Data Science and Economics

Machine learning, statistical learning, deep learning and artificial intelligence

Road signs recognition in R: pre- and post-Neural Networks approaches

Andrea Joseph Froiio

922907

Index

Chapter 1

Introduction

The goal of this paper is to show the advancing in image recognition approaches, comparing some famous algorithms used before the arrival of Neural Networks and the Convolutional Neural Networks, which are nowadays the best tool for object recognition.
Computational times and accuracies will be compared and will be the main indicators for which approach is better.
These tasks will be performed on a classification of road signs, which are easily distinguishable by the human eye because of their bright colours and different shapes, but is one of the challenges of the automotive industry of the future, which will have to implement fast and reliable algorithm for real-time recognition of all kind of road signals in self-driving cars.
The algorithm that will be used are the following:

- *Histogram of Oriented Gradients*: is the simplest of the 3, provides efficient and robust results.
- *Scale Invariant Feature Transformation*: more complex than HOG and slower, but proven to have high performances.
- *Convolutional Neural Networks*: the most recent one, the most precise, built to replicate how a human eye sees the world.

Chapter 2

Data and methods

2.1 Dataset

The *German traffic signs recognition benchmark* is a dataset containing more than 50000 photos of 43 different classes of road signs that was initially published as a competition held at the International Joint Conference on Neural Networks in 2011.
The dataset is an archive divided as follow:

- *Train.csv*: which is a dataframe containing information about the 39209 images in the training set:
  - *Width*: the width of the image.
  - *Height*: the height of the image.
  - *Roi.X1*: the upper left X-axis coordinate of the road sign inside the photo.
  - *Roi.Y1*: the upper left Y-axis coordinate of the road sign inside the photo.
  - *Roi.X2*: the lower right X-axis coordinate of the road sign inside the photo.
  - *Roi.Y2*: the lower right Y-axis coordinate of the road sign inside the photo.
  - *ClassId*: the class of the road sign, spanning from *0* to *42*.
  - *Path*: the path where to find the image once *Train.tar* is extracted.
- *Test.csv*: which is a dataframe containing the same information about the 12630 images in the test set:
  - *Width*: the width of the image.
  - *Height*: the height of the image.
  - *Roi.X1*: the upper left X-axis coordinate of the road sign inside the photo.
  - *Roi.Y1*: the upper left Y-axis coordinate of the road sign inside the photo.
  - *Roi.X2*: the lower right X-axis coordinate of the road sign inside the photo.
  - *Roi.Y2*: the lower right Y-axis coordinate of the road sign inside the photo.
  - *ClassId*: the class of the road sign, spanning from *0* to *42*.
  - *Path*: the path where to find the image once *Test.tar* is extracted.
- *Meta.csv*: which is a dataframe containing more information about 43 icons representing each class of the road signs:
  - *Path*: the path where to find the image once *Meta.tar* is extracted.
  - *ClassId*: the class of the image, in this file there is only one per class.
  - *ShapeId*: the shape of the sign.
  - *ColorId*: the colour of the sign.

  - *SignId*: the ID of the road sign according to the Ukrainian Traffic Rule. These information are not needed for our purpose and so this file will not be used.

- *Train.tar*: the archive containing the actual .png(s) of the training set.
- *Test.tar*: the archive containing the .png(s) of the test set.
- *Meta.tar*: an archive containing the .png(s) of the icons representing the road signs.

2.2 Support Vector Machines

Support Vector Machines (SVM) is a classification technique that splits the data in "the best possible way", which means that it finds the farthest hyperplane w^* from the *support vectors* by maximizing the margin between them, making it the one with the least probability of misclassification.
Support Vectors are the closest positive and negative x_t to the hyperplane, on which it has margin equal to 1: $y_t(w^*)^T x_t = 1$.
Given a linearly separable training set $(x_1, y_1), ..., (x_m, y_m) \in \mathbb{R}^d \times \{-1, +1\}$ to maximize the margin, you must solve a constrained optimization problem

$$\min_{w \in \mathbb{R}^d} \frac{1}{2}||w||^2 \tag{2.1}$$

$$\text{s.t. } y_t w^T x_t \geq 1$$
$$t = 1, ..., m$$

Where

$$w^* = \sum_{t \in I} \alpha_t y_t x_t \tag{2.2}$$

knowing that, if w_0 is an optimal solution, $\alpha \in \mathbb{R}^m$ is a vector such that $\nabla f(w_0) + \sum_{t \in I} \alpha_t \nabla g_t(w_0) = 0$ in which $I = \{1 \leq t \leq m : g_t(w_0) = 0\}$ and g_t is a differentiable function.
Since data is rarely linearly separable, the SVM objective function can be generalized for non-linearly separable cases:

$$w^* = \underset{w \in \mathbb{R}^d}{\text{argmin}} \left(\frac{1}{m} \sum_{t=1}^{m} \xi_t + \frac{\lambda}{2}||w||^2 \right) \tag{2.3}$$

Where $\xi_t \geq 0$ is the *slack* and λ is the *regularization* parameter introduced in order to balance the two.
The slack variable ξ_t measures the violation of each *support vector* by the potential solution w.
To minimize it: $\xi_t = [1 - y_t w^T x_t]_+ = h_t(w)$, which is the *hinge loss*, so the problem can be rewritten as

$$w^* = \underset{w \in \mathbb{R}^d}{\text{argmin}} \left(\frac{1}{m} \sum_{t=1}^{m} h_t(w) + \frac{\lambda}{2}||w||^2 \right) \tag{2.4}$$

2.3 Histogram of Oriented Gradients

Histogram of Oriented Gradients (HOG) is an algorithm that can be used for object detection; it works by calculating horizontal and vertical gradients, their magnitudes and direction for each pixel of the image.

p_1	p_2	p_3
p_4	p_i	p_6
p_7	p_8	p_9

Where the value of the ps are the value in RGB or greyscale.
For each pixel p_i in the image are computed:

- *Gradient value in the X-direction*: $gvX = p_6 - p_4$

- *Gradient value in the Y-direction*: $gvY = p_2 - p_8$
- *Feature vector*: $[gvX, gvY]$
- *Gradient magnitude*: $gm = \sqrt{(gvX)^2 + (gvY)^2}$
- *Gradient angle*: $ga = \tan^{-1}\left(\frac{gvX}{gvY}\right)$

Then the distribution of angles inside each cell of pixels (ex: 8×8) is plotted as a histogram where each bin represent an interval of angles (ex: 16 bins 0-180^o, 11.25^o each bin), this reduces 64 features into just 16.

2.4 Kmeans

K-means is a clustering algorithm based on Euclidean distances between data points, partitioning the observations into k sets so as to minimize the objective function:

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} ||x_i^{(j)} - c_j||^2$$

It works like this:

- Starts with k randomly selected points as centroids of the clusters
- Assign every point to a cluster whose centroid is the closest to the point
- Recompute the centroid for each cluster based on the newly assigned points in the cluster
- Repeat until the algorithm converges

2.5 Scale Invariant Feature Transformation

Scale Invariant Feature Transformation (SIFT) is an algorithm for local features detection in images.
It works like this:

- *Build a scale space*: to get rid of small scale details, an half-size copy of the original image is generated, then another half-size copy is made for the newly created one (octave), and so on until you have the desired number of images, one half the size of the previous one.
  Then, to each one, a Gaussian blur is applied and several progressively blurred (scale) images are generated:

	1st octave	2nd octave	3rd octave	4th octave
Image 0	256×256	128×128	64×64	32×32
Image 1: blurred image 0	256×256	128×128	64×64	32×32
Image 2: blurred image 1	256×256	128×128	64×64	32×32
Image 3: blurred image 2	256×256	128×128	64×64	32×32
Image 4: blurred image 3	256×256	128×128	64×64	32×32

  The parameters of octaves and scales depend on the size of the original image but it is recommended to use 4 octaves and 5 levels of blur.

– *Gaussian blurring*: it is a function applied to the image:

$$L(x, y, \sigma) = G(x, y, \sigma) \times I(x, y)$$

Where:

* L is the blurred image result.
* G is the Gaussian blur operator:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

* I is the original image the blurring is applied to.
* x, y are the coordinates of each pixel.
* σ is the level of blurring.

- *LoG Approximation*: generate another set of images useful to find keypoints within the images.
  Since computing the Laplacian of Gaussian (LoG) for images means calculating second order derivatives which is computationally slow, an approximation for it is the Difference of Gaussian (DoG), which is computed by taking the differences between two consecutive scales in each octave:

Gaussian	Difference of Gaussians
Image 1	DoG 1: Image 1 - Image 2
Image 2	DoG 2: Image 2 - Image 3
Image 3	DoG 3: Image 3 - Image 4
Image 4	DoG 4: Image 4 - Image 5
Image 5	

  These approximations are scale invariant, they do not depend on the amount of blur σ.

- *Finding the keypoints*:

  – *Locating max/min in DoGs*: in each octave, a pixel is marked as a keypoint if it is greater than all its 26 neighbours in a 3D space, where the 3rd dimension is created by adding 2 layers of DoG images, the image above and the one below it (the first and the last one are skipped):

DoG 1 [r-1,c-1]	DoG 1 [r-1,c]	DoG 1 [r-1,c+1]
DoG 1 [r,c-1]	DoG 1 [r,c]	DoG 1 [r,c+1]
DoG 1 [r+1,c-1]	DoG 1 [r+1,c]	DoG 1 [r+1,c+1]
DoG 2 [r-1,c-1]	DoG 2 [r-1,c]	DoG 2 [r-1,c+1]
DoG 2 [r,c-1]	X [r,c]	DoG 2 [r,c+1]
DoG 2 [r+1,c-1]	DoG 2 [r+1,c]	DoG 2 [r+1,c+1]
DoG 3 [r-1,c-1]	DoG 3 [r-1,c]	DoG 3 [r-1,c+1]
DoG 3 [r,c-1]	DoG 3 [r,c]	DoG 3 [r,c+1]
DoG 3 [r+1,c-1]	DoG 3 [r+1,c]	DoG 3 [r+1,c+1]

  – *Find subpixel max/min*: usually the max/min does not lie on a pixel, but somewhere "between" pixels, so a Taylor expansion of the image around the candidate keypoints is done to generate subpixel values:

$$D(x) = D + \frac{\partial D^T}{\partial x} + \frac{1}{2} x^T \frac{\partial^2 D}{\partial x^2} x$$

  Which is differentiated and put equal to 0.

- *Get rid of bad keypoints*: edges and low contrast keypoints are not useful and decrease efficiency and robustness of the algorithm:
  - *Remove low contrast keypoints*: remove all keypoints that have absolute value of the intensity in the DoG image lower than some treshold.
  - *Remove edges*: calculate two perpendicular gradients at the keypoint:
    * *both small* $\rightarrow$ **flat region**.
    * *one big, one small* $\rightarrow$ **edge**.
    * *both big* $\rightarrow$ **"corner"**.

    Since corners are good keypoints, they are kept if both gradients are big enough.

- *Orientation of keypoints*: to have rotation invariant features, the Histogram of Oriented Gradients (HOG) approach is used, computing gradients magnitude and orientation for each region around keypoints and putting them in an histogram (the higher the number of bins the higher the accuracy).
  Any bin above 80% of the highest one is converted into a new keypoint having the same location and scale, but with its different orientation.

  Magnitude:

  $$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

  Orientation:

  $$\theta(x,y) = \tan^{-1}\left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}\right)$$

- *Generate SIFT features*: this is used to create object fingerprints in images, to be able to distinguish it from other keypoints.
  - create 16×16 cell around the keypoint.
  - split it into 4×4 cells.
  - within each 4×4 region compute magnitudes and orientations.
  - multiply the magnitudes by "Gaussian weights" that are proportional to the distance from the keypoint.
  - put them in an 8 bin histogram.
  - doing this for all 16 pixels (4×4 region) in 8 bins histogram makes you end up with $16 \times 8 = 128$ features that uniquely identify your keypoint.
  - normalize them.

2.6 Convolutional Neural Networks

A Convolutional Neural Network is an algorithm that differs from a Deep Neural Network by a "filter" that is applied to the images before passing them to the DNN, otherwise some patterns would not be spotted (with convolutional layers).
The filter is a series of "weigths" applied to the value of the neighbours of the pixels to find the new value of the center pixel.
This is combined with "pooling", which groups pixels within the image and filter them down into subsets (ex: take 2×2 groups and pick the largest), thus reducing its size.
The filters are parameters that are learned by the CNN, which starts from a random set (with predetermined cardinality) and iterates layers, keeping the ones with the higher matches each epoch.
The more the layers, the more complicated patterns can be recognized.

Chapter 3

Prediction

The goal is to train and test three multi-class classifiers, each to recognize the ID of the road signs in some photos.
All computations are made on an Intel i7-8565U up to 4.60 GHz.

3.1 HOG + SVM

To make sure that the prediction is accurate and the number of features of the images are all the same, the images are read, cropped using the respective coordinates of the road sign inside the photo (some images for some reason stop the algorithm from running by giving errors, so a *tryCatch* is put up in place to pass onto the next one once it finds an error, inevitably some images are lost in the process), all resized in 50×50 and greyscaled (a colour copy is stored for later use in other approaches).

Algorithm 1: Reading, cropping, resizing and storing

Input : Training images paths
Initialize List of grey images, List of colour images
for $i = 1, ..., length(Trainingset)$ **do**
    1) Get the whole path by combining the working directory and the path
    2) Read the image
    3) Crop the image
    4) Resize the image
    5) Make a copy
    6) Greyscale it
    7) Store them each in their respective list
end

The new lists will have *NULL* entries for where the *tryCatch* found an error so those rows need to be removed from it and from the vector of labels:

Algorithm 2: Removal of *NULL* rows

Input : List of grey images, List of colour images, List of labels
Initialize $k = 1$, Vector of *NULL* rows indexes
for $i = 1, ..., length(Listofgreyimages)$ **do**
    if *row i is NULL* **then**
        1) Store index i in the vector
        2) $k = k + 1$
    end
end
The Lists of images and the List of labels becomes themselves without the rows in the new vector.

An HOG function with 16 orientation bins, which grants an acceptable precision, is then run on each image:

Algorithm 3: HOG

Parameters : orientations = 16
Input : List of grey images
Initialize List of HOG descriptors
for $i = 1, ..., length(Listofgreyimages)$ **do**
  1) Apply HOG function on image i
  2) Store the output in the list
end

The output is then reshaped by creating a matrix with a row for each image, containing the values of the HOG output one next to each other, so to have a feature correspondence for each column of the matrix:

Algorithm 4: Reshape of X

Input : List of HOG descriptors
Initialize Matrix of HOG descriptors
for $r = 1, ..., length(ListofHOGdescriptors)$ **do**
  for $c = 1, ..., length(Objectinthelist)$ **do**
    1) Copy the element in the matrix in position $[r, c]$
  end
end

The same passages are done for the test set.
SVM is then run and used to train the algorithm:

Algorithm 5: SVM

Input : Matrix of HOG descriptors, List of labels
Run SVM function
Predict and show Confusion matrix for training and test.

The algorithm is run 3 times: once with a training sample of 4000 random images, once with the "minimum cardinality" balanced training set and finally with all the set available.
Since each class contains different photos of the same type of road sign and for each one there are 30 versions of it, from the smallest to the biggest with the highest resolution, the best balanced set is obtained with the following approach, where the number of images to be taken from each set of 30 versions of images is computed like:

$$n = ceiling\ rounding\left(\frac{210}{\frac{C.C.}{30}}\right)$$

Where:

- *210* is the number of images in the class with the least amount of them.
- *C.C.* is the cardinality of that class.
- *30* is the number of versions of each image.

The last n images for each set of 30 will be taken, to work with the highest resolution possible.
The results are:

	Training set cardinality	Test set cardinality	Training set HOG time	Training SVM time	Test set HOG time
Random set	4000	1000	00:00:00.84	00:00:10.06	00:00:00.21
Balanced set	9681	2580	00:00:01.86	00:00:36.10	00:00:00.54
All set	39209	12630	00:00:07.98	00:09:28.33	00:00:02.67

	Training accuracy	Test accuracy
Random set	83.24%	62.75%
Balanced set	93.66%	61.55%
All set	94.54%	78.48%

3.2 SIFT + SVM

Since this algorithm is computationally very slow, differently from HOG and CNN will be run only on one set. The version of SIFT used works only on colour .jpeg(s) so the first thing to do is converting our previously saved list of colour images:

Algorithm 6: Conversion in jpeg

Input : List of colour images
Initialize List of paths
Create a new folder
for $i = 1, ..., length(Listofcolourimages)$ **do**
    1) Create a new path for the image i that leads inside the new folder
    2) Write a .jpeg using the object in the list and the name in the new path
end

Again, the algorithm occasionally runs into some errors and stops, so a *tryCatch* is put in place before running it:

Algorithm 7: SIFT

Input : List of paths
Initialize List of SIFT descriptors
Create a new folder
for $i = 1, ..., length(Listofpaths)$ **do**
    1) Run SIFT function
    2) Store the output in the list
end

Also here, the *NULL* elements need to be removed with the same reasoning as in Algorithm 2.
Since each image has a different number of keypoints, the output cannot be passed directly to a SVM, so a feature reduction approach is needed.
The coordinates of the keypoints for each image will be passed to a *K-means* clustering algorithm and the centroids of the new clusters will be used as features for the SVM Many Ks are tried to find the best parameter, but since the number of clusters cannot be higher than the number of points, all images with number of keypoints smaller than K will be deleted, sometimes, as the algorithm gives errors, the parameter that deletes the images will need to be increased until the algorithm runs flawlessly:

Algorithm 8: Coordinates extraction

Input : List of SIFT descriptors
Initialize List of coordinates
for $i = 1, ..., length(ListofSIFTdescriptors)$ **do**
    Initialize Matrix of coordinates
    for $k = 1, ..., length(Objectinthelist)$ **do**
        1) Store X-axis coordinates in the matrix in position $[k, 1]$
        2) Store Y-axis coordinates in the matrix in position $[k, 2]$
    end
    1)Store matrix inside list
end

Where each matrix in the list represents an image, and its rows are the coordinates of each keypoint.

Now, the images with less than the desired K will be deleted, sometimes the number of images to be deleted has to be even higher than this to not incur into errors:

Algorithm 9: Deletion of images

Parameters : t = treshold of minimum number of points needed to not be eliminated
Input : List of coordinates, List of labels
Initialize $k = 1$, Vector of row indexes to be removed
for $i = 1, ..., length(Listofcoordinates)$ **do**
  if *number of rows* $< t$ **then**
    1) Store index i in the vector
    2) $k = k + 1$
  end
end
The List of coordinates and the List of labels becomes themselves without the elements in the new vector.

A K-means function is then run on each matrix in the list:

Algorithm 10: K-means

Parameters : number of clusters k, maximum number of iterations = 1.000.000.000
Input : List of coordinates
Initialize List of K-means descriptors
for $i = 1, ..., length(Listofcoordinates)$ **do**
  1) Apply K-means function on the matrix of coordinates
  2) Store the output inside the list
end

Finally the coordinates of the centroids of the clusters need to be extracted from the list of K-means descriptors and reshaped to be passed to an SVM, similarly to Algorithm 4:

Algorithm 11: Reshape of X

Input : List of K-means descriptors
Initialize Matrix of centroids coordinates
for $i = 1, ..., length(ListofKmeansdescriptors)$ **do**
  Initialize $k = 1$
  for $r = 1, ..., numberofcentroids$ **do**
    for $c = 1, 2$ **do**
      1) Store the coordinates of the centroids in position $[r, c]$ in the new matrix in position $[i, k]$
      2) $k = k + 1$
    end
  end
end

SVM is then run:

Algorithm 12: SVM

Input : Matrix of centroids coordinates, List of labels
Run SVM function
Predict and show Confusion matrix for training and test.

The algorithm is run many times, each time increasing k by 1 until the training accuracy starts decreasing, the highest one is then used for the testing.

The results are:

Training set cardinality	Test set cardinality	Rewriting training jpeg(s) time	Rewriting test jpeg(s) time	Training set SIFT time	Test set SIFT time
4000	1000	00:00:14.89	00:00:03.47	07:21:04.13	01:48:19.24

The cardinalities are before the *NULL* SIFT descriptors are removed from the training and test sets that remain respectively with 3634 and 916 images.

k	Minimum working threshold of keypoints	Training set lost images	Training set remaining images	Training set K-means time	Training SVM time	Training accuracy
5	10	296	3338	00:05:32.25	00:00:03.10	17.53%
6	15	640	2994	00:15:28.26	00:00:02.62	20.17%
7	15	640	2994	00:00:00.99	00:00:02.57	26.51%
8	17	801	2833	00:49:44.94	00:00:02.60	26.26%
9	18	986	2648	00:15:49.56	00:00:02.39	26.51%
10	24	1451	2183	00:39:37.61	00:00:01.69	32.07%
11	27	1729	1905	00:54:07.22	00:00:01.37	31.60%
12	27	1729	1905	00:22:19.94	00:00:01.41	34.33%
13	27	1729	1905	01:20:17.19	00:00:01.45	36.48%
14	29	1921	1713	01:40:14.44	00:00:01.21	39.93%
15	34	2326	1308	00:30:15.20	00:00:00.72	42.66%
16	34	2326	1308	01:20:13.68	00:00:00.73	42.66%
17	37	2547	1087	01:27:24.10	00:00:00.53	43.05%
18	37	2547	1087	01:08:29.18	00:00:00.54	48.85%
19	40	2721	913	01:32:36.73	00:00:00.40	45.13%

Each K-means function is run with a maximum number of iterations of 1.000.000.000.
The $k = 18$ is the one with the highest training accuracy, and will be the one used for testing:

k	Minimum working threshold of keypoints	Test set lost images	Test set remaining images	Test set K-means time	Training accuracy	Test accuracy
18	38	666	250	00:00:00.08	48.85%	09.20%

3.3 CNN

Before passing the set to the algorithm, it will be reshaped like already seen before, to have an image corresponding to each row, and each column representing the same feature for each image. Like the HOG algorithm, CNN will be run 3 times, for the same 3 datasets:

	Training set cardinality	Test set cardinality	Training CNN time	Training accuracy	Test accuracy
Random set	4000	1000	00:06:08.39	98.82%	85.94%
Balanced set	9681	2580	00:10:13.22	99.59%	73.93%
All set	39209	12630	00:29:45.01	99.25%	93.48%

Needs to be said that with early stopping, the CNN reaches a training accuracy between 98% and 99% around the 6th epoch, saving a lot of computational time for big datasets.

Chapter 4

Conclusions

	Training set cardinality	Test set cardinality	Parameters	Computational time	Training accuracy	Test accuracy
HOG with best accuracy	39209	12630	16 bins	00:09:38.98	94.54%	78.48%
HOG with best computational time	4000	1000	16 bins	00:00:11.11	83.24%	62.75%
SIFT with best accuracy	4000	1000	$k = 18$	10:18:11.53	48.85%	09.20%
CNN with best accuracy	39209	12639	50 epochs	00:29:45.01	99.25%	93.48%
CNN with best computational time	4000	1000	50 epochs	00:06:08.39	98.82%	85.94%

As predictable, the CNN is the algorithm that works better and has a higher accuracy than the others even when working with less data: a CNN with $\frac{1}{10}$ of datapoints performs better than a HOG in about $\frac{2}{3}$ of the time.
When working with the same amount of data, the HOG is much faster but loses ground on its precision.
As with all machine learning algorithm it is proved that the larger the dataset is, the more precise the model will be, obviously at a time cost; that is also why the SIFT performs so badly, because when increasing the number of clusters in the K-means approach, all the images with less than k keypoints are lost (with $k = 19$, only a quarter of the training set remains).
An "official" SIFT function in R could have improved its computational time.
It is worth noticing that none of these algorithm (with these parameters) could be used in a real-life contest, as their accuracy is always below the 98%, which is what one would at least expect when putting its life in the "hand" of a self-driving car; but it is true that the big automotive houses have datasets with billions of images and far more computational power than us so they are able to train a Convolutional Neural Network that can recognize road signs with almost perfect accuracy, the problem nowadays is to legislate on the matter and make the self-driving car legal, but there are still moral and sometimes technical issues.

Chapter 5

Appendix: Code

5.1 HOG

```
library(OpenImageR)
library(e1071)
library(caret)
library(png)
library(jpeg)

train = read.csv('Train.csv', stringsAsFactors = FALSE)} #Full

set.seed(42) #Run for sample
train = train[sample(1:nrow(train), 4000, replace = FALSE),]}

baltrain = data.frame() #Run for balanced set
h = min(table(train$ClassId))
for (i in 0:42){
  t = table(train$ClassId)[i + 1]
  z = ceiling(h / (t/30))
  for (k in seq(from = (30-z+1), to = (t-z+1), by = 30)){
    baltrain = rbind(baltrain, train[which(train$ClassId == i)[(k:(k+z-1))],])
  }
}
train = baltrain
table(train$ClassId)

#HOG
#Train
trpic = vector(mode = 'list', length = nrow(train))
trpicol = vector(mode = 'list', length = nrow(train))
errors = vector(mode = 'list', length=nrow(train))
for (i in 1:nrow(train)){
  tryCatch({
    path = file.path(getwd(), train$Path[i])
    pic = readImage(path)
    n_w = train$Roi.X1[i]:train$Roi.X2[i]
    n_h = train$Roi.Y1[i]:train$Roi.Y2[i]
    pic = cropImage(pic, new_width = n_w, new_height = n_h, type = 'user_defined')
    pic = resizeImage(pic, 50, 50, method = 'bilinear')
    pic_col = pic
    pic = rgb_2gray(pic)
    trpic[[i]] = pic
```

```
    trpicol[[i]] = pic_col
  }, error = function(e){
    errors[i] = i
  })
}

e = vector()
k = 1
for (i in 1:length(trpic)){
  if (is.null(trpic[[i]]) == TRUE){
    e[k] = i
    k = k + 1
  }
}
trpic = trpic[-(which(sapply(trpic, is.null), arr.ind = TRUE))]
trpicol = trpicol[-(which(sapply(trpicol, is.null), arr.ind = TRUE))]
#saveRDS(trpicol, file = 'trcol4000.rds') #Run on the random set
#saveRDS(trpicol, file = 'trcolbal.rds') #Run on the balanced set
#saveRDS(trpicol, file = 'trcolfull.rds') #Run on the full set
tridhog = factor(train$ClassId, levels = 0:42)
tridhog = tridhog[-e]
#saveRDS(tridhog, file = 'tridhog4000.rds') #Run on the random set
#saveRDS(tridhog, file = 'tridhogbal.rds') #Run on the balanced set
#saveRDS(tridhog, file = 'tridhogfull.rds') #Run on the full set

hogtrtime = c()
tmptrhog = vector(mode = 'list', length = length(trpic))
for (i in 1:length(trpic)){
  t1 = Sys.time()
  tmptrhog[[i]] = HOG(trpic[[i]], orientations = 16) #R HOG function does not have a 'pixels per c
  t2 = Sys.time()
  hogtrtime[i] = t2 - t1
}
sum(hogtrtime)

trhog = matrix(0, nrow = length(tmptrhog), ncol = length(tmptrhog[[1]]))
for (r in 1:length(tmptrhog)){
  for (c in 1:length(tmptrhog[[1]])){
    trhog[r,c] = tmptrhog[[r]][c]
  }
}

t1 = Sys.time() #RUN THESE 3 LINES TOGETHER
trmod = svm(trhog, tridhog)
t2 = Sys.time()

svmtime = t2 - t1

predSVM = predict(trmod)
confusionMatrix(predSVM, tridhog)

#Test
test = read.csv('Test.csv', stringsAsFactors = FALSE) #Full

set.seed(42) #Run for sample
test = test[sample(1:nrow(test), 1000, replace = FALSE),]
```

```
baltest = data.frame() #Run for balanced
h = min(table(test$ClassId))
for (i in 0:42){
  t = table(test$ClassId)[i + 1]
  baltest = rbind(baltest, test[which(test$ClassId == i)[(t-h+1):t],])
}
test = baltest

tspic = vector(mode = 'list', length = nrow(test))
tspicol = vector(mode = 'list', length = nrow(test))
errors = vector(mode = 'list', length = nrow(test))
for (i in 1:nrow(test)){
  tryCatch({
    path = file.path(getwd(), test$Path[i])
    pic = readImage(path)
    n_w = test$Roi.X1[i]:test$Roi.X2[i]
    n_h = test$Roi.Y1[i]:test$Roi.Y2[i]
    pic = cropImage(pic, new_width = n_w, new_height = n_h, type = 'user_defined')
    pic = resizeImage(pic, 50, 50, method = 'bilinear')
    pic_col = pic
    pic = rgb_2gray(pic)
    tspic[[i]] = pic
    tspicol[[i]] = pic_col
  }, error = function(e){
    errors[i] = i
  })
}

e = vector()
k = 1
for (i in 1:length(tspic)){
  if (is.null(tspic[[i]]) == TRUE){
    e[k] = i
    k = k + 1
  }
}
tspic = tspic[-(which(sapply(tspic, is.null), arr.ind = TRUE))]
tspicol = tspicol[-(which(sapply(tspicol, is.null), arr.ind = TRUE))]
#saveRDS(tspicol, file = 'tscol1000.rds') #Run on the random set
#saveRDS(tspicol, file = 'tscolbal.rds') #Run on the balanced set
#saveRDS(tspicol, file = 'tscolfull.rds') #Run on the full set
tsidhog = factor(test$ClassId, levels = 0:42)
tsidhog = tsidhog[-e]
#saveRDS(tsidhog, file = 'tsidhog1000.rds') #Run on the random set
#saveRDS(tsidhog, file = 'tsidhogbal.rds') #Run on the balanced set
#saveRDS(tsidhog, file = 'tsidhogfull.rds') #Run on the full set

hogtstime = c()
tmptshog = vector(mode = 'list', length = length(tspic))
for (i in 1:length(tspic)){
  t1 = Sys.time()
  tmptshog[[i]] = HOG(tspic[[i]], orientations = 16)
  t2 = Sys.time()
  hogtstime[i] = t2 - t1
}
```

```
sum(hogtstime)

tshog = matrix(0, nrow = length(tmptshog), ncol = length(tmptshog[[1]]))
for (r in 1:length(tmptshog)){
  for (c in 1:length(tmptshog[[1]])){
    tshog[r,c] = tmptshog[[r]][c]
  }
}

predSVM2 = predict(trmod, tshog)
confusionMatrix(predSVM2, tsidhog)
```

5.2 SIFT

```
#Train
img = readRDS('trcol4000.rds')
dirtime = c()
trpath = vector(mode = 'list', length = length(img))
dir.create('trjpegs')
for (i in 1:length(img)){
    t1 = Sys.time()
    trpath[[i]] = file.path(getwd(), sprintf('trjpegs/img%d.jpeg', i))
    writeJPEG(img[[i]], target = trpath[[i]], quality = 1)
    t2 = Sys.time()
    dirtime[i] = t2 - t1
}
sum(dirtime)

trsift = vector(mode = 'list', length = length(trpath))
siftime = c()
errors = vector(mode = 'list', length = length(trsift))
for (i in 1:length(trsift)){
    t1 = Sys.time()
    tryCatch({
        trsift[[i]] = SIFT(trpath[[i]])
    },
    error = function(e){
        errors[i] = i
    }
    )
    t2 = Sys.time()
    siftime[i] = t2 - t1
}
sum(siftime)

e = vector()
k = 1
for (i in 1:length(trsift)){
    if (is.null(trsift[[i]]) == TRUE){
        e[k] = i
        k = k + 1
    }
}
tridsift = readRDS('tridhog4000.rds')
tridsift = tridsift[-e]
```

```
trsift = trsift[-(which(sapply(trsift, is.null), arr.ind = TRUE))]
saveRDS(trsift, file = 'cleantrsift.rds')
saveRDS(tridsift, file = 'cleantridsift.rds')

trsift = readRDS('cleantrsift.rds') #Run from here each time the parameters for k are changed
tridsift = readRDS('cleantridsift.rds')

coor = vector(mode = 'list', length = length(trsift))
for (i in 1:length(trsift)){
    x = matrix(NA, ncol = 2, nrow = length(trsift[[i]]))
    for (k in 1:length(trsift[[i]])){
        x[k,1] = trsift[[i]][[k]]$Peak$col
        x[k,2] = trsift[[i]][[k]]$Peak$row
    }
    coor[[i]] = x
}

k = 1
e = vector()
for (i in 1:length(coor)){
    if (nrow(coor[[i]]) < 37){ #delete images with less keypoints, changes for each value of k
        e[k] = i
        k = k + 1
    }
}
coor = coor[-e]
tridsift = tridsift[-e]

set.seed(42)
kms = vector(mode = 'list', length = length(coor))
ktime = c()
for (i in 1:length(coor)){
    t1 = Sys.time()
    kms[[i]] = kmeans(coor[[i]], centers = 18, iter.max = 1000000000) #change value of k
    t2 = Sys.time()
    ktime[i] = t2 - t1
}
sum(ktime)

x_train = matrix(NA, nrow = length(kms), ncol = nrow(kms[[1]]$centers)*2)
for (i in 1:length(kms)){
    k = 1
    for (r in 1:nrow(kms[[i]]$centers)){
        for (c in 1:2){
            x_train[i, k] = kms[[i]]$centers[r, c]
            k = k + 1
        }
    }
}

y_train = tridsift

t1 = Sys.time() #Run these 3 lines together
trmod = svm(x_train, y_train)
t2 = Sys.time()
```

```
svmtime = t2 - t1

predSVM = predict(trmod)
confusionMatrix(predSVM, y_train)

saveRDS(trmod, file = 'trainmodel.rds') #Run every time the tr acc is higher than the previous one
#Stop once the training accuracy start decreasing

#Test
img = readRDS('tscol1000.rds')
dirtime = c()
tspath = vector(mode = 'list', length = length(img))
dir.create('tsjpegs')
for (i in 1:length(img)){
    t1 = Sys.time()
    tspath[[i]] = file.path(getwd(), sprintf('tsjpegs/img%d.jpeg', i))
    writeJPEG(img[[i]], target = tspath[[i]], quality = 1)
    t2 = Sys.time()
    dirtime[i] = t2 - t1
}
sum(dirtime)

tssift = vector(mode = 'list', length = length(tspath))
siftime = c()
errors = vector(mode = 'list', length = length(tssift))
for (i in 1:length(tssift)){
    t1 = Sys.time()
    tryCatch({
        tssift[[i]] = SIFT(tspath[[i]])
    },
    error = function(e){
        errors[i] = i
    }
    )
    t2 = Sys.time()
    siftime[i] = t2 - t1
}
sum(siftime)

e = vector()
k = 1
for (i in 1:length(tssift)){
    if (is.null(tssift[[i]]) == TRUE){
        e[k] = i
        k = k + 1
    }
}
tsidsift = readRDS('tsidhog1000.rds')
tsidsift = tsidsift[-e]
tssift = tssift[-(which(sapply(tssift, is.null), arr.ind = TRUE))]
saveRDS(tssift, file = 'cleantssift.rds')
saveRDS(tsidsift, file = 'cleantsidsift.rds')

coor = vector(mode = 'list', length = length(tssift))
for (i in 1:length(tssift)){
  x = matrix(NA, ncol = 2, nrow = length(tssift[[i]]))
```

```
  for (k in 1:length(tssift[[i]])){
    x[k,1] = tssift[[i]][[k]]$Peak$col
    x[k,2] = tssift[[i]][[k]]$Peak$row
  }
  coor[[i]] = x
}

k = 1
e = vector()
for (i in 1:length(coor)){
  if (nrow(coor[[i]]) < 38){
    e[k] = i
    k = k + 1
  }
}
coor = coor[-e]
tsidsift = tsidsift[-e]

set.seed(42)
kms = vector(mode = 'list', length = length(coor))
ktime = c()
for (i in 1:length(coor)){
  t1 = Sys.time()
  kms[[i]] = kmeans(coor[[i]], centers = 18, iter.max = 1000000000)
  t2 = Sys.time()
  ktime[i] = t2 - t1
}
sum(ktime)

x_test = matrix(NA, nrow = length(kms), ncol = nrow(kms[[1]]$centers)*2)
for (i in 1:length(kms)){
  k = 1
  for (r in 1:nrow(kms[[i]]$centers)){
    for (c in 1:2){
      x_test[i, k] = kms[[i]]$centers[r, c]
      k = k + 1
    }
  }
}

y_test = tsidsift

trmod = readRDS('trainmodel.rds')
predSVM2 = predict(trmod, x_test)
confusionMatrix(predSVM2, y_test)
```

5.2.1 SIFT function

```
#R does not have a SIFT function, this implementation is provided by Huskinson on GitHub
#It can be found here: https://github.com/huksu/RSIFT

require(jpeg)
gaussian2D <- function(x, y, sigma){
    return((1/(2*pi*sigma*sigma))*(exp(-(x*x + y*y)/(2*sigma*sigma))))
}
```

```
gaussian2DNonNorm <- function(x, y, sigma){
    # Normalization is not necessary now if it happens later
    return(exp(-(x*x + y*y)/(2*sigma*sigma)))
}

gaussian1DNonNorm <- function(x, sigma){
    # Normalization is not necessary now if it happens later
    return(exp(-(x*x)/(2*sigma*sigma)))
}

gaussianFilter2D <- function(sigma, radius = 0){
    radius <- ifelse(radius == 0, as.integer(sigma*3)+1, radius)
    filter <- matrix(data = 0, nrow = 2*radius-1, ncol = 2*radius-1)
    filter <- outer(1:nrow(filter),1:ncol(filter), FUN = Vectorize(function(r,c) gaussian2DNonNorm
    return(filter)
}

gaussianFilter1D <- function(sigma, radius = 0){
    radius <- ifelse(radius == 0, as.integer(sigma*3)+1, radius)
    filter <- c(-(radius-1):(radius-1))
    filter <- unlist(lapply(filter,FUN = function(x) gaussian1DNonNorm(x,radius)))
    return(filter)
}

simpleEdgeFilter2D <- function(){
    return(matrix(c(-1,-1,-1,-1,8,-1,-1,-1,-1), ncol = 3))
}

applyGaussFilter1D <- function(img, f2d){

    radius <- as.integer(nrow(f2d)/2)+1
    print(paste("Applying filter with radius:", radius))
    f <- f2d[,radius]

    # Col-wise
    newimg1 <- img
    newimg1 <- outer(1:nrow(newimg1),1:ncol(newimg1), FUN = Vectorize(function(i,j){
        imgcolsub <- j + c(1:length(f)) - radius
        imgcolsub <- ifelse(imgcolsub<1,1,imgcolsub)
        imgcolsub <- ifelse(imgcolsub>ncol(img),ncol(img),imgcolsub)
        return(sum(img[i,imgcolsub]*f))
    }))

    # Row-wise
    newimg2 <- newimg1
    newimg2 <- outer(1:nrow(newimg2),1:ncol(newimg2), FUN = Vectorize(function(i,j){
        imgrowsub <- i + c(1:length(f)) - radius
        imgrowsub <- ifelse(imgrowsub<1,1,imgrowsub)
        imgrowsub <- ifelse(imgrowsub>nrow(img),nrow(img),imgrowsub)
        return(sum(newimg1[imgrowsub,j]*f))
    }))

    return(imageNorm(newimg2))
}

applyGaussFilter2D <- function(img, f){
```

```
    # 2D filter leaves ugly borders on the shading.
    # Best to use 1D 2 pass filter.
    newimg <- img
    radius <- as.integer(nrow(f)/2)+1
    print(paste("Applying filter with radius:", radius))
    for(i in c(1:nrow(img))){
        for(j in c(1:ncol(img))){
            f_sub <- f[c(max(1,radius-i+1):min(nrow(f),nrow(img)-i+radius)),c(max(1,radius-j+1):mi
            img_sub <- img[c(max(1,i-radius+1):min(nrow(img),i+radius-1)),c(max(1,j-radius+1):min(
            accumulation <- sum(as.vector(f_sub)*as.vector(img_sub))
            newimg[i,j] <- accumulation
        }
    }
    return(imageNorm(newimg))
}

imageNorm <- function(img){
    img <- img-min(img)
    img <- img/max(img)
    return(img)
}

imagePixClip <- function(img){
    img <- ifelse(img < 0, 0, img)
    img <- ifelse(img > 1, 1, img)
}

drawImg <- function(img){
    plot(c(0,1),c(0,1),t='n')
    rasterImage(img, 0,0,1,1)
}

grayImage <- function(colorImg){
    grayImg <- colorImg[,,1]+colorImg[,,2]+colorImg[,,3] # reduce to gray
    grayImg <- imageNorm(grayImg)
    return(grayImg)
}

downscaleImg <- function(img, level=2){
    halfheight <- floor(nrow(img)/level)
    halfwidth <- floor(ncol(img)/level)
    newimg <- matrix(0,nrow=halfheight,ncol=halfwidth)
    for(i in c(1:halfheight)){
        for(j in c(1:halfwidth)){
            newimg[i,j] <- img[(i-1)*level+1,(j-1)*level+1]
        }
    }
    return(newimg)
}

getK <- function(numScaleLevels){
    k <- 2^(1.0 / numScaleLevels)
}

getSigmas <- function(numScaleLevels, sd){
    sigmas <- rep(0,times=numScaleLevels)
```

```
    k <- getK(numScaleLevels)
    sigmas[1] <- sd
    sigmas[2] <- sd*sqrt((k^2)-1)
    for(i in c(3:(numScaleLevels+3))){
        sigmas[i] <- sigmas[i-1]*k
    }

    return(sigmas)
}

laplacianPyramid <- function(img, numOctaves = 4, numScaleLevels = 5, k = sqrt(2), sd = 1.6){

    sigmas <- getSigmas(numScaleLevels,sd)
    print("Sigma Values")
    print(sigmas)
    rootImg <- img

    lp <- list()

    for(i in c(1:numOctaves)){
        print(paste("Octave: ",i))
        octave <- array(rep(c(1:(numScaleLevels+3)), each = nrow(rootImg)*ncol(rootImg)), c(nrow(r
        print(dim(octave))

        print(paste("Octave:", i, "Scale Level:", 1))
        octave[,,1] <- rootImg
        currentImg <- rootImg
        #drawImg(imageNorm(currentImg))

        for(j in c(2:(numScaleLevels+3))){
            print(paste("Octave:", i, "Scale Level:", j))
            f <- gaussianFilter2D(sigmas[j])
            nextImg <- applyGaussFilter1D(currentImg,f)
            #drawImg(imageNorm(nextImg))
            octave[,,j] <- nextImg
            currentImg <- nextImg
        }

        lp[[length(lp)+1]] <- octave

        # prep next loop
        rootImg <- downscaleImg(rootImg)
    }

    return(lp)
}

differenceOfGaussianPyramid <- function(lp, numOctaves, numScaleLevels){
    # The Difference-of-Gaussians pyramid is a list of 3d arrays
    # Each 3d array is a 2-D DoG image at different scale levels
    # Each 3d array in the list is the octave

    D <- list()
    for(i in c(1:numOctaves)){
        print(paste("Octave: ",i))
        rootImg <- lp[[i]][,,1]
```

```
        octave <- array(rep(c(1:(numScaleLevels+2)), each = nrow(rootImg)*ncol(rootImg)), c(nrow(r
        print(dim(octave[,,]))

        for(j in c(1:(numScaleLevels+2))){
            print(paste("Octave:", i, "Scale Level:", j))
            diffImg <- imageNorm(lp[[i]][,,j+1] - lp[[i]][,,j])
            octave[,,j] <- imageNorm(diffImg)
        }

        D[[length(D)+1]] <- octave
    }

    return(D)
}

scaleSpacePeakDetection <- function(D, numOctaves, numScaleLevels, sd){
    peaks <- data.frame()
    neighbors <- do.call(expand.grid,list(x = -1:1, y = -1:1, z = -1:1))
    neighbors <- subset(neighbors, x != 0 | y != 0 | z != 0)
    for(o in c(1:numOctaves)){
        for(s in c(2:(numScaleLevels+1))){
            # for every scale (first and last notwithstanding), look for the extrema points
            print(paste("Octave:",o,"Scale:",s))
            for(r in c(2:(nrow(D[[o]][,,s])-1))){
                if(r%%10==0){
                    print(paste("Octave:",o,"Scale:",s,"Row:",r))
                }
                for(c in c(2:(ncol(D[[o]][,,s])-1))){
                    #print(paste("Col:",c))
                    # check every pixel's neighbors to see if this pixel is the maximum or minimum
                    greaterThanFail <- FALSE
                    lessThanFail <- FALSE
                    k <- 1
                    pixelValue <- D[[o]][r,c,s]
                    while((greaterThanFail == FALSE | lessThanFail == FALSE) & (k <= nrow(neighbor
                        neighborValue <- D[[o]][r+neighbors[k,]$x,c+neighbors[k,]$y,s+neighbors[k,
                        #print(paste("neighborValue:",neighborValue, "x:",r+neighbors[k,]$x, "x:",
                        greaterThanFail <- ifelse(pixelValue <= neighborValue,TRUE,greaterThanFail
                        lessThanFail <- ifelse(pixelValue >= neighborValue,TRUE,greaterThanFail)
                        k <- k + 1
                    }

                    if(greaterThanFail == FALSE | lessThanFail == FALSE){
                        # This point is an extrema
                        peaks <- rbind(peaks,data.frame(row = r, col = c, octave = o, interval = s
                        print(paste("Extrema at r =", r, "c =",c))
                    }
                }
            }
        }
    }

    peaks$trow <- transformRCOPoint(peaks$row,peaks$octave)
    peaks$tcol <- transformRCOPoint(peaks$col,peaks$octave)
    peaks$scale <- peakPointScale(peaks$octave,peaks$interval,sd,numScaleLevels)
    peaks$octaveScale <- peakPointOctaveScale(peaks$interval,sd,numScaleLevels)
```

```
    return(peaks)
}

transformRCOPoint <- function(x,o){
    # we start with baselen at octave 1
    # when we downscale, we chew off pixels at the tail end
    # so we can simply multiply x without concern for middle points
    # 1 -> 1 -> 1
    # 2 -> 3 -> 5
    # 3 -> 5 -> 9
    # 4 -> 7 -> 13
    # 5 -> 9 -> 17
    # 6 -> 11 -> 21
    #... so on
    return((x*2^(o-1)) - (2^(o-1)) + 1)
}

rgbGrayImg <- function(grayImg){
    h <- dim(grayImg)[1]
    w <- dim(grayImg)[2]
    rgbGray <- array(0, dim = c(h,w,3))
    rgbGray[,,1] <- grayImg
    rgbGray[,,2] <- grayImg
    rgbGray[,,3] <- grayImg
    return(rgbGray)
}

setPixel <- function(img,x,y,r,g,b){
    if(x <= nrow(img) & y <= ncol(img) & x > 0 & y > 0){
        img[x,y,1] <- r
        img[x,y,2] <- g
        img[x,y,3] <- b
    }
    return(img)
}

setRGBGrayPeak <- function(rgbGray,r,c,o, drawCircles, radiusMult = 5){
    radius <- o*radiusMult

    # draw midpoint
    rgbGray <- setPixel(rgbGray,r,c,1,0,0)

    if(drawCircles){
        # draw circle
        for(x in c(-radius:radius)){
            y <- round(sqrt((radius * radius) - (x * x)))
            rgbGray <- setPixel(rgbGray,r+x,c+y,1,0,0)
            rgbGray <- setPixel(rgbGray,r+x,c-y,1,0,0)
        }
        for(y in c(-radius:radius)){
            x <- round(sqrt((radius * radius) - (y * y)))
            rgbGray <- setPixel(rgbGray,r+x,c+y,1,0,0)
            rgbGray <- setPixel(rgbGray,r-x,c+y,1,0,0)
        }
    }
```

```
    return(rgbGray)
}

drawPeaks <- function(grayImg,peaks, drawCircles = TRUE){
    rgbGray <- rgbGrayImg(grayImg)
    for(i in c(1:nrow(peaks))){
        rgbGray <- setRGBGrayPeak(rgbGray,peaks[i,]$trow,peaks[i,]$tcol,peaks[i,]$octave,drawCircl
    }
    drawImg(rgbGray)
}


getRCSOffset <- function(D,r,c,s,o){
    # Get the derivative of D about r,c,s,o
    dx <- (D[[o]][r,c+1,s] - D[[o]][r,c-1,s]) / 2.0
    dy <- (D[[o]][r+1,c,s] - D[[o]][r-1,c,s]) / 2.0
    ds <- (D[[o]][r,c,s+1] - D[[o]][r,c,s-1]) / 2.0
    d <- matrix(c(dx,dy,ds),ncol=3)

    # Get the Hessian of D
    dxx <- D[[o]][r,c+1,s] + D[[o]][r,c-1,s] - 2 * D[[o]][r,c,s]
    dyy <- D[[o]][r+1,c,s] + D[[o]][r-1,c,s] - 2 * D[[o]][r,c,s]
    dss <- D[[o]][r,c,s+1] + D[[o]][r,c,s-1] - 2 * D[[o]][r,c,s]
    dxy <- (D[[o]][r+1,c+1,s] - D[[o]][r+1,c-1,s] - D[[o]][r-1,c+1,s] + D[[o]][r-1,c-1,s]) / 4.0
    dxs <- (D[[o]][r,c+1,s+1] - D[[o]][r,c-1,s+1] - D[[o]][r,c+1,s-1] + D[[o]][r,c-1,s-1]) / 4.0
    dys <- (D[[o]][r+1,c,s+1] - D[[o]][r-1,c,s+1] - D[[o]][r+1,c,s-1] + D[[o]][r-1,c,s-1]) / 4.0
    H <- matrix(c(dxx,dxy,dxs,dxy,dyy,dys,dxs,dys,dss), nrow=3, ncol=3)

    Hdet <- det(H)
    if(Hdet == 0){
        print("H determinant is 0.  Failing...")
        return(FALSE)
    }

    Hinv <- solve(H)

    rcsOffset <- round(as.vector(-Hinv %*% t(d)))
    names(rcsOffset)<-c("r","c","s")
    return(rcsOffset)
}

getContrast <- function(D,r,c,s,o,dr,dc,ds){
    # Get the derivative of D about r,c,s,o
    dx <- (D[[o]][r,c+1,s] - D[[o]][r,c-1,s]) / 2.0
    dy <- (D[[o]][r+1,c,s] - D[[o]][r-1,c,s]) / 2.0
    ds <- (D[[o]][r,c,s+1] - D[[o]][r,c,s-1]) / 2.0
    d <- matrix(c(dx,dy,ds),ncol=3)

    # Matrix form of xhat
    xhat <- matrix(c(dc,dr,dc), nrow=3)

    contrast <- D[[o]][r,c,s] + (d %*% xhat) * .5
    return(contrast)
}
```

```
peakLocalization <- function(peaks, D, numOctaves, numScaleLevels, sd, MAXSTEPS = 10, THRESHOLD =
    # This is the part where most people get stuck on SIFT, so some explanation of what is happeni
    # We are going to wander around, starting from the current peak,
    # to find the point where the gradient of the pixel passes zero.
    # Imagine the pixel/image is represented by a function.  The gradient is the rate of
    # change of that function.  Thinking of calculus, the point where the gradient
    # passes zero is the maximum or minimum of the original function.
    # Therefore, the derivative of the taylor series of D at 0 is used to approximate the gradient
    # This approximation is the offset of the current pixel, because we are measuring the gradient
    # the current pixel location.
    # We adjust x by the offset and repeat until the offset is not big enough to move the pixel ar
    # We also throw away localized peaks below a threshold.

    localizedPeaks <- data.frame()

    for(p in c(1:nrow(peaks))){
        print(paste("Peak: ",p))
        i <- 1
        success <- FALSE
        h <- dim(D[[peaks[p,]$octave]][,,peaks[p,]$interval])[1]
        w <- dim(D[[peaks[p,]$octave]][,,peaks[p,]$interval])[2]
        while(i <= MAXSTEPS){
            rcsOffset <- getRCSOffset(D,peaks[p,]$row,peaks[p,]$col,peaks[p,]$interval,peaks[p,]$o

            if(is.logical(rcsOffset)){
                print("RCSOffset Failed")
                break
            }

            if(sum(abs(rcsOffset)) == 0){
                # We didn't move, so finish
                print(paste("No movement at step:",i))
                success <- TRUE
                break
            }

            # Apply the offset
            peaks[p,]$row <- peaks[p,]$row + rcsOffset["r"]
            peaks[p,]$col <- peaks[p,]$col + rcsOffset["c"]
            peaks[p,]$interval <- peaks[p,]$interval + rcsOffset["s"]

            if(peaks[p,]$interval <= 1 | peaks[p,]$interval >= (numScaleLevels+2) |
                   peaks[p,]$row <= 1 | peaks[p,]$row >= h |
                   peaks[p,]$col <= 1 | peaks[p,]$col >= w ){
                # we went out of bounds
                print("Out of bounds...")
                break
            }

            i <- i + 1
        }

        if(i>MAXSTEPS){
            print("Too many steps...")
        }
```

```
        if(success == FALSE){
            # remove the point from the peak set
            print(paste("Fail on peak: ",p))
        } else {
            print(paste("Success on peak: ",p))
            contrast <- getContrast(D,peaks[p,]$row,peaks[p,]$col,peaks[p,]$interval,peaks[p,]$oct
            if(1){ #TODO
                print(paste("Peak passed threshold test.  Contrast:",contrast))
                localizedPeaks <- rbind(localizedPeaks,peaks[p,])
            }
            else{
                print(paste("Peak failed threshold test.  Contrast:",contrast))
            }

        }
    }

    localizedPeaks$trow <- transformRCOPoint(localizedPeaks$row,localizedPeaks$octave)
    localizedPeaks$tcol <- transformRCOPoint(localizedPeaks$col,localizedPeaks$octave)
    localizedPeaks$scale <- peakPointScale(localizedPeaks$octave,localizedPeaks$interval,sd,numSca
    localizedPeaks$octaveScale <- peakPointOctaveScale(localizedPeaks$interval,sd,numScaleLevels)
    return(localizedPeaks)
}

peakPointScale <- function(octave,interval,sd,numScaleLevels){
    return(sd * 2^(octave + interval/numScaleLevels))
}

peakPointOctaveScale <- function(interval,sd,numScaleLevels){
    return(sd * 2^(interval/numScaleLevels))
}

getStablePeaks <- function(peaks, D, numOctaves, numScaleLevels, radius = 10){
    stablePeaks <- data.frame()
    for(p in c(1:nrow(peaks))){
        print(paste("Peak: ",p))
        o <- peaks[p,]$octave
        r <- peaks[p,]$row
        c <- peaks[p,]$col
        s <- peaks[p,]$interval
        print(paste(o,r,c,s))

        dxx <- D[[o]][r,c+1,s] + D[[o]][r,c-1,s] - 2 * D[[o]][r,c,s]
        dyy <- D[[o]][r+1,c,s] + D[[o]][r-1,c,s] - 2 * D[[o]][r,c,s]
        dxy <- (D[[o]][r+1,c+1,s] - D[[o]][r+1,c-1,s] - D[[o]][r-1,c+1,s] + D[[o]][r-1,c-1,s]) / 4
        #H <- matrix(c(dxx,dxy,dxy,dyy), nrow=2, ncol=2)
        TraceH <- dxx + dyy
        DetH <- dxx*dyy - dxy^2
        test <- (TraceH^2)/DetH
        expected <- ((radius+1)^2)/radius
        print(paste("Test: ",test,"Expected:",expected))
        if(DetH <= 0){
            print("Peak fail edge test. Determinant <= 0.")
        }
        else if(test <=  expected){
            print("Peak passed edge test.")
```

```
            stablePeaks<-rbind(stablePeaks,peaks[p,])
        }
        else {
            print("Peak failed edge test.  test > expected.")
        }
    }
    return(stablePeaks)
}

assignOrientationAndMagnitude <- function(lp, peaks, sd, NUMBINS = 36, RADIUSFACTOR = 3, MAGTHRESH

    radiusLevel <- sd * RADIUSFACTOR
    orientedPeaks <- data.frame()
    peaks$orientation <- NA
    peaks$magnitude <- NA

    for(p in c(1:nrow(peaks))){
        print(paste("Orienting Peak: ",p))
        o <- peaks[p,]$octave
        r <- peaks[p,]$row
        c <- peaks[p,]$col
        s <- peaks[p,]$interval
        scale <- peaks[p,]$scale
        octaveScale <- peaks[p,]$octaveScale
        radius <- round(radiusLevel * octaveScale)
        print(paste("Pixel Radius:",radius))
        sigma <- RADIUSFACTOR * octaveScale

        # get the relevant picture from the gaussian pyramid
        img <- lp[[o]][,,s]
        h <- dim(img)[1]
        w <- dim(img)[2]

        hist <- rep(0,times = NUMBINS)

        for(i in c((-radius):radius)){
            for(j in c((-radius):radius)){
                # For each one of the surrounding points, look for the orientation and magnitude..
                # If you can calculate that point, then add it to the histogram...
                ri <- r + i
                ci <- c + i
                if(ri > 1 & ri < h & ci > 1 & ci < w){
                    # Calculate...
                    dx <- lp[[o]][ri,ci+1,s] - lp[[o]][ri,ci-1,s]
                    dy <- lp[[o]][ri-1,ci,s] - lp[[o]][ri+1,ci,s]
                    mag <- sqrt(dx*dx + dy*dy)
                    orient <- atan2(dy,dx)

                    # Add to bin...
                    offsetDragFactor <- exp((-(i*i+j*j))/(2.0*sigma*sigma))
                    bin <- round(((orient + pi) / (2*pi)) * NUMBINS) + 1 # scale -PI,PI to 0,1
                    bin <- ifelse(bin > NUMBINS, 1, bin) # in case of 360 degrees
                    hist[bin] <- hist[bin] + mag * offsetDragFactor
                }
            }
        }
```

```
        # Find the max magnitude for each histogram and see if any of the other magnitudes
        # are above a threshold.  If they are, duplicate the peak.  Then set the orientations.
        maxVal <- 0
        for(bin in c(1:NUMBINS)){
            if(hist[bin] >= maxVal){
                maxVal <- hist[bin]
            }
        }

        print(paste("Max Magnitude:",maxVal))

        for(bin in c(1:NUMBINS)){
            if(hist[bin]/maxVal >= MAGTHRESHOLD){
                # Add the peak
                orientedPeaks <- rbind(orientedPeaks,peaks[p,])
                orientedPeaks[nrow(orientedPeaks),]$orientation <- ((2*pi*(bin-1))/NUMBINS) - pi
                orientedPeaks[nrow(orientedPeaks),]$magnitude <- hist[bin]
            }
        }
    }

    return(orientedPeaks)
}

generateSIFTDescriptors <- function(lp, orientedPeaks){
    SIFTDescriptors <- list()
    #for(p in c(1:nrow(orientedPeaks))){
    for(p in c(1:nrow(orientedPeaks))){

        print(paste("Describing Peak:",p))
        SUBREGIONW <- 4
        BINSPERSUBREGION <- 8
        RADIUSFACTOR <- 3

        o <- orientedPeaks[p,]$octave
        r <- orientedPeaks[p,]$row
        c <- orientedPeaks[p,]$col
        s <- orientedPeaks[p,]$interval
        scale <- orientedPeaks[p,]$scale
        octaveScale <- orientedPeaks[p,]$octaveScale
        sigma <- RADIUSFACTOR * octaveScale

        img <- lp[[o]][,,s]
        h <- dim(img)[1]
        w <- dim(img)[2]


        hist <-  array(0, dim = c(SUBREGIONW,SUBREGIONW,BINSPERSUBREGION))
        binSizeTheta <- BINSPERSUBREGION/(2*pi) # should be 25 degrees

        for(dr in c(-8:-1,1:8)){ # rows
            for(dc in c(-8:-1,1:8)){ # cols
                neighborr <- r + dr
                neighborc <- r + dc
                if(neighborr > 1 & neighborr < h & neighborc > 1 & neighborc < w){
```

```
            # we can investigate this pixel

            if(neighborr < 0){
                subregionr <- floor(dr/SUBREGIONW)+3
            }
            if(neighborr > 0){
                subregionr <- ceiling(dr/SUBREGIONW)+2
            }
            if(neighborc < 0){
                subregionc <- floor(dc/SUBREGIONW)+3
            }
            if(neighborc > 0){
                subregionc <- ceiling(dc/SUBREGIONW)+2
            }

            neighborr <- r + dr
            neighborc <- r + dc

            # get the pixel orientation
            dx <- lp[[o]][neighborr,neighborc+1,s] - lp[[o]][neighborr,neighborc-1,s]
            dy <- lp[[o]][neighborr-1,neighborc,s] - lp[[o]][neighborr+1,neighborc,s]
            mag <- sqrt(dx*dx + dy*dy)
            orient <- atan2(dy,dx)

            # reorient the pixel to the feature orientation
            orient <- orient + orientedPeaks[p,]$orientation
            #print(paste("Orientation:",orient,orientedPeaks[p,]$orientation))
            if(orient >= pi){
                orient <- orient - 2*pi
            }

            # Add to bin...
            offsetDragFactor <- exp((-(dr*dr+dc*dc))/(2.0*sigma*sigma))
            bin <- round(((orient + pi) / (2*pi)) * BINSPERSUBREGION) + 1 # scale -PI,PI t
            bin <- ifelse(bin > BINSPERSUBREGION, 1, bin) # in case of 360 degrees
            #print(paste("Bin:",bin, "subregionr:",subregionr,"subregionc:",subregionc))
            #print(paste(mag,orient,bin,offsetDragFactor))
            hist[subregionr,subregionc,bin] <- mag * offsetDragFactor


        }
    }
}

# Reorganize the bin weights into a single array
descriptor <- list()
for(x in c(1:SUBREGIONW)){
    for(y in c(1:SUBREGIONW)){
        for(z in c(1:BINSPERSUBREGION)){
            descriptor[[length(descriptor)+1]] <- hist[x,y,z]
        }
    }
}
descriptor <- unlist(descriptor)
maxmag <- max(descriptor)
if(maxmag > 0){
```

```
            descriptor <- descriptor/maxmag # normalize to unit vector
        }
        SIFTDescriptors[[length(SIFTDescriptors)+1]] <- descriptor
    }

    return(SIFTDescriptors)
}

siftDataBuilder <- function(siftDescriptors, orientedPeaks){
    siftData <- list()
    for(p in c(1:nrow(orientedPeaks))){
        print(paste("Sift Point"))
        if(sum(siftDescriptors[[p]]) > 0){
            siftPoint <- list()
            siftPoint[[1]] <- orientedPeaks[p,]
            siftPoint[[2]] <- siftDescriptors[[p]]
            names(siftPoint) <- c("Peak","Descriptor")
            siftData[[length(siftData)+1]] <- siftPoint
        }
        else{
            print("Lack of magnitude in descriptor.")
        }
    }
    return(siftData)
}

SIFT <- function(imgFileName, numOctaves = 4, numScaleLevels = 5, sd = 1.6){
    ## Load the image
    myjpg <- readJPEG(imgFileName)
    grayImg <- grayImage(myjpg)

    ## Build the Difference-of-Gaussians pyramid
    lp <- laplacianPyramid(grayImg, numOctaves, numScaleLevels, sd)
    D <- differenceOfGaussianPyramid(lp, numOctaves, numScaleLevels)

    ## Scale space peak detection
    peaks <- scaleSpacePeakDetection(D, numOctaves, numScaleLevels, sd)

    ## Accurate keypoint localization
    localizedPeaks <- peakLocalization(peaks, D, numOctaves, numScaleLevels, sd, MAXSTEPS = 10)

    ## Eliminating Edge Responses
    stablePeaks <- getStablePeaks(localizedPeaks, D, numOctaves, numScaleLevels)

    ## Peak Orientation Assignment
    orientedPeaks <- assignOrientationAndMagnitude(lp, stablePeaks, sd)

    ## Generate the SIFT Descriptors (PHEW)
    drawPeaks(grayImg,orientedPeaks, drawCircles = FALSE)
    siftDescriptors <- generateSIFTDescriptors(lp, orientedPeaks)

    ## Combine the data into one set
    SIFTData <- siftDataBuilder(siftDescriptors,orientedPeaks)

    return(SIFTData)
}
```

5.3 CNN

```
library(dplyr)
library(keras)
library(tensorflow)

trpicol = readRDS('trcolfull.rds')  #Run for full dataset
tspicol = readRDS('tscolfull.rds')

trpicol = readRDS('trcolbal.rds') #Run for balanced dataset
tspicol = readRDS('tscolbal.rds')

trpicol = readRDS('trcol4000.rds') #Run for random dataset
tspicol = readRDS('tscol1000.rds')

w = dim(trpicol[[1]])[1] * dim(trpicol[[1]])[2] * dim(trpicol[[1]])[3]
x_train = matrix(0, nrow = length(trpicol), ncol = w)
x_test = matrix(0, nrow = length(tspicol), ncol = w)

z = 0
for (i in 1:length(trpicol)){
  z = z + 1
  x = 1
  for (s in 1:dim(trpicol[[i]])[3]){
    for (r in 1:dim(trpicol[[i]])[1]){
      for (c in 1:dim(trpicol[[i]])[2]){
        x_train[z, x] = trpicol[[i]][r, c, s]
        x = x + 1
      }
    }
  }
}

z = 0
for (i in 1:length(tspicol)){
  z = z + 1
  x = 1
  for (s in 1:dim(tspicol[[i]])[3]){
    for (r in 1:dim(tspicol[[i]])[1]){
      for (c in 1:dim(tspicol[[i]])[2]){
        x_test[z, x] = tspicol[[i]][r, c, s]
        x = x + 1
      }
    }
  }
}

rm(trpicol, tspicol)
gc()

x_train = array(x_train, c(nrow(x_train), 50, 50, 3))
x_test = array(x_test, c(nrow(x_test), 50, 50, 3))

y_train = to_categorical(readRDS('tridhogfull.rds')) #Run for full dataset
y_test = to_categorical(readRDS('tsidhogfull.rds'))
```

```
y_train= to_categorical(readRDS('tridhogbal.rds')) #Run for balanced dataset
y_test = to_categorical(readRDS('tsidhogbal.rds'))

y_train = to_categorical(readRDS('tridhog4000.rds')) #Run for random dataset
y_test = to_categorical(readRDS('tsidhog1000.rds'))

rm(list=ls()[! ls() %in% c('x_train', 'x_test', 'y_train', 'y_test')])
gc()

model = keras_model_sequential() %>%
  layer_conv_2d(filters = 16, kernel_size = c(3, 3), activation = "relu", input_shape = c(50, 50,
    layer_max_pooling_2d(pool_size = c(4, 4)) %>%
      layer_conv_2d(filters = 64, kernel_size = c(2, 2), activation = "relu") %>%
        layer_flatten() %>%
          layer_dropout(rate = 0.5) %>%
            layer_dense(units = 300, activation = "relu") %>%
              layer_dropout(0.2) %>%
               layer_dense(units = 200, activation = "tanh") %>%
                 layer_dropout(0.2) %>%
                   layer_dense(units = dim(y_train)[2], activation = "softmax")
model %>% compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = 'accuracy')

t1 = Sys.time() #Run these 3 lines together
history = model %>% fit(x_train, y_train, epochs = 50, batch_size = 50)
t2 = Sys.time()

cnntime = t2 - t1

model %>% evaluate(x_test, y_test)
```

Chapter 6

References

1. F. Suard, A. Rakotomamonjy, A. Bensrhair, A. Broggi, *Pedestrian Detection using Infrared images and Histograms of Oriented Gradients*, Intelligent Vehicles Symposium 2006, June 13-15, 2006, Tokyo, Japan
2. Lichun Zhang , Junwei Chen , Yue Lu , and Patrick Wang, *Face Recognition Using Scale Invariant Feature Transform and Support Vector Machine*, The 9th International Conference for Young Computer Scientists, November 18-21, 2008, Hunan, China
3. Johannes Stallkamp, Marc Schlipsing, Jan Salmen, Christian Igel, *The German Traffic Sign Recognition Benchmark: A multi-class classification competition*, IJCNN 2011, July 31-August 5, San Jose, California, USA
4. Rob Hess, *An Open-Source SIFT Library*, School of EECS, Oregon State University Corvallis, Oregon, USA
5. Gabriella Csurka, Christopher R. Dance, Lixin Fan, Jutta Willamowski, Cédric Bray, *Visual Categorization with Bags of Keypoints*, Xerox Research Centre Europe, Meylan, France
6. pyimagesearch, Histogram of Oriented Gradients (and car logo recognition)
7. Intel Software, Histogram of Oriented Gradients (HOG) Descriptor
8. Learn OpenCV, Histogram of Oriented Gradients
9. r/computervision, How to extract SIFT feature for SVM ?
10. stackoverflow, How to use SIFT features/descriptors as input for SVM training?
11. AIshack, SIFT: Theory and Practice
12. TensorFlow, Convolutional Neural Networks (CNN)
13. huksu, An R implementation of the SIFT algorithm