Final Project Report

By Khegay Viktor, IT-2101

Link to YouTube video: <https://youtu.be/02Q8B-ad-bk>

Link to GitHub repository: <https://github.com/khegayv/handwritten_tensorflow>

Materials available on GDrive:

<https://drive.google.com/drive/folders/1vWVtvIAQw2uDHLpvTidkq2s5f5qeJIvL?usp=share_link>

**Contents**

[**Introduction** 2](#_Toc127689676)

[**Problem and Existing Solutions** 2](#_Toc127689677)

[**Plan** 2](#_Toc127689678)

[**Data and Methods** 3](#_Toc127689679)

[**First stage** 3](#_Toc127689680)

[**Second stage** 5](#_Toc127689681)

[**Image Extraction Function (Unsuccessful)** 5](#_Toc127689682)

[**Image Extraction Function (Slightly more successful)** 7](#_Toc127689683)

[**Making Predictions** 9](#_Toc127689684)

[**Discussion on Results/Reflection/Next Steps** 10](#_Toc127689685)

[**First Stage Results** 10](#_Toc127689686)

[**Second Stage Results** 11](#_Toc127689687)

[**Reference list** 11](#_Toc127689688)

# **Introduction**

## **Problem and Existing Solutions**

Right inside the project proposal, the goal was to create app that converts **handwritten or image text into copyable text**. While starting designing the model architecture, I understood that OCR model is character based, which means that it retrieves character value from image that contains only **a single** character. After conducting some research, I analyzed solutions found in project proposal stage.

*“Optical Character recognition is one of the oldest technologies found out in sphere of machine learning.*

*Nowadays, a lot of projects are created on the basis of OCR. Mostly, these projects are tools, aimed to make work with documentation much easier.*

*The most popular ones:*

* ***Adobe Acrobat Pro DC.*** *This application has function of paper scanning that uses OCR.* [*https://www.adobe.com/acrobat/acrobat-pro.html*](https://www.adobe.com/acrobat/acrobat-pro.html)
* ***Photomath.*** *Advanced calculator with the feature of scanning needed hand-written problem into its e-format.* [*https://photomath.com/ru*](https://photomath.com/ru)*”*

The problem is that, it is more probable that these apps use not only basic OCR models but more likely to use text detection and text extraction models that are slightly different.

After these findings, I went through several new solution provided by programmers-bloggers.

Kamlesh Solanki [1] provided flask web app solution (you can see in detail by the link in ref. list). Basically, the solution is about simple OCR based on two datasets (numbers and letters) but the text extraction is implemented in web app back-end part. It divides text from the image to custom resized tensors, only after this operation inputs go through the prediction function.

Second solution I consider less flexible, provided by PYlessons [2]. In the basis of the solution lies complete dataset with already prepared words to train on. It is obvious that dataset cannot contain all words, but the beneficial side is that the actual solution is simple enough to implement.

The final solution is much different from the initial. From technical point of view, it is the most straightforward solution I’ve found, provided by Johan Dettmar (dida-do) [3]. The solution is not about giving images as an input but providing raw tensors. The model itself will take the most convenient samples because there is no need to extract characters from the text as the text given by separate characters.

## **Plan**

Finally, after conducting a research and evaluating my skills, I divided project development into the **two main** stages:

1. **Work on base model that will recognize a single handwritten image**: I’m going to use tutorials, mentioned above to develop model that will recognize alphabetic characters and digits.
2. After I receive some practical results from working web app and understand **input formats that model requires**, I will develop a function to make prediction on a string of characters **using the same single-character based** model.

I consider this approach **more efficient** than training model on a dataset that contains prepared word samples and the reason is that the final goal is to create app that will recognize long strings of text.

# **Data and Methods**

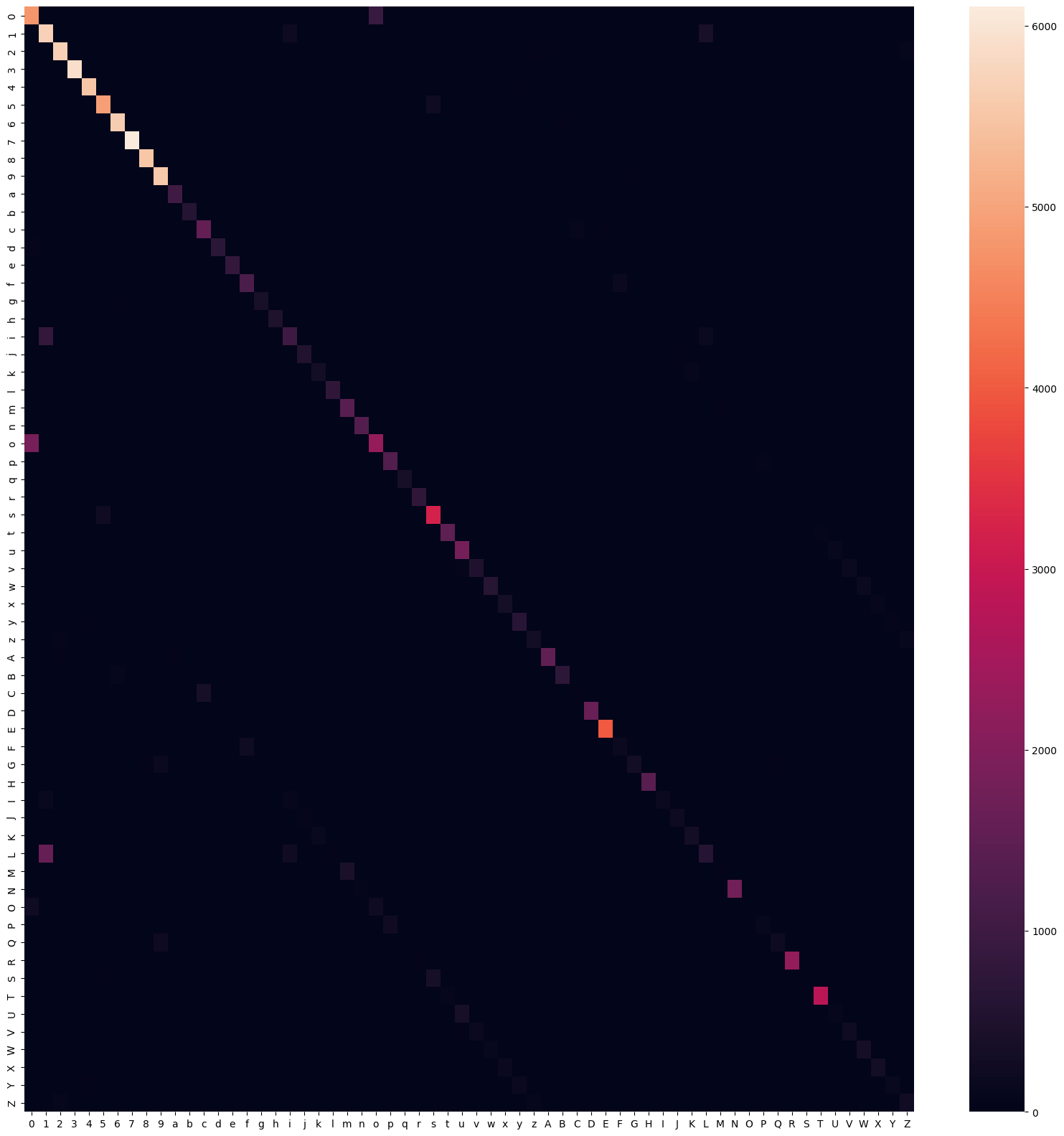
## **First stage**

As a dataset, I used EMNIST, as I mentioned in the project proposal [4]. In general, this dataset is a modified version of popular MNIST dataset. It contains not only 0-9 characters, but extended to contain A-Z and a-z letters inside.

The architecture used in the tutorial a relied on was about transfer learning: the author used tf.keras.applications.resnet [5]. I decided to write relatively small CNN model for this task (adp\_final.ipynb file).

A big confusion about the dataset is mispredictions of letter O (o) with number 0 (zero) and vice versa. Johan Dettmar [3], in his tutorial explains that this might be due to bias that appeared because of “imbalanced amount of training examples.” (watch figure below)

<blob:vscode-webview://0025g5f3k60ec0v6f9c4aed5h88c6fe2535kvstgsoq4lltrtie9/2eb6144d-1b64-4c18-85f1-1bd5621dde1a> (from adp\_fin.ipynb available via github link)



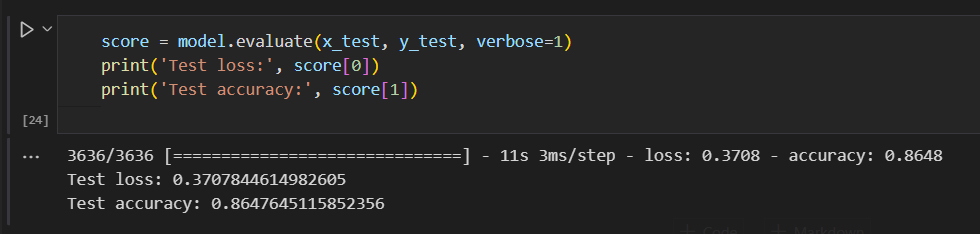
You can observe that confusion appears at “0” and “O” characters.



It is almost impossible to get “O” from the sample, corresponding results are displayed in confusion matrix above.

In general, the overall accuracy of model is low in scope of practical usage because the model makes mistakes more frequently than I supposed.

One more big problem is uppercase/lowercase situation: commonly looking characters as “C” and “c” are hard to distinguish. The picture capture works such way, that it crops out all of the space outside the character and zooms in, so that the size of letters (main feature to distinguish such letters as “C” and “c”) becomes similar for uppercase and lowercase versions of a letter.

In general, model that I implemented in several previous assignments showed result about 86.48% of accuracy for this task. 

## **Second stage**

### **Image Extraction Function (Unsuccessful)**

During the development of the app, I started designing the function to work with not a single, but with a string of characters. **The solution is provided within the “2stage.ipynb” file in submission.**

I decided to use simpler and more accurate model to test the function. The reason is that I was going to design function without any previous research done on dataset images. Thus, for testing new function I used previously created MNIST challenge model.

I found out that it is possible **to use Otsu’s algorithm** to find border of one character at one image, and then **retrieve threshold values** to cut it into separate character-images.

Initially, it was needed to convert image into a grayscale: the reason is in theoretical part – according to Chai (2021), Otsu’s method “can better cope with grayscale images…” [7]:

    grayscaled = cv2.cvtColor(cv2.imread(image\_path), cv2.COLOR\_BGR2GRAY)

Grayscaling applied through .cvtcolor(). Next stage was to use binary threshold towards already converted grayscale image (.threshold() function)

 \_, thresh = cv2.threshold(grayscaled, 0, 255, cv2.THRESH\_BINARY\_INV | cv2.THRESH\_OTSU)

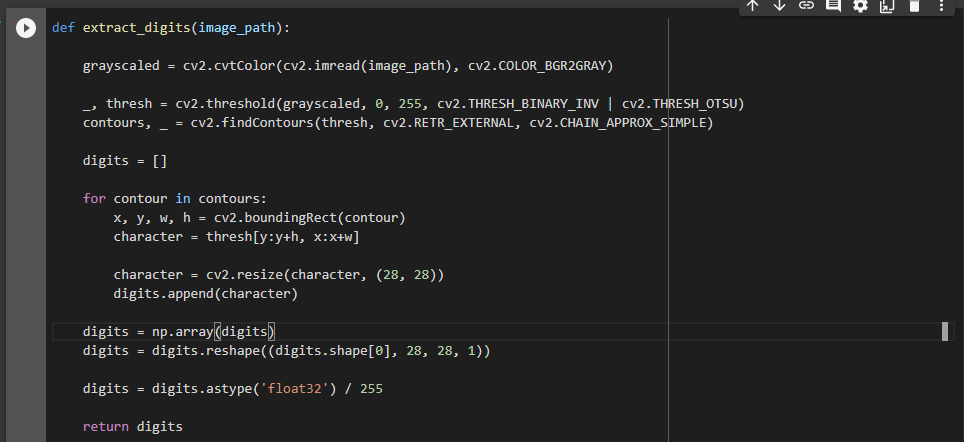
I put grayscaled image as an input and specified flags corresponding to operations available in OpenCV to automatically find thresholds (of course, by Otsu’s method) – “.THRESH\_BINARY\_INV” and “.THRESH\_OTSU”. (This theme is also interesting to explore in a separate work but in the next steps I found out some disadvantages of this method)

Final step was to **define contours** of the image using thresholds values. OpenCV also provides functions that returns list of arrays that contain coordinates of contour points ((x,y) format), so I found it unneccessary to explain code snippet.

Next part of function is to retrieve values to define bounding box for contours I retrieved previously. Obviously, there is one more OpenCV function that helps to define these values: (x,y) tuple of coordinates with width/height of bounding box. This information is enough to extract small image from source image.

In the end of second part of the function only one thing left: preprocess image to 28\*28 size, normalize pixels and get data for all digits that are available to retrieve from source image. First operations I just copied from assignments complete during the course.

The final function looked like this on this stage:



I used several samples from MNIST dataset splited into a single image. It is noticeable that the image is convenient for digit extraction, however if function performs well on this sample I consider it as a good result.

Image I used:



I plotted the outputs based on values retrieved:



Obviously, this result is not what I wanted to see, however positive thing is that separate digits were extracted separately.

### **Image Extraction Function (Slightly more successful)**

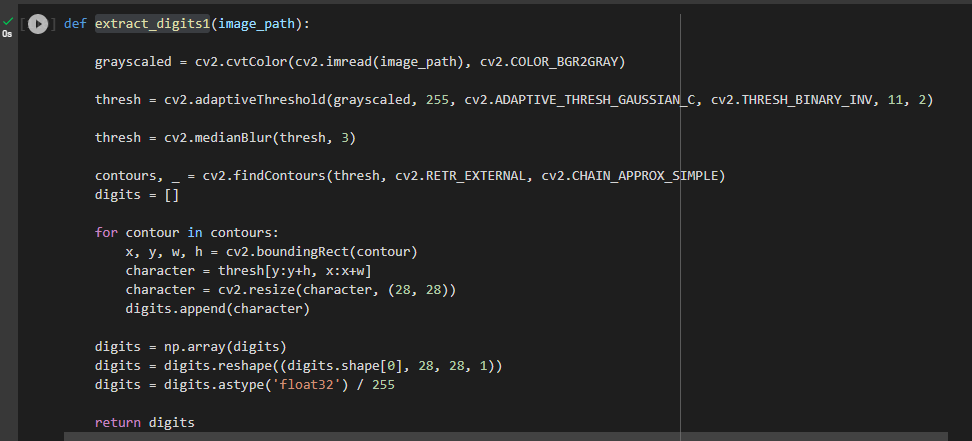
On the basis of the function developed in previous part, I decided just to add somethiing that will help to fix two things:

1. Somehow obtain more precise threshold values
2. Get rid of noise or just try to minimize it

Again, I did a little research to explore other ways to find thresholds and realized that Otsu’s method is not single existing. It is popular because of low computational power needed for it, however I found out one thing as **adaptive thresholding.**

Exactly for sample that I used, there is no a big difference between using either adaptive thresholding or Otsu’s method. I wanted to imlement it for **more realistic samples** that can contain **different lighting** above the image that may bring some difficulties when using Otsu’s method which relies more on computation of threshold **for the whole** image, when adaptive thresholding **will compute thresholds for each pixel** on the area of image. This may take more time, but right at that moment I do not have enough knowledge for optimizing this part.

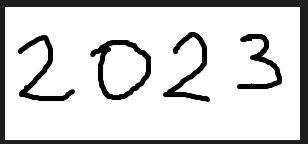
As a small addition, I just found “.medianBlur()” from OpenCV to minimize noise that may appera on the borders or corners of a digit.

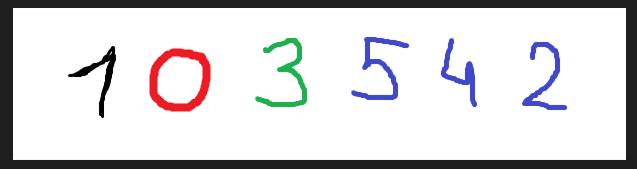
Slightly rewritten function now looks like this: 

And the result it provides became better (at least is more recognizable).



For making sure it works not only for MNIST generated examples, I created several samples in Paint:





## **Making Predictions**

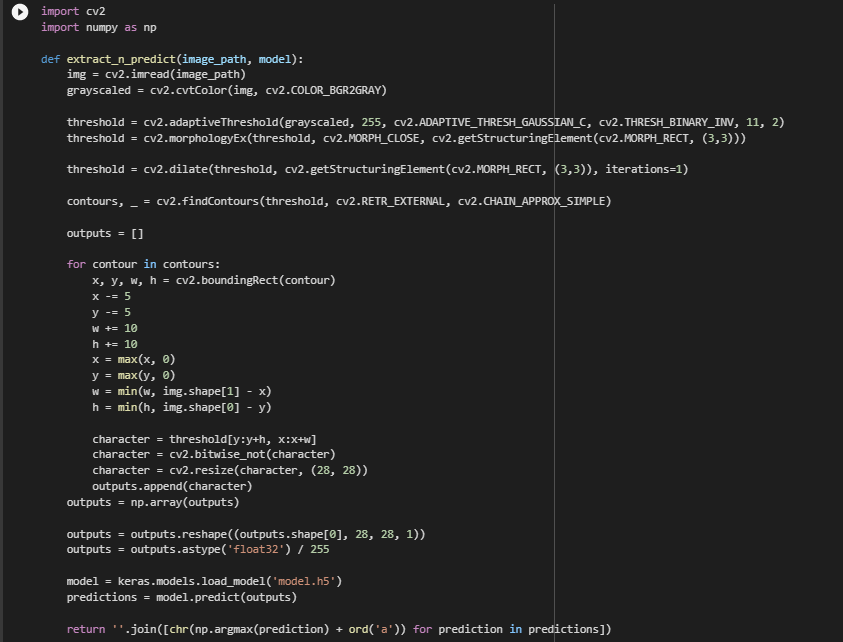
I decided test it immediately on practical dataset that contains alphabetic characters instead of digits. For predictions I will use previiously created extract\_digits1() function, in parallel creating new model based on EMNIST dataset, to predict not only digits but also alphabetic characters.

Before starting prediction part, I noticed, that for some reason, small holes appeared in preprocessed images. I searched for solution and added operations to transform output images: morphological closings for removing lost pixels and dilate function to make character images bolder.

One more thing: I added some margins inside the bounding box.

Finally, I just had to load the model, get extracted images through the loop, predict indexes and concatenate corresponding values together.

Combined function looked like this:



Let’s go through an image:



Despite some successful results, it is really hard to obtain right predictions. The main requirement is to produce convenient inputs that are easy recognizable and can be easily splited to separate character-images. So the conclusion is that this part of the project failed.

# **Discussion on Results/Reflection/Next Steps**

Obviously, I’m not satisfied with work I did. The practical convenience of the app is low because of input format that project provides. The character confusion problem left unsolved. A lot of time been wasted on exploring different solutions done previously.

After 2 stage development, these are results:

## **First Stage Results**

Only first stage ended with successful results. **As a final project I will prepare only this part for a video: prediction for a single image from the canvas-input.** And below are my suggestions on how I’m going to develop this branch of the project.

I want to develop my project in a slightly different way: my next step is to create model that recognizes mathematical operations as inputs, so that it will be possible to make simple calculations. I made a little research on existing datasets and found only advanced samples that include higher level operations [6]. Thus, creating own datasets for four basic arithmetical operations will be definitely enough for me now. I want to create datasets following example of something we did in assignment 2. After making analysis of how I’m going to implement dataset to a model’s input, I highlighted one relevant issue: still imbalance of dataset samples. Such operations as multiplication and division may create character confusion alongside alphabetic characters. Number of samples in custom dataset I can create is incomparable with number of samples that one class in EMNIST dataset contains. That is the main problem I have to make work on.

## **Second Stage Results**

I tried to explain my findings for the second stage as detailed as it possible. The aim of this stage was to prepare project to take not a single-character input, but a whole text, or at list combination of characters on one image. The model I developed will be attached to the submission, its total accuracy is high enough, however the function I developed to extract smaller images from the source image is imperfect and needs reworking. Threshold retrieving algorithm is still missing some points, so that at some cases, less convenient samples showing completely bad results in predictions.

My next step in upgrading this branch of the project is to gain more theoretical knowledge to write adaptive function to extract single-characters from image of different convenience levels (I mean different lightings, angles and etc.)

# **Reference list**

1. Solanki, K.. OCR Using Tensorflow. <https://medium.com/analytics-vidhya/optical-character-recognition-using-tensorflow-533061285dd3>
2. Pylessons. Text Recognition with Tensorflow and CTC Network. <https://pylessons.com/ctc-text-recognition>
3. Dettmar, J.. How to Distribute Tensorflow model as a Javascript Web App. <https://dida.do/blog/how-to-distribute-a-tensorflow-model-as-a-javascript-web-app>
4. EMNIST. <https://arxiv.org/pdf/1702.05373.pdf>
5. Cornell University. Deep Residual Learning for Image Recognition. <https://arxiv.org/abs/1512.03385>
6. Kaggle. Handwritten Math Symbols. <https://datasetsearch.research.google.com/search?query=Handwritten%20math%20symbols%20dataset&docid=L2cvMTFuZ3MzY3dxbQ%3D%3D>
7. Chai, R. (2021). Cognitive Computing Solutions for Complexity Problems in Computational Social Systems. Otsu’s Image Segmentation Algorithm with Memory-Based Fruit Fly Optimization Algorithm. <https://www.hindawi.com/journals/complexity/2021/5564690/>