Facial Keypoint Detection - Documentation

Henry Perschk, Lucas Stegger

7 1 2017

Contents

1 Introduction

The fkdR (short for Facial Keypoints Detection in R) package is intended for predicting facial keypoint positions like the centers of the eyes or the nose tip for example. It furthermore provides functionality to train models for facial keypoint recognition. This is achieved by using the mlR package to make use of a standardized interface for e.g. resampling or tuning. As of now we implemented a patch search learner with mlR to show the concept of our package and the way image data is handled. Our research showed, that best results for image recognition by machine-learning are achieved by Deep Neural Nets. Thus, we moreover composed first a multi-layer-perceptron as well as a convolutional neural net, both based on tensorflow. The data we use in this package is based on the Facial Keypoints Detection contest posted on Kaggle.

- structure of package?
- structure of documentation

2 Loading and Preprocessing the Data

- structure of data
- 96 x 96 pixel
- as column-wise vector
- NAs
- histogram equalization
- outlook
- transformation in 1-dimensional data

3 Patch Search Learner

The patch search learner we implemented is based on the Getting Started Tutorial from the Facial Keypoints Detection contest. We used the logic and realized it using `mlR`, which allowed us to make use of `mlR`'s standardized interface for machine learning. Thus, we tried to improve the results by making use of tuning and resampling. As usual the patch search algorithm can be devided into training and prediction.

The basic idea behind the patch search learner is to extract all pixels with a specific radius around each keypoint in each image. The extracted pixels are refered to as patch. Afterwards, for each keypoint the mean patch is calculated by calculating the mean of each pixel. This mean patch can in a next step be used on any image containing the keypoint (assuming it is also a 96x96 pixel grayscale image, where the face ideally covers the whole width and height of the image and is horizontally aligned). Predicting the keypoint is achieved by running through the desired image pixel by pixel and again extracting the the patch around this pixel. The point, where the extracted patch best matches the trained mean patch is used as prediction.

3.1 Training a Patch for a Facial Keypoint

As indicated the first step is to extract the patch for each keypoint from each training image. One limitation using mlR was, that mlR as of now does not offer an interface for regression with more than one targets. This becomes a problem, as we are trying to train and predict 15 keypoints which consist of two coordinates. As explained in section 2, we solved the problem of predicting two coordinates by transforming the two coordinates into one value. As long as the width and height of the images is known, this value can easily be converted back to two coordinates.

The problem of having 15 different keypoints can be solved by training 15 models for only one keypoint each. Normally, this can be a limitation, because the information where the eyes lie for example might give us some information about the location of the nose tip. As the Patch Search Learner is not able to respect these interdependencies, we can ignore this disadvantage. However, it has to be kept in mind and one might have to think of a different solution when using advaned learners, that are able to respect these interdependencies. One solution could be to extend the mlR functionality by an interface for multi-target regression.

Furthermore, all patches are squares where the center point is the keypoint and the width as well as height are calculated by `patch_size * 2 + 1`. This allows easier handling and storing of the patch as it would be the case with a circle. The variable `patch_size` is therefore the first of two hyperparameters the Patch Search Learner provides.

Based on the aforementioned assumptions the learner is implemented. First, a new mlR learner is initialized by ‘makeRLearnerRegr()‘as a regression learner. As indicated in the code below the two hyperparameters are defined in this function. The ’search_size’ hyperparameter is relevant for the predictions and will therefore further explained during the section 3.2.

```
makeRLearner.regr.patchSearch = function() {
  makeRLearnerRegr(
    cl = "regr.patchSearch",
    package = "fkdR",
    par.set = makeParamSet(
      makeNumericLearnerParam(id = "patch_size", default = 10, lower = 0, tunable = TRUE),
      makeNumericLearnerParam(id = "search_size", default = 2, lower = 1, tunable = TRUE)
    ),
    properties = c("numerics"),
    name = "Patch Search Learner for Keypoint Detection in Images",
    short.name = "patchSearch",
    note = ""
  )
}
```

Next step is to define the regression part of the learner. To make it visible for `mlR` the function name has to be constructed of the term "trainLearner" and the name we defined in the aforementioned `makeRLearnerRegr()` function. Relevant data are passed to our custom function for trinaing the mean patch for the current keypoint. These are the formula of the current task (containing the name of the target keypoint) and the training data.

```
trainLearner.regr.patchSearch = function(.learner, .task, .subset, ...) {
  patchSearch.train(f = getTaskFormula(.task),
                    d.tr = getTaskData(.task, .subset),
                    ...)
}
```

Respecting the assumptions from the beginning of this section training data has to be a data frame with two columns, the first containing the images as vector, the second containing the desired keypoint as single value. Next to these training data the task formula and hyperparameters are parameters of the custom Patch Search training function. The desired keypoint is extracted from the formula by using `all.vars(f)[1]`.

```
patchSearch.train <- function(f, d.tr, patch_size = 10, search_size = 2) {
  coord = all.vars(f)[1]
  cat(sprintf("computing mean patch for %s\n", coord))
```

Next step is to extract all patches for the current keypoint. The following code runs through all training images and first converts them to a 96x96 matrix. Afterwards the keypoint value is transformed back to x and y coordinate. In a next step the outer coordinates of the patch can be calculated using the `patch_size` hyperparameter. All patches are returned as vector and stored in the variable `patches`. We are using the `doParallel` package to speed up the calculation.

```
  # extract all patches
  patches <- foreach (i = 1:nrow(d.tr), .combine=rbind) %do% {
    if ((i %% 100)==0) { cat(sprintf("Extracting %s patch from training image %d/%d\n",
                                     coord, i, nrow(d.tr))) }
    im  <- matrix(data = d.tr[i,"Image"], nrow=96, ncol=96)

    # transform to x and y coordinate
    xy  <- d.tr[i, coord]
    x   <- xy %/% 96 + 1
    y   <- xy %% 96

    # determine outer coordinates of patch
    x1  <- (x-patch_size)
    x2  <- (x+patch_size)
    y1  <- (y-patch_size)
    y2  <- (y+patch_size)
    if ( (!is.na(x)) && (!is.na(y)) && (x1>=1) && (x2<=96) && (y1>=1) && (y2<=96) )
    {
      as.vector(im[x1:x2, y1:y2])
    }
    else
    {
      NULL
    }
  }
```

As soon as all patches are extracted, we can calculate the mean of them. At this point, we decided to plot the mean patch to get some insights in the mean patch calculation. Figure 1 shows one of these mean patches. Although it is based on so many different faces, one can still clearly see a mouth and the bottom part of a

nose on this figure. By returning the mean patch at the end of the function it is stored as the model internally in `mlR` and can be accessed during the predictions.

```
  # return mean patch
  mean.patch = matrix(data = colMeans(patches), nrow=2*patch_size+1, ncol=2*patch_size+1)

  # plot mean patch
  par(mar = rep(0, 4))
  image(1:(2*patch_size+1), 1:(2*patch_size+1),
        mean.patch[nrow(mean.patch):1,ncol(mean.patch):1],
        col=gray((0:255)/255), xaxt = "n", yaxt = "n", ann = FALSE, breaks = 0:256)

  # return the mean patch
  mean.patch
}
```

3.2 Predicting a Facial Keypoint using a Patch

Having computed the mean patch for a particular keypoint, we can start with predictons. For the predictions we run through the images and search for the best match of the mean patch. As it would be very inefficient and expensive to compare the match to all possible patch positions, the Patch Search Learner only searches around the mean keypoint positions of the training dataset. The size of the area around this mean keypoint, where the learner searches, is determined by the hyperparameter `search_size`. Similar to the identification of the outer coordinates of the patch, the search area is determined by setting its center to the mean position and the width and height to `2 * search_size + 1`. Afterwards the algorithm runs through every pixel in the search area and the mean patch's center is set to the pixel in order to calculate the match. The position in the search area with the best match is used as prediction.

Again we are using the `mlR` interface to provide the functionality. Therefore, we define the following function to make the prediction part of the learner available to `mlR`. Before handing over the data to our custom function, we extract the hyperparameters of the current model and to be sure they have some value set them to the default values in case they are `NULL`.

```
predictLearner.regr.patchSearch = function(.learner, .model, .newdata, ...) {
  .patch_size = .model$learner$par.vals$patch_size
  .search_size = .model$learner$par.vals$search_size

  .patch_size = ifelse(is.null(.patch_size), 10, .patch_size)
  .search_size = ifelse(is.null(.search_size), 2, .search_size)

  patchSearch.predict(.model,
                      f = getTaskFormula(.model$task.desc),
                      d.te = .newdata,
                      patch_size = .patch_size,
                      search_size = .search_size,
                      ...)
}
```

Our custom function accepts the model, the formula, test data and the hyperparameters as parameters. First, the mean patch is extracted from the model and the target variable name is read from the formula.

```
patchSearch.predict <- function(model, f, d.te, patch_size = 10, search_size = 2)  {
  mean.patch = model$learner.model

  # the coordinates we want to predict
```

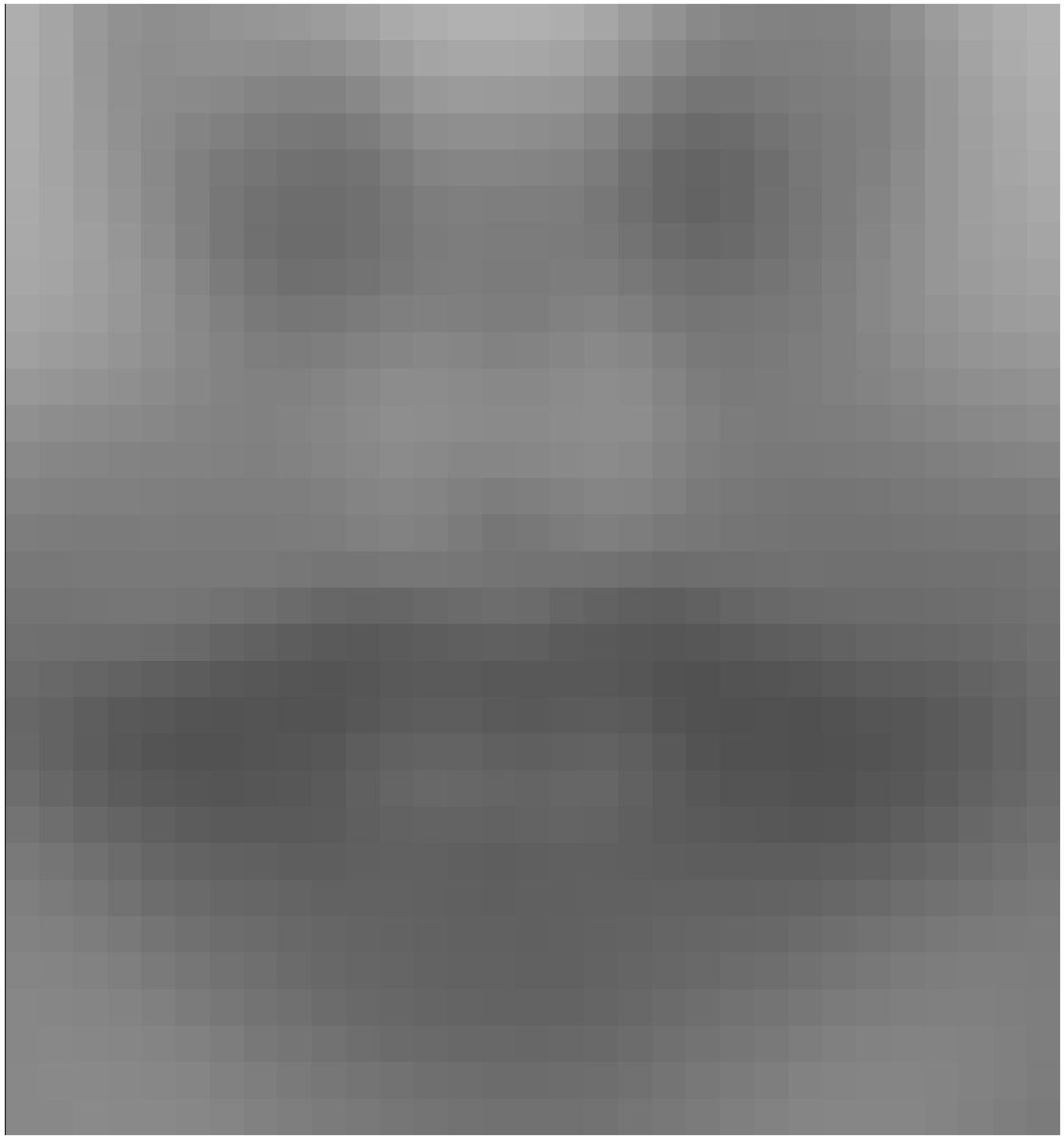

Figure 1: Mean patch for mouth center top lip with ‘patch_size = 15’

```
  coord = all.vars(f)[1]
  coord_x <- paste(coord, "x", sep="_")
  coord_y <- paste(coord, "y", sep="_")
```

Now the mean of the x and y coordinate is calculated based on the training data. By using the hyperparameter `search_size` we can calculate the search area. As the mean positions of a keypoint could lie so close to the border, that the patch would not lie completely inside the image, we have to cut the search area by these pixels. The variable `params` then contains all coordinates in the search area.

```
  # the average of them in the training set (our starting point)
  mean_x  <- mean(d.train[, coord_x], na.rm=T)
  mean_y  <- mean(d.train[, coord_y], na.rm=T)

  # search space: 'search_size' pixels centered on the average coordinates
  x1 <- as.integer(mean_x)-search_size
  x2 <- as.integer(mean_x)+search_size
  y1 <- as.integer(mean_y)-search_size
  y2 <- as.integer(mean_y)+search_size

  # ensure we only consider patches completely inside the image
  x1 <- ifelse(x1-patch_size<1,  patch_size+1,  x1)
  y1 <- ifelse(y1-patch_size<1,  patch_size+1,  y1)
  x2 <- ifelse(x2+patch_size>96, 96-patch_size, x2)
  y2 <- ifelse(y2+patch_size>96, 96-patch_size, y2)

  # build a list of all positions to be tested
  params <- expand.grid(x = x1:x2, y = y1:y2)
```

We can now run through all test images and search for the best match of the models mean patch in the search area. The outer loop is responsible for running through the images, whereas the inner loop runs through all coordinate combinations in the search area. For each coordinate combination the patch with the same size as the models mean patch is extracted. Both, the extracted patch and the mean patch are then converted to a vector and the match between them is calculated by the correlation amongst them. Afterwards, the coordinates and the correlation is stored in a data frame, such that we can easily extract the coordinates with the highest correlation value. The final step is to retransform the two coordinates back to a single value to make them compatible with the general data format we agreed upon. Thus, we have our prediction.

```
  # for each image...
  r <- foreach(i = 1:nrow(d.te), .combine=rbind) %do% {
    im <- matrix(data = d.te[i, "Image"], nrow=96, ncol=96)
    if ((i %% 100)==0) { cat(sprintf("Predicting %s for test image %d/%d\n",
                                     coord, i, nrow(d.te))) }

    # ... compute a score for each position ...
    r  <- foreach(j = 1:nrow(params), .combine=rbind) %do% {
      x     <- params$x[j]
      y     <- params$y[j]
      p     <- im[(x-patch_size):(x+patch_size), (y-patch_size):(y+patch_size)]
      score <- cor(as.vector(p), as.vector(mean.patch))
      score <- ifelse(is.na(score), 0, score)
      data.frame(x, y, score)
    }

    # ... and return the best
    best <- r[which.max(r$score), c("x", "y")]
```

```
    best
  }

  # retransform to single value
  result <- (round(r[1]) - 1) * 96 + round(r[2])
  result[1]$x
}
```

4 Tensorflow...

5 Discussion and Comparison

Patch Search * Patch Search is a rather simple yet effective and intuitive approach * limitations (composition of image, limited search area (pros and cons!, search area at the borders), rotation, etc..)

6 Conclusion and Outlook