Machine learning in R

Trainee sessie 02

Longhow Lam

Contents

---

1 Predictive modeling technieken

In R kan je veel verschillende predictive modellen fitten. We behandelen een paar in deze sessie. Lineaire regressie met de functie `lm`, logistische regressie met de functie `glm`, decision trees met de functie `rpart` en ensemble van trees met `ranger` en `xgboost`. Ook `h2o` zullen we kort aanstippen. Ik zal deze functies apart behandelen maar we zullen later in de sessie zien met het package `mlr` hoe je op een meer uniforme manier meerdere modellen kan proberen op een data set.

1.1 lineare regressie

We beginnen met simpele lineaire regressie, bruikbaar voor voorspel modellen waar de Target variable continu (numeric) is. We nemen als voorbeeld huizen prijs data die ik gescraped heb van jaap.nl. We willen de prijs van een huis voorspellen basis van een aantal input variabelen/kenmerken.

```
jaap = readRDS("data/Jaap.RDs")

library(ggplot2)
ggplot(jaap, aes( kamers, prijs)) + geom_point()
ggplot(jaap, aes( Oppervlakte, prijs)) + geom_point()

# some obvious outliers
jaap = jaap %>% filter( prijs < 1e7 )
```

Een tweetal simpele modellen.

```
modelout  = lm( prijs ~ kamers               , data = jaap)
modelout2 = lm( prijs ~ kamers + Oppervlakte , data = jaap)

modelout
modelout2
```

Modeling functies in R retourneren objecten met van alles er nog wat in. De functie `lm` levert een object af van de klasse lm.

```
class(modelout)
names(modelout)
modelout$coefficients

summary(modelout)
plot(modelout)
```

Iets mooiere diagnostische plots uit `lm` objecten krijg je met de library `ggfortify` die weet hoe je lm objecten moet interpreteren voor ggplot. Dan kan je met de functie `autoplot` uit ggplot mooiere diagnostische plots maken.

```
library(ggfortify)
library(ggplot2)

ggplot2::autoplot(modelout)
```

Je ziet dat er wat outliers in de data zitten, die kunnen we nog eens er uit filteren

```
jaap = jaap %>% filter( prijs < 1500000 )
modelout  = lm( prijs ~ kamers               , data = jaap)
modelout2 = lm( prijs ~ kamers + Oppervlakte , data = jaap)
```

1.1.1 formula objects

Modellen in R kan je specificeren met zogenaamde de formula objects. Hieronder zie je een aantal voorbeelden.

```
## gebruik eerste cijfer van postcode als een locatie variabele
jaap = jaap %>%
  mutate(PC1Positie = stringr::str_sub(PC,1,1)) %>%
  filter(!is.na(PC1Positie))
```

```
f0 = prijs ~ Oppervlakte + kamers
m0 = lm(f0, data = jaap)
summary(m0)

f1 = prijs ~ Oppervlakte + kamers + PC1Positie
m1 = lm(f1, data = jaap)
summary(m1)

## interactie termen
f2 = prijs ~ Oppervlakte + kamers + PC1Positie + Oppervlakte*PC1Positie
m2 = lm(f2, data = jaap)
summary(m2)

## interactie termen
f3 = prijs ~ Oppervlakte + kamers + PC1Positie + Oppervlakte*PC1Positie +  Oppervlakte*kamers
m3 = lm(f3, data = jaap)
summary(m3)

##  interactietermen
f4 = prijs ~ Oppervlakte*kamers*PC1Positie
m4 = lm(f4, data = jaap)
summary(m4)
```

Als je verschillende model objecten hebt gemaakt kan je de functie `anova` gebruiken om ze met elkaar te vergelijken. Dit gebeurt met behulp van F statistics.

```
anova(m0, m1, m2, m3, m4)
```

Nog een paar voorbeelden van formule objecten.

```
##  termen weglaten
f5 = prijs ~ Oppervlakte*kamers*PC1Positie - Oppervlakte:kamers:PC1Positie
m5 = lm(f5, data = jaap)
summary(m5)

## een target en de rest van de variabelen als inputs
f6 = prijs ~ . -PC6 -PC
m6 = lm(f6, data = jaap)
summary(m6)
```

1.1.2 Buckets / linear constant en splines

Als een input variable niet linear is m.b.t. de target kan je deze niet-lineariteit modelleren met buckets (linear constante stukken) of met splines.

```
library(ggplot2)
library(dplyr)
library(splines)

## In een scatterplot kan je wellicht enige vorm van niet lineariteit zijn.
jaap %>%
  filter(prijs < 1000000, Oppervlakte < 1500) %>%
  ggplot(aes(x=Oppervlakte, y= prijs)) +
  geom_point()
```

```
mybreaks = seq(0,1000, by = 25)
jaap = jaap %>% mutate(OppervlakteBucket = cut(Oppervlakte, breaks = mybreaks))


m1 = lm(prijs ~ Oppervlakte, data = jaap)
m2 = lm(prijs ~ OppervlakteBucket, data = jaap)
m3 = lm(prijs ~ ns(Oppervlakte,6), data = jaap)

summary(m1)
summary(m2)
summary(m3)
```

1.1.3 predicties

Met de functie `predict` kunnen we nieuwe huizen scoren, dat wil zeggen de prijs van andere huizen die niet in de training data set zaten voorspellen.

```
NieuweHuizen = data.frame(
  Oppervlakte = seq(20, 250, l = 100)
  ) %>%
  mutate(
   OppervlakteBucket = cut(Oppervlakte, breaks = mybreaks)
  )

# Bucket predicties
prijs2 = predict(m2, newdata = NieuweHuizen)
NieuweHuizen$prijs2 = prijs2

ggplot(NieuweHuizen, aes(x=Oppervlakte, y = prijs2)) + geom_line()

# Spline predicties
prijs3 = predict(m3, newdata = NieuweHuizen)
NieuweHuizen$prijs3 = prijs3


ggplot(NieuweHuizen, aes(x=Oppervlakte, y = prijs2)) + geom_point() + geom_point(aes(y=prijs3), col=2)
```

1.2 Splitsen in train en test

Het is gebruikelijk om een data set random te splitsen in een train en test set. Op de train set wordt een predictive model getraind. En het model dat we getraind hebben testen we op de test set.

We gebruiken hier een copy van de titanic set omdat we de data iets wijzigen.

```
perc = 0.80

## maak een categorische kolom van survived
myTitan = titanic::titanic_train
myTitan = myTitan %>% mutate(
  Survived = ifelse(Survived < 1, "N", "Y") %>% as.factor
)

## haal missende waarden weg, we gaan ons hier even niet vermoeien met missende waarden :-)
```

```
myTitan = myTitan %>%
  filter(
    !is.na(Age)
  )

N = dim(myTitan)[1]
train = sample(1:N, size = floor(perc*N))

TTrain = myTitan[train,]
TTest = myTitan[-train,]
```

Dus we hebben nu een train en test set en we zien een verhouding van survived die verschillend zijn in train en test omdat we hier redelijk kleine data setjes hebben.

```
table(TTrain$Survived)
table(TTest$Survived)
```

Bovenstaande code was op de oude manier in R een train en test set maken, er is zijn packages die dat voor je doen, een van die packages is `rsample`. Dit packages is veel algemener en kan ook gebruikt worden voor cross validation splits.

```
library(rsample)
myTitanSplit = initial_split(myTitan, prop = 0.8)
TTrain = training(myTitanSplit)
TTest = testing(myTitanSplit)
```

1.3 logistic regression

Een logistic regression is een van de simpelste predictive modellen om mee te beginnen als je classificatie wilt doen. We gaan uit van een binaire Target (Y /N). We gebruiken de `TTrain` data set die we zojuist gemaakt hebben om een model te fitten.

```
out.glm = glm(Survived ~ Sex + Age + Pclass,  data = TTrain , family = binomial)
summary(out.glm)
```

1.4 decision tree

Een decision tree genereert op basis van een algoritme regels die je kan gebruiken om te classificeren. Het is een eenvoudig algoritme dat per variabele kijkt hoe deze te gebruiken om de data set in twee stukken te splitsen (kan ook meer, maar gebruikelijk is twee).

```
tree.out = rpart(Survived ~ Sex + Age +Pclass, data = TTrain)

plot(tree.out)
text(tree.out, use.n = TRUE)

fancyRpartPlot(tree.out)

### larger trees with complexity parameter
tree.out2 = rpart(Survived ~ Sex + Age +Pclass, data = TTrain, control = list(cp=0.0005))
fancyRpartPlot(tree.out2)
```

Met visNetwork kan je nog een mooiere interactieve decison tree maken.

```
library(visNetwork)
visTree(tree.out, height = "800px", nodesPopSize = TRUE, minNodeSize = 10, maxNodeSize = 30)
```

1.5 random forest met ranger

Een random forest is een zogenaamde ensemble model. Het is de combinatie van (veel) verschillende decision trees. In R kan je met verschillende packages random forests fitten. Het package `ranger` is hier een van.

```
ranger.out = ranger( Survived ~ Sex + Age + Pclass, data = TTrain , probability = TRUE)
ranger.out
```

1.6 xgboost

Extreme gradient boosting wordt de laatste tijd ook veel gebruikt in Kaggle competities. Zoals bij random forests is een xgboost model ook een ensemble van decision trees, maar de trees zijn nu niet onafhankelijk van elkaar. Eerst wordt een tree gefit, daarna een andere op basis van de eerste, etc.

Met de library `xgboost` kan je in R extreme gradient boosting modellen fitten. De aanroep is anders dan wat we tot nu toe gezien hebben. De `xgboost` functie moet een matrix met input variabelen worden meegegeven.

```
library(Matrix)
Titan_Inputmatrix = sparse.model.matrix( Survived ~ Sex + Age + Pclass, data = TTrain)

## hoe ziet zo'n input matrix er uit? eerste 15 rijen
Titan_Inputmatrix[1:15,]

## nu kan je de xgboost aanroepen met input matrix en label
xgboost.out = xgboost(Titan_Inputmatrix, label = TTrain$Survived, nrounds = 25)
```

Bovenstaande aanroep fit een regression tree, dat is niet wat we willen we willen een binairy classificatie, label moet dan wel numeriek 0 / 1 zijn.

```
param = list(
  objective = 'binary:logistic',
  eval_metric = 'auc'
)

xgboost.out2 = xgboost(
  params=param,
  Titan_Inputmatrix,
  label = as.integer(TTrain$Survived) -1 , nrounds = 25)
```

1.7 predictie en validatie

Met een test set kan je bepalen hoe goed een model is. Gebruik het model object van een modelfit om een test set te scoren en de scores met de ware uitkomsten te vergelijken.

1.7.1 predicties

Voor binaire classificaties is het handig om response kansen uit te rekenen. Voor logistische regressie met `glm` gebeurt dit niet automatisch, we zetten extra `type = response`

```
pred_GLM = predict(out.glm, data = TTest, type='response')
hist(pred_GLM)
```

Voorspelling van de decision tree, en random forest ranger.

```
## LET OP! predieccties van tree zitten in een matrix
pred_tree = predict(tree.out, newdata = TTest)
pred_tree[1:10,]
hist(pred_tree[,2])

## LET OP hier is argument data ipv newdata en je krijgt een lijst terug
pred_ranger = predict(ranger.out, data = TTest)
pred_ranger$predictions[1:10,]
hist(pred_ranger$predictions[,2])
```

En voor xgboost moet je ook de test set als matrix veranderen

```
Titan_Testmatrix = sparse.model.matrix( Survived ~ Sex + Age +Pclass, data = TTest)
pred_xgboost = predict(xgboost.out2, newdata = Titan_Testmatrix)

## hier zitten de predicties een vector
hist(pred_xgboost)
```

1.7.2 Variable importance in trees

Als je een tree of ensemble van trees hebt getrained kan je een idee krijgen welke variabelen in het model belangrijk zijn geweest in het trainings proces. Laten we voor de tree modellen die we hierboven getraind hebben de variable importance zien.

```
# enkele decision tree
tree.out$variable.importance

# ranger random forest
ranger.out$variable.importance

# xgboost
xgb.importance( colnames(Titan_Inputmatrix), model =xgboost.out2)
```

1.7.3 Lift percentages

Als we geen model hebben kan je de overall survival kans uitrekenen (op de test set:

```
## if you know nothing :-)
TTest = TTest %>% mutate(target = ifelse(Survived == "Y",1,0))
TTest %>%  summarise(target = mean(target))
```

Als we een goed model hebben zullen de survival kansen hoger liggen voor mensen met een hogere score. Laten we de GLM (logistische regressie model) predcities in tien stukken (decielen) opdelen, en per deciel de survival kans uitrekenen.

```
testpr = predict(out.glm, newdata = TTest, type='response')
TTest$predictieGLM = testpr

### deel de GLM predicties op in tien even grote stukken
TTest = TTest %>%
```

```
  mutate(
    percPredictie = cut(
      predictieGLM,
      breaks = quantile(
        predictieGLM,
        probs = (0:10)/10
      )
    )
  )

## if you have a score!!!
TTest %>%
  group_by(percPredictie) %>%
  summarise(
    N = n(),
    target = mean(target)
  )
```

1.7.4 roc curves and hit rates

```
rocResultTEST = roc(Survived  ~ predictieGLM, data = TTest , auc=TRUE, ci =TRUE)
plot(rocResultTEST)

## HITRATES
TTest %>%
  ggplot(aes(predictieGLM, target))  +
  geom_smooth() +
  geom_abline(slope = 1,intercept = 0) +
  ggtitle("survived rate rate on test set") + scale_y_continuous(limits=c(0,1))
```

2 The h2o package

H2O is een schaalbaar machine learning platform die je vanuit R kan bedienen. Het bevat veel machine learning algoritmes, en voor grotere sets waar gewoon R moeite mee heeft kan h2o een uitkomst bieden. H2o heeft een eigen 'executie engine' geschreven in java. Bij het opstarten van h2o vanuit R wordt dan ook een apart h2o proces opgestart waar je data vanuit R naar toe moet uploaden om daar de algoritmes op los te laten.

Als je h2o opstart is er ook een eigen GUI, daar kan je naar toe localhost:54321 (standard 54321 port).

```
library(h2o)

# initialiseer h2o via R

#h2o.init(nthreads=-1, port=54323, startH2O = FALSE)
h2o.init()

### upload een R data set naar h2o: titanic train en test voorbeeldje
myTitan = myTitan %>% mutate_if(is.character, as.factor)
TTrain = TTrain %>% mutate_if(is.character, as.factor)
```

```
TTest = TTest %>% mutate_if(is.character, as.factor)
TTrain$Survived = as.factor(TTrain$Survived)
TTest$Survived = as.factor(TTest$Survived)

myTitan.h2o = as.h2o(myTitan)
ttrain.h2o = as.h2o(TTrain)
ttest.h2o = as.h2o(TTest)
### Je kan ook direct text files inlezen in h2o met
#h2o.importFile(path="C:\een file.txt", sep=",")

### welke files zijn er in h2o
h2o.ls()
```

Er zijn diverse modellen die je kan trainen, we zullen hier een aantal laten zien, neural netwerks, boosting en random forests.

```
## model op titanic
NNmodel = h2o.deeplearning(
  x = c(3,5:6),
  y = "Survived",
  training_frame  = ttrain.h2o,
  validation_frame = ttest.h2o,
  hidden = 5,
  epochs = 250,
  variable_importances = TRUE
)

show(NNmodel)
h2o.varimp(NNmodel)
```

```
GBMmodel = h2o.gbm(
  x = c(3,5:6),
  y = "Survived",
  training_frame  = ttrain.h2o,
  validation_frame = ttest.h2o
  )
GBMmodel

RFmodel = h2o.randomForest(
  x = c(3,5:6),
  y = "Survived",
  training_frame  = ttrain.h2o,
  validation_frame = ttest.h2o
  )
RFmodel

h2o.varimp_plot(RFmodel)
```

Grid search in h2o. Je kan makkelijk modellen fine-tunen, in een grid kan je verschillende waarden van hyperparameters proberen.

```
RFmodelGrid = h2o.grid(
  "randomForest",
  x = c(3,5:6),
  y = "Survived",
```

```
  training_frame  = ttrain.h2o,
  validation_frame = ttest.h2o,
  hyper_params = list(
    ntrees =c(50,100),
    mtries = c(1,2,3)
  )
)

#overzicht van het grid, gesorteerd op logloss
RFmodelGrid
```

2.1 h2o automl

h2o automl maakt het je helemaal makkelijk. . . ..

```
lb = h2o.automl(
    x = c(3,5:6),
  y = "Survived",
  training_frame  = ttrain.h2o,
  validation_frame = ttest.h2o,
  max_runtime_secs = 30
)

lb

lb@leaderboard

lb@leader
```

Geef resources terug door h2o af te sluiten als je het niet meer nodig hebt.

```
h2o.shutdown(prompt = FALSE)
```

3 The mlr package

Met het `mlr` package kan je makkelijk verschillende modellen trainen en testen op een meer uniforme manier. In R hebben alle machine learning technieken net weer verschillende aanroepen en de uitkomst is vaak een object met net steeds weer andere componenten. Dit zagen we bijvoorbeeld in de bovenstaande code voor ranger en xgboost.

Met het `mlr` package kan je dit uniform stroomlijnen. Het maken van een predictive model (welk model dan ook) bestaat altijd uit een aantal stappen. Bijvoorbeeld:

- specificeren van de target,
- specificeren van inputs,
- specificeren van variabelen die je niet wilt gebruiken,
- splitsen data,
- het model / algoritme

Deze zijn te beschrijven en uit te voeren in mlr.

We gebruiken de `titanic` data set als test in `mlr` en doorlopen een aantal stappen on een aantal modellen te benchmarken. De modellen die we willen benchmarken zijn:

- neuraal netwerk,
- gradient boosting,
- random forest
- xgboost
- decision tree,
- logistic regression via glmnet

3.1 specificeren van technieken en hun opties

In mlr kan je een aantal algemene parameters weergeven.

```
## parameters die geen beschrijving hebben willen we ook kunnen opgeven
configureMlr(on.par.without.desc = "warn")

## we kijken naar maximaal dertig variabelen
n.importance = 30

## voorspel type, we willen kansen uitrekenen
ptype = "prob"

## aantal crossvalidation splitsingen
N_CV_iter = 10
```

Naast algemene parameters, heeft elk model bepaalde parameters die je kan zetten. Dit hoeft niet, dan worden default waarden gekozen.

```
parameters_rf = list(
  num.trees  = 500
)

parameters_rpart = list(
  cp=0.0001
)

parameters_glmnet = list(
  alpha  = 1
)

parameters_NN = list(
  hidden = c(15,15)
)

parameters_xgboost = list(
  nrounds  = 5,
  max.depth = 7
)
```

Maak nu een Lijst van modelen (ook wel learners genomed) die je wilt trainen op je data.

```
RF_Learner = makeLearner(
  "classif.ranger",
  predict.type = ptype,
  par.vals = parameters_rf
)
```

```
xgboost_Learner = makeLearner(
  "classif.xgboost",
  predict.type = ptype,
  par.vals = parameters_xgboost
)

rpart_Learner = makeLearner(
  "classif.rpart",
  predict.type = ptype,
  par.vals = parameters_rpart
)

binomial_Learner = makeLearner(
  "classif.binomial",
  predict.type = ptype
)

glmnet_Learner = makeLearner(
  "classif.cvglmnet",
  predict.type = ptype,
  par.vals = parameters_glmnet
)

h2ogbm_Learner = makeLearner(
  "classif.h2o.gbm",
  predict.type = "prob"
)

h2oNN_Learner = makeLearner(
  "classif.h2o.deeplearning",
  predict.type = ptype,
  par.vals = parameters_NN
)

## lijst van de learners
learners = list(
  rpart_Learner,
  RF_Learner,
  binomial_Learner,
  glmnet_Learner,
  h2ogbm_Learner,
  h2oNN_Learner
)
```

Als je categorische variabelen in je voorspel model wilt gebruiken eist het mlr package dat ze `factor` zijn. En in het geval van een classificatie probleem moet de target variabele ook een factor variabele zijn.

```
ABT = titanic_train
ABT$Target = as.factor(ifelse(ABT$Survived < 1, "N", "Y"))
ABT = ABT %>% mutate_if(is.character, as.factor)
```

3.2 Imputeren van missende waarden

Als er missende waarden zijn kan je mlr deze laten imputeren door een bepaalde waarde.

```
impObject = impute(
  ABT,
  classes = list(
    integer = imputeMean(),
    numeric = imputeMean(),
    factor = imputeMode()
    ),
  dummy.classes = "integer"
)

ABT = impObject$data
```

3.3 Het aanmaken van een task

Maak nu een 'task' aan waarin je de data, de inputs en de target specificeert. Een classificatie taak voor categorische target of een regressie task voor een numerieke target.

Wat wil je modelleren: Kans op level "Y", dan dien je positive = "Y" op te geven. Bij een binair target met Y en N levels wordt namelijk standaard "N"gebruik (alfabetisch)

```
classify.task = makeClassifTask(id = "Titanic", data = ABT, target = "Target", positive = "Y")

## Overzicht van de taak en kolom informatie

print(classify.task)
getTaskDescription(classify.task)
summarizeColumns(classify.task)
```

3.4 Variablen hard uitsluiten

Soms zijn er variabelen die je niet wilt meenemen in je model. Deze kan je hard uitsluiten.

```
vars.to.drop = c("Name", "Survived", "Ticket")

classify.task = dropFeatures(classify.task, vars.to.drop )

## Weghalen van (bijna) constante variabelen

## Je kan ook 'bijna' constante variabelen weghalen: perc iets hoger zetten
classify.task = removeConstantFeatures(classify.task, perc = 0.01)
classify.task
```

Zeldzame levels van factors samenvoegen. Het is gebruikelijk om zeldzame levels te verwijderen of te mergen

```
classify.task = mergeSmallFactorLevels (classify.task, min.perc = 0.02,  new.level = ".merged")
summarizeColumns(classify.task)
```

Welke features hebben een effect op de target? Je kan de predictive power meten per input variable. Univariate, dus per feature kan je met mlr de relatie met de target berekenen.

Onderliggend heb je Rweka en Rjava dingen nodig daar kan je op linux wat issues mee krijgen. Dingen de je kan doen:

- run in een shell: `sudo R CMD javareconf` en doe
- sudo rstudio-server stop
- export LD_LIBRARY_PATH=/usr/lib/jvm/jre/lib/amd64:/usr/lib/jvm/jre/lib/amd64/default
- sudo rstudio-server start

Zie ook stack overflow

```
## Feature selection
fv = generateFilterValuesData(classify.task,  method = c("information.gain", "chi.squared"))

## display en plot importance

importance = fv$data %>% arrange(desc(information.gain))
head(importance, n = n.importance)
plotFilterValues(fv, n.show = 2*n.importance)
```

laat nog eens variabelen weg die helemaal niks doen.

```
vars.to.drop = c("PassengerId", "Parch", "SibSp")
classify.task = dropFeatures(classify.task, vars.to.drop )
```

3.5 Sample schema

Met mlr kan je data splitsen, niet alleen in train / test maar ook cross validation. Dit heet een sample schema.

```
SampleStrageyHO = makeResampleDesc("Holdout", split=0.75)
SampleStrageyCV = makeResampleDesc("CV", iters = N_CV_iter)
```

3.6 uitvoeren machine learning becnhamrk

Nu heb je de diverse stappen gespecificeerd en kan je een benchmark uitvoeren voor de verschillende learners,

```
br1 = mlr::benchmark(learners, classify.task, SampleStrageyHO, measures = list(mlr::mmce, mlr::auc, mlr
```

3.7 Vergelijking machine learning modellen

Na het trainen van de modellen met mlr heb je een zogenaamde benchmark object, die kan je printen en plotten om wat meer info te krijgen.

```
data.frame(br1) %>% arrange(desc(auc))
plotBMRSummary(br1, measure = mlr::auc)
```

3.7.1 ROC curves

In het benchmark object zit eigenlijk nog veel meer data. Met onderstaande code pluk je alle stukjes data per model uit om deze vervolgens in een ROC grafiek te zetten.

```
NModels = length(br1$results$Titanic)
for(i in 1:NModels)
{
```

```
  tmp2  = br1$results$Titanic[[i]]$pred$data
  rocResultTEST = roc(truth  ~ prob.Y, data = tmp2 )
  if(i==1)
  {
    plot(rocResultTEST, col=i)
  }else{
    plot(rocResultTEST, col=i, add=TRUE)
  }
}

legend( 0.6,0.6, names(br1$results$Titanic), col=1:NModels,lwd=2)
title("Titanic model")
```

3.7.2 model gebruiken om te scoren

Als je een benchmark hebt gedaan heb je al de getrainde modellen in het benchmark object zitten. Die kan je al gebruiken om een data set te scoren. je dient dit model er wel 'eerst uit te halen'.

```
## haal model er uit
titanmodel = br1$results$Titanic$classif.ranger$models[[1]]

## dit zijn de feauteures in het model
FT = titanmodel$features

## Maak even een score set van de ABT met alleen de features
ScoreSet = ABT[, FT]

outpredict = predict(titanmodel, newdata = ScoreSet)
outpredict
```

4 Unsupervised learning

De bovenstaande code was gericht op predictive modeling, ook wel supervised learning genoemd: met input variabelen een target variable proberen te voorspellen. In deze sectie zullen we een tweetal technieken laten zien waar geen target variabele is, ook wel unsupervised learning genoemd.

4.1 k-means Clustering

Dit is een van de bekendste clustering methode. Je dient een aantal clusters van te voren op te geven, de k, het algoritme gaat dan elke observatie aan een van de k clusters toekennen.

```
mycars = mtcars %>% select (mpg, wt)
cars.cluster = kmeans(mycars, 5)
cars.cluster

# in het ouput object zit informatie over het gefitte kmeans algoritme
mycars$cluster = cars.cluster$cluster
mycars
```

```
plot(mycars$mpg, mycars$wt, col = cars.cluster$cluster )
points(cars.cluster$centers, col = 1:5, pch = 8, cex = 2)
```

Met het `h2o` package kan je ook k-means clustering doen, dit is niet alleen sneller maar kan ook meteen factor variabelen aan, in de `kmeans` van R kan dat niet. Start indien nodig h2o.

```
library(h2o)
#h2o.init(nthreads=-1, port=54323, startH2O = FALSE)
h2o.init()
```

Breng data naar h2o, we gebruiken nu de sample data set mtcars in R maar we maken nog 1 extra factor kolom aan.

```
# am is de transimssie: 0 is automat en 1 is handgeschakeld, is eig
mycars = mtcars %>% mutate(am = as.factor(am))
cars.h2o = as.h2o(mycars)
```

Laat het algoritme zelf bepalen hoeveel clusters er in de data zijn.

```
cars_clustering = h2o.kmeans(cars.h2o,  k = 10, estimate_k = TRUE)
cars_clustering
```

na het trainen heb je een h2o cluster object met diverse informatie

```
cars_clustering@model
h2o.cluster_sizes(cars_clustering)
h2o.centers(cars_clustering)

## met h2o.predict kan je data scoren: bepalen tot welk cluster een observatie hoort en weer terug naar
cluster_membership = h2o.predict(
  cars_clustering,
  newdata = cars.h2o
  ) %>%
  as.data.frame()
```

4.2 DBSCAN

Density-based Spatial Clustering of Applications with Noise (DBSCAN), which does not make assumptions about spherical clusters like k-means, nor does it partition the dataset into hierarchies that require a manual cut-off point. As its name implies, density-based clustering assigns cluster labels based on dense regions of points.

```
library(dbscan)

## wat data, halve manen, hoeveel clusrers zijn dit?
x = runif(100,0,pi)
y = sin(x) + rnorm(100,0,0.1)

x1 = runif(100,-pi/2,pi/2)
y1 = 0.4+ sin(x1-pi/2) + rnorm(100,0,0.1)

plot(c(x,x1),c(y,y1))
```

k-means werkt niet echt hier....

```
df  = data.frame(x = c(x,x1), y = c(y,y1))
df.cluster = kmeans(df, 2)
```

```
df.cluster

plot(df$x, df$y, col = df.cluster$cluster )
points(df.cluster$centers, col = 1:5, pch = 8, cex = 2)
```

De data moet in een matrix zitten om dbscan te kunnen runnen.

```
X = cbind(c(x,x1),c(y,y1))
db <- dbscan(X, eps = .4, minPts = 4)
db

pairs(X, col = db$cluster + 1L)

### NOISE POINTS....
X = cbind(c(x,x1,-1, 3),c(y,y1, 1, -1))
db <- dbscan(X, eps = .4, minPts = 4)
db

pairs(X, col = db$cluster + 1L)

## local outlier factor score
pairs(X, col = db$cluster + 1L)
lof <- lof(X, k = 4)
pairs(X, cex = lof)
```

5 Market basket analyse

Met market basket analyse (ook wel association rules mining genoemd) kan je uit "transacties van klanten" vaak voorkomende combinaties of juiste hele "sterke combinaties" van producten bepalen. Hieronder volgt een voorbeeldje op een fictief grocery (boodschappen transacties) data setje.

```
library(arules)

## De meest simpele transactionele data set
trxDF = readRDS("data/boodschappen.RDs")

## Transormeer naar een transaction object
Groceries = as(
  split(
    trxDF$item,
    trxDF$id
    ),
  "transactions"
)

Groceries

## Visuele Item informatie
itemFrequencyPlot(Groceries, topN = 35, cex.names = 0.75)
```

Nu je de boodschappen als transaction object hebt kan je er market basket rules op los laten met behulp van het a-priori algoritme.

```
rules <- apriori(Groceries, parameter = list(supp = 0.001, conf = 0.8))
rules

## laat enkele regels zien
inspect(rules[1:10])

inspect( sort(rules, by = "support")[1:10])

## converteer de rules set naar een data frame
rulesDF = DATAFRAME(rules)
```

Nu je de regels hebt kan je filteren op regels. Welke regels bevatten bepaalde producten.

```
rules.subset = subset(rules, lhs %in% c("cereals", "curd"))
rules.subset
inspect(head(rules.subset, n=15))
```

Of als iemand een bepaalde reeks transacties heeft welke regels horen daar bij en welk product kan je dan aanraden.

```
PersoonA = data.frame(
  id = rep(1,3),
  item2 = c("butter","curd","domestic eggs")
)

trxs_trans = as(
  split(
    PersoonA$item2,
    PersoonA$id
    ),
  "transactions"
)
inspect(trxs_trans)

rulesMatch <- is.subset(rules@lhs,trxs_trans)

## er zijn meerdere regels, je zou degene met de hoogste lift kunnen kiezen
inspect(rules[rulesMatch[,1]])
inspect(rules[rulesMatch[,1]]@rhs)
```

Een ander manier om regels weer te geven is in een network graph, de verzameling regels vormen in feite een netwerk. A –> B, B –> C, D –> B bijvoorbeeld.

```
library(arulesViz)
plot(head(sort(rules, by = "lift"), n=50), method = "graph", control=list(cex=.8))
```

5.1 interactive MBA graphs

You can visualise rules in interactive plotly plots or interactive visNetwork plots. First, an interactive scatter plot of the rules can be made. Each rule is plotted as a point, where the x axis represents the support and the y axis represent the confidence of the rule.

```
rules <- apriori(Groceries, parameter = list(supp = 0.001, conf = 0.8) )
rulesDF = rules %>% DATAFRAME()
```

```
library(plotly)
plotly_arules(rules, max = 2000)
plotly_arules(rules, method = "two-key plot")
```

Secondly, an interactive visNetwork can be created. We need to extract the nodes and edges from the rules object.

```
library(visNetwork)

rules <- apriori(
  Groceries,
  parameter = list(
    supp = 0.0001,
    conf = 0.1,
    minlen = 2,
    maxlen=2
    )
  )

rulesDF = head(
  sort(rules, by = "lift"),
  n=250
  ) %>%
  DATAFRAME() %>%
  mutate(
    from = as.character(LHS),
    to = as.character(RHS),
    value = lift
  )

nodes = data.frame(
  id = base::unique(c(rulesDF$from, rulesDF$to)),
  stringsAsFactors = FALSE
) %>% mutate(
  title = id
)

visNetwork(nodes, rulesDF) %>%
   visOptions(highlightNearest = TRUE,  nodesIdSelection = TRUE) %>%
   visEdges(smooth = FALSE)
```

6 Deeplearning

In R kan je deeplearning modellen trainen met Keras met een Tensorflow back-end, we geven hier een simpel voorbeeld. Je kan vanuit het `keras` package tensorflow installeren. Zie ook mijn slides op slideshare voor wat verdere uitleg en bekijk dit 'fantastische' boek: deep learning in R

```
library(keras)
## De library keras bevat een makkelijke functie om de tensorflow backend  installeren,
## dit werkt makkelijk op een linux machine en is niet echt ondersteunt op een windows machine!

# keras::install_keras()
```

6.1 Een simpel model

We maken met keras eerst een simpel model, 1 hidden layer, niks fancy nog. Download en prepareer eerst de data.

```
batch_size <- 128
num_classes <- 10
epochs <- 10

# The data, shuffled and split between train and test sets
c(c(x_train, y_train), c(x_test, y_test)) %<-% dataset_mnist()

dim(x_train)

# We hebben 60.000 plaatjes die 28 bij 28 matrices zijn, bekijk het eerste plaatje
x_train[1,,]

# en de bijbehorende label
y_train[1]
```

Zo'n 28 bij 28 plaatje is ook te plotten

```
# bekijk plaatje n
n = 110
image(  x_train[n,,] )
y_train[n]
```

We gaan nu de data wat reshapen voor het model

```
# Sla de matrices plat tot een vector (array) van 784 = (28*28)
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))

dim(x_train)
x_train[1,]

# Transform RGB values into [0,1] range
x_train <- x_train / 255
x_test <- x_test / 255

cat(nrow(x_train), 'train samples\n')
cat(nrow(x_test), 'test samples\n')

# Convert class vectors to binary class matrices
y_train <- to_categorical(y_train, num_classes)
y_test <- to_categorical(y_test, num_classes)
```

Defineer het model.

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(784)) %>%
  layer_dense(units = 10, activation = 'softmax')

summary(model)

model %>% compile(
```

```
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
```

Train en evalueer het model.

```
# Fit model to data
history <- model %>% fit(
  x_train, y_train,
  batch_size = batch_size,
  epochs = epochs,
  verbose = 2,
  view_metrics = FALSE,
  validation_split = 0.2
)

plot(history)

score <- model %>% evaluate(
  x_test, y_test,
  verbose = 0
)

# Output metrics
cat('Test loss:', score[[1]], '\n')
cat('Test accuracy:', score[[2]], '\n')
```

We kunnen nu nog een tweede hidden layer toevoegen met drop outs om overfitten te voorkomen.

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 512, activation = 'relu', input_shape = c(784)) %>%
  layer_dropout(rate=0.2) %>%
  layer_dense(units = 512, activation = 'relu', input_shape = c(784)) %>%
  layer_dropout(rate=0.2) %>%
  layer_dense(units = 10, activation = 'softmax')

summary(model)

model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
```

6.2 Convolutional model.

Maak nu een meer fancy convolutional deeplearning model, input is nu een array van getallen ipv een plat geslagen vector zoals in vorige voorbeeld. En laten we nu de Fashion MNIST nemen, een leuk alternative voor de cijfers, het zijn kledingplaatjes door zalando beschikbaar gesteld.

Fashion MNIST

Data kan je krijgen door de git repo te clonen. en in data dir zitten gz files die je moet uitpakken, dit zijn

binaire files die je mer R kan inlezen, gebruikmakend van de functie `readBin`.

Er zijn 60.000 train en 10.000 test plaatjes, elk plaatje heeft een label (0-9)

Label Description 0 T-shirt/top 1 Trouser 2 Pullover 3 Dress 4 Coat 5 Sandal 6 Shirt 7 Sneaker 8 Bag 9 Ankle boot

```
### MNIST FASHION helper functie
load_mnist <- function() {
  load_image_file <- function(filename) {
    ret = list()
    f = file(filename,'rb')
    readBin(f,'integer',n=1,size=4,endian='big')
    ret$n = readBin(f,'integer',n=1,size=4,endian='big')
    nrow = readBin(f,'integer',n=1,size=4,endian='big')
    ncol = readBin(f,'integer',n=1,size=4,endian='big')
    x = readBin(f,'integer',n=ret$n*nrow*ncol,size=1,signed=F)
    ret$x = matrix(x, ncol=nrow*ncol, byrow=T)
    close(f)
    ret
  }

    load_label_file <- function(filename) {
    f = file(filename,'rb')
    readBin(f,'integer',n=1,size=4,endian='big')
    n = readBin(f,'integer',n=1,size=4,endian='big')
    y = readBin(f,'integer',n=n,size=1,signed=F)
    close(f)
    y
  }

  train <<- load_image_file('/home/longhowlam/datasets/fashion/train-images-idx3-ubyte')
  test <<- load_image_file('/home/longhowlam/datasets/fashion/t10k-images-idx3-ubyte')

  train$y <<- load_label_file('/home/longhowlam/datasets/fashion/train-labels-idx1-ubyte')
  test$y <<- load_label_file('/home/longhowlam/datasets/fashion/t10k-labels-idx1-ubyte')
}
```

Nu kan je de data inlezen

```
### Helper functie om een digit (een zalando plaatje) te laten zien
show_digit <- function(arr784, col=gray(12:1/12), ...) {
  image(matrix(arr784, nrow=28)[,28:1], col=col, ...)
}

load_mnist()

# vier plaatjes
show_digit(train$x[1816,])
train$y[1816]

show_digit(train$x[116,])
train$y[116]

show_digit(train$x[3116,])
train$y[3116]
```

```
show_digit(train$x[1316,])
train$y[1316]
```

Nu kan je het model trainen, ik heb alleen 1 epoch gezet, dit model duurt een uur op dit laptopje om te trainen. Eerst weer de data goed zetten en wat algemene settings, zoals batch size, dimensie en epochs specificeren....

```
batch_size <- 128
num_classes <- 10
epochs <- 1

# input image dimensions
img_rows <- 28
img_cols <- 28

# the data, shuffled and split between train and test sets
x_train <- train$x
y_train <- train$y
x_test <- test$x
y_test <- test$y

dim(x_train) <- c(nrow(x_train), img_rows, img_cols, 1)
dim(x_test) <- c(nrow(x_test), img_rows, img_cols, 1)
input_shape <- c(img_rows, img_cols, 1)

x_train <- x_train / 255
x_test <- x_test / 255

cat('x_train_shape:', dim(x_train), '\n')
cat(nrow(x_train), 'train samples\n')
cat(nrow(x_test), 'test samples\n')

# convert class vectors to binary class matrices
y_train <- to_categorical(y_train, num_classes)
y_test <- to_categorical(y_test, num_classes)
```

Maak in keras nu het meer complexere convolutional model, verschillende lagen en pas ook dropout toe.

```
# define a two layer conv with max pooling  model
model <- keras_model_sequential()
model %>%
  layer_conv_2d(
    filters     = 32,
    kernel_size = c(3,3),
    activation  = 'relu',
    input_shape = input_shape
  ) %>%
  layer_conv_2d(
    filters     = 64,
    kernel_size = c(3,3),
    activation  = 'relu'
  ) %>%
  layer_max_pooling_2d(
    pool_size = c(2, 2)
  ) %>%
```

```
  layer_dropout(rate = 0.25) %>%
  layer_conv_2d(
    filters     = 64,
    kernel_size = c(3,3),
    activation  = 'relu'
  ) %>%
  layer_max_pooling_2d(
    pool_size = c(2, 2)
  ) %>%
  layer_flatten() %>%
  layer_dense(
    units      = 256,
    activation = 'relu'
  ) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(
    units = num_classes,
    activation = 'softmax'
  )
```

Bekijk de beknopte smanevatting van het model

```
summary(model)
```

Compileer en train het model

```
# compile model
model %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)

# train and evaluate
model %>% fit(
  x_train, y_train,
  batch_size = batch_size,
  epochs = epochs,
  verbose = 1,
  validation_data = list(x_test, y_test)
)

scores <- model %>% evaluate(
  x_test, y_test, verbose = 0
)

cat('Test loss:', scores[[1]], '\n')
cat('Test accuracy:', scores[[2]], '\n')
```

Einde Sessie