Image Processin
<License plate Recognition [redacted]>

[redacted]

1 Introduction *(0.5 pt)*

Before we start let us describe the general pipeline of our project. To achieve correctly recognizing a license plate from a video, we first start by sampling the video by frames. In each frame we need to localize all the license plates present. For that we threshold the frame based on the colour yellow to easily extract all dutch license plates. We then use Canny edge detection and contour detection algorithms to specify the bounds of the license plate. Finally we crop it and rotate it such that we can provide a normalized input for the recognition part. The recognition then splits the cropped plate into separate segments and for each segment determines which character it is using pixel-wise template matching. After we construct the most likely string to represent the plate, we run it through a verifier that determines whether the string does indeed comply with Dutch license plate naming standards. In the very end we take all license plates and we split the output into subsections using the hamming distance between plates. This computes when a plate changes, ignoring small misclassifications up to 2 characters. We then determine the most likely license plate in each subsection using a majority vote, and output that as the final result along with its frame number and timestamp.

2 License plate localization method *(4.0 pt)*

2.1 License plate localization data description *(0.5 pt)*

The data we used for this project was already provided in the template. The data consisted of multiple videos per category, we have split these videos into two sets, the training set and the validation set. The size of the training data was 60% of the provided data, the rest was put into the validation set. We have then further divided the sets based on category so we could track the performance metric per category rather than as a whole. We also had to create the correct answers we would aim for when developing our localization method. As we implemented our own image processing algorithms, we decided that we would compare our algorithms to the ones implemented and tested in libraries. We used the OpenCV library for this, we generated the cropped pictures of license plates according to the library implementations and then we compared them with our own implementation. This created another data set of pictures which needed to be manually reviewed as to determine whether we manage to find the license plates we need. We went through all collected data manually and removed the ones that did not match the expected result. Figure 1 shows a direct example of the data we have constructed and compared from the original data we were given.

2.2 License plate localization system *(1.5 pt)*

2.2.1 Pipeline

When the Localization method receives a frame from a video, it first finds all the yellow pixels using thresholds. We apply the mask of given yellow pixels to extract them from the frame, as can be seen in Figure 2. Then we send this updated frame into the Canny edge detection algorithm that creates a binary image to represent whether a pixel is an edge or not. We proceed to collect all this data and put it into more sensible structure using contour detection, which creates bounding boxes around all found connected components of edges. For each bounding box we only store the four corners of the box. After we have collected all the bounding boxes found within the yellow mask, we check certain properties of these corners, which determine whether the contour contains a license plate or not. After we have successfully retrieved the positions of all the license plates in the frame, we proceed to crop them based on the aforementioned corners, however, we crop them from the edge image, so we omit the colours. As the final step we go through the cropped license plates and remove as much noise as possible using dilation and projection to remove any rotation. We do this to normalize the output of the localization algorithm.

2.2.2 Edge detection

We follow the Canny edge detection algorithm to create an edge image. We first start by turning the input image into black and white image. We then proceed to blur the image with a Gaussian filter so that we get rid of noise from the picture. We then apply the Sobel kernels which result in the gradient and theta. To optimize the non-max suppression part of the algorithm we create 8 kernels which check each direction (each kernel checks one direction) from the middle pixel whether there exists a pixel larger than the middle value. We create 8 convolutions and then take the pixels that is present as a maximum in all 8 results. We do these 8 convolutions as we found it more time-efficient as convolutions are optimized by a library while our iterations over all pixels once was not efficient enough. After we have successfully managed to make the edges thinner, we apply thresholds to determine which edges are strong and which ones are weak. We mark the strong edges with the maximal value of 255, while the weak edges are marked with a 1. Finally, we end up in the edge running part where we run through all the weak edges and apply a 3x3 kernel of ones. Therefore summing up all neighbouring values. This will ensure that weak edges with a neighbouring strong edges will have a value of 255 making them strong edges. Figure 3 depicts the Canny algorithm with our intermediary results. In the end of our canny algorithm we return the binary edge image, consisting of 0 where there are no edges present and 1s (255) where there is an edge present.

2.2.3 Contour finding

The contour finding algorithm we have implemented takes a binary edge image as input and outputs and array of contours defined by corners of its bounding rectangle. At first we have have reviewed the SUZUK (Suzuki85) algorithm used by the OpenCV library, however we have soon figured out that it would be inefficient and would do much more than needed, as we only need to find the license plate. We have then agreed on a simpler implementation inspired by psuedocode from (peteruithoven). We find the first position where the pixel of the input edge image is a 1. We then go to that index and perform a depth first search thank to which we connect all surrounding 1s into one cluster. When we visit a pixel, we save its index into the cluster and set its value to 0, so we do not visit it again. After we finish filling up a cluster, we check whether it connected a certain minimum of points, more specifically at least 60 points. If the contour contains enough points we save it and continue with the clustering. After we finish we loop through all the clusters and thanks to OpenCV library we find the minimal area rectangle, which defines a rectangle that fits all the point of a cluster, with the minimal possible area. We then replace all the connected points with only 4 points of that rectangle and we return a list of all clusters. Figure 4 depicts a step by step description of the contour detection algorithm we have implemented.

2.2.4 License plate properties

To detect whether a contour (4 points of the defining bounding box) are indeed a license plate we use certain properties of license plates. First of all we expect a perfect rectangle or a parallelogram in the case some rotation has been applied. Therefore we check whether the opposing sides of the rectangle are parallel and with similar lengths. Secondly the rectangle must be a certain minimum area, so that we ignore small rectangles caused by noise. Lastly we check the ratios between horizontal and vertical side of the rectangle, based on our measurements it should be around 5, therefore anything widely different from 5 would not be considered as a license plate. Figure 5 shows all of these properties on an example, we can observe that while the vertical side is approximately 1, the horizontal one is 5 and that indeed the license plate has a rectangular shape.

2.2.5 Cropping

Now that we have established exactly where the plate is, we need to standardize it to allow for character recognition. This is done by taking the 4 coordinates of the plate corners and orienting them correctly. Then we compute a projection matrix (Pradeesh [2018]) from these coordinates to a standard size of (500, 100). This is almost certainly larger than the original size of the plate, and some sort of interpolation is needed. For this we implement an inverse projection that performs a kind of nearest neighbour interpolation.

2.3 Evaluation metric for license plate localization *(1.5 pt)*

As mentioned in the data description, we have generated the correct solutions for each frame with the OpenCV library implementations of the algorithms we implemented on our own to see how well they perform against each other. We then manually compared whether we receive the license plate similar to what is output by the OpenCV implementation. We had to compare these manually as the plates might be slightly rotated, but that would not mean they are wrong. As a metric we have chose to use accuracy per video, which means that if the algorithm provided a correct, or very similar, output than what our base case would, we took the entire video as correctly classified, as it correctly output the license plate at least once.

2.4 Analysis of the license plate localization results *(0.5 pt)*

With the above mentioned metric we are able to achieve 100% accuracy on the validation sets for categories 1 and 2. For category 3 we are receiving 36% of correctly cropped plates and for category 4 we are receiving 0%. As you can see this pipeline works nicely for normal dutch cars, however, there are still some issues that could be improved upon. The first major issue are the colour thresholds, if we are given a yellow car with a dutch license plate, the algorithm would take a long time to go through the pipeline and might not recover the correct license plate. By creating a threshold based on the colour yellow we also ignore all license plates which are not yellow, thus resulting in 0% success rate in category 4. Another improvement that can be made is with error recognition, in category 3 the problem we have found is that the plates are simply too small to be correctly classified, as two cars need to be fit into the frame for them to fit. This can be improved either by creating a more robust error correction, which would not punish small rectangles as much, or by increasing the resolution of the video.

3 Character recognition method *(4.0 pt)*

3.1 Recognition data description *(0.5 pt)*

By the time we began developing the recognition algorithm of our system, we had already created an accurate localization procedure, and so we decided to test our recognition system using localized plates from the algorithm above. We followed a similar procedure to the one above, where we split the plates into a 60% training set and a 40% validation set. The validation set was used for evaluation, while the training set was used to incrementally improve the recognition algorithm. The data used for training and testing was annotated manually by us, meaning that for each plate we noted the expected string output.

3.2 Recognition system *(2.0 pt)*

3.2.1 Character localization system *(1.0 pt)*

Character localization, or as we refer to it, segmentation, is performed on a plate that has been cropped, projected, and de-noised by the localization system. We first sum all pixels vertically in order to get a "vertical projection" of the license plate, and generate a histogram. We analyzed a number of these histograms, and saw that spaces between characters consistently had around 0 pixels on their vertical projection, although sometimes some noise was present. We therefore decided that a space is identified by a vertical projection of less than 4 pixels. We created an algorithm that marks these "space" locations and automatically segments the characters into a list. If a segmented character has less than 100 pixels, it is discarded, as this is too few to be a valid character or a hyphen. Figure 6 shows the stages of this process.

3.2.2 Character recognition system *(1.0 pt)*

We now have the segmented characters ready for recognition. The first step is to crop each character horizontally and resize it in to a standard size of (60, 85) pixels. These characters have already been binarized from the previous step, so we can now apply template matching. The provided templates of Dutch license plates are normalized to the same (60, 85) size, and each suspected character is compared using a bitwise-xor to every template. In order to increase accuracy in case localization does not provide a perfectly aligned plate, we compare the character not only to the 27 templates provided, but also to a version of each template that has been skewed left and right slightly. This implies a total of 81 comparisons per character but increases accuracy significantly. An example of these templates is show in in figure 7a and figure 7b. The winning template is the one with the highest negated xor sum. In order to recognize hyphens, we check that the number of pixels in the provided character is smaller than a certain threshold. Finally, Each complete plate string was passed through a Dutch plate verifier that discards plates that cannot exist. When this incorrect classification happens, it is likely due to edge noise, and our algorithm crops the image slightly and performs the entire recognition step again, as well as adjusting the binarization threshold. If the plate is not valid after a certain number of cropping and threshold iterations, the plate is discarded completely and we move onto the next one.

3.3 Evaluation metric for recognition *(1.0 pt)*

To evaluate the correctness of our recognition, we have used the license plates generated by the OpenCV implementations and checked whether our recognition algorithm correctly classifies this kind of input. We have created two different splits. First we split the images per license plate, so to check whether the final output of the recognition software would be correct. Secondly, we check each cropped license plate individually, to check how many mistakes does the software do per license plate. To numerically evaluate the software we used accuracy, so either the algorithm correctly classifies a license plate or it does not.

3.4 Analysis of the recognition results *(0.5 pt)*

When making use of the verification system for Dutch plates, and checking only for the final result per license plate, our system managed to get a 100% accuracy for categories I & II, and a 50% accuracy for category III. Category IV was not prioritized in the design of the system, so the accuracy was 0%. When checking individual cropped images, our accuracy dropped to 90% for Category I and 80% for Category II. Category III and Category IV stayed the same. The most clear failure case for our recognition system are international license plates. Due to the difficulty of localizing them in the first place, we decided to optimize the algorithm maximally for Dutch plates and international plates are therefore never recognized. A future improvement would involve using colour data to decide how the license plate is recognized. By detecting if a plate is white rather than yellow, it would be possible to relax the verification step and allow international formats to be considered valid, while ensuring the Dutch plates are properly verified.

4 Analysis of system *(1.5 pt)*

4.1 Time analysis *(0.1 pt)*

From our analysis we observed that the average time spent on a frame is around 450 milliseconds. However, this varies a lot depending on the amount of yellow pixels found in the frame. When we run the dummy video (2:54) with no sub-sampling the project takes 816 seconds (13:36) to finish. Therefor we have started sub-sampling the video, this could lead to worse results, as we rely on as much data as possible to correctly identify incorrectly recognized plates. We have considered multiple sub-sampling rates such that they don't lower our scores and we settled for sampling every 5th frame, as it brings the overall time back to 164 ms (2:44) while still resulting in the same accuracy.

4.2 Successes and failures *(1.0 pt)*

The current system has been optimized to work well with Dutch plates, with a format verifier that minimizes the number of false positives. We are happy with this as it improves overall performance on Dutch plates, but on the other hand it makes it very difficult to recognize international plates. Our localization algorithm makes use of colour thresholding to isolate yellow areas and speed up the subsequent custom find-contours algorithm. Yellow cars make this quite difficult. Not only do they increase the overall run time of the algorithm, they add more information that can potentially be misclassified as a license plate.

4.3 Future improvements *(0.4 pt)*

As mentioned before, in the future we would improve the accuracy on international license plates. A system that detects the colour of the plate and, based on this decides how strictly to verify the format, would be able to also recognize international plates. Additionally, we would increase the number of templates used. Currently, some international plates are not recognized simply because we do not use templates that correspond to characters such as 'C'. Canny and find contours algorithms could still be optimized as they are still the main bottlenecks in our project.

Figure 1: Comparison between library implemented localization (left) our implementation of localization(right).

Figure 2: Result of thresholds based on colour yellow.

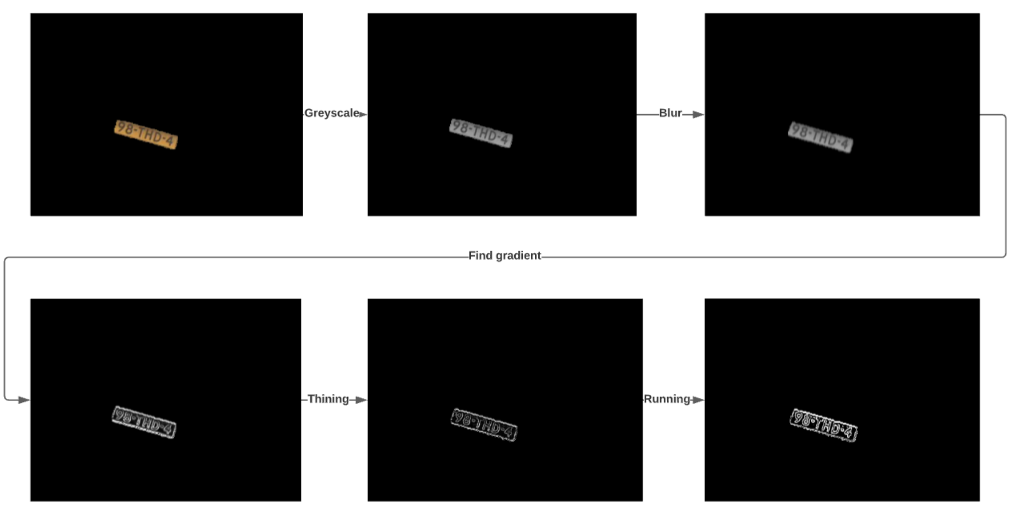

Figure 3: Showcase of the intermediate results of the Canny algorithm implementation.

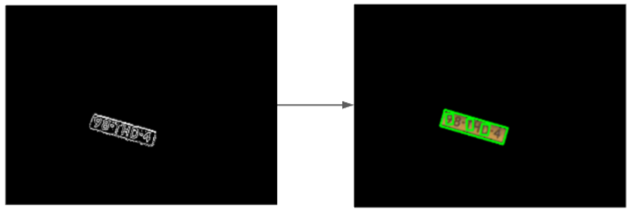

Figure 4: Results of the find contours algorithm. Given an edge image, we find the connected components and create rectangles around them as depicted with green lines above.

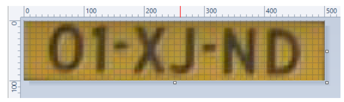

Figure 5: Cropped license plate showcasing the side ration and the rectangular shape.

(a) Example of a binary plate before de-noising.

(b) Example of a de-noised binary plate.

(c) Example of segmented characters.

Figure 6: Stages of normalization of plates.

(a) Example of a template for a 'B' character.

(b) Example of a right-skewed template for a 'B' character.

Figure 7: Characters used in recognition.

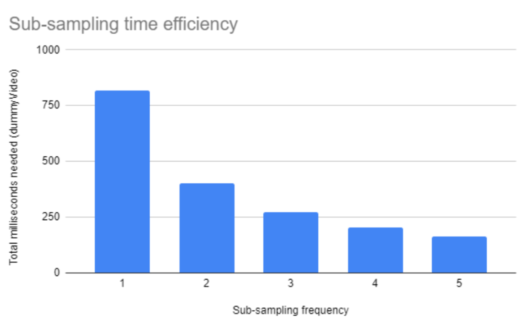

Figure 8: Depiction of different times taken to finish the provided dummy video (2:54).

References

peteruithoven. Contour-finding-experiment. URL https://github.com/Doodle3D/Contour-finding-experiment/blob/master/js/pseudocode.html.

V. Pradeesh. How to compute homography matrix h from corresponding points (2d-2d planar homography), 2018. URL https://math.stackexchange.com/questions/494238/how-to-compute-homography-matrix-h-from-corresponding-points-2d-2d-planar-homog.

S. SUZUK. Topologicalstructural analysis of digitized binary images by border following. URL https://www.programmersought.com/article/67731595799/.