**How to recode variables**

dat$Species = **as.factor**(ifelse(iris$Species=="virginica",1,0))

# to recode cleanly, you could use for instance:

dat$Species = **car::recode**(dat$Species, "0='other'; 1='virginica'")

# or:

**levels**(dat$Species) = c("other","virginica")

**Chapter 3-. Classification**

**1-. K-Nearest Neighbors (KNN)** (01\_Example Chp3 Exercise bank)

**Note: KNN strongly depends on K chosen**

* **Small K: Overfitting (low bias/large variance)**
* **Large K: Underfitting (high bias/low oiance)**

**So we need to select K to achieve a trade-off**

**Note: It is important to scale the data**

library(class) # It has function knn

**Splitting training & test (necessary for knn)**

x\_train, x\_test, y\_train, y\_test

**Passing them to knn**

**K** = alpha # Number of neighbors

ko = **knn(**x\_train, x\_test, y\_train, K)

**Note: the training variable for the knn should be numeric**

**Note: it predicts automatically (i.e. not necessary to apply function predict)**

**Confusion matrix**

**1-. Using library(caret) (Recommended)**

library(caret)

**confusionMatrix**(data = dat, reference = y.test) # they should have same levels to be compared!!

**It gives us:**

**1-.** Confusion Matrix

**2-.** Accuracy rate (also failure rate i.e. 1 – Accuracy)

**3-.** A **95% CI** for the **accuracy rate**

**4-.** Sensitivity and Specificity (using attribute $class)

**2-. Using table**

tb = table(pred, y\_test)

tb

# Calculating accuracy or failure rate

sum(diag(tb))/sum(tb)

**# How to calculate specificity**

**spec = tb[1,1]/sum(tb[,1])**

**# Sensitivity**

**sens = tb[2,2]/sum(tb[,2])**

**How to find the suitable K that maximizes the accuracy rate:**

We will perform a grid search using different values of K

Kmax = 30 (for instance)

acc = numeric(Kmax)

for(k in 1:Kmax){

# Fitting knn

ko = knn(x.train, x.test, y.train, k)

# Building table

tb = table(ko, y.test)

# Storing error rates

acc[k] = sum(diag(tb))/sum(tb)

}

# We can find **the minimum or plot them**

plot(acc, t = ‘b’, pch = …, cex = …)

or

k.best = which.max(acc)

**2-. Logistic Regression** (02\_Exercise Chp 3)

**Note: if Y is a binary classification Logistic Regression should be used as a REFERENCE (i.e. when Y has more than one class, doesn’t work well…)**

# How to code it

glm(interest ~., data = dat, subset = train, **family = binomial(link = logit)**)

**Note: We will classify in terms of a cut-off** (threshold) (remember binomial classifies successes and failures)

**Predicting with it:**

**1-. Obtaining a vector of TRUE/FALSE**

Pred\_vector = (predict(fit, newdata=data[-x\_train,],

**type = "response")** > **Threshold** )

**2-. Obtaining the probability associated with each row**

Pred\_numeric = predict(fit, newdata=dat[-is,], type="response")

**Boxplot**

We can plot a boxplot to appreaciate the overlap between classes. **Why?**

If there **is no overlap** we can ensure that these classes behave differently, and so they can be classified.

boxplot(pred\_numeric, y\_test)

**Table**

Tb = table(pred\_vector , y.test)

**We can perform a grid search to find a balance in the trade-off between sensibility and specificity (ROC CURVE ANALYSIS)**

alpha = seq(0.05, 0.95, by = 0.05)

err = numeric(length(alpha))

for (i in 1:length(alpha)){

pred.y = as.numeric(pred\_numeric>alpha[i])

tb = table(pred.y, y.test)

err[i] = (1-sum(diag(tb))/sum(tb)))

}

**3-. ROC CURVE** *(06\_exercise Chp 3 and 05\_Exercise)*

**Useful to find a trade-off between specificity and sensitivity**

**1-. If alpha is small (Highly sensitive):**

* Pi^hat is likely to be above the cut-off
* Zi^hat is likely to be 1
* Also likely to detect “false alarms”

**1-. If alpha is large (Highly specificity/not very sensitive):**

* Pi^hat is likely to be below the cut-off
* Zi^hat is likely to be 0

Table

Description automatically generated with low confidenceAhead of anything we will introduce the concepts of sensitivity and specificity in terms of the confusion matrix.

* 2x2 confusion matrix (i.e. only two classes)
* 3x3 Confusion Matrix (i.e. 3 classes)

Diagram

Description automatically generated

Recall: we can **calculate them** using

* ConfusionMatrix from caret
* Table

**# How to calculate specificity**

**spec = tb[1,1]/sum(tb[,1])**

**# Sensitivity**

**sens = tb[2,2]/sum(tb[,2])**

We will focus as well in **the Area under the curve** (ROC curve of course)

**Idea: the more we get 1 the better our model is**

**Library(pROC)**

Roc.g **= roc**(y.test, glm.p #needs probabilties here)$auc

**plot(roc.g)** # will give you the graph

# How to **plot** it

**plot**(spec, sens,

xlim = c(1,0),

type = 'b')

abline(h = 1, v = 1)

**4-. Linear Discriminant Analysis (LDA)** *(03\_Exercise Chp3)*

**Note: Natural reference technique for classification problems WITH MORE THAN 3 CLASSES (Y RESPONSE)**

**First of all we need to check the assumptions of:**

* **Normality**
* **Equal variance**

**Library(MAAS)**

*Look code in exercise 03\_Exercise Chpt3*

**How to assess normality (FOR EACH CLASS , SEPARATELY):**

**1-. Boxplot (informal):** we can assess normality here, looking if the boxplots of each class and attribute are symmetric.

**2-. Histogram (specifically and formal):** we need to address normality here as you know: paying attention to skews, symmetry, tails…

**3-. QQ-plots:** looking at substantial departures and if the fit a straight line…

**4-. Shapiro Test (do not rely on these too much):** if the p-value is large is normal and if not is not normal…

**How to assess equal variance**

**1-. Barlett’s test:** if p is large then equal variance if p is small not equal variance. We do not need to differenciate classes over here

for(j in 1:4){print( **bartlett.test**(dat[,j]~dat$Species)$p.value )}

**Note:** in this exercise in particular the assumptions were not verified but we continued fitting lda…

**Fitting LDA**

lda.o = **lda**(Species~., data=dat)

(lda.o)

Gives us:

* Probability for each class
* Group means (if numerical)
* Coefficient of linear discriminant

**How to get predict**

lda.p = predict(lda.o, newdata=dat.test)

names(lda.p)

1. **Lda.p$class**: contains the classes predicted
2. **Lda.p$posterior** : contains an array of probabilities

**Confusionmatrix**

(tb = table(**lda.p$class**, dat.test$Species))

Accuracy = sum(diag(tb))/sum(tb)

# or

confusionMatrix(**lda.p$class**, reference = dat.test[,'Species'])

**5-. Quadratic Discriminator Analysis (QDA)**

**Note: more robus than LDA when LDA assumptions fail**

# QDA:

qda.o = qda(Species~., data=dat.train)

qda.p = predict(qda.o, newdata=dat.test)

(tb = table(qda.p$class, dat.test$Species))

sum(diag(tb))/sum(tb)

# or

confusionMatrix(qda.p$class, dat.test[,'Species'])

**Chapter 4**

**6-. Decision Trees** *(01\_02\_Exercise Chp 4)*

**Note: Prunning might be requiered in some situations since trees tend to overfit (i.e. low biad large variance)**

**Note: as the trees grow more, the more overfitting you will have**

**Library(tree)**

**How to code a tree**

tree.out = tree(High~., data = dat, **subset** = i.train,

na.action = na.pass # if you need to omit Nas)

**Summary**

summary(tree.out) gives you:

1. **Misclassification** error
2. **Names of variables used**
3. Number of terminal nodes/leaf nodes

**Plot a tree**

plot(tree.out) # Will plot the edges and structure of tree

text(tree.out, pretty = 0) # will set the names on top of each split

**Pruning a tree**

**Note: we need to set a seed (if wanted) to get same results**

**Note: pruning reduces overfitting**

cv.CS = **cv.tree**(tree.out, FUN=prune.misclass)

**names**(cv.CS)

**# - size:**

# number of terminal nodes in each tree in the cost-complexity pruning sequence.

**# - deviance:**

# total deviance of each tree in the cost-complexity pruning sequence.

**# - k:**

# the value of the cost-complexity pruning parameter of each tree in the sequence.

**Plotting size vs deviance**

plot(cv.CS$size,cv.CS$dev,t='b')

abline(v = cv.CS$size[which.min(cv.CS$dev)]) # Optimal pruning

**Plotting k vs deviance**

plot(cv.CS$k,cv.CS$dev,t='b')

**How to implement the pruning** :

1. use which.min(cv.CS$dev) to get the location of the optimum

opt.size = **cv.CS$size[which.min(cv.CS$dev)]**

1. retrieve the corresponding tree size
2. pass this information on to pruning function

ptree = **prune.misclass(tree.out, best=opt.size)**

And now we could plot the ptree as explained above.

**Predicting with trees**

**We can use:**

1. Class: to get the classes
2. Vector: to get the probabilities

tree.pred = **predict**(tree.out, CS.test, type=**"class")**

tb1 = table(tree.pred,High.test)

**7-. Random Forests***(04\_Exercise Chp 4)*

**Note: random forests are important since with them we can rule out some attributes with ease. (i.e. using variable importance)**

**library(randomForest)**

**Coding a random Forest**

rf.out = **randomForest**(High~., CS)

**Predicting with a forest**

We can use:

1. **Class:** vector with predicted classes
2. **Prob:** vector with probabilities

rf.yhat = **predict**(rf.out, CS, type=**"class")**

**Note: we can get the fitted values passing the original dataframe to the function predict**

**Matrix for the OOB observations:** (it has the class error associated)

(tb.rf2 = **rf.out$confusion**)

Ex:

No Yes class.error

No 209 27 0.1144068

Yes 50 114 0.3048780

**Bagging**

**Note: Bagging can work even better than randomForests!! Remember: it decorrelates the trees (i.e. great reduction in variance , central limit theorem)**

**How to code and predict with bagging**

bag.out = **randomForest**(High~., CS**, mtry=P**)

**P:** numbers of attributes to be considered usually root(ncol())

bag.yhat = **predict**(bag.out, CS, type="class")

**Note: we can get here too the predicted values of the training set..**

**Variable importance** *(07\_Exercise Chp 4):*

**Note: we can have an overview of which variables are the most influencial for the reference (i.e. which variables should be dropped)**

rf = randomForest(High~., data = CS)

rf**$importance**

We can **plot** it

**varImpPlot(**rf, pch=15, main="Ensemble method 1"**)**

**8-. Gradient Boosting***(07\_Exercise Chp 4)*

**Watch out: it needs numerical variable as response variables!!**

**Note: we can get even better results than randomFORESTS**

**Note: the main idea is rule out useless variables**

**Library(gbm)**

**How to code it:**

gb.out **= gbm(**High~., data=CS,

**distribution="bernoulli", # use "gaussian" instead for regression**

**n.trees=5000, # size of the ensemble**

**interaction.depth=1) # depth of the trees, 1 = stumps**

**Inspect output:**

par(mar=c(4.5,6,1,1))

**summary(gb.out, las=1) Gives you variable importance**

plot(gb.out, i="Price") Graph of Price

plot(gb.out, i="ShelveLoc") Graph of ShelveLoc

**Predict with it**

gb.p **= (predict(**gb.out, newdata=CS.test, **n.trees=5000, type = “response”)) > 0.5**

**Roc Analysis**

**Note: for gbm we do not need to pass into roc function a vector of probabilties…**

roc.gb = **roc**(response=CS.test$High, **predictor**=**gb.p)**

**USING CARET**

**library(caret)**

**Coding it**

gb.out = **train**(Salary~., data=dat.train**, method='gbm'**, **distribution='bernoulli'**)

**Note: distribution = “gaussian” if we want to use regression**

**Predicting**

gb.fitted = predict(gb.out) **# corresponding fitted values**

gb.pred = predict(gb.out, dat.validation)

**Confusion Matrix**

**confusionMatrix(**reference=dat.validation$Salary, data=gb.pred,

mode="everything")