Counting visitors with `wuepix`

Jeroen Staab

2018-10-29

The *wuepix* package counts visitor numbers using computer vision. Therefore three methods (Change Detection, HOG-Descriptor, YOLO-darknet) were wrapped into this package. Additional management tools, as a Ground-Truth-Data sampler, are also included here. This vignette demonstrates a typical workflow.

Packages

```r
# Installation
# devtools::install_github("georoen/wuepix")
library(wuepix)
library(tidyverse)
```

Site configuration

Paths & Filenames

Define the directory paths and filename patterns implied by the data archive.

```r
# Where to find Images?
## Raw data
img.folder_raw <- "IMG_raw/"
# Preprocessed (croped, scaled, enhanced,...)
img.folder  <- "IMG/"
# Remove corrupted images by filesize (in byte)
threshold <- 10000
# How to grep date?
gsub.Date <- function(Filename){gsub("picam-", "", gsub(".jpg", "", Filename))}
# Date code
date.code <- "%Y%m%d-%H%M"

# Aggregation Scale
T_scale <- "20 mins"

# Hubland
# threshold <- 1000
# gsub.Date <- function(Filename){gsub("Camera1_M_", "", gsub(".jpg", "", Filename))}
# date.code <- "%Y-%m-%d-%H%M%S"
```

Extent of interest

To speed up processing an extend of interest (EOI) should be selected. Using the Linux comandline tool *ImageMagick*, this can also include rotations as well as other image operations. However identifying the correct command involves visual interpretation of the results. To do so I proceeded as follows.

1. Using Gimp / Photoshop

Initially use *GIMP* to identify the pre-process routine (bounding-box, optional rotation).

> Tipp: Overlay several images to cover different scenarios.

2. Test comandline

After identifying the pre-process routine try to put the parameters into *ImageMagick* and test command on a single image using `convert`.

```
# Test
# SizeX x SizeY + PostionX + PositionY
convert.string <- "-crop 1600x800+0+1030"
cmd <- paste("convert extra/Ref_raw.jpg", convert.string, "extra/Ref.jpg")
system(cmd)
message("Please check cropped Ref.jpg, then proceed")
```

This results in the following extend of interest. Only this part of the image will be further analysed, so please only proceed if satisfied with the result.

3. Preprocess image archive

Next all images will be pre-processed according to the routine developed above using `mogrify`. Please pay attention to the slightly different syntax of the `mogrify -crop ... -path IMG/ IMG_raw/*.jpg`. This will preprocess all images from `IMG_raw/` and save them in `IMG/`.

```
# Preprocess
dir.create(img.folder)
cmd <- paste("mogrify", convert.string, "-path", img.folder,
             paste0("\"",img.folder_raw, "*.jpg", "\""))
system(cmd)
message("Finished preprocessing")
```

List images

First all images need to be listed. The following chunk does so, plus enhances the data frame according to *Site configuration*: (1) due to external effects (eg. transmission) images can be corrupted. Here files with a file size smaller than the `threshold` will be exluded. (2) The Timestamp gets interpreted, therefore first the filenames are cropped with help of `gsub.Date`. Because filenames can be very different and the corresponding regular expression can very complex, it seemed easiest to do with a function. This also makes developing it more simple due better testing option. After cropping the timestamp it will be converted to a *POSIXlt* time object using `date.code`. (3) Last but not least the relative filepaths are reconstructed. Note, that this should also work with `list.files(..., fullnames=TRUE)` but I remeber then struggeling with grepping the datecode.

```
Files <- data.frame(Filename=list.files(img.folder, pattern = "*.jpg"),
                    stringsAsFactors = FALSE)

# Remove corrupted images
Files$Size <- file.size(paste0(img.folder, Files$Filename)) > threshold
Files <- Files[which(Files$Size),]
Files <- select(Files, -Size)
```

```r
# Add Timestamp
Files$Timestamp <- strptime(gsub.Date(Files$Filename), date.code)
Files$Timestamp <- as.POSIXct(Files$Timestamp)
Files <- Files[order(Files$Timestamp),]  # Order by Timestamp

# Full Filename
Files$Filename <- paste0(img.folder, Files$Filename)
```

To get an overview about the data beeing processed, here some metadata summarys are promted.

```
## 21 files to analize
```

```
## Dates from 24.06.2017 12:30 to 24.06.2017 12:50
```

```
## Time difference of 20 mins
```

Ground-Truth-Data

To latter assess the classifiers accuracies, Ground-Truth-Data is mandatory. Use `GTD_list()` to manually count pedestrians in `Files$Filename`. Here all images (100%) got evaled, for sampling uncomment orange lines.

```r
start <- Sys.time()  # Get start time
#GTD <- GTD_list(sample(x = Files$Filename, size = 10))
#the.sample <- sample(c(1:nrow(Files)), size = 100)
#Files <- Files[the.sample,]
Files$GTD <- GTD_list(Files$Filename)
Files$GTD <- as.numeric(Files$GTD)
(Sys.time() - start)  # Print runtime

save(Files, file = "Results/GTD.RData")
write.csv(Files, file = "Results/GTD.csv")
```

Sum of visitors in GTD.

To aggregate the time-series by `T_scale` use `fun_Aggregation()`. Use `tidyverse` gramar to select wished aggeration method (mean or sum).

```r
# Aggregation
Files_res <- fun_Aggregation(Files$Timestamp, Files$GTD, T_scale) %>%
  select(-MEAN) %>%
  rename(GTD = SUM)
```

Processing

Finally we can start processing the (preprocessed) image archive. For a detailed description of the methods, the reader is refered to the authors master thesis.

Method 1: Change detection

This approach is inspired by methods used remote sensing and biotech. Using algebra, two pictures are applied against each other, revealing changes. Use `CD_list()` for processing a list of images, including parallel processing. See `?CD_single()` for the available parameters.

```r
# Processing
start <- Sys.time()  # Get start time
CD <- CD_list(Files$Filename, Min = 0.9, method = "ratio",
                     predictions = "CD_Predictions")
(Sys.time() - start)  # Print runtime
```

```
## Time difference of 7.604359 secs
```

```r
Files$Hum <- CD
# Aggregation
Files_res <- fun_Aggregation(Files$Timestamp, Files$Hum, T_scale) %>%
  select(-SUM) %>%
  rename(CD = MEAN) %>%
  left_join(Files_res)
```

```
## Joining, by = "Timestamp"
```

Now convert the number of changed pixels into visitor numbers by calibrating them.

```r
# Calibration
lm_cal <- lm(GTD ~ 0+CD, data = Files_res)
summary(lm_cal)
```

```
##
## Call:
## lm(formula = GTD ~ 0 + CD, data = Files_res)
##
## Residuals:
##      1      2
## -3.155  3.442
##
## Coefficients:
##    Estimate Std. Error t value Pr(>|t|)
## CD  0.07309    0.03516   2.079    0.285
##
## Residual standard error: 4.669 on 1 degrees of freedom
## Multiple R-squared:  0.8121, Adjusted R-squared:  0.6242
## F-statistic: 4.322 on 1 and 1 DF,  p-value: 0.2854
```

```r
Files_res$CD_pred <- round(predict(lm_cal, select(Files_res, -GTD)))
```

Method 2: Histogramms of Oriented Gradients

The second method uses *Histogramms of Oriented Gradients* to detect pedestrians [DALAL_2005]. The HOG-Descriptor, focuses on a feature class of the same name. Here the *OpenCV*-Python implementation was wrapped into `hog_list()`. For an installation guide please see `?hog_install()`. First however upscale the images according to the trainingset.

```r
# Resize
dir.create("IMG_resize/")
```

```
## Warning in dir.create("IMG_resize/"): 'IMG_resize' existiert bereits
```

```r
cmd <- paste("mogrify -resize 270x240 -path IMG_resize/",
             paste0("IMG/", "*.jpg"))
system(cmd)
```

```r
message("Finished preprocessing")
```

```
## Finished preprocessing
```

```r
Files_resized <- gsub("IMG/", "IMG_resize/", Files$Filename)
```

```r
# Processing
start <- Sys.time()  # Get start time
HOG <- hog_list(Files_resized, resize = 1, padding = 24, winStride = 2,
                Mscale = 1.05, predictions = "HOG_Predictions/")
(Sys.time() - start)  # Print runtime
```

```
## Time difference of 1.489887 secs
```

To access the object-based accuracies, run `GTD_truePositives()`. It will `cor.test()` the input for you and returns the miss-rate, FPPW and so forth.

```r
Files$HOG <- HOG
GTD_truePositives(Files$GTD, Files$HOG)
```

```
##
##  Pearson's product-moment correlation
##
## data:  GTD and PRD
## t = 9.7321, df = 19, p-value = 8.136e-09
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.7936218 0.9643845
## sample estimates:
##       cor
## 0.9126413
```

```
##   FN TP FP        MR FPPW       cor
## 1  6  8  0 0.4285714    0 0.9126413
```

```r
# Aggregation
Files_res <- fun_Aggregation(Files$Timestamp, Files$HOG, T_scale) %>%
  select(-MEAN) %>%
  rename(HOG = SUM) %>%
  left_join(Files_res)
```

```
## Joining, by = "Timestamp"
```

Method 3: Convolutional-Neural-Network

The third and last object detector discussed in this study bases on Convolutional-Neural-Networks. Here *YOLO* [REDMON_2016] was utilized, as it comes pre-trained and has a open license. To install it use `yolo_install()`. Accordingly it was wrapped into `yolo_list()` and `yolo_single()`, whereby unfortunately only the latter is capable of saving the predictions. Using `sapply()` is possible, however note, that processing does take much longer, as the weights need to get loaded repeatedly. Attention, after every *wuepix* installation it is necessary to run `yolo_update()` as well, as this write a small file into the package installation linking to the *YOLO* installation. This links aids running the wrapper functions conveniently.

```r
# Processing
start <- Sys.time()  # Get start time
YOLO <- yolo_list(Files_resized)
```

```r
# YOLO <- sapply(Files_resized, yolo_single, predictions = "YOLO_Predictions")
(Sys.time() - start)  # Print runtime
```

```
## Time difference of 8.530758 mins
```

You can also use `GTD_truePositives()` here.

```r
Files$YOLO <- YOLO
GTD_truePositives(Files$GTD, Files$YOLO)
```

```
##
##  Pearson's product-moment correlation
##
## data:  GTD and PRD
## t = 58.938, df = 19, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.9931527 0.9989179
## sample estimates:
##       cor
## 0.9972763
```

```
##   FN TP FP         MR FPPW       cor
## 1  1 13  0 0.07142857    0 0.9972763
```

```r
# Aggregation
Files_res <- fun_Aggregation(Files$Timestamp, Files$YOLO, T_scale) %>%
  select(-MEAN) %>%
  rename(YOLO = SUM) %>%
  left_join(Files_res)
```

```
## Joining, by = "Timestamp"
```

As YOLO detects a lot of objects, `yolo_list()` logs a complete list of all detections to a separate file `yolo_detections.txt`. To read it in a tidy way use `yolo_Read()`.

```r
# Read, group and count detections
yolo.results <- yolo_Read("yolo_detections.txt")
unique(yolo.results$Class)
```

```
## [1] "backpack"    "bicycle"     "person"      "umbrella"    "sports ball"
```

Save Results

Last but not least, save the *R* environment for further studies.

```r
save.image(file = "Results/Enviroment.RData")
```