Vergleich von verschiedenen KI-Klassifizierungsmethoden am Beispiel des Fashion MNIST-Datensatzes in RStudio und Python (Teil 2)

Lars Mehnen

11/03/2022

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. Im ersten Blogbeitrag dieser Serie haben wir erstmal die Daten untersucht, für die Analyse vorbereitet und gelernt, wie man die Kleidungskategorien der Fashion MNIST-Daten mithilfe eines Go-to-Modells klassifiziert: einem künstlichen neuronalen Netzwerk in Python. In diesem zweiten Blogbeitrag werde wir eine Dimensionsreduktion an den Daten durchführen, um einige der maschinellen Lernmodelle zu beschleunigen, die wir in den nächsten Beiträgen ausführen werden (einschließlich baumbasierter Methoden und Support-Vektor-Maschinen), und untersuchen, ob diese Modelle in Verbindung mit den reduzierten Daten eine ähnliche Güte wie die neuronalen Netze aus dem ersten Beitrag dieser Serie erreichen kann.

Daten Vorbereiten

Wir haben bereits gesehen wie Sie die Daten laden, einige der Bilder der Daten anzeigen und sie für die weitere Analyse vorbereiten. Deshalb werden wir das hier nicht noch einmal im Detail ausführen.

```
library(devtools)
devtools::install_github("rstudio/keras")
library(keras)
#install_keras()
fashion_mnist = keras::dataset_fashion_mnist()
```

Wir erhalten separate Datensätze für die Trainings- und Testbilder sowie die Trainings- und Testlabels.

```
library(magrittr)
c(train.images, train.labels) %<-% fashion_mnist$train
c(test.images, test.labels) %<-% fashion_mnist$test
```

Als nächstes normalisieren wir die Bilddaten, indem wir die Pixelwerte durch den Maximalwert von 255 dividieren.

```
train.images = data.frame(t(apply(train.images, 1, c))) / max(fashion_mnist$train$x)
test.images = data.frame(t(apply(test.images, 1, c))) / max(fashion_mnist$train$x)
```

Wir führen dann die Trainingsbilder `train.images` und Labels `train.labels` und die Testbilder `test.images` and Labels `test.labels` in jeweils die separaten Datensätzen `train.data` und `test.data` zusammen.

```
pixs = ncol(fashion_mnist$train$x)
names(train.images) = names(test.images) = paste0('pixel', 1:(pixs^2))
train.labels = data.frame(label = factor(train.labels))
test.labels = data.frame(label = factor(test.labels))
```

```
train.data = cbind(train.labels, train.images)
test.data = cbind(test.labels, test.images)
```

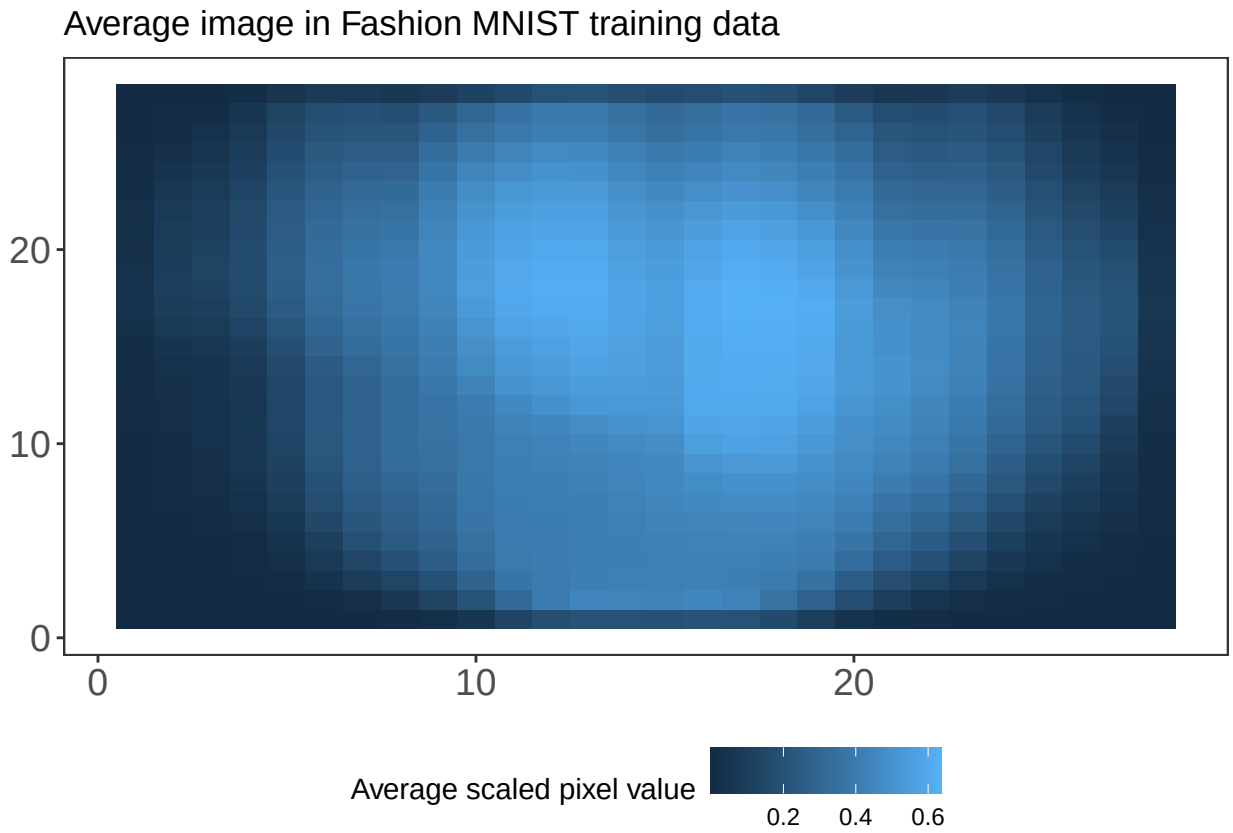
Da `train.labels` und `test.labels` lediglich ganzzahlige Werte für die Kleidungs-Kategorie / -Klasse enthalten (z. B. 0, 1, 2 usw.), erstellen wir nun die Objekte train.classes und test.classes, die die Klassennamen (z. B. Top, Hose, Pullover usw.). .) für die Bekleidungsklasse inkludieren.

```
cloth_cats = c('Top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Boot')
train.classes = factor(cloth_cats[as.numeric(as.character(train.labels$label)) + 1])
test.classes = factor(cloth_cats[as.numeric(as.character(test.labels$label)) + 1])
```

Principal Components Analysis (PCA)

Unsere Trainings- und Testbilddatensätze enthalten derzeit 784 Pixel und damit Variablen. Wir können davon ausgehen, dass ein großer Anteil dieser Pixel (insbesondere die an den Rändern der Bilder), relativ geringe Varianz aufweisen, da die meisten Modeartikel in den Bildern zentral positioniert sind. Mit anderen Worten, unser Datensatz wird einige redundante Pixel enthalten. Um zu überprüfen, ob dies der Fall ist, zeichnen wir den durchschnittlichen Pixelwert in einem 28 x 28-Raster auf. Wir erhalten zuerst die durchschnittlichen Pixelwerte und speichern diese in `train.images.ave`, danach plotten wir diese Werte in ein Raster (siehe unten). Wir definieren auch ein benutzerdefiniertes Plotthema `my_theme`, um sicherzustellen, dass alle unsere Bilder die gleiche benutzerdefinierte Farbskala haben. Bitte beachten Sie, dass in dem erstellten Bild ein höherer Pixel-Wert bedeutet, dass der Mittelwert dieses Pixels höher ist und dass das Pixel im Durchschnitt dunkler ist (Pixelwert von 0 ist Weiß und einer von 255 ist schwarz).

```
train.images.ave = data.frame(pixel = apply(train.images, 2, mean),
                              x = rep(1:pixs, each = pixs),
                              y = rep(1:pixs, pixs))
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'))
}
ggplot() +
  geom_raster(data = train.images.ave, aes(x = x, y = y, fill = pixel)) +
  my_theme() +
  labs(x = NULL, y = NULL, fill = "Average scaled pixel value") +
  ggtitle('Average image in Fashion MNIST training data')
```



Wie wir im Bild sehen können, gibt es einige Pixel mit einem recht niedrigen Durchschnittswert. Das bedeutet, dass sie in den meisten Bildern unserer Trainingsdaten hell (weiß) sind. Diese Pixel sind hier also redundant, tragen aber zum Rechenaufwand bei. Daher sind wir hier möglicherweise besser dran, die Dimensionalität in unseren Daten zu reduzieren, um Redundanz, Überanpassung und Rechenaufwand zu minimieren. Eine Methode hierfür ist die Hauptkomponentenanalyse (PCA), die wir heute anwenden. Im Wesentlichen reduziert die PCA die Dimensionen eines Datensatzes korrelierter Variablen mit Hilfe von Statistik, indem sie sie in eine kleinere Anzahl linear unkorrelierter Variablen (Hauptkomponenten) umwandelt, die im wesentlichen linear Kombinationen der ursprünglichen Variablen sind. Die erste Hauptkomponente beinhaltet den Hauptteil der Varianz, gefolgt von der zweiten Hauptkomponente und so weiter. Für eine ausführlichere Erklärung von PCA verweise ich Sie auf das AIAV Video über die PCA. Schauen wir uns zunächst an, wie viele Variablen welchen Anteil der Varianz in unseren Daten erklären können. Wir berechnen die 784 x 784 Kovarianzmatrix unserer Trainingsbilder mit der `cov()`-Funktion. Danach führen wir die PCA auf der Kovarianzmatrix mit der `prcomp()`-Funktion in der `stats`-Bibliothek aus. Wenn wir uns die Ergebnisse ansehen, stellen wir fest, dass die ersten 50 Hauptkomponenten in unseren Daten 99,902 % der Varianz in den Daten erklären können. Dies lässt sich gut in einem Diagramm des kumulierten Anteils der Varianz gegen die Komponentenindizes zeigen. Beachten Sie, dass die Komponentenindizes hier nach ihrer Fähigkeit sortiert sind, die Varianz in unseren Daten zu erklären, und nicht nach ihrer Pixelposition im 28 x 28-Bild.

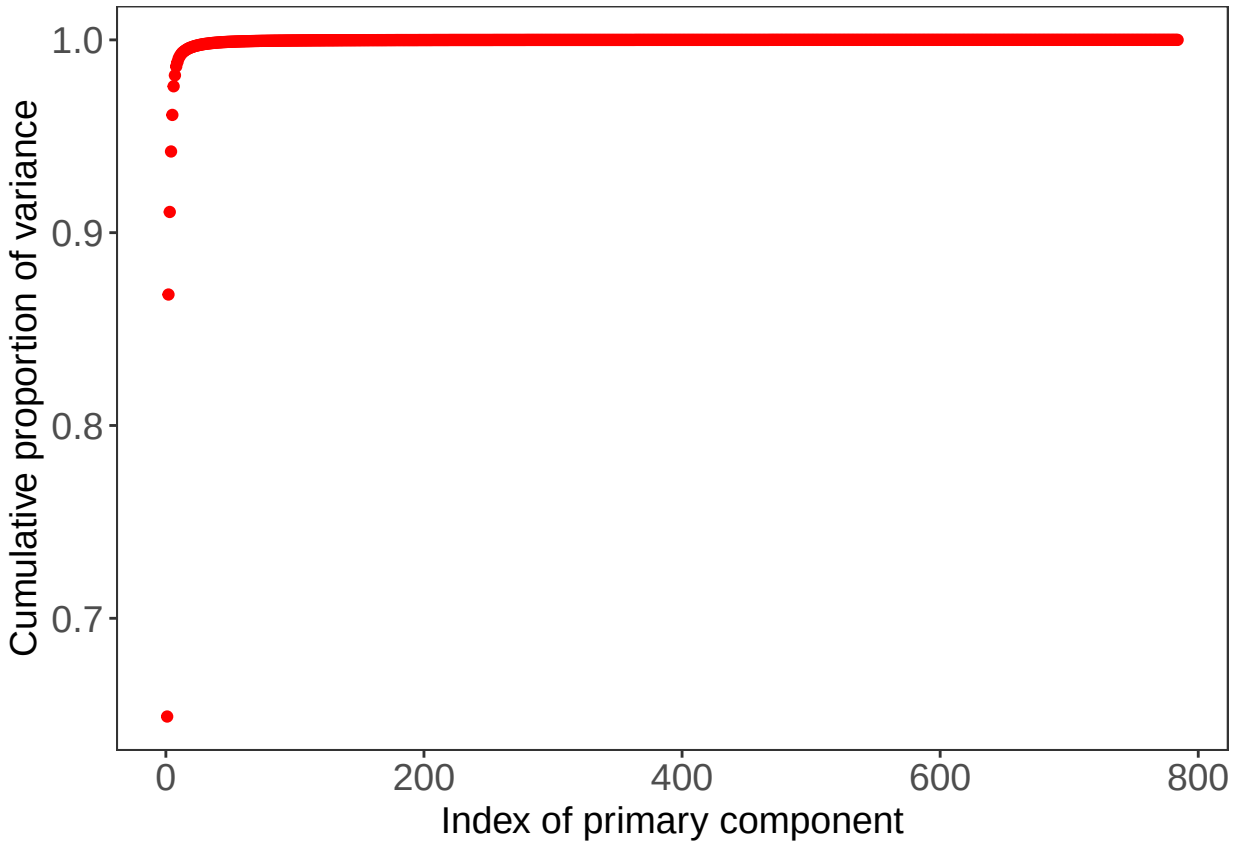
```
library(stats)
cov.train = cov(train.images)
pca.train = prcomp(cov.train)

plotdf = data.frame(index = 1:(pixs^2),
                    cumvar = summary(pca.train)$importance["Cumulative Proportion", ])
t(head(plotdf, 50))
```

```
##            PC1     PC2     PC3     PC4     PC5     PC6     PC7     PC8     PC9
## index  1.00000 2.00000 3.00000 4.00000 5.00000 6.00000 7.00000 8.00000 9.00000
```

```
## cumvar 0.64905 0.86793 0.91075 0.94212 0.96108 0.97593 0.98162 0.98622 0.98854
##            PC10     PC11     PC12     PC13     PC14     PC15     PC16     PC17
## index  10.00000 11.00000 12.00000 13.00000 14.00000 15.00000 16.00000 17.00000
## cumvar  0.99059  0.99182  0.99279  0.99352  0.99408  0.99454  0.99498  0.99537
##            PC18     PC19     PC20     PC21     PC22     PC23     PC24     PC25
## index  18.00000 19.00000 20.00000 21.00000 22.00000 23.00000 24.00000 25.00000
## cumvar  0.99572  0.99599  0.99624  0.99647  0.99669  0.99687  0.99705  0.99721
##            PC26     PC27     PC28     PC29     PC30     PC31     PC32     PC33
## index  26.00000 27.00000 28.00000 29.00000 30.00000 31.00000 32.00000 33.00000
## cumvar  0.99737  0.99751  0.99764  0.99776  0.99787  0.99796  0.99805  0.99814
##            PC34    PC35     PC36     PC37    PC38     PC39     PC40     PC41
## index  34.00000 35.0000 36.00000 37.00000 38.0000 39.00000 40.00000 41.00000
## cumvar  0.99822  0.9983  0.99837  0.99844  0.9985  0.99856  0.99862  0.99867
##            PC42     PC43     PC44     PC45     PC46     PC47     PC48     PC49
## index  42.00000 43.00000 44.00000 45.00000 46.00000 47.00000 48.00000 49.00000
## cumvar  0.99872  0.99877  0.99881  0.99885  0.99888  0.99892  0.99896  0.99899
##            PC50
## index  50.00000
## cumvar  0.99902
```

```
ggplot() +
  geom_point(data = plotdf, aes(x = index, y = cumvar), color = "red") +
  labs(x = "Index of primary component", y = "Cumulative proportion of variance") +
  my_theme() +
  theme(strip.background = element_rect(fill = 'white', colour = 'black'))
```



Wir haben gesehen, dass 99,5 % der Varianz durch nur 17 Hauptkomponenten erklärt werden. Da ein 0,5 % Fehler tollerabel ist und wir die Anzahl der Pixel (Variablen) so weit wie möglich reduzieren möchten,

um die Rechenzeit für die kommenden Modelle zu reduzieren, wählen wir nun diese 17 Komponenten aus. Wir speichern auch den relevanten Teil der Rotationsmatrix, die von der `prcomp()`-Funktion erstellt und in `pca.train` gespeichert wird, sodass ihre Dimension jetzt 784 x 17 beträgt. Dann multiplizieren wir unsere Trainings- und Testbilddaten mit dieser Rotation Matrix `pca.rot`. Anschließend führen wir die transformierten Bilddaten (`train.images.pca` und `test.images.pca`) mit den ganzzahligen Labels für die Bekleidungskategorien in `train.data.pca` und `test.data.pca` wieder zusammen. Wir werden nun diese reduzierten Daten in unseren weiteren Analysen verwenden, um die Rechenzeit zu verkürzen.

```
pca.dims = which(plotdf$cumvar >= .995)[1]
pca.rot = pca.train$rotation[, 1:pca.dims]

train.images.pca = data.frame(as.matrix(train.images) %*% pca.rot)
test.images.pca  = data.frame(as.matrix(test.images) %*% pca.rot)

train.data.pca = cbind(train.images.pca, label = factor(train.data$label))
test.data.pca = cbind(test.images.pca, label = factor(test.data$label))
```

Model Performanz

Um die Modelle, die wir in diesem Beitrag bearbeiten, einfacher vergleichen zu können, erstellen wir nun die Funktion `model_performance()`, die in der Lage ist, einige Leistungsmetriken für jeden Modelltyp auszugeben, den wir schätzen werden (Random Forest, Gradient-Boosted Trees, Support Vector Machines). Die Funktion sagt im Wesentlichen das geschätzte Modell sowohl für die Trainingsdaten als auch für die Testdaten voraus (`pred_train`, `pred_test`) und berechnet dann die Maße für Genauigkeit, Präzision (auch positiver Vorhersagewert genannt), Recall (auch als Sensitivität bekannt, true_positives dividiert durch die Summe von true_positives und false_negatives) und das F1-Maß sowohl für die Trainings- als auch für die Testdatensatzvorhersagen. Sehen Sie sich diesen Blogbeitrag an, wenn Sie sich nicht sicher sind, was diese Leistungskennzahlen bedeuten.

Je nach Modelltyp erfordern die Eingaben `testX` und `testY` manchmal die Kategorie-Klassen (z. B. Oberteil, Hose, Pullover usw.) wie z.B. in `train.classes` und `test.classes`, während andere Modelle wie der `random forests` Schätzer, die ganzzahligen Klassen (z. B. 0, 1, 2 usw.) schätzen und daher Klassen wie in `train.data$label` und `test.data$label` erfordern. Beachten Sie, dass wir für die mit dem `caret`-Paket implementierten Modelle die Out-of-Bag-Vorhersagen verwenden müssen, die in den Modellobjekten enthalten sind (`fit$pred`), und nicht die manuell berechneten In-Sample-Vorhersagen (non-out-of-bag) Vorhersagen für die Trainingsdaten. Da `fit$pred` die Vorhersagen für alle vom Benutzer angegebenen Tuning-Parameterwerte enthält, während wir nur die Vorhersagen benötigen, die zu den optimalen Tuning-Parameterwerten gehören, nehmen wir aus `fit$pred` nur diese Vorhersagen und Beobachtungen, die in den Indizes-`rows` enthalten sind. Beachten Sie auch, dass wir `fit$pred` in ein `data.table`-Objekt umwandeln, um diese Indizes zu erhalten, da Berechnungen auf `data.table`-Objekten für große Datensätze viel schneller sind (z.B. hat `xgb_tune$pred` über 29 Millionen Zeilen, wie wir später sehen werden).

```
model_performance = function(fit, trainX, testX, trainY, testY, model_name){

  # Predictions on train and test data for models estimated with caret
  if (any(class(fit) == "train")){

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

```
    pred_test = predict(fit, newdata = testX)
    trainY = fit$pred$obs[rows]

  } else {

    print(paste0("Error: Function evaluation unknown for object of type ", class(fit)))
    break

  }

  # Performance metrics on train and test data
  library(MLmetrics)
  df = data.frame(accuracy_train = Accuracy(trainY, pred_train),
                  precision_train = Precision(trainY, pred_train),
                  recall_train = Recall(trainY, pred_train),
                  F1_train = F1_Score(trainY, pred_train),
                  accuracy_test = Accuracy(testY, pred_test),
                  precision_test = Precision(testY, pred_test),
                  recall_test = Recall(testY, pred_test),
                  F1_test = F1_Score(testY, pred_test),
                  model = model_name)

  print(df)

  return(df)
}
```