Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Image detection applied in mobile application development

Bachelor thesis

2016
Richard Ižip

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Image detection applied in mobile application development

Bachelor thesis

Study programme:	Computer Science
Study field:	2508 Computer Science, Informatics
Department:	Department of Computer Science
Supervisor:	Phd. Tomáš Kulich, RNDr.

Bratislava, 2016
Richard Ižip

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Richard Ižip
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science, Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Image Detection Applied in Mobile Application Development

Aim: (1) Identify photo of a certain telecast (displayed in a common TV) in a given (video) origin
(2) Integrate the algorithm with the mobile application

Supervisor:	RNDr. Tomáš Kulich, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	doc. RNDr. Daniel Olejár, PhD.

Assigned: 29.10.2015

Approved: 29.10.2015 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Richard Ižip
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Image Detection Applied in Mobile Application Development
Rozpoznávanie obrazu aplikované vo vývoji mobilnej aplikácie

Cieľ: (1) Identifikovať nepresnú snímku programu v bežnom televízore oproti danej (video) predlohe
(2) Integrovať algoritmus do vývoja mobilnej aplikácie

Vedúci: RNDr. Tomáš Kulich, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 29.10.2015

Dátum schválenia: 29.10.2015 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgment:

I would like to thank my supervisor RNDr. Tomáš Kulich, PhD. for his advices, help and patience.

Abstrakt

Práca pokrýva aplikáciu základných techník z oblasti spracovania obrazu, akými sú prahovanie a detekcia hrán. Cieľom je lokalizácia pozície užívateľom vytvorenej snímky vo videu. Tieto techniky sú kombinované s rôznymi perceptuálnymi hešovacími algoritmami. Neskôr je v práci používaná metóda lineárnej regresie pri riešení podobného problému. V tomto postupe sa snažíme lokalizovať pozíciu kratšieho videa v dlhšom. Práca je tiež zameraná na výber vhodných vlastností potrebných pre lokalizáciu videa a postup, v ktorom je video rozdelené na menšie časti.

Kľúčové slová:

Spracovanie obrazu, lineárna regresia, metóda najmenších štvorcov, opencv

Abstract

The thesis covers usage of the basic image processing techniques like thresholding and edge detection. The goal is to locate a user taken snapshot in a video with an emphasis on accuracy. It combines these techniques with perceptual hashes algorithms. Next we use a linear regression technique to solve similar problem by locating a shorter video in a longer video. We also discuss various properties used for locating the video. Finally, we introduce a strategy, where the video is being divided into parts.

Key words:

Image processing, linear regression, ordinary least-squares, opencv

Contents

List of Figures

Introduction

Nowadays mobile applications are responsible for enormous amount of work. Their usage ranges from entertainment to bank transfers. The main goal of this thesis is to create an algorithm, that can be used and integrated into a mobile application. The application will be targeted mainly at entertainment. The main idea is to show user a frame which will appear in a specific video. The videos should be television advertisements. The user will be challenged to capture the frame in the real advertisement, the most accurate he can, using his mobile phone. The application will offer a camera or a video recording to do so. The accuracy, with which the user captures the frame, will define user score or other user profits. Exact details are not important for the content of this thesis. What makes the task difficult to implement, is that the user should be able to capture the frame from a comfortable distance e.g his sofa. It means, that the data we will receive will also contain unnecessary information, for example a living room background.

In the thesis we expect, that the algorithm will be used on a server side, so we would not consider client side details. Our main focus is on the algorithm. We will mention some basic image processing techniques which may help us to develop the solution. For example various thresholding techniques or the Canny edge detection. We will also mention some perceptual hash algorithms which help us to recognize how pairs of images are similar. Linear regression technique is also involved in our experiments. In the thesis we consider two approaches. In the first approach, the user is challenged to take a photo of his television, when he thinks he sees the right frame. We then determine the accuracy from the original video and the frame the user sends. In the second approach, the user is forced to record a video and label the frame, he thinks is the correct one.

The experiments we made are programmed in Python. We chose Python programming language mainly because of its simplicity and currently available binding of opencv library, that we also use. The library offers many methods for image processing techniques. There are many articles about various usage of detection techniques, but we consider our problem to be quite unique. Our goal is not necessary the image extraction. It is just one of the possibilities to solve the problem. Generally we are looking

for anything that can detect the image the most accurate way. It is accuracy that is our main goal.
All the resources can be found on the attached DVD and in our Github repository *https://github.com/rizip1/Image-processing*.

Chapter 1

Image processing tools

In this chapter we will describe what is an image processing, introduce fundamental terms, color spaces, and some basic image processing techniques that we use in our thesis. We will consider those which are already implemented in opencv library. We will shortly cover their details.

1.1 Fundamentals

It is difficult to properly define where image processing starts and where it ends. Gonzalez (et. al) [13] confirm that statement.

> There is no general agreement among authors regarding where image processing stops and other related areas, such as image analysis and computer vision, start. Sometimes a distinction is made by defining image processing as a discipline in which both the input and output of a process are images.

As it can be seen from the statement above, image processing may be defined as a discipline where input and output are images. On the other hand, if the definition would be that strict, it would mean that a simple operation such as computing an average intensity of an image would not belong to image processing.
For the purpose of this thesis, we consider image processing to be a discipline where input is always an image and output is an image or an information related to the input image, describing the structure of the image. Image processing does not analyse a meaning of a given picture, that is what distinguishes it from image analysis or computer vision.
When referencing to term *image*, we usually consider a matrix of size $M * N$ where M represents image width and N it's height. Both width and height are measured in

Figure 1.1: Example of a grayscale image.

pixels. However, for better illustration, we often display *image* in a non-matrix form. Each element of that matrix is represented by a finite number of items. These items represent various image properties (hue, intensity, etc). Usually we consider one item for grayscale images (representing intensity) or three items representing entries from HSV or RGB color space. There is more about color spaces in section 1.2. An example of a grayscale image is shown in Figure 1.1.

Intensity represents total amount of lightness. It can be computed as $\frac{1}{3}(R + B + G)$ where R, G, B values are values taken from the RGB color space. It is in range from 0 to 255. Where 255 stands for white color and 0 for black. Other values from that interval represent different shades of grey.

Grayscale image is an image whose pixels express intensity values in range 0 to 255.

1.2 Color spaces

Color space describes how a color appears. We divide color spaces into additive and subtractive. In additive color space, combining colors results in lighter color (higher values) what is an opposite to subtractive models. We do not mention subtractive models in our thesis because we do not use them.

1.2.1 RGB

RGB color space has its application in electronic systems (television, monitors) or in photography. It is an abbreviation for red, green and blue color. These colors are called primary. Primary colors are the root of any other colors, therefore from primary colors any existing color can be constructed. Values of R, G, B are usually integers in range [0,255], but in so called *normalized form*, they can be decimals in range from zero to one [6]. When consider the *normalized form*, we can imagine a cube (*normalized rgb cube*) which vertices are in a form of $V[x, y, z]$ where x, y and z are bits. Diagonal values

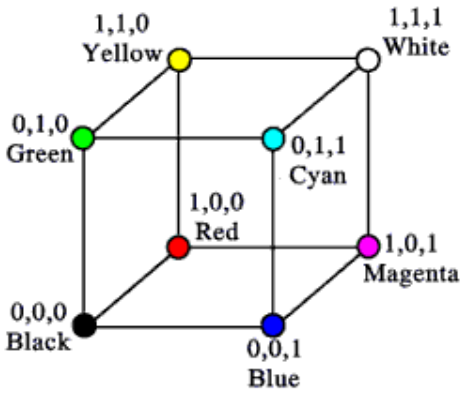
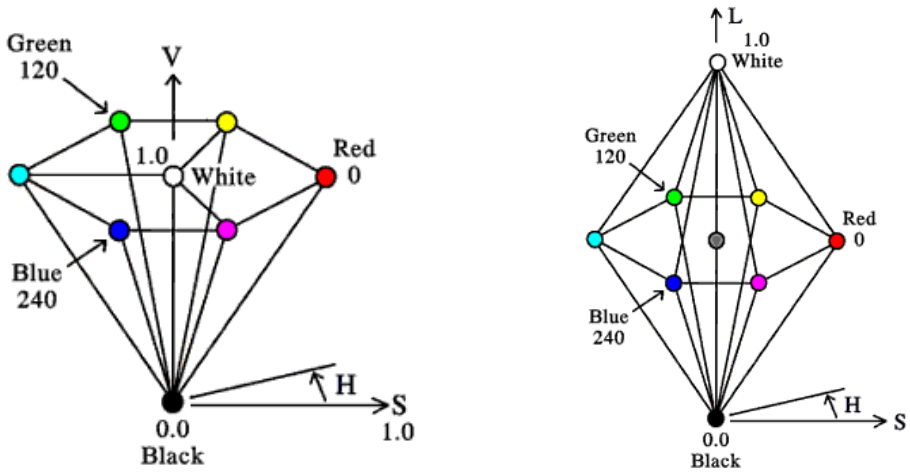


Figure 1.2: Cube representation of RGB color space [1].



(a) Single cone HSV representation [1]. (b) Double cone HSL representation [1].

Figure 1.3: HSV and HSL color spaces.

are shades of grey (all coordinates have the same value). Example of the *normalized rgb cube* is shown in Figure 1.2.

1.2.2 HSL, HSV

These color spaces are more intuitive for human perception than RGB. They are often used in color pickers, image editing software or image analysis. They are all just a mapping of RGB. HSV color space is visualized by a single cone and HSL by a double cone as shown in Figure 1.3a and 1.3b. H in both color spaces represents hue, S represents saturation, L stands for lightness and V for value. Hue is an angle around a vertical axis of a cone. In color pickers it is usually represented by a wheel containing basic colors (red, orange, yellow, green, blue and magenta). Definition of saturation

differs in these spaces. To simplify, we can define it as an amount of chroma in a color. In cone representation, it is the distance from the vertical axis. Intensity is an average of R, G, B values. Value is the largest component of a color.
In image processing we often need to convert images from RGB to HSV or HSL.

$$V = max(R, G, B) \tag{1.1}$$

$$S = \begin{cases} \frac{V - min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

$$H = \begin{cases} \frac{60(G-B)}{V - min(R,G,B)} & \text{if } V = R \\ 120 + \frac{60(B-R)}{V - min(R,G,B)} & \text{if } V = G \\ 240 + \frac{60(R-G)}{V - min(R,G,B)} & \text{if } V = B \end{cases} \tag{1.3}$$

Formulas 1.1 to 1.3 describes converting from RGB to HSV implemented in opencv [11].

$$V_{max} = max(R, G, B) \tag{1.4}$$

$$V_{min} = min(R, G, B) \tag{1.5}$$

$$L = \frac{V_{max} + V_{min}}{2} \tag{1.6}$$

$$S = \begin{cases} \frac{V_{max} - V_{min}}{V_{max} + V_{min}} & \text{if } L < 0.5 \neq 0 \\ \frac{V_{max} - V_{min}}{2 - (V_{max} + V_{min})} & \text{if } L >= 0.5 \neq 0 \end{cases} \tag{1.7}$$

$$H = \begin{cases} \frac{60(G-B)}{S} & \text{if } V_{max} = R \\ 120 + \frac{60(B-R)}{S} & \text{if } V_{max} = G \\ 240 + \frac{60(R-G)}{S} & \text{if } V_{max} = B \end{cases} \tag{1.8}$$

Formulas 1.4 to 1.8 describes converting from RGB to HSL implemented in opencv [11]. All properties may be normalized to fit required ranges of values. There are more color spaces we have not mentioned. More on this topic can be found in Ibraheem's article [6].

1.3 Filtering

Before applying various image processing methods, we usually need to smooth or blur the given image. This is achieved by using various filters. Opencv offers many filtering methods. We have used these filters:

- Average blurring
- Median blurring
- Gaussian blurring

1.3.1 Average blurring

Average filter is probably the simplest filter. Average filtering uses a convolution kernel [1] which it moves along all the pixels in the image. The kernel might be arbitrary size, but it's width and height should be odd. Otherwise central pixel would not be in the middle of the kernel. The kernel is filled with ones what means, that all pixels in the kernel have same weights. If we choose a pixel which is on the edge of the kernel, we can compute missing pixels as an average of known pixels. The effect after averaging is a blurred image. Kernel of size $3 * 3$ can be described as in Formula 1.9.

$$K = \frac{1}{width * height} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{1.9}$$

1.3.2 Median blurring

In median blurring an idea is also simple. Create a grid as in the average filtering, but instead of averaging pixels values, take their median value. Unlike using average or Gaussian kernel, pixel is always assigned a value that exists in the kernel. Kernel size has to be odd. Median blurring is highly used in removing *salt and pepper* noise [11].

1.3.3 Gaussian blurring

More sophisticated approach than an average or a median filter is a Gaussian blurring. Is uses a weighted Gaussian kernel. Gaussian kernel is computed from the Gaussian

[1] Kernel can be understood as a matrix

function for the two dimensional space [4].

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-x^2+y^2}{2\sigma^2}} \tag{1.10}$$

Formula (1.10) shows two dimensional Gaussian function where x and y are distances from the origin in horizontal and vertical direction, σ is a standard deviation of Gaussian distribution. Smaller value of σ implies thinner peak of the Gaussian function. Values we get from the Gaussian function are used to construct a kernel which is applied to the image. Of course the kernel dimensions has to be odd. For each pixel we compute it's value as the weighted average of it's pixels neighbourhood. The original pixel has the highest weight and the further the neighbour pixel is from that pixel the smaller weight it is assigned. Gaussian function of every pixel is positive (non-zero). Gaussian filter preserves edges better than the averaging filter and it is usually used before edge detection.

$$K = \frac{1}{ksum} \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \tag{1.11}$$

Formula (1.11) shows kernel made from discrete approximation to Gaussian function with $\sigma = 1.0$ [12]. Variable $ksum$ refers to the sum of all pixels inside the kernel. In this case it is 273.

1.4 Image segmentation

Image segmentation is the process of partitioning a digital image into multiple areas (set of pixels) that cover it. These areas are called segments. The main aim is to simplify image representation into something that is easier to analyse. Image segmentation is usually used to locate object boundaries.

1.5 Thresholding

Thresholding is basic image processing method and simplest method of image segmentation. Output of thresholding is a binary image. Binary image is an image which has

Figure 1.4: Simple thresholding applied to Figure 1.1

only two possible values for each pixel. The idea of thresholding is to establish threshold value such that if the pixel has greater value than the threshold it is assigned foreground value (usually white) otherwise it is assigned background value (usually black). It's example shows Figure 1.4. Opencv offers three thresholding methods [11].

- Simple thresholding
- Adaptive thresholding
- Otsu's binarization

All of these methods expect input image to be 8-bit single channel. Usually grayscale.

1.5.1 Simple thresholding

As title indicates, this is the simplest and the quickest thresholding method. Threshold value is global for all pixels in an image. If the pixel value is higher than the given threshold it is assigned one value otherwise second.

1.5.2 Adaptive thresholding

When using simple thresholding, we have to set threshold value globally. It might be good enough when applied to specific picture however, when dealing with images from different spectrum it might not work well. The threshold which may suit to one picture may fail for very simple picture, but with different lightness intensity. For more global purpose adaptive thresholding can be used. In adaptive thresholding, threshold is calculated for small regions of an image. More thresholds for different regions of the image give's us better results for images, where illumination varies. It can be computed using a grid which represents pixel's neighbourhood. Threshold is then computed either as an average, median or using similar approach, using values from the local grid. The size of the grid is important because if it was too small or too large, it would not work

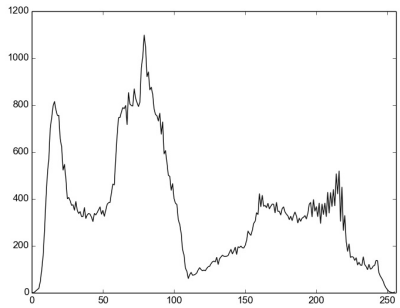


Figure 1.5: Sample histogram representing amount of intensity for Figure 1.1.

as desired. If it was too large, it would behave similarly to the global thresholding. If it was too small, the information about the pixel area would be inadequate.

1.5.3 Otsu's binarization

Stronger technique than the simple thresholding is the Otsu's binarization. It is named after Nobuyuki Otsu [5]. When using simple thresholding, it is difficult to guess the best threshold value. When performing Otsu's binarization we consider bimodal images. Bimodal image is the image whose intensity *histogram* has two 'peaks'. Gonzalez (et. al) [13] define histogram as:

> The *histogram* of a digital image with gray levels in the range $[0, L-1]$ is a discrete function $h(r_k) = n_k$, where r_k is the kth gray level and n_k is the number of pixels in the image having gray level r_k.

Intensity histogram can be imagined as a plot, which supports an overall information about the intensity distribution in the image. On X-axis there are values from 0 to 255 which represent intensity. Y-axis corresponds to the number of pixels of the concrete intensity. Values on Y-axis indicate shape of 'peaks'. Despite the above definition which considers only intensity, histogram might be constructed for arbitrary property not only intensity. Example of histogram is shown in Figure 1.5.

When using Otsu's method the threshold is found as a value in the middle of the histogram's 'peaks' such that the within-class variance is minimal for both classes. Although Otsu's binarization works well for bimodal images, it does not work fine for other images. Image is usually not bimodal when it consist a lot of noise. Fortunately, we can usually get rid of it using filters. Gaussian filter might work well and minimizing the noise, it often converts an image into bimodal.

1.6 Edge detection

Edge detection is a technique used to identify points in the image with significant intensity contrast. Result of the edge detection is a binary image. The selected points are grouped into edges. It reduces the amount of data however still keeps useful information. To have better idea how output of the edge detection looks like, we include Figure 1.6.

1.6.1 Canny edge detection

Canny edge detection is an implementation of the edge detection technique. It was developed by John F. Canny in 1986 [11]. The algorithm is known as an optimal edge detector [3]. Implementation of Canny's algorithm in opencv can be described in four separate steps according to opencv documentaion [11] and Bill Green's tutorial [3].

Step 1 First we need to filter out the noise. Opencv uses $5 * 5$ Gaussian kernel for that purpose.

Step 2 After blurring we need to find intensity gradient. Sobel kernel is used to perform this operation. The image is filtered with that kernel along horizontal and vertical axis. The outputs we get are two new images G_x and G_y. G_x is the result of the horizontal and G_y of the vertical filtering. Sobel kernel behaves as a simplification of a derivative. Formula 1.12 displays horizontal and formula 1.13 vertical Sobel kernel. From these two picture we can determine edge gradient (Formula 1.14) and each pixel's direction (Formula 1.15).

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{1.12}$$

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{1.13}$$

$$Edge_{gradient}(G) = \sqrt{G_x^2 + G_y^2} \tag{1.14}$$

$$angle(\Theta) = \tan^{-1}(\frac{G_y}{G_x}) \tag{1.15}$$

Figure 1.6: Sample output of canny edge detection method. As an input image we used figure 1.1

We have to fix the case when G_x is equal zero. If G_y was zero, we set direction to be 0 degrees. Otherwise we set it to be 90 degrees. After we know the direction, we round it to one of the four angles. D in Formula 1.16 represents direction.

$$D = \begin{cases} 0° & \text{if } \in [0, 22.5) \cup [157.5, 180] \\ 45° & \text{if } \in [22.5, 67.5) \\ 90° & \text{if } \in [67.5, 112.5) \\ 135° & \text{if } \in [112.5, 157.5) \cup (157.5, 180] \end{cases} \tag{1.16}$$

Step 3 In this step we need to remove pixels which do not form any edge. If a pixel value is not a local maximum in it's neighbourhood, it is set to zero. Otherwise it is set to one. The scan is performed in the gradient direction. Output is a binary image which simply determines if the pixel belongs to some edge or not.

Step 4 At last, edges are reduced even more. Two threshold values are required for this stage. They decides if an edge will be preserved. Let's call them *mint* and *maxt*. If edge intensity gradient is more than *maxt* we can call it *true edge* an we preserve it. If the edge intensity is below *mint* we no longer consider it to be edge. If it lies below these values we determine it's affiliation based on the fact, if it is connected to some *true edge*. If it is we preserve it, otherwise we discard it.

1.7 Contours

After usage of segmentation method we usually need to extract some part of a resulting binary image. This is what contours are used for. Contour is a curve connecting continuous points with the same color or intensity. Because segmentation techniques often use grayscale images, more often we assume intensity to be the point of interest.

1.7.1 Contour approximation

We hardly get perfect shape of desired object after image segmentation. Consider we want to extract a rectangle, but there are some holes along it's boundary. We may want to extract the rectangle with those holes rather than having a different shape. To achieve this we may use *contour approximation*. It approximates the contour shape to a similar shape with less vertices. In opencv a user chooses a shape he wishes to approximate and a value (ϵ) that determines precision. Opencv uses implementation of Douglas-Peucker algorithm [9]. The algorithm starts with the first and the last point on the curve and imagines a straight line between them. Then from the points between the first and the last point, it chooses a point whose distance from the imaginary line is the farthest. If that distance is smaller than ϵ, it removes all the points from the start point up to the current point. Otherwise the curve is divided into two parts.

1. From the first point up to the current point.
2. From the current point up to the end point.

After the division, the procedure is recursively called on both the curves. Finally, the two divided curves are joined.

Chapter 2

Recognition techniques

In this chapter we will describe recognition techniques which we have used in our work. We will describe various perceptual hashes algorithms and a method of ordinary least-squares.

2.1 Perceptual hashes

The goal of perceptual hashes is to identify a similarity of two images. For humans, it is a trivial task to look at the pictures and decide whether they are similar (one is just slightly modified copy of another) or not. For computers, the problem is not that easy. When constructing cryptographic hash, the hash is randomized. It means that two hashes for the same user, using exactly the same cryptographic algorithm, would hardly be the same.

The situation changes for perceptual hashes. Perceptual hash is constructed with respect to the given entity (in our understanding it is a picture) properties. Perceptual hash algorithm may use arbitrary approach to construct the hash, but with respect to certain relations (in our case it might be relation between pixels values). When two hash strings, constructed from the two different pictures using the same perceptual hash algorithm, are the same or differs just in one or very few bits, we may suppose that the pictures are very similar. However their size or brightness may vary. The distance between these hashes is known as a *Hamming distance*. It represents the number of the positions at which the two given hashes varies.

Basic step in perceptual hashes it to shrink an image to very small size and convert each pixel's value from RGB into grayscale. We do it to get rid of high frequencies and use intensity as a single attribute for each pixel. High frequencies describes detail in the image and low frequencies describes image's structure. Pictures with many

details has lot of high frequencies, but very small pictures have only low frequencies. There simply is not enough space to capture the detail. If we take some string which represents intensity values for the given image, then the places where the intensity values between the neighbour pixels changes rapidly are high frequencies. For example edges are high frequency content.
Grayscale is used because it is not as prone to luminance changes as RGB. After the image is shrunk, chosen algorithm constructs a hash with regard to given criteria. The shrunk dimensions may differ from implementation to implementation.

2.1.1 aHash

The aHash algorithm calculates intensity average for the given image [8]. Then it iterates all the pixels and for every pixel appends one bit to a hash string. The bit value is determined according to if a pixel's value is higher or lower than the image's average. The image is usually shrunk to $8 * 8$ pixels.

2.1.2 dHash

The dHash algorithm constructs the hash with regard to an intensity difference between pixels [7]. Instead of shrinking the image into $n*n$ dimension it shrinks it to $(n+1)*n$ dimension. The value of n is usually 8 . Then it iterates all the pixels from top-left to bottom-right and set the corresponding bit in the hash string to zero or one depending on whether $p[i][j] < p[i][j+1]$. The $p[i][j]$ is the intensity value for the pixel on ith row from the top and jth column from the left. After the procedure ends the hash is $n * n$ long.

2.1.3 pHash

This algorithm is more robust than the two mentioned above. To reduce high frequencies it uses discrete cosine transformation (DCT) [8]. DCT converts image into cosine waves which oscillate on different frequencies. It is the transformation from spatial to frequency domain. The comprehensive explanation of DCT is beyond the scope of this thesis.
The algorithm shrinks image into $32 * 32$ pixels. Then it computes the average DCT value using top-left $8*8$ pixels and ignoring the first item. The top-left square represents the lowest frequencies. First item (DC term) is not counted because it corresponds to zero frequency, that represents the average brightness across the whole image and may be very different.

Then it iterates pixels in the top-left square and set hash string values depending on whether iterated DCT value is above or below the average value.

Implementation of all the above algorithms can be found on Github or on the attached DVD in the *hashes* module.

2.2 Linear regression and least-squares solution

To describe the method of ordinary least-squares (OLS) which we use later in our experiments it makes sense to first briefly introduce a linear regression technique which goes hand by hand with it. Linear regression is the technique used to describe relations between predictor variables ($X_1, X_2, ..., X_n$ where $n \in 1, 2, ...$) and response variable Y. The predictor variables are also called *independent* variables and the response is *dependent* variable. The predictor variables can be understood as properties using which we can predict the value of Y. The predictor X is a vector consisting from these variables. We will call them properties. Assume we want to predict person's age based on amount of water he drinks per day and amount of sleep. Then the properties are drunk water per day and amount of sleep. The response is age. Consider we have m tuples of these properties. These tuples are predictors.

2.2.1 Mean function

Relationships are described using *mean function*. The function depends on it's predictors. First we introduce it's definition for predictor with single property and one response. Formula 2.1 shows that definition.

$$E(Y|X = x) = \beta_0 + \beta_1 x \tag{2.1}$$

Y is a response when the predictor's value is equal x. The β_0 parameter stands for intercept and β_1 for slope [14]. These parameters need to be estimated from given data set. Consider Figure 2.1. Every response Y has a single property predictor X. For simplicity we do not mark it as a vector but as a scalar. Also consider that there are known responses for $X = 0, 1, 2, 3$ respectively. Imagine we want to estimate value of Y for $X = 4$. There is no linear function which crosses all these points. So we want to find a linear (mean) function which has the lowest distance from these points. One function which tries to minimize the distances from all the points is drawn using dashed line. The distances are drawn using solid, vertical lines. Distances represent errors (how much real value differs from the estimated value). For predictor X_i we

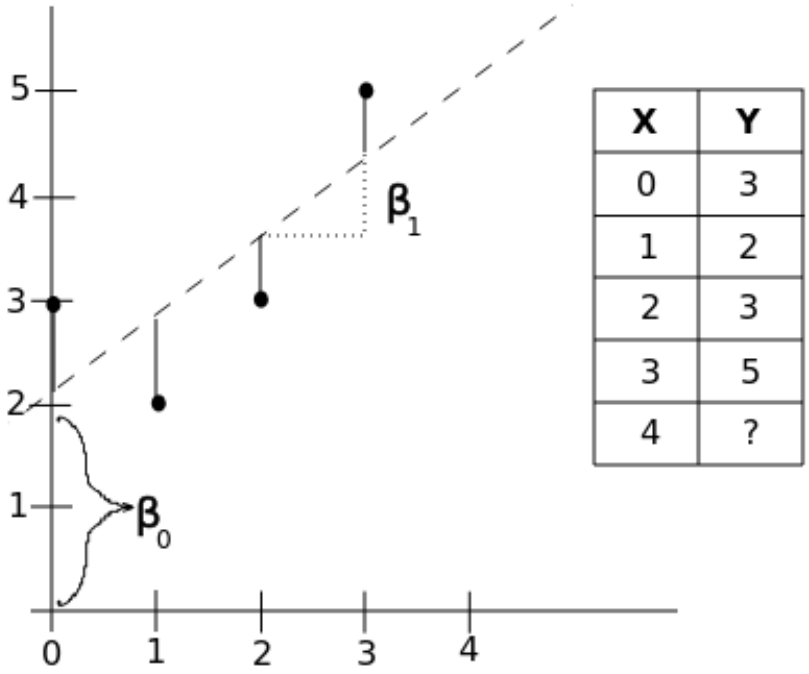


Figure 2.1: Mean function example.

define corresponding error $e_i = (Y_i - E(Y_i, X_i))^2$ where Y_i is the response for X_i and i is in the range of predictors count. The best fitted linear function is the function with the lowest sum of these errors for all predictors. Determine searched function using that idea is known as a *least-squares solution.*

More practical example would be to estimate a price of a house according to it's age. In that case the single predictor's property will be the house's age and the response will be the house's price. It might be drawn similarly to Figure 2.1 except that the x axis would contain the house's age and y axis it's corresponding price. We may again fit the line with least distances from all the points and predict new values from it.

We have defined mean function for single property predictor but usually we have more than just one property. General mean function for n properties shows Formula 2.2. X is the predictor and $X_1, ..., X_n$ are it's properties.

$$E(Y|X) = \beta_0 + \beta_1 X_1 + ... + \beta_n X_n \tag{2.2}$$

So for n properties we have $n + 1$ beta terms to estimate including intercept. It is however not possible to draw or imagine a plot with more than two properties (two properties and one response can be drawn in 3D). On the other hand we may capture all the predictors properties and responses with matrices as is shown in Figure 2.3.

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{bmatrix} X = \begin{bmatrix} 1 & x_{11} & . & . & . & x_{1m} \\ 1 & x_{21} & . & . & . & x_{2m} \\ . & . & & . & & . \\ . & . & & . & & . \\ . & . & & . & & . \\ 1 & x_{n1} & . & . & . & x_{nm} \end{bmatrix} \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ . \\ . \\ . \\ \beta_m \end{bmatrix} e = \begin{bmatrix} e_1 \\ e_2 \\ . \\ . \\ . \\ e_n \end{bmatrix} \tag{2.3}$$

Y is the vector of n items. X is the $n * (m + 1)$ matrix for predictors properties. We add a column full of ones because of the intercept (β_0). β is a vector of $m + 1$ items of β_s values estimated from the data where $s \in [0, m]$. And finally e is a error vector of n values containing squared distances from the estimated linear function. The count of predictors is n and m is the count of their properties.
Regression model using above notation can be written as $Y = X\beta + e$ [14]. When written for single rows $Y_i = X_i\beta + e_i$. $X_i = X_{i1}, ..., X_{im}$ and $\beta = \beta_0, \beta_1, ..., \beta_m$.

2.2.2 OLS calculation

Now when we know the idea behind OLS we need to know how to calculate it. We need to estimate β values such that $\sum_{i=1}^{n}(Y_i - X_i\beta)^2 = \sum_{i=1}^{n}(Y_i - (\beta_0 + \beta_1 X_{i1} + ... + \beta_m X_{im}))^2$ is minimal. The predictors count is n and m is the count of the predictors properties. Calculation of beta values shows formula 2.4 [14] where X^T is a transposed matrix to matrix X and X^{-1} is a inverse matrix to X.

$$\beta = (X^T X)^{-1} X^T Y. \tag{2.4}$$

OLS is very handy when used for prediction but it has also other usage. We may have two data sets represented by multidimensional matrices and we may want to know how are they similar. In other words we want to know if one is a linear combination of other. However in complex models this is almost impossible to achieve. So we would rather know their similarity error which describes the difference. Now if we have more of these models we can determine which are the most similar.

Chapter 3

Implementation

In this chapter we describe our implementation approach and review the results. We consider two different approaches that we used. Segmentation techniques in combination with perceptual hashes algorithms, for single snapshot, and a solution with two videos using least-squares calculation.

3.1 Perceptual hash approach

Firstly, we will describe our solution for image extraction using segmentation techniques. One of the possible solutions to solve our problem was to use edge detection or thresholding to extract a part of the picture we need. In this case the application could work the way, that the user will be requested to take a snapshot, when we thinks he sees the desired frame in his television. It is the lightweight solution to our problem when consider the data transfer between the client and the server. The snapshot does not have large size (in comparison to an approach that we use later). The most straightforward solution would be, to force the user, to take the photo containing only his television and a very little of other information. However, that approach would be too restrictive. The advantage we would get from this approach is, that we could skip the object extraction and immediately apply some perceptual hash algorithm technique.
To have a better user experience, we allow the user to take a snapshot from a comfortable place e.g it's sofa. Our only constraint is that the television has to cover at least $\frac{1}{20}$ from the total space in the photo.
We want to create a solution which would work during arbitrary lightness conditions. Therefore we do not restrict the user to take the snapshot only during a specific part of a day or amount of a light in the room.

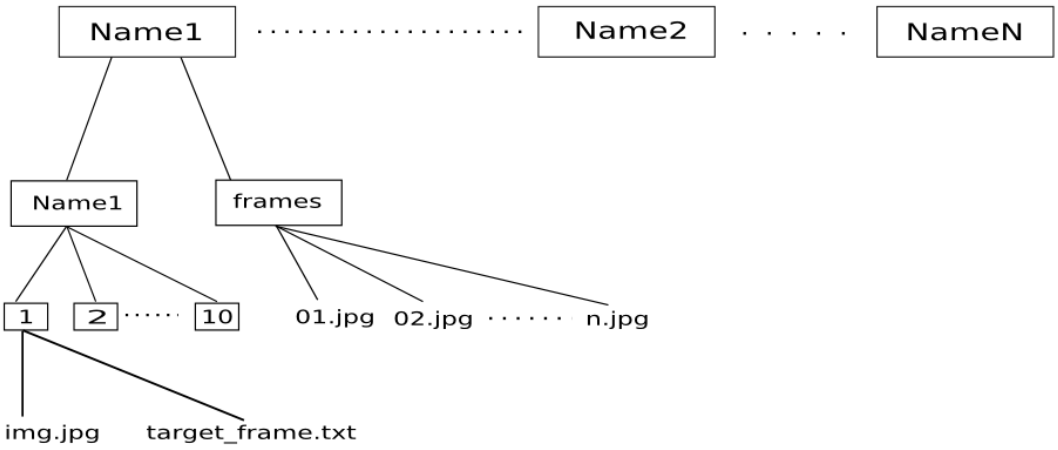


Figure 3.1: Data structure for testing script.

3.1.1 Data set

To determine the best solution, we needed to test our ideas against a meaningful data set. We created the data set containing one hundred of photos, from which fifty have dark and fifty have bright background. We also took these photos from different distances and different angles, but with respect to the fact, that a television covers at least $\frac{1}{20}$ of the whole picture. To collect the photos we downloaded sample advertisements from youtube using *youtube_dl* utility. Then we parsed these videos into single frames using our *frame_parser* function from the *utils* module. We displayed chosen frames on our laptop's fullscreen and connected it to a television. Then we took the photos using a mobile device. To be able to test our results, we wrote the chosen frame numbers to text files. That way we got our data set for our future experiments. To test our approaches effectively, we needed to create a testing script which would automate the process. But before we did it, we needed to create an exact structure for our data set, so the script could iterate it easily.

The data set structure is shown in Figure 3.1. For each advertisement we created a folder with it's name. The folder contains a folder with the same name and a folder named *frames*. The *frames* folder contains advertisement's frames with jpg extension. The folder with the same name as the advertisement contains ten folders named 1,2 ... 10. Each of these folders contains *img.jpg* and *target_frame.txt* files. A snapshot is in the *img.jpg* file and it's position in the advertisement is in the *target_frame.txt* file. To speed up the process of creating the structure a little, we created *create_hash_structure* method. It's arguments are a folder where to create the structure and a count of directories to be created in each advertisement folder. We used ten as a count. It also creates *target_frame.txt* in all of these folders. The functions are located in the *utils*

module.

3.1.2 Testing script

Once we had our data set in the specific structure, we could create a testing script. The idea is to iterate the advertisements folders and repeatedly apply chosen algorithm to every snapshot. The algorithm can be passed as a command line argument. We named the testing script *screen_detector*. The first CLI argument is a source folder with the data structure described above. Before running the algorithm, the script creates a destination folder where it will save extracted pictures. It is created in *cwd/src/screen_detector/* directory, where *cwd* is the current working directory. If the folder exists and is not empty, we clean it's content. If the second argument is passed to the script, we pass it as the algorithm name otherwise we choose a default algorithm and run the tests. The third argument which can be passed is a perceptual hash algorithm. The options are *d* for dHash, *a* for aHash and the default perceptual hash used is pHash. All the testing stuff is handled by a class called *ScreenDetector* which is located in the *screen_detector.py* file.
The script invokes *find_screen* method which only argument is a recognition technique to be used. Simplified method body is shown below. In it's beginning we first initialize *error* variable to zero. Then we iterate all the *img.jpg* files. For every file we also read it's position in the advertising and use it to load the original frame from the *frames* folder. To perform this we wrote *_get_original_image_data* method which returns a dictionary containing a position in the advertisement and a snapshot image read by the *imread* opencv method.
To find the snapshot position in the advertisement, we first create a perceptual hash from the snapshot image (*img.jpg*) and save it into *hash_original* property. To get the hash we use *a_hash*, *d_hash* and *p_hash* functions from the *hashes* module we wrote. They are used in the *_get_original_hash* method which has an image as an argument and returns a hash from that image. We also set convert flag to *False*, to do not use image as a path, but as a real source in the perceptual hash methods.
In *_call_recognition_method* we call one of the image extraction methods. After the chosen method is finished, we have the best extracted image saved in the *best_image* property. We will introduce the recognition methods later in Section 3.1.4.
In *_get_position_accuracy* method we iterate a fixed amount of the original frames before and after the *img.jpg* file in the advertising. We set the value to be 50, so we iterate 50 frames before and after the desired frame. These are the frames from the *frames* directory. For all these files, we create a perceptual hash and compare it with the hash for *img.jpg* file using *compare_hashes* function from the *hashes* module.

We choose the frame with the least difference from the snapshot image (*img.jpg*) and return it's absolute distance from the correct position. The correct position is loaded from the (*target_frame.txt*) file. After we get the accuracy, we increment the *error* variable. We also add accuracy to *stats* dictionary which keys are the advertisements names postfixed with an underscore and the number of the snapshot. In our case the number is in range from one to ten. It is because for each advertising we have ten snapshots.

Finally, we call *_put_stats_into_file* and *create_hist* methods. The first method saves the results in the *cwd/src/results/results.txt* file using the *stats* property values and the second creates a histogram in the *cwd/src/results/histogram.png* file. The histogram describes the distances from the original frame. These two methods give as quite a good view about the algorithm accuracy before it's usage in the application. In the end of the method, we divide the *error* variable by the length of the *stats* and return that value as an average error.

To check only the success of extracting the television, we looked manually at the images in the destination folder (*cwd/src/screen_detector*). This could of course be done by remembering all the corner points of the television in a specific file and perform other checks. However, to create the data set it would take much longer. We first needed to know if the approach had sense to be used in the application, so this solution was enough for us. We could also iterate all the frames, when finding the best hash score, but it would make our testing script run longer. Our main aim in this stage was to find out if the solution makes sense therefore iterating only fifty frames was both sufficient and quick.

```
1  def find_screen(self, method):
2    self._clean_or_create_dest_dir()
3    error = 0
4    os.chdir(self.source)
5
6    for advertising in os.listdir(os.getcwd()):
7      os.chdir(advertising)
8      os.chdir(advertising)
9      for sample in os.listdir(os.getcwd()):
10       self.best_score = 1
11       os.chdir(sample)
12       image_name = ... # here we combine advertisement and sample names
13
14       # Read and convert into grayscale
15       img = cv2.imread('img.jpg', 0)
16       image_data = self._get_original_image_data()
```

```
17          self.hash_original = self._get_original_hash(image_data['img'])
18          self._call_recognition_method(img, method)
19
20          cv2.imwrite(os.path.join(self.dest, image_name), self.best_image)
21          accuracy_error = self._get_position_accuracy(image_data['position'])
22          error += accuracy_error
23          self.stats[image_name] = accuracy_error
24
25          os.chdir('../')
26      os.chdir('../')
27      os.chdir('../')
28
29  average_error = float(error) / len(self.stats)
30  y_max = self._put_stats_into_file(average_error)
31  self._create_hist(y_max)
32  return average_error
```

find_screen

3.1.3 Extracting contours

Before we describe approaches, that we used in our work, we have to explain contours extraction. We use it in the end of every method called by the *_call _recognition _method* which applies chosen segmentation technique. After applying segmentation technique, we need to extract the television from the picture. Opencv offers *findContours* method. The method takes the binary image as it's first argument, but it also modifies that image. To preserve the binary image we can copy it. Next two parameters are a retrieval structure and an approximation method. Their description can be found in opencv documentation [11]. The method finds all contours in the image. We are interested only in one contour (the television contour), so we sort the contours by their area size and pick only the four largest. Then we find a perimeter of the current contour using opencv *arcLength* method. We use it to approximate a polygonal curve with the given accuracy value. This is performed by opencv *approxPolyDP* method. If the contour has more than four edges, we expect to be a television in it. Then we check if it is at least $\frac{1}{20}$ from the total area in the original image. If the condition is met, we use given perceptual hash algorithm to determine, if it is more similar to the original picture than the previous best extracted picture (contour). If it is, we set it to be the best extracted picture (*self.best_ image*) for the currently processed image. The *_find_ contours* method code is shown below.

```
1  def _find_contours(self, img, filtered):
2    _, cnts, _ = cv2.findContours(filtered, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
3    cnts = sorted(cnts, key = cv2.contourArea, reverse = True)[:4]
4
5    for contour in cnts:
6      perimeter = cv2.arcLength(contour, True)
7      approx = cv2.approxPolyDP(contour, 0.04 * perimeter, True)
8
9      if len(approx) >= 4:
10       x, y, w, h = cv2.boundingRect(approx)
11       if (self._is_big_enough(img, w, h)):
12         extract = img[y:y+h, x:x+w]
13         if (self._has_best_score(extract)):
14           self.best_image = extract
```

_find_contours

3.1.4 Approaches

We have skipped the *_call _recognition _method* description. The function calls next function based on the method which it gets as a parameter. We will shortly cover the idea of the methods it can call.

Canny edge detection

The first approach was the Canny edge detection technique. The idea was straightforward. The television has a rectangular shape so we expected, that after the edge detection applied, there would be a continuous rectangle. The solution worked when tested against pictures with significant contrast between a television's display and a background. However, when applied to more images we got non applicable results. The biggest problem was that even the small gap in a rectangular outline, made it impossible to extract television area as a contour. The gap usually happened when the color at the edge of the frame was similar to the color of the television frame. The usage of more filters or smoothing also did not improved the results.
To perform edge detection we used *Canny()* function from the opencv library. The function takes image and two thresholds (described earlier in the chapter one) as it's arguments. We were inspired by Adrian Rosebrock's article [2], to determine these threshold values for an arbitrary picture. We set variable *sigma* to 0.33 and calculated median intensity for the given picture using *median* method from *numpy* library. Then

we set the threshold based on these values as described in Rosebrock's article [2]. In the end we call *_find_contours* method to find contours and pick the best image from them.

```
1 def _canny_edge(self, img):
2   sigma = 0.33
3   med = np.median(img)
4   l_bound = int(max(0, (1.0 − sigma) ∗ med))
5   u_bound = int(min(255, (1.0 + sigma) ∗ med))
6   canny = cv2.Canny(img, l_bound, u_bound)
7   self._find_contours(img, canny)
```

_canny_edge

Thresholdings

Next we have tried thresholding techniques. The main advantage is that we could extract the contours easier. It does not make sense to use global thresholding when looking for a general solution, so we tried the Otsu's technique. We get significantly better results compared to non working edge detection. We used opencv *threshold* method with OTSU's flag and we blurred the image using the Gaussian kernel.

```
1 def _otsu(self, img):
2   blur = cv2.GaussianBlur(img, (5, 5), 0)
3   ret, thresh = cv2.threshold(blur, 0, 255,
4                               cv2.THRESH_BINARY+cv2.THRESH_OTSU)
5   self._find_contours(img, thresh)
```

_otsu

We have also tried an adaptive thresholding technique, but it did not gave us very good results. The only difference to *_otsu* method is that we have used *adaptiveThreshold* function with *ADAPTIVE_THRESH_MEAN_C* and *ADAPTIVE_THRESH_GAUSSIAN_C* flags. On the other hand, the adaptive thresholding was still better than the edge detection approach. Finally, we tried an approach which iterates the list of numbers representing chosen global thresholds. The used thresholds values were 50, 70, 90, 110, 130, 150, 170, 190 and 210. So what we did was a global thresholding for each value and then we picked the best result. The best result was again determined by chosen perceptual hash algorithm. We named this approach an iterative solution.

3.1.5 Observation

The best results gave us our iterative solution in combination with the pHash algorithm. Average error was 11.68 frames (maximum error could be 50 because we iterated fifty frames before and after the desired frame). When we considered images with well extracted television, we got 10.94 frames error. The second best results gave us the Otsu's method in combination with the dHash. We had 14.14 error on all frames and 13.31 on only well extracted frames. When we considered only extraction success, the best approach was the Otsu's method with the aHash. We got 89 well extracted pictures and only 11 went wrong. When using the iterative approach, the smallest extraction error gave us the aHash also. It was 15 wrong images. The extraction worked much better for images with a dark background. Using the pHash with the iterative method we got 15 wrong extracted images from 50 images with bright background and only 2 wrong extracted from 50 images with dark background. Adaptive thresholding worked best with the dHash using ADAPTIVE_THRESH_MEAN flag. The error was 18.8 frames error. Using the Canny edge detection we could not even extract the television. After all these experiments with the segmentation techniques and the perceptual hashes, we found out that the accuracy we got was not good enough for the further usage. So we had to look for more precise approach.

3.1.6 Histograms and results

More results can be found on attached DVD. Table 3.1 shows the most relevant. The *error* axis shows distance from the desired position and the *advertising count* axis shows number of frames with this error.

Results		
Method	Error	Bad extracted
Iterate and pHash	11.68	17
Iterate and aHash	13.71	15
Iterate and dHash	12.04	16
Iterate and pHash (only good)	10.94	0
Otsu and pHash	17.68	14
Otsu and aHash	17.63	11
Otsu and dHash	14.14	12
Otsu and dHash (only good)	13.31	0
Adaptive mean and dHash (only good)	18.8	44

Table 3.1: Results for segmenation and percuptual hashes approach.

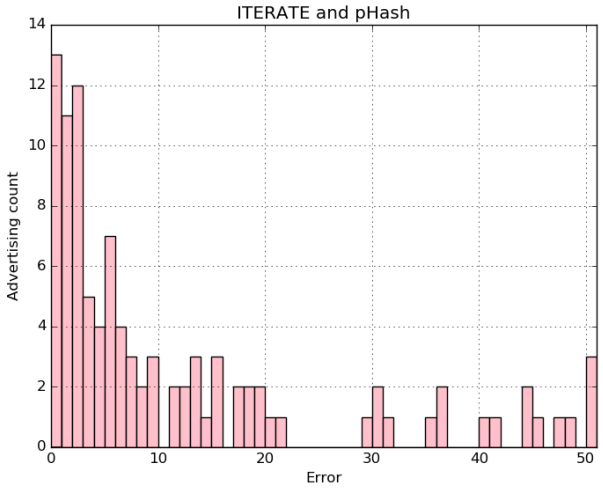


Figure 3.2: Iterative and pHash. Error was 11.68 frames.

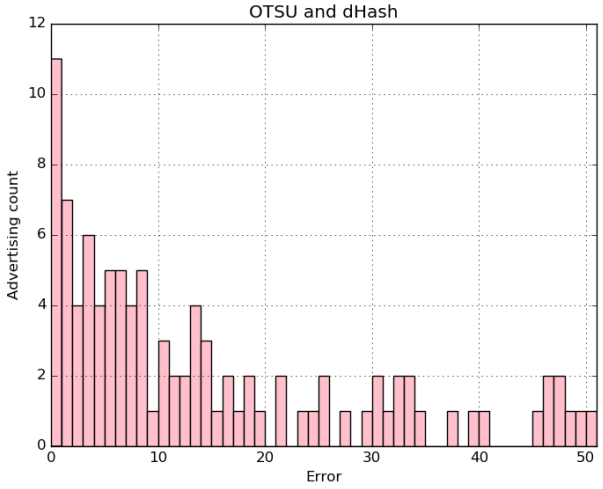


Figure 3.3: Otsu and dHash. Error was 14.14 frames.

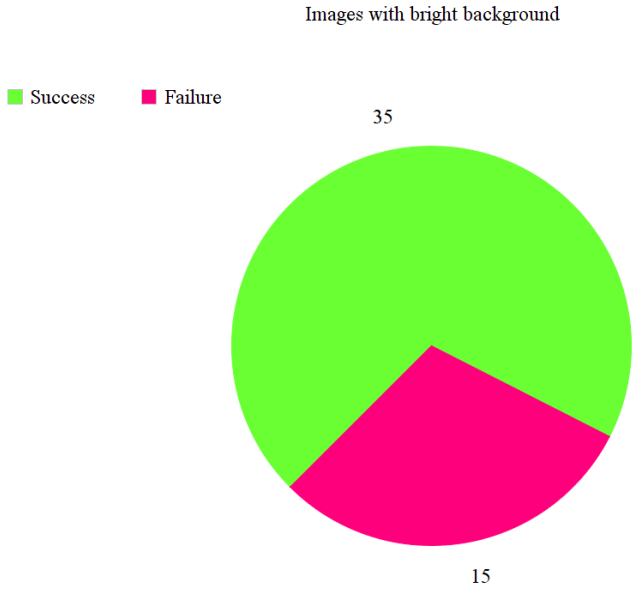


Figure 3.4: Results for frames with bright background.

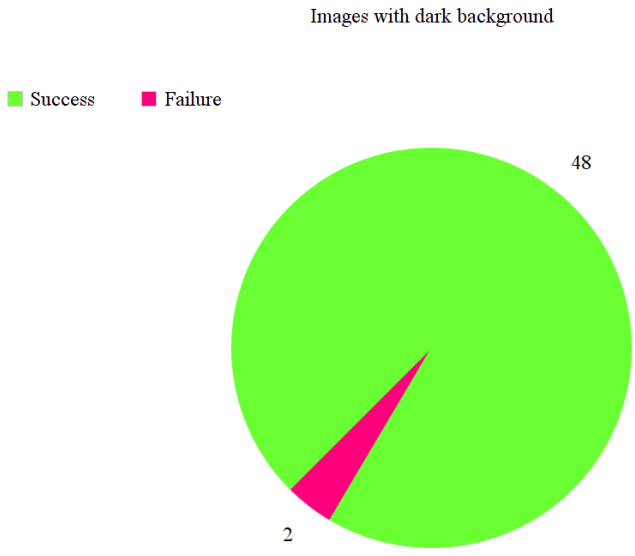


Figure 3.5: Results for frames with dark background.

3.2 Regression approach

After the basic segmentation techniques did not gave us enough accuracy, we decided to try a different approach. Until now we expected the user to take a snapshot when he thinks he sees the desired frame. Now we will expect the user to send a whole video with an original advertisement, containing a piece of a previous and a next advertisement. We also expect the user, to send the time he pressed a button, detecting the desired frame occurrence. The new approach idea is to detect a position of the original advertisement in the user's video. From this information we can easily compute a user's accuracy error.

To solve the problem we will imagine both videos as matrices. We know that a video consist of a number of frames. Let's call this number n for the original video which we will call O. Also let m be the count of frames in the video taken by the user and let's call this video U. Every row in the matrix represents a single frame. It is important to wisely chose columns properties. We will later chose them as a frame properties. For now, let's think of R, G and B properties, from the RGB color space, for every column.

We expect that the original video (O) is shorter than the video taken by the user (U) because the user's video contains piece from a previous and a next advertisement. Of course, it might have more frames because the fps (frames per second) value may vary, but using fps values for both the videos, we can edit the videos to the shape we want. Then we iterate all the possible beginnings of O in U. So we take n frames from the current beginning in the video U and make matrix from them. Let's call the matrix C. We also have matrix for video O of the same rows and columns size. Then we use least-squares calculation (mentioned in Section 2.2.2) to determine the linear coefficients matrix (X), such that the value $((CX) - O)^2$, which represents an error, is the lowest. In other words, we try to map C matrix to O the best we can using X. The position which gave us the smallest error is the searched position.

3.2.1 Data set and testing script

Again we needed to create a data set to test the approach accuracy. We also needed to create another testing script. We collected one hundred user videos, taken from four mobile devices, during different lightness conditions and distances, keeping the condition, that the television covers at least $\frac{1}{20}$ in the screen. We collected them from three different televisions and one laptop. As the original videos, we used the videos from youtube. The folder structure we made has the following format. We created a couple of directories named by a chosen advertisement. In each of these folders we

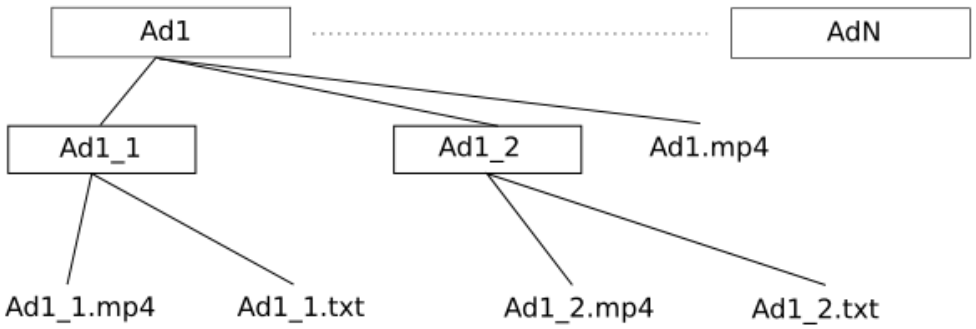


Figure 3.6: Data structure for testing script.

created couple of folders for users videos. They are named the same as the current advertisement postfixed with an identifier. The folder also contains original advertising video, in the mp4 format, named same as the folder. All the folders with users videos, contain a user video advertisement, in the mp4 format, named same as the folder they are placed in. They also contain a text file with the same name as the user video. The file contains two integers on the first line separated by a dash. The numbers represent the position of the original video in the user video. The structure is shown in Figure 3.6. The testing script structure is very similar to the testing script used for the previous approach. We therefore only explain how it works. The script again expect exact folder structure that we have described. Then it iterates all the advertisements and for each it computes the accuracy. In the end, it saves a histogram and a text file with results into a destination folder.
The drawback of this approach is, that it is much slower and more CPU demanding. It takes about twenty minutes to run the testing script for one hundred videos on a basic laptop. We have therefore rewrote the script a little to create more processes. We could then run it on a remote server with more CPUs. We used a python *multiprocessing* module to run an evaluating function as a new process for every advertisement. On a personal computer it is however safer to run one process version because the multiprocess version can freeze the computer until it is finished.

3.2.2 Choosing properties and sections usage

To get the mapping error we use *numpy.linalg.lstsq* function. It's second return argument is a sum of residuals. We provide two matrices (multi dimensional arrays) as it's arguments. Our first idea was to chose red, green and blue components. For every

frames we took a mean of these components which gave us three columns in a matrix and we added next column of ones. Why we add one to every row is explained in the Section 2.2.1.

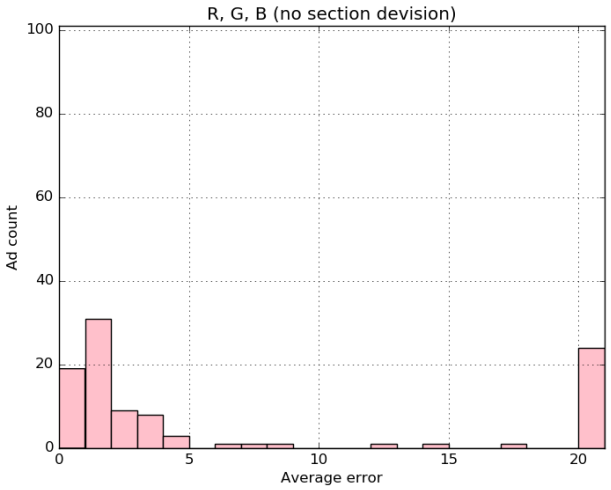


Figure 3.7: Video detection using R,G,B values. 24 videos were mapped with an error bigger than twenty frames. 66 videos were mapped with an error lower than 5 frames.

When we first run the script using R, G, B values we could map 76 videos with an error less than 20 frames. We could map 66 videos with an error less than 5 frames. We considered the count of the videos mapped with an error bigger than 20 frames to be unsatisfying.
Firstly, we found out that there might be a problem with outliers (values which are far away from a regression line when compared to the rest of data). These values occurs when the user shakes the video or there is some sudden change in lightness. To solve this we determined to map videos after sections. Consider an original video of length 700 frames and an user video of length 900 frames. For simplicity we would omit FPS correction. Using mapping of the whole video we would have 200 possible beginnings of the original video in the user video and would map 700 frames to 700 frames for each possible beginning. Now imagine we would map the videos after parts. Let one part be 100 frames. Then we would again iterate all the possible beginnings but instead of mapping all the frames at once we would map first 100 frames of the original video to the first 100 frames of the user video from the current position. Then we map second 100 frames and so on until we reach the end. For each couple of mapped frames, we get some mapping error. In the end, we sum up these errors what give us a final error for all sections. If some unexpected event, for example shaking, happens in some part of the video, it reflects only one or two sections. Figure 3.9 shows the idea of dividing into sections. After applying this strategy the results got better. Only seven videos were mapped with an error bigger than twenty frames and 89 were mapped with an error

less than five frames. However, there were still videos where we obtained wrong results.

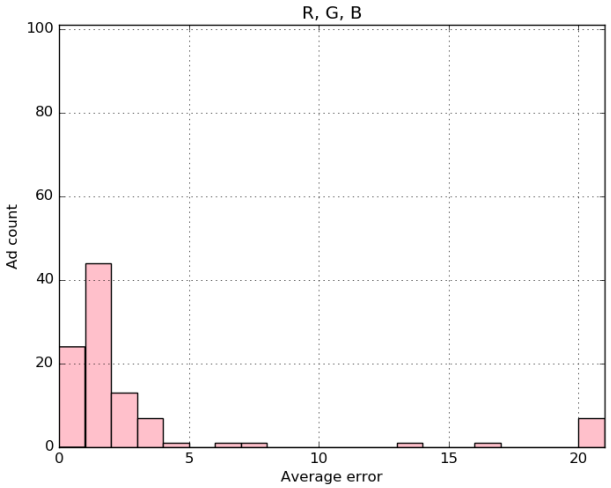


Figure 3.8: Video detection using R,G,B values with division to sections applied. Only 7 videos were mapped with an error bigger than twenty frames and 89 videos were mapped with an error less than five frames.

Red, green and blue channels seemed to be prone to wrong mapping. When looking at a linear combination matrix determined by the OLS the combination coefficients seemed unexpected. It was difficult to determine reasonable relationship between the channels. After trying various normalisations on these channels and not getting better results we determined to try other colour spaces. The next candidates were HSL and HSV. When mapped with HSL or HSV properties we got better results. Then we have started to experiment with selecting the best mapping distance (the length of the sections). We found 70 frames to be the best length for our purpose. There was no big difference between 30-80 frames. We then tried various combinations of color properties.

We were sceptic about the hue property because of it's shape. The hue is circular so the distance between the values is meaningful unless we cross origin (0 angles). Consider hue values to be 50 and 70 for two frames. Then their distance is 20 frames. But if we chose two frames which hue mean is 10 and 350 (we consider the hue to be in a range from 0 to 359) then their distance should be only 20, but in fact is 340. This is why we have though that when we substitute the hue with some of it's transformation, we could get even better results. We have tried using *sin* and *cos* transformations. However, the results did not improved. Soon we found out that using *sin* or *cos* we just move the hue property problem to different position. On one hand we have nice transit between end points (0, 359), but from *sin* and *cos* nature opposite values, which should have the greatest distance, are considered the same. When we found out that

sin and *cos* do not improve the results, we have tried to divide the hue into three values. First described distance from 0 point in the hue, second distance from 120 point and the last distance from 240. We have combined these properties with the saturation and the value/lightness property from HSV or HSL. Actually, this solution gave as the worst results, what was quite unexpected. The last boost we tried, was usage of a kernel regression instead of the linear. However, the computation was even longer and the results did not improved, so we dropped this approach. Finally, we have realised that the results which we had with the hue were good enough and terminated the experiments. The best working properties were the hue, intensity, saturation and the value from the HSV color space. For these properties we had only three videos mapped with an error bigger than twenty frames and 94 videos were mapped with an error lower than five frames. Compared to the results achieved with the same properties, but without dividing into sections, we got 13 frames mapped with an error more than twenty frames.

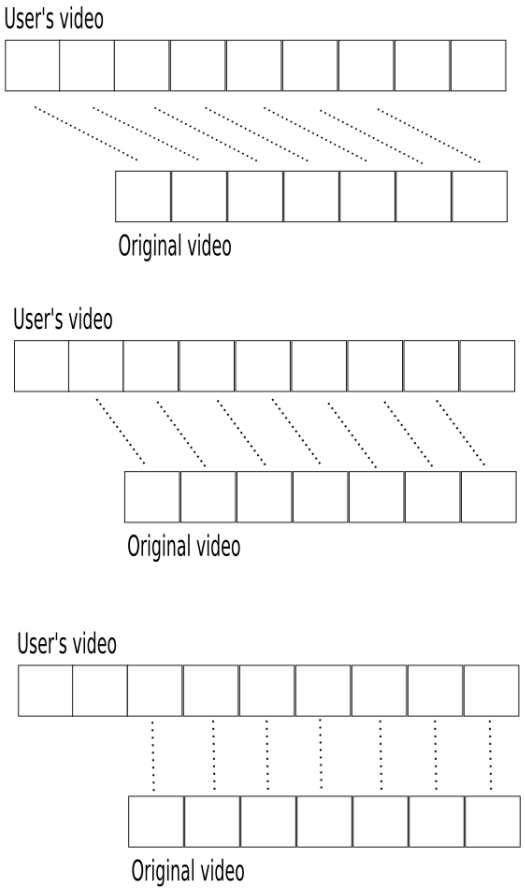


Figure 3.9: Figure shows a process of mapping two videos when using dividing into sections. Rectangles represent these sections and dashed lines represents mapping.

3.2.3 Algorithm integration

This section deals with integrating the algorithm into the application. The algorithm code with a tester script code is located on the server. We provide two methods for later calculations. They are *resolve* and *preprocess* methods. The first method gets an original video and an output folder as it's arguments. It decomposes the video into JSON data, that it stores in the output folder. The JSON contains fps field and props field. In the fps field there is a single float which represents the fps rate. In the props field there is a matrix containing the video properties used for later mapping. We get those properties by iterating the frames and for each frame taking the mean of it's pixels properties.

More complicated work is performed by the *resolve* function. It gets an user video, path to the original preprocessed video and a string array of times as it's arguments. The times represents the times when the user pressed the button detecting the desired frame. We allow to pass more values in this array because the user may potentially label more than one frame. What we do in this method, is that we get the preprocessed JSON file. Then we process our current video the same way as we did for the original video. We have to normalise these videos because they potentially differ in the fps rate. We rescale one of those videos so both the videos behave like they had same fps rate. If the video being rescaled had lower fps rate than the other video we duplicate some frames, otherwise we remove some. Then we try all the possible positions of the original video in the user video, pick the best position and calculate distances from the desired position for each time and send those values back.

In this point, we have all our basic tools ready to be used in the application. We have also created REST like API to be used by mobile devices to communicate with the server. Our functions can be used the way, that before the user has a possibility to send the videos to the server, we preprocess all the original videos. Later we call *resolve* on each valid request we get. We also set some time intervals when user can or can not post videos. The application expects many requests in the same time so we have integrated Celery in it. The Celery is an asynchronous task queue job manager. Using it we can process many request in same time and after they all ends we have results stored in the database. For testing purposes we have only used SQLite database. Django REST Framework was used on the server. Details of these approaches are beyond the scope of the thesis.

3.2.4 Histograms and results

The best results we got when combining hue, saturation and value properties from the HSV color space and the intensity. We set sections to be 70 frames. Other results can be found in the Table 3.2. We reduced the scale of an error from fifty to twenty frames when compared to the segmentation technique approach. We consider every video with an error more than twenty frames to be mapped absolutely wrong. Therefore we wanted all the videos with the error more that twenty frames to have the same weights. Average error was calculated as a maximum distance from start and end values of an original video in a user video, compared to values which were determined by our algorithm. So for example if the original video occurred from 20 to 700 frame in the user video and our algorithm determined it to be from 15 to 702 frame, then the maximum distance would be 5 frames. This value is divided by two and rounded down. We divide it by two because usually there was some fps rounding error and even if the videos were mapped properly, the beginning and the end positions determined by the algorithm, form a little shorter or a little longer interval than the original video does. Therefore the error varies as we iterate this interval. It is the smallest it the middle and the largest at the interval edges. The example we used above shows this situation. Because this small rounding error happens in many cases and we do not have many videos mapped wrong, we can divide the maximum distance by two, so the error should better reflect the reality.
Below is a comparison of the best results for the segmentation and the regression approach with the same error scale set to twenty frames. The regression approach gave us an average error of 1.81 frames while the iterative solution with pHash had the error 10.6 frames for the same scale. We run the iterative solution on the whole video length, divided the errors by two and then rounded down so it can be fully comparable with the regression approach. Both histograms are made for data set of one hundred samples. From these values it is clear, that the regression approach, focused on the video in video detection, is much more accurate than our first approach.

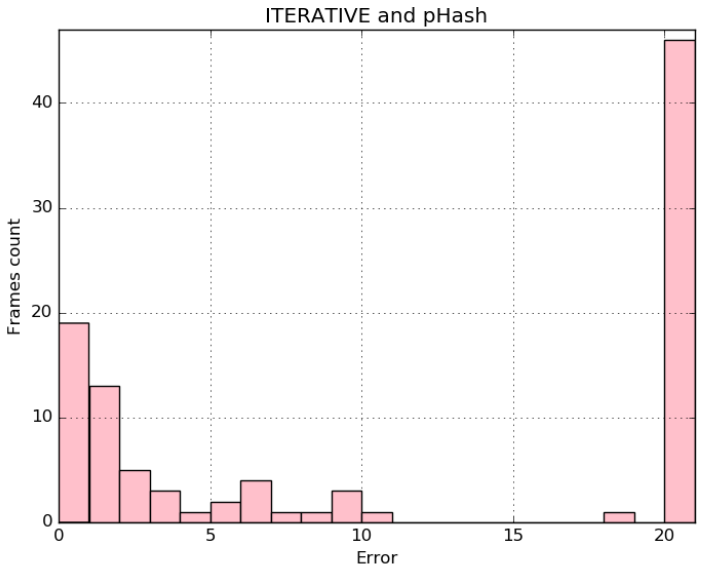


Figure 3.10: Best image processing approach results with an error scale of twenty frames. All frames in videos were iterated (not only 50), the error was divided by two and rounded down to be fully comparable with the linear regression approach. Average error was 10.6 frames.

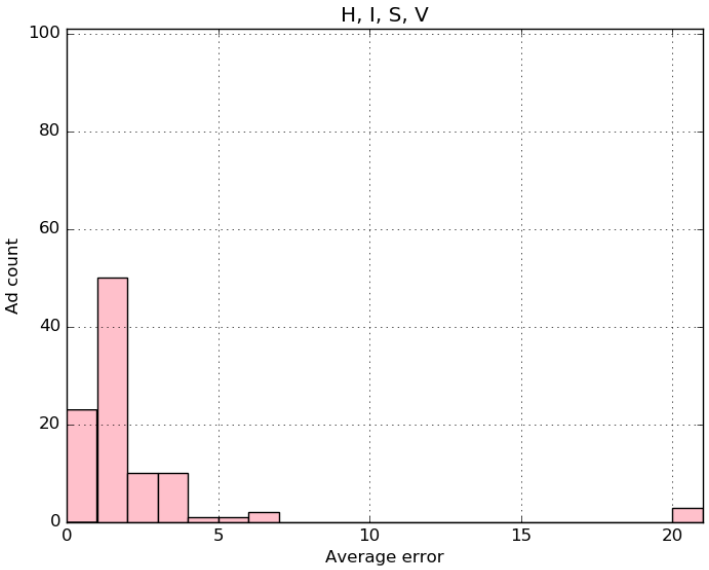


Figure 3.11: Best linear regression approach results. Average error was 1.81 frames.

Results	
Properties	Average error
h, i, s, v	1.81
r, g, b, s, v	1.92
cos(h), i, s, l	2.01
i, s, v	2.25
h, i, s, l	2.28
h, s, l	2.29
cos(h), i, s, v	2.29
cos(h), s, v	2.37
cos(h), s, l	2.42
h, s, v	2.47
r, g, b	2.77
sin(h), cos(h), s, v	3.1
sin(h), s, v	3.19
s, l	3.5
s, v	3.63
h1, h2, h3, s, v (abs)	18.33
h1, h2, h3, s, v	19.02

Table 3.2: Results for the linear regression approach.

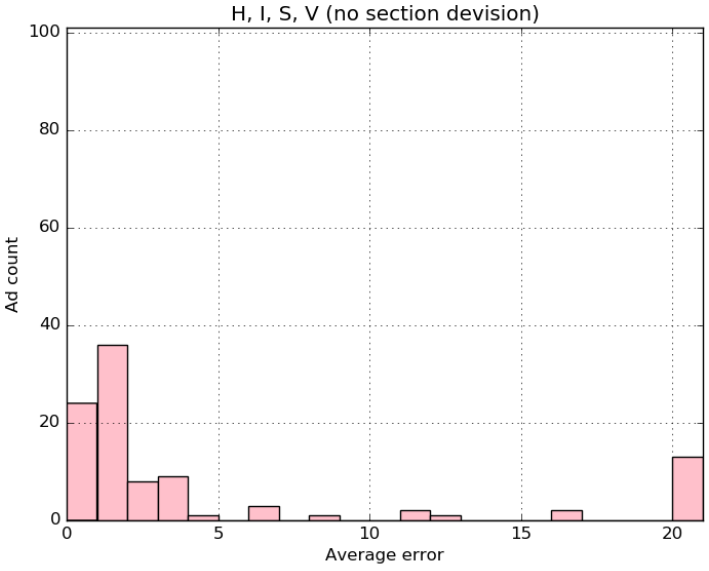


Figure 3.12: H,I,S,V without dividing into sections. Average error was 4.35 frames.

Conclusion

We have found that applying basic image processing techniques like thresholding and edge detection, in combination with perceptual hashes, might not always be the best approach for recognition purposes. Mostly in the cases where accuracy is the most important aspect. From our experiments, it seems that using Canny edge detection for extraction of an object, during arbitrary conditions, almost always leads to failure. On the other hand, thresholding gave us better results. Especially Otsu's and self constructed iterative approach. It may be useful to try iterative method with perceptual hash to locate a desired object.

After obtaining unsatisfying results with segmentation techniques, we have tried to solve the problem using ordinary least-squares calculation. Even if linear regression is the basic machine learning technique, we got better results with it. We have shown that using only red, green and blue channels might not be desirable and that the alternative models (HSL, HSV), and more models combinations, are better for matching purposes. We have also found that division into sections boosted the accuracy.

We believe that a summary of the approaches we tried, with their accuracy results, can be used as a basis for similar works. Details about various combinations of color properties, can be handy when choosing correct color properties for arbitrary problem using regression. Division into sections may also be a useful approach to try later because it can improve the linear regression results.

Bibliography

[1] Organizing color. Retrieved April 19, 2016, from http://viz.aset.psu.edu/gho/sem_notes/color_2d/html/primary_systems.html.

[2] Adrian Rosebrock. Zero-parameter, automatic canny edge detection with python and opencv. Retrieved March 29, 2016, from http://www.pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/.

[3] Bill Green. Canny edge detection tutorial. Retrieved January 21, 2016, from http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_weg22/can_tut.html.

[4] Daniel Rákoz. Efficient gaussian blur with linear sampling. Retrieved January 23, 2016, from http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/.

[5] Dr. Andrew Greensted. Otsu thresholding. Retrieved January 22, 2016, from http://www.labbookpages.co.uk/software/imgProc/otsuThreshold.html.

[6] Noor A. Ibraheem et al. Understanding color models: A review. *ARPN Journal of Science and Technology*, 2(3):i265–i275, 2012.

[7] hackerfactor.com. Kind of like that. Retrieved January 27, 2016, from http://www.hackerfactor.com/blog/?/archives/529-Kind-of-Like-That.html.

[8] hackerfactor.com. Looks like it. Retrieved January 27, 2016, from http://www.hackerfactor.com/blog/?/archives/432-Looks-Like-It.html.

[9] Marius Karthaus. Javascript implementation of the ramer douglas peucker algorithm. Retrieved January 24, 2016, from http://karthaus.nl/rdp/.

[10] Numpy developers. Numpy documentation. Retrieved November 9, 2015, from http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linalg.lstsq.html.

[11] Opencv dev team. Opencv documentation. Retrieved January 25, 2016, from `http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_tutorials.html`.

[12] R. Fisher et al. Gaussian smoothing. Retrieved January 21, 2016, from `http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm`.

[13] Richard E. Woods Rafael C. Gonzalez. *Digital Image Processing, Second Edition.* Prentice Hall, 2002.

[14] Sanford Weisberg. *Applied Linear Regression, Third Edition.* WILEY, 2005.

Appendix A

Other chosen histograms for image processing approach. Scale for an error is from zero to fifty frames. *Advertising count* is a number of frames with the given error. *Good* flag means, that the error was measured only on images with well extracted television.

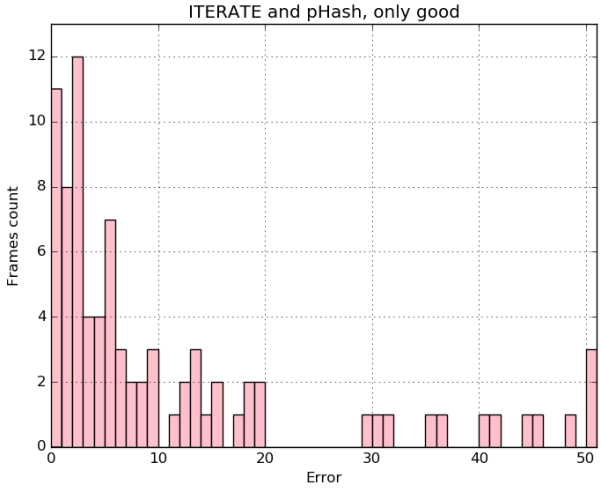


Figure 3.13: Error was 10.94.

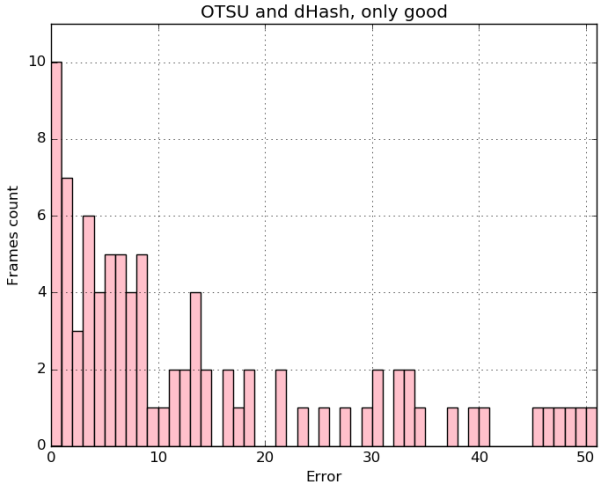


Figure 3.14: Error was 13.31.

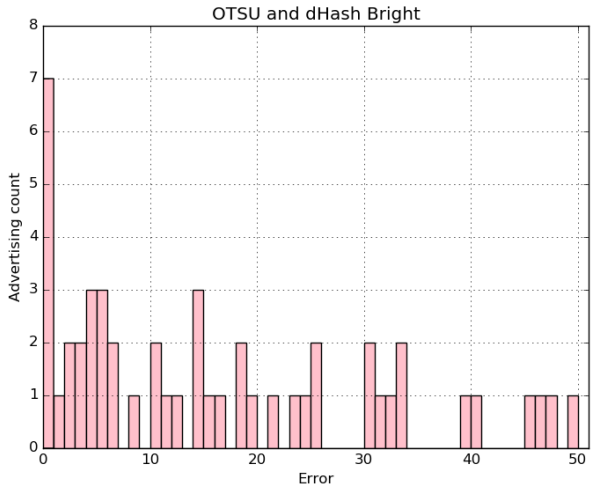


Figure 3.15: Error was 16.04.

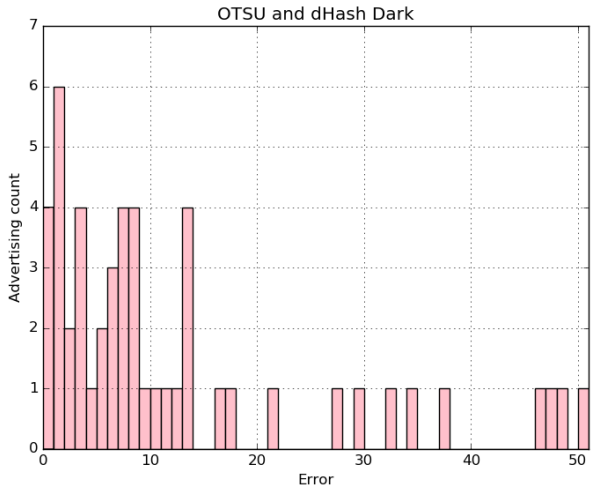


Figure 3.16: Error was 12.24.

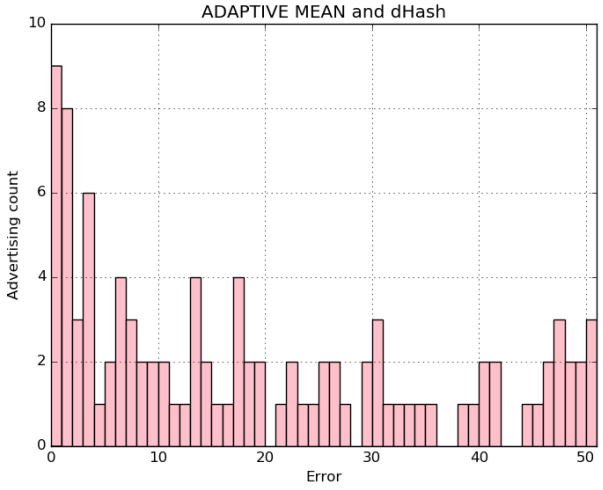


Figure 3.17: Error was 18.8.

Appendix B

Other chosen histograms for linear regression approach. Scale for an average error is from zero to twenty frames. Every error which was more than twenty frames was set to be twenty. *Ad count* is a count of an advertisements with the given average error.

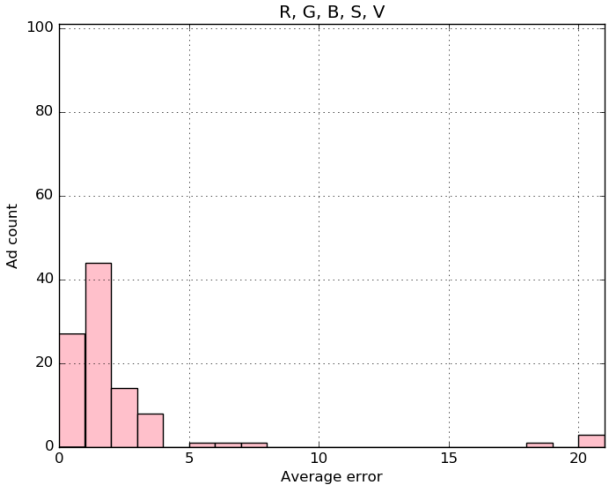


Figure 3.18: Average error was 1.92 frames.

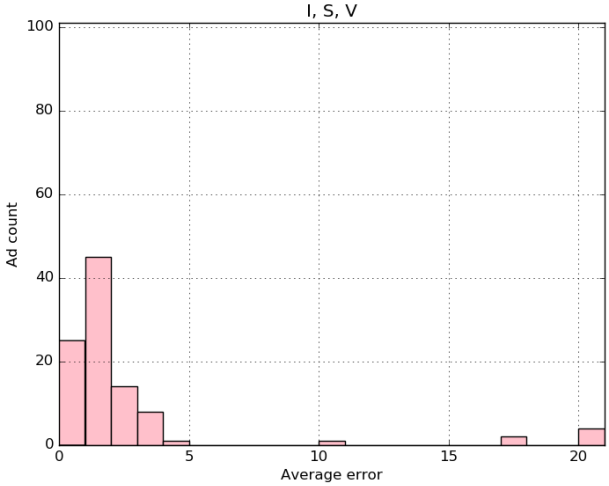


Figure 3.19: Average error was 2.25 frames.

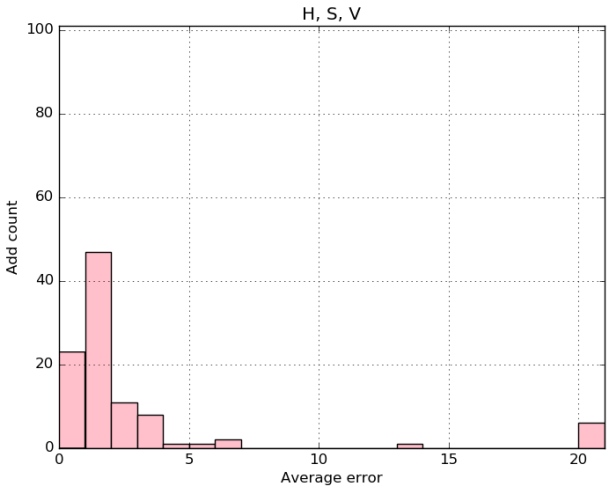


Figure 3.20: Average error was 2.47 frames.

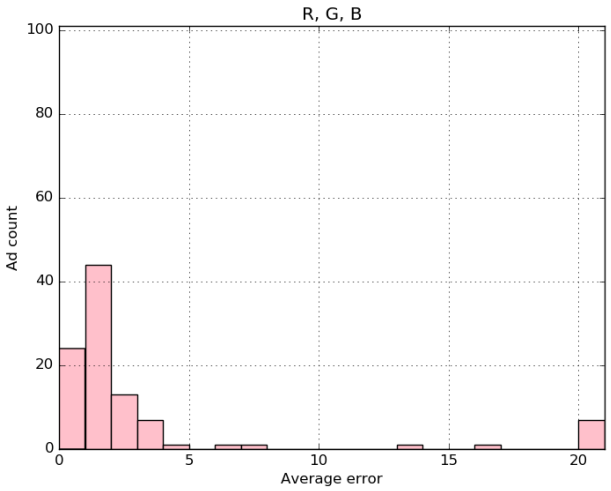


Figure 3.21: Average error was 2.77 frames.

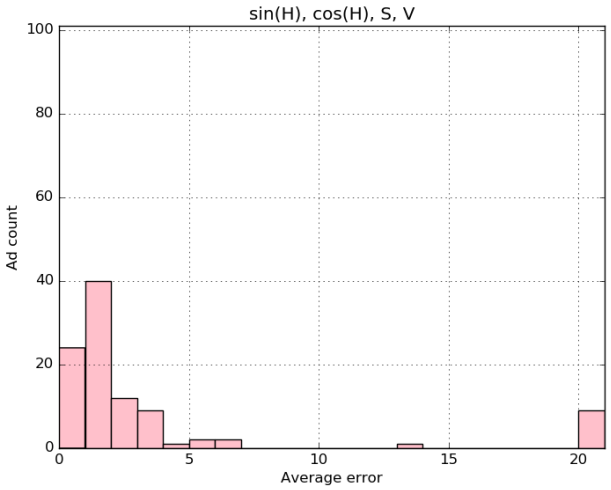


Figure 3.22: Average error was 3.1 frames.

Appendix C

Attached DVD contains application source code for the perceptual hash approach, data sets and results for both approaches.