An Evaluation of the $P Point-Cloud Gesture Recognizer

International Baccalaureate Diploma Program

Computer Science Extended Essay

Research Question: To What Extent is the $P Point-Cloud Recognizer Suitable to Recognize Various Touch Interface Gestures?

Word Count: 3959

Candidate Number: hhw881

Abstract

This paper explores how feasible it is to use the $P gesture recognition algorithm to classify a gesture in from a selected number of gesture sets with different characteristics. A gesture is a drawing made with one or multiple strokes on a touch user interface, and this paper specifically investigates quickly drawn and relatively simple gestures.

The specifications most developers look for while searching a gesture recognizer algorithm is explored in the paper, and the $P algorithm was chosen as the most promising algorithm to test in this paper. The $P gesture recognizer works by first converting the gestures needed into point clouds that have the same number of equally distanced points. Later the algorithm finds the closest matching gesture to the candidate gesture from a training set using a greedy point cloud matching algorithm. In the investigation the algorithm was tested with different datasets that have specific characteristics.

The data set includes almost 70 different gesture sets, with each gesture having 50 repetitions, divided into 8 different gesture set categories for testing specific cases. The gesture sets were tested with randomly chosen gestures among the given dataset, because testing every combination of gestures was unfeasible for the scope of this paper. The tests were done on accuracy and execution time in relation to the number of training data samples and resampling resolution of the $P algorithm, and the datasets are explored on the concepts of internal and in between variance.

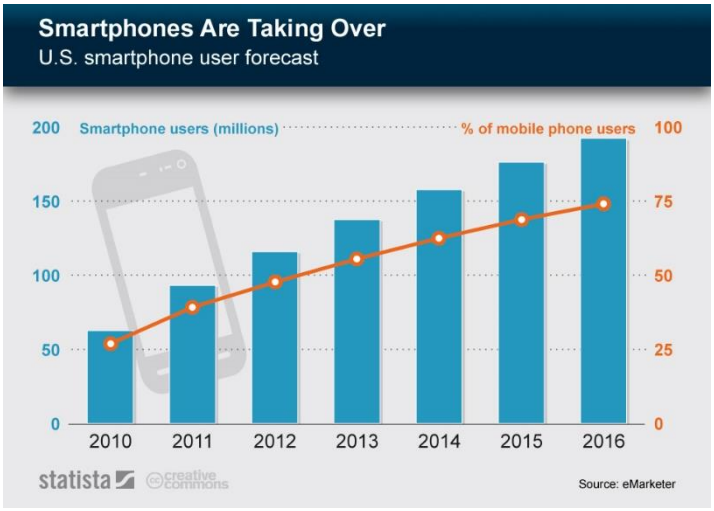
The accuracy was found to be linearly proportional to the logarithm of the number of training samples. No straightforward relationship was found between accuracy and resampling resolution, but the accuracy tended to increase up to a point of resampling resolution (32 for most data sets) and then tended to stay the same. The execution times were linearly proportional to the number of training samples and linearly proportional to the square of the resampling resolution. There were some remarks made about internal and in between variance, however the data was insufficient to make any big claims.

The algorithm was found to be useful for most use cases by the expected audience, however the training data, the training sample number, and the resampling resolution needs to be chosen carefully to achieve good results with more difficult datasets.

Abstract word count: 382

Contents

Introduction



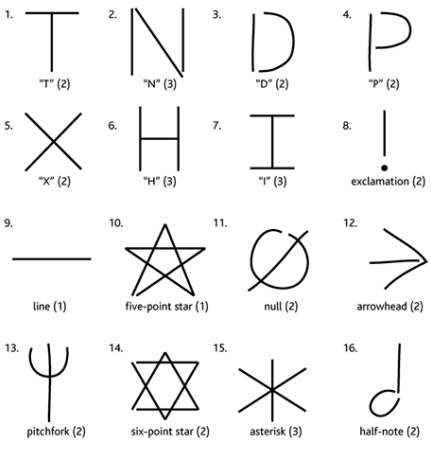
Picture 1: An informative char about smartphone use
From www.statista.com/chart/210/smartphones-are-taking-over/

After the smartphone revolution of the 21st century, most people now use a mobile device every day. Smartphones let the user make calls, access the internet, listen to music, play games, manage emails, and more. These services are provided by different applications in the smartphone. Most of the usual interactions with smartphones are done by simple hand gestures, such as tapping or dragging. Other advanced apps, like the maps app, utilize multi-touch gestures, such as pinching with two fingers to zoom. Some apps though, may need more varied inputs, without using buttons that occupy small screen space. For this reason and for more, a gesture input, a shape drawn on the screen by the user, can be utilized. Examples where this can be used are: a hand-writing input app, a gesture based way to navigate around the phone[1], a game where the user draws magic shapes, or an app where lots of actions are needed but there is limited screen space available.

For me personally, I stumbled upon this subject when I was trying to create a mobile game where the user would draw shapes to cast spells. I tried to quickly implement a basic algorithm by myself, however soon I realized how complicated gesture recognition actually is. After about a week of testing different ideas, my best algorithm was fiddly at best, only guessing half of the gestures correctly when the gestures were drawn with great care, and quite unreliable at worst, not even guessing 10% correctly when drawing as quickly as I wanted during normal gameplay. So to find a better algorithm, I did some research about the topic.

Challenges with Gesture Recognition

There can be many different types of gestures. Some are unistroke, meaning the shape includes only one continuous line, and some can be multistroke. In multistroke gestures, there are problems with which stroke is drawn first, or in which direction. Some shapes can be drawn in multiple ways, with any number of stroke counts, orders, and directions. Some shapes are harder to draw, and have a lot of variance in between the shapes. Some shapes don't have an exact method, and can be drawn in a multitude of ways. Some shapes can be drawn accurately and only have one way to



Picture 2: Different types of gestures
From faculty.washington.edu/wobbrock/pubs/icmi-12.pdf

[1] Mouad. "3 Of The Best Navigation Gesture Apps For Android - Make Tech Easier"

draw, but they may be needed to be differentiated from very similar shapes. Because of these reasons, shape recognition algorithms are needed to be versatile, and robust.

The algorithms that try to recognize user gestures on mobile also have another big problem: performance constraints. Although computers are faster than ever, mobile devices are still limited performance and storage wise. And for gesture recognition, fast and accurate classification is necessary, with optimal execution times being under 100 milliseconds. Because of the randomly drawn nature of gestures, it is difficult for traditional straightforward and fast algorithms to recognize gestures. For this reason, machine learning algorithms are often used for shape recognition. Machine learning algorithms work not by having the task programmed in the system, but by "learning" the task based on given training data. Machine learning based shape recognizers can be "fed" different shapes as training data and learn to recognize them. This is useful because it means one recognition system can virtually recognize any shape and can be improved further by more training data. However machine learning algorithms can be computationally expensive to execute, and sometimes require big storage spaces. A mobile gesture recognition algorithm needs to be fast to execute.

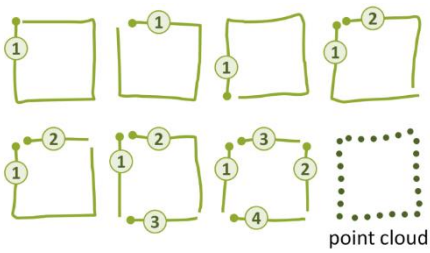
Another problem a mobile gesture recognition algorithm faces is the difficulty of implementation. Mobile is a platform that has a very low barrier of entry, as almost anyone can put an app on the various mobile application stores. Consequently, most mobile developers don't have the expertise and resources of big corporations. Companies like Google can implement complex neural network algorithms with gigabytes of training data for their gesture recognition software. Favourably, not every gesture recognition algorithm is as complex. Also some types of machine learning algorithms require training data in the range of gigabytes to terabytes to be viable. Gathering such data may be difficult or even impossible for small developers. An optimal algorithm should be easy to implement and require minimal data to train.

So a gesture recognizer for mobile developers needs to be able to recognize various different types of gestures with robustness. Because of the limitations of the platform, needs to be able to execute very quickly, ideally under 100 milliseconds. The algorithm also needs to be simple to code, and require minimal training data to work. The gesture recognizers need to fit all these criteria, while reaching high levels of accuracy, for example over 90% correct recognition rate. A research of the existing gesture recognizers resulted in multitude of algorithms, like the simple $1 unistroke recognizer[2], or the more complex Gestimator[3]. However, the $P Point-Cloud Recognizer[4] fit all the criteria, so it will be tested in this paper. The research question is: "*To what extent is the $P Point-Cloud Recognizer suitable to recognize various touch interface gestures?*"

[2] "$1 Unistroke Recognizer". Depts.Washington.Edu, 2019
[3] Ye, Yina, and Petteri Nurmi. "Gestimator".
[4] "$P Recognizer". Depts.Washington.Edu, 2019

$P Point-Cloud Recognizer

The $P Point-Cloud Recognizer is a 2D point cloud template based closest match finder. This means that the algorithm will try to find the user input gesture in a database of training samples, by looking at the distance between the input and the samples. The algorithm is developed by a group of researchers, and is "designed for rapid prototyping of gesture-based user interfaces."

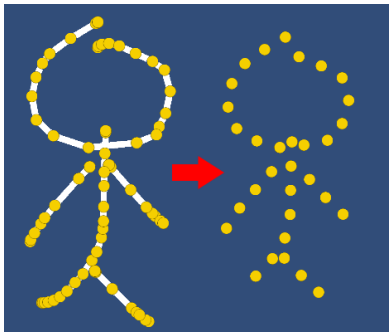


Picture 3: All the different ways a square can be drawn using strokes, and a point cloud that can represent all the variations
From faculty.washington.edu/wobbrock/pubs/icmi-12.pdf

The algorithm can recognize gestures that can are drawn in any number of strokes and in any stroke order, because the point cloud representation disregards all data about draw order and stroke groups.

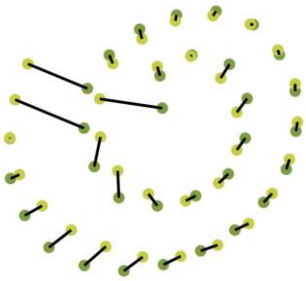
The paper named "Gestures as Point Clouds: A $P Recognizer for User Interface Prototypes"[5] about the $P Recognizer includes a much in depth explanation about how the design of the algorithm was created, and is outside the scope of this paper. However, the basic steps of the algorithm are as follows:

1. Normalize both the training data and the current input that will be tested.
    a. The training data can be pre-normalized before launch
    b. Normalization Steps:
        i. Normalize scale: Map all of the points' x and y values between 0 and 1
        ii. Normalize position: translate the center of the shape to the origin
        iii. Resample points: Resample the whole drawing to include only n amount (*the resampling resolution*) of equally spaced points.



Picture 4: A stickman shape, before and after normalization

2. Perform a greedy cloud match to find the template from the training samples with the minimum distance to the candidate
    a. To find the distance between two point clouds that are two arrays of points with the same length; start with an index i and find the closest unmatched point from the points2 array to the point1[i] point.
    b. Mark that point in the points2 array and move on, until all points in the arrays are matched.
    c. The distance between the point clouds is the sum of all the distance between the matched points, multiplied by a



Picture 5: A visualization of two matched point clouds
From faculty.washington.edu/wobbrock/pubs/icmi-12.pdf

[5] Wobbrock, Jacob O. et al. "Gestures As Point Clouds: A $P Recognizer For User Interface Prototypes".

confidence coefficient. (the later matched points matter less, as most of the other points were matched before, so their match is likely not optimal)

d. Also if at any point the sum of the distances surpasses the already found minimum distance value, quit calculations early.

e. Repeat the cloud distance match with different starting index points and find the minimum distance.

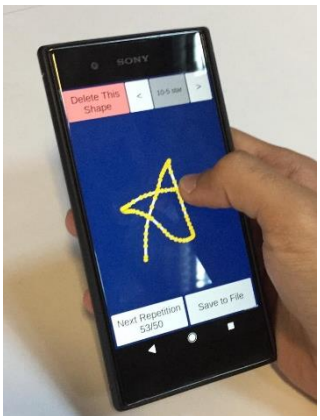
For a better understanding of the algorithm, a pseudo-code version and the Python implementation used in the testing is included in the appendix.

Investigation

As the $P algorithm needs to check the distance between the candidate and each one of the training samples multiple times, I suspected the algorithm execution times may rise rapidly to unfeasible levels with more training samples used. And as I checked the original research papers about the algorithm, I found graphs that would show the relation between training sample count and execution times curiously missing. Also because of the way the algorithm worked, it was inherently not very good at differentiating certain types of gestures. So by my research, I aimed to explore how those factored played for the feasibility of the algorithm.

The Method

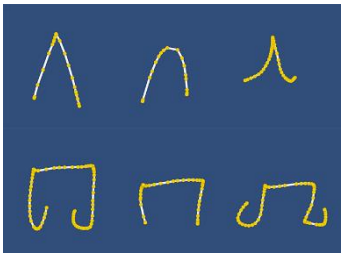
For testing the $P algorithm by various metrics, I designed a number of different gesture sets. Then I coded an android app with Unity to gather the gestures. I drew the gestures with my finger and tried to draw the gestures as quickly as possible, as this was my expected use case. For each gesture, I drew 50 copies, trying to include different draw orders and stroke counts when applicable. For the experiment, I decided to test 8 different gesture sets. You can check the appendix to find all the gestures in each set.



Picture 6: The way all the data was gathered

1. Basic Set – 16 different gestures
    - This was the set of gestures shown in the $P dollar's website. To check my own implementation's reliability, I decided to test the exact same gestures too.
    - This set includes shapes like "T", "N", "X", 5 point star, 6 point star, asterisk etc.
2. High Variability Set – 10 different gestures
    - I wanted to see how successful the algorithm was with shapes that had many different ways to draw, but also was quite distinct from each other.
    - This set includes shapes like fire, water, grass, stickman, car, plane etc.

3. Lines Set – 3 different gestures
    - I also wanted to see how the algorithm performed with shapes that had many possible ways to draw, but also quite similar to each other.
    - This set includes jaggy line, wavy line, and square line
4. Dash Set – 3 different gestures
    - To see how the algorithm performed when the shapes were very similar, both to each other and in-between sets, I included 3 quick lines, “dashes”, that followed the same standard as the lines set. This set is for a direct comparison to the lines set, to see the effect of variability.
    - This set includes 3 lines as well: jaggy line, wavy line, and square line
5. Parenthesis Set – 6 different gestures
    - To test the algorithm for less abstract and more widely used shapes I choose to include a set of parenthesises
    - This set includes “{“, ”}”, “[“, “]”, “(“, “)”.
6. Tick Set – 3 different gestures
    - I also wanted to test if the algorithm was able to differentiate between different number of strokes. As this algorithm forgoes strokes entirely I expect some problems with this.
    - This set includes “/”, “//”, “///“.
7. Abstract Shapes Set – 24 different gestures
    - Because the dash set wasn’t comprehensive enough to test low variability in shapes with high similarity between sets, I also drew some abstract shapes to test the algorithm.
    - This set includes shapes like pointy arrowhead, circular arrowhead, octopus like arrowhead etc.



Picture 7: Some of the “abstract shapes” tested

8. Star Set – 4 different gestures
    - I also wanted to see how the algorithm performed for very long and very complicated shapes, different pointed stars were a great fit.
    - This set includes 5,6,7, and 8 point stars.

For each set, I wanted to test two different independent variables and two different dependent variables. I ran the $P algorithm using Python, and collected data directly using the standard time library. I wrote the results into a text file and later used excel to process my results. Here is a breakdown of different variables:

Independent Variables:

The Number of Training Samples: The way the $P algorithm works is by comparing the candidate user drawn gesture with a set of training data and see which training data matches best with the candidate. So more training samples mean more variance in the data can be accounted for with the training data. More training data per gesture should result in better accuracy. However more training data also means the candidate gesture needs to be compared with more shapes, which should increase execution time. I kept the resampling resolution same (at 32) while testing for the number of training samples.

The Resampling Resolution: Because users draw gestures in an unpredictable manner, the raw data collected from the screen needs to be pre-processed. If the user draws faster or slower, the distance between the points change. Also the user input usually includes too many unnecessary points. Before classification, each gesture is resampled to have n number of equally distanced points, the n number being the resampling resolution. Increasing the resolution means there are more points representing the gesture, and the resolution may increase accuracy. However, more points per gesture also require more calculation per gesture, so higher execution time. I kept the number of training samples same (at 2) while testing for resampling resolution.

The Specific Training Samples Used: While doing the testing, I had to choose specific training samples to test my dataset against, and I needed to be careful as different training samples would affect my results. Because testing each different possible training data set would be impossible I choose a random set of the training samples. To even out the randomness, I ran each test multiple times (10-15 times) and averaged out the results. I also collected the minimum and maximum results to see how much the training sample choice would affect my results

Dependent Variables:

Accuracy: The accuracy is the number of times the algorithm classified the candidate shape correctly divided by the total number of trials. For different trials different random training samples were chosen, resulting in different accuracies. The final accuracy value is an average of the all accuracies. Also the maximum and minimum accuracies were collected, for the most successful training data set and the worst training set.

Execution Time: The execution time is the time the algorithm takes to classify one candidate. Just like the accuracy, the final value used will be an average of all the random trials. A maximum and minimum time

were also collected for the execution time because of the variance caused by the early exit nature of the program. When selected the training samples are good, the program may find a very close match quickly, drastically reducing the execution time for all the later cloud match checks.

Controlled Variables:

The Computer: To have consistent executions times all the tests were run on my own computer, and each of the computer's 4 cores of the CPU was used to minimize the run time. However still all the testing took more than 6 hours to complete. To make sure the results weren't affected by other tasks on the CPU, the computer was left idle during the testing, with all the applications closed, and the Wi-Fi disconnected. The computer specifications are written in the appendix.

Similarity Score:

Also apart from the accuracy/execution time testing a similarity score for each gesture set was calculated. I wanted to put a number on the vague term "similarity" and for this I calculated an internal variance score and an in between variance score for each of the gesture sets.

Internal Variance Score: Internal variance score stands for how much variation between each shape of a certain gesture set. For example, two minus sign drawings are usually much similar to each other than two 5 star drawings, so the minus sign set would be expected to have a lower internal variance score than 5 star drawing set does. To find the "distance" between each shape, I used the already existing $P's greedy cloud match. Ideally I would find the distance between each shape in set, however each set has 50 elements, so I would need 50! distance calculations, which is impossibly big. So instead I calculated the distance between two random different elements a lot of times to get an average value.

In Between Variance Score: Similarly to internal variance score, in between variance score stands for how distinct are the gesture sets in a testing set. For example, two gesture sets "A" and "K" would score a higher in between variance score than "L" and "I". As each testing set has many gesture sets, (for example, the basic set has 16 gesture sets) and each gesture set has 50 gesture repetitions, finding a real in between variance score would take too much computational time. So like the internal variance score, I calculated a lot of distances between two random elements from two random gesture sets to have a close to real value.

The Results

Raw Data

After finding the results the Python program writes the results into a CSV file which is then put into an excel file for processing. Here is an example classification test result:

Test Name	Training Sample Count	Resampling Resolution	Test Trial Count	Gesture Copies Count	Total Trial count	One Trial Count
Main Set - TraSamp 1-8	1	32	10	30	4640	464
Main Set - TraSamp 1-8	2	32	10	30	4480	448
…	…					
Main Set - TraSamp 1-8	8	32	10	30	3520	352

Test Name	Total Match Count	Max Match Count	Min Match Count	Total Check Time (s)	Total Exec Time (s)	Min Check Time (s)	Max Check Time (s)
Main Set - TraSamp 1-8	4479	459	432	392.926	392.968	0.023	0.438
Main Set - TraSamp 1-8	4387	441	434	714.398	714.434	0.041	0.398
…	…						
Main Set - TraSamp 1-8	3489	351	346	1500.105	1500.131	0.118	0.851

Table 1: Some of the raw data from the classifications tests

As all of the data was collected via a modern computer, the uncertainties in the time data were negligible.

```
#Check for each gesture in our check set
for gesId in range(len(checkSet)):
    start = time.time () #start timing
    #find the classification name
    matchName = TesterMethod(checkSet[gesId],trainingSet)
    end = time.time () #stop timing
    checkTime = end-start;

    totalCheckTime += checkTime
    minCheckTime = min(minCheckTime, checkTime)
    maxCheckTime = max(maxCheckTime, checkTime)

    if( matchName == checkSet[gesId].Name): #if the match was correct
        curMatchCount += 1;
```

Code 1: The relevant code section that was used to gather the data in the *table 1*

In Between Variance	Sampling Res	Trial Count	Gesture Rep Count	Total Trial Count	Total Distance	Max Distance
Lines Set	32	30	30	900	158.788	0.391
…	…					
Tick Set	32	30	30	900	612.527	1.956

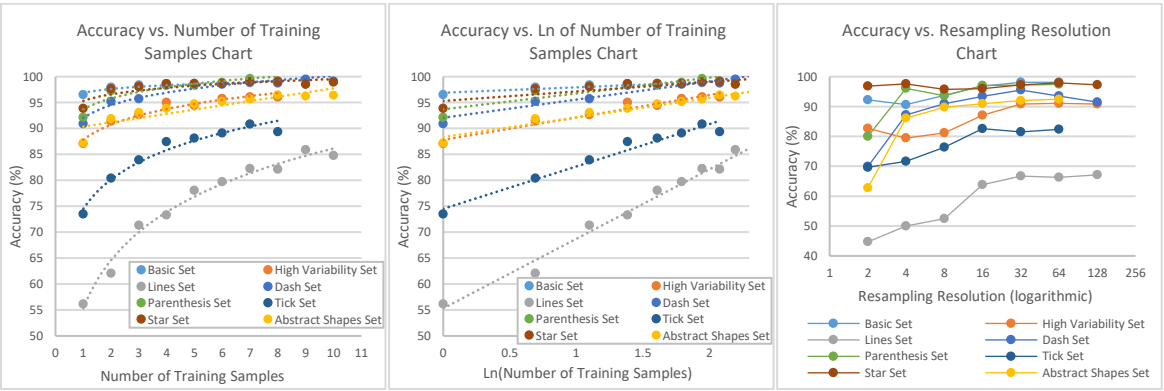
Table 2: Some of the raw data from the in between variance tests

A complete version of the code used to collect the data can be found in the appendix.

Analysis

To reach any conclusions, the average accuracy and execution times of various tests were calculated and using excel.

The first area that will be analysed is accuracy:



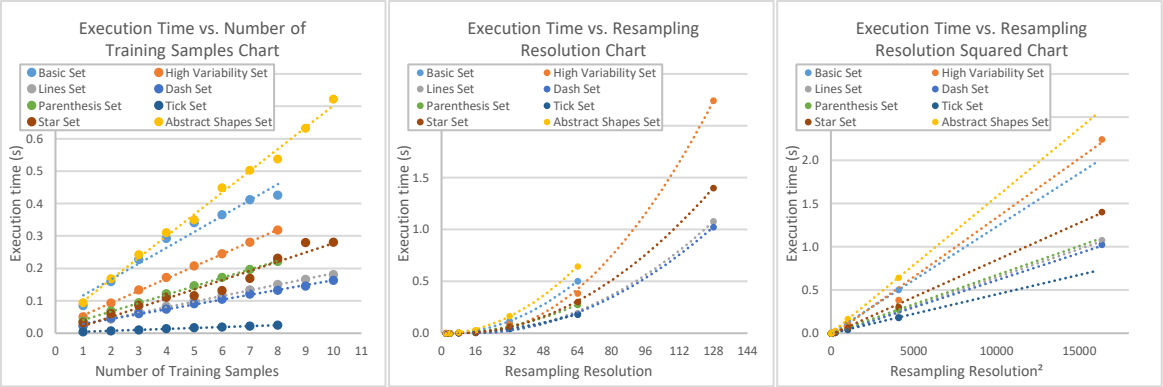
Graphs 1, 2, 3: Graphs showing the accuracy data

As expected, the number of training samples affects the accuracy, for all the gesture sets. The accuracy was linearly proportional to the logarithm of number of training samples. Check *table 3* for specific slope values.

The effect of resampling resolution wasn't as clear as the effect of training samples. Generally, as the resampling increased, so did the accuracy, until the resolution was 32, after which the increased resolution didn't affect the results. However, for some of the sets, especially in sets where the gesture start locations where mostly the same, the resolution didn't matter a lot. Especially in the star set, where I drew each shape starting from the same location, instead of keeping it varied.

The reason why the lines set had especially low accuracy is probably because of how the lines set included very similar shapes that were drawn in a multitude of ways. Please refer to *picture 8* below:

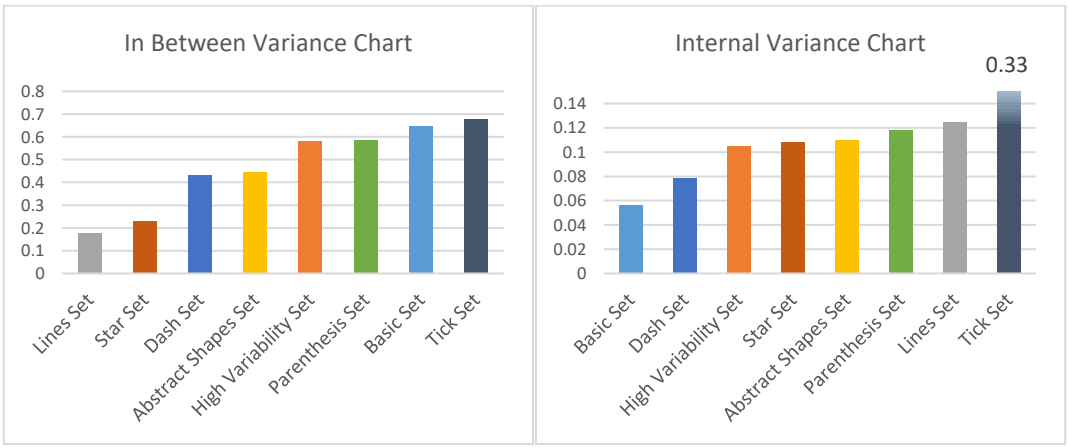
Picture 8: Variance between members of jaggy line, wavy line, and square line, all from the lines set



Graphs 4, 5, 6: Graphs showing the execution time data

The connection between the execution time and both the number of training samples and resampling resolution is very clear. As the samples or the resolution increased, the execution time increased linearly or by the polynomial, as there were that many more shapes or points to calculate. The execution times were linearly proportional to the number of training samples and linearly proportional to the square of the resampling resolution. Check *table 3* for specific slope values.

The data on the internal and in between variance wasn't very clearly related to accuracy or execution times, however some comments can still be made.



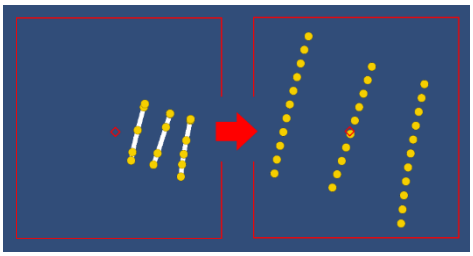
Graphs 7, 8: Graphs showing the in between and internal variance data

In general, higher in between variance and smaller internal variance meant an increase in accuracy, meaning more distinct shapes are easier to classify. The star set clearly deviates from this claim, as it has very low in between variance, but a high accuracy.

As expected, the basic set scores high on in between variance and low on internal variance. This shows itself clearly in the accuracy graph as the basic set has high accuracies overall.

The lines set has the least in between variance and a high internal variance, which resulted in the lowest accuracy of all, and the biggest benefit from an increased number of training samples.

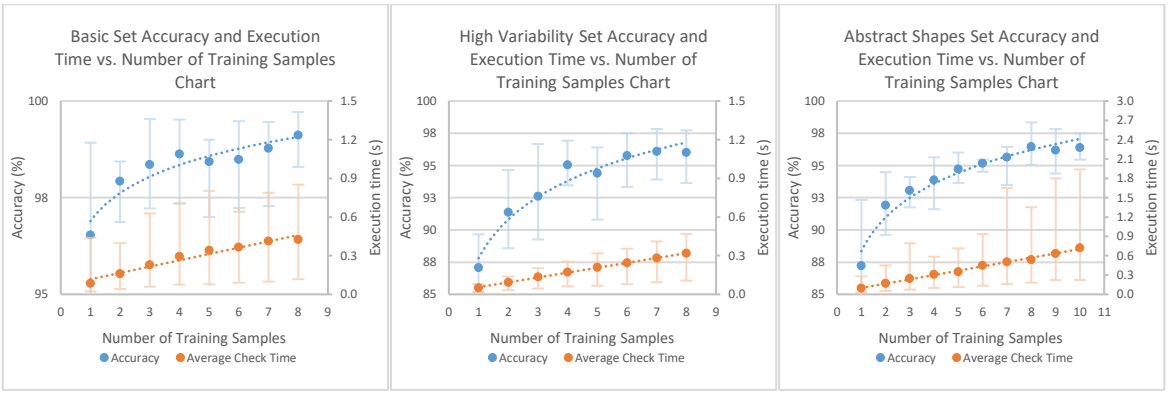
The tick set was an outlier in internal variance. After investigations, the reason was I concluded was that although I drew the ticks very small, close, and similarly, after the software normalized the gesture small variances became very big (see *picture 9*).



Picture 9: The three ticks shape, before and after normalization

A high internal variance score is found to be loosely correlated with a low execution time. This is most likely because of the way early exits worked in the algorithm. Because the code stop executing once the distance value gets higher than the minimum found distance, a higher internal variance meant that this minimum distance was reached more quickly, reducing the execution times.

I also looked into the effects of variance on min/max accuracy and execution times. And their connection to the variance scores



Graphs 9, 10 ,11: Accuracy and Time vs. Number of training samples graphs for various data sets. The uncertainties are from the collected minimum and maximum data values for each measurement.

High variability set and lines set both have high internal variance scores, and their uncertainty in execution time is low, as the code can exit quickly. However, for the other two, both the uncertainty and average execution times are higher.

Also, high interval variance loosely correlates with the uncertainty in accuracy, as the difference between each sample in a set increase, the effect of the specific training samples chosen increase.

	Internal Variance		In Between Variance		Training Samples vs		Resampling Rate vs
Set Name	**Max**	**Average**	**Max**	**Average**	**Accuracy Slope**	**Time Slope**	**Time Slope * 1000**
Basic Set	0.30	0.06	2.03	0.65	1.048	0.049	0.124
High Variability Set	0.33	0.11	1.34	0.32	4.323	0.038	0.137
Lines Set	0.29	0.12	0.39	0.18	13.392	0.017	0.066
Dash Set	0.26	0.08	1.30	0.43	3.494	0.015	0.063
Parenthesis Set	0.96	0.12	1.12	0.59	3.056	0.026	0.067
Tick Set	0.99	0.33	1.96	0.68	8.161	0.003	0.045
Abstract Shapes Set	0.63	0.11	1.59	0.45	1.829	0.028	0.086
Star Set	0.30	0.11	0.40	0.23	3.779	0.067	0.158

Table 3: All of the final calculated data

Conclusion

For most developers in need of a simple mobile gesture recognition algorithm, the $P seems to be sufficient. For the more advanced needs though, some internal testing with the specific data is necessary to find the optimal number of training data and resampling resolution.

Although the $P algorithm worked nicely for most of the use case data sets, the more difficult data sets that are very similar in nature and that can be drawn in a multitude of ways makes the algorithm require much more training data to reach acceptable levels of accuracy. However, this increase in training data also result in increased processing time and may result in the recognition being too slow, although this was not the case for the datasets used in this investigation.

Even the most difficult data sets used in this paper all managed to get >85% accuracy with less than 200 milliseconds of execution time. Most other data sets managed >90-95% accuracy with less than 100 milliseconds of execution time.

So, in conclusion, it is reasonable to answer the question "*To what extent is the $P Point-Cloud Recognizer suitable to recognize various touch interface gestures?"* with a yes, but also with a however, as the harder datasets can require lots of testing and training data to be feasible.

Limitations

There were numerous limitations as a result of the scope of this paper, limiting a more thorough and insightful conclusion. The limitations are caused by insufficient computing power, lack of a bigger data set, and more algorithms to test.

The more of a surface limitation of this paper was that because of huge computation times for checking each combination, there was a need to limit the checks by choosing random gestures from a bigger list, and doing multiple trials to at least get an average result. A better computer, the technical skill to carry out the computations on the GPU instead of CPU would hugely increase the trial numbers, giving more accurate results.

The more important limitation of this paper was a lack of more gesture sets. As I needed to manually gather multiple tens of repetitions for each gesture set, I couldn't include more than a set number of gestures. More types of gestures, testing for a bigger range of internal and in between variance would give much deeper insight into how these two scores affect accuracy and execution time. Also, all the data was gathered from one participant, which is not a realistic scenario and reduces the variance in data greatly. Inclusion of multiple participants is needed for a better user experience understanding, as gestures from different people would be harder to recognize than a single person.

This paper also tested one algorithm's strength and weaknesses only. Although I tried to choose the most logical and popular algorithm and tested specifically to try to exploit this algorithm, there are many other gesture recognition software that can be very powerful in areas that the $P algorithm fails. However, a study with a much bigger scope is needed to properly compare and contrast a multitude of algorithms in even the variables used in this paper. The inherent high variance nature of the subject makes this kind of research difficult.

References

"$1 Unistroke Recognizer". *Depts.Washington.Edu*, 2019, https://depts.washington.edu/aimgroup/proj/dollar/.

"$P Recognizer". *Depts.Washington.Edu*, 2019, https://depts.washington.edu/madlab/proj/dollar/pdollar.html.

Mouad. "3 Of The Best Navigation Gesture Apps For Android - Make Tech Easier". *Make Tech Easier*, 2019, https://www.maketecheasier.com/best-navigation-gesture-app-android/.

Richter, Felix. "Infographic: Smartphones Are Taking Over". Statista Infographics, 2019, https://www.statista.com/chart/210/smartphones-are-taking-over/.

Wobbrock, Jacob O. et al. "Gestures As Point Clouds: A $P Recognizer For User Interface Prototypes". Faculty.Washington.Edu, 2019, https://faculty.washington.edu/wobbrock/pubs/icmi-12.pdf.

Ye, Yina, and Petteri Nurmi. "Gestimator". *Proceedings Of The 2015 ACM On International Conference On Multimodal Interaction - ICMI '15*, 2015. *ACM Press*, doi:10.1145/2818346.2820734. Accessed 26 Feb 2019.

Appendix

Relevant Computer Specifications:

Processor: Intel Core i5-6300HQ CPU @2.30GHz

RAM: 8,00 GB

Operating System: 64-bit Windows 10

Language used: Python 3

Pseudo Code of the $P Algorithm:

$P Point-Cloud Gesture Recognizer Pseudocode

Radu-Daniel Vatavu
University Stefan cel Mare of Suceava
Suceava 720229, Romania
vatavu@eed.usv.ro

Lisa Anthony
UMBC Information Systems
1000 Hilltop Circle
Baltimore MD 21250
lanthony@umbc.edu

Jacob O. Wobbrock
Information School | DUB Group
University of Washington
Seattle, WA 98195-2840 USA
wobbrock@uw.edu

In the following pseudocode, POINT is a structure that exposes x, y, and $strokeId$ properties. $strokeId$ is the stroke index a point belongs to (1, 2, ...) and is filled by counting pen down/up events. POINTS is a list of points and TEMPLATES a list of POINTS with gesture class data.

Recognizer main function. Match *points* against a set of *templates* by employing the Nearest-Neighbor classification rule. Returns a normalized score in [0..1] with 1 denoting perfect match.

```
$P-RECOGNIZER (POINTS points, TEMPLATES templates)
 1: n ← 32 // number of points
 2: NORMALIZE(points, n)
 3: score ← ∞
 4: for each template in templates do
 5:     NORMALIZE(template, n) // should be pre-processed
 6:     d ← GREEDY-CLOUD-MATCH(points, template, n)
 7:     if score > d then
 8:         score ← d
 9:         result ← template
10: score ← MAX((2.0 − score)/2.0, 0.0) // normalize score in [0..1]
11: return ⟨result, score⟩
```

Cloud matching function. Match two clouds (*points* and *template*) by performing repeated alignments between their points (each new alignment starts with a different starting point index i). Parameter $\epsilon \in [0..1]$ controls the number of tested alignments ($n^{\epsilon} \in \{1, 2, ...n\}$). Returns the minimum alignment cost.

```
GREEDY-CLOUD-MATCH (POINTS points, POINTS template, int n)
1: ε ← .50
2: step ← ⌊n^(1−ε)⌋
3: min ← ∞
4: for i = 0 to n step step do
5:     d1 ← CLOUD-DISTANCE(points, template, n, i)
6:     d2 ← CLOUD-DISTANCE(template, points, n, i)
7:     min ← MIN(min, d1, d2)
8: return min
```

Distance between two clouds. Compute the minimum-cost alignment between *points* and *tmpl* starting with point *start*. Assign decreasing confidence *weights* ∈ [0..1] to point matchings.

```
CLOUD-DISTANCE (POINTS points, POINTS tmpl, int n, int start)
 1: matched ← new bool[n]
 2: sum ← 0
 3: i ← start // start matching with points_i
 4: do
 5:     min ← ∞
 6:     for each j such that not matched[j] do
 7:         d ← EUCLIDEAN-DISTANCE(points_i, tmpl_j)
 8:         if d < min then
 9:             min ← d
10:             index ← j
11:     matched[index] ← true
12:     weight ← 1 − ((i − start + n) MOD n)/n
13:     sum ← sum + weight · min
14:     i ← (i + 1) MOD n
15: until i == start // all points are processed
16: return sum
```

The following pseudocode addresses gesture preprocessing (or normalization) which includes resampling, scaling with shape preservation, and translation to origin. The code is similar to $1 and $N recognizers[1,2] and we repeat it here for completeness. We highlight minor changes.

Gesture normalization. Gesture points are resampled, scaled with shape preservation, and translated to origin.

```
NORMALIZE (POINTS points, int n)
1: points ← RESAMPLE(points, n)
2: SCALE(points)
3: TRANSLATE-TO-ORIGIN(points, n)
```

Points resampling. Resample a *points* path into n evenly spaced points. We use $n = 32$.

```
RESAMPLE (POINTS points, int n)
 1: I ← PATH-LENGTH(points) / (n − 1)
 2: D ← 0
 3: newPoints ← points_0
 4: for each p_i in points such that i ≥ 1 do
 5:     if p_i.strokeId == p_(i−1).strokeId then
 6:         d ← EUCLIDEAN-DISTANCE(p_(i−1), p_i)
 7:         if (D + d) ≥ I then
 8:             q.x ← p_(i−1).x + ((I − D)/d) · (p_i.x - p_(i−1).x)
 9:             q.y ← p_(i−1).y + ((I − D)/d) · (p_i.y - p_(i−1).y)
10:             q.strokeId ← p_i.strokeId
11:             APPEND(newPoints, q)
12:             INSERT(points, i, q) // q will be the next p_i
13:             D ← 0
14:         else D ← D + d
15: return newPoints
```

```
PATH-LENGTH (POINTS points)
1: d ← 0
2: for each p_i in points such that i ≥ 1 do
3:     if p_i.strokeId == p_(i−1).strokeId then
4:         d ← d + EUCLIDEAN-DISTANCE(p_(i−1), p_i)
5: return d
```

Points rescale. Rescale *points* with shape preservation so that the resulting bounding box will be $\subseteq [0..1] \times [0..1]$.

```
SCALE (POINTS points)
 1: x_min ← ∞, x_max ← 0, y_min ← ∞, y_max ← 0
 2: for each p in points do
 3:     x_min ← MIN(x_min, p.x)
 4:     y_min ← MIN(y_min, p.y)
 5:     x_max ← MAX(x_max, p.x)
 6:     y_max ← MAX(y_max, p.y)
 7: scale ← MAX(x_max − x_min, y_max − y_min)
 8: for each p in points do
 9:     p ← ((p.x − x_min)/scale, (p.y − y_min)/scale, p.strokeId)
```

Points translate. Translate *points* to the origin $(0, 0)$.

```
TRANSLATE-TO-ORIGIN (POINTS points, int n)
1: c ← (0, 0) // will contain centroid
2: for each p in points do
3:     c ← (c.x + p.x, c.y + p.y)
4: c ← (c.x/n, c.y/n)
5: for each p in points do
6:     p ← (p.x - c.x, p.y - c.y, p.strokeId)
```

[1] http://depts.washington.edu/aimgroup/proj/dollar/index.html
[2] http://depts.washington.edu/aimgroup/proj/dollar/ndollar.html

This pseudocode is modified slightly from that which appears in the original ACM ICMI 2012 publication by Vatavu, Anthony, and Wobbrock (http://dx.doi.org/10.1145/2388676.2388732) to better highlight the use of the *strokeId* property and to provide a normalized matching score. It also contains more comments to assist the implementation. This algorithm's logic remains unchanged.

Shapes Tested, divided by the group:

Basic Set:

High Variability Set:

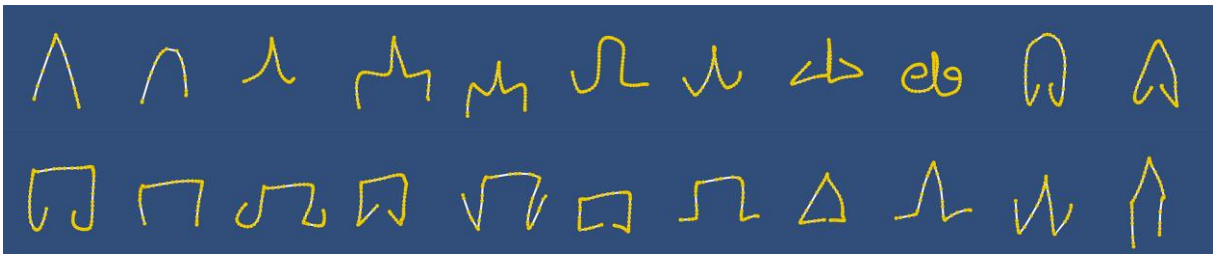
Lines Set:

Dash Set:

Parenthesis Set:

Tick Set:

Abstract Shapes Set:



Star Set:

A Section from the All Gesture Sets xml file (The file that stores all the gestures I gathered):

```
<Save_x0020_Data xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <gestureSetCount>92</gestureSetCount>
  <GestureSets>
    <GestureSet GroupName="0-X" RepetitionCount="50">
      <repetitions>
        <Gesture GestureName="0-X" RepetitionIndex="0" StrokeCount="2">
          <strokes>
            <Stroke Index="0" PointCount="15">
              <points>
                <Point x="1.08288574" y="1.97790527" t="6.58464575" />
                <Point x="-0.780822754" y="-0.5250244" t="6.60231924" />
                <Point x="-0.780822754" y="-0.5250244" t="6.60580826" />
                <Point x="-0.95199585" y="-0.813476563" t="6.6131" />
                <Point x="-0.95199585" y="-0.813476563" t="6.62371826" />
                <Point x="-1.08346558" y="-1.0748291" t="6.63145447" />
                <Point x="-1.23782349" y="-1.3972168" t="6.64916134" />
                <Point x="-1.32049561" y="-1.61535645" t="6.66662359" />
                <Point x="-1.35046387" y="-1.75634766" t="6.68423271" />
                <Point x="-1.43289185" y="-1.99536133" t="6.69966459" />
                <Point x="-1.52731323" y="-2.24169922" t="6.71621132" />
                <Point x="-1.57263184" y="-2.36206055" t="6.7329917" />
                <Point x="-1.6088562" y="-2.41992188" t="6.749442" />
                <Point x="-1.61520386" y="-2.42675781" t="6.766033" />
                <Point x="-1.61526489" y="-2.42675781" t="6.80112362" />
              </points>
            </Stroke>
            <Stroke Index="1" PointCount="24">
              <points>
                <Point x="-1.38104248" y="1.6796875" t="7.05828667" />
                <Point x="-1.38104248" y="1.6796875" t="7.065092" />
                <Point x="-1.38104248" y="1.6796875" t="7.085833" />
                <Point x="-1.38104248" y="1.6796875" t="7.099013" />
                <Point x="-1.38104248" y="1.6796875" t="7.11602259" />
                <Point x="-1.38104248" y="1.6796875" t="7.13401747" />
                <Point x="-1.36907959" y="1.65075684" t="7.14865875" />
                <Point x="-1.1171875" y="1.22900391" t="7.165985" />
                <Point x="-0.67099" y="0.5534668" t="7.18279362" />
                <Point x="-0.458099365" y="0.28137207" t="7.20065641" />
                <Point x="-0.189544678" y="-0.0322265625" t="7.216377" />
                <Point x="0.0686035156" y="-0.3491211" t="7.23269558" />
                <Point x="0.219329834" y="-0.5649414" t="7.24778748" />
                <Point x="0.315002441" y="-0.7529297" t="7.265894" />
                <Point x="0.513458252" y="-1.17333984" t="7.282333" />
                <Point x="0.591430664" y="-1.39489746" t="7.29811525" />
                <Point x="0.613311768" y="-1.49926758" t="7.31456566" />
                <Point x="0.707550049" y="-1.75305176" t="7.33175373" />
                <Point x="0.7836914" y="-1.954834" t="7.348508" />
                <Point x="0.8614197" y="-2.13586426" t="7.366568" />
                <Point x="0.9220581" y="-2.27709961" t="7.382053" />
                <Point x="0.9633484" y="-2.35131836" t="7.399236" />
                <Point x="0.9883728" y="-2.38793945" t="7.41512632" />
                <Point x="1.00979614" y="-2.40759277" t="7.43586349" />
              </points>
            </Stroke>
          </strokes>
        </Gesture>
      ...
```

The original file has more than 263.000 lines, and is over 17mb.

My own Python implementation of the algorithm, based on the C# sample code provided on the $P website (only relevant sections included):

The Test Runner Script's Relevant Sections:

```
#main test method
def DoTesting(dataSet, traSampleCount, samplingRes, TesterMethod, gestureRepCount = 30, trials = 10):
    totalTrialCount, totalMatchCount, oneTrialCount, totalCheckTime, totalExecTime = 0;
    maxMatchCount, maxCheckTime = 0;

    minMatchCount = len(dataSet)*gestureRepCount;
    minCheckTime = 10;

    #Setting up
    allGestures = []
    # import all gestures into an Array of Arrays
    for gesSet in dataSet:
        allGestures.append ([])
        for n in range (gestureRepCount):
            allGestures[len (allGestures) - 1].append (Gesture (GetPoints (gesSet[0][n]),
                  gesSet.attrib["GroupName"], samplingRes))

    randIndex = []
    for n in range (gestureRepCount): randIndex.append (n);

    trainingSet = []
    checkSet = []

    for gesSetId in range (len (allGestures)):
        for repId in range (gestureRepCount):
            if repId < traSampleCount:
                trainingSet.append (None)
            else:
                checkSet.append (None)

    #Testing
    for trial in range(trials):
        triStart = time.time ()
        print("trial: " + str(trial))

        traCounter = 0
        checkCounter = 0
        for gesSetId in range(len(allGestures)):
            # random list to choose the training samples randomly
            random.shuffle (randIndex);
            for repId in range(gestureRepCount):
                if repId < traSampleCount:
                    trainingSet[traCounter] = (allGestures[gesSetId][randIndex[repId]])
                    traCounter += 1
                else:
                    checkSet[checkCounter] = (allGestures[gesSetId][randIndex[repId]])
                    checkCounter += 1

        oneTrialCount = len(checkSet)
        curMatchCount = 0;
        for gesId in range(len(checkSet)):
            if(gesId % min( int( (len(checkSet) / 5) + 1), 50) == 0):
                print("tra sampl #:" + str(traSampleCount) + " - re sampl #:" + str(samplingRes)
          + " - trial id:" + str(trial) + " - gesture id: " + str(gesId) + "/" + str(len(checkSet)))
            start = time.time ()
            matchName = TesterMethod(checkSet[gesId],trainingSet)
            end = time.time ()
            #if (gesId % 50 == 0): print("Average time:" + str(end-start))
            checkTime = end-start;
            totalCheckTime += checkTime
            minCheckTime = min(minCheckTime, checkTime)
            maxCheckTime = max(maxCheckTime, checkTime)

            if( matchName == checkSet[gesId].Name):
                curMatchCount += 1;

        totalTrialCount += oneTrialCount;
        totalMatchCount += curMatchCount;
        minMatchCount = min(minMatchCount,curMatchCount);
        maxMatchCount = max(maxMatchCount,curMatchCount);
        triEnd = time.time ()
        trialTime = triEnd-triStart
        totalExecTime += trialTime
    #put everythin in a findings array here - I removed it to fit this into one page
    return findings;

#Do this for each of the sets we want to test
allResults = []
for n in range(1, 9):
    results = DoTesting(AllGesturesData[1][1:16+1], n, 32, PointCloudRecognizer.GreedyClassify, 30, 10);
    results["testName"] = "Main Set - TraSamp 1-8"
    allResults.append(results)

WriteToFile(allResults)
```

The Similarity Finder Script (Calculates the internal and in between variance scores):

The beginnings of these methods are the same as the test runner script's DoTesting method.

Internal Variance Finder:

```
for trial in range(trials):
    print("trial: " + str(trial))

    for gesSetId in range(len(allGestures)):
        # random list to choose the training samples randomly
        random.shuffle (randIndex);
        for repId in range(gestureRepCount):
            if repId < gestureRepCount/2:
                set1[repId] = (allGestures[gesSetId][randIndex[repId]])
            else:
                set2[repId-int(gestureRepCount/2)] = (allGestures[gesSetId][randIndex[repId]])

    for gesId in range (len (set1)):
        distance = DistanceMethod(set1[gesId].Points, set2[gesId].Points,100)
        #print(matchName + " - " + checkSet[gesId].Name)
        totalTrialCount += 1
        totalDistance += distance
        maxDistance = max(maxDistance, distance)
```

In Between Variance Finder:

```
for trial in range(trials):
    print("trial: " + str(trial))

    for gesSetId in range(len(allGestures)):
        # random list to choose the training samples randomly
        random.shuffle (randIndex);
        for repId in range(gestureRepCount):
            set1[repId] = (allGestures[randIndexGesSet[repId %
len(allGestures)]][randIndex[repId]])
            set2[repId] = (allGestures[randIndexGesSet[(repId+1) %
len(allGestures)]][randIndex[repId]])

    for gesId in range (len (set1)):
        distance = DistanceMethod(set1[gesId].Points, set2[gesId].Points,100)
        #print(matchName + " - " + checkSet[gesId].Name)
        totalTrialCount += 1
        totalDistance += distance
        maxDistance = max(maxDistance, distance)
```

The Point Cloud Recognizer:

```
def GreedyClassify(self, candidate,  trainingSet):
    minDistance = float("inf")
    gestureClass = ""
    for template in trainingSet:
        dist = self.GreedyCloudMatch(candidate.Points, template.Points, minDistance)
        if dist < minDistance:
            minDistance = dist
            gestureClass = template.Name
    return gestureClass


def GreedyCloudMatch(self, points1, points2, minFoundDist):
    n = len(points1) #  the two clouds should have the same number of points by now
    eps = 0.5;       # controls the number of greedy search trials (eps is in [0..1])
    step = int(math.floor(pow(n, 1.0 - eps)))
    minDistance = float("inf")
    for i in range (0, n, step):
        dist1 = self.CloudDistance(points1, points2, i, minFoundDist);    # match points1 -->
points2 starting with index point i
        dist2 = self.CloudDistance(points2, points1, i, minFoundDist);    # match points2 -->
points1 starting with index point i
        minDistance = min(minDistance, min(dist1, dist2))
    return minDistance


def CloudDistance(self, points1, points2, startIndex, minFoundDist):
    n = len(points1);       # the two clouds should have the same number of points by now
    matched = [] #  matched[i] signals whether point i from the 2nd cloud has been already
matched
    for i in range(n): matched.append(False),

    sum = 0;  # computes the sum of distances between matched points (i.e., the distance
between the two clouds)
    i = startIndex + 1
    isFirst = True
    while i != startIndex:
        if sum > minFoundDist:
            break;

        if isFirst:
            isFirst = False
            i = startIndex;

        index = -1
        minDistance = float("inf")
        for j in range(n):
            if not matched[j]:
                dist = Geometry.SqrEuclideanDistance(points1[i], points2[j]);  # use squared
Euclidean distance to save some processing time
                if dist < minDistance:
                    minDistance = dist
                    index = j
        matched[index] = True #  point index from the 2nd cloud is matched to point i from the
1st cloud
        weight = 1.0 - ((i - startIndex + n) % n) / (1.0 * n)
        sum += weight * minDistance #  weight each distance with a confidence coefficient that
decreases from 1 to 0
        i = (i + 1) % n
    return sum
```

The Gesture Class:

```
def __init__(self, points, gestureName = "", samplingRes = 32):
    self.Name = gestureName

    self.SAMPLING_RESOLUTION = int(samplingRes)

    #  normalizes the array of points with respect to scale, origin, and number of points
    self.Points = self.Scale(points)
    self.Points = self.TranslateTo(self.Points, self.Centroid(self.Points))
    self.Points = self.Resample(self.Points, self.SAMPLING_RESOLUTION)

def Scale(self, points):
    minx = float("inf")
    miny = float("inf")
    maxx = float("-inf")
    maxy = float("-inf")
    for i in range(len(points)):
        if minx > points[i].X: minx = points[i].X;
        if miny > points[i].Y: miny = points[i].Y;
        if maxx < points[i].X: maxx = points[i].X;
        if maxy < points[i].Y: maxy = points[i].Y;

    newPoints = []
    scale = max(maxx - minx, maxy - miny)
    for i in range(len(points)):
        newPoints.append(Point((points[i].X - minx) / scale, (points[i].Y - miny) / scale,
points[i].StrokeID))
    return newPoints

def TranslateTo(self, points, p):
    newPoints = []
    for i in range(len(points)):
        newPoints.append(Point(points[i].X - p.X, points[i].Y - p.Y, points[i].StrokeID))
    return newPoints

def Centroid(self, points):
    cx = 0
    cy = 0
    for i in range(len(points)):
        cx += points[i].X
        cy += points[i].Y
    return Point(cx / len(points), cy / len(points), 0)

def Resample(self, points, n):
    newPoints = []
    newPoints.append(Point(points[0].X, points[0].Y, points[0].StrokeID))
    numPoints = 1

    I = self.PathLength(points) / (n - 1) #  computes interval length
    D = 0
    for  i in range(1,len(points)):
        if points[i].StrokeID == points[i - 1].StrokeID:
            d = Geometry.EuclideanDistance(points[i - 1], points[i])
            if D + d >= I:
                firstPoint = points[i - 1]
                while D + d >= I:
                    #  add interpolated point
                    t = min(max((I - D) / d, 0.0), 1.0);
                    if t is None: t = 0.5;
                    numPoints += 1;
                    newPoints.append(Point(
                        (1.0 - t) * firstPoint.X + t * points[i].X,
                        (1.0 - t) * firstPoint.Y + t * points[i].Y,
                        points[i].StrokeID
                    ))

                    #  update partial length
                    d = D + d - I
                    D = 0
                    firstPoint = newPoints[numPoints - 1]
                D = d
            else: D += d

    if numPoints == n - 1: # sometimes we fall a rounding-error short of adding the last point, so add it
if s:
        newPoints.append(Point(points[len(points) - 1].X, points[len(points) - 1].Y, points[len(points) -
1].StrokeID))
    return newPoints
```

The Point Class:

```
class Point:
    X = 0
    Y = 0
    StrokeID = 0

    def __init__(self, x, y, strokeId):
        self.X = float(x)
        self.Y = float(y)
        self.StrokeID = int(strokeId)
```

The Geometry Class:

```
def  SqrEuclideanDistance (self, a, b):
    return (a.X - b.X) * (a.X - b.X) + (a.Y - b.Y) * (a.Y - b.Y)


def EuclideanDistance(self, a, b):
    return math.sqrt(self.SqrEuclideanDistance(a, b))
```