**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")

**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'])

**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**

**Note: FUN = prune.misclass – classification**

**FUN = Prune.tree - regression**

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

**Note: prune.tree for regression**

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)**

**Note: it is bases on the basis of decrease in MSE.**

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")

**9-. Support Vector Machine***(02\_Exercise Chapter 5)*

**Property:** only observations that either lie on the margin or violate it can “change” the hyperplane. It does NOT matther where the other observations lie. (i.e. SVM has a **low sensitivity to outliers/extreme values)**

**library(e1071)**

**Watch out: there can not be the presence of categorical variables. They should be recoded into a numerical variable.**

**How to recode into numerical predictors in case of presence of categorical predictors**

xm = model.matrix(y~.+0, data=x)

**Note: of course here class(y) = ‘categorical’**

**How to code it:**

svmo.lin = **svm**(xm.train, y.train, **kernel**='linear')

There can be **different kernels:**

1. **Linear:** a linear kernel
2. **Polynomial**
3. **Radial**
4. **Sigmoid**

**How to get fitted values**

svmy.lin = **fitted**(svmo.lin)

**We can perform here as usual confusion matrixes as so forth…**

table(y.train, svmy.lin)

**Note: there is a very good way of visualizing the data in 01\_Exercise Chapter 5.**

**How to obtain support vector machine points**

svmo.pol**$SV**

**How to predict:**

**In order to predict we have to add in the svm function the parameter probability = “TRUE”**

**Note: u can predict as well without parameter probability = True. It will give u the class instead so we can get the confusionMatrix**

svmo.lin = **svm**(xm.train, y.train, kernel='linear', **probability**=**TRUE**)

pred.lin = **predict**(svmo.lin, newdata=xm.test, **probability**=**TRUE**)

**How to get probabilities to build a roc curve (AUC)**

p.lin = **attributes**(pred.lin)$**probabilities**[,2]

**roc**(response=y.test, predictor=p.lin)$**auc**

**Note: you could have requested the ConfusionMatrix using caret that has specificity and sensitivity**

**Tunning svm**

svm.tune = **e1071::tune**(svm, train.x=x.train, train.y=y.train,

kernel='radial',

**ranges=list(cost=10^(-2:2),**

**gamma=c(0.5,1,1.5,2)))**

bp = svm.tune$**best.parameters** **# gives you the best parameters in the whole grid, best performance and so forth**

**Note: it might happen that the svm tuned is worse than the actual svm**

**Using caret**

svmR.out = **caret::train**(Salary~., data=dat.train, **method**="svmRadial")

**Method can be:**

1. **svmRadial**
2. **svmLinear**
3. **rf (randomForest)**

**And we can do all described above**

**SVM for regression**

# specify statistical training settings:

ctrl = **caret::trainControl**(method='cv')

# perform statistical training:

svm.o = **caret::train**(y~., data=dat.train, method="svmLinear",

**trControl=ctrl**)

**Note: remember use MSE OR RMSE not classification loss functions**

**10-. Neural Networks***(02\_Exercise Chapter 6)*

* **Note: the neural networks need a really large amount of observations and it takes time be executed**
* **Note: NNs are very sensitive to scaling, because they rely on gradient descent.**
* **Note: scaling the feature set may help reducing loss function more gradually and regularly. It reduces the risk of large model gradient descent, which may in turn help mantaining appropiate weights and gradient steps magnitudes**
* **Note: we need numeric variables for both libraries**

There are **two libraries that we can use**

**Library(nnet)**

**Library(neuralnet)**

**How to code it**

out.nn.tanh = **neuralnet**(f, data=Boston, hidden=c(10), rep=5,

linear.output=FALSE, **act.fct**='tanh',

**threshold =** alpha)

nno1 = **nnet**(medv~., data=dat.train, size=5, **linout**=TRUE,

**decay =** alpha)

**Note: nnet has fitted values summary(nnol$fitted.values)**

**Decay: parameter for weight decay. Default 0.**

**Threshold: a numeric value specifying the threshold for the partial derivatives of the error function as stopping criteria.**

**Note: f is a formula in this case it is**

**medv ~ crim + zn + indus + chas + nox + rm + age + dis + rad +**

**tax + ptratio + black + lstat**

There are a **few actation functions:**

1. **sigmoid**
2. **ReLu**
3. **Tanh**
4. **Logistic**

**How to predict with it:**

p1 = **predict**(out.nn, newdata=Boston)

**Note: same for nnet and neuralnet**

**Plotting a neural net**

Plot(out.nn)

**To use regression:**

nno = nnet(Salary~., data=dat.train, size=10, decay=c(0.1), **linout=1**)

**Variable importance**

**library(NeuralNetTools)**

**How to code it:**

nno = **nnet**(Species~., data=dat, size=c(7), linout=FALSE, entropy=TRUE)

**# One for each class**

vimp.setosa = **olden**(nno, out\_var='setosa', bar\_plot=FALSE)

vimp.virginica = **olden**(nno, out\_var='virginica', bar\_plot=FALSE)

vimp.versicolor = **olden**(nno, out\_var='versicolor', bar\_plot=FALSE)

**# Plotting variable importance**

plot(**olden**(nno, out\_var='setosa'))

plot(**olden**(nno, out\_var='virginica'))

plot(**olden**(nno, out\_var='versicolor'))

**We can acce0ss their values using**

vimp.setosa$**importance**

**Note: so we can combine all the variable importance in one single dataframe**

**Note: there is a good graph to implement in the exercise 06\_exercise chapter 6**

**11-. Model/Feature Selection***(Chapter 7)*

**Library(leaps)**

**Stepwise selection**

**How to code it**

reg.full = **regsubsets**(Salary~., data=Hitters, **method**="exhaustive",

**nvmax =** model\_size,

**subset =** itrain # if needed)

Where that **method can be:**

* **Exhaustive:** looks all the possible combinations of variables
* **Forward:** adds variables in each step
* **Backward:** removes one variable in each step
* **Seqrep**

**Summary(reg.full)** # easy to interpret the variables here

**Plotting RSS vs Number of covariates (nº of variables in the model)**

RSS = **summary(reg.full)$rss**

plot(RSS, pch=20, t='b', xlab="Number of covariates", ylab='RSS')

**Plotting Adjusted R sqaured vs number of covariates**

R2adj = **summary(reg.full)$adjr2**

plot(R2adj, pch=20, t='b', xlab="Number of covariates",

ylab='Adjusted R^2')

**Getting best model in terms of R2adj**

R2adj.index = which.max(R2adj)

summary(reg.full)$**outmat**[R2adj.index,] # or

which(summary(reg.bwd**)$which**[R2adj.index,-1]

**Plot a heat map**

plot(reg.full, **scale="adjr2")**

**Getting nth covariate index variable (ie a model with n variables)**

coef(reg.fwd, **id**=4) # or

names(which(summary(reg.full)$which[4,]==TRUE))

**fit and predict using stats::step()**

step.bth = **step**(lm.out, **direction**="both")

These **direction can be:**

1. **Both**
2. **Forward**
3. **Backward**

**Easier to predict with step**

pred = **predict**(reg.fwd, newdata=dat[-itrain,])

**Recursive feature elimination using Caret**

subsets <- c(1:5, 10, 15, 20, ncol(features)) # size of set of variables to be tested

ctrl <- **rfeControl**(**functions = rfFuncs**,

method = "cv",

number = 10,

# method = "repeatedcv",

# repeats = 5,

verbose = FALSE)

These **function can be:**

1. **rfFunc:** randomForest
2. **lmFunc:** linear model

rf.rfe <- **rfe**(x.train, y.train,

**sizes = subsets,**

**rfeControl = ctrl**)