Brian Cornet
DSC 499
Dr. Donghui Yan, Dr. Scott Field
4/7/21

Capstone Project II: Final Report

Project Title: Optical Character Recognition and Correction for Images of Digital Displays

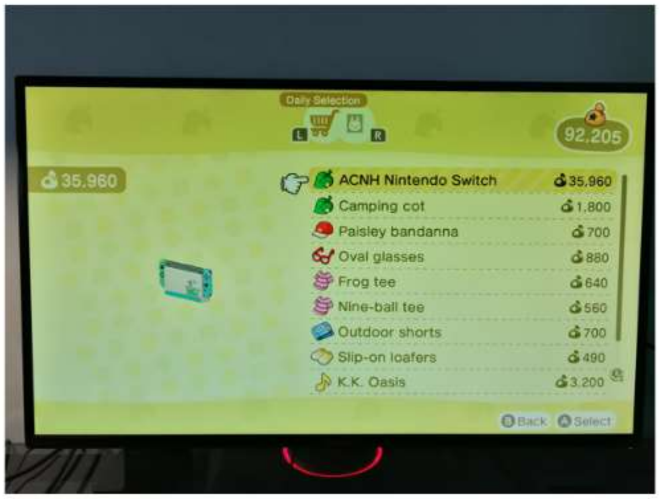
Abstract

Data structures such as tables and lists are commonly presented through graphical user interfaces (GUIs). In many cases, accessing the data directly is inconvenient or prohibited by the platform itself (Apple iOS, Nintendo consoles, etc.) or unavailable locally as with video streaming. Notably, the data is not private to the user, but collection must be performed by hand. This project introduces a program for Optical Character Recognition (OCR) using Tesseract 4 and Python to convert data displayed in images or videos into discrete structures. Image transformations such as binarization, scaling, skewing, edge detection, and denoising enable compatibility with elaborate GUIs and photographs of digital displays. Predictive methods using Levenshtein and character-shape distances provide text correction to significantly improve output accuracy. Error analysis is also conducted for individual correction methods using several sample images, dictionaries, and fonts.

Introduction

This project aimed to create a data collection method for camera images and videos of digital displays. Specifically, it intended for a user to be able to provide an input (an image, video, etc.) of a source device's display (a television, monitor, phone screen, etc.) that contains some kind of quantitative data collection (a list or table of text strings, a collection of images, etc.) to collect and organize that data. The intent was to provide a quick means of copying structured information from one device to another for instances where accessing the raw data directly is inconvenient or prohibited. Typically, this restriction is enforced by design for closed platforms such as video game consoles (Nintendo Switch, Sony PlayStation, etc.) and iOS devices from Apple Inc., file formats that may be encrypted, or data streams that may not be recorded locally. Furthermore, many users lack the technical skills to access and manipulate arbitrary data structures directly and instead rely on existing software appropriate to the file format to do so. In all instances, the data itself is not private to the user, but any collection will have to be done by hand.

For example, a user may be playing a video game with the goal of collecting over 10,000 distinct items. The game provides a text list of every item obtained thus far (displaying 10 items at a time) but offers no hints towards which items are missing or how many items remain (i.e. a list of distinct items $N =$ {“Apple”, “Ball”, “Car”} with no indications as to whether “Apricot” or “Xylophone” are in $\overline{N}$ or how large $|N \cup \overline{N}|$ may be). Suppose the user wants to make a checklist of which items they currently had and cross-reference it with a list of known items obtained online to determine which items they’re missing. While the data exists in the internal memory as a sequence of binary values (0 = missing, 1 = obtained), the internal memory itself is inaccessible to the user. The user also has thousands of items, many of which have similar names (“Wooden End Table”, “Wooden Low Table”), are difficult to spell or type (“Myllokunmingia”, “Übel/Equité“), or are simply long (“Archive Tower Giant Door Key”, ”Imperial Army Identification Tag”), making compilation by hand tedious and prone to errors. Instead, the user could easily record a video of themselves scrolling through the entire list with their personal smartphone, creating a digital basis for their data. By using optical character recognition (OCR) methods, the program would examine the frames of the video for text strings representative of every item the user has obtained, transforming their original digital basis into a more concise and appropriate data structure for the cross-reference operation.



ACNH Nintendo Switch	35.960
Camping cot	1.800
Paisley bandanna	700
Oval glasses	880
Frog tee	640
Nine-ball tee	560
Outdoor shorts	700
Slip-on loafers	490
K.K. Oasis	3.200

$\longrightarrow$

ACNH Nintendo Switch
Camping cot
Paisley bandanna
Oval glasses
Frog tee
Nine-ball tee
Outdoor shorts
Slip-on loafers
K.K. Oasis

Figure 1. Converting an image of an item list into a set of strings. Image from Animal Crossing: New Horizons, Nintendo 2020.

A screen capture from any digital display would be preferable for an accurate interpretation over a photograph of a screen. Generally, most captured text and graphics will be static, smooth, properly aligned, correctly sized, framed, and free of environmental noise or errors. Camera images meanwhile are subject to any number of errors caused by the angle of the lens relative to the screen, the lighting of the environment, objects outside the display's boundaries (including the display's frame), unsteady positioning when held by hand, detection of interlacing between frames and display pixels (moiré), or filters and corrections applied by the camera's software. However, one of the primary objectives of this software was to improve the convenience of collecting data, and not all users will have access to screen capturing and transfer methods. Comparatively, mobile phones equipped with cameras are extremely popular and convenient. Compatibility with camera images was thus deemed a major goal of the project.

The bulk of programming work in this project was conducted in Python due to the abundance of machine learning libraries associated with image recognition and correction. OCR for text-based images was achieved through Tesseract 4 and implemented in Python using the *PyTesseract* library. Per Tesseract's documentation, retraining methods were not explored as they are considered unlikely to help improve output accuracy. Instead, images themselves were transformed in various ways to improve quality and allow Tesseract to better "see" the appropriate text. Image correction methods are mandatory in the correction of camera images regardless and became the initial focus of this project. Several individual methods for image correction were employed with varying degrees of success, the majority of which saw implementation through OpenCV's *cv2* library in Python.

Partway through development, focus shifted towards implementing dictionary-based methods for text correction. OCR errors are commonplace, even when parsing text from flawless images. Assuming the user knows every possible character or string for a single block, a method for comparing the OCR result with a list of these possibilities to determine the best match could provide the most likely interpretation. Supporting this idea would be a method for recognizing screens by form, allowing the program to remember text region locations and their associated transformations and dictionaries for improved performance speed and accuracy.

It should be noted that this program does not take advantage of advanced machine learning techniques (specifically convolutional neural networks) beyond the implementation of Tesseract and OpenCV. While the efficacy of such methods may improve the output, a lack of formal training and experience with these methods until later in the project's lifespan was recognized as a limiting factor early in development. This project was meant to be a proof-of-concept – these methods were deliberately unexplored and left for any future works.

Image Correction Methods

OCR output relies on two specific tasks: recognizing the locations of text in an image (**location identification**) and correctly identifying the characters in these locations (**character identification**). Short of developing a new detection model or direct user intervention, image correction is notable for being the primary means of improving location identification. Subsequently, character identification improves when applying these methods, though some design decisions (unusual fonts, noisy backgrounds, etc.) typically limit the upper accuracy bounds. While Tesseract applies some of these methods to an individually scanned section, the platform's documentation recommends independently improving the quality of the image prior to scanning. Three such quality improvement methods were explored in this project: binarization, rotation, skewing, and noise removal. Other unexplored methods such as dilation, erosion, border addition, and border removal may be considered in future works.

Binarization

This process assigns each pixel to be either black or white, providing a clear contrast between edges and making shape recognition easier. Tesseract natively performs this through Otsu's method but recommends other methods for uneven background colors from the OpenCV and scikit-image modules. Note that OCR generally performs better when examining dark text on a light background; light text on dark backgrounds can be easily corrected by inverting the colors on an entire image or section.

One method explored was to separate the image into bitplanes by splitting the color values of each pixel into its own binary image (**bitplane separation**). Input images are assumed to have a 24-bit color map, with 8 bits (1 byte) assigned to red, green, and blue color brightness respectively. In many cases, the most significant bits define the most significant features such that scanning these sub-images may have better output than scanning the original image. Applying a grayscale filter to the image and collecting the most significant bit of the newly-created 8-bit color map provides an alternative method of evaluating these bits with less run time. These methods were very successful in images where the text and background colors are consistent, though eliminating less significant bit values managed to reduce some noise effects such as those introduced from camera images.

Another method is **thresholding**, where the value at each pixel is rounded to 0 or 255 based on a given brightness cutoff. Iteration over a range of cutoffs can be performed over a given image to determine a likely prediction. The output of each scan at each threshold is recorded, and the most commonly-reported output at each section is used as the prediction. Using this method in conjunction with bitplane separation and grayscale filtering can provide an incredibly large sample size of outputs to consider and use in predictions. However, the relatively long runtime for each scan encourages the use of a limited selection of possibilities. This can be found using a smaller set of sections within the image. For example, using the sample image “mhgu3.png”, the accuracies for each of the five values over a threshold range of 1 to 254 were collected by comparing the Levenshtein distance between the outputs and the actual values. The threshold range between 167 to 176 was the largest range with the highest accuracy between these values, and so it is used in evaluating “mhgu2.png” (from which the previous image originates).

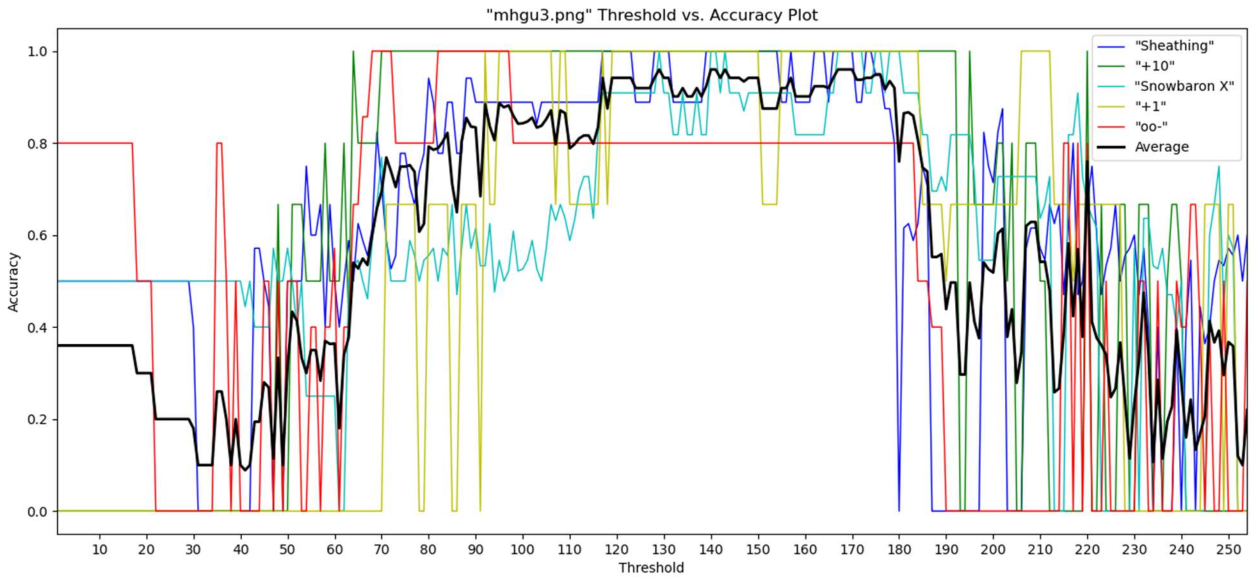


Figure 2. Evaluating OCR accuracy at threshold values from 1 to 254.

Rotation and Skewing

All camera images are taken at some angle relative to their display – no human should be expected to hold or mount a camera perfectly parallel to a screen. To account for this, rotation and skewing methods are needed to realign the image so that Tesseract can interpret the shapes of the characters correctly. Frequently, images where the region of interest (ROI) is only a small part of the screen will see more difficulty in performing these methods reliably. Cropping out the relevant section manually may be necessary in such cases.

One successful rotation and skewing method was **matching**, where similar features are detected between the camera image and a template image using the **Features from Accelerated Segment Test (FAST)**, **Binary Robust Independent Elementary Features (BRIEF)**, and **Oriented FAST and Rotated BRIEF (ORB)** algorithms. The locations of these features (as x,y coordinates) are used to create transformation matrices that rotate and skew the camera image to match the orientation of the template. This is usually successful when a clean template image is available (such as an empty table using the same design as the camera image) and neither image has too many or too few features. Given that these prerequisites may not always be realistic, other methods must be considered.

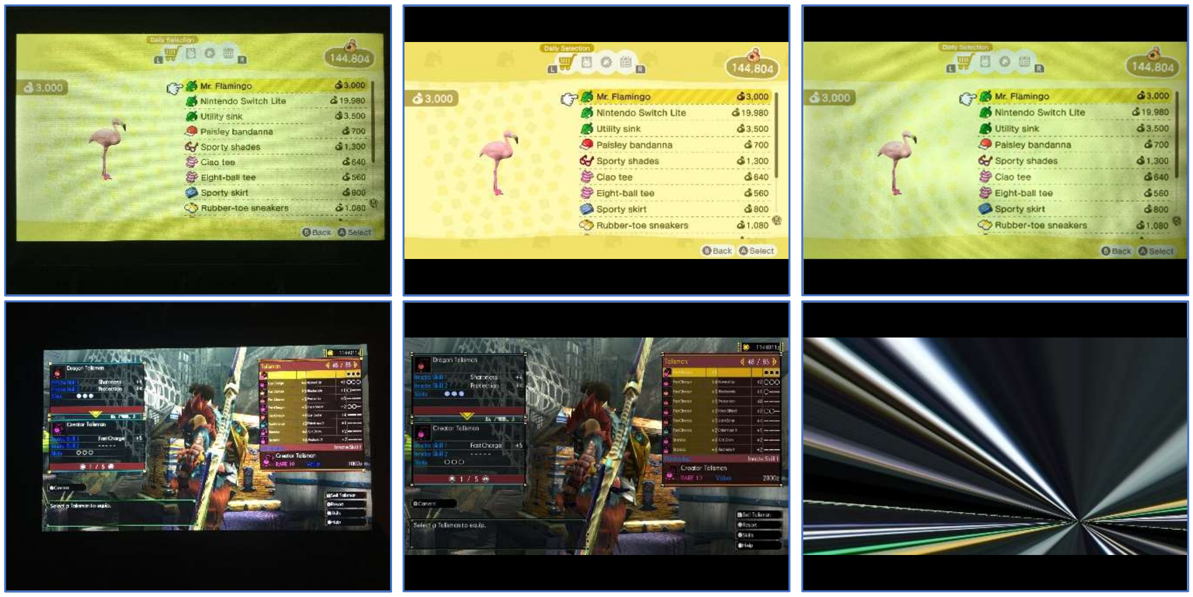


Figure 3. Matching attempts on images from Animal Crossing: New Horizons *(Nintendo, 2020) and* Monster Hunter Generations Ultimate *(Capcom, 2018).*

Another method involves using **Canny edge detection** and **Hough lines** to determine the location of the screen boundary or major lines that indicate the direction of the x and y axis on the original display. The former of these methods relies on a series of transformations: first, a **grayscale filtered** image is smoothed using a **Gaussian filter**, then **Sobel filters** are applied to determine horizontal and vertical derivatives which are used to find intensity gradients. **Lower bound cut-off suppression** is applied such that only pixels brighter than both of its adjacent pixels (adjacency direction based on the angle θ returned by the arctangent of the two derivatives) retain their gradient value. **Double thresholding**, where values are mapped into “suppressed” (0), “weak” (0 <= *n* <= 255), and “strong” (255) categories, is applied based on cutoff values relative to the brightest gradient. Finally, edge tracking by **hysteresis** maps any weak pixels to strong if any adjacent pixels are strong (8-directional) and suppresses it otherwise. An additional method to filter out interior Canny edges can also be applied, though this assumes no irrelevant exterior Canny edges are detected (typically objects and regions outside of the display device such as walls, furniture, or the user’s body). Hough lines can then be predicted, allowing the **Harris** or **Shi-Tomashi corner detection** methods to calculate the parameters used in the rotation and skewing process. Note that radial distortion from certain camera lens can introduce curving which affects the accuracy of the Hough line prediction; methods for removing this distortion were not fully realized at this time.

Noise Removal

Image noise (random variation in color and brightness) is introduced when taking any camera image and should be eliminated whenever possible. Additionally, some interfaces feature “noisy” fonts or backgrounds, especially those using transparency over some other image layer. Camera images of digital displays also usually introduce moiré patterns from overlaying the camera and display’s scan lines, which may be trickier to eliminate in particularly heavy regions. Typically, the simplest way to remove noise is to use blurring filters (such as Gaussian filters) and binarization methods described previously. Taking pictures with the camera rotated 30° and parallel to the screen with the ROI in the center of the image should ideally minimize moiré patterns. High brightness on the display, low environmental lighting, and a clean screen free from scratches, smudges, dust, etc. are also recommended for these images.

Dictionary-based Correction Methods

In many cases, no amount of image correction will improve OCR output accuracy. This is common when scanning unusual fonts and symbols or shapes. Rather than focusing on the input, another transformation layer can be added to the output itself. Assuming the results of an individual scan has a predetermined set of possible values (e.g. only numeric characters), it should be possible to determine which of these values is the most likely interpretation based on some distance metric between the two strings. This requires three components: the OCR output string x, the set of possible distinct string values $Y = \{y_1, y_2, \dots , y_n\}$, and the distance function $f(x, y)$ to compare the two. This should provide a result set Z of similarity measurements such that $z_i = f(x, y_i), 0 \leq z_i \leq 1$. The program then selects the optimal string y_i as the string corresponding to the highest similarity determined as $z_i = argmax_y f(x, y)$. It should be assumed that $f(x, y) = 1 \leftrightarrow x = y$ given all distinct values of y and $|x| = 0 \wedge |y| > 0 \rightarrow f(x, y) = 0$. Lastly, $f(x, y_i) = f\left(x, y_j\right) \nrightarrow y_i = y_j$. Note that this requires additional input on behalf of the user which may not be possible in all circumstances.

Implementation and Limits

For this project, the **Levenshtein ratio** is used to calculate the relative distance between strings. The **Levenshtein distance** is calculated as the total number of edits needed to one string to match the other, where an edit may be a character insertion, deletion, or substitution, whereas the Levenshtein ratio divides this distance by the combined lengths of both strings. Other string distance methods may be considered such as **Damerau-Levenshtein distance** (which allows transposition for adjacent characters as those are common in human spelling errors). Given the independence of individual character errors relative to their neighbors in OCR, such methods may be less accurate here.

Testing with the Levenshtein distance method showed promising results when applied to dictionaries with large, varied strings. The list of skill names from *Monster Hunter Generations Ultimate*, along with the possible numeric ranges (-10 to +13 excluding 0, includes sign on positive values) and slot symbol values ("---" for zero slots, "o--" for one, "oo-" for two, "ooo" for three) were used as dictionaries for the image "mhgu2.png" to correct predictions made over 10 thresholds. Without dictionary support, the reported accuracy was 86.41% (89.74% for skill name columns 1 and 3). With dictionary support, accuracy improved to 92.96% overall (100% for skill name columns). All errors stemmed from the software's inability to properly identify symbols such as "+", "-", and "o" in the 2nd, 4th, and 5th columns, as well as the dictionaries' failures in handling distance ties. For example, if the OCR output of a scan is "5", then the dictionary cannot determine whether "-5" or "+5 is more accurate. An examination of each dictionary suggests that accuracy improves when using dictionaries with longer string lengths and a lower mean distance ratio between all possible combinations. For example, a list of all U.S. state names ("Massachusetts", "Louisiana", etc.) would see far better results than a list of their abbreviations ("MA", "LA", etc.). Similarly, a **Levenshtein distance matrix** as a heatmap may be used to visualize the overall similarity between possible terms.

Character-shape Tiebreaking

In cases where the Levenshtein distance is unable to select a single maximum, an alternative tiebreaking solution may be applied at a character level. Rather than comparing the number of edits needed to correct a word, the images of the dissenting characters themselves are compared. For example, "+4" and "-4" both have a Levenshtein distance of 0.5 with "$4". In such a case, the dictionary would normally return the first match by default. Implementing the character-shape tiebreaker would instead try to compare the shape distance between "$" with "+" and "-", where each potential shape is drawn (using either a provided font or some common font with similar shapes) as a numeric matrix. The simplest method for determining distances is to sum the square difference at each pixel, though additional methods and transformations may also be implemented. The character with the smallest distance is then selected for the final predicted result. Note that this method is sensitive to the style of the font itself: variations from serifs, character designs (compare "a" with "**ɑ**" or "0" with "0"), and nonstandard characters (Greek letters, accents, umlauts, etc.) will have a significant impact on the similarity between various characters.

This method has no effect on OCR empty string results, nor can it improve accuracy against missing characters – a result of “4” would still be unable to differentiate between “-4” and “+4”. One solution: assume that a minimum string length must be met based on the shortest item in the dictionary. If the OCR result produces a string shorter than the minimum, assume that the empty characters must be replaced with the character with the most empty space from the set of possible characters in that position. The test image “mhgu2.png” was able to achieve 100% accuracy when utilizing this method to account for the inability to offer corrections with the dictionary method. Notably, this method was shown to improve accuracy when ignoring the character filter option in Tesseract itself, which appears to favor dropping invalid character strings rather than approximating them.

Conclusion

Though the final platform has not fully developed for a truly autonomous text extraction method, the methods utilized in this project demonstrate the viability of such a program. More testing with other sample images and methods is required, however, as is the necessity for algorithms to optimize performance speed and accuracy. Experiments on videos and the development of video-specific methods should be considered. In addition, further structural improvements should be made to the code itself to enable software creation (along with the implementation of a UI). The originally-proposed “spot the differences” feature for comparing similar images and the collection of unstructured lists may be implemented as well with many of the same methods discussed here.