*** ML Project

This document will explain the steps taken in order to perform OCR on the provided dataset.

1) Environment

I have worked on Ubuntu 18.04 and Anaconda (Miniconda 3). The preprocessing has been done mainly with OpenCV and the OCR has been performed by the Google API.
An *environment.yaml* configuration file is present in the main directory. In order to use it for creating an environment with Anaconda, the command is:

```
$ conda env create -f environment.yaml
```

In order to call the Google API further configuration is needed, but it will be described in a further section.

2) Corrections and first preprocessing

There were some errors in the naming of the images and in the *./training-data-values.csv* file. The corrected csv is *./training-data-orientations.csv*.
Since the registration certificate scans could have any orientation and coloring, a first preprocessing phase has been done using ImageMagick script by Fred Weinhaus. All the images have been processed with the command:

```
$ textcleaner -g -e stretch -T -u <input_img>
<output_img>
```

Where:

- -g: convert to greyscale
- -e stretch: enhance image brightness before cleaning
- -T:
- -u: unrotate image; cannot unrotate more than about 5 degrees

The processed image are in *./dataset_cleaned_with_textcleaner*.

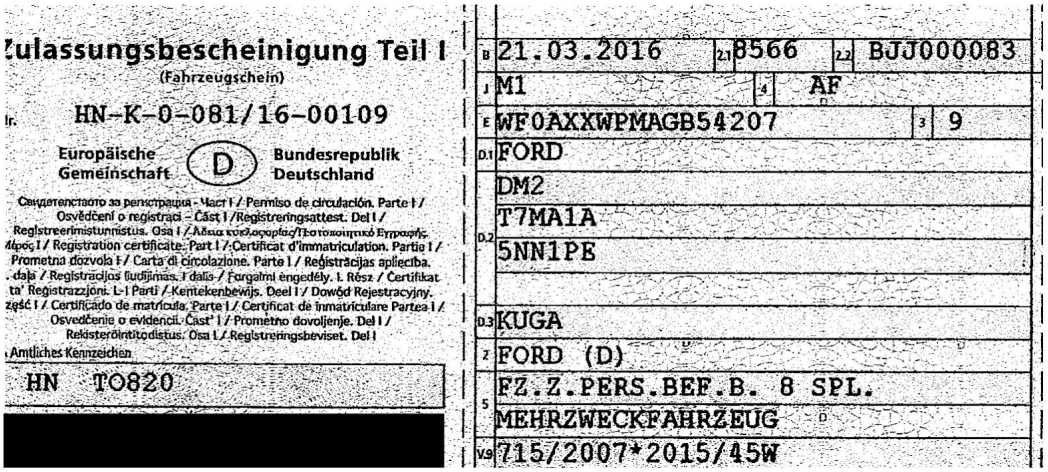
ImageMagick does a good job, but the images could still be skewed, dirty, or rotated to the wrong angle (ImageMagick could have skewed it to the closest 90-degree rotation): further preprocessing will be needed.

3) Bounding boxes of interest and justification

Instead of feeding into an OCR model the whole image and then find the relevant text between the result, it is more efficient to feed to the model, for every registration certificate, one small image containing the date, another the license plate and another one with the VIN. This approach has several advantages:

- A way smaller image to process for the model
- Custom preprocessing for every type of data: the license plate characters are bigger than the date and the VIN characters, the dates could use different thresholding…
- A -possible- custom model for every type of data. For instance, the date model could be instructed to output only digits.

Finding the bounding boxes in an automatic way is easier said than done, though; the images are rotated in any direction and with different scales.



This symbol (which I call -erroneously- logo) will serve the purpose:

It compares in the same position in every image (except one out of 294) and it is close to the 3 data fields of interest. With its identification, it is possible to rotate the image correctly (one could have used HoughLines to detect the horizontal lines of the form and use their angle to deskew the image; however this way the image could still be flipped outside down), and given the width and the height of the logo's bounding box, estimate a correct position, width and height of the other 3 bounding boxes.
The first part of the code will, therefore, be about finding this logo, using template matching.

4) Logo search

The code described in these sections will be in *./template_matching/template_matching_v3.ipynb*.

The first function is *find_orientation_threshold_rotate*: it takes as input an image from *dataset_cleaned_with_textcleaner* and tries to find the logo. First, it searches the logo in every 90-degrees possible rotation: this is because the best confidence out of the 4 pattern matchings will determine the correct orientation of the image.
Once the logo has been found with enough confidence, the image is rotated into the correct position and the logo bounding box is calculated, in order to pass the dimensions to the function *find_bounding_boxes_fixed*, which will return the 3 bounding boxes of interest.

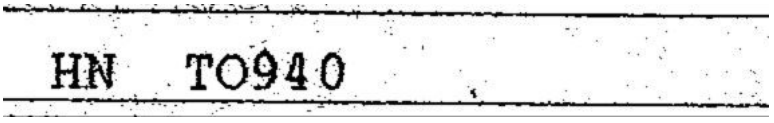
If the confidence is too low, the logo has not been found in any orientation: maybe it does not appear, or the image is too noisy, but another possibility is that the scale of the logo in the image is not close enough to the scale of the ones used for the searching: in this case, searching not only by logo orientation but also by image scale could provide a solution. Thus the function *find_logo_scale_rotation* is called. *Find_logo_scale_rotation* works similarly as the method above, but it implements a double search by scale and logo rotation. Once the most

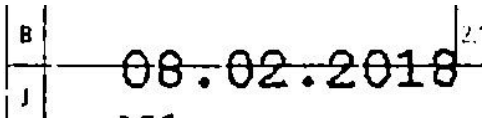
probable rotation is found, the function *find_logo_scale_rotation_v1* is called, to calculate the scaled logo bounding box and pass the dimensions to *find_bounding_boxes_fixed*.

This approach finds the logo almost every time in the dataset: out of 294 images, only 10 do not have correct bounding boxes for the 3 data fields of interest.[1] A more specific phase of preprocessing may help, but having to deal with such a vast array of image qualities in the dataset requires general preprocessing techniques.

Find_bounding_boxes_fixed calculates and returns the bounding boxes of the license, date and VIN fields (as their own images) present in the registration certificates. Sadly the bounding boxes cannot be extremely close on the field, because a small skew could still be present in the image, or each data field could be written in a slightly different location; this will not, however, represent a big issue, as it will be shown in another section.

Once the images are returned, a custom preprocessing phase can be applied to each field type.





[1] The images with a mission logo are number: 6, 11, 58, 88, 93, 199, 242, 271, 276, 278, 279.
Out of these:

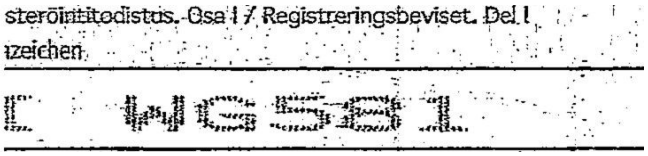
- 6 has the logo in another location
- 11, 199, 271, 276, 278 and 279 have a strange shading, derived from its original colors. Also, the symbol is dark on the inside, hindering the template matching performances. Custom thresholding would help, but with such dark images, the application of a required strong threshold on the whole dataset will affect the brighter images.
- 88 does not have any requested information
- 93, 242 have a particular perspective: no doubt that with a more accurate preprocessing the logo would be found.

Examples of images returned by the *find_bounding_boxes_fixed* function.

5) Field preprocessing

Now is the time of a precisation: at the beginning of the project I wanted to implement an OCR model based on Tesseract; however the performances were not satisfying and I switched to Google OCR, and the results were definitely superior. Another advantage of Google OCR is that it requires considerably less image preprocessing than Tesseract. For example, regarding dates and VIN, there is no need to remove the horizontal form line that crosses the characters sometimes. Another great advantage is that in the VIN and date images other unrelated characters can appear: Google OCR will detect them but it is possible to isolate them from the actual date or VIN (considering the lenght of the returned parts of text found). Since Google OCR applies its preprocessing to the image, both the date and the VIN image type do not have a significant custom preprocessing; just a simple Otsu thresholding appears to be enough.

Considering the license plate, there is a more subtle problem: the 2-4 license letters could be confused with other text that appears in the image, so a more accurate bounding box. The method *find_horizontal_lines_license* detects the horizontal lines in the image with Hough Lines algorithm, and reshapes the bounding box according to the closes line to the middle height point (with some margin).



Sometimes too much extra text can appear in a license image. Reshaping the image using the two horizontal lines solves the problem.

Once preprocessed for the last (personal) time, the images are saved in *./dataset_particulars/*

6) Google OCR

In order to use the Google OCR API, a Google Cloud account is needed. Moreover, an authentication key must be exported on the system in order to send requests to the cloud service. Tutorial here.

Since I am using Google OCR, I will not have a train/val/test split: all of the data will be used as test.

In *./call_to_google_OCR ,* the 3 pyhton scripts *test_google_date.py, test_google_vin.py* and *test_google_license.py,* work the same way: they submit the images to the API and retrieve the correct text field. A bit of custom processing of the answer is then applied to each of the fields, such as lenght / type of chars checking.[2] This way it is possible to know for sure when the OCR failed. The predictions csvs are saved in ./results/

7) Results

I calculated the accuracy for each data type at the end of *./template_matching/template_matching_v3.ipynb*.
If at least one character of a prediction differs from the true value, the prediction is considered wrong. I also calculted how many times an error is undetected, and whether there are false negatives.

Out of 294 samples:
License score:

License errors: 121
Plate errors: 65
Total errors: 124
Unchecked errors: 50
False positives: 0

[2] There are 2 errors in the VIN column of *training-data-values.csv*: reg-cert-179 misses a P, reg-cert-290 misses a Z. Correct VINs: WVWZZZAUZHP335525, VSSZZZ5FZJR178235

Date score:

Total errors: 161
Unchecked errors: 113
False positives: 0

Vin score:

Total errors: 220
Unchecked errors: 82
False positives: 0

9) Tesseract

As said before, I first wanted to use Tesseract for this project. I will include the code in *./east/tesseract_samples_v1.ipyinb* for completion, but it is not part of the final project.

10) Conclusions and further development

Could the registration certificates can be read out automatically with good confidence? Absolutely. With enough time and custom models (even better for each field), one can achieve good performances even with Tesseract. The almost fixed structure of the fields of interest makes not only possible to detect erros, but also to correct them with an extent: corrections like ‘O’ into ‘0’ and ‘1’ into ‘I’ should be easy, but surely even more complex ones can be made.

This was an interesting experience, and you are definitely working on an interesting problem; I thank *** for the opportunity of make me improve my skills on such a dataset.

Marco Perotti