Nicholas Lytal
ID: 991834259
STA 250
3/21/14

Image Classification & Security Applications

Introduction

Many people are familiar with the existence of CAPTCHAs—miniature Turing tests designed to distinguish humans and computers. Most frequently, if someone wants to access a sensitive webpage they will receive a picture of some distorted letters and numbers, and required to enter them correctly to proceed. Ideally, this protects against computers gaining unwanted access by presenting a challenge only a human can solve, but advances in image identification algorithms have become increasingly effective at identifying such images automatically.

An alternative to CAPTCHA exists, known as Asirra. With this system, those who wish to access a webpage receive pictures of dogs and cats instead of letters and numbers, and prompted to identify twelve such animals in a row. While more complex than the CAPTCHA system, it may still be possible to build an algorithm that can distinguish cats from dogs a reasonable portion of the time and answer accordingly. If such an algorithm could exist, it may compromise the effectiveness of Asirra, requiring security programmers to explore another alternative.

Our goal in this project is to construct a few different types of classifiers that can distinguish cats from dogs based on features present in each image. By using linear regression and random forests, we may be able to construct a classifier that provides a better than average chance of identifying the animal depicted in an image. However, the main focus in this report is data management. Converting our available data into a usable format is the most important and involved part of the process, while the actual statistical techniques needed to create the classifiers are often time-consuming, yet relatively straightforward.

Data Management

Our data set consists of 25,000 jpeg files divided evenly between cats and dogs and identified as such. In their original state, the pictures come in many different sizes, and even pictures labeled as containing a cat or dog may contain several other subjects as well.

In order to construct our classifiers, we must first convert the data to a usable form. Our initial task is to standardize all the images in question to construct a classification matrix. We begin by converting each image in our available data to a 250x250 jpeg. Once the images are standardized, we can use the "jpeg" and "grDevices" packages in R to convert a given image's data into HSV (hue, saturation, and value) values that will form our color features. An image thus converted becomes a 250x250x3 matrix array, with one 250x250 matrix of values ranging from 0 to 1 corresponding to each feature (H, S, or V).

Using these results, we can split each picture into a number of horizontal and vertical strips, resulting in several cells of equal size. For each of these cells, we also divide the range of HSV values into equal sized blocks that cover the entire range of possible values, separately for each component. Once we establish these divisions, we then create a binary vector for each picture indicating the presence of any number of pixels in each HSV block for each of the created cells.

In our case, we split the picture matrices with five horizontal strips and five vertical strips to create cells that are 50 pixels each on a side. With each HSV component divided into eight separate bands, there are 8^3 HSV level combinations per cell, which for a group of 25 cells results in a size 12800 binary vector. Once we convert our pictures into these vectors, they will form a 25000x12800 matrix that represents all the color values present in our data set.

Conversion Tools

Due to the sheer number of pictures and the time our algorithm takes to run, we can't perform all the conversions on a single machine. Instead, we use a server to run the appropriate calculations. To convert several files simultaneously, we divide the 25000 images into 250 groups of 100 images each, whose calculations we can submit to Gauss as an array job. The code submitted to accomplish this consists of an R script file and a .sh shell script file, detailed in the Code Appendix. For each individual job we perform the data management steps above while also sorting the files appropriately into each job and saving the results as a CSV file. Once all of the jobs have finished, we then combine the resulting CSVs into a single large matrix. This resulting matrix contains all the color features for our data set.

Classification Methods

Now that we have all 25000 pictures converted, we divide the groups into training and test data sets, each containing an equal number of cat and dog pictures. After forming classifiers based on the training data set, we can then apply them to the test data set to measure their effectiveness at classifying future images. We can design these classifiers around a few different methods, such as regression and ensemble methods.

For our first method, we consider generalized linear regression. Given a large number of observations in p-dimensional space, we can construct the p-dimensional plane that most effectively separates our cat pictures from our dog pictures. However, due to the sheer number of properties in each picture, we have more variables than images to regress upon. As a result, we must use penalized regression to fit our general linear model by penalized maximum likelihood. The R package "glmnet" is effective at using sparse matrices like our color features matrix, which have only a few non-zero entries among many zeroes, to perform the necessary regression. The formula[1] it uses for regression is:

[1] Formula taken from the Glmnet Vignette at http://www.stanford.edu/~hastie/glmnet/glmnet_alpha.html

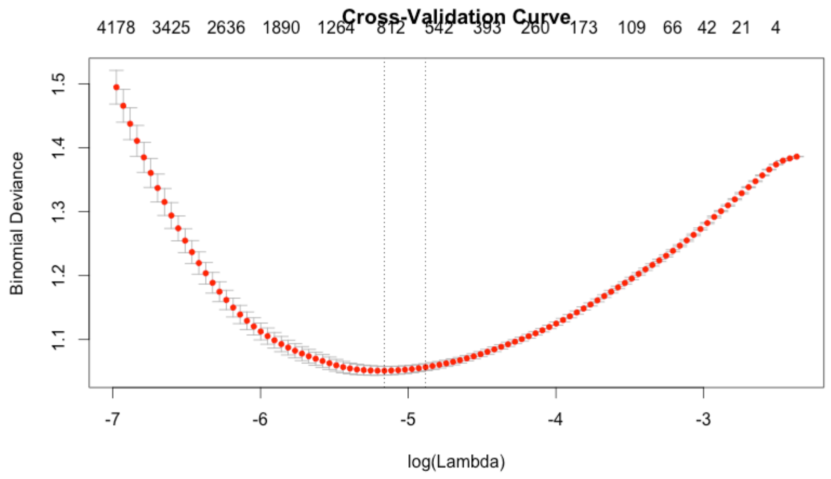
$$\min_{\beta,\beta_0} \frac{1}{N}\sum_{i=1}^{N} w_i l(y_i,\beta_0+\beta^T x_i)+\lambda\left[\frac{(1-\alpha)\|\beta\|_2^2}{2}+\alpha\|\beta\|_1\right]$$

where l represents the negative log-likelihood, and the formula above is tested over a large range of penalty terms λ.

Our other method, random forests, is based on a simpler ensemble method known as "bagging" or bootstrap aggregating. Bagging involves the construction of many decision trees, which continuously classify data into separate groups based on the value of certain variables in a data set. We can average the findings of the resulting trees to obtain a classifier for our data. Random forests expand on this method by using a random selection of the existing variables at each tree's decision splits to avoid overfitting to the training data. Over time, this method considers a greater variety of potentially useful variables, rather than simply the best one every time.

Results

When designing our fit for penalized regression, we use cross-validation to obtain the ideal λ on which to predict our test data and we get the following results:

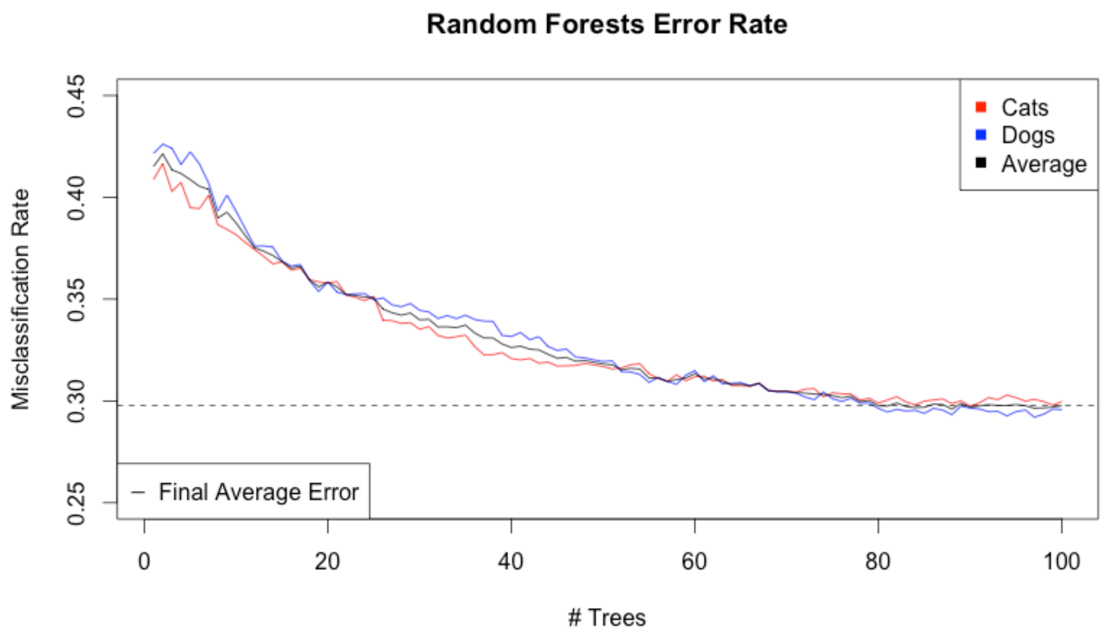


The initial fit suggests we should use a tuning parameter of about λ = 0.00573, or log(λ) = -5.16. By applying this tuning parameter to the prediction, we then predict the test data with the following accuracies:

```
                    ACTUAL
                     cats dogs
PREDICTED       cats 4801 1449
                dogs 1821 4429
```

This works out to about a 73.8% chance of identifying pictures correctly.

When using random forests, we can get progressively more accurate results (to a certain extent) the more trees we consider, and we can observe the prediction accuracy for the first 100 trees constructed:



After 100 trees, we achieve about a 70% chance of correctly identifying a single picture. We could construct additional trees beyond 100, but the gains are likely to be minimal and would not be worth the additional time. Just 100 trees take over an hour to construct with default settings, and even after that length of time, the end result is less effect than using penalized linear regression, which takes only about 18 minutes in comparison.

Conclusions

Taking our penalized regression classifier as our best result, we obtain about a 73.8% accuracy rate at identifying a cat or dog picture correctly. Though this is not particularly reliable, it is a significant improvement over random chance, and over 100 times more likely to succeed at completing a test run of twelve consecutive pictures. This is still only a 2.6% chance of success for such a run, but this percentage may well be large enough to reconsider implementing the Asirra system if a relatively straightforward classifier can yield such a result. With a few improvements, this chance could increase enough to be a credible threat to the security of the system.

Future Improvements

The classification matrix used consisted of only one set of parameters. With considerably more time and a revised, more efficient algorithm, we could compare the effectiveness of classification matrices with a different number of cells or HSV component divisions. Theoretically, a larger number of cells would allow for a more accurate representation of the color features within, though the size of the matrix would be considerably greater. Naturally, a larger number of pictures for our training data would provide a better basis for the classifiers, though alone they will be limited by the methods used.

Color features are only one aspect of the image. We could in addition consider texture features, which depict actual shapes in small portions of the images rather than just coloration. By constructing a large collection of reasonably disparate texture cells, we could compare each of the training images to the collection and record a vector of “distances” between the pixels of the image and the collection’s tiles in terms of RGB values.

Other classifiers may offer a more effective means of identifying pictures correctly. For instance, support vector machines were the classifier of choice mentioned by the original Kaggle project, and could offer improvements over both penalized linear regression and random forests. There may well be other methods even more effective at classification if time constraints are not an issue, such as deep neural networks, that would be worth pursuing as well.

R Code Appendix

```
# dogcat_misc.R

# Contains miscellaneous data conversion code not
# present in the R file submitted to Gauss for conversion

# ****** Initial Image Conversion ******

library(jpeg)

# Due to differences in dimensions, we should STANDARDIZE
# the image sizes by rescaling to a common size (250 x 250)

# Shell code to do this
# Rescaled images to 250x250
convert dog.*.jpg -resize 250x250\! dog.rescaled.jpg

convert cat.*.jpg -resize 250x250\! cat.rescaled.jpg
# WARNING: This does NOT work when applied to all 15000
# jpgs at once...a memory leak occurs. Up to 1000 seems
# to work without a problem though.

# SOLUTION: shell script as follows:

for f in cat.*.jpg
do
echo "Processing $f"
newname=${f/cat/rescaled.cat}
echo "Converting to: $newname"
convert $f -resize 250x250\! $newname
echo "done."
done

for f in dog.*.jpg
do
echo "Processing $f"
newname=${f/dog/rescaled.dog}
echo "Converting to: $newname"
convert $f -resize 250x250\! $newname
echo "done."
done

# dogcat_sim.R contains functions not shown here

# ****** Converting Complete Jobs into Usable Form ******

# B = number of jobs
# files.per = files contained in each job
# cols = number of columns in data (12800 here)
csvComboAlt = function(dir,B,files.per, cols)
{
    output = matrix(NA,B*files.per,cols)
    for(i in 0:(B-1))
    {
        file = paste0(dir,"hsvfeat.job.", i, ".csv")
        data = read.csv(file, quote="\"", header = FALSE)


        output[(1+(i*files.per)):((i+1)*(files.per)),]=
        matrix(as.integer(unlist(data)),nrow=files.per, byrow = F)
    }
    # Turns matrix of doubles into one of integers to save space
    output
}

start = proc.time()
x = csvComboAlt(directory,25,100,12800)
time = proc.time() - start

# dogcat_class.R

# Contains classification methods for the Dog/Cat Asirra Project

library("e1071")
library("jpeg")
library("randomForest")

# Take matrix and test half the files against the other half
# (12500 vs 12500)

# METHODS

# 1) Logistic Regression

# 2) Random Forests

# *********** DEFINING TRAINING & TEST DATA ***************

# rows: number of rows in original matrix
# sampsize: number of entries in teach train & test set
# (defaults to 500), so train 1000, test 1000
# if set to 6250: train 12500, test 12500
# Returns the indices to use in order to sample
# two equal sized sets of cats and dogs
samp.nums = function(rows, sampsize = 500)
{

    cats.nums = sample(1:(rows/2), sampsize*2, replace = FALSE)
    cats.nums.tr = cats.nums[1:sampsize]
    cats.nums.ts = cats.nums[(1+sampsize):(sampsize*2)]
    dogs.nums = sample((1+(rows/2)):rows, sampsize*2, replace = FALSE)
    dogs.nums.tr = dogs.nums[1:sampsize]
    dogs.nums.ts = dogs.nums[(1+sampsize):(sampsize*2)]

    indices = data.frame(cats.nums.tr, cats.nums.ts,
                         dogs.nums.tr, dogs.nums.ts)
    indices
}

# FULL TRAINING & TEST DATA

FULL DATA: nums = samp.nums(25000, 6250)
# SAMPLE ONLY: nums = samp.nums(25000, 500)

train.samp.c = x[(nums[,1]),] # Select cats.nums.tr indices
train.samp.d = x[(nums[,3]),] # Select dogs.nums.tr indices
train.samp = rbind(train.samp.c, train.samp.d) # Combine
train.indices = c(nums[,1],nums[,3]) # Identifies indices chosen
train.y = c(rep(0,6250),rep(1,6250))
test.samp.c = x[(nums[,2]),] # Select cats.nums.ts indices
```

```r
test.samp.d = x[(nums[,4]),] # Select dogs.nums.ts indices
test.samp = rbind(test.samp.c, test.samp.d) # Combine
test.indices = c(nums[,2],nums[,4]) # Identifies indices chosen
test.y = c(rep(0,6250),rep(1,6250))


# Used for testing only a small number of pictures
# mini.train.y = train.y[6201:6300]
# mini.train.samp = train.samp[6201:6300,]
# mini.test.y = test.y[6201:6300]
# mini.test.samp = test.samp[6201:6300,]

# ****** GLM CLASSIFICATION ******

# For a set with fewer variables, could use:
# fit1 = glm(mini.train.y~mini.train.samp, family = binomial)
# NOTE: This will NOT work! There's too few entries, even with the entire
# training data, to get a result that isn't NAs.
# Instead we must use PENALIZED GLM.

# Cross-validated penalized GLM with glmnet

glm.start = proc.time()
fit2 = cv.glmnet(x = train.samp, y = train.y, family = "binomial")
pred2 = predict(fit2, newx = test.samp, type = "class", s = fit2$lambda.min)
glm.time = proc.time() - glm.start

# Error Rate of Classification
glm.cat.err = sum(as.numeric(pred2[1:6250]))/6250
glm.dog.err = 1 - sum(as.numeric(pred2[6251:12500]))/6250
glm.avg.err = mean(c(glm.cat.err, glm.dog.err))

# Constructing a table of error values
glm.cats = c(6250-(glm.cat.err*6250),glm.cat.err*6250)
glm.dogs = c(glm.dog.err*6250, 6250-(glm.dog.err*6250))
glm.df = as.data.frame(matrix(c(glm.cats,glm.dogs),2,2, byrow = TRUE))
row.names(glm.df) = c("cats", "dogs")
colnames(glm.df) = c("cats", "dogs")
glm.df

plot(fit2, main = "Cross-Validation Curve")
coef(fit2, s = "lambda.min")

# ****** RANDOM FOREST CLASSIFICATION ******
train.y.f = as.factor(train.y)
test.y.f = as.factor(test.y)


# Sample version
rf.start = proc.time()
rf.train = randomForest(x = train.samp, y = train.y.f,
                        xtest = test.samp, ytest = test.y.f,
                        ntree=50)
rf.time = proc.time() - rf.start

# Full version - 100 trees
train.y.f = as.factor(c(rep(0,6250),rep(1,6250)))
test.y.f = as.factor(c(rep(0,6250),rep(1,6250)))

rf.start.100 = proc.time()
rf.trees.100 = randomForest(x = train.samp, y = train.y.f,
                        xtest = test.samp, ytest = test.y.f,
                        ntree=100)
rf.time.100 = proc.time() - rf.start.100
# Mistake: take rf.time.100 = rf.time.100 + rf.start - rf.start.100
rf.time.fixed = rf.time.100 + rf.start - rf.start.100

# Once we have rf.trees.100, we can observe the error present
# for a given number of trees.

rf.cat.err = rf.trees.100$err.rate[,2]
rf.dog.err = rf.trees.100$err.rate[,3]
rf.avg.err = c(rf.cat.err + rf.dog.err)/2

# Error Rate Plot - Random Forests
plot(rf.cat.err, type = "l", main = "Random Forests Error Rate",
     col = "red", ylim = c(0.25,0.45), xlab = "# Trees",
     ylab = "Misclassification Rate")
lines(rf.dog.err, col = "blue")
lines(rf.avg.err, col = "black")
abline(h = rf.avg.err[100], lty=2)
legend("topright", legend = c("Cats", "Dogs", "Average"),
       col = c("red", "blue", "black"), pch = 15)
legend("bottomleft", legend = "Final Average Error",
       col = "black", pch = "_")

# dogcat_sim.R
# This job array comprises 250 jobs of 100 files each.

HSVConvert = function(file)
{
    jpg = readJPEG(file) # Imports JPEG in RGB format

    # Orders RGB values into a matrix
    x = matrix(jpg, nrow = dim(jpg[,,1])[1]*dim(jpg[,,1])[2],
               ncol = 3, byrow = FALSE)
    x = t(x) # Now in correct format (3x250^2)

    # Converts to HSV format
    x.hsv = rgb2hsv(x)
    x.hsv = t(x.hsv)
    x.hsv[,3] = x.hsv[,3]*255 # Naturally ranges 0 - 1/255 otherwise

    # Each pixel is listed in "order", want to recompose the picture
    hsv.array = array(x.hsv, c(250,250,3))

    # This is the original structure of the RGB data,
    # now in HSV form
    hsv.array
}

# x is a single row of the pixel matrix
# idx.
outer3 = function(x, idx.h, idx.s, idx.v)
{
    h = x[idx.h]
    s = x[idx.s]
    v = x[idx.v]
    as.vector(outer(as.vector(outer(h,s)),v)) # returns outer product
}
```

```
# This function takes an individual 250x250 jpeg and extracts
# all the hue, saturation, and value (HSV) features in the
# form of a vector.
# col = # of column strips
# row = # of row strips
# t = # of table divisions for H, S, and V respectively
# N = # pixels on a side for the picture
# jpeg = The file in question
# NOTE: Assumes square image, but could probably modify
# to allow for different picture side length, so long as
# they are multiples of col & row respectively.
HSVfeatures = function(col=5, row=5, t=c(8,8,8), N=250, jpeg)
{
    data = HSVConvert(jpeg) # Convert a JPEG into HSV values
    cells = numeric(0) # vector to contain HSV values
    pixel.h = numeric(0)
    pixel.s = numeric(0)
    pixel.v = numeric(0)
    # MUST define pixel to contain values

    # This code defines the intervals that each cell
    # contains. For col = 10, this produces a
    # 25x10 matrix, with each column representing the values
    # for a single cell.
    col.start = seq(1,N,N/col)
    col.end = seq((N/col),N,N/col)
    col.ints = matrix(0,N/row,col)
    for(i in 1:col)
    {
        # Each column of col.ints contains the values contained
        # in each start-end range pair
        # EX: If N = 5, col.ints[,j] = 1:50
        col.ints[,i] = col.start[i]:col.end[i]
    }
    row.start = seq(1,N,N/row)
    row.end = seq((N/row),N,N/row)
    row.ints = matrix(0,N/col,row)
    for(j in 1:row)
    {
        # Each column of row.ints contains the values contained
        # in each start-end range pair
        # EX: If N = 5, row.ints[,j] = 1:50
        row.ints[,j] = row.start[j]:row.end[j]
    }


    # full_start = proc.time() # Used for testing speed

    # The features vector we ultimately want for a single picture
    # We need values for each of the three features (H, S, and V,
    # divided into t table entries each), for each cell
    # (col*row different cells exist)
    # (250/col vertical strips and 250/row horizontal strips)
    pic.features = rep(NA,prod(t)*col*row)

    block.ctr = 1       # Counting variable
    hsv.ncomb = prod(t) # Number of HSV table value combinations

    for(c in 1:col)
```

```
{
    for(r in 1:row)
    {
        # Selects the group of vector indices of length 512
        # to represent the current cell
        block.idx <- c((1 + ((block.ctr-1)*hsv.ncomb)):(hsv.ncomb*block.ctr))
        block.ctr = block.ctr + 1
        # For one row of the cell vector
        # Considers row r, column c, and slice s
        # t[s] is used to determine the table sizes

        # End result: 2500x512 vector: each column is a combination of
        # table values.

        start = proc.time()
        pixel.h <- matrix(NA,nrow=(N/row)*(N/col),ncol=t[1])
        pixel.s <- matrix(NA,nrow=(N/row)*(N/col),ncol=t[2])
        pixel.v <- matrix(NA,nrow=(N/row)*(N/col),ncol=t[3])

        # Next task: Create a SINGLE pixel matrix for h, s, and v
        # combined, which will be sum(t) columns
        pixel <- matrix(NA,nrow=(N/row)*(N/col),ncol=sum(t))

        idx.h <- c(1:t[1])
        idx.s <- c((t[1]+1):(t[1]+t[2]))
        idx.v <- c((t[1]+t[2]+1):sum(t))

        idx <- 1
        nsub_r <- (N/row)
        nsub_c <- (N/col)
        if (abs(nsub_r - as.integer(nsub_r))>.Machine$double.eps){
            stop("'N/row' must be an integer")
        }
        if (abs(nsub_c - as.integer(nsub_c))>.Machine$double.eps){
            stop("'N/col' must be an integer")
        }

        for(pr in 1:nsub_r)
        {
            for(pc in 1:nsub_c)
            {
                # t[1:3] = C.h,C.s,C.v
                pixel[idx,idx.h] = table(cut(data[row.ints[pc,r],col.ints[pr,c],1],
                                             seq(0,1,1/t[1])))
                pixel[idx,idx.s] = table(cut(data[row.ints[pc,r],col.ints[pr,c],2],
                                             seq(0,1,1/t[2])))
                pixel[idx,idx.v] = table(cut(data[row.ints[pc,r],col.ints[pr,c],3],
                                             seq(0,1,1/t[3])))
                idx <- idx+1
                # CHANGE: Need to count through pixel by pixel, so 1:N/col, 1:N/row
                # cat("Finished pr=",pr,",pc=",pc,"...\n")
            }
        }
        result = apply(pixel, 1, outer3, idx.h = idx.h, idx.s = idx.s, idx.v = idx.v)
        # applies by row to the whole pixel matrix
        # yields 62500x512 matrix
        # Take column sums: if any are >1, reduce to 1.
        # pic.features will be a numeric(12800) vector of HSV features
        pic.features[block.idx] = as.numeric(rowSums(result) > 0)
```

```
            # Result: vector of length prod(t) (here, 8^3 = 512)
            #
            time = proc.time() - start
            # This gives a 250^2 x 8 grid of table values for each pixel variable
            # Now we need to overlay all three grids to pinpoint which combinations
            # of the three yield at least one "hit"

            cat("Finished r=",r,",c=",c,"...\n",sep="") # Monitors progress

        }

    }
    pic.features
}
########################################################################################
########################################################################################
## Handle batch job arguments:

# 1-indexed version is used now.
args <- commandArgs(TRUE)

cat(paste0("Command-line arguments:\n"))
print(args)

####
# sim_start ==> Lowest simulation number to be analyzed by this particular batch job
###

#######################
sim_start <- 0
length.datasets <- 25000
batch.size = 100 # Can alter to test
jobs = length.datasets/batch.size
jobint = as.integer(jobs)
if(abs(jobs-jobint) > .Machine$double.eps)
{
    stop("Number of images not divisible by number of jobs.")
}
#######################

if (length(args)==0){
    sinkit <- FALSE
    sim_num <- sim_start + 1
    set.seed(1162977)
} else {
    # Sink output to file?
    sinkit <- TRUE
    # Decide on the job number, usually start at 1000:
    # NEED TO CHANGE THIS TO BE LENGTH batch.size
    sim_num <- sim_start + as.numeric(args[1]) - 1
    # Set a different random seed for every job number!
    set.seed(649*sim_num + 1162977)
}

# This identifies picture names properly to be included in
# a list of files to convert

if (sim_num < (jobs/2))
{
    animal = "cat"
    img.list = (sim_num*batch.size):(((sim_num+1)*batch.size) - 1)
} else {
    animal = "dog"
    img.list = ((sim_num-(jobs/2))*batch.size):(((sim_num+1-(jobs/2))*batch.size) - 1)
}

################################################################################
################################################################################

# Identifies all jpg files in the directory
library("jpeg")

# infile consists of all files numbered according to "num" and
# identified according to their animal
infile = lapply(img.list, FUN=function(x) paste0("rescaled.", animal, ".", x, ".jpg"))

N=250
grid=c(8,8,8)
col=5
row=5

results.mat = matrix(NA,batch.size,prod(grid)*col*row)

for(i in 1:batch.size)
{
    results.mat[i,] = HSVfeatures(jpeg=infile[[i]], col=col,row=row,t = grid, N=N)
    cat("Finished image",i,"\n")
}

##################################################

outdir <- "data/"
outfile_data <- paste0(outdir,"hsvfeat.job.", sim_num, ".csv")

# We save the data through this code
write.table(results.mat,file=outfile_data,sep=",",quote=FALSE,row.names=FALSE,col.names=F
ALSE)

# **************************
# dogcat_sim.sh

#!/bin/bash -l

###############################################################################
##
## NOTES:
##
## (1) When specifying --range as a range it must start from a positive
##     integer e.g.,
##       #SARRAY --range=0-9
##     is not allowed.
##
## (2) Negative numbers are not allowed in --range
##     e.g.,
##    %   #SARRAY --range=-5,-4,-3,-2,-1,0,1,2,3,4,5
##     is not allowed.
##
## (3) Zero can be included if specified separately.
##    e.g.,
##       #SARRAY --range=0,1-9
##     is allowed.
##
## (4) Ranges can be combined with specified job numbers.
##    e.g.,
##       #SARRAY --range=0,1-4,6-10,50-100,1001-1002
##     is allowed.
##
###############################################################################

module load R

# Name of the job - customized.
#SBATCH --job-name=dog_cat_data
# Specify range of jobs to run - passed into R as 'args'
#SARRAY --range=1-250

# Standard out and Standard Error output files with the job number in the name.
#SBATCH --output=dump/dog_cat_data_%j.out
#SBATCH --error=dump/dog_cat_data_%j.err

# Execute each of the jobs with a different index (the R script will then process
# this to do something different for each index):
srun R --no-save --vanilla --args ${SLURM_ARRAYID} < dogcat_sim.R
```