This blog post illustrates the new functionality of the OpenImageR package (*Gabor Feature Extraction*). The Gabor features have been used extensively in [image analysis and processing](https://en.wikipedia.org/wiki/Gabor_filter) (Character and Face recognition). [Gabor](https://en.wikipedia.org/wiki/Dennis_Gabor) (Nobel prize winner, an electrical engineer, and physicist) used the following wording which I think it’s worth mentioning in this blog post, *“You can’t predict the future, but you can invent it.”* ([source](http://www.it.lut.fi/project/gabor/)).

In the following lines I’ll describe the *GaborFeatureExtract* R6 class which includes the following methods,

| **GaborFeatureExtract** |
| --- |
| gabor\_filter\_bank() |
| gabor\_feature\_extraction() |
| gabor\_feature\_engine() |
| plot\_gabor() |

These methods are based on the Matlab *Gabor Feature Extraction* of the paper *M. Haghighat, S. Zonouz, M. Abdel-Mottaleb, “CloudID: Trustworthy cloud-based and cross-enterprise biometric identification,” Expert Systems with Applications, vol. 42, no. 21, pp. 7905-7916, 2015*, http://dx.doi.org/10.1016/j.eswa.2015.06.025. The initial Matlab code was modified (I added the **Magnitude** feature and the **gabor\_feature\_engine()** method) and parallelized using Rcpp wherever it was possible.

**Gabor Features**

I came across the Gabor Features last month when I had to process images and I needed an additional function besides the already existing [HoG features](https://www.rdocumentation.org/packages/OpenImageR/versions/1.0.8/topics/HOG?tap_a=5644-dce66f&tap_s=10907-287229). I did the regular search on CRAN (Comprehensive R Archive Network) but I couldn’t find anything related to Gabor Feature Extraction (as of August 2018), therefore I decided to port the Matlab code into R. There are many resources on the web if someone intends to deepen his / her knowledge on the subject and I’ll add some of these that I found useful at the end of the blog post (References). I’ll explain the methods of the *GaborFeatureExtract* R6 class using an image.

GaborFeatures.M

|  |
| --- |
| function featureVector = gaborFeatures(img,gaborArray,d1,d2) |
|  |  |
|  | % GABORFEATURES extracts the Gabor features of an input image. |
|  | % It creates a column vector, consisting of the Gabor features of the input |
|  | % image. The feature vectors are normalized to zero mean and unit variance. |
|  | % |
|  | % |
|  | % Inputs: |
|  | % img : Matrix of the input image |
|  | % gaborArray : Gabor filters bank created by the function gaborFilterBank |
|  | % d1 : The factor of downsampling along rows. |
|  | % d2 : The factor of downsampling along columns. |
|  | % |
|  | % Output: |
|  | % featureVector : A column vector with length (m\*n\*u\*v)/(d1\*d2). |
|  | % This vector is the Gabor feature vector of an |
|  | % m by n image. u is the number of scales and |
|  | % v is the number of orientations in 'gaborArray'. |
|  | % |
|  | % |
|  | % Sample use: |
|  | % |
|  | % img = imread('cameraman.tif'); |
|  | % gaborArray = gaborFilterBank(5,8,39,39); % Generates the Gabor filter bank |
|  | % featureVector = gaborFeatures(img,gaborArray,4,4); % Extracts Gabor feature vector, 'featureVector', from the image, 'img'. |
|  | % |
|  | % |
|  | % |
|  | % Details can be found in: |
|  | % |
|  | % M. Haghighat, S. Zonouz, M. Abdel-Mottaleb, "CloudID: Trustworthy |
|  | % cloud-based and cross-enterprise biometric identification," |
|  | % Expert Systems with Applications, vol. 42, no. 21, pp. 7905-7916, 2015. |
|  | % |
|  | % |
|  | % |
|  | % (C) Mohammad Haghighat, University of Miami |
|  | % haghighat@ieee.org |
|  | % PLEASE CITE THE ABOVE PAPER IF YOU USE THIS CODE. |
|  |  |
|  |  |
|  | if (nargin ~= 4) % Check correct number of arguments |
|  | error('Please use the correct number of input arguments!') |
|  | end |
|  |  |
|  | if size(img,3) == 3 % Check if the input image is grayscale |
|  | warning('The input RGB image is converted to grayscale!') |
|  | img = rgb2gray(img); |
|  | end |
|  |  |
|  | img = double(img); |
|  |  |
|  |  |
|  | %% Filter the image using the Gabor filter bank |
|  |  |
|  | % Filter input image by each Gabor filter |
|  | [u,v] = size(gaborArray); |
|  | gaborResult = cell(u,v); |
|  | for i = 1:u |
|  | for j = 1:v |
|  | gaborResult{i,j} = imfilter(img, gaborArray{i,j}); |
|  | end |
|  | end |
|  |  |
|  |  |
|  | %% Create feature vector |
|  |  |
|  | % Extract feature vector from input image |
|  | featureVector = []; |
|  | for i = 1:u |
|  | for j = 1:v |
|  |  |
|  | gaborAbs = abs(gaborResult{i,j}); |
|  | gaborAbs = downsample(gaborAbs,d1); |
|  | gaborAbs = downsample(gaborAbs.',d2); |
|  | gaborAbs = gaborAbs(:); |
|  |  |
|  | % Normalized to zero mean and unit variance. (if not applicable, please comment this line) |
|  | gaborAbs = (gaborAbs-mean(gaborAbs))/std(gaborAbs,1); |
|  |  |
|  | featureVector = [featureVector; gaborAbs]; |
|  |  |
|  | end |
|  | end |
|  |  |
|  |  |
|  | %% Show filtered images (Please comment this section if not needed!) |
|  |  |
|  | % % Show real parts of Gabor-filtered images |
|  | % figure('NumberTitle','Off','Name','Real parts of Gabor filters'); |
|  | % for i = 1:u |
|  | % for j = 1:v |
|  | % subplot(u,v,(i-1)\*v+j) |
|  | % imshow(real(gaborResult{i,j}),[]); |
|  | % end |
|  | % end |
|  | % |
|  | % % Show magnitudes of Gabor-filtered images |
|  | % figure('NumberTitle','Off','Name','Magnitudes of Gabor filters'); |
|  | % for i = 1:u |
|  | % for j = 1:v |
|  | % subplot(u,v,(i-1)\*v+j) |
|  | % imshow(abs(gaborResult{i,j}),[]); |
|  | % end |
|  | % end |

GaborFilterBank.m

|  |
| --- |
| function gaborArray = gaborFilterBank(u,v,m,n) |
|  |  |
|  | % GABORFILTERBANK generates a custum Gabor filter bank. |
|  | % It creates a u by v cell array, whose elements are m by n matrices; |
|  | % each matrix being a 2-D Gabor filter. |
|  | % |
|  | % |
|  | % Inputs: |
|  | % u : No. of scales (usually set to 5) |
|  | % v : No. of orientations (usually set to 8) |
|  | % m : No. of rows in a 2-D Gabor filter (an odd integer number, usually set to 39) |
|  | % n : No. of columns in a 2-D Gabor filter (an odd integer number, usually set to 39) |
|  | % |
|  | % Output: |
|  | % gaborArray: A u by v array, element of which are m by n |
|  | % matries; each matrix being a 2-D Gabor filter |
|  | % |
|  | % |
|  | % Sample use: |
|  | % |
|  | % gaborArray = gaborFilterBank(5,8,39,39); |
|  | % |
|  | % |
|  | % |
|  | % Details can be found in: |
|  | % |
|  | % M. Haghighat, S. Zonouz, M. Abdel-Mottaleb, "CloudID: Trustworthy |
|  | % cloud-based and cross-enterprise biometric identification," |
|  | % Expert Systems with Applications, vol. 42, no. 21, pp. 7905-7916, 2015. |
|  | % |
|  | % |
|  | % |
|  | % (C) Mohammad Haghighat, University of Miami |
|  | % haghighat@ieee.org |
|  | % PLEASE CITE THE ABOVE PAPER IF YOU USE THIS CODE. |
|  |  |
|  |  |
|  |  |
|  | if (nargin ~= 4) % Check correct number of arguments |
|  | error('There must be four input arguments (Number of scales and orientations and the 2-D size of the filter)!') |
|  | end |
|  |  |
|  |  |
|  | %% Create Gabor filters |
|  | % Create u\*v gabor filters each being an m by n matrix |
|  |  |
|  | gaborArray = cell(u,v); |
|  | fmax = 0.25; |
|  | gama = sqrt(2); |
|  | eta = sqrt(2); |
|  |  |
|  | for i = 1:u |
|  |  |
|  | fu = fmax/((sqrt(2))^(i-1)); |
|  | alpha = fu/gama; |
|  | beta = fu/eta; |
|  |  |
|  | for j = 1:v |
|  | tetav = ((j-1)/v)\*pi; |
|  | gFilter = zeros(m,n); |
|  |  |
|  | for x = 1:m |
|  | for y = 1:n |
|  | xprime = (x-((m+1)/2))\*cos(tetav)+(y-((n+1)/2))\*sin(tetav); |
|  | yprime = -(x-((m+1)/2))\*sin(tetav)+(y-((n+1)/2))\*cos(tetav); |
|  | gFilter(x,y) = (fu^2/(pi\*gama\*eta))\*exp(-((alpha^2)\*(xprime^2)+(beta^2)\*(yprime^2)))\*exp(1i\*2\*pi\*fu\*xprime); |
|  | end |
|  | end |
|  | gaborArray{i,j} = gFilter; |
|  |  |
|  | end |
|  | end |
|  |  |
|  |  |
|  | %% Show Gabor filters (Please comment this section if not needed!) |
|  |  |
|  | % Show magnitudes of Gabor filters: |
|  | figure('NumberTitle','Off','Name','Magnitudes of Gabor filters'); |
|  | for i = 1:u |
|  | for j = 1:v |
|  | subplot(u,v,(i-1)\*v+j); |
|  | imshow(abs(gaborArray{i,j}),[]); |
|  | end |
|  | end |
|  |  |
|  | % Show real parts of Gabor filters: |
|  | figure('NumberTitle','Off','Name','Real parts of Gabor filters'); |
|  | for i = 1:u |
|  | for j = 1:v |
|  | subplot(u,v,(i-1)\*v+j); |
|  | imshow(real(gaborArray{i,j}),[]); |
|  | end |
|  | end |

**gabor\_filter\_bank**

The **gabor\_filter\_bank** method *“… generates a custom-sized Gabor filter bank. It creates a UxV cell array, whose elements are MxN matrices; each matrix being a 2-D Gabor filter”*. The following code chunk shows how this works in R,

library(OpenImageR)

init\_gb = GaborFeatureExtract$new()

#------------------

# gabor-filter-bank

#------------------

gb\_f = init\_gb$gabor\_filter\_bank(scales = 5, orientations = 8, gabor\_rows = 39,

gabor\_columns = 39, plot\_data = TRUE)

#-----------------------------------------------

# plot of the real part of the gabor-filter-bank

#-----------------------------------------------

plt\_f = init\_gb$plot\_gabor(real\_matrices = gb\_f$gabor\_real, margin\_btw\_plots = 0.65,

thresholding = FALSE)

For the *gabor\_filter\_bank* I use 5 *scales* and 8 *orientations* to build filters of size 39 x 39. The output of the method is a list of length 3,

str(gb\_f)

List of 3

$ gaborArray :List of 40

..$ : cplx [1:39, 1:39] -1.58e-12-0.00i 0.00-5.02e-12i 1.50e-11-0.00i ...

..$ : cplx [1:39, 1:39] 4.86e-08-3.96e-08i 1.02e-07+4.63e-08i 6.31e-09+1.93e-07i ...

..$ : cplx [1:39, 1:39] 6.24e-06-6.24e-06i 1.18e-05+0.00i 1.10e-05+1.10e-05i ...

..$ : cplx [1:39, 1:39] -6.69e-05-3.18e-05i -4.63e-05-7.20e-05i -1.60e-06-9.81e-05i ...

..$ : cplx [1:39, 1:39] 1.40e-04+5.81e-05i 1.15e-04+1.15e-04i 6.68e-05+1.61e-04i ...

.......

$ gabor\_imaginary:List of 40

..$ : num [1:39, 1:39] -4.65e-27 -5.02e-12 -1.10e-26 4.21e-11 -2.99e-25 ...

..$ : num [1:39, 1:39] -3.96e-08 4.63e-08 1.93e-07 1.53e-07 -3.04e-07 ...

..$ : num [1:39, 1:39] -6.24e-06 4.84e-20 1.10e-05 2.01e-05 1.81e-05 ...

..$ : num [1:39, 1:39] -3.18e-05 -7.20e-05 -9.81e-05 -9.58e-05 -5.78e-05 ...

..$ : num [1:39, 1:39] 5.81e-05 1.15e-04 1.61e-04 1.86e-04 1.83e-04 ...

.......

$ gabor\_real :List of 40

..$ : num [1:39, 1:39] -1.58e-12 5.54e-27 1.50e-11 -4.12e-26 -1.11e-10 ...

..$ : num [1:39, 1:39] 4.86e-08 1.02e-07 6.31e-09 -2.85e-07 -4.28e-07 ...

..$ : num [1:39, 1:39] 6.24e-06 1.18e-05 1.10e-05 -8.11e-20 -1.81e-05 ...

..$ : num [1:39, 1:39] -6.69e-05 -4.63e-05 -1.60e-06 5.73e-05 1.12e-04 ...

..$ : num [1:39, 1:39] 1.40e-04 1.15e-04 6.68e-05 -3.77e-19 -7.57e-05 ...

.......

The first sublist (*gaborArray*) consists of 40 matrices (5 scales x 8 orientations) of type complex, where each matix is of dimension 39 x 39 (gabor filter). The second sublist (*gabor\_imaginary*) is the imaginary part (numeric) whereas the third sublist is the real part (*gabor\_real*). The real part (numeric) is used to plot the gabor filters.

The documentation includes more details for each of the parameters used.

**gabor\_feature\_extraction**

The **gabor\_feature\_extraction** method extracts the Gabor Features of the image. This method is modified in comparison to the initial Matlab code to give users the option to downsample the image or to normalize the features. Moreover, I added the **Magnitude** feature because according to literature it improves predictability.

Based on the previously mentioned *car.png* image,

# read image

#-----------

pth\_im = system.file("tmp\_images", "car.png", package = "OpenImageR")

im = readImage(pth\_im) \* 255

# gabor-feature-extract

#----------------------

gb\_im = init\_gb$gabor\_feature\_extraction(image = im, scales = 5, orientations = 8,

downsample\_gabor = FALSE, downsample\_rows = NULL,

downsample\_cols = NULL, gabor\_rows = 39,

gabor\_columns = 39, plot\_data = TRUE,

normalize\_features = FALSE, threads = 3)

This function again returns a list of length 3,

str(gb\_im)

List of 3

$ gaborFeatures :List of 2

..$ magnitude : num [1, 1:206670] 0 0 0 0 0 0 0 0 0 0 ...

..$ energy\_aptitude: num [1, 1:80] 115823 27566 33289 20074 26918 ...

$ gabor\_features\_imaginary:List of 5

..$ : num [1:166, 1:249, 1:8] 0 0 0 0 0 0 0 0 0 0 ...

..$ : num [1:166, 1:249, 1:8] 0 0 0 0 0 0 0 0 0 0 ...

..$ : num [1:166, 1:249, 1:8] 0 0 0 0 0 0 0 0 0 0 ...

..$ : num [1:166, 1:249, 1:8] 0 0 0 0 0 0 0 0 0 0 ...

..$ : num [1:166, 1:249, 1:8] 0 0 0 0 0 0 0 0 0 0 ...

$ gabor\_features\_real :List of 5

..$ : num [1:166, 1:249, 1:8] 0 0 0 0 0 0 0 0 0 0 ...

..$ : num [1:166, 1:249, 1:8] 0 0 0 0 0 0 0 0 0 0 ...

..$ : num [1:166, 1:249, 1:8] 0 0 0 0 0 0 0 0 0 0 ...

..$ : num [1:166, 1:249, 1:8] 0 0 0 0 0 0 0 0 0 0 ...

..$ : num [1:166, 1:249, 1:8] 0 0 0 0 0 0 0 0 0 0 ...

where *gaborFeatures* are the extracted features with number of rows equal to nrow(im) x ncol(im) (or 166 x 249) and number of columns equal to scales x orientations (or 5 x 8). The second and third sublists are the imaginary and real part of the image resulted after the convolution with the gabor filters. The following code chunk allows the user to plot the different *scales* and *orientations* of the image,

plt\_im = init\_gb$plot\_gabor(real\_matrices = gb\_im$gabor\_features\_real,

margin\_btw\_plots = 0.65, thresholding = FALSE)

By thresholding the *gb\_im$gabor\_features\_real* object (scales, orientations) to [0,1] the images can be visually explored,

# thresholding parameter is set to TRUE

#--------------------------------------

plt\_im\_thresh = init\_gb$plot\_gabor(real\_matrices = gb\_im$gabor\_features\_real,

margin\_btw\_plots = 0.65, thresholding = TRUE)

**gabor\_feature\_engine**

The *gabor\_feature\_engine* method is an extension of the initial Matlab code and allows the user to extract gabor features from multiple images. This method works in the same way as the *HOG\_apply* method, which takes a matrix of images – such as the mnist data set – and after processing it returns the features. The following example illustrates how to use the *gabor\_feature\_engine* method with the mnist data set,

# for instance downloading of the mnist data on a linux OS

#----------------------------------------------------------

system("wget https://raw.githubusercontent.com/mlampros/DataSets/master/mnist.zip")

# or just navigate to "https://github.com/mlampros/DataSets/blob/master/mnist.zip" and click the download button

#-------------------------------------------------------------------------------------------

mnist <- read.table(unz("mnist.zip", "mnist.csv"), nrows = 70000, header = T,

quote = "\"", sep = ",")

mnist = as.matrix(mnist)

x = mnist[, -ncol(mnist)]

y = mnist[, ncol(mnist)] + 1

dat = init\_gb$gabor\_feature\_engine(img\_data = x, img\_nrow = 28, img\_ncol = 28,

scales = 4, orientations = 8, gabor\_rows = 13,

gabor\_columns = 13, downsample\_gabor = FALSE,

downsample\_rows = NULL, downsample\_cols = NULL,

normalize\_features = FALSE, threads = 6,

verbose = TRUE)

Time to complete : 4.111672 mins

> str(dat)

List of 2

$ magnitude : num [1:70000, 1:3136] 0 0 0 0 0 0 0 0 0 0 ...

$ energy\_aptitude: num [1:70000, 1:64] 2682 2576 1399 1178 2240 ...

The *dat* object is a list of length 2. The first sublist corresponds to *magnitude* whereas the second sublist to *local-energy* and *mean-aptitude*. The first thing to do before calculating the accuracy for the mnist data is to reduce the dimensionality of the *magnitude* feature (I’ll use the irlba package for this purpose),

Library(irlba)

svd\_irlb = irlba::irlba(as.matrix(dat$magnitude), nv = 100, nu = 100, verbose = T)

new\_x = as.matrix(dat$magnitude) %\*% svd\_irlb$v

and I’ll create a centered-scaled matrix of the reduced *magnitude* and the *energy-aptitude* data,

dat = ClusterR::center\_scale(cbind(new\_x, dat$energy\_aptitude))

dim(dat)

[1] 70000 164

then I’ll utilize the *nmslibR* library (approximate method ‘hnsw’) to calculate the accuracy for the mnist data,

M = 30

efC = 100

num\_threads = 6

index\_params = list('M'= M, 'indexThreadQty' = num\_threads, 'efConstruction' = efC,

'post' = 0, 'skip\_optimized\_index' = 1 )

efS = 100

query\_time\_params = list('efSearch' = efS)

fit\_gab = nmslibR::KernelKnnCV\_nmslib(dat, y, k = 20, folds = 4, h = 1,

weights\_function = 'biweight\_triangular\_tricube\_MULT',

Levels = sort(unique(y)), Index\_Params = index\_params,

Time\_Params = query\_time\_params, space = "cosinesimil",

space\_params = NULL, method = "hnsw",

data\_type = "DENSE\_VECTOR", dtype = "FLOAT",

index\_filepath = NULL, print\_progress = T,

num\_threads = 6, seed\_num = 1)

time to complete : 30.82252 secs

acc\_fit\_gab = unlist(lapply(1:length(fit\_gab$preds),

function(x) {

mean(y[fit\_gab$folds[[x]]] == max.col(

fit\_gab$preds[[x]], ties.method = 'random')) }))

mean(acc\_fit\_gab)

[1] 0.9752714

I’ll do the same using the *HOG\_apply* function,

# hog-features for the mnist data

#--------------------------------

hog = OpenImageR::HOG\_apply(x, cells = 6, orientations = 9, rows = 28, columns = 28, threads = 6)

fit\_hog = nmslibR::KernelKnnCV\_nmslib(hog, y, k = 20, folds = 4, h = 1,

weights\_function = 'biweight\_triangular\_tricube\_MULT',

Levels = sort(unique(y)), Index\_Params = index\_params,

Time\_Params = query\_time\_params, space = "cosinesimil",

space\_params = NULL, method = "hnsw",

data\_type = "DENSE\_VECTOR", dtype = "FLOAT",

index\_filepath = NULL, print\_progress = T,

num\_threads = 6, seed\_num = 1)

time to complete : 41.00309 secs

acc\_fit\_hog = unlist(lapply(1:length(fit\_hog$preds),

function(x) {

mean(y[fit\_hog$folds[[x]]] == max.col(

fit\_hog$preds[[x]], ties.method = 'random')) }))

mean(acc\_fit\_hog)

[1] 0.9787429

By averaging both the *gabor* and the *HoG* features the mean accuracy increases to 98.34 % which is very close to the score of the HoG + brute force method of KernelKnn (98.4),

acc\_fit\_gab\_hog = unlist(lapply(1:length(fit\_gab$preds),

function(x) {

mean(y[fit\_gab$folds[[x]]] == max.col(

(fit\_gab$preds[[x]] + fit\_hog$preds[[x]]) / 2,

ties.method = 'random')) }))

mean(acc\_fit\_gab\_hog)

[1] 0.9833857

Kernel k nearest neighbors

k nearest neighbors

In pattern recognition the k nearest neighbors (KNN) is a non-parametric method used for classification and regression. Although KNN belongs to the 10 most influential algorithms in data mining, it is considered as one of the simplest in machine learning.  
  
The most important parameters of the KNN algorithm are **k** and the **distance metric**. The parameter k specifies the number of neighbor observations that contribute to the output predictions. Optimal values for k can be obtained mainly through resampling methods, such as *cross-validation* or *bootstrap*. The distance metric is another important factor, which depending on the data set can affect the performance of the algorithm. Widely used distance metrics are the *euclidean*, *manhattan*, *chebyshev*, *minkowski* and *hamming*.  
  
The simple KNN algorithm can be extended by giving different weights to the selected k nearest neighbors. A common weighting scheme consists in giving each neighbor a weight of 1/d, where d is the distance to the neighbor. The purpose of the *KernelKnn* package is to use different weight functions (kernels) in order to optimize the output predictions in both regression and classification.

KernelKnn function

The following code snippets appear in the package Vignettes. I’ll illustrate the package functionality using mainly classification and in-between I’ll point out the differences for regression and classification, which could lead to potential errors.

data(ionosphere, package = 'KernelKnn')

apply(ionosphere, 2, function(x) length(unique(x)))

## V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12

## 2 1 219 269 204 259 231 260 244 267 246 269

## V13 V14 V15 V16 V17 V18 V19 V20 V21 V22 V23 V24

## 238 266 234 270 254 280 254 266 248 265 248 264

## V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 class

## 256 273 256 281 244 266 243 263 245 263 2

# the second column will be removed as it has a single unique value

ionosphere = ionosphere[, -2]

When using an algorithm where the output depends on distance calculation (as is the case in k-nearest-neighbors) it is recommended to first scale the data,

X = scale(ionosphere[, -ncol(ionosphere)])

y = ionosphere[, ncol(ionosphere)]

Both in regression and classification the **KernelKnn** function accepts a numeric vector as a response variable (here y). In **classification** the user should additionally give the unique values of the labels, which should range from 1:Inf. This is important otherwise the internal functions do not work. The *KernelKnn* function returns a vector of numeric values in case of regression or a matrix of class probabilities in case of classification.

# convert y from factor to numeric in classification

y = c(1:length(unique(y)))[ match(ionosphere$class, sort(unique(ionosphere$class))) ]

# random split in train-test

spl\_train = sample(1:length(y), round(length(y) \* 0.75))

spl\_test = setdiff(1:length(y), spl\_train)

str(spl\_train)

## int [1:263] 56 10 224 249 109 223 221 146 93 194

str(spl\_test)

## int [1:88] 2 4 7 9 11 15 20 23 33 34 ...

# evaluation metric

acc = function (y\_true, preds) {

out = table(y\_true, max.col(preds, ties.method = "random"))

acc = sum(diag(out))/sum(out)

acc

}

A simple k-nearest-neighbors model can be run with weights\_function = NULL and the parameter ‘regression’ should be set to FALSE in case of classification.

library(KernelKnn)

preds\_TEST = KernelKnn(X[spl\_train, ], TEST\_data = X[spl\_test, ], y[spl\_train], k = 5 ,

method = 'euclidean', weights\_function = NULL, regression = F,

Levels = unique(y))

head(preds\_TEST)

## class\_1 class\_2

## [1,] 0 1

## [2,] 0 1

## [3,] 0 1

## [4,] 0 1

## [5,] 0 1

## [6,] 0 1

K-nearest-neigbor calculations in the KernelKnn function can be accomplished using the following distance metrics : *euclidean*, *manhattan*, *chebyshev*, *canberra*, *braycurtis*, *minkowski* (by default the order ‘p’ of the minkowski parameter equals k), *hamming*, *mahalanobis*, *pearson\_correlation*, *simple\_matching\_coefficient*, *jaccard\_coefficient* and *Rao\_coefficient*. The last four are similarity measures and are appropriate for binary data [0,1].

There are two ways to use a kernel in the KernelKnn function. The **first option** is to choose one of the existing kernels (*uniform*, *triangular*, *epanechnikov*, *biweight*, *triweight*, *tricube*, *gaussian*, *cosine*, *logistic*, *silverman*, *inverse*, *gaussianSimple*, *exponential*). Here, I’ll use the canberra metric and the tricube kernel because they give optimal results (according to my RandomSearchR package),

preds\_TEST\_tric = KernelKnn(X[spl\_train, ], TEST\_data = X[spl\_test, ], y[spl\_train], k = 10 ,

method = 'canberra', weights\_function = 'tricube', regression = F,

Levels = unique(y))

head(preds\_TEST\_tric)

## [,1] [,2]

## [1,] 1.745564e-02 0.98254436

## [2,] 9.667304e-01 0.03326963

## [3,] 0.000000e+00 1.00000000

## [4,] 6.335040e-18 1.00000000

## [5,] 4.698239e-02 0.95301761

## [6,] 0.000000e+00 1.00000000

The **second option** is to give a self-defined kernel function. Here, I’ll pick the density function of the normal distribution with mean = 0.0 and standard deviation = 1.0 (the data are scaled to have mean zero and unit variance),

norm\_kernel = function(W) {

W = dnorm(W, mean = 0, sd = 1.0)

W = W / rowSums(W)

return(W)

}

preds\_TEST\_norm = KernelKnn(X[spl\_train, ], TEST\_data = X[spl\_test, ], y[spl\_train], k = 10 ,

method = 'canberra', weights\_function = norm\_kernel, regression = F,

Levels = unique(y))

head(preds\_TEST\_norm)

## [,1] [,2]

## [1,] 0.26150003 0.7385000

## [2,] 0.84170089 0.1582991

## [3,] 0.00000000 1.0000000

## [4,] 0.07614579 0.9238542

## [5,] 0.09479386 0.9052061

## [6,] 0.00000000 1.0000000

The computations can be speed up by using the parameter **threads** (utilizes openMP). There is also the option to exclude **extrema** (minimum and maximum distances) during the calculation of the k-nearest-neighbor distances using extrema = TRUE. The bandwidth of the existing kernels can be tuned using the **h** parameter, which defaults to 1.0.

The KernelKnnCV function

The *KernelKnnCV* function can be employed to return the prediction accuracy using n-fold cross-validation. The following parameter pairs give optimal results according to my RandomSearchR package,

| **k** | **method** | **kernel** |
| --- | --- | --- |
| 10 | canberra | tricube |
| 9 | canberra | epanechnikov |

fit\_cv\_pair1 = KernelKnnCV(X, y, k = 10 , folds = 5, method = 'canberra',

weights\_function = 'tricube', regression = F,

Levels = unique(y), threads = 5)

str(fit\_cv\_pair1)

## List of 2

## $ preds:List of 5

## ..$ : num [1:71, 1:2] 0.00648 0.25323 1 0.97341 0.92031 ...

## ..$ : num [1:70, 1:2] 0 0 0 0 0.999 ...

## ..$ : num [1:70, 1:2] 0.353 0 0.17 0.212 0.266 ...

## ..$ : num [1:70, 1:2] 0 0 0 0 0 ...

## ..$ : num [1:70, 1:2] 0.989 0 1 0 0 ...

## $ folds:List of 5

## ..$ fold\_1: int [1:71] 5 26 233 243 30 41 237 229 19 11 ...

## ..$ fold\_2: int [1:70] 262 89 257 67 58 266 253 85 275 268 ...

## ..$ fold\_3: int [1:70] 127 128 295 287 134 288 130 277 125 101 ...

## ..$ fold\_4: int [1:70] 313 301 317 318 316 142 175 157 146 147 ...

## ..$ fold\_5: int [1:70] 195 326 225 332 342 347 206 219 218 214 ...

fit\_cv\_pair2 = KernelKnnCV(X, y, k = 9 , folds = 5,method = 'canberra',

weights\_function = 'epanechnikov', regression = F,

Levels = unique(y), threads = 5)

str(fit\_cv\_pair2)

## List of 2

## $ preds:List of 5

## ..$ : num [1:71, 1:2] 0.0224 0.255 1 0.9601 0.8876 ...

## ..$ : num [1:70, 1:2] 0 0 0 0 0.998 ...

## ..$ : num [1:70, 1:2] 0.36 0 0.164 0.185 0.202 ...

## ..$ : num [1:70, 1:2] 0 0 0 0 0 ...

## ..$ : num [1:70, 1:2] 0.912 0 1 0 0 ...

## $ folds:List of 5

## ..$ fold\_1: int [1:71] 5 26 233 243 30 41 237 229 19 11 ...

## ..$ fold\_2: int [1:70] 262 89 257 67 58 266 253 85 275 268 ...

## ..$ fold\_3: int [1:70] 127 128 295 287 134 288 130 277 125 101 ...

## ..$ fold\_4: int [1:70] 313 301 317 318 316 142 175 157 146 147 ...

## ..$ fold\_5: int [1:70] 195 326 225 332 342 347 206 219 218 214 ...

Each cross-validated object returns a list of length 2 ( the first sublist includes the predictions for each fold whereas the second gives the indices of the folds)

acc\_pair1 = unlist(lapply(1:length(fit\_cv\_pair1$preds),

function(x) acc(y[fit\_cv\_pair1$folds[[x]]],

fit\_cv\_pair1$preds[[x]])))

acc\_pair1

## [1] 0.9154930 0.9142857 0.9142857 0.9285714 0.9571429

cat('accurcay for params\_pair1 is :', mean(acc\_pair1), '\n')

## accurcay for params\_pair1 is : 0.9259557

acc\_pair2 = unlist(lapply(1:length(fit\_cv\_pair2$preds),

function(x) acc(y[fit\_cv\_pair2$folds[[x]]],

fit\_cv\_pair2$preds[[x]])))

acc\_pair2

## [1] 0.9014085 0.9142857 0.9000000 0.9142857 0.9571429

cat('accuracy for params\_pair2 is :', mean(acc\_pair2), '\n')

## accuracy for params\_pair2 is : 0.9174245

Adding or multiplying kernels

In the KernelKnn package the user can also combine kernels by adding or multiplying from the existing ones. For instance, if I want to multiply the tricube with the gaussian kernel, then I’ll give the following character string to the weights\_function, *“tricube\_gaussian\_MULT”*. On the other hand, If I want to add the same kernels then the weights\_function will be *“tricube\_gaussian\_ADD”*.

I’ll illustrate this option of the package using two image data sets in form of matrices, i.e. the *MNIST* and the *CIFAR-10*. From within R one can download the data in a linux OS using,

system("wget https://raw.githubusercontent.com/mlampros/DataSets/master/mnist.zip")

and

system("wget https://raw.githubusercontent.com/mlampros/DataSets/master/cifar\_10.zip")

Moreover, the **irlba** and the **OpenImageR** packages are needed for comparison purposes, which can be installed directly from CRAN using the install.packages() function. A 4-fold cross-validation using the KernelKnnCV function can take approximately 26 minutes (depending on the system configuration) utilizing 6 threads (for each data set). An alternative to reduce the computation time would be a train-test split of the data at the cost of performance validation.

**MNIST data set**

The MNIST data set of handwritten digits has a training set of 70,000 examples and each row of the matrix corresponds to a 28 x 28 image. The unique values of the response variable *y* range from 0 to 9. More information about the data can be found in the *DataSets* repository (the folder includes also an Rmarkdown file).

system("wget https://raw.githubusercontent.com/mlampros/DataSets/master/mnist.zip")

mnist <- read.table(unz("mnist.zip", "mnist.csv"), nrows = 70000, header = T,

quote = "\"", sep = ",")

X = mnist[, -ncol(mnist)]

dim(X)

## [1] 70000 784

# the KernelKnn function requires that the labels are numeric and start from 1 : Inf

y = mnist[, ncol(mnist)] + 1

table(y)

## y

## 1 2 3 4 5 6 7 8 9 10

## 6903 7877 6990 7141 6824 6313 6876 7293 6825 6958

K nearest neighbors do not perform well in high dimensions due to the *curse of dimensionality* (k observations that are nearest to a given test observation x1 may be very far away from x1 in p-dimensional space when p is large [ An introduction to statistical learning, James/Witten/Hastie/Tibshirani, pages 108-109 ]), leading to a very poor k-nearest-neighbors fit. One option to overcome this problem would be to use truncated svd (irlba package) to reduce the dimensions of the data. A second option, which is appropriate in case of images, would be to use image descriptors. Here, I’ll compare those two approaches.

**KernelKnnCV using truncated svd**

I experimented with different settings and the following parameters gave the best results,

| **irlba\_singlular\_vectors** | **k** | **method** | **kernel** |
| --- | --- | --- | --- |
| 40 | 8 | braycurtis | biweight\_tricube\_MULT |

library(irlba)

svd\_irlb = irlba(as.matrix(X), nv = 40, nu = 40, verbose = F) # irlba truncated svd

new\_x = as.matrix(X) %\*% svd\_irlb$v # new\_x using the 40 right singular vectors

fit = KernelKnnCV(as.matrix(new\_x), y, k = 8, folds = 4, method = 'braycurtis',

weights\_function = 'biweight\_tricube\_MULT', regression = F,

threads = 6, Levels = sort(unique(y)))

# str(fit)

acc\_fit = unlist(lapply(1:length(fit$preds),

function(x) acc(y[fit$folds[[x]]],

fit$preds[[x]])))

acc\_fit

## [1] 0.9742857 0.9749143 0.9761143 0.9741143

cat('mean accuracy using cross-validation :', mean(acc\_fit), '\n')

## mean accuracy using cross-validation : 0.9748571

Utilizing truncated svd a 4-fold cross-validation KernelKnn model gives a 97.48% accuracy.

**KernelKnnCV and HOG (histogram of oriented gradients)**

In this chunk of code, besides KernelKnnCV I’ll also use HOG. The histogram of oriented gradients (HOG) is a feature descriptor used in computer vision and image processing for the purpose of object detection. The technique counts occurrences of gradient orientation in localized portions of an image. This method is similar to that of edge orientation histograms, scale-invariant feature transform descriptors, and shape contexts, but differs in that it is computed on a dense grid of uniformly spaced cells and uses overlapping local contrast normalization for improved accuracy (Wikipedia).

library(OpenImageR)

hog = HOG\_apply(X, cells = 6, orientations = 9, rows = 28, columns = 28, threads = 6)

##

## time to complete : 1.802997 secs

dim(hog)

## [1] 70000 324

fit\_hog = KernelKnnCV(hog, y, k = 20, folds = 4, method = 'braycurtis',

weights\_function = 'biweight\_tricube\_MULT', regression = F,

threads = 6, Levels = sort(unique(y)))

#str(fit\_hog)

acc\_fit\_hog = unlist(lapply(1:length(fit\_hog$preds),

function(x) acc(y[fit\_hog$folds[[x]]],

fit\_hog$preds[[x]])))

acc\_fit\_hog

## [1] 0.9833714 0.9840571 0.9846857 0.9838857

cat('mean accuracy for hog-features using cross-validation :', mean(acc\_fit\_hog), '\n')

## mean accuracy for hog-features using cross-validation : 0.984

By changing from the simple svd-features to HOG-features the accuracy of a 4-fold cross-validation model increased from 97.48% to 98.4% (approx. 1% difference)

**CIFAR-10 data set**

CIFAR-10 is an established computer-vision dataset used for object recognition. The data I’ll use in this example is a subset of an 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes ( 6000 images per class ). Furthermore, the data were converted from RGB to gray, normalized and rounded to 2 decimal places (to reduce the storage size). More information about the data can be found in the *DataSets* repository (the folder includes also an Rmarkdown file).

I’ll build the kernel k-nearest-neighbors models in the same way I’ve done for the mnist data set and then I’ll compare the results.

system("wget https://raw.githubusercontent.com/mlampros/DataSets/master/cifar\_10.zip")

cifar\_10 <- read.table(unz("cifar\_10.zip", "cifar\_10.csv"), nrows = 60000, header = T,

quote = "\"", sep = ",")

**KernelKnnCV using truncated svd**

X = cifar\_10[, -ncol(cifar\_10)]

dim(X)

## [1] 60000 1024

# the KernelKnn function requires that the labels are numeric and start from 1 : Inf

y = cifar\_10[, ncol(cifar\_10)]

table(y)

## y

## 1 2 3 4 5 6 7 8 9 10

## 6000 6000 6000 6000 6000 6000 6000 6000 6000 6000

The parameter settings are similar to those of the mnist data,

svd\_irlb = irlba(as.matrix(X), nv = 40, nu = 40, verbose = F) # irlba truncated svd

new\_x = as.matrix(X) %\*% svd\_irlb$v # new\_x using the 40 right singular vectors

fit = KernelKnnCV(as.matrix(new\_x), y, k = 8, folds = 4, method = 'braycurtis',

weights\_function = 'biweight\_tricube\_MULT', regression = F,

threads = 6, Levels = sort(unique(y)))

# str(fit)

acc\_fit = unlist(lapply(1:length(fit$preds),

function(x) acc(y[fit$folds[[x]]],

fit$preds[[x]])))

acc\_fit

## [1] 0.4080667 0.4097333 0.4040000 0.4102667

cat('mean accuracy using cross-validation :', mean(acc\_fit), '\n')

## mean accuracy using cross-validation : 0.4080167

The accuracy of a 4-fold cross-validation model using truncated svd is 40.8%.

**KernelKnnCV using HOG (histogram of oriented gradients)**

Next, I’ll run the KernelKnnCV using the HOG-descriptors,

hog = HOG\_apply(X, cells = 6, orientations = 9, rows = 32,

columns = 32, threads = 6)

##

## time to complete : 3.394621 secs

dim(hog)

## [1] 60000 324

fit\_hog = KernelKnnCV(hog, y, k = 20, folds = 4, method = 'braycurtis',

weights\_function = 'biweight\_tricube\_MULT', regression = F,

threads = 6, Levels = sort(unique(y)))

# str(fit\_hog)

acc\_fit\_hog = unlist(lapply(1:length(fit\_hog$preds),

function(x) acc(y[fit\_hog$folds[[x]]],

fit\_hog$preds[[x]])))

acc\_fit\_hog

## [1] 0.5807333 0.5884000 0.5777333 0.5861333

cat('mean accuracy for hog-features using cross-validation :', mean(acc\_fit\_hog), '\n')

## mean accuracy for hog-features using cross-validation : 0.58325

By using hog-descriptors in a 4-fold cross-validation model the accuracy in the cifar-10 data increases from 40.8% to 58.3% (approx. 17.5% difference).