Lukas Jurisica (101045391)
Teodor Kirilov (101021403)

COMP4102: Computer Vision

Final Report: Meme-Alyzer

APR 18, 2020

1 Abstract

Memes are a form of humorous images/videos/content created by individuals that are shared in a widespread manner all over the internet. It is quite common for people to use templates/meme formats to simplify this creation process. As a result, memes are often created by combining a well known image, and a specialized piece of text to add new meaning and humour to the situation. The goal of our project is to create a piece of software that can analyze a meme; extract the template that was used in it's construction and attempt to detect text in the image. This will be accomplished in the Python programming language, with the assistance of various utilities in the OpenCV library. Text detection/recognition is a known difficult problem, and in addition to this, memes are often created by non professionals, which leads to poor image quality and design decisions (sometimes this adds to comedic effect). In addition to this, there are many ways meme formats are modified (deep frying, crossovers) that make template recognition a difficult problem. Our hope is to build a robust system that can handle as many of these situations as possible.

2 Introduction

The application can be divided into having 2 larger over-arching categories. These being the template recognition and text detection problems discussed in the abstract. Each had their own methods for extracting information from the meme that was used to infer additional parameters.

Our main objective with this assignment was to create an application capable of distinguishing between different images and figuring out the underlying templates behind each provided image. This is executed by extracting keypoints and descriptors form each image and finding the number of optimal matches between said images and a collection of externally stored ”Meme Templates”.

In the original proposal we described a possible scoring algorithm, which sadly we could not implement effectively and decided to simply leave out of the application.

As a stretch goal, we initially wanted to do text recognition on the memes that were fed through the software. Text recognition deals with getting a textual output from the software, meaning actually reading the test. Text detection, on the other hand, simply deals with finding the location of text in an image. In our case, we opted to perform text detection, as the complications which would arise from attempting to perform recognition were undesirable. There were 3 methods of performing text recognition that we considered, however we ultimately found that none of them would be suitable. The reasons for this, and the final algorithm for text detection will be further discussed in the approach section of this report.

3 Background

Dividing out system, we have 2 main features. Determining the meme template used, and text detection. There are plenty of articles and papers that talk about optimizations in solving these problems, making them more robust, more accurate and more efficient. Due to the nature of this assignment, we do not have sufficient time to go as in depth as some of these methods explored in papers. As for other software packages, text recognition/detection is often accomplished with the use of machine learning, which is not in the scope of this course.

4 Approach

The primary application feature is finding the backing templates behind the provided images. During the initial research stages of the project, we discovered two main alternative methods of completing this task.
The first method, OpenCV has a function build specifically for the purpose of finding templates inside of an image. There are several restrictions associated with this process, some of which are the type of templates that the function accepted the other major concern was the way the provided algorithm went about of completing this task. The provided image is slid across the X and Y axis of the source image and storing the value of the comparison at each location in a separate matrix. This method proved to be inefficient and not suited for our needs, since even after heavy image processing, there was no guarantee that this function would be able to find similarities between provided images.
The second method we experimented with and eventually implemented was feature detection and feature matching. The overall comparison algorithm consists of feature extraction, feature matching, thresholding and finally a comparison between the different templates.

- The feature extraction is done using the ORB detector provided by OpenCV and developed by the ”OpenCV Labs”[2]. Orb is combination of the FAST keypoint detector and the BRIEF descriptor with some enhancements made to increase performance. The drawback of this method is the low number of keypoints extracted compared to SIFT or SURF,although the original paper states that to compensate the extraction is much faster than SURF and SIFT.
- The Feature matching is done using a Brute Force matching method to locate the k-th nearest neighbours.
- During the thresholding stage, using Lowe’s Ratio Test[3] with the value of 0.7 as the multiplier. We conducted experiments where we increased this value up to 0.75, but regardless of the increase of ”good” matches between an image and the saved templates, we did not notice any increase in the prediction accuracy. The only notable difference was that some images without correctly identified templates simply changed the template they we’re matched to (incorrectly).

For each image - template combination, the previously described four steps are executed sequentially, resulting in a collection of ”Good” matches, which represent common features between the image and a specific template. Finally, the most likely template for a provided image is the template that results in the highest number of common features between itself and the image.q Before the execution of the main process loop, all images (provided by the user and available templates) are resized to a new width of 500(experimentally tested), since comparisons of images with similar scales appear to yield more accurate results.

For the secondary feature, as was previously eluded to earlier in the introduction, we decided to move forward with text detection rather than recognition. The methods we explored, and ultimately dismissed, and our thoughts through the process are included below:

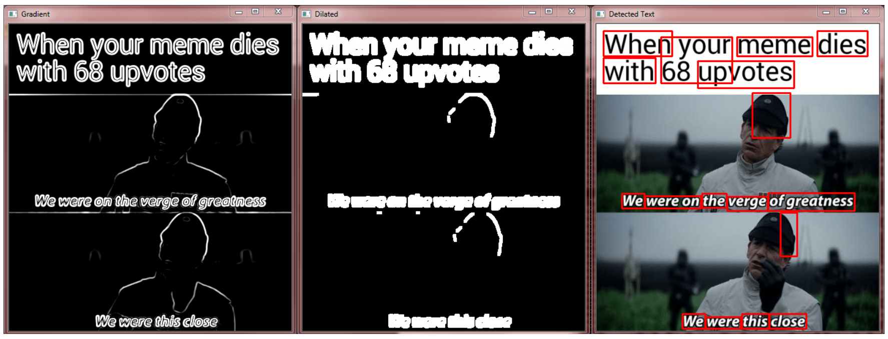
- The next possible method was to use the OpenCV matchTemplate function, which would perform a kind of cross correlation with a target image on the source image, getting a similarity score for each pass, and then returning the position of highest confidence of similarity. This might have worked in some situations, but due to the fact that we need need to check for every character, both uppercase and lowercase, plus some additional symbols and digits, and that it would not necessarily match text of a different font or rotation very well, we decided to pass on it.

- The last method leads into the implementation in the software, but stops before recognition. This method involves using the OpenCV findContours functions to detect discrete objects in the scene. Ideally, it would be used to find windows in the original image of individual characters, and then with a neural network determine the character contained within the window. This would have linked nicely into our neural network class, as we performed digit recognition on the Mnist data set. If it was pre-processed, it might even allow detection of various colours, fonts, and sizes of text. Unfortunately, due to the nature of memes, and the fact that text is often embedded in complex ways with poor contrast to it's surroundings, it would be extremely difficult to perform this task.

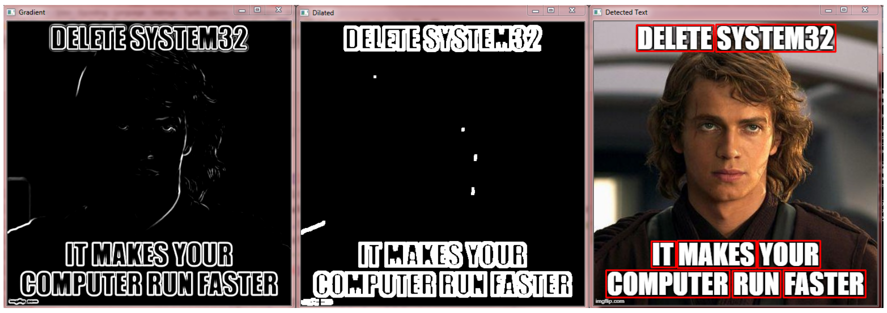
As such, we settled on text detection, merely finding positions of text in the image. Although our results are not perfect, I think it is quite impressive how well the system performs (especially so if care has been taken when making the meme). The method is loosely based on this answer[1] from stack overflow. Essentially, the example begins by converting the image to gray scale, and then thresholding the result multiple times to remove noise. In my testing, this was inadequate, so I decided to take another approach and first performed the first steps of edge detection as we did in our first assignment (blurring image with Gaussian kernel, getting horizontal and vertical gradient with sobel operator, and computing gradient at each point in the image). Results were better if I kept the gradient matrix, rather than perform the final steps of edge detection (such as local maxima suppression). After this, our solutions converge again, utilising OpenCV to dilate the image. This essentially makes all of the edges ”thicker” and removes some unnecessary detail that we do not need. Choosing the right size for the kernel used in this operation is quite important. Since in the ideal situation we wanted the system to identify individual words, we have to make sure that there is not too much bleed as to conjoin words, but also so that characters get combined with the other characters in the same word. This is extremely difficult to do since the size of the text is not known beforehand, and there could be multiple instances of text in the same image of different sizes too. Since this cannot be computed during run time either, we have manually tested various values and found a 5x5 kernel worked fairly well. In addition to this, since the kernel size would be dependent on the resolution of the image, we have also decided to resize all images to be roughly the same size so the parameters are more comparable between instances of running the software. The last computational step is to run the findContours function included with OpenCV. This attempts to find closed shapes in the image, and is the basis for finding regions that have a high probability of containing text. The last step is to perform some thresholding on the contours, only keeping ones that have a reasonable size (aren't too small to be text).

5 Results

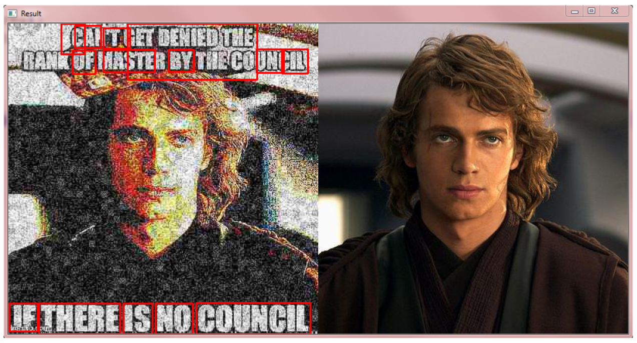
The following is a very good example for text detection. not only did it find the black on white text at the top (the ”your” and ”68” merged because the lines of text are too close together), but it also found the text inside the image. Unfortunately it picked up part of Krenik’s head, due to the high contrast between his cap and background



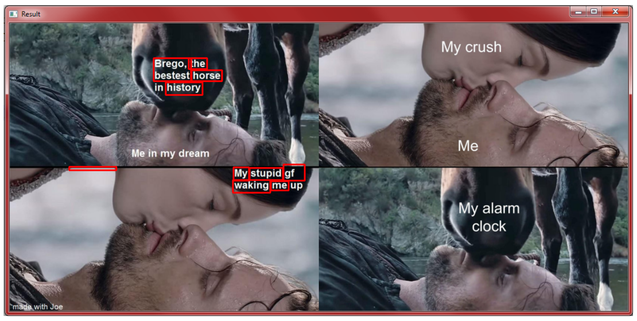
This is another example of very clear text detection with no artifacts.



This is a good example of a very noisey image still being predicted correctly, albeit with top text not correctly captured due to noise and a lot of white around the white text.



And finally, the meme we opened our project proposal with, demonstrating the ability to recognize flipped templates.


Result
Brego, the bestest horse in history
Me in my dream
My crush
Me
My stupid gf waking me up
My alarm clock
made with Joe

Overall we are very pleased with the results. Depending on the quality of the image our detection and recognition accuracy will vary, but for the most part I would consider it to be very successful.

6 List of Work

Equal work was performed by both project members

7 GitHub Page

The source code for this project and some example templates/tests can be found at the following link: https://github.com/LukasJurisica/Meme-Alyzer
There is also video proof of the code running at the same location, and this report has been included for completeness.

8 References

1. https://stackoverflow.com/q/24385714

2. Ethan Rublee, Vincent Rabaud, Kurt Konolige, Gary R. Bradski: ORB: An efficient alternative to SIFT or SURF. ICCV 2011: 2564-2571.

3. Lowe, David. (2004). Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision. 60. 91-. 10.1023/B:VISI.0000029664.99615.94.