** to the large tree in order to obtain a sequence of best subtrees, as a function of  $\alpha$  (*a cost function*)
2. Use K-fold **cross-validation** to choose  $\alpha$  (*usually a parameter*). That is, divide the training observations into K folds. For each  $k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

Improvement over NB (*sensitivity of .21*)



	Reference	
Prediction	No	Yes
No	3834	107
Yes	14	45

Accuracy : 0.9698  
95% CI : (0.964, 0.9748)  
No Information Rate : 0.962  
P-Value [Acc > NIR] : 0.004692

Kappa : 0.4141

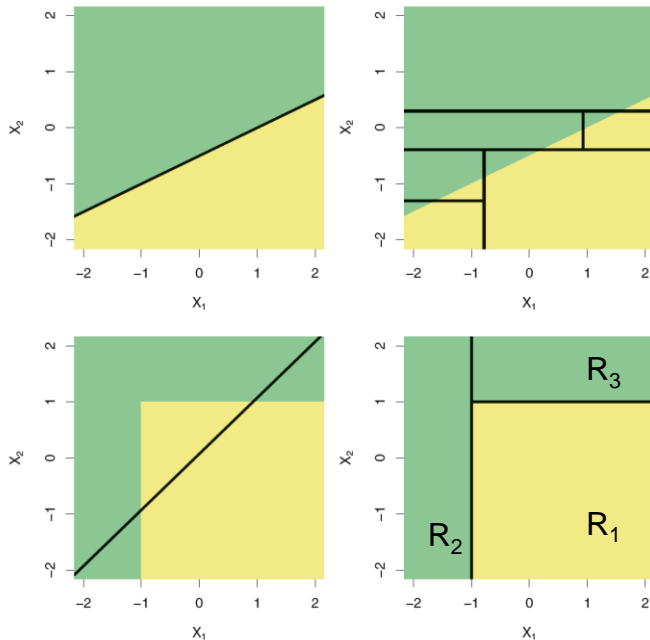
Mcnemar's Test P-Value : < 2.2e-16

Sensitivity : 0.29605  
Specificity : 0.99636  
Pos Pred Value : 0.76271  
Neg Pred Value : 0.97285  
Prevalence : 0.03800  
Detection Rate : 0.01125  
Detection Prevalence : 0.01475  
Balanced Accuracy : 0.64621

'Positive' Class : Yes

## 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

Many algorithms can create new trees as needed

Maximum depth of the d... 


32

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

Number of random splits... 


128

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

Minimum number of sam... 

1

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

☒ Allow unknown value... 

<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)
```

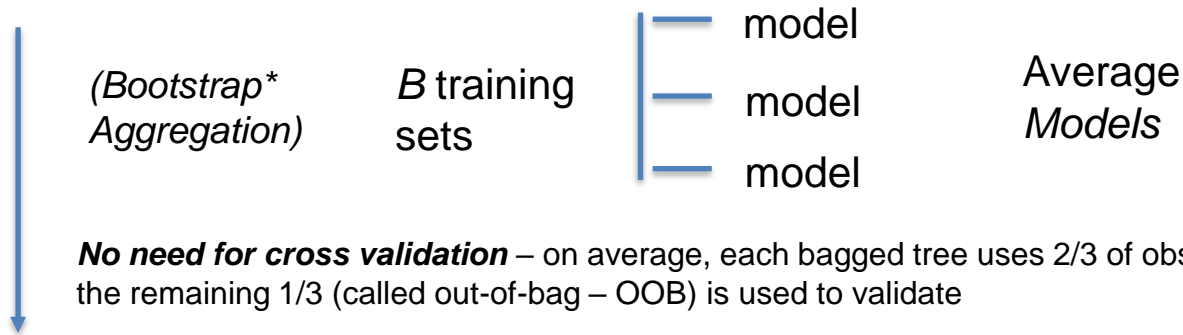
R tree documentation is unique to each package. Here's ranger:

<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



**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

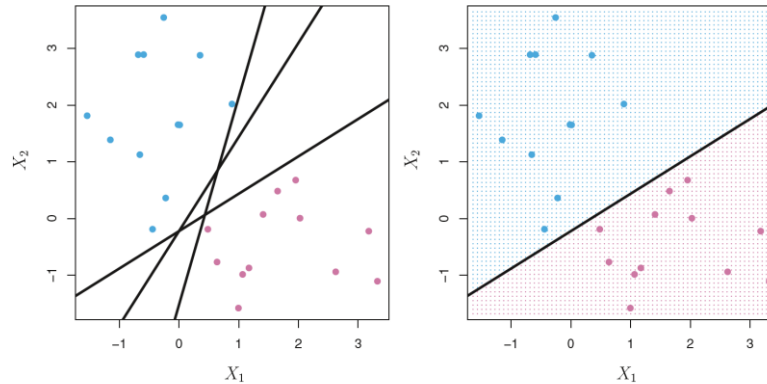
Does not use bootstrapping, it uses a modified sampling that **fits the residuals rather than the predicted value**. More complex, slower learning algorithm (*slower learners are often more accurate, but with higher processing costs*)

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



# Chapter 9: Support Vector Machines

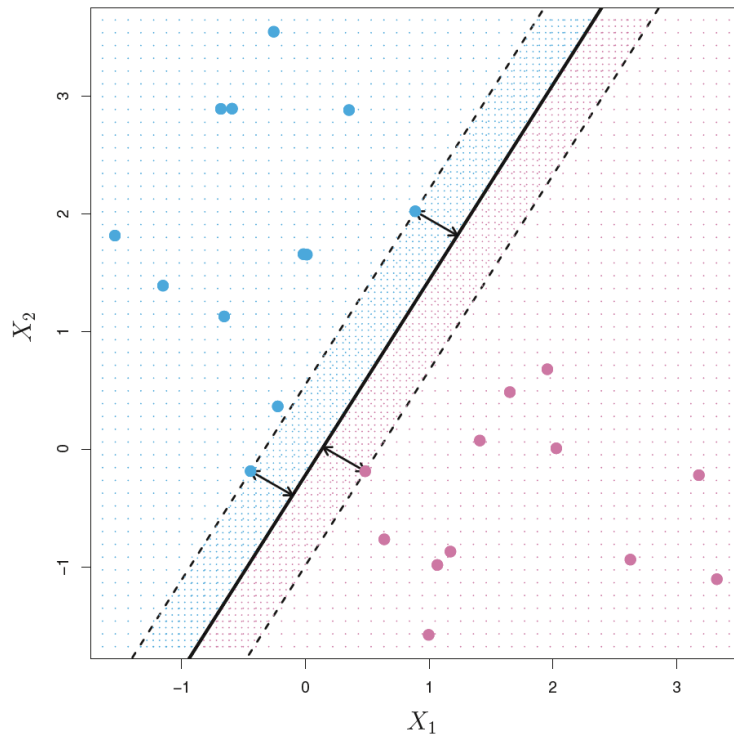
## The Separating Hyperplane



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

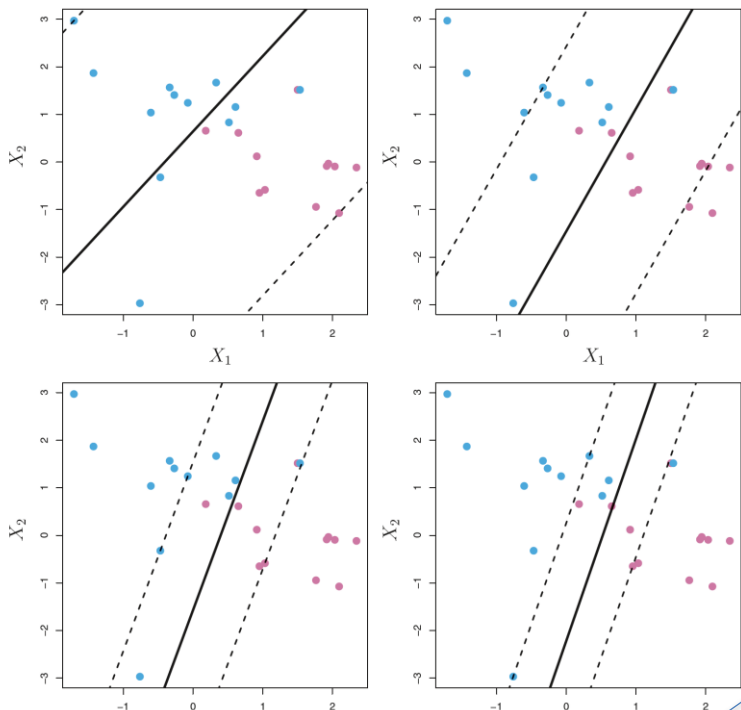
The right-hand panel shows an example of such a classifier. That is, we classify the test observation  $x$  based on the **sign** of  $f(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots$ . If  $f(x)$  is positive, then we assign the test observation to class 1, and if  $f(x)$  is negative, then we assign 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*)



**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.

$$\begin{aligned} & \text{maximize} \\ & \beta_0, \beta_1, \dots, \beta_p, \epsilon_1, \dots, \epsilon_n, M \end{aligned} \quad (9.12)$$

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

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

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

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*

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

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

Notice how the transformation yields completely different values (*also notice that the dimensions are not  $n \times n$* )

	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	1.0000
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

```
dfDefault <- dfDefault %>% rownames_to_column("SampleID")
xTrain <- sample_n(dfDefault, round(nrow(dfDefault)*.6,0))
xTest <- dfDefault %>% anti_join(xTrain, by = "SampleID")
```

```
# showing equation with one categorical variable
```

```
svmMod <- svm(default ~ student + balance + income, data = xTrain)
summary(svmMod)
```

SVMs generally do not perform well on imbalanced datasets. It finds a hyperplane decision boundary that best splits the examples into two classes. The split is made soft through the use of a margin that allows some points to be misclassified. This margin favors the majority class on imbalanced datasets, although it can be tuned (*we'll get into tuning later*).

	Reference	
Prediction	No	Yes
No	3844	133
Yes	4	19

Accuracy : 0.9658  
 95% CI : (0.9596, 0.9712)  
 No Information Rate : 0.962  
 P-Value [Acc > NIR] : 0.1141

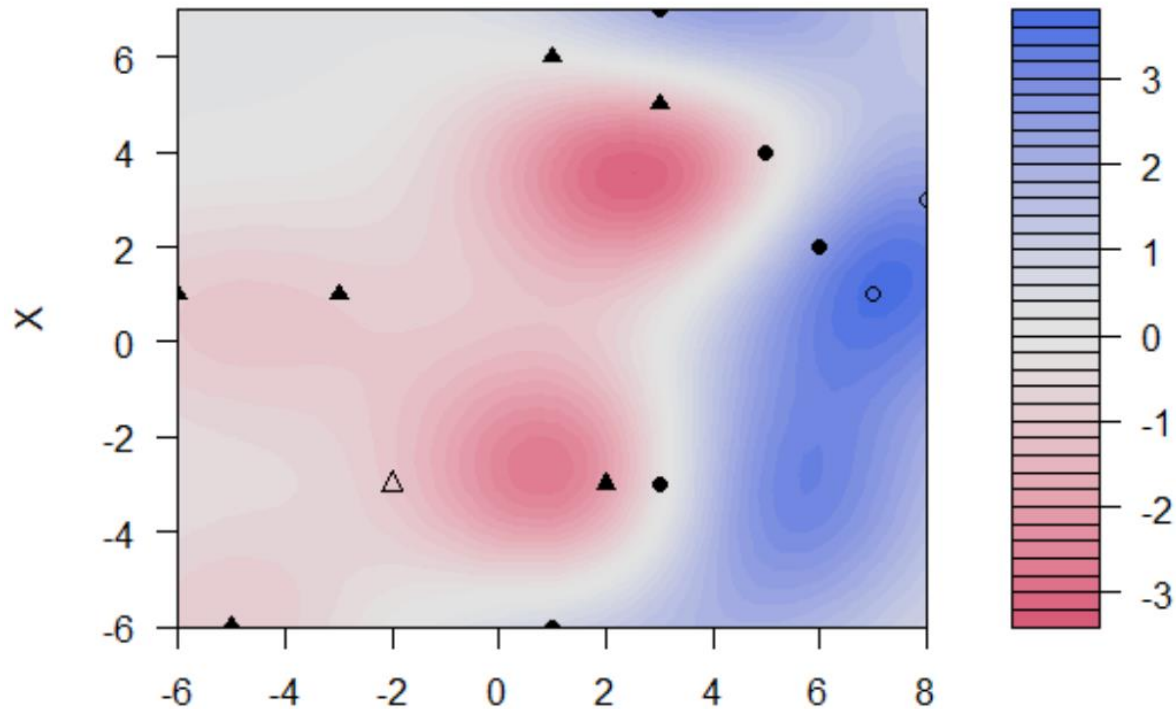
Kappa : 0.2092

Mcnemar's Test P-Value : <2e-16

Sensitivity : 0.12500  
 Specificity : 0.99896  
 Pos Pred Value : 0.82609  
 Neg Pred Value : 0.96656  
 Prevalence : 0.03800  
 Detection Rate : 0.00475  
 Detection Prevalence : 0.00575  
 Balanced Accuracy : 0.56198

'Positive' Class : Yes

**SVM classification plot**

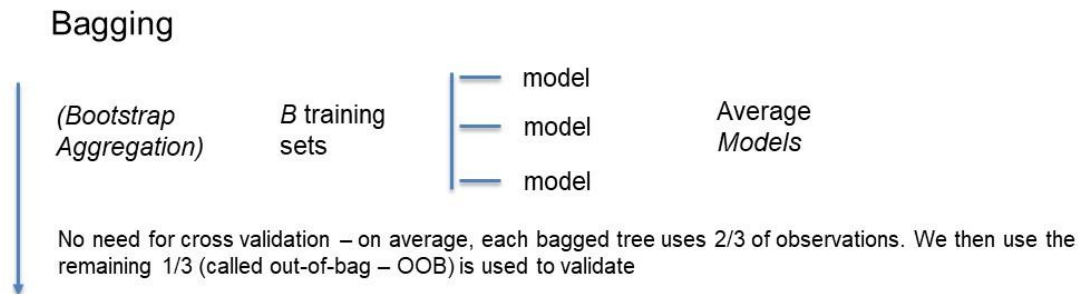


After the kernel transformation, the margins are still linear, but become non-linear with regard to the data. This is a pretty cool visualization.

[https://www.youtube.com/watch?time\\_continue=3&v=3liCbRZPrZA&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=3&v=3liCbRZPrZA&feature=emb_logo)

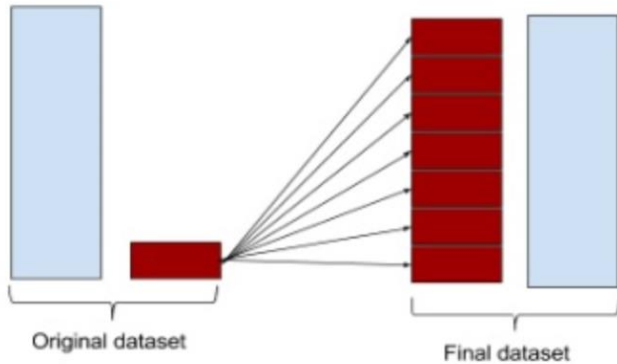
*If you don't have enough data to train your model, you may have to rely on **bootstrapping** where you (repeatedly sample the data with replacement).*

Bootstrapping is used to estimate errors in many models and is integral to ensemble methods. Recall from classification lecture:



*Its also useful for finding starting values and estimating priors in Bayesian modeling*

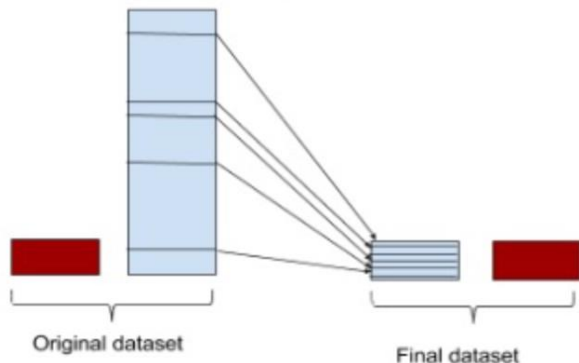
## Dealing with Unbalanced Data: Up-Sampling, Down-Sampling and SMOTE



Up-Sampling randomly replicates minority instances to increase their population.

Down-Sampling randomly downsamples the majority class.

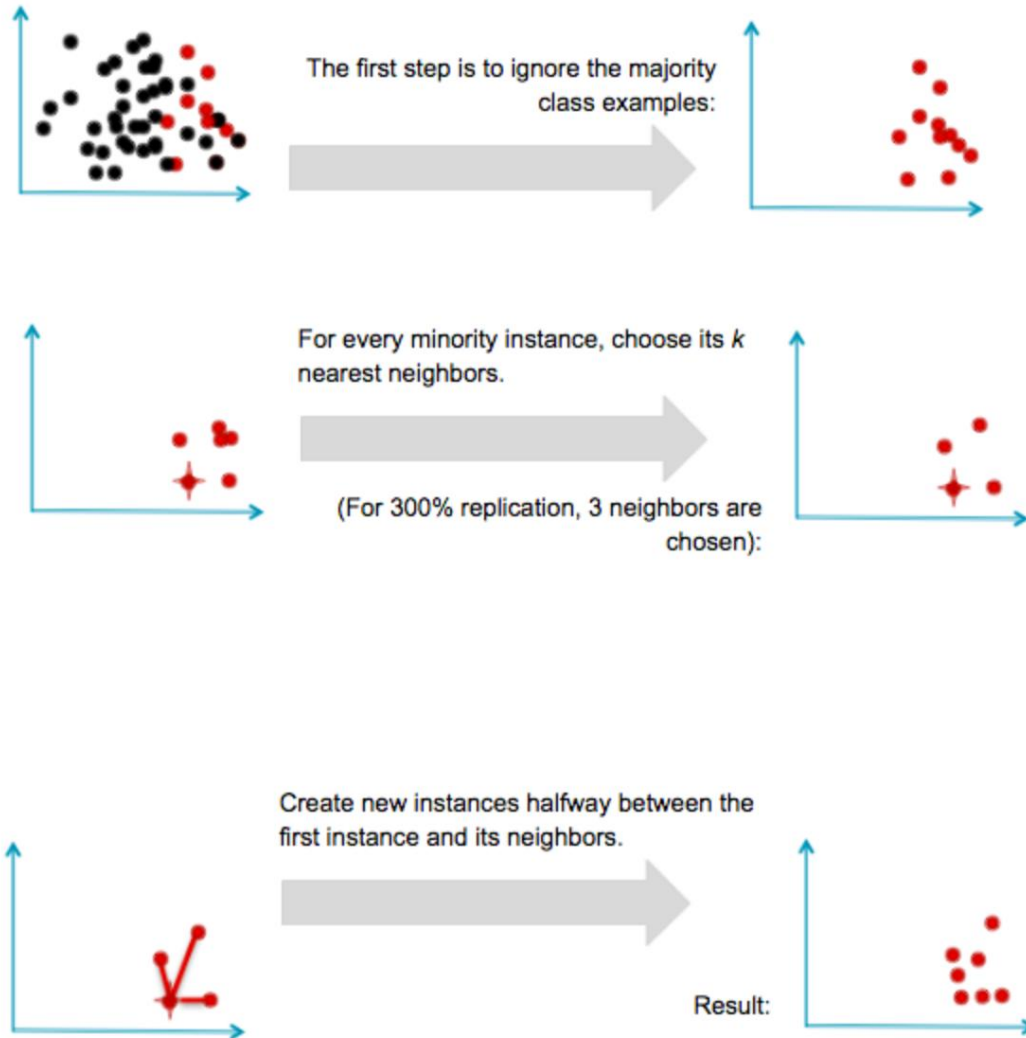
Up-Sampling is NOT necessarily superior because it results in more data, and replicating data is not without consequence—since it results in duplicate data, it makes variables appear to have lower variance than they do.



The positive consequence is that it duplicates the number of errors: if a classifier makes a false negative error on the original minority data set, and that data set is replicated five times, the classifier will make six errors on the new set. Conversely, down-sampling can make the independent variables look like they have a higher variance than they do.

Because expect mixed results and consider hybrid approaches.





SMOTE (Synthetic Minority Oversampling TEchnique) system creates new minority examples by interpolating between existing ones. The process is basically as follows:

Assume we have a set of majority and minority examples. SMOTE is generally successful and has led to many variants, extensions, and adaptations to different concept learning algorithms.

It is important to note a substantial limitation of SMOTE. Because it operates by interpolating between rare examples, it can only generate examples within the body of available examples—never outside.

```
> smoteData <- SMOTE(default ~ student + balance + income, data = Default,
+                     perc.over = 350, perc.under=130)
> # SMOTE only works with factors, so be careful
> prop.table(table(smoteData$default))
```

```
      No      Yes
0.4935361 0.5064639
<
```

So this has the same effect as lowering the threshold: increasing Sensitivity (*proportion of positives that are correctly identified*), but decreasing Pos Pred Value (*% of predicted positives that were correctly identified*).

	Reference	
Prediction	No	Yes
No	3838	118
Yes	10	34

Accuracy : 0.968  
95% CI : (0.9621, 0.9732)  
No Information Rate : 0.962  
P-Value [Acc > NIR] : 0.02375

Kappa : 0.3356

Mcnemar's Test P-Value : < 2e-16

Sensitivity : 0.2237
Specificity : 0.9974
Pos Pred Value : 0.7727
Neg Pred Value : 0.9702

Prevalence : 0.0380  
Detection Rate : 0.0085  
Detection Prevalence : 0.0110  
Balanced Accuracy : 0.6105

'Positive' Class : Yes

	Reference	
Prediction	No	Yes
No	3237	19
Yes	611	133

Accuracy : 0.8425  
95% CI : (0.8308, 0.8537)  
No Information Rate : 0.962  
P-Value [Acc > NIR] : 1

Kappa : 0.2495

Mcnemar's Test P-Value : <2e-16

Sensitivity : 0.87500
Specificity : 0.84122
Pos Pred Value : 0.17876
Neg Pred Value : 0.99416

Prevalence : 0.03800  
Detection Rate : 0.03325  
Detection Prevalence : 0.18600  
Balanced Accuracy : 0.85811

'Positive' Class : Yes

	Reference	
Prediction	No	Yes
No	3838	118
Yes	10	34

Accuracy : 0.968  
 95% CI : (0.9621, 0.9732)  
 No Information Rate : 0.962  
 P-Value [Acc > NIR] : 0.02375  
  
 Kappa : 0.3356  
 McNemar's Test P-Value : < 2e-16  
  
 Sensitivity : 0.2237  
 Specificity : 0.9974  
 Pos Pred Value : 0.7727  
 Neg Pred Value : 0.9702  
 Prevalence : 0.0380  
 Detection Rate : 0.0085  
 Detection Prevalence : 0.0110  
 Balanced Accuracy : 0.6105  
  
 'Positive' Class : Yes

	Reference	
Prediction	No	Yes
No	3237	19
Yes	611	133

Accuracy : 0.8425  
 95% CI : (0.8308, 0.8537)  
 No Information Rate : 0.962  
 P-Value [Acc > NIR] : 1  
  
 Kappa : 0.2495  
 McNemar's Test P-Value : < 2e-16  
  
 Sensitivity : 0.87500  
 Specificity : 0.84122  
 Pos Pred Value : 0.17876  
 Neg Pred Value : 0.99416  
 Prevalence : 0.03800  
 Detection Rate : 0.03325  
 Detection Prevalence : 0.18600  
 Balanced Accuracy : 0.85811  
  
 'Positive' Class : Yes

Sometimes, SMOTE can improve both sensitivity and Pos Pred Val. It depends on the data. Keep in mind that SMOTE is creating data much like you generate data in the regression exercises – random values within a range. But if that range overlaps with the other classes, it can make prediction more difficult. You have to decide based on the goals of your project.

# 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