Vergleich von verschiedenen KI-Klassifizierungsmethoden am Beispiel des Fashion MNIST-Datensatzes in RStudio und Python (Teil 4)

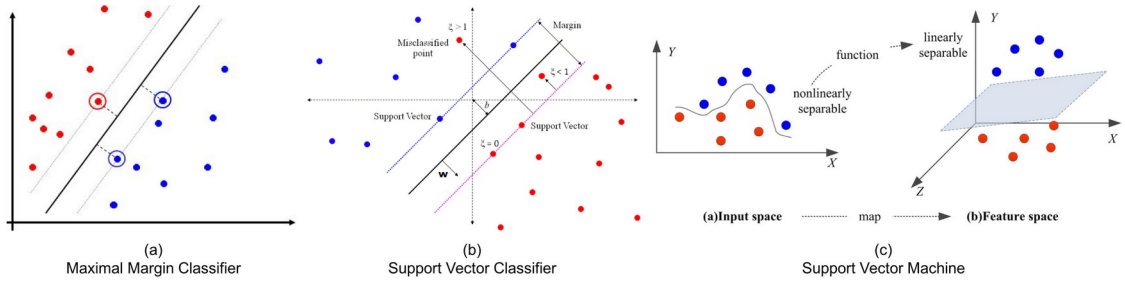
Lars Mehnen

2022-05-15

In dieser Reihe werde wir verschiedene maschinelle und Deep-Learning-Methoden vergleichen, um Kleidungsklassen anhand von Bildern des Fashion MNIST-Datensatzes zu erstellen bzw. mit dem erstellten Modellen zu klassifizieren. In diesem Beitrag werden wir sehen, wie Sie Support Vector Machines schätzen, die Ergebnisse aller Modelle vergleichen, die wir in dieser Beitragsserie gesehen haben, und alles zusammenfassen.

Support Vector Machine

Support Vector Machines (SVMs) bieten eine weitere Methode zur Klassifizierung der Kleidungskategorien in den Fashion MNIST-Daten. Um besser zu verstehen, was SVMs beinhalten, müssen wir einige komplexere Erklärungen durchgehen. Die folgende Abbildung kann Ihnen helfen, die verschiedenen Klassifikatoren zu verstehen, die ich in den nächsten Abschnitten besprechen werde (Abbildungen stammen von hier, hier und hier).



(a) Maximal Margin Classifier (b) Support Vector Classifier (c) Support Vector Machine

Für eine $n \times p$-Datenmatrix und eine binäre Ergebnisvariable $y_i \in \{-1, 1\}$ ist eine Hyperebene ein flacher affiner Unterraum der Dimension $p-1$, der den p-dimensionalen Raum in zwei Hälften teilt, definiert durch $\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p$. Einer Beobachtung in den Testdaten wird eine Ergebnisklasse zugeordnet, je nachdem, auf welcher Seite der perfekt trennenden Hyperebene sie liegt, vorausgesetzt, dass eine solche Hyperebene existiert. Cutoff t für den Wert einer Beobachtung $\hat{f}(X) = \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2 + \ldots + \hat{\beta}_p X_p$ bestimmt, welcher Klasse der Wert zugeordnet wird. Je weiter eine Beobachtung von der Hyperebene (bei Null) entfernt ist, desto sicherer ist sich der Klassifikator bezüglich der Klassenzuordnung. Falls vorhanden, können unendlich viele trennende Hyperebenen konstruiert werden. Eine gute Option in diesem Fall wäre die Verwendung des Maximal Margin Classifier (MMC), der

den Rand um die Mittellinie des breitesten Streifens maximiert, der zwischen den beiden Ergebnisklassen eingefügt werden kann.

Wenn keine perfekt trennende Hyperebene existiert, können „fast trennende“ Hyperebenen mit Hilfe des Support Vector Classifier (SVC) verwendet werden. Der SVC erweitert die MMC, da Klassen nicht durch eine lineare Grenze trennbar sein müssen, indem Schlupfvariablen ϵ_i eingeschlossen werden, die es zulassen, dass einige Beobachtungen auf der falschen Seite des Randes oder der Hyperebene liegen. Das Ausmaß, in dem fehlerhafte Platzierungen vorgenommen werden, wird durch die Einstellung der Parameterkosten $C \geq \sum_{i=1}^{n} \epsilon_i$ bestimmt, wodurch der Bias-Varianz-Kompromiss gesteuert wird. Der SVC ist dem MMC vorzuziehen, da er aufgrund der größeren Margen sicherer in Klassenzuordnungen ist und eine größere Robustheit gewährleistet, da lediglich Beobachtungen am Rand oder eine Verletzung des Rands die Hyperebene beeinflussen.

Sowohl MMCs als auch SVCs gehen von einer linearen Grenze zwischen den beiden Klassen der Ergebnisvariablen aus. Nichtlinearität kann behandelt werden, indem der Merkmalsraum unter Verwendung von Funktionen von Prädiktoren vergrößert wird. Support-Vektor-Maschinen kombinieren SVCs mit nichtlinearen (z. B. radialen, polynomialen oder sigmoiden) Kernel $K\left(x_i, x_{i'}\right)$, um effiziente Berechnungen zu ermöglichen. Kernel sind Verallgemeinerungen der inneren Produkte, die die Ähnlichkeit zweier Beobachtungen quantifizieren. Normalerweise wird der radiale Kernel für nichtlineare Modelle ausgewählt, da dieser einen guten Standardkernel in Ermangelung vorheriger Kenntnisse von Invarianzen bezüglich Translationen liefert. Der radiale Kernel ist definiert als $K\left(x_i, x_{i'}\right) = \exp\left(-\sigma \sum_{j=1}^{p} \left(x_{ij} - x_{i'j}\right)^2\right)$, wobei σ eine positive Konstante ist, die die Anpassung mit zunehmender Größe nichtlinearer macht. Das Abstimmen von C und σ ist notwendig, um den optimalen Kompromiss zwischen der Verringerung der Anzahl von Trainingsfehlern und der unregelmäßigeren Gestaltung der Entscheidungsgrenze (durch Erhöhen von `C`) zu finden. Da SVMs nur die Berechnung von $\binom{n}{2}$ Kernels für alle unterschiedlichen Beobachtungspaare erfordern, verbessern sie die Effizienz erheblich.

SVMs können mit dem One-versus-one-Ansatz oder dem One-versus-all-Ansatz auf mehr als zwei Klassen erweitert werden. Im Fall von K-Klassen konstruiert die erstere $\binom{K}{2}$ SVMs und ordnet Testbeobachtungen der Klasse zu, der sie in diesen $\binom{K}{2}$-Klassifikatoren am häufigsten zugeordnet wurden. Letztere passt zu K SVMs und vergleicht jedes Mal eine der K Klassen mit den verbleibenden $K-1$ Klassen, wobei jeder Beobachtung die Klasse zugeordnet wird, für die die Konfidenz, richtig zu sein, am höchsten ist.

Wie oben erwähnt, sind die Parameter, die abgestimmt werden müssen, die Kosten `C` und im Fall eines radialen Kernels die Nichtlinearitätskonstante `sigma`. Beginnen wir damit, diese Parameter mit einem zufälligen Suchalgorithmus abzustimmen, wobei wiederum das

caret-Framework verwendet wird. Wir stellen die Steuerelemente so ein, dass sie eine 5-fache Kreuzvalidierung durchführen, und wir verwenden die Funktion multiClassSummary() aus der MLmetrics-Bibliothek, um eine Mehrklassenklassifizierung durchzuführen. Wir spezifizieren einen radialen Kernel, verwenden Genauigkeit als Leistungsmetrik1 und lassen den Algorithmus eine zufällige Suche nach dem Kostenparameter C über pca.dims (=17) zufälligen Werten durchführen. Beachten Sie, dass der Zufallssuchalgorithmus nur nach Werten von C sucht, während er einen konstanten Wert für sigma beibehält. Außerdem setzen wir im Gegensatz zu früheren Aufrufen von trainControl() jetzt classProbs = FALSE, da das Basispaket, das zum Schätzen von SVMs in caret und kernlab verwendet wird, aufgrund der Verwendung eines sekundären Regressionsmodells zu geringeren Genauigkeiten führt, wenn classProbs = TRUE angegeben wird.

```
library(MLmetrics)

##
## Attache Paket: 'MLmetrics'

## Das folgende Objekt ist maskiert 'package:base':
##
##     Recall

library(caret)

## Lade nötiges Paket: ggplot2

## Lade nötiges Paket: lattice

##
## Attache Paket: 'caret'

## Die folgenden Objekte sind maskiert von 'package:MLmetrics':
##
##     MAE, RMSE

svm_control = trainControl(method = "repeatedcv",
                           number = 5,
                           repeats = 5,
                           classProbs = FALSE,
                           allowParallel = TRUE,
                           summaryFunction = multiClassSummary,
                          savePredictions = TRUE)

set.seed(815)
svm_rand_radial = train(label ~ .,
                data = cbind(train.images.pca, label = train.classes),
                method = "svmRadial",
                trControl = svm_control,
                tuneLength = pca.dims,
                metric = "Accuracy")
svm_rand_radial
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 60000 samples
##    17 predictor
##    10 classes: 'Bag', 'Boot', 'Coat', 'Dress', 'Pullover',
'Sandal', 'Shirt', 'Sneaker', 'Top', 'Trouser'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 48000, 48000, 48000, 48000, 48000,
48000, ...
## Resampling results across tuning parameters:
##
##   C         Accuracy   Kappa      Mean_F1    Mean_Sensitivity
Mean_Specificity
##       0.25  0.8481267  0.8312519  0.8467962  0.8481267
0.9831252
##       0.50  0.8558967  0.8398852  0.8548124  0.8558967
0.9839885
##       1.00  0.8630867  0.8478741  0.8621885  0.8630867
0.9847874
##       2.00  0.8679867  0.8533185  0.8671696  0.8679867
0.9853319
##       4.00  0.8715300  0.8572556  0.8707298  0.8715300
0.9857256
##       8.00  0.8740967  0.8601074  0.8733118  0.8740967
0.9860107
##      16.00  0.8759467  0.8621630  0.8752003  0.8759467
0.9862163
##      32.00  0.8758333  0.8620370  0.8751317  0.8758333
0.9862037
##      64.00  0.8745667  0.8606296  0.8739354  0.8745667
0.9860630
##     128.00  0.8724833  0.8583148  0.8718764  0.8724833
0.9858315
##     256.00  0.8695333  0.8550370  0.8689398  0.8695333
0.9855037
##     512.00  0.8653567  0.8503963  0.8648410  0.8653567
0.9850396
##    1024.00  0.8606433  0.8451593  0.8601643  0.8606433
0.9845159
##    2048.00  0.8564433  0.8404926  0.8559755  0.8564433
0.9840493
##    4096.00  0.8520000  0.8355556  0.8515218  0.8520000
0.9835556
##    8192.00  0.8478500  0.8309444  0.8473895  0.8478500
0.9830944
##   16384.00  0.8432300  0.8258111  0.8427903  0.8432300
0.9825811
##   Mean_Pos_Pred_Value  Mean_Neg_Pred_Value  Mean_Precision
```

```
Mean_Recall
##   0.8469423           0.9831737           0.8469423
0.8481267
##   0.8548617           0.9840274           0.8548617
0.8558967
##   0.8621761           0.9848191           0.8621761
0.8630867
##   0.8671578           0.9853608           0.8671578
0.8679867
##   0.8707271           0.9857539           0.8707271
0.8715300
##   0.8732913           0.9860383           0.8732913
0.8740967
##   0.8751779           0.9862425           0.8751779
0.8759467
##   0.8751326           0.9862286           0.8751326
0.8758333
##   0.8738879           0.9860849           0.8738879
0.8745667
##   0.8717839           0.9858521           0.8717839
0.8724833
##   0.8688357           0.9855238           0.8688357
0.8695333
##   0.8647479           0.9850571           0.8647479
0.8653567
##   0.8601044           0.9845325           0.8601044
0.8606433
##   0.8559269           0.9840656           0.8559269
0.8564433
##   0.8514912           0.9835725           0.8514912
0.8520000
##   0.8473994           0.9831112           0.8473994
0.8478500
##   0.8428689           0.9825980           0.8428689
0.8432300
##   Mean_Detection_Rate  Mean_Balanced_Accuracy
##   0.08481267          0.9156259
##   0.08558967          0.9199426
##   0.08630867          0.9239370
##   0.08679867          0.9266593
##   0.08715300          0.9286278
##   0.08740967          0.9300537
##   0.08759467          0.9310815
##   0.08758333          0.9310185
##   0.08745667          0.9303148
##   0.08724833          0.9291574
##   0.08695333          0.9275185
##   0.08653567          0.9251981
##   0.08606433          0.9225796
##   0.08564433          0.9202463
```

```
##   0.08520000           0.9177778
##   0.08478500           0.9154722
##   0.08432300           0.9129056
##
## Tuning parameter 'sigma' was held constant at a value of 0.04052872
## Accuracy was used to select the optimal model using the largest
value.
## The final values used for the model were sigma = 0.04052872 and C =
16.

mp.svm.rand.radial = model_performance(svm_rand_radial,
train.images.pca, test.images.pca,
                                       train.classes, test.classes,
"svm_random_radial")

##   accuracy_train precision_train recall_train  F1_train
accuracy_test
## 1      0.8759467       0.9569655       0.9688 0.9628464
0.8668
##   precision_test recall_test   F1_test             model
## 1       0.954902       0.974 0.9643564 svm_random_radial
```

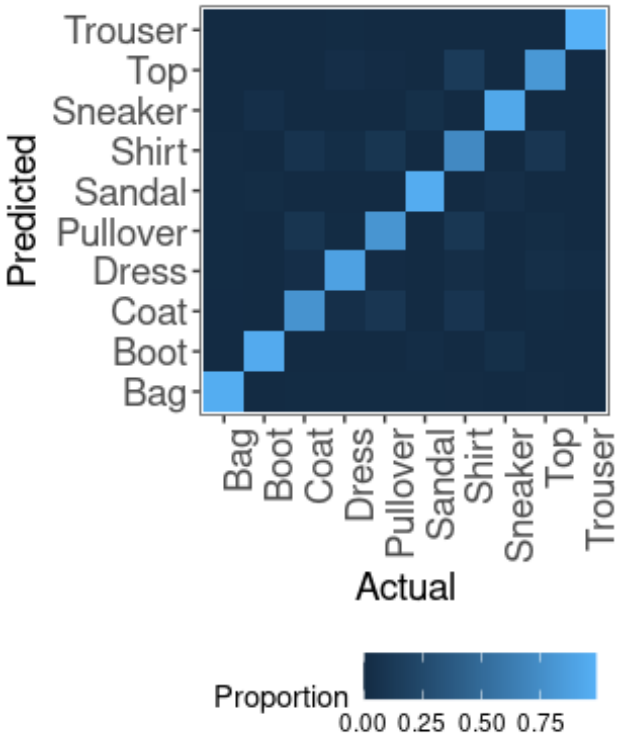
Die Ergebnisse zeigen, dass das Modell relativ hohe Genauigkeiten von 88 % bzw. 87 % bei den Trainings- und Testsätzen erreicht, wobei `sigma = 0.040` und `C = 32` als optimale Parameter ausgewählt wurden. Werfen wir einen Blick auf die Confusion-Matrix, um zu sehen, welche Kleidungskategorien am besten und welche am schlechtesten vorhergesagt werden, indem die Confusion-Matrix visualisiert wird. Zuerst berechnen wir die Vorhersagen für die Trainingsdaten, wie beim Definieren der bereits bekannten `model_performance`-Funktion.

```
library(data.table)
pred_dt = as.data.table(svm_rand_radial$pred[,
names(svm_rand_radial$bestTune)])
names(pred_dt) = names(svm_rand_radial$bestTune)
index_list = lapply(1:ncol(svm_rand_radial$bestTune), function(x, DT,
tune_opt){
  return(which(DT[, Reduce(`&`, lapply(.SD, `==`, tune_opt[, x])),
.SDcols = names(tune_opt)[x]]))
}, pred_dt, svm_rand_radial$bestTune)
rows = Reduce(intersect, index_list)
pred_train = svm_rand_radial$pred$pred[rows]
trainY = svm_rand_radial$pred$obs[rows]
conf = table(pred_train, trainY)
```

Als Nächstes formen wir die Konfusionsmatrix in einen Datenmatrix mit drei Spalten um: eine für die wahren Kategorien (`trainY`), eine für die vorhergesagten Kategorien (`pred_train`) und eine für den Anteil korrekter Vorhersagen für die wahre Kategorie (`Freq`). Wir zeichnen dies als Kacheldiagramm mit einer blauen Farbskala, wobei hellere Werte einen größeren Anteil an Übereinstimmungen zwischen einer bestimmten Kombination aus wahren und vorhergesagten Kategorien anzeigen und dunklere Werte

einen kleinen Anteil an Übereinstimmungen zwischen ihnen anzeigen. Beachten Sie, dass wir wieder das benutzerdefinierte Plotdesign `my_theme()` verwenden.

```
library(ggplot2)
my_theme = function () {
  theme_bw() +
    theme(axis.text = element_text(size = 14),
          axis.title = element_text(size = 14),
          strip.text = element_text(size = 14),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          panel.background = element_blank(),
          legend.position = "bottom",
          strip.background = element_rect(fill = 'white', colour =
'white'))
}
conf = data.frame(conf / rowSums(conf))
ggplot() +
  geom_tile(data = conf, aes(x = trainY, y = pred_train, fill = Freq))
+
  labs(x = "Actual", y = "Predicted", fill = "Proportion") +
  my_theme() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
   scale_fill_continuous(breaks = seq(0, 1, 0.25)) +
  coord_fixed()
```



Wir sehen aus diesem Diagramm, dass die meisten Klassen genau vorhergesagt werden, da sich das Hellblau (hohe Prozentsätze korrekter Vorhersagen) auf der Diagonale des Kacheldiagramms befinden. Wir können auch beobachten, dass die Kategorien, die am häufigsten verwechselt werden, Hemden, Tops, Pullover und Mäntel sind, was Sinn macht, da dies alles meist Oberkörperbekleidungsteile mit ähnlichen Formen sind. Das Modell sagt Hosen, Taschen, Stiefel und Turnschuhe gut voraus, da diese Zeilen und Spalten bis auf das diagonale Element besonders dunkel sind. Diese Ergebnisse stimmen mit denen aus dem Random Forest und den gradientenverstärkten Bäumen aus dem vorherigen Bbeitrag dieser Serie überein.

Als nächstes wiederholen wir den obigen Prozess zum Anpassen einer Support-Vektor-Maschine, aber anstelle einer zufälligen Suche nach den optimalen Parametern führen wir eine Gittersuche durch. Daher können wir Werte vorgeben, mit denen das Modell bewertet werden soll, nicht nur für `C`, sondern auch für `sigma`. Wir definieren die Gitterwerte in `svm_grid_radial`.

```
svm_grid_radial = expand.grid(sigma = c(.01, 0.04, 0.1), C = c(0.01,
10, 32, 70, 150))
set.seed(815)
svm_grid_radial = train(label ~ .,
                            data = cbind(train.images.pca, label =
train.classes),
                            method = "svmRadial",
                            trControl = svm_control,
                            tuneGrid = svm_grid_radial,
                            metric = "Accuracy")
svm_grid_radial
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 60000 samples
##    17 predictor
##    10 classes: 'Bag', 'Boot', 'Coat', 'Dress', 'Pullover',
'Sandal', 'Shirt', 'Sneaker', 'Top', 'Trouser'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 48000, 48000, 48000, 48000, 48000,
48000, ...
## Resampling results across tuning parameters:
##
##   sigma  C       Accuracy   Kappa      Mean_F1    Mean_Sensitivity
##   0.01     0.01  0.7572200  0.7302444  0.7559290  0.7572200
##   0.01    10.00  0.8596267  0.8440296  0.8588098  0.8596267
##   0.01    32.00  0.8662200  0.8513556  0.8654531  0.8662200
##   0.01    70.00  0.8690533  0.8545037  0.8683006  0.8690533
##   0.01   150.00  0.8712733  0.8569704  0.8705124  0.8712733
##   0.04     0.01  0.7921867  0.7690963  0.7909768  0.7921867
##   0.04    10.00  0.8749100  0.8610111  0.8741338  0.8749100
##   0.04    32.00  0.8758800  0.8620889  0.8751705  0.8758800
```

```
##   0.04    70.00  0.8744933  0.8605481  0.8738728  0.8744933
##   0.04   150.00  0.8720867  0.8578741  0.8714726  0.8720867
##   0.10     0.01  0.8008733  0.7787481  0.7997988  0.8008733
##   0.10    10.00  0.8755367  0.8617074  0.8749494  0.8755367
##   0.10    32.00  0.8690567  0.8545074  0.8685283  0.8690567
##   0.10    70.00  0.8638467  0.8487185  0.8633855  0.8638467
##   0.10   150.00  0.8585633  0.8428481  0.8580842  0.8585633
##   Mean_Specificity  Mean_Pos_Pred_Value  Mean_Neg_Pred_Value
Mean_Precision
##   0.9730244         0.7605863            0.9731197
0.7605863
##   0.9844030         0.8588181            0.9844323
0.8588181
##   0.9851356         0.8654163            0.9851625
0.8654163
##   0.9854504         0.8682628            0.9854768
0.8682628
##   0.9856970         0.8704544            0.9857234
0.8704544
##   0.9769096         0.7917107            0.9769608
0.7917107
##   0.9861011         0.8741060            0.9861283
0.8741060
##   0.9862089         0.8751726            0.9862341
0.8751726
##   0.9860548         0.8738208            0.9860763
0.8738208
##   0.9857874         0.8713720            0.9858082
0.8713720
##   0.9778748         0.8001941            0.9779182
0.8001941
##   0.9861707         0.8748767            0.9861909
0.8748767
##   0.9854507         0.8683827            0.9854680
0.8683827
##   0.9848719         0.8632713            0.9848872
0.8632713
##   0.9842848         0.8579941            0.9843011
0.8579941
##   Mean_Recall  Mean_Detection_Rate  Mean_Balanced_Accuracy
##   0.7572200    0.07572200           0.8651222
##   0.8596267    0.08596267           0.9220148
##   0.8662200    0.08662200           0.9256778
##   0.8690533    0.08690533           0.9272519
##   0.8712733    0.08712733           0.9284852
##   0.7921867    0.07921867           0.8845481
##   0.8749100    0.08749100           0.9305056
##   0.8758800    0.08758800           0.9310444
##   0.8744933    0.08744933           0.9302741
##   0.8720867    0.08720867           0.9289370
```

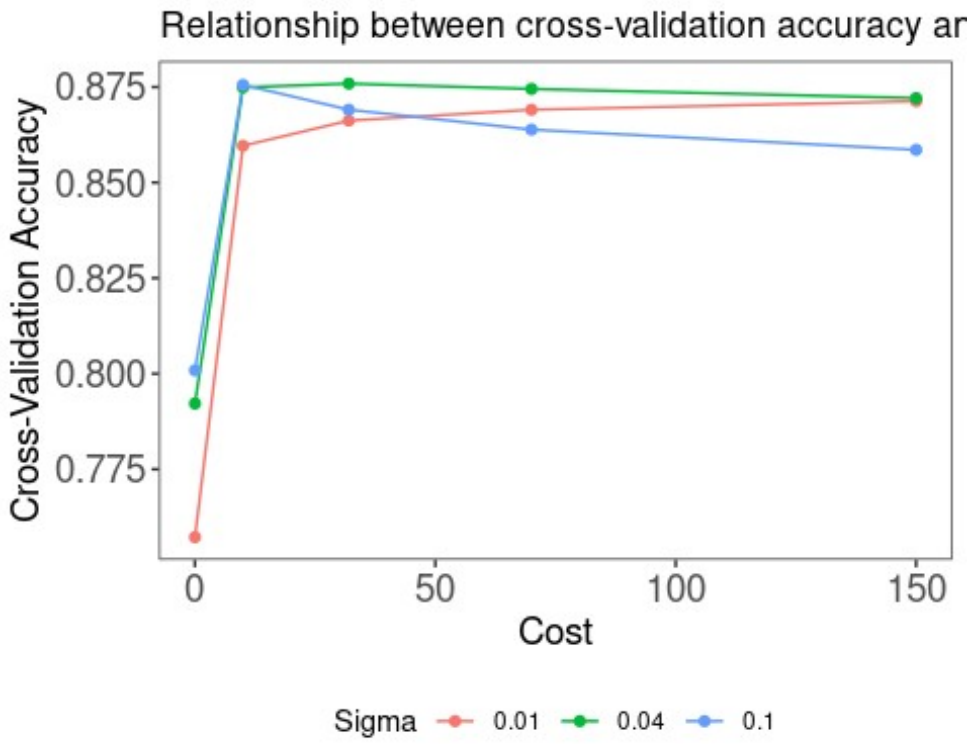
```
##   0.8008733    0.08008733           0.8893741
##   0.8755367    0.08755367           0.9308537
##   0.8690567    0.08690567           0.9272537
##   0.8638467    0.08638467           0.9243593
##   0.8585633    0.08585633           0.9214241
##
## Accuracy was used to select the optimal model using the largest
value.
## The final values used for the model were sigma = 0.04 and C = 32.
```

```
mp.svm.grid.radial = model_performance(svm_grid_radial,
train.images.pca, test.images.pca,
                                       train.classes, test.classes,
"svm_grid_radial")
```

```
##   accuracy_train precision_train recall_train  F1_train
accuracy_test
## 1        0.87588       0.9551978    0.9672333 0.9611779
0.8684
##   precision_test recall_test   F1_test           model
## 1      0.9576355       0.972 0.9647643 svm_grid_radial
```

Die Rastersuche wählt die gleichen optimalen Parameterwerte wie die Zufallssuche (`C=32` und `sigma = 0.040`), was ebenfalls zu 88 % und 87 % Trainings- und Testgenauigkeit führt. Um eine Vorstellung davon zu bekommen, wie `C` und `sigma` die Genauigkeit des Trainingssatzes beeinflussen, zeichnen wir die Genauigkeit der Kreuzvalidierung als Funktion von `C` auf, wobei Linien für jeden Wert von `sigma` getrennt sind.

```
ggplot() +
  my_theme() +
  geom_line(data = svm_grid_radial$results, aes(x = C, y = Accuracy,
color = factor(sigma))) +
  geom_point(data = svm_grid_radial$results, aes(x = C, y = Accuracy,
color = factor(sigma))) +
  labs(x = "Cost", y = "Cross-Validation Accuracy", color = "Sigma") +
  ggtitle('Relationship between cross-validation accuracy and values
of cost and sigma')
```



Das Diagramm zeigt, dass die grüne Linie (`sigma = 0.04`) die höchste Kreuzvalidierungsgenauigkeit für alle C-Werte aufweist, mit Ausnahme kleinerer C-Werte wie 0,01 und 10. Obwohl die Genauigkeit bei `C=10` und `sigma = 0.1` (blaue Linie) dieser nahe kommt, ist die höchste erreichte Gesamtgenauigkeit für `C=32` und `sigma=32` (grüne Linie). Werfen wir auch einen Blick auf den Leistungsunterschied zwischen der Verwendung eines radialen und eines linearen Kernels. Denken Sie daran, dass wir für einen linearen Kernel nur mögliche Werte für den Kostenparameter C und nicht `sigma` angeben müssen, da es hier klarerweise keine Nichtlinearität gibt.

```
svm_grid_linear = expand.grid(C = c(1, 10, 32, 75, 150))
set.seed(815)
svm_grid_linear = train(label ~ .,
                        data = cbind(train.images.pca, label =
train.classes),
                        method = "svmLinear",
                        trControl = svm_control,
                        tuneGrid = svm_grid_linear,
                        metric = "Accuracy")
svm_grid_linear
```

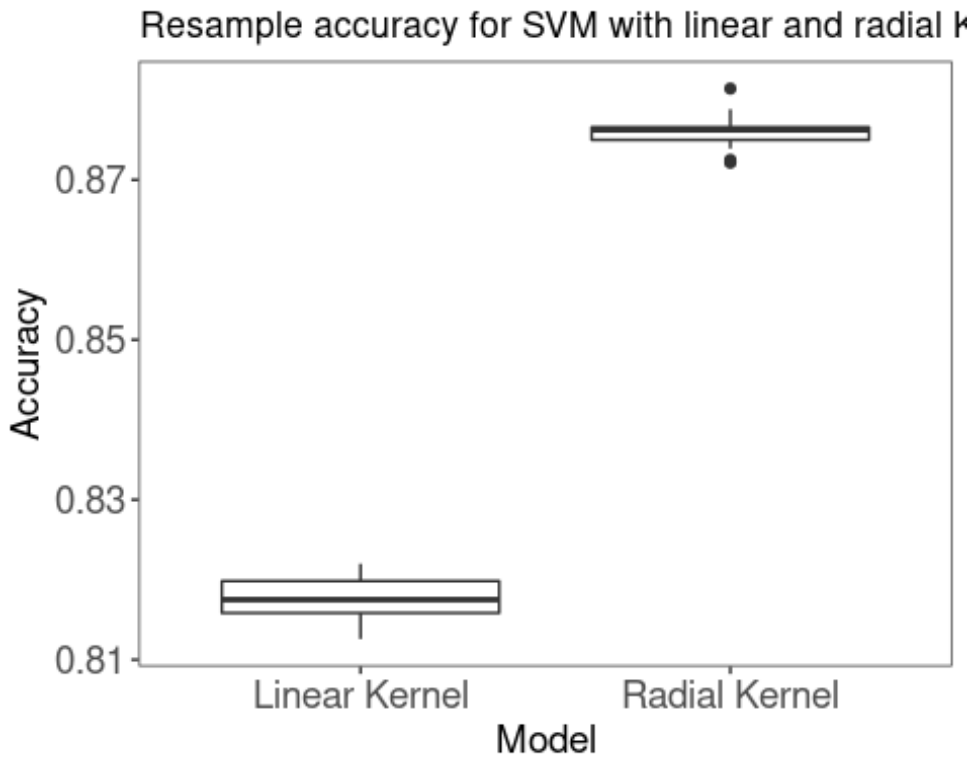
```
## Support Vector Machines with Linear Kernel
##
## 60000 samples
##    17 predictor
##    10 classes: 'Bag', 'Boot', 'Coat', 'Dress', 'Pullover',
'Sandal', 'Shirt', 'Sneaker', 'Top', 'Trouser'
```

```
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 48000, 48000, 48000, 48000, 48000,
48000, ...
## Resampling results across tuning parameters:
##
##   C    Accuracy   Kappa      Mean_F1    Mean_Sensitivity
Mean_Specificity
##     1  0.8171300  0.7968111  0.8144894  0.8171300         0.9796811

##    10  0.8175333  0.7972593  0.8148851  0.8175333         0.9797259

##    32  0.8175867  0.7973185  0.8149309  0.8175867         0.9797319

##    75  0.8176167  0.7973519  0.8149544  0.8176167         0.9797352

##   150  0.8175633  0.7972926  0.8149395  0.8175633         0.9797293

##   Mean_Pos_Pred_Value  Mean_Neg_Pred_Value  Mean_Precision
Mean_Recall
##   0.8144392            0.9797734            0.8144392
0.8171300
##   0.8148289            0.9798184            0.8148289
0.8175333
##   0.8148688            0.9798245            0.8148688
0.8175867
##   0.8148958            0.9798282            0.8148958
0.8176167
##   0.8148692            0.9798208            0.8148692
0.8175633
##   Mean_Detection_Rate  Mean_Balanced_Accuracy
##   0.08171300           0.8984056
##   0.08175333           0.8986296
##   0.08175867           0.8986593
##   0.08176167           0.8986759
##   0.08175633           0.8986463
##
## Accuracy was used to select the optimal model using the largest
value.
## The final value used for the model was C = 75.

mp.svm.grid.linear = model_performance(svm_grid_linear,
train.images.pca, test.images.pca,
                                       train.classes, test.classes,
"svm_grid_linear")

##   accuracy_train precision_train recall_train  F1_train
accuracy_test
## 1      0.8176167       0.9215471       0.9499 0.9355088
```

```
0.8095
##   precision_test recall_test   F1_test           model
## 1      0.9099617        0.95 0.9295499 svm_grid_linear
```

Wir können die neu abgetasteten Genauigkeiten der SVMs auch visuell mit linearen und radialen Kernal-Spezifikationen mithilfe eines Boxplots vergleichen. Dies ist möglich da wir vor dem Training beider Modelle denselben Random-Seed-Startwert festlegen.

```
resamp_val = resamples(list(svm_radial = svm_grid_radial, svm_linear =
svm_grid_linear))
plotdf = data.frame(Accuracy =
c(resamp_val$values$`svm_radial~Accuracy`,
resamp_val$values$`svm_linear~Accuracy`),
                    Model = rep(c("Radial Kernel", "Linear Kernel"),
rep(nrow(resamp_val$values), 2)))
ggplot() +
  geom_boxplot(data = plotdf, aes(x = Model, y = Accuracy)) +
  ggtitle('Resample accuracy for SVM with linear and radial Kernel') +

  my_theme()
```



Obwohl die lineare Kernel-Spezifikation ebenfalls C=32 auswählt, wie es die radiale Kernel-Spezifikation dies getan hat, schneidet sie schlechter ab als die radiale Kernel-Spezifikation (6 %) sowohl bei der Trainings- als auch bei der Testsatzgenauigkeit. Daher sieht es so aus, als ob ein nichtlineares Modell bei der Klassifizierung der Fashion-MNIST-Daten besser abschneidet.

Zusammenfassung

Vergleichen wir abschließend die Leistung aller Modelle, die wir uns angesehen haben, auch die der neuronalen Netze, die wir im ersten Bbeitrag dieser Serie entwickelt haben. Werfen wir einen Blick auf die Leistung der Modelle:

```
mp.df = rbind(mp.svm.rand.radial, mp.svm.grid.radial,
mp.svm.grid.linear)
mp.df[order(mp.df$accuracy_test, decreasing = TRUE), ]

##   accuracy_train precision_train recall_train  F1_train
accuracy_test
## 2      0.8758800       0.9551978    0.9672333 0.9611779
0.8684
## 1      0.8759467       0.9569655    0.9688000 0.9628464
0.8668
## 3      0.8176167       0.9215471    0.9499000 0.9355088
0.8095
##   precision_test recall_test   F1_test             model
## 2      0.9576355       0.972 0.9647643   svm_grid_radial
## 1      0.9549020       0.974 0.9643564 svm_random_radial
## 3      0.9099617       0.950 0.9295499   svm_grid_linear
```