

Facial Keypoint Detection - Documentation

Henry Perschk, Lucas Stegger

7 1 2017

Contents

1	Introduction	1
2	Loading and Preprocessing the Data	1
3	Patch Search Learner	1
3.1	Training a Patch for a Facial Keypoint	2
4	Tensorflow...	4
5	Discussion and Comparison	4
6	Conclusion	4

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 divided 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 referred 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 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 advanced 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 next section.

```
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 training 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
}

```

4 Tensorflow...

5 Discussion and Comparison

6 Conclusion

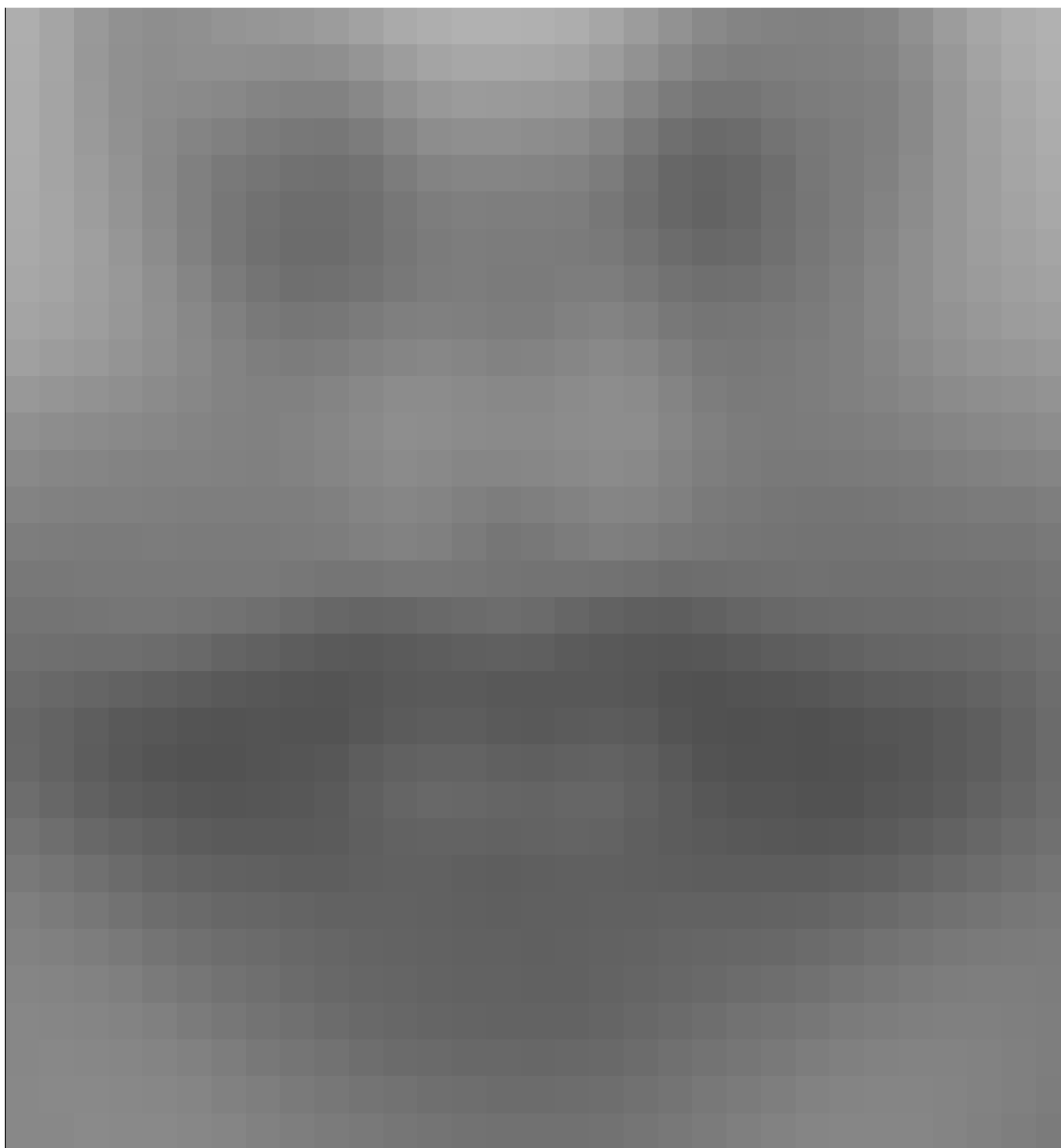


Figure 1: Mean patch for mouth center top lip with 'patch_size = 15'