# HCIRA Project #2 Final Report

Sri Chaitanya Nulu

**Abstract**—In this project, the $1 Unistroke + Protractor Recognizer Algorithm was implemented along with visualizing the steps in both algorithms. For testing and comparing, two different datasets were used: the Unistroke gesture logs dataset provided on the website, and a custom dataset consisting of the gestures of six different users was collected. The total average scores and recognition time computed by $1 algorithm were 0.985 and 38.727 ms respectively for the Unistroke gesture logs data set, and 0.919 and 35.759 ms respectively for the collected custom data set and for $1+Protractor algorithm they are 0.992 and 0.403 ms respectively for the Unistroke gesture logs data set, and 0.92 and 0.432 ms respectively for the collected custom data set.

## 1 Introduction

In the current scenario, many people try to use Machine Learning for every single task. There are many ways one can recognize gestures in this way, but for that to happen, we need a lot of data. And collecting such large datasets or creating them is easier said than done. This project implements an algorithm called $1+Protractor Recognizer [1, 4, 8]. It is simple to understand, faster than $1 Recognizer, doesn't need a lot of data (just one example per gesture), and gives accurate results.

This algorithm is completely implemented using Python language with the help of some external libraries like Tkinter [5] and PIL [6] to implement the GUI. External libraries like Seaborn and MatPlotLib were used for plotting graphs and visualizing the steps in the algorithm. For testing purposes, the Unistroke gesture logs dataset provided on the website [4], and a custom collected dataset (consisting of the gestures of five different users) were used. These datasets record some details about the gestures such as the number of points, time, points, etc. These datasets are then used to do offline testing. We also perform live testing where a user can draw the gesture, and both $1 and protractor algorithms predicts the result immediately and display it.

## 2 Related Work

This project was implemented by understanding the "$1 Recognizer for User Interface Prototypes" paper. It describes the major steps and provides the pseudo-code for all the functions. The implementation of the program is also provided in JavaScript on their website [4]. The $1 Recognition Algorithm has a set of templates for each supported gesture. And for the given test template, it processes the points and then tries to determine how close it is to the default templates set. This is done by computing the Euclidean distance between them and choosing the shape that is closest to the given test template [1,4].

Once $1 is implemented, the protractor algorithm was integrated into the program. Everything about protractor was understood and implemented with the help of paper, "Protractor: A Fast and Accurate Gesture Recognizer". Its approach is like $1 Recognizer, but some steps vary. One of the notable differences is that we use Optimal Cosine Distance instead of Euclidean distance. More details about protractor will be discussed in the further sections [8].

One of the tools called GECKo (GEsture Clustering toolKit) made it easier to understand how users draw the gestures from an input dataset. This tool visualizes the order, direction and shows how the gesture was drawn by providing a playback. It also helps to modify the clusters if necessary. This step was necessary after custom dataset collection because we need to check if all the gestures drawn by the users were drawn properly [2].

To identify the variation of the user-drawn gestures, GHoST (Gesture HeatmapS Toolkit) tool was used. This tool takes gesture datasets within a user or among different users and shows the articulation properties of the gestures via gesture heatmaps. These heatmaps are colored, and the color depends on how the variation is among the gestures drawn by the user(s). We can also see the gestures in the background if needed. This tool helped in getting several insights which are discussed in the other sections [3].

## 3 Dataset Details

### 3.1 Existing Dataset

The first dataset that was used to perform the offline test was the Unistroke gesture logs dataset provided by the authors on the website [7]. This dataset consists of 16 different gestures from 11 different users. Each gesture is drawn 10 times at 3 different speeds. All these gestures are sorted into folders accordingly. For this project, the medium speed dataset of the last 10 users was used (The first user was already used initially as a template).

All the gesture files are recorded as XML files. They have information about the gesture shape, sample number, speed, unique user ID, time taken to draw the ges-

ture, and all the points along the path of that gesture. During the offline recognition, each of these XML files is analyzed and a corresponding row with some set of output columns (discussed in the following sections) is written as output.

### 3.2 New Dataset Collected

Six different users were asked to draw gestures. They all used a stylus to draw all 16 gestures. Each user drew 10 samples for each gesture. The program's interface lets them choose any gesture in any order. These strokes were recorded in the same format as the Unistroke dataset [7], with some differences. This dataset didn't require users to draw gestures at a different speeds or records date and time.

The XML files saved details about the gesture name and sample number along with the unique user Id and the number of points. Then the points were written inside the file. These files were processed just like how the Unistroke Dataset was processed by the offline recognition module of the program.

## 4 Implementation Details

### 4.1 Language and Code Structure

As mentioned earlier, the complete program is written using Python. There is a canvas implemented in the program, which allows users to draw shapes over it. The program is divided into the following modules [1,4] (also see Figure 1):

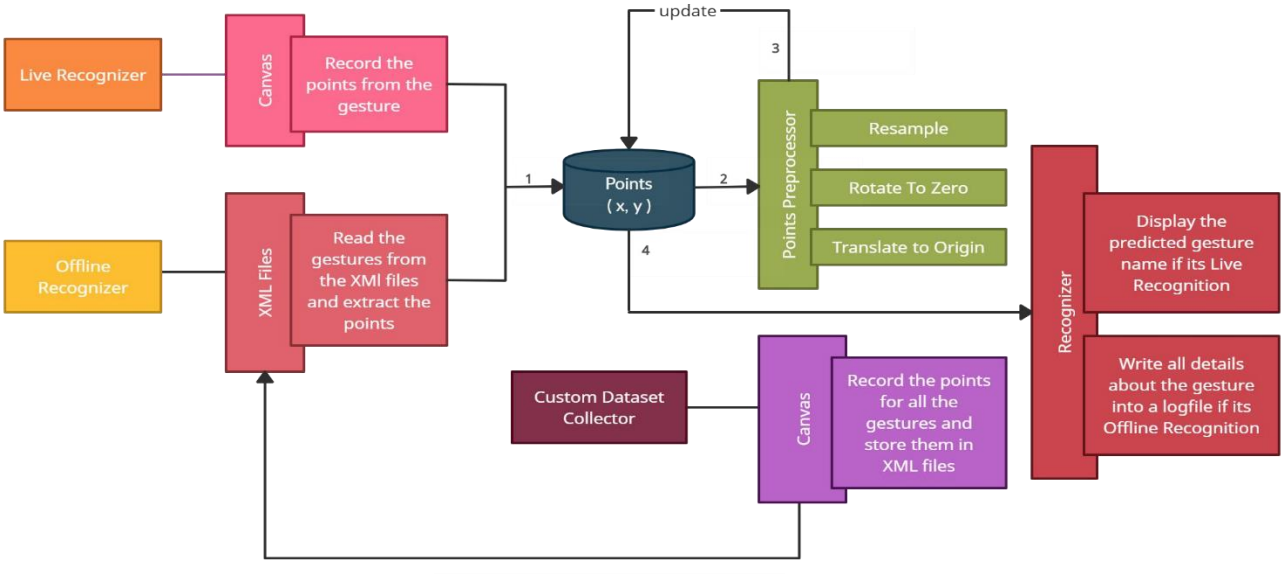


Figure 1: System Architecture Diagram

- **Points Preprocessor:**

  All the points recorded on the canvas are sent here for preprocessing. Depending on the algorithm used, the steps vary. These are the following steps for $1 and Protractor algorithms:

  **$1 Recognizer Algorithm**
  - First, we resample the points so that there are only 64 equidistant points.
  - Then we rotate the whole shape such that the indicative angle is at 0 degrees.
  - Finally, we scale all the points so that the sizes match and then translate to origin.

  **Protractor Algorithm**
  - First, we resample the points so that there are only 16 equidistant points.
  - Then we vectorize these points. In this step, we first find the centroid and and translate the points to origin. After that we generate a normalized vector of length 32 using these 16 points [8].

- **Gesture Recognizer:**

  It takes the preprocessed points and then compares them with all the templates. It tries to find which template is closest to this set of points. If the algorithm used is $1 Recognizer, then Euclidean distance is employed, or else in case of Protractor, we use Optimal Cosine Distance.

- **Online Recognizer:**

  The program displays the canvas to the user where they try to draw a gesture. These points are recorded once the user finishes drawing them and sent to preprocessing. Both $1 and protractor try to find the best match from the templates and displays the shape name with accuracy and time on to the screen. The user can see both results from $1 and

protractor.

- **Visualization:**
  Once the online recognition is done, the user gets an option to visualize all the steps in both $1 and protractor algorithms. Then the program displays an image showing how each step is performed. Once that's done, the user can also see the template matching step with that all the default templates compared against the gesture they had drawn.
- **Offline Recognizer:**
  It scans the specified folders in the path and reads all the XML files one by one. It predicts the shapes and saves it along with many other details (user id, n-best list, etc.) into a log file (csv format). We repeat the recognition process by increasing the number of training sets for each gesture from 1 to 9. This module iterates for 100 times and the accuracy is then averaged. Average recognition time is also stored to compare both $1 and protractor algorithms.
- **Custom Dataset Collector:**
  It provides interface for the user to draw all the gestures. Once the points are recorded, they are written into an XML file with details like shape name, sample number, number of points, time taken to draw, and all the points.

### 4.2 Runnable Components

Figure 2 shows the default interface of the program. There are four different buttons: Live Recognition, Offline Recognition, Offline Recognition on Custom Dataset, and Custom Dataset Collection [1, 4]. There is a checkbox to toggleProtractor algorithm for offline recognition. ust like the names suggest, each button does exactly what they mean.

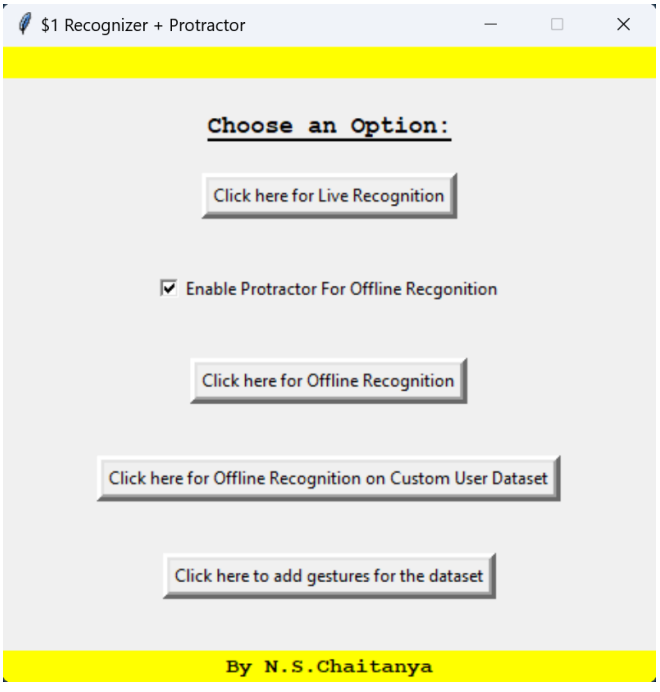


**Figure 2: Default Program Interface**

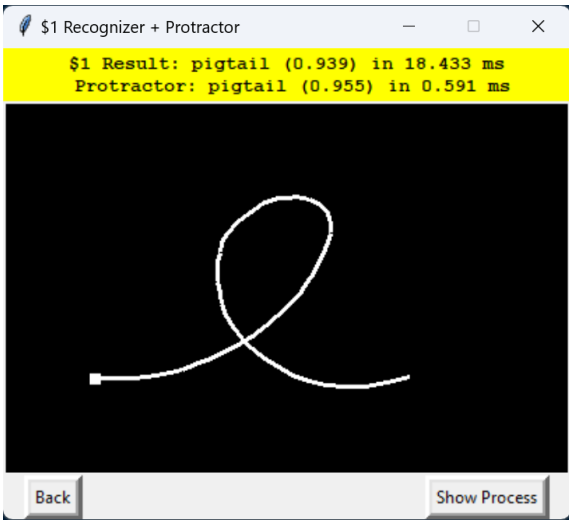


**Figure 3: Live Recognition**

Figure 3 shows what happens during live recognition. The program predicted the gesture drawn as pigtail along with accuracy and time taken for both $1 and protractor algorithms. You can also see that there is a "show process" button. This button will show the visualization of the steps for both $1 and protractor algorithms.

Figure 4 and Figure 5 demonstrate the offline recognition process for the protractor algorithm on Custom Dataset.

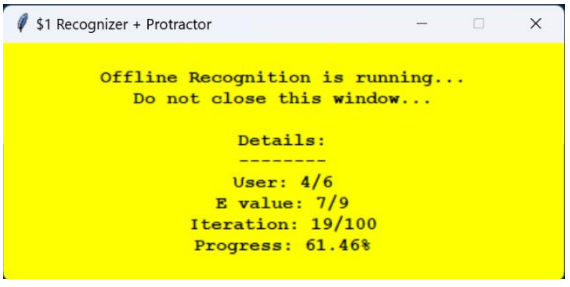


**Figure 4: Offline Recognition by Protractor (In-progress)**

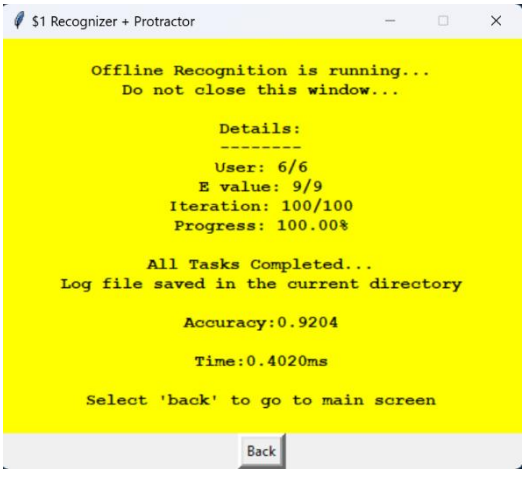


**Figure 5: Offline Recognition by Protractor (End)**

As shown in figure 3, there is button called “show process”. When clicked, this button shows the steps in both $1 and protractor algorithms. The following figures demonstrate the process in both algorithms.

In figure 6 and 8, we can see the $1 and protractor Algorithm Process respectively. All the major steps in these algorithms are visualized. And in figure 7 and 9, we see the user gesture is being checked for the best match with all the templates.

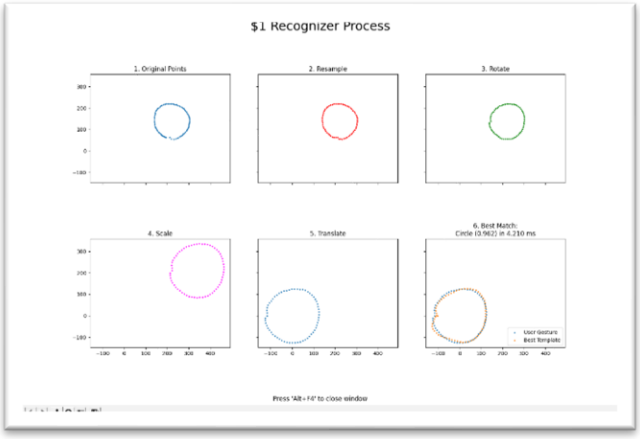


**Figure 6: Steps in $1 Algorithm**

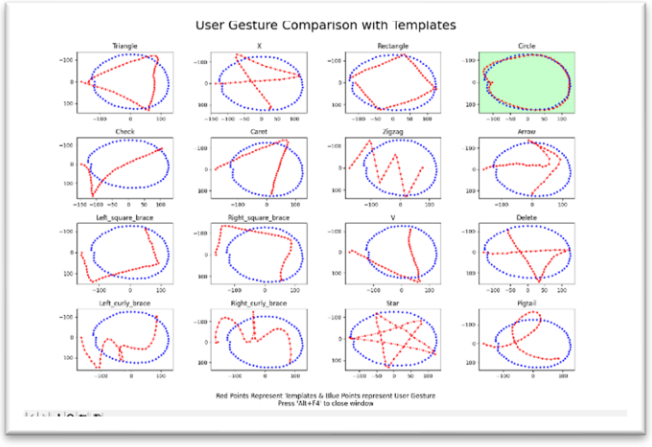


**Figure 7: Template Matching in $1 Algorithm**

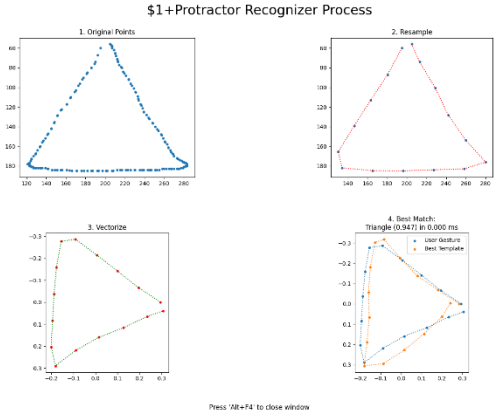


**Figure 8: Steps in Protractor Algorithm**

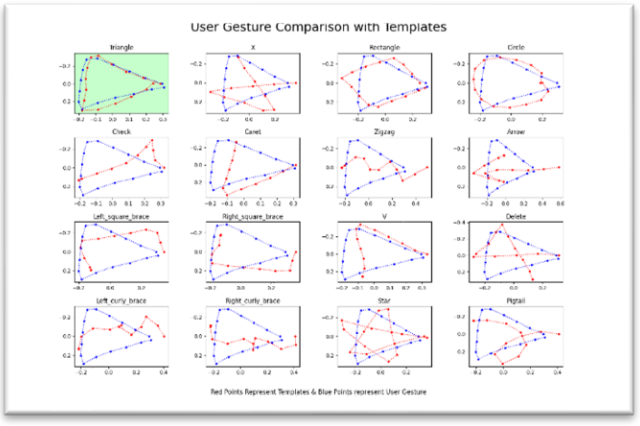


**Figure 9: Template Matching in Protractor Algorithm**

### 4.3 Implementation Challenges and Solutions

Some of the challenges were:

- Integrating protractor into existing code and visualization of the steps and process.
- Understanding why the custom dataset has lower accuracy than the Unistroke dataset.
- Understanding why certain gestures have high mismatch count than others.

These were solved by creating heatmaps shown in the next sections and in appendices.

## 5 Offline Recognition Results

For the offline recognition results, the program checks each gesture drawn by the user stored in an XML file, it then runs the offline recognition for all of them and writes some details into a log file.

These details include User Id, Iteration Number, Number of training Examples, Total Size of Training Set, Candidate (Current user gesture), Recognition Result, A boolean variable (0 or 1) to check if the recognition is correct or not, Recognition Score, Recognition Time, Recognition template that was the best match and the N-best Sorted list containing all templates in decreasing order of the scores.

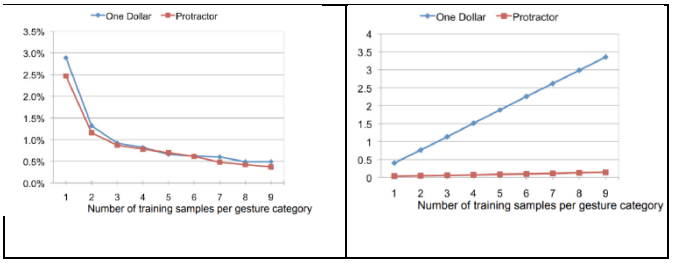


**Figure 11: Graphs from [8] showing The Error Rate (left) and Recognition Time (right) as the number of Templates increase (Unistroke Dataset).**

It was observed that as the number of training examples increased, the error rate decreased, while the recognition time increased. This can be observed in the Figures 11 (Taken from Paper), 12 and 13 (Implemented). This is because as the number of training examples increase, the algorithm has more templates to compare them with the test gesture.

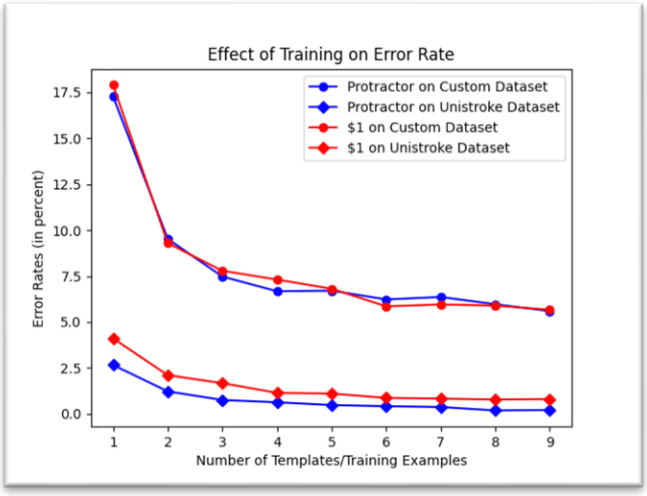


**Figure 12: Graphs showing The Error Rate as the number of Templates increase for both datasets.**

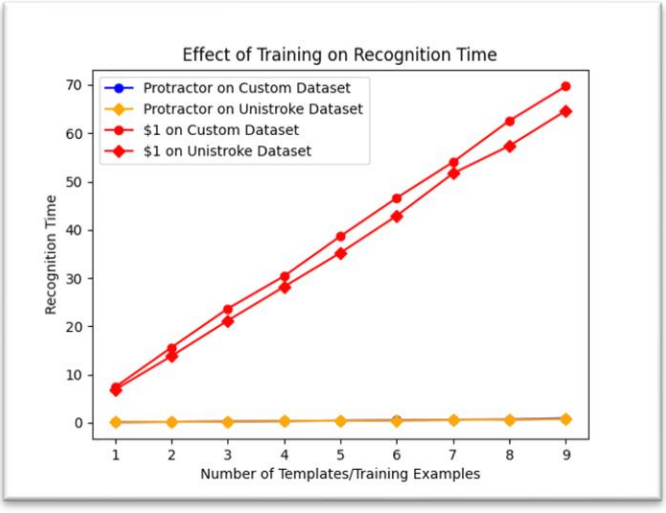


**Figure 13: Graphs showing Recognition Time (right) as the number of Templates increase for both datasets.**

## 6 Understanding Data

Unlike in project 1, where a heatmap was generated using GhoST [Appendix B, Figure B6], this time the heatmaps were generated by the program itself. This is to get much more detailed information.

Firstly, there is a major difference in the accuracies between the Unistroke Dataset and the Custom Dataset. If we see the figure 12, the error rate when the number of samples was 1 is 17.5% for protractor for custom dataset, but only around 4% for unistroke dataset. This is because the users who had drawn the gestures for unistroke had several specifications, but for custom dataset, no such things were done. From figure 14 below, the mismatches per user (extreme right column) are more in custom dataset when compared to the unistroke dataset. Other insights were already covered in the original report.

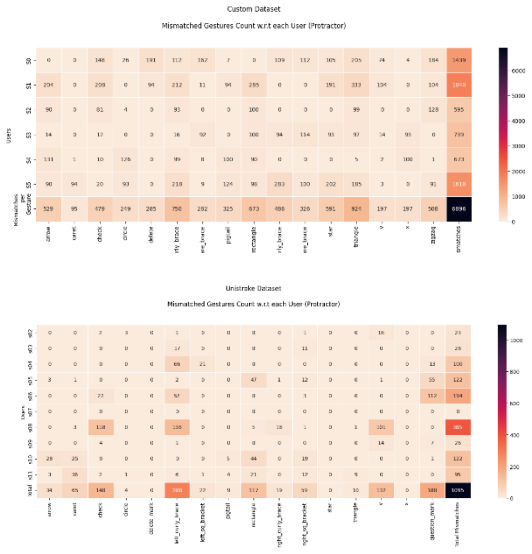


**Figure 14: Heatmaps showing mismatched gestures per user in Custom (top) and Unistroke (bottom) Datasets.**

## 7 Outcome and Conclusion

It is proved that the protractor algorithm even though gives accuracy like $1 algorithm, is much faster. When performing the 100-loop offline test, protractor was quick, compared to the slower $1 algorithm. The total accuracy and average recognition times are shown below:

| | Unistroke | Custom |
|---|---|---|
| $1 | 0.985 and 38.727ms | 0.919 and 35.759ms |
| Protractor | 0.992 and 0.432ms | 0.92 and 0.403ms |

The heatmaps in Figure 14 helped in understanding why there is a difference between accuracies of Unistroke and Custom Datasets. Although not included here the figure in appendix [A, B] also shows heatmaps and matrices on which gestures have higher mismatches. All these heatmaps were comparable to the heatmap generated by GHoST.

The main goal of this project was to provide a way for users/ students to understand how the process works. The visualization of the steps would certainly help them out and give a clear understanding of the algorithms. These two projects gave a clear view that Machine Learning is not always the solution, especially when we have less data. And it is also clear that we can always design algorithms that perform as good as ML algorithms for certain tasks.

## References

[1] Wobbrock, J.O., Wilson, A.D. and Li, Y. (2007). Gestures without * libraries, toolkits or training: A $1 recognizer for user in-

terface * prototypes. Proceedings of the ACM Symposium on User Interface * Software and Technology (UIST '07). Newport, Rhode Island (October * 7-10, 2007). New York: ACM Press, pp. 159-168. * https://dl.acm.org/citation.cfm?id=1294238

[2] Lisa Anthony, Radu-Daniel Vatavu, and Jacob O. Wobbrock. 2013. Understanding the consistency of users' pen and finger stroke gesture articulation. In Proceedings of Graphics Interface 2013 (GI '13). Canadian Information Processing Society, CAN, 87–94.

[3] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. 2014. Gesture Heatmaps: Understanding Gesture Performance with Colorful Visualizations. In Proceedings of the 16th International Conference on Multimodal Interaction (ICMI '14). Association for Computing Machinery, New York, NY, USA, 172–179. DOI: https://doi.org/10.1145/2663204.2663256

[4] Wobbrock, J. O., Wilson, A. D., & Li, Y. (n.d.). $1 recognizer. Retrieved from https://depts.washington.edu/acelab/proj/dollar/index.html

[5] Python. (n.d.). Tkinter - Python interface to TCL/TK¶. tkinter - Python interface to Tcl/Tk - Python 3.10.3 documentation. Retrieved from https://docs.python.org/3/library/tkinter.html

[6] Alex Clark and Contributors. (n.d.). Pillow. Retrieved from https://pillow.readthedocs.io/en/stable/

[7] Wobbrock, J. O., Wilson, A. D., & Li, Y. (n.d.). $1 recognizer. $1 recognizer. (n.d.). Retrieved from http://depts.washington.edu/acelab/proj/dollar/index.html

[8] Li, Y. (2010). Protractor: A fast and accurate gesture * recognizer. Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '10). Atlanta, Georgia (April 10-15, 2010). New York: ACM Press, pp. 2169-2172. https://dl.acm.org/citation.cfm?id=1753654

## APPENDIX A – UNISTROKE DATASET HEATMAPS


**Figure A1: Mismatches per gesture using $1 and protractor**

Figure A1 shows the number of mismatches that occurred per gesture when $1 recognizer and protractor algorithms were used. The extreme right column shows the total number of mismatches using these two algorithms.


**Figure A2: Mismatches in gestures per user using $1 recognizer**

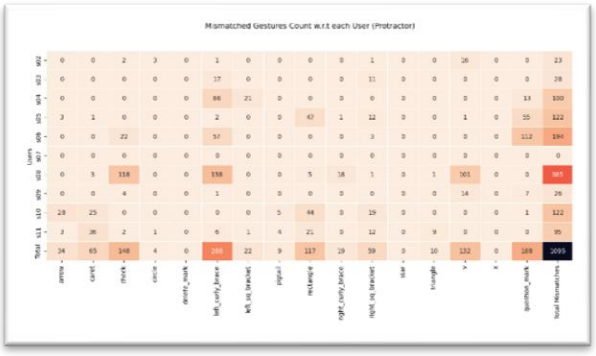


**Figure A3: Mismatches in gestures per user using Protractor**

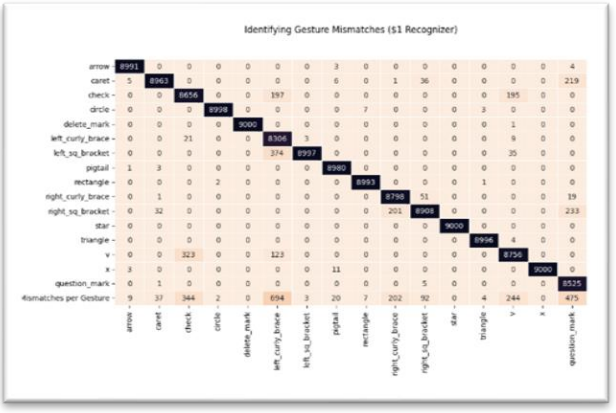


**Figure A4: Mismatches among gestures using $1 Recognizer**

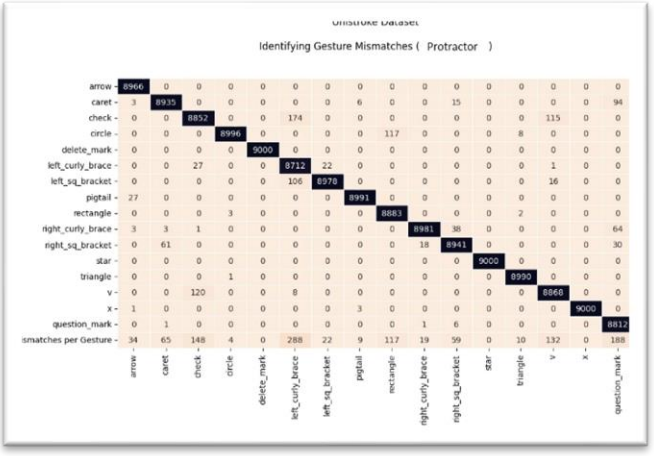


**Figure A5: Mismatches among gestures using Protractor**

## APPENDIX B – CUSTOM DATASET HEATMAPS

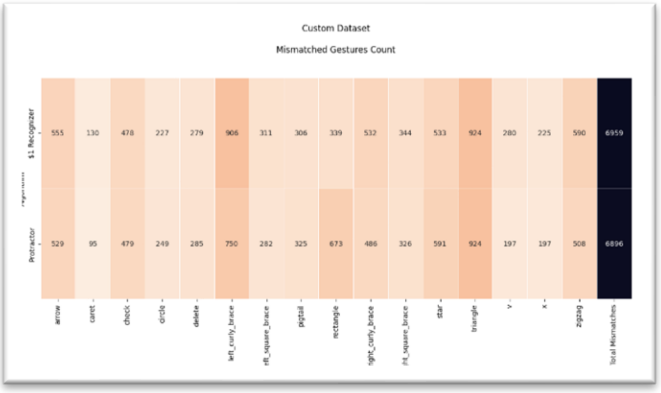


**Figure B1: Mismatches per gesture using $1 and protractor**

Figure B1 shows the number of mismatches that occurred per gesture when $1 recognizer and protractor algorithms were used. The extreme right column shows the total number of mismatches using these two algorithms.

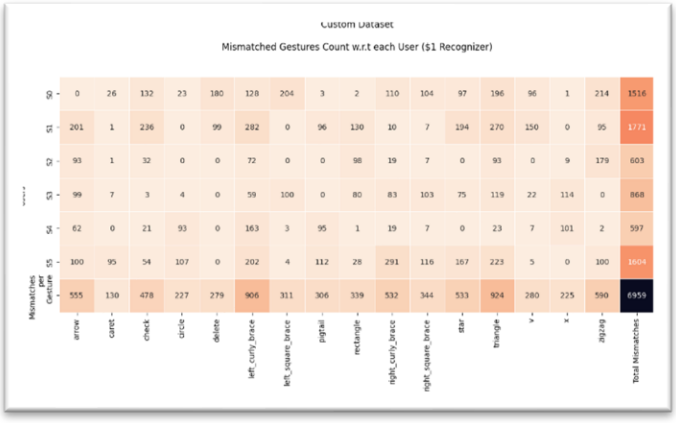


**Figure B2: Mismatches in gestures per user using $1 recognizer**

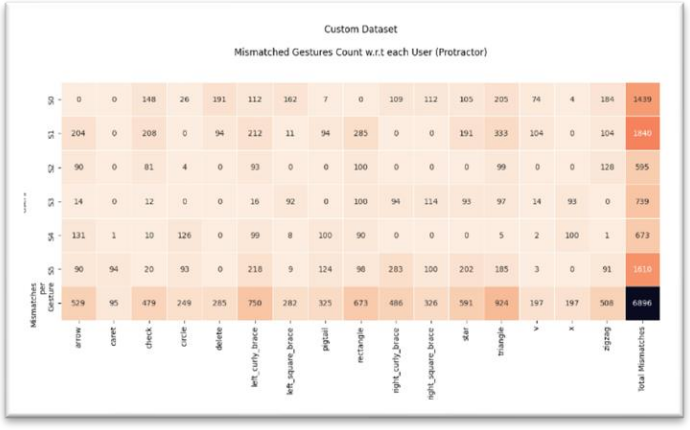


**Figure B3: Mismatches in gestures per user using protractor**

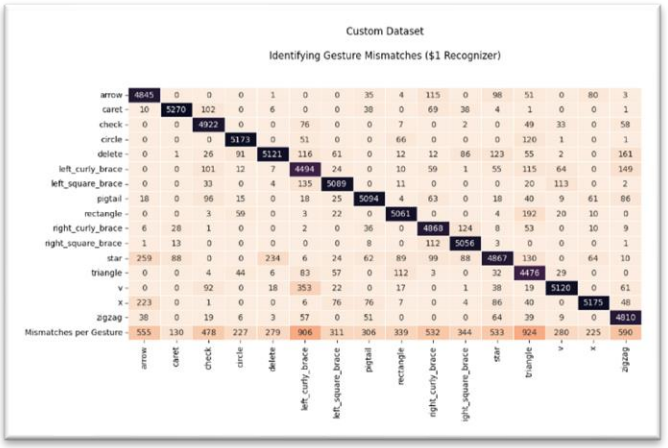


**Figure B4: Mismatches among gestures using $1 Recognizer**

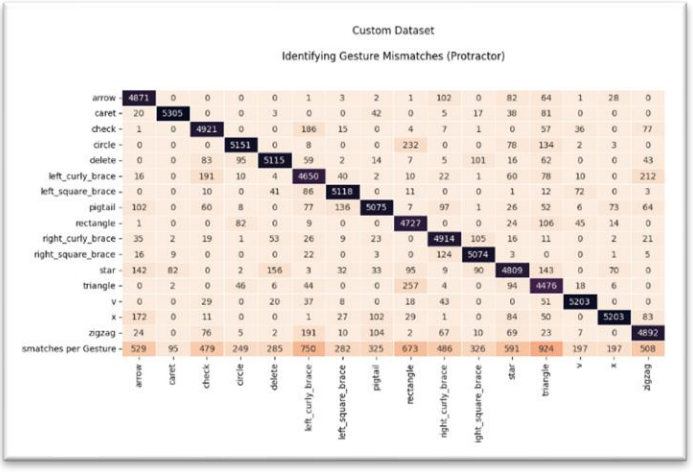


**Figure B5: Mismatches among gestures using Protractor**

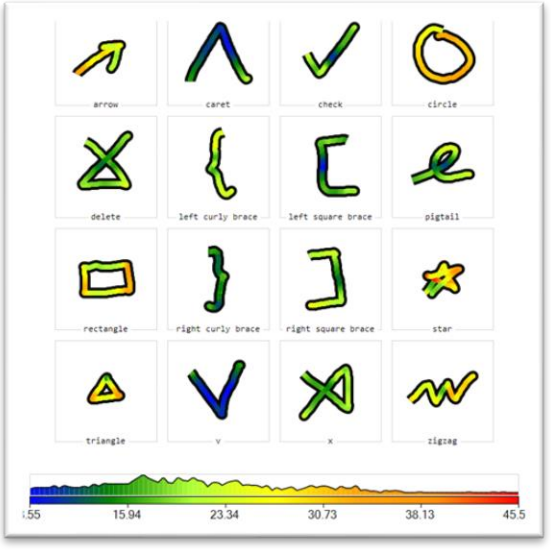


**Figure B6: Heatmap generated by GHoST**