Blatt07

Burr, Luebeck, Ott

22.06.2020

Aufgabe 7.1:

```
library(RSNNS)
```

```
## Loading required package: Rcpp
```

```
library(DiceKriging)
library(plotly)
```

```
## Loading required package: ggplot2

##
## Attaching package: 'plotly'

## The following object is masked from 'package:ggplot2':
##
##     last_plot

## The following object is masked from 'package:stats':
##
##     filter

## The following object is masked from 'package:graphics':
##
##     layout
```

```
library(gtools)
library("checkmate")
```

```
##
## Attaching package: 'checkmate'

## The following object is masked from 'package:gtools':
##
##     assert

## The following object is masked from 'package:DiceKriging':
##
##     checkNames
```

```
library("ggplot2")
library("ggrepel")
library("ParamHelpers")
library("mlr")
```

```
## 'mlr' is in maintenance mode since July 2019. Future development
## efforts will go into its successor 'mlr3' (<https://mlr3.mlr-org.com>).

##
## Attaching package: 'mlr'

## The following object is masked from 'package:RSNNS':
##
##     train
```

```
library("checkmate")
library("smoof")
library("mlrMBO")
library("Rcpp")
library("lhs")

load("po20_blatt07_data.RData")

fx = function(x) x**2 - 10*cos(5*x)
fxy = function(x,y) x**2 *(1+cos(y))+ y**2 * sin(x)

set.seed(2020)
```

Berechne alle drei Metamodelle fuer die Funktion 1.

Erstelle zunaechst ein polynomielles Modell vom Grad $k = 2$:

```
fit1 = lm(value ~ I(x^1) + I(x^2), data = data1)
```

Nun erstelle die Vorhersagen:

```
pred_fit1 = predict.lm(fit1, newdata = grid1)
```

Fahre fort mit dem RBF-Netz:

```
rbf1 = rbf(x = data1[, 1], y = data1[, 2], size = 5)
```

Die zugehoerigen Vorhersagen sind:

```
pred_rbf1 = predict(rbf1, newdata = grid1)
```

Erstelle das letzte Metamodell mithilfe von Kriging:

```
krig1 = km(value ~., design = data.frame(x = data1[, 1]),
           response = data1[, 2], covtype = "exp", multistart = 1,
           nugget = 0.2,
           control = list(trace = FALSE))
```

Die Vorhersagen hierzu sind:

```
pred_krig1 = predict(krig1, newdata = grid1, type = "SK")
```

Um die jeweiligen Modelle zu evaluieren, werden im Folgenden die mittleren quadratischen Abweichungen sowie die mittleren absoluten Abweichungen zwischen den Vorhersagen und den wahren Werten berechnet.

```
mse = function(pred, true) mean((pred - true)^2)

mad = function(pred, true) mean(abs(pred - true))


fit_mse = mse(pred_fit1, true1)
fit_mad = mad(pred_fit1, true1)
rbf_mse = mse(pred_rbf1, true1)
rbf_mad = mad(pred_rbf1, true1)
krig_mse = mse(pred_krig1$mean, true1)
krig_mad = mad(pred_krig1$mean, true1)

res1 = data.frame(fit = c(fit_mse, fit_mad), rbf = c(rbf_mse, rbf_mad),
                  krig = c(krig_mse, krig_mad))
rownames(res1) = c("mse", "mad")
res1
```

```
##           fit      rbf     krig
## mse 18.018943 14.96789 17.12633
## mad  3.253527  3.31688  3.09640
```

Zu sehen ist, dass das polynomielle Modell die schlechtesten Resultate auf den Daten liefert. Das Kriging-Modell weist ebenfalls einen schlechten MSE auf. Nur das RBF-Netz weist den geringsten MSE auf. Somit schein das RBF-Netz auf den Daten1 am besten zu funktionieren. Interessanterweise unterscheiden sich die MADs der drei Modelle unwesentlich voneinander.

Visualisierung der Modelle fuer die erste Funktion sowie der wahren Zielfkt.

```
fig1 = plot_ly() %>%
    add_trace(data = data.frame(x = grid1[, 1], y = true1), x = ~x, y = ~y,
              name = "Zielfunktion", type = "scatter", mode = "lines")
fig1 = fig1 %>% add_trace(data = data.frame(x = grid1[, 1], y= pred_fit1), x = ~x, y = ~y,
              name = "Polynomielles Modell (k=2)", type = "scatter",
              mode = "lines")
fig1 = fig1 %>% add_trace(data = data.frame(x = grid1[, 1], y = pred_rbf1), x = ~x, y = ~y,
              name = "RBF-Netz", type = "scatter", mode = "lines")
fig1 = fig1 %>% add_trace(data=data.frame(x = grid1[, 1], y = pred_krig1$mean), x = ~x,
              y = ~y, name = "Kriging", type = "scatter", mode = "lines")
fig1 = fig1 %>% add_trace(data = data1, x = ~x, y = ~ value, name = "Samples", type = "scatter",
              mode = "markers")
fig1 = fig1 %>% layout(title = "Metamodelle fuer Funktion 1")
```

Anhand der Grafik ist zu erkennen, dass das polynomielle Modell einen linearen Trend schaetzt; dies wird aber insbesondere auf die wenigen Datenpunkte im Datensatz zurueckzufuehren sein. Auch das Kriging-Modell weist einen linearen Trend auf, wobei hier die Datenpunkte (exakt) interpoliert werden. Eventuell liegt hierbei eine Art Overfitting dar, da eben die Datenpunkte interpoliert werden und deswegen die Testdaten falsch vorhergesagt werden. Das RBF-Netz weist im Gegensatz dazu einzig eine kurvenfoermige Funktion auf.

Berechne alle drei Metamodelle fuer die Funktion 2.

Erstelle zunaechst ein polynomielles Modell vom Grad $k = 2$:

```
fit2 = lm(value ~ x + c + I(x * c) + I(x^2) + I(c^2), data = data2)
```

Nun erstelle die Vorhersagen:

```
pred_fit2 = predict.lm(fit2, newdata = grid2)
```

Fahre fort mit dem RBF-Netz:

```
rbf2 = rbf(x = data2[, 1:2], y = data2[, 3], size = 15)
```

Die zugehoerigen Vorhersagen sind:

```
pred_rbf2 = predict(rbf2, newdata = grid2)
```

Erstelle das letzte Metamodell mithilfe von Kriging:

```
krig2 = km(value ~ x + c + I(x * c) + I(x^2) + I(c^2),
           design = data.frame(data2[, 1:2]),
           response = data2[, 3], covtype = "matern5_2", multistart = 2, nugget = 0.95,
           control = list(trace = FALSE))
```

```
## Warning: executing %dopar% sequentially: no parallel backend registered
```

Die Vorhersagen hierzu sind:

```
pred_krig2 = predict(krig2, newdata = grid2, type = "SK")
```

Evaluierung der Metamodelle:

```
fit_mse = mse(pred_fit2, true2)
fit_mad = mad(pred_fit2, true2)
rbf_mse = mse(pred_rbf2, true2)
rbf_mad = mad(pred_rbf2, true2)
krig_mse = mse(pred_krig2$mean, true2)
krig_mad = mad(pred_krig2$mean, true2)

res2 = data.frame(fit = c(fit_mse, fit_mad), rbf = c(rbf_mse, rbf_mad),
                  krig = c(krig_mse, krig_mad))
rownames(res2) = c("mse", "mad")
res2
```

```
##            fit        rbf      krig
## mse 148.126500 2036.27594 14.487878
## mad   9.907646   28.89599  2.177457
```

Nach der Evaluierung der Metamodelle faellt auf, dass das Kriging-Modell mit Abstand die Testdaten am besten vorhersagen kann, da hier der niedrigste MSE berechnet wird. Am zweitbesten schneidet das polynomielle Modell auf und am schlechtesten ist dieses mal das RBF-Netz. Nach dem MSE hat dieses Verfahren die groessten Schwieirgkeiten auf der gegeben Datenstruktur.

Visualisierung:

```
fig2 = plot_ly() %>%
   add_trace(data = data.frame(x = grid2[, 1], c = grid2[, 2], z = true2),
             x= ~ x, y = ~c, z = ~z,
             type = "mesh3d", opacity = 0.5, name = "Zielfunktion")
 fig2 = fig2 %>% add_trace(data = data.frame(x = grid2[, 1], c = grid2[, 2], z = pred_fit2),
             x = ~x, y = ~c, z = ~z, type = "mesh3d",
             name = "Polynomielles Modell (k=2)")
 fig2 = fig2 %>%  add_trace(data = data.frame(x = grid2[, 1], c = grid2[, 2], z = pred_rbf2),
             x = ~x, y = ~c, z = ~z,
             type = "mesh3d", name = "RBF-Netz")
 fig2 = fig2 %>%  add_trace(data = data.frame(x = grid2[, 1], c = grid2[, 2], z = pred_krig2$mean),
             x = ~x, y = ~c, z = ~z,
             type = "mesh3d", name = "Kriging")
 fig2 = fig2 %>% add_trace(data = data2, x = ~x, y = ~ value, name = "Samples", type = "scatter",
               mode = "markers")
 fig2 = fig2 %>% layout(title = "Metamodelle fuer Funktion 2")

#Das Speichern beider Grafiken in einer html wollte leider nicht funktionieren...
```

Zu sehen ist, dass die wahre Zielfunktion eine Dreiecksform besitzt. Wie die MSE Werte bereits verwiesen haben, naehrt das Kriging-Modell die wahre Zielfkt am besten an, da sie ebenfalls das “Dach” abbilden kann. Das polynomielle Modell bildet zwar das Dach nicht wirklich gut ab, dafuer aber den Rest. Insbesondere faellt auf, dass das RBF-Netz schlecht abliefert, da es die wahre Zielfunktion kaum darstellen kann und eine ganz andere Struktur aufweist.

Berechne alle drei Metamodelle fuer die Funktion 3.

Erstelle zunaechst ein polynomielles Modell vom Grad $k = 2$:

```
fit3 = lm(value ~ x1 + x2 +c1 + c2 + I(x1*x2) + I(x1 * c1) + I(x1 * c2) +
                    I(x2 * c1) + I(x2 * c2) + I(c1 * c2) + I(x1^2) + I(x2^2) +
                    I(c1^2) + I(c2^2), data = data3)
```

Nun erstelle die Vorhersagen:

```
pred_fit3 = predict.lm(fit3, newdata = grid3)
```

Fahre fort mit dem RBF-Netz:

```
rbf3 = rbf(x = data.frame(x1 = data3$x1, x2 = data3$x2, c1 = data3$c1,
                          c2 = data3$c2), y = data3$value, size = 18)
```

Die zugehoerigen Vorhersagen sind:

```
pred_rbf3 = predict(rbf3, newdata = grid3)
```

Erstelle das letzte Metamodell mithilfe von Kriging:

```
krig3 = km(value ~ x1 + x2 + c1 + c2 + I(x1 * x2) + I(x1 * c1) + I(x1 * c2) +
                   I(x2 * c1) + I(x2 * c2) + I(c1 * c2) + I(x1^2) + I(x2^2) +
                   I(c1^2) + I(c2^2), design = data.frame(x1 = data3$x1,
                   x2 = data3$x2, c1 = data3$c1, c2 = data3$c2),
                   response = data3$value, covtype = "powexp", nugget = 0.3,
                   nugget.estim = FALSE, multistart = 3,
                   control = list(trace = FALSE))
```

Die Vorhersagen hierzu sind:

```
pred_krig3 = predict(krig3, newdata = grid3, type = "SK")
```

```
fit_mse = mse(pred_fit3, true3)
fit_mad = mad(pred_fit3, true3)
rbf_mse = mse(pred_rbf3, true3)
rbf_mad = mad(pred_rbf3, true3)
krig_mse = mse(pred_krig3$mean, true3)
krig_mad = mad(pred_krig3$mean, true3)

res3 = data.frame(fit = c(fit_mse, fit_mad), rbf = c(rbf_mse, rbf_mad),
                  krig = c(krig_mse, krig_mad))
rownames(res3) = c("mse", "mad")
res3
```

```
##           fit        rbf      krig
## mse 267.25376 2380.14687 23.131953
## mad  12.60647   38.49093  3.185373
```

Ganz analog zur zweiten Funktion sehen auch hier die Ergebnisse aus: Am besten kann das Kriging-Modell die Testdaten vorhersagen, (weiter) gefolgt vom polynomiellen Modell. Auch hier hat das RBF-Netz grosse Schwierigkeiten die Daten gut anzupassen, was am sehr großen MSE ersichtlich wird. Eine Problemloesung waere, mehr Hyperparameter zu tunen, was aber aufgrund der Laufzeit nur ueber ein Rechencluster moeglich waere. DIe Daten scheinen zu komplex zu sein, als das ein “kleines” Tuning hier ausreichen wuerde.

Tuning der Hyperparameter des RBF-Netzes:

Beim RBF-Netz sollen die Anzal der Neuronen sowie die Architektur des Netzes optimiert werden. Hierfuer wird zur Validierung die Kreuzvalidierung verwendet, die die Daten in k etwa gleichgrosse Gruppen aufteilt. Dabei werden jeweils $k-1$ Gruppen als Lerndatensatz für das RBF-Netz verwendet und die k-te Gruppe stellt jeweils die Testdaten dar. Um die Ergebnisse zu evaluieren wird der MSE berechnet und zum Schluss wird der gemittelte MSE ueber alle k Gruppen zurueckgegeben.

```r
kCV_rbf = function(data, target = "value", k, size) {
  Ziel = which(names(data) == target)
  Index = sample(nrow(data))
  gruppen_gr = floor(nrow(data)/k) # Gruppengroessen
  letzte_grup = nrow(data) - (k - 1) * gruppen_gr # letzte Gruppe ggf. groesser
  end = c(seq(gruppen_gr, (k - 1) * gruppen_gr, gruppen_gr), nrow(data)) # Endindizes der  Gruppen
  daten = data[Index,]
  hilfsfunc = function(lauf){
    if(lauf == 1) index1 = 1
    else index1 = end[lauf - 1] + 1
    train = daten[-(index1:end[lauf]), -Ziel]
    train_target = daten[-(index1:end[lauf]), Ziel]
    test = daten[index1:end[lauf],]
    test_target = daten[index1:end[lauf], Ziel]
    rbf_mod = rbf(x = train, y = train_target, size = size)
    pred = predict(rbf_mod, newdata = test[, -Ziel])
    mse = mean((pred - test_target)^2)
    return(mse)
  }
  erg = mean(sapply(1:k, hilfsfunc), na.rm = TRUE)
  return(erg)
}
```

Die folgende Funktion stellt die eigentliche Tuning-Funktion dar. Hierbei werden alle moeglichen Kombinationen zwischen den Hyperparameter erstellt und mithilfe der Kreuzvalidierung evaluiert. Das Modell mit dem kleinsten MSE wird als Resultat uebergeben. Das endgueltige Ergebnis des Tunings ist daraufhin der Parameter size.

```r
tune_rbf = function(data, k, max_layers, min_neuron, max_neuron){
  best_size = NULL
  best_mse = as.double("Inf")
  neuron = seq(min_neuron, max_neuron)

  for(i in seq_len(max_layers)) {
    arch = permutations(length(neuron), i, v = neuron, repeats.allowed = TRUE)
    for(j in seq_len(length(neuron)^i)) {
      tmp_size = arch[j, ]
      tmp_mse = kCV_rbf(data, "value", size = tmp_size, k = k)

      if(tmp_mse < best_mse){
        best_size = tmp_size
        best_mse = tmp_mse
      }
    }
  }
  return(list(size = best_size))
}
```

Fuer den ersten Datensatz werden 11 Gruppen gebildet, was dem Resampling-Verfahren leave-one-out entspricht. Dies ist darauf zurueckzufuehren, dass data1 nur 11 Datenpunkte beinhaltet und so gewaehrleistet wird, dass keine wertvollen Infos verloren gehen. Weiterhin werden maximal 10 Neuronen gewaehlt, da es nur 11 Daten gibt. Zudem wird das Tuning 10 mal wiederholt, da die Ergebnisse stochastisch sind. Um Laufzeit zu sparen, ist der folgende Code auskommentiert; das Ergebnis des Tunings steht jeweils im Chunck daneben.

```
# rbf_opt1 = median(replicate(10, tune_rbf(data = data1, k = 11,
#                                          max_layers = 1, min_neuron = 2,
#                                          max_neuron = 10))$size)
#
# rbf_opt1 #5
```

Analog geht dies auch fuer die Funktionen 2 und 3.

```
# rbf_opt2 = median(replicate(3, tune_rbf(data = data2, k = 10,
#                                          max_layers = 2, min_neuron = 2,
#                                          max_neuron = 20))$size)
#
# rbf_opt2 #15
```

```
# rbf_opt3 = median(replicate(3, tune_rbf(data = data3, k = 20,
#                                          max_layers = 2, min_neuron = 2,
#                                          max_neuron = 20))$size)
#
# rbf_opt3 #18
```

Tuning der Hyperparameter des Krigings:

Analog zum RBF-Netz wird auch fuer das Kriging-Modell eine Kreuzvalidierung berechnet. Die zu optimierenden Hyperparameter sind hier multistart, covtype und nugget. Allgemein wirdein Nugget-Effekt verwendet, um ggf. Overfitting entgegenzuwirken. Durch ein zusaetzliches Rauschen wird eine Ueberanpassung entgegen gesteurt.

```
kCV_krig = function(data, target = "value", k, formula_string, multistart, covtype, coln, nugget) {
  Ziel = which(names(data) == target)
  Index = sample(nrow(data))
  gruppen_gr = floor(nrow(data)/k) # Gruppengroessen
  letzte_grup = nrow(data) - (k - 1) * gruppen_gr # letzte Gruppe ggf. groesser
  end = c(seq(gruppen_gr, (k - 1) * gruppen_gr, gruppen_gr), nrow(data)) # Endindizes der  Gruppen
  daten = data[Index,]
  hilfsfunc = function(lauf){
    if(lauf == 1) index1 = 1
    else index1 = end[lauf - 1] + 1
    train = daten[-(index1:end[lauf]), -Ziel]
    train = data.frame(train)
    colnames(train) = coln
    train_target = data.frame(daten[-(index1:end[lauf]), Ziel])
    test = daten[index1:end[lauf], -Ziel]
    test_target = daten[index1:end[lauf], Ziel]
    km_mod = km(as.formula(formula_string), design = train,
                response = train_target,
                covtype = covtype, nugget = nugget, nugget.estim = F,
                multistart = multistart,
                control = list(trace = FALSE))
    pred = predict(km_mod, newdata = test, type = "SK")
    mse = mean((pred$mean - test_target)^2)
    return(mse)
```

```
  }
  erg = mean(sapply(1:k, hilfsfunc), na.rm = TRUE)
  return(erg)
}
```

In der folgenden Funktion findet das Tuning statt. Auch wird fuer jedes moegliche Modell, was aus den Hyperparameter erstellt werden kann, ein Modell gerechnet und das mit dem geringsten MSE wird uebernommen.

```
tune_krig = function(data, formula_string = "~1", k,
                     multistart_val = 1:10, coln,
                     nugget_val = seq(.05, 1, .05)){
    best_covtype = NULL
    best_nugget = NULL
    best_multistart = NULL
    best_mse = as.double("INF")
    for(tmp_type in c("gauss", "matern5_2", "matern3_2", "exp", "powexp")) {
      for(tmp_nugget in nugget_val) {
        for(tmp_multistart in multistart_val) {
          tmp_mse = kCV_krig(data, "value", k = k, formula_string, multistart = tmp_multistart,
                             covtype = tmp_type, coln = coln, nugget = tmp_nugget)
          if(tmp_mse < best_mse) {
            best_mse = tmp_mse
            best_covtype = tmp_type
            best_nugget = tmp_nugget
            best_multistart = tmp_multistart
          }
        }
      }
    }
    return(list(covtype = best_covtype, multistart = best_multistart,
                nugget = best_nugget))
}
```

```
# krig_opt1 = suppressWarnings(tune_krig(data = data1,
#    formula_string = "value ~ x", k = 11, multistart_val = 1:5, coln = "x"))
# krig_opt1 #exp, 1, 0.2
```

```
# krig_opt2 = suppressWarnings(tune_krig(data = data2,
#                            formula_string = "value ~ x + c + I(x*c) + I(x^2) + I(c^2)",
#                            multistart_val = 1:5, k = 10, coln = c("x", "c")))
#
# krig_opt2 #"matern5_2, 2, 0.95"
```

```
# krig_opt3 = suppressWarnings(tune_krig(data = data3,
#                              formula_string = "value ~ x1 + x2 + c1 + c2 + I(x1 * x2) + I(x1 * c1) + I
#                              k = 20, multistart_val = 1:5,
#                               col = c("x1", "x2", "c1", "c2")))
#
# krig_opt3 #"powexp", 3, 0.3
```

Aufgabe 7.2

MBO findet Anwendung, wenn aus unterschiedlichen Gruenden die Auswertung der Zielfunktion sehr teuer oder gar unmoeglich ist. Stattdessen, versucht man sie durch ein Metamodell zu approximieren, welches im Laufe der Optimierung zielgerichtet immer genauer wird, indem man "interessante", also "vielversprechende" Bereiche identifiziert. Man beginnt mit einem initialen Design. Dies koennte man durch voellig zufaellige Punkte oder durch ein Raster verwirklichen. Die intelligentere Variante ist das Latin Hypercupe Design, das einen sinnvollen Kompromiss zwischen Abdeckung und Zufaelligkeit darstellt. An den gewaehlten Stellen wertet man die Zielfunktion aus. Auf Basis dieser Punkte wird nun ein Metamodell M geschaetzt. Durch eine Modelloptimierung werden "vielversprechende" Stellen identifiziert, an denen eine weitere Funktionsauswertung lohnend erscheint. Diese werden zum bisherigen Design hinzugefuegt. Das wird wiederholt, bis ein Stoppkriterium, zum Beispiel ein erschoepftes Budget, erreicht ist. Das Model und das beste Design werden zurueckggegeben.

In der Spezifikation des Algorithmus, muss man sich ue¼berlegen, wie man "vielversprechend" umsetzen kann. Zu diesem Zwecke ist es wichtig, dass das Metamodell in der Lage ist, Unsicherheit zu beziffern. Bei "Lower Confidence Bound" waehlt man einen Hyperparameter κ, der den "Optimismus, dass man sich noch verbessern kann" repraesentiert. κ multipliziert man mit der Unsicherheit an jeder Stelle und zieht das Ganze vom geschaetzten Wert ab. Bei "Expected Improvement" macht man eine Normalverteilungsannahme und schaetzt so die erwartete Verbesserung an jeder Stelle. Bei beiden Varianten waehlt man dann die Stelle des niedrigsten, also besten, zu erwartetenden Wert.

Initiales Design per Latin Hypercube

```
set.seed(7.2)
lhsa = improvedLHS(5,1) #5 Punkte im 1D
lhsb = improvedLHS(10,2) # 10 Punkte in 2D

#Skalieren:
lhsa_sc = (lhsa * 20) -10
lhsb_sc = (lhsb * 20) - c(10,10)

#In DF benennen
dfa = data.frame(lhsa_sc)
colnames(dfa) = "x"

dfb=data.frame(lhsb_sc)
colnames(dfb)= c("x1","x2")
```

Funtion a)

```
obj.fun = makeSingleObjectiveFunction(
  fn = function(x) x**2 - 10*cos(5*x),
  par.set = makeNumericParamSet(id="x", lower=-10,upper=10, len=1)
)
```

Expected Improvement

```
surr.km = makeLearner("regr.km", predict.type="se", covtype="matern3_2",
                      control=list(trace=FALSE))
# so wird laut Dokumentation ein Krieging Meta-Modell erstellt
control = makeMBOControl()
control = setMBOControlTermination(control, iters=10)
control = setMBOControlInfill(control, crit=makeMBOInfillCritEI())

run_EI_a = mbo(obj.fun, design=dfa, learner = surr.km, control=control, show.info = FALSE)
```

EI: Recommended parameters: x=1.24 Objective: y = -8.431

Lower Confidence Bound

Hier haben wir das Infill-Kriterium gewaehlt, bei dem κ automatisch gewaehlt wird.

```
control_CB = makeMBOControl()
control_CB = setMBOControlTermination(control_CB, iters=10)
control_CB = setMBOControlInfill(control_CB, crit=makeMBOInfillCritCB())
# "Confidence Bound with lambda automatically chosen"
run_CB_a = mbo(obj.fun, design=dfa, learner = surr.km, control=control_CB, show.info = FALSE)
```

LCB: Recommended parameters: x=1.24 Objective: y = -8.431

Beide Infill-Kriterien kommen zum gleichen Ergebnis.

Funtion b)

```
obj.fxy = makeSingleObjectiveFunction(
  fn = function(x) x[1]**2 *(1+cos(x[2]))+ x[2]**2 * sin(x[1]),
  par.set = makeNumericParamSet(id="x", lower=-10,upper=10, len=2)
)
```

Expected Improvement

```
ctrl_EI = makeMBOControl()
ctrl_EI = setMBOControlTermination(ctrl_EI, iters=15)
ctrl_EI = setMBOControlInfill(ctrl_EI, crit=makeMBOInfillCritEI())
run_EI_b = mbo(obj.fxy, design=dfb, learner = surr.km, control=ctrl_EI, show.info = FALSE)
```

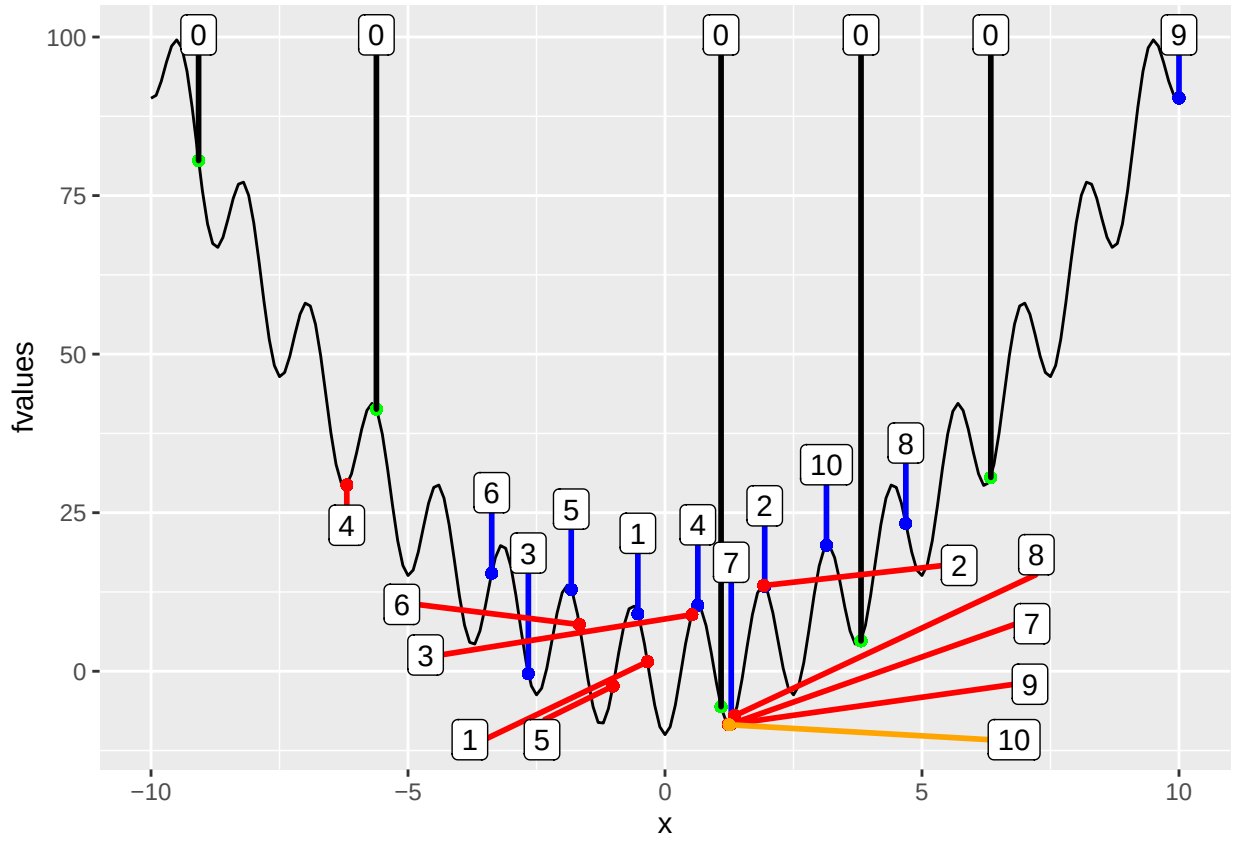
EI: Recommended parameters: x=-1.54,10 Objective: y = -99.537

Lower Cofnidence Bound

```
ctrl_CB = makeMBOControl()
ctrl_CB = setMBOControlTermination(ctrl_CB, iters=15)
ctrl_CB = setMBOControlInfill(ctrl_CB, crit=makeMBOInfillCritCB())
run_CB_b = mbo(obj.fxy, design=dfb, learner = surr.km, control=ctrl_CB, show.info = FALSE)
```

CB: Recommended parameters: x=-1.57,10 Objective: y = -99.586

Plotting



Interpretation

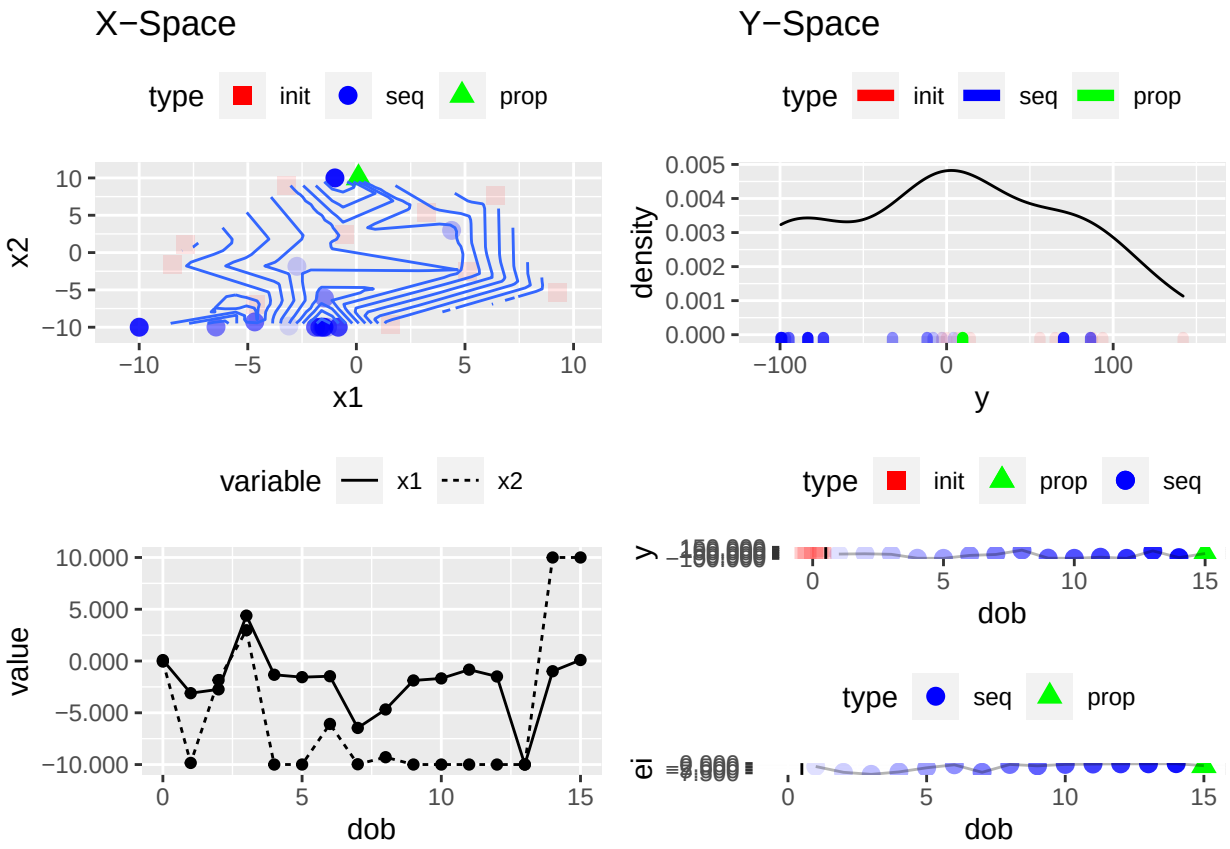
CB(rot) wertet vor allem mittig in der Naehe des Optimums aus, waehrend EI (blau) den Suchraum breiter abdeckt. Dabei faellt allerdings auf, dass haeufig an lokalen Maxima ausgewertet wurde. Woran das liegen koennte, oder ob das in diesem Fall reiner Zufall ist, ist unklar. Von diesem Einzelfall ausgehend, wuerde ich vermuten, dass CB "robuster", weil es den Suchraum systematischer untersucht. EI dagegen wirkt "dynamisch", was potentiell zu besseren Loesung fuehren koennte, aber das Risiko birgt, bessere Loesungen komplett zu uebersehen.

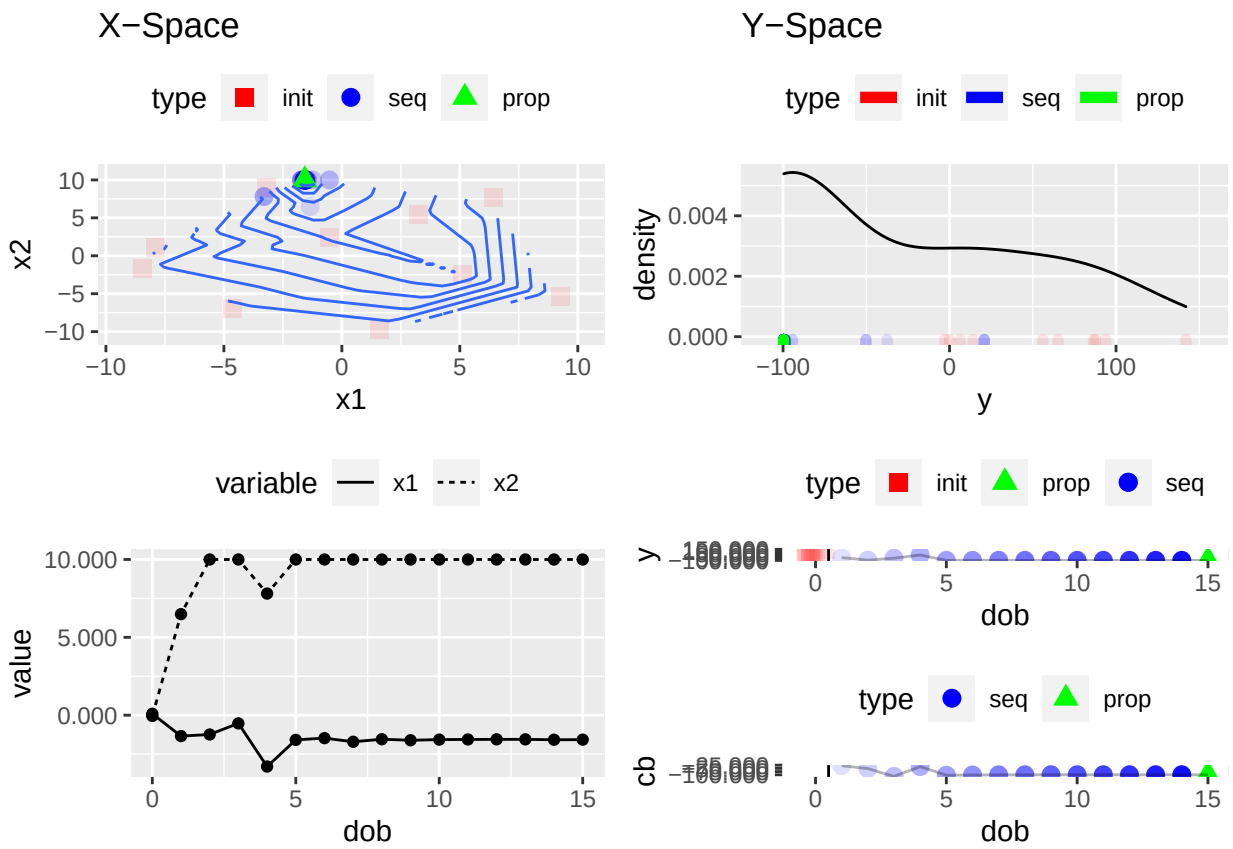
Dass beide Varianten genau dieses lokale Optimum finden, liegt hoechstwahrscheinlich aber auch daran, dass ein Punkt des initialen Designs knapp daneben lag.

Plot der b)

EI:

```
## Registered S3 method overwritten by 'GGally':
##   method from
##   +.gg   ggplot2
```


X−Space
type
init
seq
prop
x2
x1
Y−Space
type
init
seq
prop
density
y
variable
x1
x2
value
dob
type
init
prop
seq
y
dob
type
seq
prop
ei
dob



####CB:

Oder alternativ im 3D-Plot, der hier leider nicht angezeigt werden kann.

```
r  # library("rgl")  # x = rep(x = c(-40:40)/4, times = 84)  # y = rep(-40:40, each =
84)/4  # z = t(t(fxy(x, y)))  # EI_b_datapoints = print(run_EI_b)  # EI_b_x = t(t(EI_b_datapoints$x1))
# EI_b_y = t(t(EI_b_datapoints$x2))  # EI_b_z = t(t(EI_b_datapoints$y))  # EI_b = cbind(EI_b_x,
EI_b_y, EI_b_z)  #   # CB_b_datapoints = print(run_CB_b)  # CB_b_x = t(t(CB_b_datapoints$x1))
# CB_b_y = t(t(CB_b_datapoints$x2))  # CB_b_z = t(t(CB_b_datapoints$y))  # CB_b = cbind(CB_b_x,
CB_b_y, CB_b_z)  #   #   # dfb$z = fxy(dfb[,1], dfb[,2])  #   # dataframe = data.frame(
x, y, z )  #   # plot3d(dataframe, col = "grey")  # points3d(dfb, size = 6, col = "black")
# text3d(dfb, texts = 0)  # points3d(EI_b, size = 6, col="red")  # text3d(EI_b, texts
= 6:15)  # points3d(CB_b[1:9,], size = 6, col="blue")  # text3d(CB_b[1:9,], texts =
6:14)  # points3d(x=CB_b[10,1], y=CB_b[10,2], z=CB_b[10,3], size = 12, col = "green")
# text3d(CB_b[10,], texts = 15)
```

Erklaerung der Infill-Kriterien

Lower Confidence Bound

Bei LCB zieht man vom Punktschaetzer an der Stelle x die entsprechende Unsicherheit mal einen Hyperparameter κ ab.

```r
comp_LCB = function(f_dach,s_dach,kappa=1){
  LCB = f_dach - kappa * s_dach
  return(LCB)
}
```

Wir lesen nun aus dem Data Frame die Werte mean und se aus und vergleichen unser Ergebnis mit dem in der Spalte cb.

```
#print(run_CB_a)

comp_LCB(-1.7029,7.8270) == -9.5299
```

```
## [1] TRUE
```

```
#Tabelle: -9.5299

comp_LCB(-5.716831,2.35961483)
```

```
## [1] -8.076446
```

```
#Tabelle: -8.076446

comp_LCB(-7.615328,6.73106941)
```

```
## [1] -14.3464
```

```
#Tabelle: -14.346
```

Passt.

Expected Improvement

Hier kommt eine Normalverteilungsannahme hinzu. EI berechent sich wie folgt:

```
comp_EI = function(f_dach,f_min,s_dach){
  u = (f_min - f_dach) / s_dach
  eI = s_dach * (u * pnorm(u) + dnorm(u))
  return(eI)
}
```

Wir lesen wieder mean und se aus der Tablle aus. Der kleinste bisher gefunde Funktionswert stammt aus dem initialen Design.

```
#print(run_EI_a)

min= min(fx(lhsa_sc))

comp_EI(-0.84258,min,8.6198)
```

```
## [1] 1.576634
```

```
#Tabelle: 1.576645

comp_EI(-6.8222524,min,3.757011)
```

```
## [1] 2.200875
```

```
#Tabelle: 2.200875

comp_EI(24.8654400,min,27.026855)
```

```
## [1] 1.759435
```

```
#Tabelle: -1.7594348
```

Die Ergebnisse stimmen hier betragsmaessig ueberein, allerdings mit umgedrehtem Vorzeichen.

Das koennte daran liegen, dass wir standartmaessig minimieren. In diesem Kontext ist eine "erwartete Verbesserung" in negativer Richtung zu verstehen.