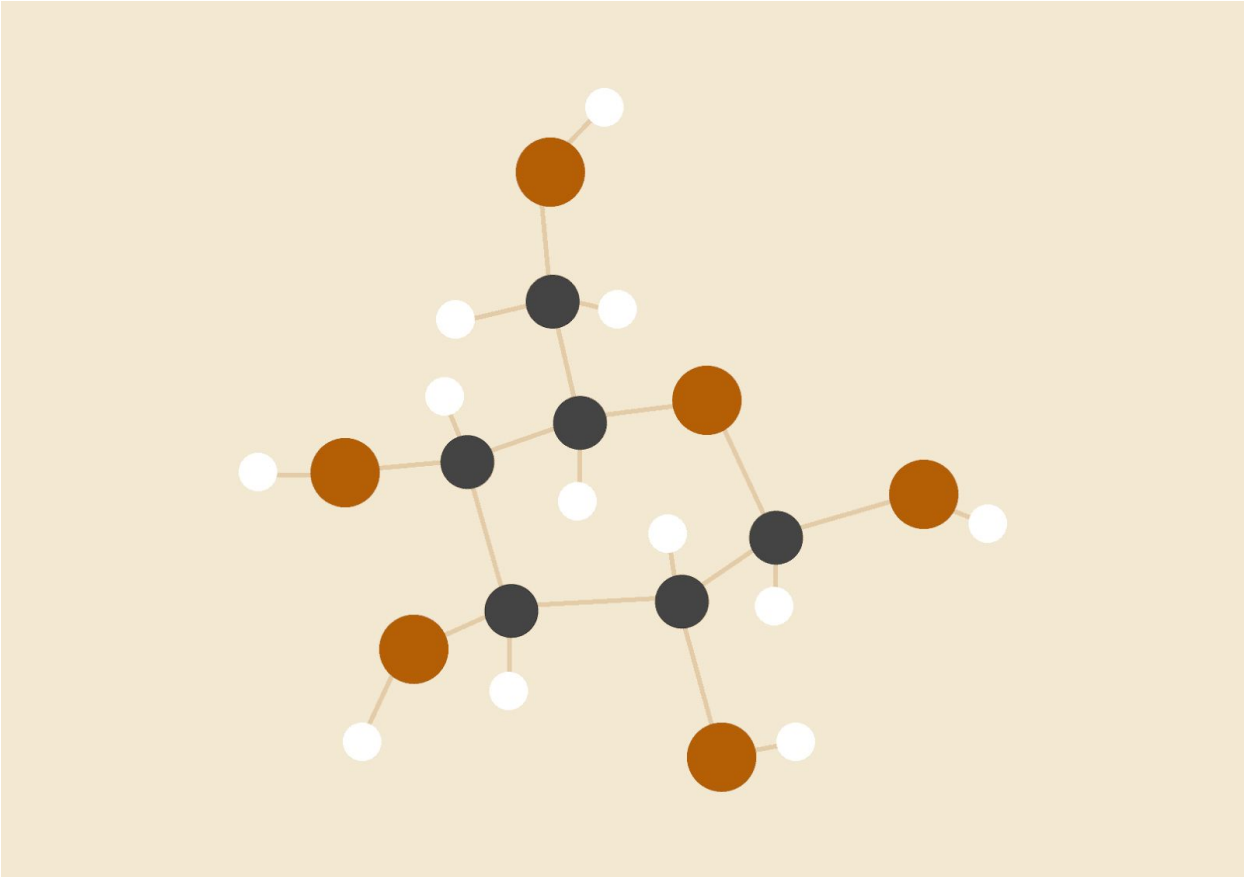
COMPUTER VISION REPORT

Investigation into the different approaches and their effects to accuracy in computer vision



Devon Wright

01.04.2018

COMP316-18A

INTRODUCTION

Over this assignment I have been building the means for a computer to process images in a way that it can then identify different images and then compare them to others. This means is done through breaking down the image into its edges and creating a basic number based description from that edge-image.

The detailed process is this:

Input Grayscale Image:

```
(defn load-image
  [filename]
  (let [file (File. filename)]
    (ImageIO/read file)))

(defn get-width
  [image]
  (.getWidth image))

(defn get-height
  [image]
  (.getHeight image))
```

Apply the kirch filter in 8 directions and store the result in separate images:

```
(defn calc-pixel-w-filter
  "Returns a pixel value after the application of the filter to the location"
    [image x y filter-matrix]
      ; need to deal with negative values so do clamping here
      (let [filtered-value (+ (int (apply + (map #(* (first (get-rgb image (+ x %1) (+ y %2))) (get-pos-in-3x3
      filter-matrix %3 %4)) [1 0 -1 1 0 -1 1 0 -1] [1 1 1 0 0 0 -1 -1 -1] [0 1 2 0 1 2 0 1 2] [0 0 0 1 1 1 2 2 2]))) 127)]
        (cond
          (< filtered-value 0) 0
          (> filtered-value 255) 255
          :else filtered-value)))

; This goes through each pixel from the orginal - calls calc-pix.. and saves that to the new image
(defn filter-image
  "Returns a filtered image"
  [image image-width image-height filter-matrix]
  (let [filtered-image (new-image image-width image-height)]
        (dotimes [x (- image-width 2)]
            (dotimes [y (- image-height 2)]
                (set-grey filtered-image (+ x 1) (+ y 1) (calc-pixel-w-filter image (+ x 1) (+ y 1) filter-matrix))))
        filtered-image))

(defn kirch
  [filename rotation-index]
  "Returns a BufferedImage containing the reuslts of the application of the kirsch filter"
  (let [loaded-image (load-image filename)
        width (get-width loaded-image)
        height (get-height loaded-image)]
      (filter-image loaded-image width height (get kirch-filters-vec rotation-index))))
```

Calculate the edge magnitude histogram from the brightest pixels of the kirched-images:

```
(defn edge-magnitude-hist
  [filename]
  "Returns a normalised frequency histogram with 8 bins containing edge magnitudes"
  (let [kirch-image-list (map #(kirch filename %) kirch-rotations) ;1 2 3 4 5 6 7
        width (get-width (first kirch-image-list))
        height (get-height (first kirch-image-list))
        final-kirch-image (new-image width height)]
    (dotimes [x width]
      (dotimes [y height]
        (let [list-of-grey-values (map #(first (get-rgb %1 x y)) kirch-image-list)]
          (set-grey final-kirch-image x y (apply max list-of-grey-values)))))
    ;(fill-empty-hist-bins (create-histogram-from-image final-kirch-image final-bin-size) final-bin-size)))
    final-kirch-image))
```

Calculate the direction histogram from the filter's direction used:

```
(defn edge-direction-hist
  [filename]
  "Returns a normalised direction histogram with 8 bins containing edge directions"
  (let [kirch-image-list (map #(kirch filename %) kirch-rotations) ;1 2 3 4 5 6 7
        width (get-width (first kirch-image-list))
        height (get-height (first kirch-image-list))
        final-kirch-direc-image (new-image width height)]
    (dotimes [x width]
      (dotimes [y height]
        (let [list-of-grey-values (map #(first (get-rgb %1 x y)) kirch-image-list)
              max-value (apply max list-of-grey-values)]
          (set-grey final-kirch-direc-image x y (.indexOf list-of-grey-values max-value))))) ;<- need to find the index
          of the max value since that will be our direction value
    ;(fill-empty-hist-bins (create-histogram-from-image final-kirch-direc-image final-bin-size 8) final-bin-size)))
    final-kirch-direc-image))
```

Create a basic histogram from the original image and normalise all histograms:

```
(defn create-histogram-from-image
  "Returns the histogram of the provided image"
  ([image bin-size]
    (let [flat-list (get-list-of-image image)
          distinct-pixels 256
          bins bin-size
          histogram (frequencies (bin-pixel-list flat-list (/ bin-size distinct-pixels)))]
      (fill-empty-hist-bins histogram bin-size)))

  ([image bin-size restrictor-pixels]
    (let [flat-list (get-list-of-image image)
          distinct-pixels restrictor-pixels
          bins bin-size
          histogram (frequencies (bin-pixel-list flat-list (/ bin-size distinct-pixels)))]
      (fill-empty-hist-bins histogram bin-size))))
```

```
(defn normalise-histogram
  [histogram]
  "Returns a normalised histogram [0-1]" ;Sort them first
  (let [sum-of-all-bins (apply + (vals histogram))]
    (map #(float (/ (second %) sum-of-all-bins)) (sort (seq histogram)))))

(defn normalise-image-histogram
  [filename] ;Sort them first
  "Returns a histogram that has been normalised from an image"
  (let [loaded-image (load-image filename)
        histogram (fill-empty-hist-bins (create-histogram-from-image loaded-image final-bin-size) final-bin-size)]
    (normalise-histogram histogram)))
```

Compare two image input using the previous steps:

The Image Descriptor:

The image descriptor is a length-24 value vector which contains the resulting histogram bins from the edge-magnitude, edge-direction, and original histograms, and when applied to another image descriptor it will show how similar the two images are.

```
(0.005088985 0.0 0.0 5.7720055E-5 0.10854257 0.03656468 0.019656468 0.075050995 0.06392496 0.029108964 0.023419216 0.
043448772 0.041154403 0.037745573 0.019924853 0.02220612 1.1272141E-4 0.008292449 0.01929774 0.026960077 0.038926464
0.032222223 0.02923242 0.022169312)
```

The sum of the list adds to one - the result from the normalisation - which we can then check each value in this list with another image descriptor. The closer to 1 the result is the more similar the two images are.

PREVIOUS RESULTS

Below is the results of checking the same image against itself, checking two different cars together, and checking a car and a plane together. (note that the results for all three tests are on the last line).

```
Results for image-similarity of ./resources/vehicle_images/car1.jpg & ./resources/vehicle_images/car1.jpg equals:
Results for image-similarity of ./resources/vehicle_images/car1.jpg & ./resources/vehicle_images/car2.jpg equals:
Results for image-similarity of ./resources/vehicle_images/car1.jpg & ./resources/vehicle_images/plane1.jpg equals:
(1.0000000016152626 0.7031076859057066 0.39795021594909485)
```

As you can see the two same cars are identical as expected, the two cars are quite similar (although it needs to be better than 0.7 as when we apply the functions to a large data set we will start to find we have a large amount of errors along with it), and lastly the car vs plane is dissimilar (although again at 0.4 similarity we will get a large list of errors on a larger dataset).

TWEAKING VALUES

In order to get a image using the unedited code, we use bin-sizes, filtering as our two options to tweak to get a better comparison between images. Currently bin size is set a size of 8 per histogram which results in a large range of grey pixel values to be mapped down to such a small range. This may lose a large amount of data that we may want in order to get a better descriptor to match images.

```
(def final-bin-size 8)
```

```
(def final-bin-size 32)
```

Here I am significantly increasing the bin-size which in theory should allow for a more accurate descriptor function.

```
assignment-3.core=> (image-similarity "./resources/vehicle_images/car1.jpg" "./resources/vehicle_images/car2.jpg")
Results for image-similarity of ./resources/vehicle_images/car1.jpg & ./resources/vehicle_images/car2.jpg equals:
0.7031076859057066
```

As it turns out there is no change at all in accuracy. It seems that even highly increasing the bin-sizes does not affect the accuracy in any way.

Next is to use the filters, here we will apply a single blur to the images before processing them.

```
assignment-3.core=> (image-similarity "./resources/vehicle_images/car1.jpg" "./resources/vehicle_images/car2.jpg")
Results for image-similarity of ./resources/vehicle_images/car1.jpg & ./resources/vehicle_images/car2.jpg equals:
0.7228448050454972
```

Applying a slight blur to the image beforehand produces a slightly better result when comparing to alike images.

Next I apply the blur filter three times before then taking it through the original functions.

```
assignment-3.core=> (image-similarity "./resources/vehicle_images/car1.jpg" "./resources/vehicle_images/car2.jpg")
Results for image-similarity of ./resources/vehicle_images/car1.jpg & ./resources/vehicle_images/car2.jpg equals:
0.8363796529488354
```

As you can see, having more filter blurring seems to make the image-descriptors more accurate. This being said it must be noted that this might just be better on this one image and maybe on the rest it may be making the descriptor worse.

CONCLUSION

Directly applying filters to the original functions makes the overall similarity value more accurate although it is all trial and error when it comes to knowing what makes an image better to be edge-detected on. You must also note that with more filters and modification you are making the overall runtime to increase drastically which in modern applications like self-driving cars or facial-recognition is not something that can be sacrificed, so in the end it is up to the limits and specifications you have when creating your computer vision program.