WARSAW UNIVERSITY OF TECHNOLOGY

Image and Speech Recognition

Faculty of Mathematics and Information Science

Social Distance Monitoring

- Final Report -

By **Elie SAAD & Matthieu Dabrowski**

May 7, 2021

Contents

1 Definitions

Definition 1.1 (Projectivity)**.** A projectivity - also called a collineation, a projective transformation or a homography - is an invertible mapping $h \in \mathbb{P}^2$ from $\mathbb{P}^2$ to itself such that three points x_1, x_2, x_3 lie on the same line if and only if $h(x_1), h(x_2)$, and $h(x_3)$ do.

Definition 1.2 (Projective Transform)**.** A planar projective transformation is a linear transformation on homogeneous 3-vectors represented by a non-singular 3×3 matrix

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \end{pmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

more briefly, $x' = Hx$.

2 Theoretical Background

2.1 Homography Estimation

We know that any 2-d point $x = (x, y)$ in a plane can be represented as a 3-d vector $x = (x_1, x_2, x_3)$ where $x = \frac{x_1}{x_3}, x_n = \frac{x_2}{x_3}$ and $x_3 \neq 0$. This is called the homogeneous representation of a point and it lies on the projective plane $\mathbb{P}^2$. Using definition 1.1 and a theorem that states that a mapping from $\mathbb{P}^2 \implies \mathbb{P}^2$ is a projectivity if and only if there exists a non-singular 3×3 matrix, H, such that for any point in $\mathbb{P}^2$ represented by vector x it is true that its mapped point equals Hx, tells us that in order to calculate the projective transformation (or the homography) that maps each x_i to its corresponding x'_i, we need to calculate the 3×3 homography matrix, H.

Thus, this reduces our task to the following problem: Given a set of 3-d vector correspondences, x_i and x'_i, we want to solve for the 3×3 matrix H such that for each x_i

$$x'_i = Hx_i \tag{1}$$

It should be noted that H can be changed by multiplying by an arbitrary nonzero constant without altering the projective transformation. Thus H is considered a homogeneous matrix and only has 8 degrees of freedom even though it contains 9 elements. This means there are 8 unknowns that need to be solved for.

2.1.1 Direct Linear Transform Using Point Correspondences

The Direct Linear Transform (DLT) algorithm is a simple algorithm used to solve for H given a sufficient set of point correspondences. Since we are working in homogeneous coordinates, equation (1) can be rewritten as

$$c \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = H \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{2}$$

where c is any non-zero constant, $(u, v, 1)^T$ represents x', $(x, y, 1)^T$ represents x, and $H = \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{pmatrix}$.

Dividing the first row of equation (2) by the third row and the second row by the third row we get the following two equations

$$-h_1x - h_2y - h_3 + (h_7x + h_8y + h_9)u = 0 \tag{3}$$
$$-h_4x - h_5y - h_6 + (h_7x + h_8y + h_9)u = 0 \tag{4}$$

Equations (3) and (4) can be rewritten in matrix form as

$$A_i h = 0$$

where

$$A_i = \begin{pmatrix} -x & -y & -1 & 0 & 0 & 0 & ux & uy & u \\ 0 & 0 & 0 & -x & -y & -1 & vx & vy & v \end{pmatrix}$$

and $h = (h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9)^T$. Since each point correspondence provides 2 equations, 4 correspondences are sufficient to solve for the 8 degrees of freedom of H. The restriction is that no 3 points can be collinear (i.e., they must all be in "general position"). Four 2×9 A_i matrices (one per point correspondence) can be stacked on top of one another to get a single 8×9 matrix A. The 1-d null space of A is the solution space for h. If all of the point correspondences are exact, A will still have rank 8 and there will be a single homogeneous solution no matter how many additional correspondences are found.

In practice, there will be some uncertainty, the points will be inexact and there will not be an exact solution. The problem then becomes to solve for a vector h that minimizes a suitable cost function. But we ignore this for our solution since our task is rather comparatively simple. The solution to this

problem is the unit singular vector corresponding to the smallest singular value of A. This can be found using Singular Value Decomposition (SVD) analysis, which is a very well known algorithm from Linear Algebra and for that reason we have decided not to cover it in this report. We will select the points later on in the report in the experimentation section.

2.2 Histogram of Oriented Gradients as a Feature Descriptor

To understand what the Histogram of Oriented Gradients (HOG) do, we must first understand what a feature descriptor is. A feature descriptor is a representation of an image or an image patch that simplifies the image by extracting useful information and throwing away extraneous information. In other words, a feature descriptor is a simplified representation of the image that contains only the most important information about the image. A feature descriptor typically converts an image of size $width \times height \times 3$ - where 3 is the number of channels - to a feature vector of length n.

In case of HOG descriptor, the input image is of size $64 \times 128 \times 3$ and the output feature vector is of length 3780. We do note that the HOG descriptor can be calculated for other sizes, but we will be sticking with the size that has been described in the original paper.

HOG is a feature descriptor that can be used to extract features from image data. To understand what HOG is, let us first look at some important aspects of HOG that makes it different from other feature descriptors:

- The HOG descriptor focuses on the structure or the shape of an object. Now, how is this different from the edge features we extract for images? In the case of edge features, we only identify if the pixel is an edge or not. HOG is able to provide the edge direction as well. This is done by extracting the gradient and orientation (in other words, magnitude and direction) of the edges.

- Additionally, these orientations are calculated in 'localized' portions. This means that the complete image is broken down into smaller regions and for each region, the gradients and orientation are calculated. We will discuss this in much more detail in the upcoming sections.

- Finally the HOG would generate a Histogram for each of these regions separately. The histograms are created using the gradients and orientations of the pixel values, hence the name 'Histogram of Oriented Gradients'

To give the HOG a formal definition: The HOG feature descriptor counts the occurrences of gradient orientation in localized portions of an image.

Now, what kinds of "features" are useful for classification tasks? Suppose we want to build an object detector that detects buttons of shirts and coats. A button is circular (may look elliptical in an image) and usually has a few holes for sewing. An edge detector can be ran on the image of a button, and easily tell if it is a button by simply looking at the edge image alone. In this case, edge information is "useful" and color information is not. In addition, the features also need to have discriminative power. For example, good features extracted from an image should be able to tell the difference between buttons and other circular objects like coins and car tires.

In the HOG feature descriptor, the distribution (histograms) of directions of gradients (oriented gradients) are used as features. Gradients (x and y derivatives) of an image are useful because the magnitude of gradients is large around edges and corners (regions of abrupt intensity changes) and we know that edges and corners pack in a lot more information about object shape than flat regions.

2.2.1 Preprocessing of the Data

As mentioned earlier HOG feature descriptor used for pedestrian detection is calculated on a 64×128 patch of an image. Of course, an image may be of any size. Typically patches at multiple scales are analyzed at many image locations. The only constraint is that the patches being analyzed have a fixed aspect ratio. In our case, the patches need to have an aspect ratio of $1 : 2$. This is because we will be dividing the image into 8×8 and 16×16 patches to extract the features. The paper [2] mentions gamma correction as a preprocessing step, but the performance gains are minor and so we are skipping the step.

2.2.2 Calculation of the Gradients

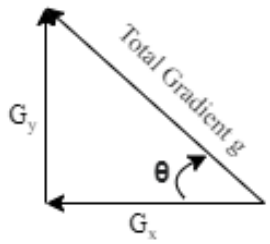
The next step is to calculate the gradient for every pixel in the image. Gradients are the small change in the x and y directions. Now, to determine the gradient (or change) in the x-direction, we need to subtract the value on the left from the pixel value on the right. Similarly, to calculate the gradient in the y-direction, we will subtract the pixel value below from the pixel value above the selected pixel. This can also be easily achieved by filtering the

image with the following kernels: $\begin{pmatrix} -1 & 0 & -1 \end{pmatrix}$ and $\begin{pmatrix} -1 \\ 0 \\ -1 \end{pmatrix}$. This process will give us two new matrices – one storing gradients in the x-direction and the other storing gradients in the y-direction. This is similar to using a Sobel Kernel of size 1. The magnitude would be higher when there is a sharp change in intensity, such as around the edges.

We have calculated the gradients in both x and y direction separately. The same process is repeated for all the pixels in the image. The next step would be to find the magnitude and orientation using these values.

2.2.3 Calculating the Magnitude and Orientation

Using the gradients we calculated in the last step, we will now determine the magnitude and direction for each pixel value. For this step, we will be using the Pythagoras theorem.



The gradients are basically the base and perpendicular here. Thus, applying the Pythagoras theorem to calculate the total gradient magnitude gives us

$$g = \sqrt{G_x^2 + G_y^2}.$$

Next, we calculate the orientation (or direction) for the same pixel. We know that we can write the tangent for the angles as

$$\tan(\theta) = \frac{G_y}{G_x},$$

hence, we get the value of the angle as

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

So now, for every pixel value, we have the total gradient (magnitude) and the orientation (direction). We need to generate the histogram using these gradients and orientations.

2.2.4 Calculating the Histogram of Gradients

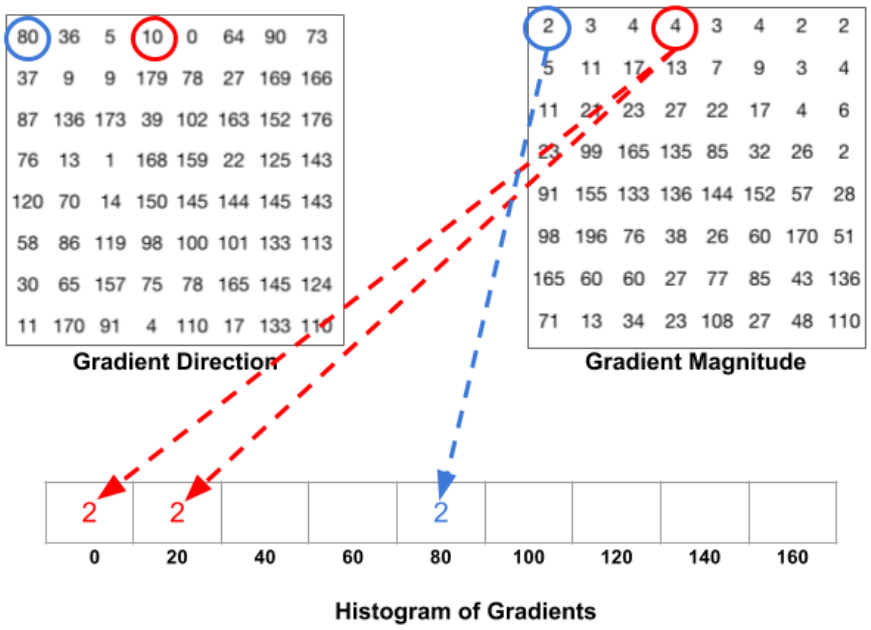
In this step, the image is divided into 8×8 cells and a histogram of gradients is calculated for each 8×8 cells. One of the important reasons to use a feature descriptor to describe a patch of an image is that it provides a compact representation. An 8×8 image patch contains $8 \times 8 \times 3 = 192$ pixel values. The gradient of this patch contains 2 values (magnitude and direction) per pixel which adds up to $8 \times 8 \times 2 = 128$ numbers.

But why 8×8 patch? Why not 32×32? It is a design choice informed by the scale of features we are looking for. HOG was used for pedestrian detection initially. 8×8 cells in a photo of a pedestrian scaled to 64×128 are big enough to capture interesting features (e.g. the face, the top of the head etc.).

The histogram is essentially a vector of 9 bins (numbers) corresponding to angles $0, 20, 40, 60, ..., 160$. The representation of the gradients in the 8×8 cells is done where the angles are between 0 and 180 degrees. These are called "unsigned" gradients because a gradient and it's negative are represented by the same numbers. In other words, a gradient arrow and the one 180 degrees opposite to it are considered the same. But, why not use the 0 - 360 degrees? Empirically it has been shown that unsigned gradients work better than signed gradients for pedestrian detection.

The next step is to create a histogram of gradients in these 8×8 cells. The histogram contains 9 bins corresponding to angles $0, 20, 40, ..., 160$. The following figure illustrates the process. We are looking at magnitude and direction of the gradient of the same 8×8 patch as in the previous figure. A bin is selected based on the direction, and the vote (the value that goes into the bin) is selected based on the magnitude. Let us first focus on the pixel encircled in blue. It has an angle (direction) of 80 degrees and magnitude of 2. So it adds 2 to the 5th bin. The gradient at the pixel encircled using red has an angle of 10 degrees and magnitude of 4. Since 10 degrees is half way between 0 and 20, the vote by the pixel splits evenly into the two bins.

There is one more detail to be aware of. If the angle is greater than



160 degrees, it is between 160 and 180, and we know the angle wraps around making 0 and 180 equivalent. So, the pixel with angle 165 degrees contributes proportionally to the 0 degree bin and the 160 degree bin. The contributions of all the pixels in the 8×8 cells are added up to create the 9-bin histogram.

2.2.5 Block Normalization

Although we already have the HOG features created for the 8×8 cells of the image, the gradients of the image are sensitive to the overall lighting. This means that for a particular picture, some portion of the image would be very bright as compared to the other portions. We cannot completely eliminate this from the image. But we can reduce this lighting variation by normalizing the gradients by taking 16×16 blocks.

Here, we will be combining four 8×8 cells to create a 16×16 block. And we already know that each 8×8 cell has a 9×1 matrix for a histogram. So, we would have four 9×1 matrices or a single 36×1 matrix. To normalize this matrix, we will divide each of these values by the square root of the sum of squares of the values. Thus, for a given vector $v = (a_1, a_2, ..., a_{36})$, we calculate the root of the sum of squares $k = \sqrt{a_1^2 + ... + a_{36}^2}$, and divide

all the values in the vector v by k as follows

$$\text{Normalized } v = \left(\frac{a_1}{k}, ..., \frac{a_{36}}{k}\right),$$

the resultant would be a normalized vector of size 36×1.

2.2.6 Calculating the HOG feature vector

We are now at the final step of generating HOG features for the image. So far, we have created features for 16×16 blocks of the image. Now, we will combine all these to get the features for the final image. We would have 105 of (7×15) blocks (7 horizontal and 15 vertical positions making a total of 105) of 16×16. Each of these 105 blocks has a vector of 36×1 as features. Hence, the total features for the image would be $105 \times 36 \times 1 = 3780$ features.

2.3 Linear Support Vector Machine as a Classifier

To explain what a Linear Support Vector Machine (SVM) is, we must first begin by explaining what support vectors are. Support vectors are defined as data points that lie closest to the decision line, place, or hyperplane (depending on the dimension in which the support vectors belong to). In our case, the feature vectors that will be represented by the SVM are of 3780-d, and thus, our support vectors are those vectors that are closest to the decision hyperplane. Support vectors are usually the hardest to classify, since they fall directly close to the decision hyperplane, and they have a direct impact on the selection of the decision hyperplane itself since they will be taken into consideration by the SVM while selecting the decision hyperplane.

An SVM is thus defined as a learning algorithm that maximises the distance between all of the support vectors and the decision hyperplane. The process of finding the decision hyperplane is known as a quadratic programming problem, where we are trying to solve a linear quadratic function. As discussed with the professor, we assumed linear seperability, and thus, we used a linear SVM which we will explain its working in the 2-d rather than in its actual hyperspace - for obvious reasons i.e. representation.

To train the SVM algorithm, it takes in a set of training pairs (input, output): where the input are feature vectors $x_1, ..., x_n$ where $n \in \mathbb{N}^+ - \{0\}$ is the number of training feature vectors (those vectors are taken from the HOG output), and the output is the result y. The output is a vector

of set of weights w, each element w_i corresponds to a feature x_i, whose linear combination predicts the value y. The optimization - that is, the maximization of the distance between the support vectors and the decision line - is used to reduce the number weights w_i that are nonzero to just a few that correspond to the important features that matter in deciding the separating line. Thus, these nonzero weights correspond to the support vectors.

Given the 2-d case, we need to find a decision line that satisfies

$$ax + by \geq c \text{ for the positive examples,}$$
$$ax + by < c \text{ for the negative examples.}$$

It is intuitive that there are an infinite number of solutions for a, b, and c to the above system. And thus, the problem of finding the optimal line (or in our case, the hyperplane) is an optimization problem and can be solved by optimization techniques. We use the Lagangian multipliers to convert this problem into a form that can be solved analytically.

Based on the definition of the support vectors, we can be confident in selecting the vectors that are on the boundary of the margin between the support vectors and the decision line, as shown in figure 1

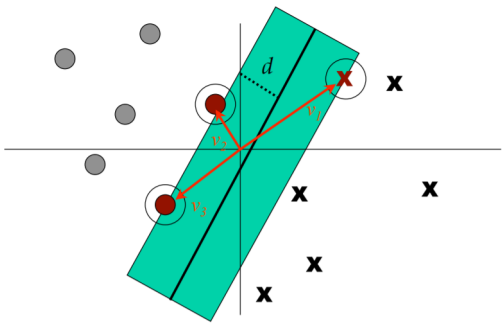


Figure 1: Support vectors represented in 2-d.

We take the boundary to be of width 1, thus the support vectors satisfy the relation

$$w_0^T x + b_0 = 1 \text{ or } w_0^T x + b_0 = -1,$$

and since, in our case, we took the width of the margin to be equal to 1, then $d = \frac{1}{2}$.

In our solution, we have the following definitions that apply:

- The hyperplanes are defined as:

$$w \cdot x_i + b \geq +1 \text{ when } y_i = +1$$
$$w \cdot x_i + b \leq -1 \text{ when } y_i = -1$$

- H_1 and H_2 are the planes:

$$H_1 : w \cdot x_i + b = +1$$
$$H_2 : w \cdot x_i + b = -1$$

- The plane H_0 is the median in between H_1 and H_2 where $w \cdot x_i + b = 0$.
- The equation defining the decision surface separating the two classes (humans and non-humans) is a hyperplane of the form $w^T x + b = 0$, where w is the weights vector, x is the input feature vector, and b is the bias vector. And thus we took

$$w^T x + b \geq 0 \text{ for } d_i = +1$$
$$w^T x + b < 0 \text{ for } d_i = -1$$

- The total distance between H_1 and H_2 is $\frac{2}{||w||}$ can be derived.

With all of the background and intuition covered, we are now ready to explain the method used in maximizing the margin separating our two classes.

2.3.1 Maximizing the Margin

To maximize the margin, it is enough to maximize $||w||$ (can be seen from the distance between H_1 and H_2 from our definitions) with the constraint that no datapoints are between H_1 and H_2. Thus, we combine

$$x_i \cdot w + b \geq +1 \text{ when } y_i = +1$$
$$x_i \cdot w + b \leq -1 \text{ when } y_i = -1$$

into

$$y_i(x_i \cdot w) \geq 1.$$

And thus, our problem is reduced to solving a quadratic programming problem: we need to minimize $||w||$ such that the boundary constraint is obeyed, and so our function minimization problem becomes $\min f(x)$ s.t. $g(x) = 0$, which can be rewritten as

$$\min f : \frac{1}{2}w^2 \text{ w.r.t. } g : [y_i(w \cdot x_i) - b] - 1 = 0.$$

Therefore, to solve this constraint optimization problem, we use the Lagrangian multipler method. We do note that because our function f is quadratic with superimposed constraint g, it forms a paraboloid with a single global minimum. And we know from calculus that the gradients of f and g, at a tangent solution p, are parallel, creating another constraint. And thus, we are left with the two constraints: (i) the parallel normal constraint, (ii) and $g(x) = 0$. Thus we want to find a solution to point p

$$\nabla f(p) = \nabla \lambda g(p) \text{ w.r.t } g(x) = 0,$$

or simply applying the Lagrangian as

$$L(x, a) = f(x) - ag(p) \text{ w.r.t. } \nabla(x, a) = 0,$$

and then generalizing it

$$L(x, a) = f(x) - \sum_i a_i g_i(p)$$

which is a function of $n + m$ variables (m for the a's), where $f(x)$ is the gradient minimum of f, and $\sum_i a_i g_i(p)$ is the constraint condition g. Thus after differentiation, we get $n+m$ equations: differentiating the n equations with respect to the x_i's gives the gradient condition, while differentiating the m equations with respect to the a_i's recover the constraints g_i. In our case, the Lagrangian is thus

$$\min L = \frac{1}{2}||w||^2 - \sum_i^l a_i[y_i(w \cdot x_i + b) - 1] \text{ w.r.t } w, b$$

$$\min L = \frac{1}{2}||w||^2 - \sum_i^l a_i y_i(w \cdot x_i + b) + \sum_i^l a_i \text{ s.t } \forall i, a_i > 0$$

where l is the number of training points. And finally, from the property that at $\min = 0$, we get

$$\frac{\partial L}{\partial w} = w - \sum_i^l a_i y_i x_i = 0 \text{ and } \frac{\partial L}{\partial b} = \sum_i^l a_i y_i = 0$$

so

$$w = \sum_{i}^{l} a_i y_i x_i, \sum_{i}^{l} a_i y_i = 0$$

We do want to note that in our implementation we do ignore the bias variable b.

2.4 Non-maxima suppression for overlapping bounding boxes

The humans in the image can be of different sizes and shapes, and to capture each of them perfectly, the human detection algorithm returns multiple bounding boxes. Ideally, for each person in the image, we must have a single bounding box. To select the best bounding box, from the multiple predicted bounding boxes, we chose to use non-max suppression. This technique is used to "suppress" the less likely bounding boxes and keep only the best one.

The purpose of non-max suppression is to select the best bounding box for each human and reject or "suppress" all other bounding boxes. The non-maxima supression (NMS) takes two things into account

1. The objectiveness score is given by the human detection model, this score denotes how certain the model is.

2. The overlap of the bounding boxes.

The non-max suppression will first select the bounding box with the highest objectiveness score. And then remove all the other boxes with high overlap. The same process goes for the remaining boxes. This process runs iteratively until there is no more reduction of boxes.

The following is the process of selecting the best bounding box using NMS:

Step 1: Select the box with highest objectiveness score.
Step 2: Then, compare the overlap (intersection over union) of this box with other boxes.
Step 3: Remove the bounding boxes with overlap (intersection over union) $> 50\%$.
Step 4: Then, move to the next highest objectiveness score.
Step 5: Repeat steps 2-4 until all the boxes have been assessed.

3 Proposed Solution

Our proposed solution is to first of all compute a distance reference from the pixel space. Then we compute the homography matrix H using the point correspondences, based on the distance reference points along with two more points, for the homography estimation algorithm. Once we have the homography matrix H we will be able to compute the estimation of the real distance between any two points in the projected space.

Second of all, we need to train our Support Vector Machine (SVM) using the INRIA dataset while applying the Histogram of Oriented Gradients (HOG), which is a feature descriptor, to use as a parameter for the training of the SVM algorithm.

Once the training is complete, we use sliding windows on the original image to compute the HOG, which is a feature descriptor, on each window. The HOG windows will be used as features, or parameters, for the SVM algorithm which will classify each window as either containing a human or not.

It is worth stating here, for transparency purposes, that even though we managed to implement by ourselves a perfectly functional algorithm to compute the HOG of a $(64, 128)$ image, we will not be able to use this algorithm in our code, as its computing time is too long compared to the capacity of the hardware at our disposal. Indeed, it computes the HOG feature vector of an image in 0.2 seconds, and each image contains several hundred windows for each of which we need to compute this HOG, and each image has to be resized several times for a proper detection of the humans present in it. Because of this, the time taken by our algorithm to detect our humans on an image can be of several hours, depending on the size of the image. And to use it on a video would take even longer. This is why we decided to use a built-in function to compute the Histogram of Oriented Gradients. However our code for this part of our algorithm is available with the rest of the code regardless. This function is the "hog" function from the "skimage.feature" package for python. The source code of this function is available in our bibliography.

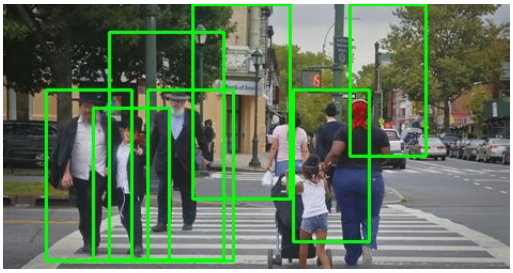
Finally, we take the center pixel from the bottom side of the windows that contain a human, then we multiply the coordinates (taken from the pixel space) of that pixel by the homography matrix H to get its coordinates in the projected space. We then have a set of coordinates for each detected

human, that we can use to compute distances between said humans, based on the distance reference computed in the training phase.

4 Experimentation

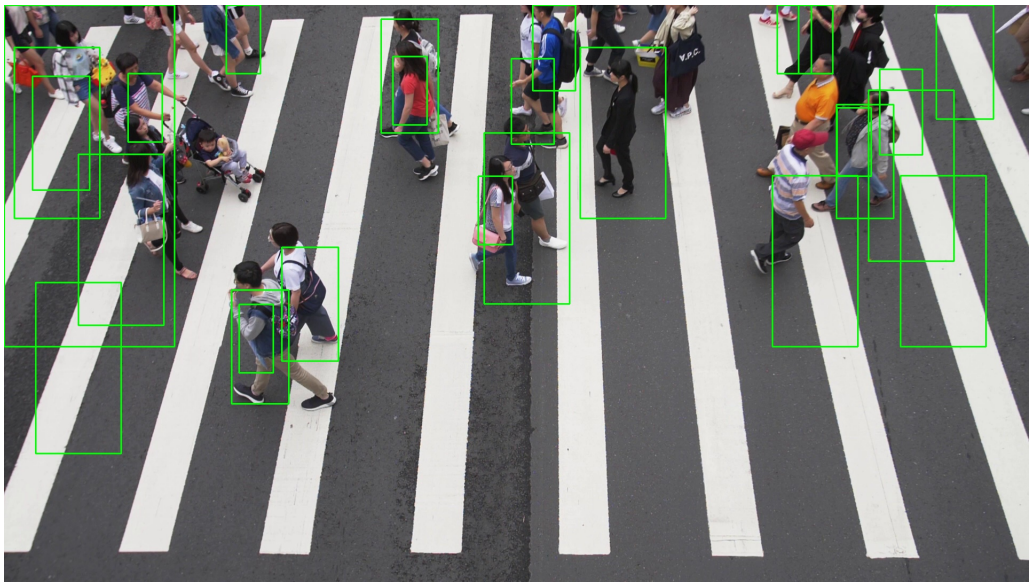
4.1 Human Detection

During the implementation phase of our project, we started by designing and implementing a human detection algorithm. It is taking as input an image in which humans were supposed to be contained, and returns three objects: a list of the four coordinates of each window where a human had been detected, a list of pair of coordinates that we were going to use as coordinates of the humans to compute the distance between several humans, and the original image on which green rectangles had been drawn using the coordinates from the first output list. In other words, those green rectangles are supposed to appear around the humans that have been detected by our algorithm. Figure 1 shows some samples of the results we got using this algorithm, on two photos and on one screenshot of a video that we are going to use in the end.



(a) Picture#1

(b) Picture#2



(c) Screenshot from the video

Figure 2: Results from the first version of the human detection algorithm

We can see that the algorithm seems to perform in what we consider to be a satisfying way on the two first pictures, however it is much less efficient on the screenshot from the video we want to use as our main example. We wondered what was causing such bad results, and noticed that one of the main difference between this screenshot and the two other pictures is the point of view from which they are taken. In the screenshot, the point of view and camera angle from which it has been recorded result in the stripes from the crosswalk appearing vertical in the two-dimensional space of the image. In the other two images, if we were to represent the humans and stripes with lines, the lines of the humans would appear almost perpendicular to the lines of the stripes. In the screenshot, those lines would almost be parallel.

So we decided to copy and edit than screenshot to remove the stripes, by replacing the white color on the ground by the color next to it. Figure 2 shows the result of the application of that method.

We consider this result to be as good as for the same image with the

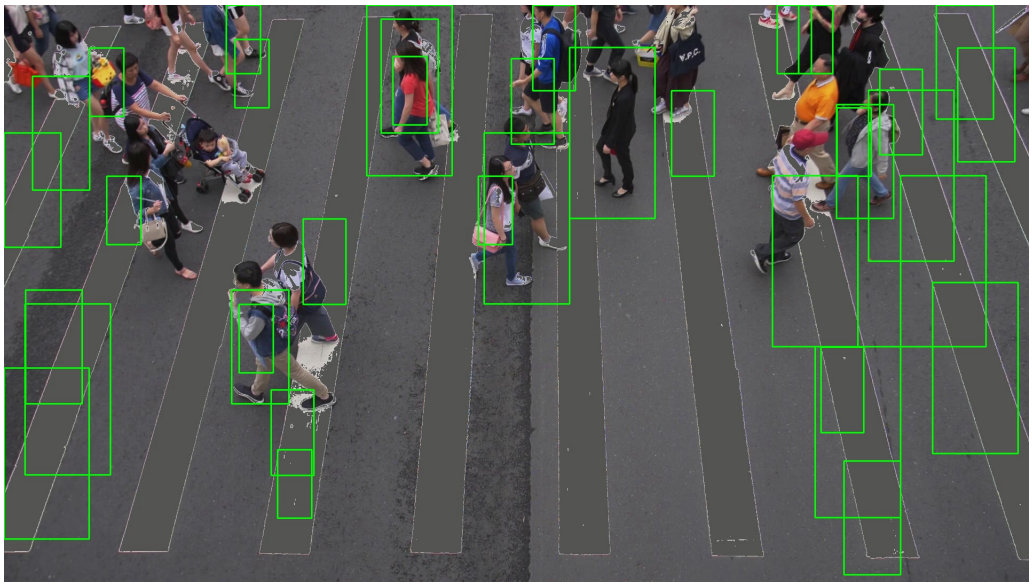


Figure 3: Result on the screenshot from the video with the stripes removed

stripes from the crosswalk. But the comparison of these results had allowed us to identify our problem, which was coming from those stripes. Indeed, the fact that the detection algorithm's output is different when we apply it

on an image without stripes means modifying this part of the image indeed had an impact on the detection. It then became clear for us that the stripes were still detected as humans by the algorithm, even though they are not detected the exact same way. It is worth stating here that we were not aware of this problem when we selected this video as the testing dataset for our algorithm. We now had to come up with a solution.

At this point, we decided to re-train the SVM by adding $(64, 128)$ images extracted from the original video, where stripes would be clearly visible, as negative samples. We were hoping that this would in a way allow the SVM to learn the difference between stripes and humans. The result of that method is visible on figure 3. We observe almost no stripes being labeled as humans, which proves our algorithm is indeed working as expected this time.

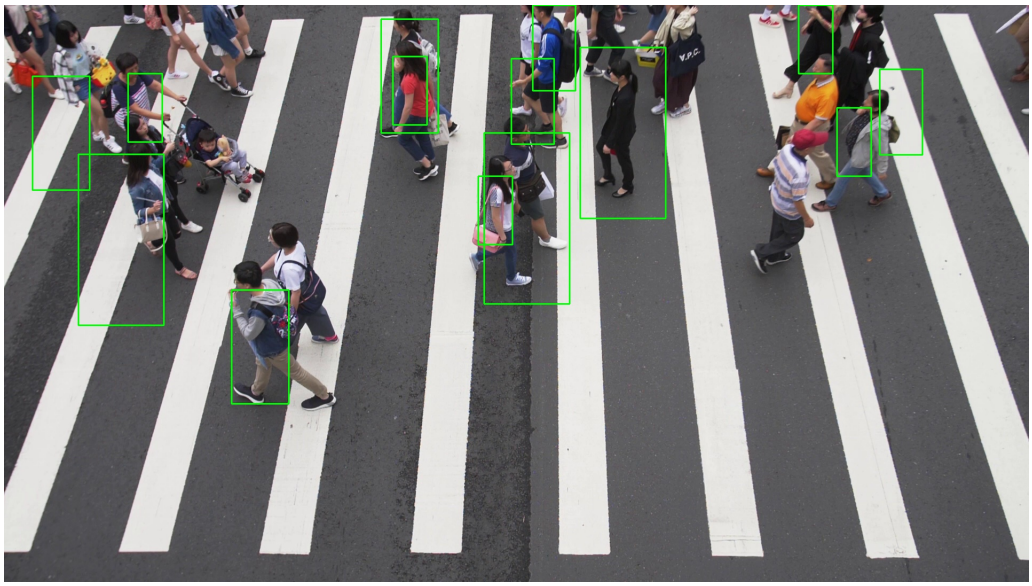


Figure 4: Result on the screenshot from the video with the stripes removed

When we decided to re-train our SVM algorithm, we also decided to modify a few things about the way it was designed. Indeed, we added some new metrics to be computed during the testing phase, along with the accuracy which was already computed. Those new metrics are the false positive

rate (corresponding to the number of false positives over the total number of positive predictions made), and the the missed rate (corresponding to the number of false negatives over the total number of negative predictions made).

We decided to do so because what were interested in here was more specific than just making our model more accurate, but more to make sure it detected less false positives.

We were also able to compute those metrics for the first version of the SVM, as the data needed for such a computation is still the same. The following table show the respective scores of the two versions of our SVM algorithm., tested on the same test set.

Metric	First version of the SVM	Second version of the SVM
Accuracy	0.964	0.959
False positive rate	0.003	0.003
Missed rate	0.043	0.047

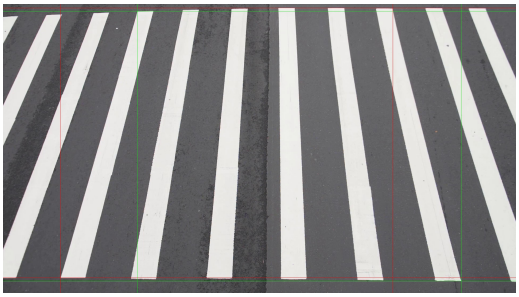
So we can see that, on this test set, the second version of the SVM has a slightly higher missed rate, and is performing overall very slighlty worse than the first version. There is no significative difference between the False positive rates of the two versions on this dataset. These results are likely caused by the fact that this test set does not contain the negative samples obtained from images of the stripes from the crosswalk labeled as non-humans. Moreover, the second version of the SVM has been trained with more negative samples, the goal of such a method being specifically to train it to recognise less objects as humans: such a result was then predictable. Therefore it is hard to perform an adequate evaluation on such a test set, and it is better to compare those models on the task of human detection with scaling window on the screenshot from our video.

4.2 Homography and Perspective Shift

Another phase of the implementation process of our algorithm was to compute the homography matrix using four point correspondences that we have picked. The four points that we have picked are shown in figure 4 (a): the first point is at the bottom right corner of the image where the two

green lines meet, the second one is at the bottom left corner where the two red lines meet, the third one is at the top right corner where the two red lines meet, and the final fourth point is at the top left corner where the two green lines meet.

Our goal in this homography is to straighten the lines that we have picked the correspondence points on, and thus by doing so we would have found a homography matrix that would, if applied to an image matrix, transform or warp it - and by doing so warping the perspective as well - into a new plane that would allow us to compute the distances between any two points on this place more accurately compared to the distance in the real world that underlines the image,



(a) Before changing the perspective

(b) After changing the perspective

Figure 5: Results of the change of perspective using the computed homography matrix

And thus, the lines that are drawn on the images in figure 4 help us see where our four point correspondences should be transformed to for the perspective to be warped correctly. We have decided to fixate the bottom right point, and assumed that the bottom left point should be on the same horizontal

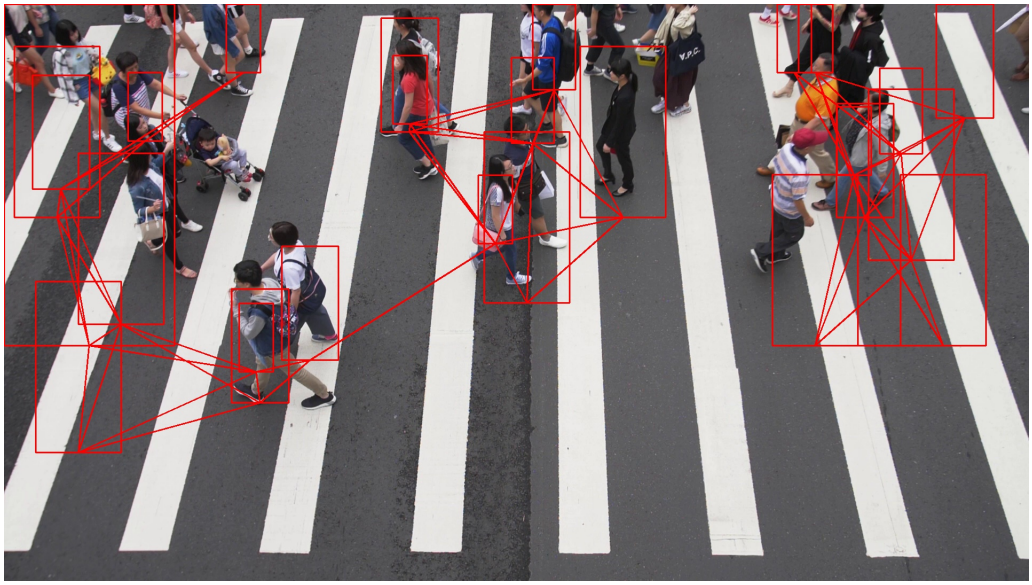
line, thus the bottom left point can be seen to lie on the red and green line intersection. We also assume that the top left point needs to lie on the vertical green line for it to form a vertical straight line perpendicular to the bottom horizontal green line. And thus, the top right line needs to be transformed into the point intersection of the green and red lines. And finally, we see that the top left line is assumed to fall on the same top horizontal red line to be on the same line as the top right point, and for it to from a straight vertical line with the bottom left point, the top left point needs thus to fall on the point that forms the red and green line intersection.

After getting the coordinates of the eight points in the pixel space, we find the homography matrix in the exact same way that was explained in the theoretical section. To see the effects of the homography matrix that has been obtained, we have applied it to the first frame of our test video input to try warping the perspective to see whether the homography matrix applies a proper transformation and shift of perspective. On the result of this process, shown by figure 4 (b), we can see that the homography works exactly as expected. Indeed, we can observe the crosswalk as if we were looking at it directly from above, and as we also made the assumption that each stripe of the crosswalk is 50cm wide, this allows us to estimate the real distances between people walking on this crosswalk. We do note that the green and red lines were drawn to simply show the affects of the homography transformation on the perspective for an easier result observation for the human eye.

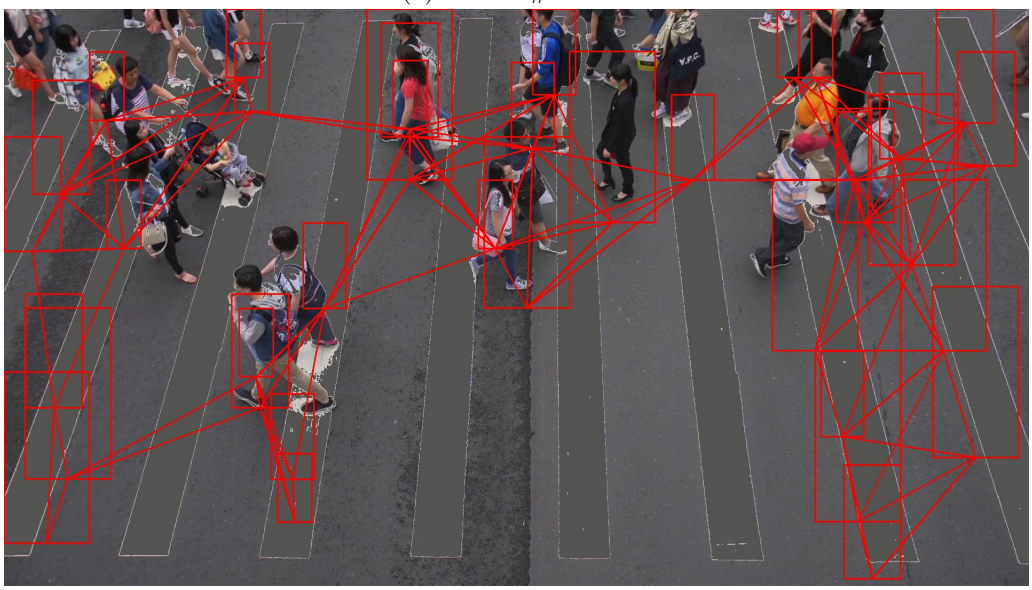
5 Result Analysis

The figure 5*a* shows a frame extracted from the video produced by our algorithm with the original video as input, and the figure 5*b* shows a frame from the video produced by the same algorithm but with the video without stripes from the crosswalk as input. Here the considered output is not only the human detection, but also the computing of the distances between detected humans.

We can clearly see that the result 5*a* is not exploitable, as an important part of what the algorithm detects as humans is actually composed of the crosswalk, and some humans are labeled as not respecting social distancing with a part of the ground. In the result 5*b*, some other parts of the ground are detected as humans than in the result 5*a*, but the result is still not



(a) Picture#1



(b) Picture#2

Figure 6: Results from the first version of the Social Distancing Monitoring algorithm on two different images

exploitable for the same reasons.

This due to the technique used to remove the stripes from the crosswalk: all the pixels in a certain range of values that correspond to the white color of the stripes from the crosswalk have had their values change to match a specific shade of grey. Therefore the shape of those stripes is still visible, and it may still be detected as a human being by our algorithm.

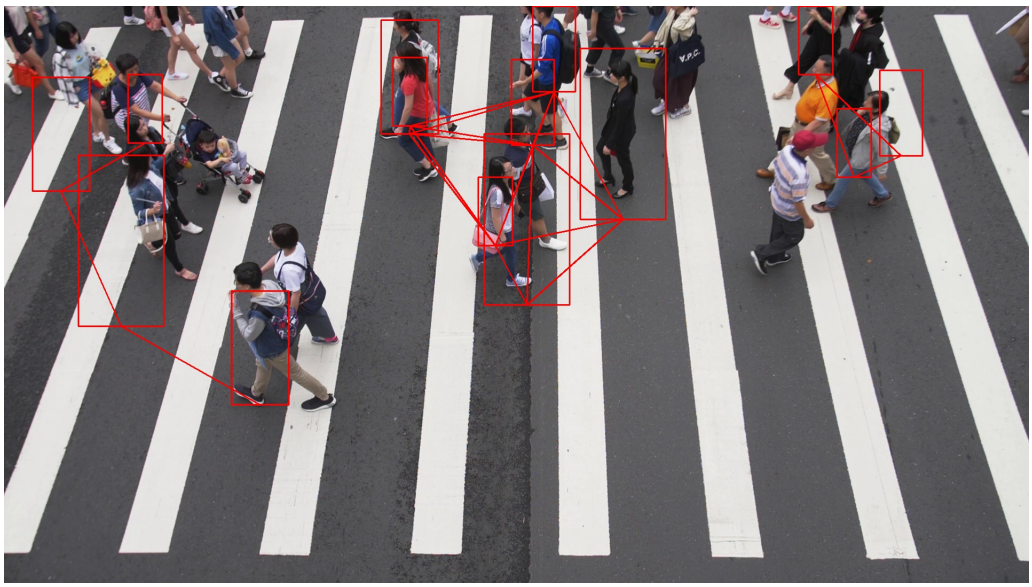


Figure 7: Result on the screenshot from the video with the new version of the SVM algorithm

The fact that our algorithm detected non-human shapes as human beings is due to the composition of the chosen dataset for the training of the SVM. Indeed, in this dataset, the "negative" images, that is to say the samples not belonging to the class "human" were simply windows of size $(64, 128)$ extracted from any part of an image not containing humans. We figured the performance achieved by our algorithm would be better if we added more various "negative" samples in the training set, and in our case especially better if photos of crosswalks of shape similar to those detected as humans in our video had been originally added as "negative" samples to our training

set.

This is why we decided to extract such images from our video, to add them as negative samples in our dataset and to re-train the SVM algorithm on this new dataset, to see if this would improve our results. The figure 6 shows the obtained output from our algorithm.

This shows that by adding negative samples to our dataset, we managed to reduce the number of false positives detected by our human recognition algorithm. This result is so far the best result we had, and we decided to use this method to compute the video as a final output of our algorithm.

6 Practical considerations

The content of this part can also be found in the "README.txt" file in our code repository.

6.1 Architecture of the published code repository

In this section, we will describe briefly the architecture of the code repository published with this report.

The file "SocialDistancingMonitoring.ipynb" corresponds to the code file we showed during the presentation, and contains the most interesting functions, procedures, tests and experimentations we did during this work.

The file "Preprocessing.ipynb" corresponds to the pre-treatments we applied to the images in order to then be able to use them as a dataset. It takes as inputs the images from the folder "DataToPreProcess" and saves its outputs in a folder named "Datasets".

The file "HOG and creation of dataframes.ipynb" contains:
- The definition of our function computing the Histogram of Oriented Gradients for each image, and the corresponding built-in function from the skimage package that we had to use.
- A treatment applied to our images to build training and test datasets in the form of csv files.

The "Saved" folder contains elements that were computed one time and re-used, especially during the presentation. We find here "SVM_b" which is the first version of the SVM, and "SVM_b_2.0" which is the second version, as described in the report. We can also find the file "Homography_matrix" which contains the homography matrix used for our video.

6.2 Data not sent in the original repository

We have not been able to send all the necessary data for our work into one zip file of less than $10Mo$. We will state here where the necessary data is accessible.

The Datasets (considering both csv files and images used to build those files) are available on GitHub at the following url: `https://github.com/DabMatt/SDM`.

The purpose of the "DataToPreProcess" folder, found on GitHub, is to contain raw images that have been preprocessed before using them in our dataset. It contains subfolders "Train" and "Test" which themselves contain subfolders "Pos" and "Neg" corresponding to samples respectively belonging and not belonging to the class "human", for our training set and our test set.

The first purpose of the "Datasets" folder, found on GitHub, is to contain the preprocessed images that are fit to be used for building the dataset, and its architecture is the same (using "Train", "Test", "Pos" and "Neg" subfolders).

Its second purpose is to contain three csv files representing our datasets. One is called "test_data_b.csv", corresponds to the test set, and is located in the "Test" folder, outside "Pos" and "Neg" subfolders.

In a similar way, in the "Train" folder we find two csv files "train_data_b.csv" and "train_data_b2.csv".
"train_data_b.csv" has been used to train the first version of the SVM described in the report.
"train_data_b2.csv" has been used to train the second version.

Moreover, to send the code files, we had to delete all the outputs from

all the cells in the notebook (some of these outputs being images, they were taking way too much space). The same code files with outputs are available on the GitHub repository as well.

Considering the images and videos used as inputs to our algorithm, as well as the ones corresponding to outputs produced by our algorithm, they can be found on the following Google Drive: `https://drive.google.com/drive/folders/1yHc6ppXA98T4zdSWuPmoEoexpBe5q9TY?usp=sharing`

The "Pedestrians" folder corresponds to inputs while the "Results" folder corresponds to outputs.

When downloaded, all these folders (both from GitHub and GoogleDrive) should keep the same name, and be placed in the first folder (named "FinalCode") of the repository.

References

[1] Source code of the function computing the histogram of oriented gradient, from the package scikit-image. `https://github.com/scikit-image/scikit-image/blob/master/skimage/feature/_hog.py`, 2019 (last update).

[2] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. IEEE, 2005.

[3] Elan Dubrofsky and Robert J Woodham. Combining line and point correspondences for homography estimation. In *International Symposium on Visual Computing*, pages 202–213. Springer, 2008.

[4] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.