Arbeitsblatt 4

Aufgabe 1: Nächste-Nachbarn-Klassifizierer

Der Datensatz `Smarket` aus dem Paket `ISLR` enthält die Renditen (relative Änderung) des Aktienindex S&P 500 von Anfang 2001 bis Ende 2005: das sind 1250 Handelstage. An jedem Tag sind die Renditen der vorhergehenden 5 Handelstage angegeben (Variablen `Lag1`, ..., `Lag5`). Das Volumen des Handelstages ist in der Variable `Volume` enthalten (Anzahl Aktien die am Vortag gehandelt wurden in Milliarden). Die Variable `Today` enthält die Renditen des aktuellen Tages und die Variable `Direction` beschreibt, ob die aktuelle Rendite positiv oder negativ war. Hier soll versucht werden, die Richtung ("Direction") der Marktentwicklung vorherzusagen, wobei als erklärende Variablen die Renditen der 2 vorangegangenen Tage (`Lag1` und `Lag2`) benutzt werden sollen. Wir wollen die Jahre 2001-2004 zum Trainieren benutzten und dann den Klassifikator auf das Jahr 2005 anwenden. Die Daten findet man im Package `ISLR` und kann mit dem Befehl `Smarket` aufgerufen werden.

Ziel: KNN Klassifikation anhand der Aktienmarktbewegungen durchführen.

a) Verschaffen Sie sich einen Überblick über den Datensatz (R-Hinweis: `summary()`, `str()`, `dim()`, `pairs()`, `ts.plot()` ...). Datensatz auf Moodle oder direkt aus dem Paket `ISLR`.

```r
# Wir verschaffen uns einen Überblick Über die Daten.
library(ISLR)
data <- Smarket
str(data)   # Struktur
```

```
'data.frame':   1250 obs. of  9 variables:
 $ Year     : num  2001 2001 2001 2001 2001 ...
 $ Lag1     : num  0.381 0.959 1.032 -0.623 0.614 ...
 $ Lag2     : num  -0.192 0.381 0.959 1.032 -0.623 ...
 $ Lag3     : num  -2.624 -0.192 0.381 0.959 1.032 ...
 $ Lag4     : num  -1.055 -2.624 -0.192 0.381 0.959 ...
 $ Lag5     : num  5.01 -1.055 -2.624 -0.192 0.381 ...
 $ Volume   : num  1.19 1.3 1.41 1.28 1.21 ...
 $ Today    : num  0.959 1.032 -0.623 0.614 0.213 ...
 $ Direction: Factor w/ 2 levels "Down","Up": 2 2 1 2 2 2 1 2 2 2 ...
```

```r
summary(data)
```

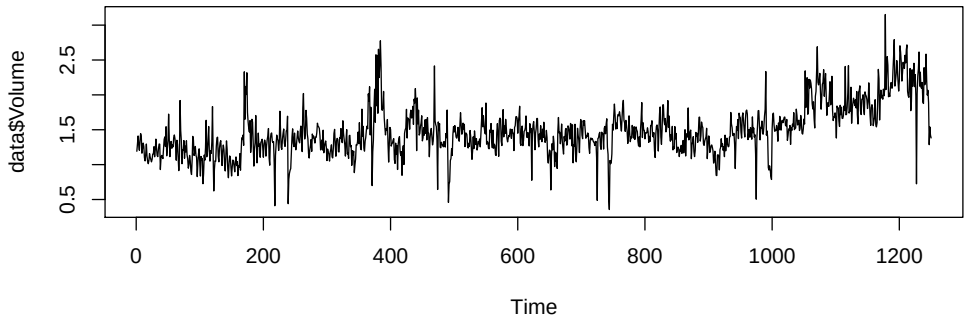
```
      Year           Lag1                Lag2                Lag3          
 Min.   :2001   Min.   :-4.922000   Min.   :-4.922000   Min.   :-4.922000  
 1st Qu.:2002   1st Qu.:-0.639500   1st Qu.:-0.639500   1st Qu.:-0.640000  
 Median :2003   Median : 0.039000   Median : 0.039000   Median : 0.038500  
 Mean   :2003   Mean   : 0.003834   Mean   : 0.003919   Mean   : 0.001716  
 3rd Qu.:2004   3rd Qu.: 0.596750   3rd Qu.: 0.596750   3rd Qu.: 0.596750  
 Max.   :2005   Max.   : 5.733000   Max.   : 5.733000   Max.   : 5.733000  
      Lag4                Lag5              Volume           Today          Direction
 Min.   :-4.922000   Min.   :-4.92200   Min.   :0.3561   Min.   :-4.922000   Down:602  
 1st Qu.:-0.640000   1st Qu.:-0.64000   1st Qu.:1.2574   1st Qu.:-0.639500   Up  :648  
 Median : 0.038500   Median : 0.03850   Median :1.4229   Median : 0.038500             
 Mean   : 0.001636   Mean   : 0.00561   Mean   :1.4783   Mean   : 0.003138             
 3rd Qu.: 0.596750   3rd Qu.: 0.59700   3rd Qu.:1.6417   3rd Qu.: 0.596750             
```

```
 Max.   : 5.733000   Max.   : 5.73300   Max.   :3.1525   Max.   : 5.733000
```

```
## Verteilung der Klassen
table(data$Direction)
```

```
Down   Up
 602  648
```

```
## Volume über die Zeit
ts.plot(data$Volume, type = "l")
```



Alles sind numerische Variablen bis auf die Direction, welche ein Faktor-Variable ist.
Das Handelsvolumen ist recht volatil. Wir sehen einen klaren Trend. Es liegt also keine Stationarität vor. Warum das Handelsvolumen am Ende der Zeitreihe so ansteigt, muss zusätzlich untersucht werden. Nach dem Platzen der Dot-Com Blase ist der S&P 500 Index im Oktober 2002 auf seinen Tiefststand gefallen. Danach begann der Index wieder zu steigen. Der Anstieg wird erst durch die globale Finanzkrise 2008 gebrochen.

b) Wählen Sie alle Beobachtungen in den Jahren vor 2005 mit den Variablen *Lag 1* und *Lag 2* als Trainings-Datensatz und die Beobachtungen in den restlichen Jahren als Test-Datensatz (auch mit den Variablen *Lag 1* und *Lag 2*).

```
train_index <- data$Year < 2005
table(train_index)
```

```
train_index
FALSE  TRUE
  252   998
```

```
train <- data[train_index, c("Lag1", "Lag2", "Direction")]
test <- data[!train_index, c("Lag1", "Lag2", "Direction")]
```

c) Trainieren Sie mit der Funktion `train` aus `caret` ein knn-Modell. Starten Sie mit einem nearest neighbor (`tuneGrid=data.frame(k=1)`). Verwenden Sie das Modell für eine Vorhersage auf dem **Trainings-Datensatz** (`predict(model, newdata=train)`. Quantifizieren Sie die Performance, indem Sie eine Kreuztabelle der wahren und vorhergesagten Klassen – also eine Konfusions-Matrix für den Trainings-Datensatz erstellen (Befehl `table` oder `confusionMatrix` aus 'caret). Was fällt Ihnen auf?

```
library(caret)
knn1 <- train(Direction ~., data = train, method="knn",
```

```
                    tuneGrid=data.frame(k=1))
knn.pred1_train <- predict(knn1, newdata=train)

confusionMatrix(data=knn.pred1_train, reference=train$Direction)
```

```
Confusion Matrix and Statistics

          Reference
Prediction Down  Up
      Down  491   0
      Up      0 507

               Accuracy : 1
                 95% CI : (0.9963, 1)
    No Information Rate : 0.508
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 1

 Mcnemar's Test P-Value : NA

            Sensitivity : 1.000
            Specificity : 1.000
         Pos Pred Value : 1.000
         Neg Pred Value : 1.000
             Prevalence : 0.492
         Detection Rate : 0.492
   Detection Prevalence : 0.492
      Balanced Accuracy : 1.000

       'Positive' Class : Down
```

```
# Alternative
table(knn.pred1_train, train$Direction)
```

```
knn.pred1_train Down  Up
           Down  491   0
           Up      0 507
```

Der Klassifikationsansatz hat alle Beobachtungen des Trainingsdatensets richtig klassifiziert. Dies ist nicht weiter überraschend, da er alle Beobachtungen der eigenen Beobachtung zugeordnet hat.

d) Nun wollen wir das knn-Modell mit k=1 richtig testen. Wenden Sie dazu die Prediction auf den Test-Datensatz an. Führen Sie `predict`-Funktion auch einmal mit dem Argument `type = prob"` durch. Was ist der Zweck des Types `prob`.

```
knn.pred1 <- predict(knn1, newdata=test)
knn.pred1_prob <- predict(knn1, newdata=test, type = "prob")
## Wahrscheinlichkeiten für Klassenzuweisung
```

Mit dem Argument type = "prob"kann man neben den Vorhersagen auch die Wahrscheinlichkeit für die zugewiesen Klasse erhalten. Das heisst, wie gross der Anteil der k-nächsten Nachbarn in

der vorhergesagten Klasse ist. Bei k = 1 muss die Wahrscheinlichkeit 1 sein, da nur 1 Nachbar die Klasse bestimmt und damit keine Uneinigkeit entstehen kann.

e) Erstellen Sie eine Konfusionsmatrix für den Test-Datensatz und beurteilen Sie damit die Performance Ihres Klassifikators anhand der Accuracy.

```
## Konfusionsmatrix
confusionMatrix(data=knn.pred1, reference=test$Direction)
```

```
Confusion Matrix and Statistics

          Reference
Prediction Down Up
      Down   43 58
      Up     68 83

               Accuracy : 0.5
                 95% CI : (0.4366, 0.5634)
    No Information Rate : 0.5595
    P-Value [Acc > NIR] : 0.9751

                  Kappa : -0.0242

 Mcnemar's Test P-Value : 0.4227

            Sensitivity : 0.3874
            Specificity : 0.5887
         Pos Pred Value : 0.4257
         Neg Pred Value : 0.5497
             Prevalence : 0.4405
         Detection Rate : 0.1706
   Detection Prevalence : 0.4008
      Balanced Accuracy : 0.4880

       'Positive' Class : Down
```

Die Performance ist wesentlich schlechter als das perfekte Ergebnis, wenn nur mit dem Trainingsset gearbeitet wird. Das war auch zu erwarten! Das heisst, obwohl im Trainingssample alles richtig klasssifiziert wurde, ist die Fehlerrate in einem neuen Testsample 50%, das heisst für uns, dass der Klassifikator absolut nutzlos ist. Da es nur zwei Möglichkeiten gibt (Up oder Down) ist das gleich gut wie raten.

f) Wiederholen Sie nun die Teilaufgaben d) - e) mit der Anzahl nearest neighbors von k=2-7 (`tuneGrid=data.frame(k=2:7)`). Welches k führt zum besten Resultat?

```
set.seed(1) # bei 2 Nachbarn wird gewürfelt, wenn Uneinigkeit besteht:
             # durch seed geht würfeln immer gleich aus
knn <- train(Direction ~., data = train, method="knn",
                   tuneGrid=data.frame(k=2:7))
knn
```

```
k-Nearest Neighbors
```

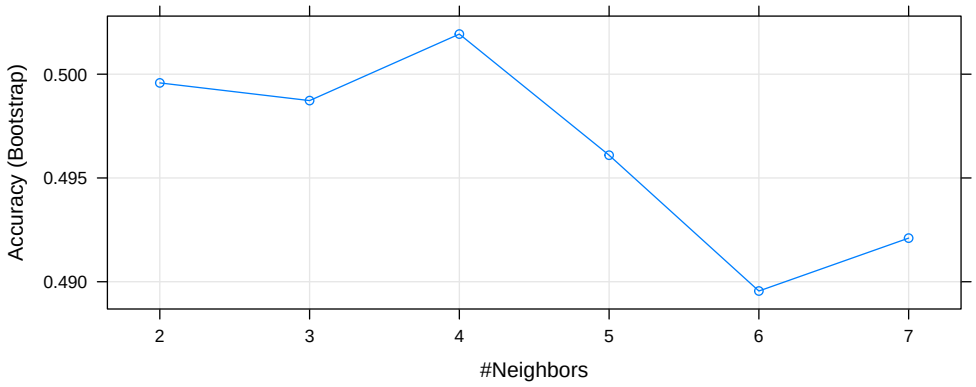
```
998 samples
  2 predictor
  2 classes: 'Down', 'Up'

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 998, 998, 998, 998, 998, 998, ...
Resampling results across tuning parameters:

  k  Accuracy   Kappa
  2  0.4995801  -0.0005764467
  3  0.4987284  -0.0027363959
  4  0.5019327   0.0041440873
  5  0.4960967  -0.0071721632
  6  0.4895533  -0.0204319286
  7  0.4921021  -0.0152456160

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was k = 4.
```

```
plot(knn)
```




```
knn.pred <- predict(knn, newdata=test)
knn.pred_prob <- predict(knn, newdata=test, type = "prob")

## Konfusionsmatrix
confusionMatrix(data=knn.pred, reference=test$Direction)
```

```
Confusion Matrix and Statistics

          Reference
Prediction Down Up
      Down   48 63
      Up     63 78

               Accuracy : 0.5
                 95% CI : (0.4366, 0.5634)
    No Information Rate : 0.5595
```

```
    P-Value [Acc > NIR] : 0.9751

                  Kappa : -0.0144

 Mcnemar's Test P-Value : 1.0000

            Sensitivity : 0.4324
            Specificity : 0.5532
         Pos Pred Value : 0.4324
         Neg Pred Value : 0.5532
             Prevalence : 0.4405
         Detection Rate : 0.1905
   Detection Prevalence : 0.4405
      Balanced Accuracy : 0.4928

       'Positive' Class : Down
```

Die Analyse zeigt, dass der optimale Wert für k im bereich von 4 liegt. Der Wert ist jedoch nicht sehr sensitiv und kann je nach Durchführung bei einem leicht anderen Wert liegen. Für die Wahl des k verwenden wir hier nur die Trainingsdaten. Das ist auch das korrekte vorgehen, da wir sonst auf den Testdaten optimieren. Auf den Testdaten erhalten wir eine Accuracy von ca. 54%. Bei zwei Klassen sollte man eigentlich immer eine ungerade Anzahl nächste Nachbarn wählen, damit immer eine eindeutige Wahl gefällt wird. Bei einem Patt wird die Klassenzugehörigkeit ausgelost. Mit dem Argument type=prob"kann man anhand der Wahrscheinlichkeiten sehen, in welchen Fällen eine zufällige Zuordnung erfolgt. Bei mehr als 2 Klassen kann man ein Patt bei $k > 1$ nicht immer verhindern. Allgemein funktioniert die Klassifikation hier nicht besonderes gut.

Aufgabe 2: Kreuzvalidierung

In dieser Aufgabe wird das Churn-Verhalten (Wechseln zu einem anderen Anbieter) von Telekommunikationskunden untersucht. Der Wettbewerb unter den Telekommunikationsanbietern ist sehr gross. Viele Kunden wechseln jedes Jahr den Anbieter. Telekommunikationsunternehmen sind deshalb bestrebt, Kunden durch geeignete Marketing-Programme vom Wechseln abzuhalten. Dazu müssen jedoch die zu einem Wechsel neigenden Kunden identifiziert werden, was z.B. mit einem Klassifikationsverfahren aufgrund der Kundenmerkmale gemacht werden kann.

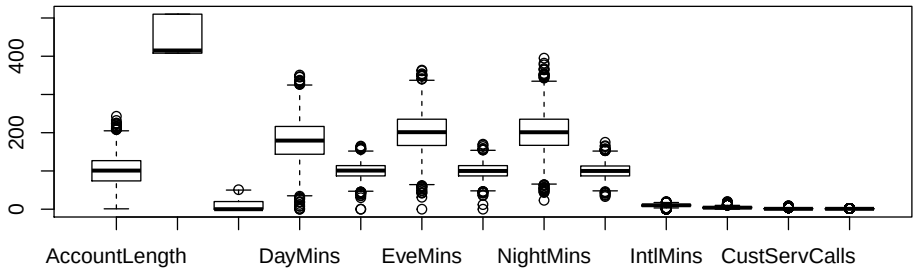
Ihnen steht nun ein Datensatz zur Verfügung, in dem von 3'333 Kunden eines US-Telekommunikationsunternehmens bekannt ist, ob sie den Anbieter gewechselt haben oder nicht. Zugleich kennen wir noch folgende Kundenmerkmale:

Variablenname	Kurzbeschreibung	Typ
AccountLength	Vertragsdauer in Monaten	integer
AreaCode	Postleitzahl	*drei* Integerwerte
IntlPlan	Spezialvertrag für internationale Anrufe	binär
VMailPlan	Voice-Mail-Plan	binär
VMailMsg	Anzahl Voice-Mail-Nachrichten	integer
DayMins	Zeitdauer mit Gesprächen am Tag (in Minuten)	stetig
DayCalls	Anzahl Anrufe am Tag	integer
EveMins	Zeitdauer mit Gesprächen am Abend (in Minuten)	stetig
EveCalls	Anzahl Anrufe am Abend	integer
NightMins	Zeitdauer mit Gesprächen in der Nacht (in Minuten)	stetig

Variablenname	Kurzbeschreibung	Typ
NightCalls	Anzahl Anrufe in der Nacht	integer
IntlMins	Zeitdauer mit internationalen Gesprächen (in Minuten)	stetig
IntlCalls	Anzahl internationale Anrufe	integer
CustServCalls	Anzahl Anrufe an den Kundenservice	integer
Churn	Kunde hat den Anbieter gewechselt oder nicht	binär

a) Laden Sie die Daten `churn` (`Churn1.rdata`). Die Faktorvariablen `IntlPlan` und `VMailPlan` wollen wir hier nicht berücksichtigen, obwohl das mit der caret-Funktion auch funktionieren würde. Löschen Sie diese Features (z.B. `churn$IntlPlan <- NULL`). Skalieren Sie die restlichen Daten, falls nötig (Funktion `scale()` oder `preProcess = c(center", ßcale")` in der Funktion `train`. Verwenden Sie 80 Prozent der Daten als Trainingset und 20 Prozent der Daten als Testset (z.B. mit der Funktion `createDataPartition(y=churn_sc$Churn, p=0.8, list=F)`). Wie gross ist die Performance eines k=1 Klassifikators auf dem Trainings- und Testset. Begründen Sie, wieso die Fehlerrate (`mean(res.knn != y_test)`) auf dem Trainingset kleiner ist.

```
load("Daten/Churn1.rdata")
churn$IntlPlan <-  NULL
churn$VMailPlan <- NULL
boxplot(churn)
```




```
intrain <- createDataPartition(y=churn$Churn, p=0.8, list=F)
churn_train <- churn[intrain,]
churn_test <- churn[-intrain,]

knn <- train(Churn ~., data = churn_train, method="knn",
                    tuneGrid=data.frame(k=1),
             preProcess = c("center", "scale"))


pred_train <- predict(knn, newdata=churn_train)
mean(pred_train != churn_train$Churn)
```

```
[1] 0
```

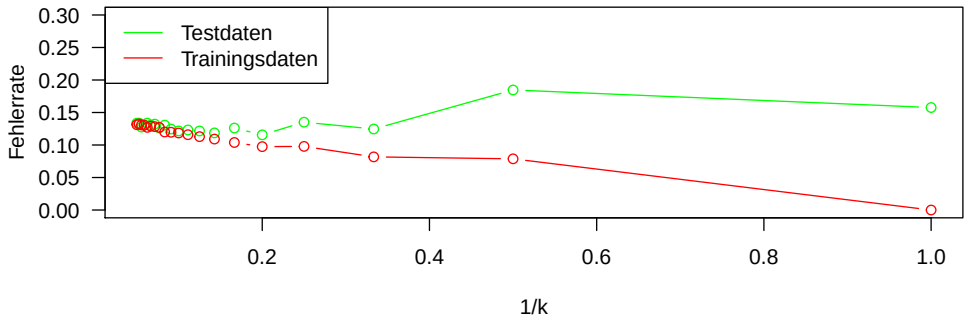
```
pred_test <- predict(knn, newdata=churn_test)
mean(pred_test != churn_test$Churn)
```

```
[1] 0.1576577
```

Bei $k = 1$ muss die Fehlerrate bei den Trainingsdaten immer 0 sein, da nur das eigene Label die Klasse definiert.

b) Wiederholen Sie a) für alle Werte von $k = 1, \ldots, 20$ und plotten Sie die Fehlerrate von Trainings- und Testdaten gegenüber 1/k. Für welches k performt der knn-Klassifizierer am besten.

```
error_test <- error_training <- rep(NA,20)
set.seed(8)
for (k in 1:20){
  knn <- train(Churn ~., data = churn_train, method="knn",
                   tuneGrid=data.frame(k=k),
               preProcess = c("center", "scale"))
  pred_train <- predict(knn, newdata=churn_train)
  error_training[k] <- mean(pred_train != churn_train$Churn)
  pred_test <- predict(knn, newdata=churn_test)
  error_test[k] <- mean(pred_test != churn_test$Churn)
}
plot(x=1/(1:20),y=error_test, ylim = c(0,0.3),type='b', col='green',
     xlab="1/k",
     ylab="Fehlerrate", las=1)
lines(x=1/(1:20),y=error_training,col='red', type="b")
legend("topleft", legend=c("Testdaten", "Trainingsdaten"),
       col=c("green", "red"), lty=1)
```




```
which.min(error_test)
```

```
[1] 5
```

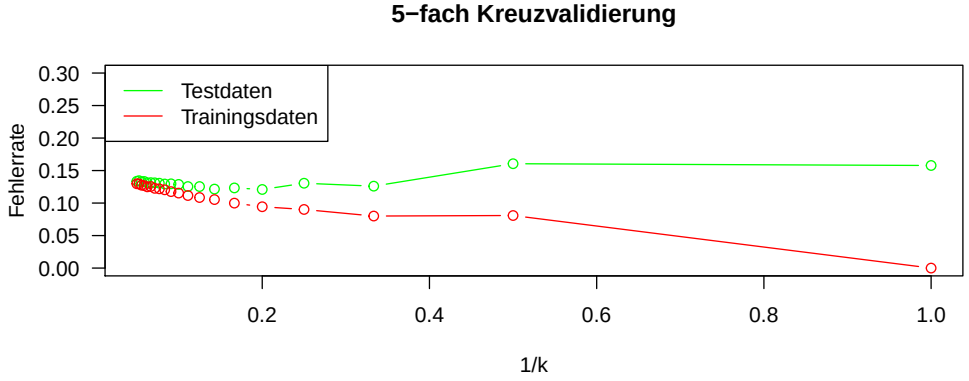
Die Abbildung zeigt die Fehlerrat in Abhängigkeit des reziproken k-Wertes für die Anzahl berücksichtiger Nachbarn für Test- und Trainingsdaten. Bis zu einen Wert von 0.3 ist die Fehlerrate in etwa konstant. Anschliessend fällt Sie bei den Trainingsdaten ab und steigt bei den Trainingsdaten leicht an. Bei dieser Realisierung wird der minimale Fehler bei den Testdaten bei $k = 9$ erreicht. Dieser Wert ist aber sehr variable. Je nach Dateneinteilung in a) und dem set.seed in b) ist für k

irgendein Wert im Bereich von 5 bis 20 möglich.

c) Wiederholen Sie die Analyse aus b). Führen Sie diesmal aber eine 5-fache Kreuzvalidierung auf dem gesamten Datensatz durch. Verwenden Sie dazu folgenden Code und versuchen Sie ihn nachzuvollziehen.

```
x_fold <- 5
groups <- rep(1:x_fold, length.out=dim(churn)[1])
samp <- sample(x=groups, size=dim(churn)[1], replace=FALSE)
error_test <- error_training <- rep(NA,20)
for (k in 1:20){
  error_test_fold <- rep(NA, x_fold)
  error_train_fold <- rep(NA, x_fold)
  for (i in 1:x_fold){
    churn_train <- churn[samp!=i,]
    churn_test <- churn[samp==i,]
    knn <- train(Churn ~., data = churn_train, method="knn",
                 tuneGrid=data.frame(k=k),
            preProcess = c("center", "scale"))
    pred_train <- predict(knn, newdata=churn_train)
    error_train_fold[i] <- mean(pred_train != churn_train$Churn)
    pred_test <- predict(knn, newdata=churn_test)
    error_test_fold[i] <- mean(pred_test != churn_test$Churn)
  }
  error_training[k] <- mean(error_train_fold)
  error_test[k] <- mean(error_test_fold)
}
```

```
plot(x=1/(1:20),y=error_test, ylim = c(0,0.3),type='b', col='green', xlab="1/k",
     ylab="Fehlerrate", las=1,
     main=paste(x_fold, "-fach Kreuzvalidierung", sep=""))
lines(x=1/(1:20),y=error_training,col='red', type="b")
legend("topleft", legend=c("Testdaten", "Trainingsdaten"),
       col=c("green", "red"),
       lty=1)
```



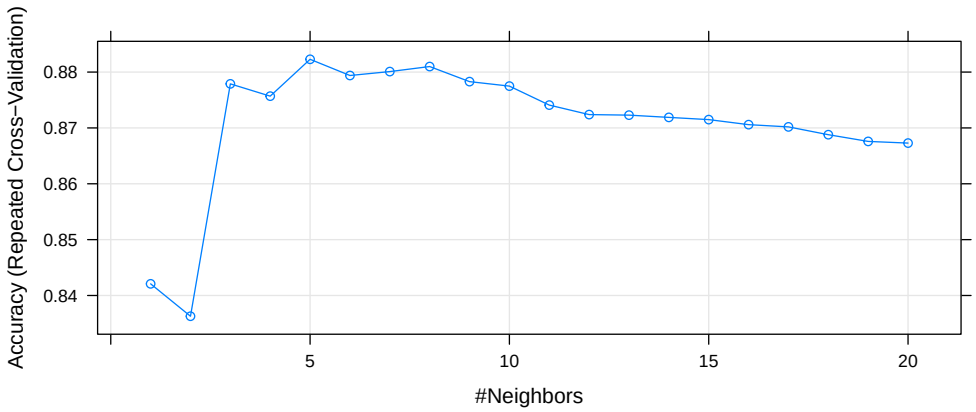

```
which.min(error_test)
```

```
[1] 5
```

Die Evaluierung der optimalen Anzahl Nachbaren mit der 5-fach Kreuzvalidierung ist stabiler als diejenige mit dem Standard-Validierungsansatz.

d) Viel einfacher kann man die Kreuzvalidierung direkt mit dem **Paket `caret`** durchführen (Code unten). Dort kann man mit der **Funktion `trainControl()`** die Parameter für die Validierung festlegen. Wir verwenden `method = 'repeatedcv'`, bei dieser Methode wird die Kreuzvalidierung mehrmals wiederholt und der Mittelwert verwendet. Das reduziert die Schwankungen, dauert aber auch länger. Die Anzahl an Wiederholung wird mit dem Argument `repeats` festgelegt. Bei `method = cv` wird die Kreuzvalidierung nur einmal durchgeführt. Das k für die k-fache Kreuzvalidierung wird über den Parameter `number` gesteuert. `number = 10` bedeutet zum Beispiel 10-fache Kreuzvalidierung. Das Trainieren des Klassifizierer geschieht mit der **Funktion `train()`**. Der Funktion müssen die Parameter aus dem `trainControl` übergeben werden (Argument `trControl`). Die zu evaulierenden Parameter können in einem Dataframe an das Argument `tuneGrid` übergeben werden. Auch das Preprocessing der Daten ist direkt möglich. Mit `preProcess = c('center', 'scale')` werden die Daten zum Beispiel zentriert und skaliert. Mit einen Plot auf den Output, erhält man direkt eine Abbildung der Accuracy in Abhängigkeit der Anzahl Nachbarn. (Das Vorgehen kann einfach auf einen anderen Klassifizier übertragen werden, es muss nur eine andere Methode gewählt werden und allenfalls die Parameter angepasst werden).

```
ctrl <- trainControl(method = "repeatedcv",
                     number =10,
                     repeats = 3)
train_knn <- train(Churn ~ ., data=churn, method = "knn",
                   trControl=ctrl,
                   tuneGrid = data.frame(k=1:20),
                   preProcess = c("center", "scale"))
plot(train_knn)
```



e) Wieso empfiehlt es sich in diesem Fall, eine 10-fache Kreuzvalidation und keine Leave-one-out-Kreuzvalidierung zu verwenden?

Der Rechenaufwand bei der Leave-one-out-Crossvalidation wird zu gross. Die Leave-one-out-Kreuzvalidierung könnten Sie durchführen, wenn Sie im R-Code von c) x_fold auf 3333 setzen.

Aufgabe 3: Diskriminanzanalyse

Verwenden Sie den Boston Datensatz `Boston.RData` um vorherzusagen, ob ein Stadtteil (Zeile im Datensatz) eine grosse Kriminalitätsrate hat. Die Zielvariable ist dabei `Boston$crime`

a) Laden Sie die Daten und unterteilen Sie den Datensatz in einen Trainings und einen Testdatensatz, wobei die Daten gleich verteilt werden sollen. Verwenden Sie dazu den Befehl `createDataPartition` aus dem Packet `caret`.

```
load("Daten/Boston.RData")
index = createDataPartition(y=Boston$crime, p=0.5, list=FALSE)

train = Boston[index,]
test = Boston[-index,]
```

b) Verwenden Sie mit dem Befehl `train` aus dem Packet `caret` die LDA Methode und sagen Sie anhand der Trainingsdaten die Testdaten voraus. Erstellen Sie anschliessend eine Konfusionsmatrix daraus.

```
lda.fit = train(crime ~ ., data=train, method="lda",
                trControl = trainControl(method = "cv"),
                   preProcess = c("center", "scale"))
pred.crime = predict(lda.fit, test)
confusionMatrix(data= pred.crime, reference = test$crime)
```

```
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 118  33
         1   8  93

               Accuracy : 0.8373
                 95% CI : (0.7858, 0.8806)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.6746

 Mcnemar's Test P-Value : 0.0001781

            Sensitivity : 0.9365
            Specificity : 0.7381
         Pos Pred Value : 0.7815
         Neg Pred Value : 0.9208
             Prevalence : 0.5000
         Detection Rate : 0.4683
   Detection Prevalence : 0.5992
      Balanced Accuracy : 0.8373

       'Positive' Class : 0
```

c) Gehen Sie gleich wie bei Teilaufgabe b) vor, dieses mal einfach mit der QDA Methode. Erstellen Sie wiederum daraus die Konfusionsmatrix.

```
qda.fit = train(crime ~ ., data=train, method="qda",
                trControl = trainControl(method = "cv"),
                   preProcess = c("center", "scale"))
```

```
pred.crime2 = predict(qda.fit, test)
confusionMatrix(data= pred.crime2, reference = test$crime)
```

```
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 119  27
         1   7  99

               Accuracy : 0.8651
                 95% CI : (0.8166, 0.9047)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : < 2e-16

                  Kappa : 0.7302

 Mcnemar's Test P-Value : 0.00112

            Sensitivity : 0.9444
            Specificity : 0.7857
         Pos Pred Value : 0.8151
         Neg Pred Value : 0.9340
             Prevalence : 0.5000
         Detection Rate : 0.4722
   Detection Prevalence : 0.5794
      Balanced Accuracy : 0.8651

       'Positive' Class : 0
```

d) Welcher Ansatz funktioniert besser?

Die Evaluationsmetriken für QDA sind allgemein etwas besser.

Aufgabe 4: Leistungsnachweise

- Versuchen Sie einen unser Klassifizierer auf Ihren Datensatz anzuwenden. Beachten Sie dabei die Aspekte, welche wir heute besprochen haben.