(2) TWO – Introduction to R

R code	Description	Example
c( )	Used to concatecodnate numbers	x <- c(1, 3, 2, 5) > x > 1 3 2 5
seq( )	Creates a sequence of numbers - Useful for defining grid values to plot with	seq(a, b) makes a vectors of integers (starts with *a* and increments by 1 till *b*) seq(0, 1, length = 10) makes a seq of 10 numbers that are **equally spaced** between *0* and *1* 1:10 and seq(1, 10) are equivalent seq(-pi, pi, length = 50) creates something useful for a trigo plot
ls( )	Shows a list of all the objects saved in the workspace	rm(list = ls()) is useful in removing all objects from the workspace
rm( )	Used to delete objects from the workspace	

matrix( )	Creates a matrix of numbers
	(fills in columns first, by default)
		x <- matrix(data = c(1, 2, 3, 4),
		nrow = 2,
		ncol = 2)
		> x
		> 1 3
		2 4
		x <- matrix(..., ..., ..., byrow = T)

rnorm( )	Generates normal random variables
	- get pseudorandom numbers that we can treat as random
	- almost sure to get a different number each time it's called
	- by default, ~N(0,1)
		x <- rnorm(50)
		generates 50 random *x*-s
		x <- rnorm(50, mean = 50, sd = 0.1)

cor( )	Compute sample correlation	Creating 2 correlated sets of numbers,
		x <- rnorm(50)
		y <- x + rnorm(50, 50, 0.1)
		cor(x, y)

mean( )
var( )
sd( )
	Compute the mean and sample variance of a vector of numbers	The following are equivalent:
		sd(y) and sqrt(var(y))

set.seed( )	To reproduce same results, 'resets' randomness to a reproducible state - useful in simulation studies	

plot(x, y)	Produces a scatterplot of *y* against *x*	plot(x, y,
		xlab = "...", ylab = "...",
		main = "...")

pdf( )
jpeg( )
	Saves the output
	(alternative to using 'Export' function under "Plots" in RStudio)
		pdf("filename.pdf") or jpeg("name.jpeg")
		plot(x, y, ...)
		dev.off( )

contour( )	Produces a contour plot like a **topographical map**
	- needs *x*, *y* grid coordinates and a matrix *z* for the height values
		y <- x
		f <- outer(x, y,
		function(x, y) cos(y)/(1-x^2))
		contour(x, y, f)
		contour(x, y, f, nlevels = 45, add = T)

image( )	Similar to *contour()* but gives a **heatmap** instead of contour lines
	- lighter colours correspond to higher values
		fa <- (f-t(f))/2
		image(x, y, fa)

persp( )	Generates **3D plots** - *thets*: horizontal rotation - *phi*: vertical rotation	persp(x, y, fa, thets = 30, phi = 40)

hist( )	Plots a histogram	hist(mpg)
		hist(mpg, col = 2, breaks = 15)

pairs( )	Plots a scatterplot matrix for the entire data set
	- can also specify just a subset of variables
	(useful many variables)
		pair(Auto)
		pairs(~ mpg + displacement, Auto)

identify( )	** **Label points on plot**
	- returns the row number of points identified
		plot(horsepower, mpg)
		identify(horsepower, mpg, name)
		then 1. click on interested points
		2. click 'finish'

[ ]	Indexes data - Not specifiying *row_idx*: all columns - Not specifying *col_idx*: all rows - Negative indexing to omit/remove elements	> A > 1 5 9 13 2 6 10 14 3 7 11 15 4 8 12 16 A[c(1, 3), c(2, 4)] where [*row_idx, col_idx*] > 5 13 7 15
dim( )	Gives the number of rows and columns of a matrix	
setwd( )	Sets working directory	

read.table( )	Reads in a text file and stores it as a *dataframe*
	- if it is text data, can use external text editor to peek at data to get things right the first time, i.e. input extra arguments (recommended)
		auto <- read.table("Auto.data".
		header = T, na.strings = "?")

fix( )	Launches a spreadsheet-like window to view data - not recommended - use *View()* instead	
na.omit( )	Removes rows containing missing data (*NA*)	auto <- na.omit(Auto)
names( )	Displays all the variable names in the data frame	names(Auto)

attach( )
detach( )
	Makes variables in a data frame available by name
	- good to detach after you are done
	- no need to use $ to refer to a variable within a data frame
		attach(Auto)
		plot(cylinders, mpg)
		detach(Auto)

as.factor( )	**** Converts quantitative variables into qualitative variables**
	- using *plot()* will generate a boxplot instead now
	(where half of the points are in the box)
		cylinders <- as.factor(cylinders)
		plot(cylinders, mpg, varwidth = T)
		where *varwidth* = *T* tell the number of points in a bar by its width/"fatness"

summary( )	Provides a numerical summary of each variable in the data frame - can also use on a single variable	

q( )	To quit R
	- Recommended to save into a workspace to be loaded instead of default workspace that gets quto-loaded
		save.image()
		load()

(3A) Three A – Linear Regression (simple, multiple)		

library(MASS)
library(ISLR)
	- *Mass* contains a large collection of data sets and functions (in base R)
	- *ISLR* includes data sets from the textbook	

Simple Linear Regression		
lm(y ~ x)	Runs a simple linear regression	model <- lm(data = Boston, medv ~ lstat)
model	Gives the basic coefficient estimates - Just calling the regression model - "*Model*" here is just the name of the variable where output of *lm()* is stored	> model > Call: lm(formula = medv ~ lstat) Coefficients: (Intercept) lstat 34.55 -0.95
**summary(model)	Gives more details - Summary of residuals - Standard errors, t-values, p-values of coefficients - Residual standard error, degree of freedom - Multiple R-squared, adjusted R-squared - F-statistic, p-value **- ** In SLR, p-value of coefficient (beta1) should be the same as F-statistic**	summary(model)
names(model)	Get names of all variables in the list where output of *lm()* is a list - Can use $ to extract out the values of these variables	> names(model) > "coefficients" "residuals" "effects" "rank" "fitted.values" "assign" "qr" "df.residual" "xlevels" "call" "terms" "model"
model$coefficients model$coef coef(model)	Returns coefficient estimates - Can use shortcut and type till variable name is unique	
**confint(model)	Creates confidence intervals for coefficients	> confint(model) > 2.5 % 97.5% (Intercept) 33.448457 35.6592247 lstat -1.026148 -0.8739505
**predict( )	Get confidence intervals for specified values of *x or* get prediction intervals for specified values of *x or* for training set - Make sure that data frame for specified values of *x* have the **same column name** as in the model	> predict(model, data.frame(lstat = c(5, 10, 15)), interval = "confidence") > fit lwr upr 1 … … …

	- These values have to be in a data frame because *predict()* is a general function that works for MLR as well (i.e. more variables)	2 ... ... 3 ... ... shows the confidence interval for the respective specified *x* values > predict(model, data.frame(...), interval = "prediction") > predict(model)

abline(model)	Adds a line to an existing plot
	- Useful for adding the least-squares regression line to a scatterplot of the training points
		plot(lstat, medv)
		abline(model, lwd = 3, col = "red")

plot(1:20, 1:20, pch = 1:20)	Shows the different symbols available to plot	

plot(model)	Generates multiple plots that can be toggled through
	1. Residuals against fitted values
	2. QQ plot
	3. Scale-location
	4. Residuals against leverage
		par(mfrow = c(2, 2))
		where default is *par(mfrow = c(1, 1))*
		plot(model)
		dev.off() closes the plotting device/resets it

residuals(model)
rstudent( )
	Get residuals plot directly
	(i.e. residuals against fitted values)
		>plot(predict(model),
		residuals(model))
		>plot(predict(model),
		rstudent(model))
		gives standardised residuals (i.e. dividing by standard error)

hatvalues(model)	Computes leverage statistics - Works for multiple linear regression objects as well	> plot(hatvalues(model)) > which.max(hatvalues(model)) to return the index of the largest element (i.e. the largest leverage point)
Multiple Linear Regression		
lm(y ~ x1 + x2 + x3 + ...)	Fits a multiple linear regression model	
lm(y ~ . , data = ...)	Regress on **all** variables	
summary(model)$...	Access individual components of *summary()*	> names(summary(model)) "call" "terms" "residuals" "coefficients" "aliased" "sigma" "df" "r.squared" "adj.r.squared" "fstatistic" "cov.unscaled" > summary(model)$r.squared > summary(model)$sigma

(3B) Three B – Other considerations in linear regression

vif(model)	Computes variance inflation factors - Measures collinearity - Overly high VIF values (>10)	
var1:var2 var1*var2	Interaction terms - Includes the interaction of *var1* and *var2* - Usually *var1*var2* is used because of hierarchical principle	model <- lm(data = Boston, medv ~ lstat*age) includes the main effects *lstat* and *age*terms
lm(y ~ .-var1)	Modelling using all variables except one	
update(model, ~.-var1)	Update a model that is already fitted - Either adding or removing a variable	

lm(data =...,
y ~ f(var1))
	Non-linear transformation of predictors
	- Need *I( )* as a wrapper when using ^, which has a different meaning in *lm( )*
	(i.e. order of interaction effect)
		model2 <- lm(medv ~ lstat + I(lstat^2))
		model3 <- lm(medv ~ log(rm))

anova(model1, model2)	Compare 2 models - *Null hypothesis*: 2 models fit the data equally well - *Alternative hypothesis*: full model fits better - Can be used to compare nested model when adding multiple predictors	
poly(var, order)	Used for higher order polynomial regression - Creates an orthogonal basis - Each *poly(var, order)x* have slightly different coefficients and also different t-statistic values - Generally, high degree polynomials are a bad thing (they have a high increase at the end)	model5 <- lm(medv ~ poly(lstat, 5)) creates a 5th order polynomial fit > summary(model5) > Call: lm(formula = medv ~ poly(lstat, 5)) Residuals: Min 1Q Median 3Q Max ... ... Coefficients: Estimate Std.Error t value Pr(>|t|) (Intercept) poly(lstat, 5)1 poly(lstat, 5)2 poly(lstat, 5)3 poly(lstat, 5)4 poly(lstat, 5)5 ...

Qualitative Predictors

- *lm( )* will code qualitative predictors automatically

Example:
- For *Carseats* data, we model response *Sales* with a number of predictors
- One of them is the shelving location, *ShelveLoc*, is a qualitative predictor with 3 levels – *Good*, *Medium*, *Bad*

```
model <- lm(data = Carseats,
               Sales ~ . + Income:Advertising + Price:Age) # everything and includes interaction terms
```

- In *summary(model)*, *ShelveLoc* appears as 2 dummy variables – *ShelveLocGood* and *ShelveLocMedium*
- There is no need for *ShelveLocBad* since it can be determined from the 2 dummy variables

contrasts(var)

Shows the current set of contrasts being used
- This coding can be changed according to preference (check *?contrasts*)

```
> contrasts(ShelveLoc)
>               Good      Medium
  Bad            0          0
  Good           1          0
  Medium         0          1
```

where *Medium* can be changed to (0, 0) to set it as the benchmark

(4) Four B – Classification

- Train set:
  - Used to train model
  - Size of set depends on number of testing that will be done
- Validation set:
  - Used to **tune parameters** (e.g. *K*) or **select features** (e.g. how many variables)
  - Size of set depends on method
- Test set:
  - Used to make sure the results so far are valid

- Other tips:
  - If working on a ***time series***, might not want to randomise because can give glimpse of future (fix first part as train and second part as test *(train data comes before test)*, how large each part is can be chosen)

cor( )	Gives correlation matrix - for **quantitative** predictors	

Logistic Regression

- Tip: consider fitting smaller models by just including predictors that have the smallest p-values in the full model

Note of caution:

- Use test set sparingly

- Repeatedly using unseen test set to evaluate error may result in overfitting
- Similar to making use of test set as training set to *select best classifier*
- If keep testing many different classifiers, bound to have one where results are "good" *by chance*
- One way to validate results is to use yet another untouched test set

glm(y ~ x1 + x2 + x3 +…, data = df, family = binomial)	Fits generalised linear models - class of models including logistic regression	
summary(model)	Provides the usual estimated coefficients, standard errors, p-values etc.	- Also gives "*Number of Fisher Scoring Iterations*" - This is how much more times the processing took compared to LS (logistic regression uses iterated related LS to get solutions)
coef(model)	Extracts coefficients	
preds <- predict(model, type = "response") predict(model, newdata = …, type = "response")	Get fitted **probabilities** for the *training* set or predicted probabilities for new *test* data	- If *type = "response"* is not specified, the default for logistic regression would be to output the logit - Note that if you get fitted probabilities, later on this will be used in the calculation of the **training error rate**
contrasts(y)	To find out what the probabilities are predicting	> contrasts(Response) > Up Down 0 Up 1

ifelse(probs > threshold, 1, 0)
 	
preds <- rep(0, nrow(df))
preds[probs > threshold] <- 1
	To get predicted class **labels** by setting a rule	- The second method updates the labels for those whose predicted value is of class '1'

con_mat <- table(preds, y)	Creates confusion matrix	> table(pred, Direction)
		> Direction
		   glm.pred Down Up
		   Down 145 141
		   Up 457 507

mean(preds == y)
 	
sum(diag(con_mat))/ sum(con_mat)
	Gets accuracy rate	

df[!train,]	Subsetting out the test set - This can be done with a vector of indexes, or a **logical** vector	> train <- Year < 2005 > Smarket.2005 <- Smarket[!train,] > Direction.2005 <- Direction[!train,]
glm(y ~ x1 + x2 + x3 +…, data = df, family = binomial, subset = train)	Using logistic regression on a dataframe, separating into train and test sets - Make use of the argument *subset* to specify rows you want for training	
	Linear Discriminant Analysis	
library(MASS)	*lda()* function is part of the *MASS* library	
lda_model <- lda(y ~ x1 + …, data = df, subset = train)	Fit an LDA model	
lda_model	Call the model directly to view details about model - Unlike logistic regression, there are no more coefficients - Estimated prior probabilities and means of each class is given directly in output - In practice, for computational efficiency, equivalent form of LDA decision rule is computed - For binary case, this is a single vector of coefficients to be multiplied by predictors	> lda_model Call: lda(Direction ~ Lag1 + Lag2, data = Smarket, subset = train) Prior probabilities of groups: Down Up 0.491984 0.508016 Group means: Lag1 Lag2 Down 0.04279022 0.03389409

	- Large values of these linear combination corresponds to a higher posterior for the class coded as '1' in *contrasts()* - Lower values correspond to class '0'	*(for days that are 'Down')* Up -0.03954635 -0.03132544 *(for days that are 'Up')* Coefficients of linear discriminants: LD1 Lag1 -0.6420190 Lag2 -0.05135293
preds <- predict(lda_model, df_test)	Gives predicted class and also posterior probabilities - Can use other thresholds	
names(preds)	There are 3 attributes to the predicted output: **1) class***** 2) posterior 3) x (which is used to compute (2))	> names(lda.pred) [1] "class" "posterior" "x"
preds$posterior	Gives posterior probabilities of each class - Use *colnames(preds$posterior)* to determine what each column is predicting for - Different thresholds can be used to recreate the predicted class labels - Note: Sometimes the thresholds should be changed in small increments, especially if the greatest posterior probability is not so big	For a 50% threshold, find the number of observations predicted as 'Down' (col 1) > sum(lda.pred$posterior[, 1] >= 0.5) [1] 70 > sum(lda.pred$posterior[, 1] < 0.5) [1] 182 Be wary of setting the threshold too high, > sum(lda.pred$posterior[, 1] > 0.9) [1] 0 > max(lda.pred$posterior[, 1]) [1] 0.520235
Quadratic Discriminant Analysis		
library(MASS)	*qda()* is also part of the *MASS* library	
qda_model <- qda(y ~ x1 + x2 + ..., data = df, subset = train)	Fit a QDA model	
qda_model	Output is similar to *lda()* but missing QDA computation terms (coefficient of discriminants)	
qda_pred_class <- predict(qda_model, df_test)$class		
table(qda_pred_class)	Creates confusion matrix (similar to above examples)	
K-Nearest Neighbours		

library(class)	*knn()* is part of the *class* library	

train_X <- cbind(x1, x2)[train,]
test_X <- cbind(x1, x2)[!train,]
train_y <- y[train,]
	There is no need to train in *knn()*, which predicts in a single command that requires:
	1. Matrix/dataframe containing **variables** from **train** set
	2. Matrix/dataframe containing **variables** from **test** set (observations to predict)
	3. Vector containing **class labels** for train set
	4. ***K***, number of nearest neighbours to look at	

set.seed(1)	Set a seed for reproducibility since *knn()* **breaks ties randomly**	A tie happens when - 2 neighbours are equally far apart or - *K* is an even number and there are equal number of votes
knn_pred <- knn(train_X, test_X, train_y, k = ...)		
mean(test_y != knn_pred)	Gives the error rate (because classifier, either prediction correct or wrong)	# Comparison to a *trivial classifier* # ('No' all the time) > mean(test_y != 'No')
*** stdised_x <- scale(df)	***** When predictors are not all on the same scale** - Standardise data such that all variables have mean = 0 and sd = 1 - Important since KNN uses Euclidean distance which does take into account the scale - Variables on a large scale will have a larger effect on the distance between observations	# Example 1 # Difference of $1000 in *salary* is enormous compared to difference of 50 years in *age* (but common sense says otherwise) # Example 2 # Variable in column 1 is gonna dominate by 1000 > var(Caravan[, 1]) [1] 165.0378 > var(Caravan[, 2]) [1] 0.1647078
Application: Caravan Insurance Data		

(5) FIVE – Resampling Methods

- Cross-validation is a very general method that can be used for different methods and performance metrics
- To adapt to other settings, knowing how to code out CV will be useful (using *sample()*, *for()*)

R code	Description	Example
set.seed( )	- Set a seed for R's random number generator - Use when performing analysis that contains element of randomness for reproducible work	set.seed(1)
library(boot)	Library needed to do CV and bootstrap	
Validation Set Approach		
sample( )	Generate random subset - simple random sample	train <- sample(n, n/2) takes half of indices as training obs

lm.fit <- lm(y ~ x,
data = df,
subset = train)
	Fit model only on training set using *subset()* option	

mean((test_y - predict(model, df)[-train])**2)	Calculate test MSE - note: this value is random (if re-run, samples change) - Predict/estimate all responses but only take errors corresponding to validation set	
fit( )	For train set	
predict( )	For validation set	
lm.fit2 <- lm(y ~ poly(x, poly_num), data = df, subset = train)	Higher order polynomial regression	...y ~ var1... (linear) ...y ~ poly(var1, 2)... (quadratic) ...y ~ poly(var1, 3)... (cubic)
Leave-One-Out Cross-Validation (LOOCV)		
glm(y~ x, data = df)	Without *family = "binomial"*, *glm()* is just the usual linear regression and not logistic regression - Use *glm()* for linear regression to do CV	
library(boot)	Library for cross-validation	
cv.error <- cv.glm(df, model) cv.error$delta	Do LOOCV using fitted model - *cv.glm()* does not take advantage of shortcut for least squares	glm.fit <- glm(mpg ~ horsepower, data = Auto) cv.glm <- cv.glm(Auto, glm.fit) cv.err$delta [1] 24.23151 24.23114 these are the cross-validation results: **1. CV estimate of MSE** **2. bias-corrected version** (only possible if form of bias is known, not possible to correct for bias analytically for different learning methods and error metrics)
k-Fold Cross-Validation (k-fold CV)		

cv.glm(y~x, data = df, K = num_folds)	Does k-fold cross-validation - common choice for *k = 10* or *5* - much faster than LOOCV - each model is only fitted *k* times	
Coding k-Fold CV yourself		
sample(n)	Samples n times from 1 to n without replacement - A useful method when coding out k-fold CV by yourself - Used to randomise **indexes** of dataframe in the beginning - **Don't modify** dataframe - Use indexes to **extract out relevant rows**	

index = sample(n)
foldbreaks = c(0, floor(n/k*1:k))
or
folds <- sample(rep(1:k, length.out = n), n)
	Getting indexes of observations of each fold
	- Rearrange the index (not the data)	

Bootstrap		
- Estimates the accuracy of a statistic of interest		
library(boot)		

fn <- function(df, index) { ... }	To estimate accuracy of a statistic of interest, write a function that computes statistic given data and relevant row indexes
	- E.g. *mean, median* etc
		mean.fn <- function(data, index) {
		X = data$X[index]
		Y = data$Y[index]
		return(mean(X))
		}

fn(df, sample(n, n, replace = T)) ***	To get bootstrap estimate of statistic, generate a **bootstrap sample index** by sampling with replacement - If run this many times using a loop, can store all the bootstrap estimates - With estimates can look at distributions of estimates or do further analysis	
boot(df, fn, R = num) ***	Automates the above procedure - Gets all bootstrap estimates for different bootstrap samples - Computes **common bias** and **standard error** estimates	> boot(Portfolio, alpha.fn, R = 1000) where ***R*** is the number of bootstrap replicates ORDINARY NONPARAMETRIC BOOTSTRAP Call: boot(data = Portfolio, statistic = alpha.fn, R = 1000) Bootstrap Statistics

		original bias std. error t1* 0.5758321 -7.315422e-05 0.08861826

fn <- function(df, index) { …}	To estimate accuracy of a linear regression model, write a function that computes the coefficient estimates
	- Compare bootstrap estimates with those derived from theory (using *summary(model)$coef*)
	- Can compute standard error estimates analytically because coefficient estimates are unbiased
	- Note that the estimates from theory and bootstrap may differ because standard error formulas reply on certain theoretical assumptions
	(e.g. estimate of variance relies on linear model being correct, but true relationship between X and Y may be non-linear)
	- If true relationship is non-linear then the coefficient estimates should be compared to a different model (e.g. *lm(y~x**2)* in both the bootstrap function and model for full dataset)
		boot.fn <- function(data, index) {
		    lm_model =
		        lm(mpq ~ horsepower,
		        data = data, subset = index)
		    return(coef(lm_model))
		}
		
		> boot.fn(Auto, 1:392) # Full dataset
		(Intercept) horsepower
		39.9358610 -0.1578447

boot( )	Similar to runing function through a loop with bootstrap samples and recording down bootstrap estimates (in a matrix with a column for each coefficient estimate/beta)	> boot(Portfolio, boot.fn, R = 1000) ORDINARY NONPARAMETRIC BOOTSTRAP Call: boot(data = Portfolio, statistic = boot.fn, R = 1000) Bootstrap Statistics original bias std. error t1* 39.9358610 0.02972191 0.860007896 t2* -0.1578447 -0.00030823 0.007404467
Others - Estimates of statistics		
mean(X)	Estimate of population mean/sample mean (*mule_hat*)	

sqrt(var(X)/n)
sd(X)/sqrt(n)
	Estimate of standard error of *mule_hat*	

median(X)	Estimate of population median value	
quantile(X, 0.1)	Estimate of tenth percentile of X in population	

(6A) SIX A – Subset Selection Methods

R code	Description	Example
- Note that subset of variables chosen by Best Subset Selection, Forward and Backward Stepwise Selection can be different with different models		
Best Subset Selection		
library(leaps)		
regsubsets(y ~ x, data = df)	Identifies best model for a given number of predictors, quantified by SS_{Res} - Similar syntax to *lm( )* - * indicates that the variable is included in the corresponding model	> regfit_full <- regsubsets(Salary ~ ., data = Hitters) > summary(regfit_full) By default, regsubsets() reports up to 8 best variable model – this can be adjusted: > regsubsets(Salary ~., data = Hitters, nvmax = 19)
names(reg.summary)	Summary of best subset model also returns other measures of performance	> names(reg.summary) [1] "which" "rsq" "rss" "adjr2" "cp" [6] "bic" "outmat" "obj"
reg.summary$rsq	Returns R^2 - Naturally increases as number of predictors increase, due to greater flexibility	
	Plot various measure for all models at once - Visualises results Additional: - Argument *type* = *"l"* connected plotted points with lines - *points()* adds points on a plot already created	> par(mfrow = c(2, 2)) 1) To plot SSRes: > plot(reg.summary$rss, type = "l") 2) To plot adjusted R2: > plot(reg.summary$adjr2, type = "l") To find and label the maximum point: > which.max(reg.summary$adjr2) [1] 11 > points(11, reg.summary$adjr2[11], cex = 2, pch = 20) 3) To plot Mallows' Cp: > plot(reg.summary$cp, type = "l") > which.min(reg.summary$cp) [1] 10 > points(10, reg.summary$cp, cex = 2, pch = 20) 4) To plot BIC: > plot(reg.summary$bic, type = "l") > which.min(reg.summary$bic) [1] 6 > ...

plot(regfit.full, scale = "…")	Show models ranked by performance measure using *regsubsets()*'s built-in *plot()* function - black boxes in the top row show variables in the optimal model according to the statistic	> … scale = "bic" … > … scale = "cp" …
coeff(regfit.full, n)	Extract coefficients for each predictor size - Argument *n* is the number of coefficients to extract - Gives coefficient estimates for intercept and each of the selected variables in subset	
Forward and Backward Stepwise Selection		
- Use same function *regsubsets()* but with argument *method* specified		
regsubsets(y ~ x, data = df, nvmax = n, method = "forward")	Forward stepwise	
regsubsets(y ~ x, data = df, nvmax = n, method = "backward")	Backward stepwise	
Choosing models		
- Important to use **ONLY TRAINING OBSERVATIONS** for model fitting and **variable selection**		
Validation Set Approach		
set.seed(1) train = sample(c(T, F), nrow(df), rep = T) test = !train	Split observations into training and test set	
regsubsets(y ~ x, data = df[train,], nvmax = n)	Run *regsubsets()* on **training** set - Best subsets for training data only	
test.mat = model.matrix(y ~ ., data = df[test,])	Do something similar to *predict()* which is not available 1. Create a test "X" matrix - *model.matrix()* adds a colum of "1"s for intercept term, beta_0 - Creates dummy variables for qualitative variables	

val.errors = rep(NA, n) for (i in 1:n) { coefi = coef(regfit.best, id = i) pred = test.mat[, names(coefi)] %*% coefi val.errors[i] = mean(df$y[test] – pred)**2) } which.min(val.errors)	2. Compute validation error for each of the *n* subset models (specified by *nvmax* argument) - Validation error is MSE of all observations in test set - Choose number of variables with minimum validation error	Own predict function that returns predicted values for specifief number of variables in model, *id*: > function(object, newdata, id, …) { # object is a *regsubsets()* model form = as.formula(object$call[[2]]) mat = model.matrix(form, newdata) # id is number of variables coefi = coef(object, id = id) xvars = names(coefi) return(mat[, xvars] %*% coefi) }
regsubsets(y ~ ., data = df, nvmax = n)	3. **Refit with full data set**	

Cross-validation Approach

- Each fold: 1) fit
  2) variable selection
- **DO NOT** do variable selection before k-fold CV ***

	Do k-fold cross validation: 1. Assign data to different folds	> fold_index <- sample(rep(1:k, nrow(df), nrow(df))
	2. Set up matrix to store CV error estimates	# Each fold has n errors for different number of variables > cv.errors <- matrix(NA, k, n)
	3. Put Validation Set Approach code through a loop across the k folds	> for (j in 1:k) { best.fit = regsubsets(Salary ~ ., data = Hitters[folds != j,] nvmax = 19) for (i in 1:19) { pred = predict(best.fit, Hitters[folds == j,], id = 1) cv.errors[j, i] = mean((Hitters$Salary[folds == j,] – pred)**2) } }

	4. Aggregate across all folds to see mean CV errors for models of different number of variables	# Apply by column > mean.cv.errors = apply(cv.erros, 2, mean) > which.min(mean.cv.errors) [1] 11
	5. Refit model with full data set and get coefficients for number of variables determined in Step 4.	> regsubsets(Salary ~ ., data = Hitters, nvmax = 19) > coef(reg.best, 11)

(6B) SIX B – Shrinkage Methods

R code	Description	Example
library(glmnet)	- Package written to perform elastic net on generalised linear models where *elastic net* is a hybrid version of ridge regression and lasso - Takes in quantitative response *y* and quantitative predictor matrix *x*	

x = model.matrix(
y ~ ., df)[, -1]
y = df$y
	Prepare data for *glmnet()* function as a matrix	

Ridge Regression		
grid = 10^seq(10, -2, length = 100)	Vector of lambda values used to fit the model - Argument for *lambda* in *glmnet()* - If argument left empty, *glmnet()* will automatically select a range - Example: range of lambda values from 10^{10} to 10^{-2} - To cover full range of scenarios for data	

glmnet(x, y,
alpha = 0,
lambda = grid)
	Fit a ridge regression model
	- By default, variables are standardised before fitting and returns coefficients in the original scale
	- Turn off by setting argument *standardize = F*
	- Especially when variables are already in same units	

ridge.mod$lambda [index] coef(ridge.mod)[, index]	Returns vector of RR coefficient estimates for each value of lambda - First, find lambda at specified index - Next, find coefficients for this value of lambda - Larger values of lambda, have smaller coefficients, in terms of *l2* norm	In this example, there are 100 lambda values and each value has 20 coefficient estimates for each predictor + intercept: > dim(coef(ridge.mod)) [1] 20 100 > ridge.mod$lambda[50] [1] 11497.57 > coef(ridge.mod)[, 50] (Intercept) AtBat ... 407.356050200 0.036957182 ...

predict(ridge.mod,
s = *new_lambda_value*,
type = "coefficients")
	Predict coefficients for a new value of lambda	

ridge.pred =
predict(ridge.mod,
s = *lambda,*
newx = *test_set*)
	Returns predictions for a new data set
	- Usually for test data
	- For lambda values (s) that were not specified during fitting of the model, *predict()* interpolates/extrapolates	

predict(ridge.mod,
s = lambda,
newx = test_set,
exact = T,
x = train_set,
y = train_set$y,
thresh = 1e-16)
	- Set argument *exact* = *T* to recompute exact coefficients, original *x* and *y* need to be supplied as well
	
	- Set lower threshold (algorithm runs longer) for closer approximation to coefficient estimates because *glmnet()* is a numerical fitting method	

cv.out =
cv.glmnet(x_train,
y_train,
alpha = 0,
nfolds = k)
 	
plot(cv.out)
bestlam =
log(cv.out$lambda.min)
	Choose optimal lambda using cross-validation	

LASSO		

glmnet(x[train,], y[train],
alpha = 1,
lambda = grid)
	- Fit model using train set
	- Find best lambda using CV on train set
	- Predict test set
	- Find test MSE
		> lasso.mod = glmnet(x[train,], y[train,], alpha = 1, lambda = grid)
		> plot(lasso.mod)
		
		> set.seed(1)
		> cv.out = cv.glmnet(x[train,], y[train], alpha = 1)
		> plot(cv.out)
		> bestlam = cv.out$lambda.min
		
		> lasso.pred = predict(lasso.mod, s = bestlam, newx = x[test,])
		> mean((lasso.pred – y.test)**2)

out = glmnet(x, y,
alpha = 1,
lambda = grid)
predict(out,
type = "coefficients",
s = bestlam)
	Returns coefficient estimates
	- Note that some coefficients can be exactly zero in lasso	

(6C) SIX C – Dimension Reduction Methods

R code	Description	Example
library(pls)		
Principal Components Regression		
- Attempts to maximise the amount of variance explained in predictors		

set.seed(2)
 	
pcr(y ~ .,
    data = df,
    subset = train,
    scale = T,
    validation = "CV")
	Fit a PCR on train set
	- PCR is closely related to Ridge Regression
	- *scale = T* to standardise each predictor
	- *validation = "CV"* to compute ten-fold CV error for each possible value of M	

summary(model)	View resulting fit of PCR model - CV score is sqrt(MSE) - So MSE = CV^2 - Cumulative PVE in predictors and response also provided - Amount of information about the predictors or the response that is captured using M principal components - Using all M = p will increase cumulative PVE to 100%	> summary(pcr.fit) Data: X dimension: 263 19 Y dimension: 263 1 Fit method: svdpc Number of components considered: 12 VALIDATION: RMSEP Cross-validated using 10 random segments (Intercept) 1 comps 2 comps … CV 452 348.9 … adjCV 452 348.7 … … TRAINING: % variance explained 1 comps 2 comps 3 comps … X 38.31 60.16 70.84 … Salary 40.63 41.58 42.17 …
validationplot(model, val.type = "MSEP")	Plot CV scores - MSEP against number of components - CV errors are more variable/higher variance because: 1. train set choice has much higher variance across different CV-folds 2. smaller dataset to train	

predict(model, x[test,], ncomp = M)	Estimate test error using M that gave lowest CV error	
full_model <- pcr(y ~ x, scale = T, ncomp = M) summary(full_model)	Fit PCR on **full** dataset to get final PCR coefficient estimates	
Partial Least Squares		
- No guarantee that *x, y* will be optimal (especially when n is small) - Uses correlation between *x* and *y* to find approximate direction - Searches for directions that explain variance in both predictors and response - Variance of *y* plays a huge part		
plsr(y ~ ., data = df, subset = train, scale = T)	Fit on training set	
summary(model)	Find best number of components, M	
predict(model, test_set, ncomp = M)	Find test MSE - Useful when comparing models (E.g. PCR model)	
plsr(y ~ ., data = df, scale = T, ncomp = 2)	Fit on full data set for best M components found	

(10A) TEN A – PCA, K-means

R code	Description	Example
Principal Components Analysis		
prcomp(df, scale = T)	Perform PCA - By default, variables centered to have mean = 0 - *scale* = *T* for sd = 1	

names(pca_output)	Attributes of the output of an *procomp()* object	`> names(pr.out)`
		`[ 1 ] "sdev"  "rotation"  "center"  "scale"  "x"`

Means pca_output$center **S.d.** pca_output$scale	Means and standard deviations of variables used for scaling prior to implementing PCA	
pca_output$rotation	Gives rotation matrix of size p by M/corresponding principal component **loading** vectors - If multiplied with X of size n by p, returns principal component scores/coordinates in rotated coordinate system	
pca_output$x	Returns principal component **score** vectors of size n by M - kth column is for kth PC	

pca_score <- pca_output$x
plot(
pca_score[, c(PC_num1, PC_num2)],
col = Cols(df$y))
	Plot first few PC scores to visualise data	# To assign colours to corresponding points instead of words
		`Cols = function(vec) {`
		`    cols = rainbow(length(unique(vec))`
		`    return(cols[as.numeric(as.factor(vec))])`
		`}`
		# Plot PC2 against PC1
		`> plot(pca_output$x[, 1:2])`
		# Plot PC3 against PC1
		`> plot(pca_output$x[,c(1,3)])`

biplot(pca_output, scale = 0) pca_output$rotation = -pca_output$rotation pca_output$x = -pca_output$x	Plots first 2 principal components in a **biplot** - *scale* = *0* ensures that loadings are in the right scale - PCs unique up to a sign change - Can get flipped version by flipping signs in loadings and scores	
pca_output$sdev	Returns standard deviation of each principal component - With respect to loadings	

pr_var = pca_output$sdev**2 pve = pr_var/sum(pr_var)	Compute PVE of each component - Squared of *$sdev* divided by total variance	

plot(pve,
xlab = "PC",
lab = "PVE",
ylim = c(0, 1),
type = 'b')
	Plot PVE
	- *type* = *'b'* for points to be represented as a circle and connected by lines	

plot(cumsum(pve), ...)	Plot cumulative PVE	
K-Means		
- If data has more than 2 variables, 1) Perform PCA first 2) Plot the first 2 PC score vectors		
kmeans(x, k, nstart = n)	Perform K-means clustering - Argument *nstart* can be used to specify how many initial cluster assignments to run algorithm through - Generally a good idea to run with a large value of *nstart* (e.g. 20, 50) to discard undesirable local optima - Algorithm is very efficient	
km_out$cluster	View cluster assignments for each observation by index	> km.out = kmeans(x, 2, nstart = 20) > km.out$cluster [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...

plot(x,
col= (km_out$cluster + 1),
pch = 20, cex = 2)
	Plot data, with each observation coloured according to its cluster assignment	

km_out$tot.withinss	Returns local minimum	
km_out	Returns *kmeans()* object	> km_out ... Available components: [1] "cluster" "centers" "totss" [4] "withinss" "tot.withinss" "betweenss" [7] "size" "iter" "ifault"

(10B) TEN B – Hierarchical Clustering

R code	Description		Example

hclust(dist(x),
method = "…")
 	
hclust(dist(scale(x)), …)
	Hierarchical clustering
	- EUCLIDEAN-BASED DISTANCE
	- *dist()* outputs a distance output/distance matrix
	- Can scale variables before performing clustering		

	method =		
	"complete"	Complete linkage - Default	
	"average"	Average linkage	
	"single"	Single linkage - Commonly gives a trailing cluster phenomenon	
	"centroid"	Centroid linkage	

dd <- as.dist(1-cor(t(x)))
hclust(dd, …)
	Hierarchical clustering
	- CORRELATION-BASED DISTANCE
	- convert arbitrary square symmetric correlation matrix into a form that *hclust()* can recognise as a distance matrix
	- Highly correlated/ correlation value close to 1, smaller distance
	- Correlation value close to -1, larger distance (to consider close to -1 as close, take *|cor(x)|*)
	
	- Dissimilarity function here is **(1-r)** where **r** is sample correlation
	
	- Other choices:
	1. **sqrt(1-r)**
	2. **1 - |r|**
	3. **sqrt(1 - |r|)**		

plot(hclust(…),
labels = df$y,
cex = 0.9)
	Plot dendrograme directly from *hclust()* output		

abline(
h = height_of_cut,
col = "red")
	Plot cut		

cutree(hclust(…), c)	Cut tree for a specified number of clusters - *c* is number of clusters		

hc_clusters <-
cutree(...)
table(hc_clusters,
df$y)
	View how observations were clustered against their true labels
	- Something similar to a confusion matrix
		`hc_clusters BRE CNS COL K5A K5B LEU`
		`     1       2   3   4   0   0   0`
		`     2       3   2   0   0   0   0`
		`     3       0   0   0   1   1   6`
		`     4       2   0   5   0   0   0`
		- *BRE* and *CNS* are very spread out across different clusters
		- *LEU* all clustered together

km_cluster <-
km_out$cluster
hc_cluster <-
cutree(..., k)
 	
table(km_cluster,
hc_cluster)
	Compare hierarchical cluster assignments with k-means cluster assignments	`            hc_clusters`
		`km_clusters  1   2   3   4`
		`     1      11   0   0   9`
		`     2       0   0   8   0`
		`     3       9   0   0   0`
		`     4      20   7   0   0`
		- Cluster 1 for *hc_cluster* has observations spread out across different clusters in *km_clusters*
		- Cluster 3 confirmed by both clustering methods

	Perform clustering based on first few PC score vectors - Results can be different from using full dataset - Sometimes PCA yields better results if most of signals captured in first few components **- Using PCA as a denoising pre-processing step** (get a cleaner result) **- Looking at scree-plot decides where to make the cut**	# Do for first 5 PCs > hc_out <- hclust(dist(pr_out$x[1:5])) > plot(hc_out, labels = y, main = "Hier. Clust. on First Five Score Vectors") > table(cutree(hc_out, 4), y)