UCLouvain

École polytechnique de Louvain

Benchmarking of Stroke-gesture Recognition Algorithms in Multiple Contexts of Use

Model, Method and Software Tool

Author: **Alexandre Rucquoy**
Supervisor: **Jean Vanderdonckt**
Readers: **Nathan Magrofuoco, Sébastien Combéfis**
Academic year 2019–2020
Master [120] in Computer Science

Abstract

In our modern societies, human computer interactions became ubiquitous. A critical area of research in this subject is the ability, for an artificial intelligence, to recognize gestures performed by humans. There are many different approaches, each approach also having many different solutions and algorithms. The main objective of this work is to offer a tool that is able to benchmark different algorithms that perform 2D stroke gesture recognition, a particular sub-type of gestures, mostly using pattern matching techniques. Our solution is a simple web application, focused on ease of maintenance and extension. We hope that it will be of some use for researchers who want to perform benchmarks of said algorithms, or that it will serve as baseline for further works in the domain of gesture recognition. Since some parts of our application are related to work that is yet to be released, the code of our application is not, as of June 2020, publicly available online. You can, however, ask permission from either Prof. Vanderdonckt or Mr. Magrofuoco to have access to the sources.

Acknowledgements

I would like to thank the members of the jury: Professor Vanderdonckt, Mr Magrofuoco and Professor Combéfis for the precious time they dedicated to answer my questions and to help me. I would also like to thank all professors and teaching assistants from UCLouvain who are passionate about what they teach and who are dedicated to transmit their passion to students : you really make the difference.

Contents

Chapter 1

Introduction

1.1 Context of The Problem

With the advent of smart devices, human computer interaction [1] (*HCI*) became ubiquitous: it is now part of our everyday lives. In the process of interaction between human beings and smart devices, gestures appeared as a dominant interface. Gesture recognition is thus a crucial issue in *HCI*. With the rise of artificial intelligence, many different techniques and algorithms emerged with the following common objective : interpret human gestures. Such techniques and algorithms are simply called (and will be referred in this writing as) gesture recognizers, or simply, *recognizers*. Now, the problem is, we have to determine which one of them is best suited in some specific context of use. Unfortunately, there is (currently) no simple answer to this question because it depends on many different criteria. The performance of a recognizer may vary more or less strongly depending on the type of gesture it has to interpret, their respective number of occurrences, the context of use, the platform, the quality of the gesture itself and many more... It is even more difficult to compare them since their ability to correctly recognize gestures (their *recognition rate*) is not a sufficient criterion : imagine two recognizers A and B, tested in the same conditions, the same context, with exactly the same input. Both will much likely show different recognition rate (for example, A has a rate of, say, 90 percent and B 95 percent). Then our choice become simple : pick B and discard A. But should we still favor B if it is much slower than A? This is why, a second important criterion in our evaluation process is the *recognition time*. All these recognizers were of course tested, but not necessarily under the same conditions and experimental setup than their peers. The challenge we are tackling is thus, how do we compare them? There are many relevant features and properties that could be used to characterize and compare gesture recognizers. We already mentioned the recognition rate and time. These are, however, not the only interesting measures of performance we could think about : time and space complexity, for example, are both interesting properties of recognizers. Or, we could also consider the memorability of gestures, since there seems to be a relation between the capacity of a human being to produce and remember gestures more or less accurately depending on the origin of the gesture [12]. In our work, we will focus on recognition rate and time because they are both interesting empirical measures of performance.

[1] Definition: https://www.interaction-design.org/literature/topics/human-computer-interaction

1.2 Mission Statement

1.2.1 Objective

This thesis pursues two objectives. The main objective is to provide a way to compare different gesture recognition algorithms in different conditions, in different contexts. Our suggested solution to reach this objective is a web application that supports all our targeted algorithms and that should help its users to perform benchmarks. The second objective is that our application should allow on reproducibility and replicability [2] of experiments made on the algorithms.

The core work of this thesis is thus to offer a single tool, a single platform that will help us to better evaluate all these recognizers. The tool should help its users to answer the following questions:

- For a given gesture set, which recognizer is the fastest ? (The lower *recognition time* is, the better)
- For a given gesture set, which recognizer is the most accurate ? (The higher *recognition rate* is, the better)
- For a given set of gesture classes, which recognizer is the fastest / the most accurate? What is the impact on both criteria if we add / remove one or multiple classes?
- What is the impact on performances if we increase/decrease the number of *sampling points* per gesture?
- How does the performance of recognizers changes when the context changes?

1.2.2 Work Hypothesis

- We suppose that there is no recognizer who always performs better than all the others (in terms of both recognition rate and recognition time) in all possible contexts. It means that the contexts, once implemented in the application, should highlight that, among all supported recognizers in this application, none can be considered as "superior" to all the others.
- Given the nature of most of the recognizers we'll consider in our tool, it is much likely that:
  - increasing the number of training samples per gesture class used to train recognizers,
  - increasing the number of sampling points per gesture

  will have a positive impact on the recognition rate but also a negative impact on the recognition time. In other words, there should be an inversely proportional relation between the recognition rate and the recognition time and the application should be able to show it.
- Performance will vary depending on the platform the gesture set was collected from.

[2] ACM artifact review: https://www.acm.org/publications/policies/artifact-review-badging

- Performance will vary depending on the degree of similarity between gesture samples from the same classes.

1.2.3 Research Method

This section will be a brief discussion of the research methodology of this thesis. Each part of the methodology will be presented here as an overview. Further details will be discussed in the next chapters of this document.

State of the art. The first thing we have to do is to establish a state of the art in the field of gesture recognition. This phase will help us to select our algorithms, our candidates, for the next steps.

Defining contexts of use. It is also important to define what are the contexts in which our recognizers are going to be evaluated. A context is, in our case, always characterized by three different things :

- A participant : a human being whose gesture samples were collected.
- A platform : the device which the data was collected on.
- An environment : the testing procedure and device used when running the actual benchmark.

Any change in one of those three elements induce inevitably a change in the context. It is therefore crucial for us to consider the following aspects regarding the features our application should have.

Defining a benchmark methodology. We will use here a well-known benchmark methodology [5] defined by Robert Camp. Explanations and definitions of each of the twelve steps of this benchmark methodology will be provided in the next chapter of the thesis. This methodology is not meant to be strictly followed, however it will serve as "guidelines" for the application development. In other words, the app should be designed in a way that makes it easy for its users to run their tests if they wish to follow this twelve steps methodology.

Building a tool to run the tests. Now, we're ready to build the actual tool: it should be able to bring answers, or at least tracks to answers for our work hypothesis and maybe, open new research questions. The tool will be a web application with a simple client-server architecture. It will be developed as a full stack JavaScript application, using the *M.E.R.N* stack.

1.3 Overview of the Thesis

This section will contain some small paragraphs, each one of them containing informations about the other chapters of the thesis. If you want to spare some time or just have a quick look at this document rather than an in-depth reading, I suggest you read this overview beforehand.

Chapter 2, entitled "A brief history of gesture recognition". This chapter will contain an introduction to gesture recognition. It will also contain an overview of prior works relevant to stroke gesture recognition, giving background knowledge about our topic. We will talk about different approaches for gesture recognition and give informations about those who are interesting for us. Then, we will go through different algorithms, explain the basics of their behaviour, what are their purpose, advantages and constraints.

Chapter 3, entitled "Gester, a software to benchmark stroke gesture recognizers". This chapter will be dedicated to our created benchmark tool, the "software". It will contain general explanations about its features and what purpose each feature serves. It will also contain details about its architecture and its components. Topics like development process and development tools will be discussed here. The provided description about the software in this chapter will rather be structural and conceptual than purely technical. The objective here is to show how it is built and justify (if relevant) the implementation choices made on the way. A more technical description will be provided in another chapter, dedicated to the documentation.

Chapter 4, entitled "benchmarking of gesture recognizers". We will introduce a famous benchmark methodology [5] and then suggest how it can be applied in our field of interest. We will also provide examples of tests performed using our tool to prove that it is usable and useful to gather results and informations about recognizers.

Chapter 5, entitled "documentation and deeper insights of Gester" will be a chapter containing two different kind of documentations. First, a user manual including screenshots of the *GUI* alongside with instructions to use our tool. The second documentation will be more technical. There will be informations about how the whole benchmark process has been implemented, how are the algorithms and scenarios integrated. These informations will serve to ease potential future extensions of the application by other developers. There will also be a discussion about the tests provided with our application.

Chapter 6, the conclusion. It will be a summary of all the work that has been done in this thesis. We will also identify limits of our application, future works and possible tracks for further improvements. We will also try to provide some suggestions for the identified limits.

Chapter 2

A Brief History of Gesture Recognition

2.1 Overview of This Chapter

Before we dive into technical details and specificities of considered algorithms, it is useful to introduce gesture recognition as a discipline, what does it consist of, what are the different approaches when we talk about the process of "recognizing a gesture". In this thesis we are, in fact, only covering a small portion of this discipline. We restrain ourselves to stroke gestures, which are a particular sub-type of gestures. Furthermore, we only consider 2D contact-based stroke gestures, which is yet another limitation in our study field. We will start by giving broad definitions of "gesture" and "gesture recognition" and state what are the different existing approaches for gesture recognition. The area of "gesture recognition" as a whole is, honestly, way too large to be exhaustively detailed in a single master thesis. Therefore, we will only focus on an overview of different approaches, different algorithms, to guarantee that our readers will have enough background knowledge about the subject to understand the purpose of our work.

2.2 Gesture and Gesture Recognition

"Gesture is a label for actions that have the features of manifest deliberate expressiveness" [8]. A gesture is thus, simply, a non-verbal way to communicate with your surroundings. But what characteristics really define what a gesture is ? If the objective is to communicate, to share some informations, you better ensure that there is a well defined protocol between the one who emits the gesture and the one who receives it. Therefore, we suggest the following aspects to characterize a gesture :

- The "semantic" of a gesture (what) : its meaning in the real world, the perception we have of the gesture. There must be a way, not specifically unique, to interpret gestures, and consequently, to differentiate them. We can see the semantic as a mapping between the concrete representation of a gesture (the emitter waves his/her hand) and our perception of it (the emitter says hello). There is, unfortunately, no universal standard regarding gestures semantic : a sign of the hand could be considered very friendly in some culture, but the same gesture could be interpreted as a provocation in others. In this thesis, we will refer to the semantic of a gesture as its *class*.

- The "origin" of the gesture (where) : what support (physical device, body part...) did the emitter used to perform the gesture ? In this thesis, we will focus on contact-based 2D stroke gestures. Contact-based means that there is a surface which serves as recipient/interface for human gestures: this could be, for example, the screen of your smartphone.
- The "emitter" of the gesture (who) : what is the source performing the gesture ? Depending on what it is, it can have varying "shape" but the same semantic. In our context of 2D stroke gestures, if you ask different people to write down the same character, it is much likely that both of them will write it differently.

Based on those criteria, we can deduce that the critical stakes when you wish to recognize a gesture are : first, how do you represent the gesture and second, how do you interpret it. These are actually the two phases most automated *recognizers* go through when running. We will now define the representation that is interesting to us : stroke gestures.

A "stroke" gesture ? To define what a stroke gesture is, we will refer to gesture's taxonomy given by Aigner et al. in [1]. Stroke gestures are a sub-type of semaphoric gestures, which are "*hand postures and movements, used to convey specific meanings*". What sets a stroke gesture apart from others, is that it is trajectory sensitive. A stroke is nothing but a series of time-ordered points. These points, linked to each other with a predecessor/successor relation defines a path : our stroke. A sequence of ordered path(s) (or stroke(s)) defines a gesture.

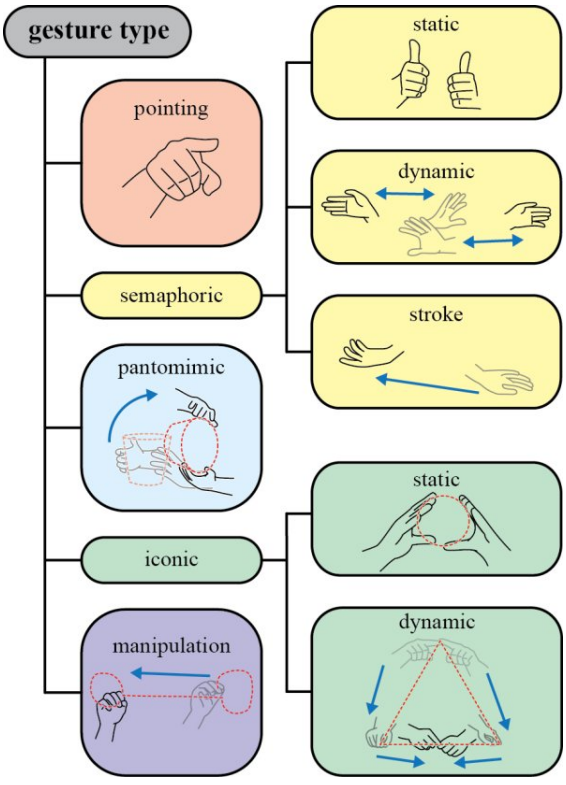


Figure 2.1: Aigner et al. gesture's taxonomy, from [1]

"Gesture recognition pertains to recognizing meaningful expressions of motion by a human, involving the hands, arms, face, head and/or body...It is the process by which the gestures made by the user are recognized by the receiver [10]". In our case, it means that recognizing a gesture, will consist of giving it a *class*, a semantic, a signification. Concretely, a human being provide the gesture (the "meaningful expressions of motion by a human"), which is then handled by an algorithm, a recognizer (the "receiver") which is going to interpret it. This interpretation could be based, for example, on its prior knowledge about gestures and their semantic.

2.3 Approaches

2.3.1 Hidden Markov Model

A Markov model is a way to represent a system who is evolving randomly and whose next state only depend on the current state. Which means, every actions that occurred before the most recent past action must not have any impact on the probabilistic distribution of the next action : this is the Markov property. A hidden Markov Model, or colloquially speaking, a *HMM*, is a Markov model (a finite set of states and transition with a probability attached to each transition) where each state also has an associated random function. When, at time t, the modeled system is in some state $s_t \epsilon S$, what we will perceive is not the actual state s_t but the output of the random function associated to s_t : $f(s_t)$. The state of the HMM appears thus as "hidden" because it can't be possible to retrieve which sequence of states we went through by observing the sequence of outputs from the random functions $f(s_0), ..., f(s_t)$ associated to each state (of course, each random function associated to each state in the Markov model has the same set of possible output than all the others). As example of application of HMM in gesture recognition, we can cite Yamato et al. paper, where they used HMM's to represent 6 different classes of gesture performed by tennis player and built a recognizer able to differentiate instances of those 6 classes with a recognition rate of +/- 90 % in a user-dependent context [22]. More generally, HMM are very convenient when your objective is to retrieve a hidden sequence of data based on a set of observable features, which make them suitable to classify/recognize gestures.

2.3.2 Finite State Machines

It is also possible to define a gesture with a finite state machine, where each state represent a *sampling point* of the gesture and transitions are triggered upon presence of specific characteristics from the input gesture, for example, the spatio-temporal position of the sampling point. To classify a gesture, we just need to see its ordered sequence of states and we can consequently deduce its class. Usually, the training of the recognizer is done using as many samples as possible of each gesture class as training data, and the parameters (criteria or characteristics) of each state of the *FSM* are then derived from the train samples. Testing is done by extracting feature vectors from your test data (those vectors should be consistent with those used in the training phase to derive the FSM) and then feeding these feature vectors to the machine. Using these vectors, the machine should decide if, yes or no, it fires a state transition. For more information and a concrete example of usage of FSM in gesture recognition, we suggest the reader to have a look at *Gesture Modeling and Recognition Using Finite State Machines* from Hong et al. where they built a recognizer who plays the game "Simon says" with human users [7]. They used

sequences of 2D coordinates of human heads and hands as feature vectors for their gesture representation. Each state of their machine is then derived, using the centroïd of these coordinates. Each state represent thus a region, a surface, of the 2D space.

2.3.3 Other Traditional Machine Learning Approaches

Since gesture recognition is nothing but a classification problem, we can, in theory, apply any method, any algorithm who tries to classify data samples from the machine learning literature. We are not going through an extensive list of all used techniques to solve gesture recognition. But, for your information, be aware that studies and researches about machine learning techniques of all kind applied to gesture recognition have been made. As an example, we can cite Patsadu et al., who published results about experiments of gesture classification over different gesture classes (stand, sit down, lie down) collected with a "Kinect" camera [13]. After pre-processing of the samples of all three classes, they trained and tested different classical machine learning models : Bayesian classifier, decision tree, support vector machine...

2.3.4 Soft Computing

Due to their adaptive nature, artificial neural networks (*ANN*) are a possible approach for gesture recognition. There is a variety of different connectivists techniques used to solve gesture recognition problems. To illustrate this approach, we can cite, for example, Time delay neural network, which are used in hand writing recognition [**?**], among other applications. This approach tends to produce very robust models, able to handle real life like situations, since they are tolerant to some flaws like uncertainty and imprecision. Such flaws are very frequent when we rely on sensors of any kind to gather our input data (noise produced by devices capturing human gestures...).

2.3.5 Nearest Neighbor

(k-)nearest neighbor(s) classification (*k-NN*) is a very common approach for pattern recognition. Training of such algorithms is made by extracting feature vectors from our input data and storing instances of these vectors with a class name attached to each one of them. For testing, we feed our model a new feature vector. It is then classified by computing a (dis)similarity coefficient between feature vectors. The attributed class to the input feature vector is the class of the previously known closest feature vector. The algorithm we will consider in our application are mostly working like k-NN classifiers.

How do they work in our case ? To apply these methods in the field of stroke gesture recognition, we simply need to remember what are our gestures made of, what kind of data do we give to our classifiers. The short answer is : gesture samples. A sample is nothing but a series of strokes. As explained earlier in this chapter, each stroke is characterized by an ordered set of points, what we call our *sampling points*. All the sampling points characterize the path of the stroke. Our classifiers, when submitted to a new gesture sample, will compare it to the templates, the gesture samples they have in their memory. The classification is then made by comparison of the input stroke gesture with all the templates of all classes our models have been trained on. Nearest neighbor classifiers, in our case, go through the following steps of execution, as shown in figure 2.2 :

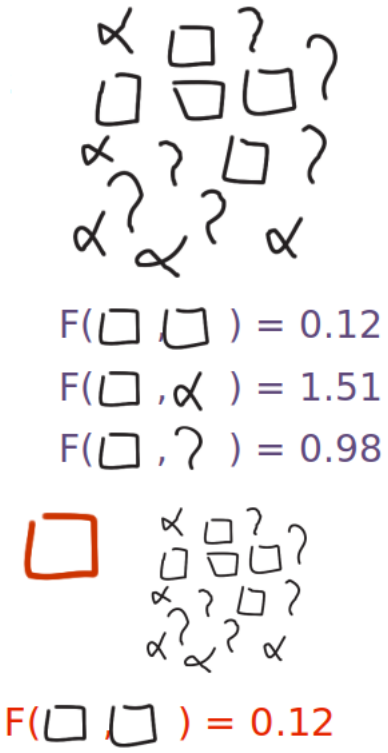


Figure 2.2: Example of a nearest neighbour classification of a gesture, here with a dissimilarity score

- Defining a set of templates, a set of different gestures from the different targeted classes. This will be our training data.
- Defining a (dis)similarity function between a template gesture and a candidate gesture.
- Report the k gesture(s) with the smallest (resp. biggest) dissimilarity (resp. similarity) score.

2.4 Considered Algorithms in Our Application

Why do we mainly consider k-NN here? In this thesis, our main interest is on algorithms that are simple and concise. Their implementation should be easy to understand by other developers. They should also be easy to integrate into our benchmark tool, since one of the main focus and development constraint is that the application should be easy to extend. This ease of integration is directly proportional to the inherent complexity of considered algorithms. Another advantage of such algorithms is that they are easy to parametrize since they require no tweaking. They also have a very small set of considered features when they classify data : that makes them less exposed to problems like over fitting. Finally, we can also mention the fact that such algorithms offer the possibility of a simple, human understandable geometric interpretation. It is indeed possible to use geometric features of the input data: the sequence of strokes, length and angle of path, are all important features we can consider when we wish to classify gestures. Thanks to this simple and straightforward geometric representation of gestures, such algorithms offer an

easy interpretation, manageable and understandable by humans [16]. Last but not least, we could also mention the fact that adding new samples of known gesture classes or even adding a new gesture class can be done without having to re-build and train the whole model again. This is very convenient for systems using user-defined gestures. All these advantages make k-NN algorithms suitable for stroke gesture recognition : the results provided by tests made with such algorithms could reflect real-life situations. Since, here, our objective is, for each unknown gesture sample, to attribute it one and only one class, all our k-NN approaches will actually be 1-NN.

2.4.1 Rubine's Algorithm

Considered as one of the "pioneer" algorithm used in stroke gesture recognition, Rubine presented his algorithm in his Ph.D thesis in 1991 with his gesture recognition framework GRANDMA (Gestures Recognizers Automated in a Novel Direct Manipulation Architecture) [14]. Although this algorithm uses a quite large set of features extracted from gestures (13, to be precise) and his classification method (Linear discriminant analysis) is very different from the targeted approach of this work, we wanted to integrate it in our benchmark tool, at least for its historical value.

2.4.2 The Dollar Family

The dollar family offers a k-NN approach to gesture classification. It started with the development of $1 in 2007 [21].

$1 *Wobbrock et al.* proposed to use average euclidian distance between gestures to compute a similarity score between an unknown gesture and all the templates it has in its memory. $1 also has an important pre-processing phase during which it re samples all gesture samples we feed it to a fixed number of sampling points, to achieve sampling rate invariance. Then, to cope with potential scaling factor differences, $1 rescale every gesture. This pre-processing phase (resampling and rescaling) is performed for any gesture, during both the training phase and the test phase. The most constraining limitation of $1 is the fact that it is unable to handle multi-stroke gestures (i.e, gesture whose strokes count is strictly greater than 1). Technically, you can give multi-stroke gestures as input for $1, but they will be interpreted as a single stroke gesture.

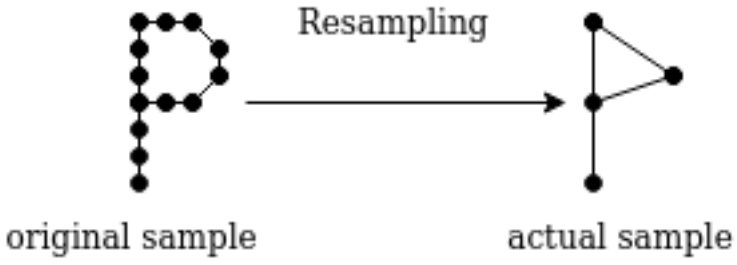


Figure 2.3: Resampling of a gesture sample that represents the letter "P"

$N To bear the aforementioned limit of $1, *Anthony et al.* proposed, in 2010, an extension of $1, called $N [2]. This extension adds two important improvements to $1 while using the same core concepts (in fact, $N uses $1 "under the hood").

- Support of multi-stroke gestures : they modeled "multi-stroke as a set of uni-strokes representing all possible stroke orders and directions". [2] What it means, is that every stroke composing the input gesture will be considered independently from the others regarding its path direction. Then, every possible combinations of all stroke with all their possible directions will be considered.

- Introduction of bounded rotation invariance: some gesture only differ by their orientation (think about the letter "A" and the universal quantifier symbol "∀", for example). $1 was unable to differentiate these two symbols because it used full rotation invariance. To enlarge the panel of gesture classes $N can handle, they introduced this feature.

$N-Protractor in 2010, *Yang Li* proposed a novel approach to perform gesture classification that require much less computations [9]. By calculating optimal angle between vector forms of an unknown gesture (g, noted in its vector form as v_g) and a known template gesture (sim. t, t_g) using the inverse cosine distance of their respective vector representation, he managed to drastically reduce computation costs to classify gestures, with a nearest neighbor approach.

$$S(t, g) = \frac{1}{cos^{-1} * \frac{v_t * v_g}{|v_t| * |v_g|}}$$

Figure 2.4: Inverse cosine distance used in Protractor

in 2012, the authors of $N integrated the solution of Mr. Yang in their multi-stroke recognizer : $N-Protractor [3].

$P / $P+ The main issue with $N is the potential combinatorial explosion generated by its multi-stroke model, even with Protractor, to a lesser extent. Imagine a gesture that represents a cube. This cube can be drawn with 1, 2, 3 or 4 strokes and each stroke can have different orders and directions. If we consider such a gesture, there will be up to 442 possible cases... For a single gesture sample that represents a square. To cope with this combinatorial explosion, Vatavu et al. proposed a "point-cloud" representation of gestures, dropping thus features like stroke order and direction [19] . Classification is performed by finding a point-by-point matching between a test gesture and all gesture templates.

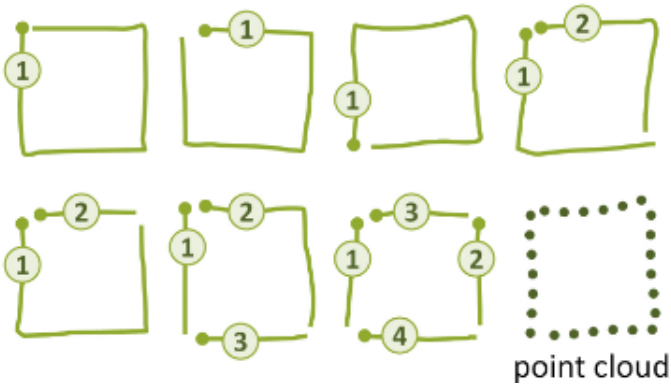


Figure 2.5: Point-cloud representation of a "square" gesture in $P, vs samples of possibles multi-stroke model representation of $N, from [19]

5 years later, Pr. Vatavu published an improved version of $P, called $P+ [18]. He managed to improve the performance of $P in specific contexts of use, both in terms of recognition rate and time, by improving the algorithm in 2 different ways :

- Improvement of $P point-by-point matching procedure by computing optimal alignment between points.
- Exploiting the "shape" of the point-cloud representation : $P completely discards order of points in their stroke and stroke order, as we mentioned in the previous paragraph. The only thing it sees is a set of 2D coordinates. $P+, however, adds to the euclidian distance between points another criteria in its point-matching procedure: it takes into account the turning angle between consecutive points.

$Q In 2018, Vatavu et al. released another member of the dollar family, claimed to be the most advanced and fastest of the family : $Q [20]. Built on top of $P, this version is optimized to run on devices with low computation abilities, like low-end mobile devices. The main improvement they brought to $P is a change in the point-cloud matching computation procedure. $P uses a dissimilarity score when comparing a new gesture to all templates in memory. This dissimilarity score is computed as a sum, it is therefore possible that, mid-way during the computation, we realize that this dissimilarity score is already higher than the lower one yet. Just by keeping track of the lowest score and updating it at each iteration, we can decide to abandon the cloud-point matching as soon as the dissimilarity score goes higher than this lower bound.

2.4.3 Penny Pincher

Many algorithms from the dollar family go through a resource intensive pre-processing phase of gesture samples before they can start any recognition process : rotation, translation, scaling, combination of strokes... In 2015, Taranta et al. proposed another algorithm which uses a lot less pre-processing of gestures, resulting in a lighter model and a faster recognition process [15]. The philosophy of this algorithm is very simple :

- Pre-processing: every gesture is resampled with a constant number of ordered points. That's it. That's the only thing Penny Pincher does before being able to start its recognition process.

- For classification: Penny Pincher defines a similarity measure between two gestures by computing the sum of vector angle between each matching points of both gestures. Other assumptions are made to further simplify calculations of the between-gestures similarity score. Result : the computation is nothing but additions and multiplications. There is no complex geometric procedure involved.

This come of course with limitations. For instance, Penny Pincher is, strictly speaking, not a multi-stroke recognizer. Authors of the algorithms argue however that this constraint is not a problem if there are enough samples per gesture class: *"We find that with a sufficient number of training samples, this achieves the same effect as $N's scheme to internally generate all possible permutations of a gesture from a single sample"* [15].

2.4.4 !FTL, !NFTL

With the same objective than Penny Pincher on their mind (which is, reducing the heavy pre-processing phase each gesture has to go through) researchers proposed a novel approach for nearest neighbor gesture recognition [16]. They proposed a vector based representation of gestures and defined a *Local Shape Distance (LSD)*, which measures dissimilarity between two gestures. The idea is that this vector-based representation of gestures holds more informations that point-based representations, like position and direction in space, velocity.. Which makes thus the pre-processing phase irrelevant. How does it work ?

- Gestures are sampled into vectors.
- Pairs of vectors, in their turn, define triangles.
- Triangles of both gesture (training and candidates) are compared : their dissimilarity score is given by (normalized) euclidian distance between the similarity ratios of triangles for (!NFTL) !FTL.

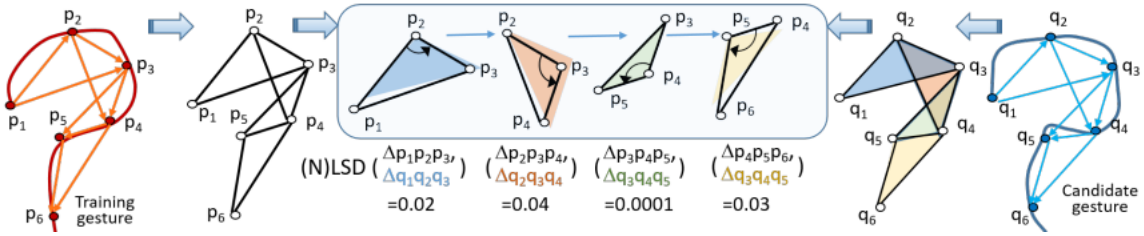


Figure 2.6: Classification of gesture in (N)FTL, from [16]

2.5 Summary

We went through a brief tour of gesture recognition. We will focus on nearest neighbour approach for all reasons mentioned above (rapidity and ease of deployment, easy to understand and use...). The problem is that, to the best of our knowledge, there is no tool which gives the possibility to evaluate them under the same circumstances and contexts, nor that is able to see how their performance changes (or not) when experimental environment changes. If such a tool already exists, we are not aware of it or it has not been made public. All existing algorithms are tested, but not under the same context, not

with the same input, not by different teams. They have not been confronted to different datasets, different use cases nor contexts. Therefore, our main concern is to create a tool focused on replicability and reproducibility [1].

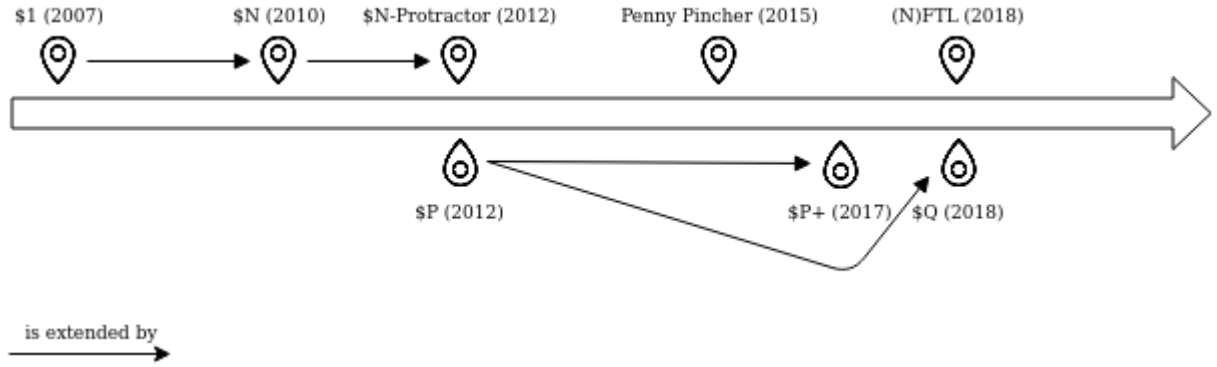


Figure 2.7: Timeline of considered template matching algorithms

[1] https://www.acm.org/publications/policies/artifact-review-badging

Chapter 3

Gester, a Software to Benchmark Stroke Gesture Recognizers

3.1 Overview of this chapter

This chapter is the one dedicated to the core of our work, the provided software we built to perform benchmarks of stroke gesture recognizers : Gester. We will start by giving the specifications of our applications : what are our features ? We will then justify our choice of tools and technologies we used to build the software and give useful informations about them. We will then, based on the specifications, describe the features of our application in its current, as of June 2020, then state and give details about the structure and architecture of the software.

3.2 Specifications of Gester

Based on the pursued objectives we mentioned in the introduction chapter of this document, we will now state what are the different functionalities that should be in our application. We will first express them in the form of user stories, using the format defined by *Mike Cohn* [6]. We will here differentiate the "user" from the "developper". The "user" is the one who only wants to perform tests by using features of Gester. The "developer" is at the same time the one who wants to use Gester but also the one who wishes to extend it, by adding new features, new functionalities. Since our application is mainly designed for researchers, it makes sense, here, to make user stories dedicated to developers, not only users. Those users and developers stories will constitute our work baseline to develop the application.

3.2.1 User Stories

- As a user, I want to upload a gesture set, in order to configure tests.
- As a user, I want to chose my candidates, in order to choose which one I want for evaluation.
- As a user, I want to set the number of training templates, in order to train my candidates.

- As a user, I want to set the number of sampling points, in order to choose the complexity of the training templates.
- As a user, I want to select the input samples, in order to choose what templates could be used for my candidates.
- As a user, I want to run tests, in order to know what candidate is the fastest.
- As a user, I want to run tests, in order to know what candidate is the most accurate.
- As a user, I want to test my candidates in different scenarios, in order to see how they behave in different contexts.
- As a user, I want to export the results of my tests, in order to use them elsewhere.
- As a user, I want to create an account, in order to be authenticated.
- As a user, I want to log to my account, in order to be authenticated.
- As an authenticated user, I want to store my results, in order to retrieve them later.

3.2.2 Developer Stories

- As a developer, I want the code to be syntactically neat, in order to read it easily.
- As a developer, I want the code to follow some standard, in order to manipulate it easily.
- As a developer, I want to add the support of a new algorithm, in order to have more candidates to test.
- As a developer, I want to add the support of a new scenario, in order to diversify the contexts of tests.
- As a developer, I want to easily add new type of visuals, in order to have better insights of my tests' results.

3.3 The Tools

We now have to choose how are we going to build our software. In this section, we are going to quickly review the different possibilities, explain why we preferred some over the others, and then describe how do the chosen tools and technologies work, how we used them.

Desktop, mobile, web? The first question is "what kind of app do we want to develop?" Since one of the main objective is to develop an application that can be used in multiple contexts, including the device it will run on, we immediately discarded desktop applications. We are absolutely not saying that it is impossible to build a desktop application and then port it to mobile or the web, but it certainly requires a lot more work. We are now left with two possibilities : mobile or web? We chose web, since it is surely easier to go from a web application (running on a PC, for example) to a mobile app than the other way around. Furthermore, with modern web application development frameworks, there is a

good chance that your web application will work, once deployed, on mobile browsers as well.

Why a web application? Mostly, convenience. A web application is a "develop once, use everywhere" solution. Since our focus is on simplicity, flexibility and ease of maintenance, choosing a web application seems sound to us. The biggest advantage is that any modern device (smartphone, tablet, computer, some other smart devices...) come pre-equipped with a run time environment for web applications : a web browser. Once deployed, that's the only thing a user will require to run the application. It is thus difficult to make it easier. Another reason, much more simpler, is the fact that all considered algorithms already had a JavaScript implementation.

What architecture for a web application? The most common and probably most famous model is *MVC*. Many different frameworks implement this model and are widely used : Ruby on rails (Ruby), ASP.NET (C#), Django (Python), to cite a few of them. However, this architecture has also its set of drawbacks: setting up MVC architectures can be difficult and their is sometimes a big dependency between the different elements of the architecture. As a result, if you want to change something in your model, for example, it is much likely that the whole entity will be impacted : you will have to change the view and the controller too. If we keep in mind that we want Gester to be easy to extend / maintain / change, we absolutely need to avoid this. However, we do like the flexibility and the adaptive interfaces that offers MVC. Is there a way to keep those advantages and to mitigate these drawbacks ? It turns out that yes. There are a couple of modern JavaScript framework who propose their own architecture, which are "refactored" version of the good old MVC architecture. As of 2020, three names stood out : React, Angular, Vue. Those are the three currently most endorsed JavaScript front end framework [1]. Those three tools offer each their own highly flexible model to build web applications. How to decide ? The first framework we discarded was Angular, for the following reason : its the only one (among those three) where the developer has to interact directly with the *DOM*, whereas the two others (React and Vue) both use a virtual DOM. Using a virtual DOM offers many advantages : first, an abstraction between your components and the *HTML* of the rendered pages of your application, which makes it simpler for the developer to declare components. Second, it offers many optimizations : when your interface should refresh, the framework calculates itself, depending on what has changed in the virtual DOM, what should really be updated in the DOM. Now, how to decide between React and Vue ? We preferred React over Vue, simply because it is much more mature and has a much larger community, which makes it really easy to learn. Since our application needs a client-server architecture, React alone is not sufficient, because it is a pure front-end framework. Our choice was thus directed to the most common, the most famous full-stack using React : the *M.E.R.N* stack.

The M.E.R.N stack. The four technologies of this stack, coupled together, define a simple web client-server architecture :

- **M**ongoDB, a cross-platform document-oriented database program. The advantage of MongoDB is that it stores data in flexible, JSON-like documents without a pre-

[1] source : https://2019.stateofjs.com/front-end-frameworks/ (last consulted 4th of June, 2020)

defined structure. This is what makes it very convenient when you work with web applications, or more generally, JavaScript.

- **E**xpress.js, a web-application open source framework designed specifically to build web applications and APIs. The most consequent part of the software is clearly the client, since all intensive computations are performed client side. The server is only used, in our case, as storage mechanism interacting with the client. Therefore, a simple framework like Express, providing "out of the box" working back-end, is a good perk.

- **R**eact, a JavaScript library for building user interfaces developed and maintained by Facebook. This is clearly the most important tool used to build our software. Therefore, we will focus on it and give much more details in the next section, dedicated to react, how and why did we use it.

- **N**odejs, a JavaScript run time environment mostly used for running server side scripting. Node is useful to run JavaScript "out of your browser". This is the environment used by Express to run the server. A side note : the client, in React, runs in its own environment : your web browser.

3.3.1 React

React has many advantages. As mentioned in the introduction chapter and in the conclusion of our "state of the art" from previous chapter, our main focus is to build an app that is : simple, flexible, easy to extend and maintain. Those are precisely the main advantages and focus of React itself. React is a "learn once, apply everywhere" library. The same design principles can be used to build all the components of your application. This also offers a good portability potential since React's mobile version, "React native" uses the same concepts and design principles to build mobile apps for iOS and Android. React is also easy to learn : it is really straightforward to build and render component, since we are, with one exception, never directly interacting with the *DOM*. The component renders when it is "mounted" and re-renders automatically when there is an update in its properties. Here are the main characteristics and important concepts of React we want to define for our reader to have a better understanding of this tool. If you wish to have more informations than what is provided below, we suggest the reader to have a look at the official React documentation[2] or the book from Banks and Porcello, which covers all the essentials of React [4].

Divide and conquer. Because components can be used in the declaration of other components, React offers a "divide and conquer" approach for building applications. It means that, all components should be very little and a change in one component should not require changes in any other component, even if they are used together. This is the main reason we consider React to be really easy to extend and maintain.

Declarative nature. To define components and how they should render, React uses *JSX*, an extension of EcmaScript that uses an HTML like syntax to declare the visual "shape" of our components. It is a way to inject HTML into JavaScript. JSX allows,

[2]https://fr.reactjs.org/docs/getting-started.html

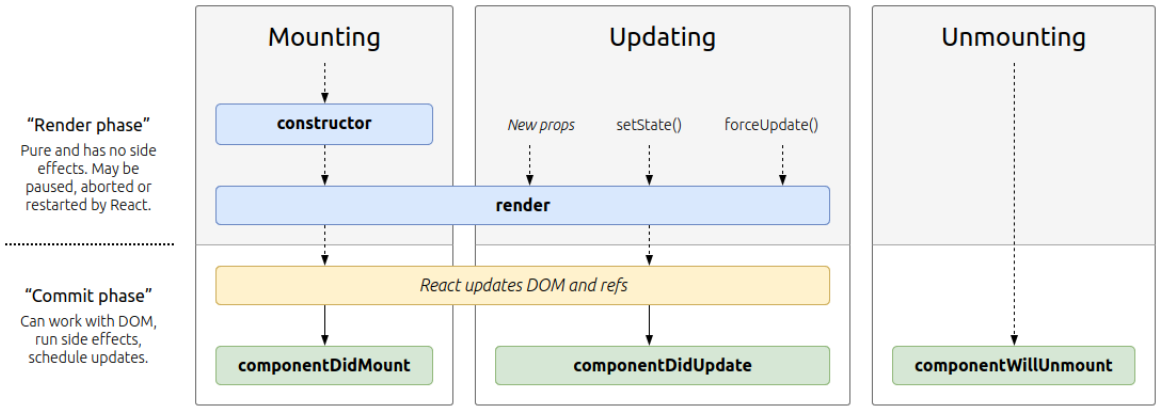


Figure 3.1: React's most commonly used life cycle methods. Diagram from "wojtekmaj" on github: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram

thanks to this declarative approach that blends with pure JavaScript, we can easy build component and specify how they should render.

State, props and life cycle methods. The state is a local object maintained by a React component. Only the component in which it is defined can access its fields. It is useful when you want to tweak the way your component gets displayed. Props, which is the short version of "properties" is what you pass to other components when you build your application. You may want some components to adapt their behaviour depending on the informations it received from its parent component: this is where props come in handy. Finally, life cycle methods allows you to have more customization possibilities for your components. All these methods are defined by React and each of your component can (but does not necessarily have to) override these methods.

To illustrate these definitions and concepts, have a look at figure 3.2 below. It is a simple yet very illustrative example of how React works.

```
// HelloWorld.js
import React from "react";

Class HelloWorld extends React.Component {
    constructor(props){
        super(props);
        this.state = {
            name: "world"
        }
    }

    render(){
        const {name} = this.state;
        const {textColor} = this.props;
        return(<h1 style={textColor}> `Hello ${name}` </h1>);
    }
}

export default HelloWorld;

//index.js

import React from "react";
import ReactDOM from "react-dom";
import HelloWorld from "./HelloWorld";

ReactDOM.render(
                <HelloWorld textColor={{textColor:"#FFFFFF"}} />,
                document.getElementById("root")
               );
```

Figure 3.2: A simple React application

3.3.2 Redux

React's natural data flow can be viewed as a one-way parent to child stream. If you want to pass data to another component, the most common way, as explained in previous section, is via props. However, this also imply the fact that there is a parent / children relationship between the component who send the data and the one who receives the data. If you want to send data from a children to the parent, it is possible, but a bit less trivial than just passing props down the tree. To do this, the most common way is to pass a function from the parent to the child. The child will, in its turn, call this function with the data it wants to send back to the parent. Now, imagine another, even less trivial case : how do two sibling components should communicate? If you only uses React's built-in props and state, your components will get really messy and hard to maintain because of this one-way data flow. This is where Redux [3] comes in to save the day. Redux offers a centralized store for our application. We can, virtually, store any data we want to the redux store. But what is even more interesting, is the fact that we can map any piece of data stored in redux to React components' props ! How do we do this :

- 1) Bridge our React application to our Redux store.

[3] https://redux.js.org/

- 2) Connect our component to the Redux store.
- 3) Indicate which item from the Redux store we want our component to "listen to".

Redux is thus very useful to simplify our components in two different ways: first, it offers a centralized store, simplifying a lot the data flow between our components. Second, it offers the possibility to treat, manage and manipulate data outside our React components.

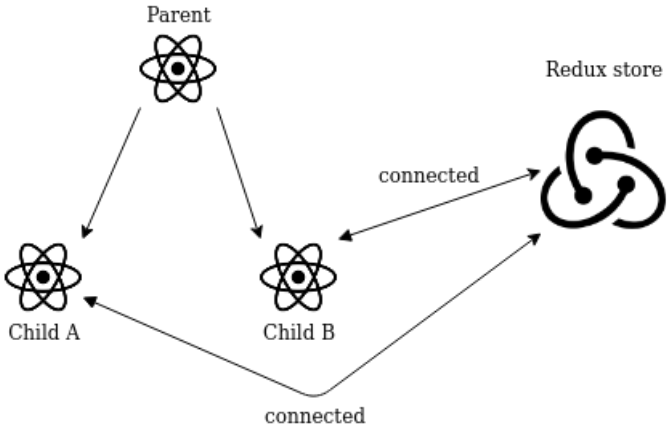


Figure 3.3: Data flow between sibling components with redux

3.3.3 Other Notable Libraries

Like any JavaScript application, there are many different libraries that were used. We will mention here the most notable ones and tell how they were useful to us.

React bootstrap. It is the bridge between the famous and quintessential Twitter's framework "bootstrap" and React components. This simply permits us to handle bootstrap components as pure React components, which drastically reduces the number of *SLOC* needed to declare our components. This library makes *GUI* design of our application much easier. It provides most major bootstrap components as pure React components, by rewriting all the JavaScript part of bootstrap, but still entirely relying on bootstrap's *CSS*. Result: tidier components that still profits from bootstrap's responsiveness and ease of use [4].

Lodash. This is an utility library that provides a lot of function to handle JavaScript data structures like arrays, objects, maps, sets... Its main concern is performance and flexibility, by offering a large range of possible operations[5]. Our app deals with a lot of objects and arrays, since the input data for our benchmarks, our gesture samples, are *JSON* objects and arrays. Lodash has been very useful to manipulate our input data, but also data from our Redux store.

CanvasJS. It is a graph plotting library that is highly flexible and customizable. It offers a lot of parameters, including but not limited to : animations, multi-axis display, many different graph types, colors, scale, export functions... It is very convenient because

[4]https://react-bootstrap.github.io/
[5]https://lodash.com/

it offers an abstraction layer above HTML canvas. we can generate any type of plot we want without having to deal with canvas operation[6].

Mongoose. This library provides an interface between our server and our MongoDB database[7]. Its main objective is to make connection between a server and a database easy, but also to provide schema-based solutions to model our application data. A mongoose schema is, roughly, a JSON object defining a structure for the documents we will store in our database. Schema comes with nice features, such as type restriction, documents constraints and validation...

Jest. It's a JavaScript testing library giving many possibilities to thoroughly test your applications[8]. It is very simple to write tests with this library, although we'd argue that it is very messy to setup and does not blend really well within the project's structure: there are indeed a lot of "boiler plates" required to use Jest. It is however very convenient since it has a big community, increasing popularity and is regularly updated. It is fully compatible with React.

3.4 Development Process

Learning phase. According to the thesis' specifications, Gester must be a web application must be developed using the aforementioned M.E.R.N stack. A considerable amount of effort will be placed in the learning of these technologies as I, the author, have absolutely no experience with some of these technologies and vestigial experience with others. This learning phase is rather continuous than punctual : I will have to learn on the way, this learning phase will thus be in parallel of all other phases of the development process.

Analysis. During this phase, the objective is to translate the specifications into user stories, and then translate these user stories into abstract features. Those abstract features will then be developed during the next phase. It is thus important to gather the main knowledge about both gesture recognizers and development tools in this phase. That's the phase where a first draft of the structure and architecture of the software will be defined.

Development. The phase where most of the application code will be produced. The objective of this phase is, in the end, to have a first version of a functional application. In this phase, features extracted from the specifications during the analysis phase will be concretely shaped. In parallel, test suites will be provided to ensure good quality of essential features of the applications.

Review. At this point we should have matter to discuss : is the application good enough with respect to the specifications ? What are the prospects for improvement we can bring to this application ? During this phase, we will establish and address a list of potential improvements and possible limits of Gester. If necessary, this review phase may induce another development phase and another review phase, until the application satisfies the specifications.

[6]https://canvasjs.com/
[7]https://mongoosejs.com/
[8]https://jestjs.io/

Documentation. During this phase, the main objective will be to provide a complete suite of documents aiming to help future developers or users to use/extend the application. This phase is dedicated to both documenting the code with specifications, comments, linting, indentation fixes... But also, documentation you will find in the dedicated chapter of this document.

3.5 Features

Regarding the specifications, we identified a set of features that are required to be implemented in order to perform benchmarks: these are qualified of "core" features. Each of these features were addressed to solve the requirements identified during the analysis, respecting our user stories. They will be described here. Further explanations, including a user guide can be found in section 5.2.

3.5.1 Core Features of Gester

Dataset Upload

Description. Our users should be able to upload a dataset (also called in our case "gesture set") in the application to perform their tests, following the data format specified as follows :

- A gesture set is a folder, containing as many sub-folders as there are distinct human being(s) who made the gestures that were gathered in the gesture set. All gestures within the same gesture set are supposed to come from the same kind of device (smartphones, tablets, ...).
- Each sub-folder regroups thus all the gestures from one and only one human being, also called in our context : a participant. A participant is defined by the name of its folder.
- Each gesture is in its own file, inside a participant's folder. Each gesture has a JSON format and is expected to have at least the two following properties :
  - name: a string which gives the gesture *class* of this sample
  - strokes: a 2D array, where each sub-array contains all the points of one stroke from the gesture sample. Each point is an object and is expected to contain at least the three following properties : "x" and "y", its coordinates in a 2D landmark and "stroke_id", which is a number identifying the stroke to which this point belong to.

Implementation. The implementation of this feature is simply two buttons, each one of them allowing our users to upload a whole folder, the gesture set. Each of these buttons serves to upload distinct gesture sets. Our applications can handle thus two different gestures sets in parallel, which is really useful if you want to compare how recognizers behave when confronted to gestures coming from different platforms. Note that, a user does not necessarily have to upload two gesture sets in order to perform tests. This is why, only the first gesture set is mandatory. Our upload feature also automatically extracts useful informations, such as :

- The identifier of each participant.
- All gestures grouped per participant.
- The distinct gesture classes the gesture set contains.

We also added a small "verification" functionality : we warn the user in case of problem encounter while reading the gesture set.

Scenarios : our Different Contexts of Use

Description. One of the main interest of this application is to regroup different algorithms, compare them under the same testing conditions and observe how their respective behaviour changes when we change their testing conditions. This is why, the application should handle different contexts. One way to diversify the contexts is to handle different scenarios to test the algorithms.

Implementation. We defined four different scenarios to which we can submit all the supported recognizers. It is very likely that recognizer will receive input from many different users, having a (more or less) different writing from each other. So the question is, will its performance vary if the participant(s) who feed gestures for the training phase are not the same participant(s) than those we'll use during the test phase ? That's what we will call "user (in)dependent", abbreviated by *UD / UI* from now on. This notion was first introduced by Vatavu R. in [17]. In today's world, it is also true that we have many different platforms where *HCI* is involved: billions of smartphones, personal computers, desktops, tablets etc... We want to observe how the recognizer behaves depending on what platform the input come from : that's "platform (in)dependence" *PD / PI*. More formally, we have :

- User dependent (UD) : training and test data both come from the same user, the same participant.
- User independent (UI) : training and test data are not from the same participants. Typically, we train the recognizer with gestures randomly drawn from all participants but one and we test it with samples from the remaining participant.
- Platform dependent (PD) : all data were collected from one and only one platform.
- Platform independent (PI) : samples used for training and for test come from different platforms.

Our four resulting scenarios are thus the combinations of user (in)dependency and platform (in)dependency as shown in figure 3.4. Note that, in accordance with what we wrote for the dataset upload feature, if a user wishes to run PI tests, then there should be two different gesture sets uploaded in the application.

PD \ UD	YES	NO
YES	Scenario 1	Scenario 2
NO	Scenario 3	Scenario 4

Figure 3.4: Test scenarios

Test Case Configuration

Description. It is also really important to have parameters for our tests. In fact, it is not enough to observe if there are variations in terms of performance and accuracy if we change the scenario. It is also important to see how well (or not) a recognizer behaves when we change :

- Its number of training template
- the number of sampling points per gesture sample
- The number of time we repeat the train / test procedure

Implementation. We added support for all these parameters. A user can thus create one or several tests, and for each test, the user can select : the tested algorithm, the tested scenario, the number of sampling points for each gesture sample, the number of training templates per gesture class to train the recognizer and the number of repetitions for the whole process. Furthermore, all these parameters come with a validation procedure : a user can't create a test with faulty parameters. Also, we know that, due to the nature of most considered algorithm we integrated in Gester (see 2.3.5), it is much likely that the time required to perform the test will greatly increase with the number of training samples per gesture class. Therefore, for this parameter, the user can chose precisely which values he/she wants, instead of specifying a range of values.

Gesture Set(s) Configuration

Description. Our users should not have to use the whole gesture set to perform their tests. Therefore, it should be possible to (de)select at will, all samples of a given participant, one specific sample, all samples from a given gesture class. This will also make our users able to test whether there's a difference in terms of mean recognition rate and / or recognition time if we add or remove a specific gesture class or a specific participant.

Implementation. This feature is implemented as lists, one for each handled gesture set. Our users have the possibility, through these lists, to select or deselect sample(s). All the selected samples from all selected participants will be remembered. Samples that are not remembered won't be used in the benchmark procedure, neither in training phase nor in testing phase. We consider a participant to be selected once at least one of its gesture sample has been selected by our user. To limit the memory usage of our application, which is already consequent depending on the size of gesture sets, we impose a one-to-one relation between configuration of gesture sets and a series of tests. In other words, if a user defines,

say, four different test cases, he/she can have up to four different test configuration(s) but only one gesture set configuration. If he/she wishes to change the gesture set configuration but keep the test cases, then the testing procedure must be re-run with another gesture set configuration. This limit is actually not that constraining since our users can "stack" results of different series of tests (they can perform a second benchmark without losing results from the first one).

Results Display

Description. Our users have the possibility to configure different test scenarios with a full sample per sample choice of what candidates, what gesture samples will be used during their tests. Now, all these features are pointless if their is no result to show in the end. Our users should therefore have a way to see the results, test by test, in terms of recognition rate and recognition time.

Implementation. When the test procedure is done, we transform "raw results", that is, the output of classification functions of the recognizers into the following results :

- Confusion matrix[9] : a two dimension array with gesture classes names as both rows and columns names. The row's name indicates the expected gesture class, the column indicates the gesture class attributed to the test sample by the recognizer. Each cell is expressed in percent, the sum of each row must thus be equal to 100. It is very useful if we want to have the recall (the "true positive rate") and the miss rate (false negative rate) per gesture class. More formally : if we have G, a set of distinct gesture classes, $M \epsilon \mathbb{R}^{|G|x|G|}$, our confusion matrix, we have: $M(g,g) = recall(g)$, with $g \epsilon G$. Also, we have that, $\sum M(g,h) = missRate(g)$, with $g,\ h \epsilon G$ and $g \neq h$.
- Bar Charts : the user also has the possibility to see results in terms of recognition rate and mean recognition time per gesture class or per participant. They are displayed under the form of colored "bar charts", each bar being associated with one participant / one gesture class.
- Spline charts : if (and only if) the user selected different values for the number of training samples per gesture class, a spline chart will be displayed. This charts has a double y-axis scale, showing at the same time the evolution of both recognition rate and mean recognition time for a recognizer.

3.5.2 Additional Features of Gester

Consistency Model

Description. There should be a set of rules defining what is a correct gesture set configuration with respect to some test(s). Our users should not be able to run tests if their gesture set configuration is not consistent with their tests configurations. If there's no such "consistency model" then the application could potentially produce tests results that do not reflect the actual performance of the tested recognizer. Let's take a concrete example to clarify : if a user wants to run a test on some specific gesture class with some

[9] https://en.wikipedia.org/wiki/Confusion_matrix

specific participant, using for both train and test phase samples from the same participant (i.e, scenario UD / PD), but its current gesture set configuration only contain one sample of the said gesture class... Then, the train samples and test samples will be the same gesture, resulting in a perfect match every time : the application will report a recognition rate of 100 %, which is obviously not representative of a real test case.

Implementation. We added a consistency model between tests configurations and gesture set(s) configuration to ensure that no unrepresentative results will ever be produced by our application. The nice thing is that this feature is entirely automatic. No action from our users is required to verify the consistency between their tests and their samples selection. However, the user will have to either fix his/her gesture set(s) configuration or adapt his/her test(s) configuration(s), since this consistency is a hard constraint to be able to run the tests.

User Account

Description. Our users should be able to register and authenticate in our application in order to perform their tests.

Implementation. We have two similar pages, one for sign up and one for login. The user needs to provide an e-mail and a password of at least 6 characters. Those pages come with front-end validation to avoid sending faulty credentials to the server. Of course, when the server receives a login / sign-up request with credentials in request's payload, it will check the validity of credentials : for login, if the email correspond to a user in the database and if the password match. For sign-up, if the e-mail is not already attached to a user from the database. When our users provide credentials to login or sign-up, either one of the two following will occur: an error message is displayed to inform of front-end / back-end validation error. The user provided correct credentials and is redirected to the homepage.

Save Results

Description. Our users should have the possibility to save results of their tests to retrieve them later.

Implementation. Since our application handles user accounts, we decided to add the possibility, for an authenticated user, to save his/her results. We will bind them to the corresponding user account, which is also very easy since we impose a uniqueness constraint on our users e-mail address. The user can, from the homepage, go to a dedicated page where they can see all their previously saved results. Our users have to give a name to their results before saving them, which will help them to retrieve them easily later.

Visual Representation of Gesture Samples

Description. To have a clear idea of what each gesture sample look like (perhaps, manually inspect its shape to see how close it looks to other samples) our users should have the possibility to see what it looks like.

Implementation. Our users have the possibility to see how does any gesture sample looks like. We draw, point by point, the gesture in an HTML canvas. Each point is then connected to its predecessor with a line. Since every gesture sample has a "strokes" array in its raw form (see section 3.5.1), containing all the points defining the stroke(s) of the gesture, it is very convenient. Since our application should be able to handle gesture sets with several thousands of samples, we chose not to automatically display gesture samples and to limit the number of displayed gesture to one at a time. Canvas rendering can become a really heavy computation when there are many of them and we really wanted to keep the swiftness of our current interface.

Results Export Functions

Description. Our users should be able to export their results if they wish to change their format or use them outside of the application. Since we only offer a limited range of different results (see 3.5.1) maybe our users will want to change the results display, or perhaps use them to compute other values.

Implementation. We added the possibility to trigger browser file download of all our results : our confusion matrices can be exported to CSV, all our graphs can be exported as either images (PNG or JPEG) or CSV.

3.6 Structure and Architecture

The structure, here, refers to the different accessible resources of our application. We will, in the following sections, give the structure of both client and server, explain the content of each sub-parts of their respective structure, for both client and server. We will then conclude by giving you a general graphic view of the application's architecture. Reminder: the sole purpose of this chapter is to give explanation about what Gester is, what is it made of, giving the least possible technical details. If you wish to have more technical informations about, for example, the structure of our client, server, interactions between client and server, we refer you to the "documentation" chapter of this document.

3.6.1 Client

The client is divided in five different pages. We will cite them and give their purpose here. For more informations and explanations about the different user interactions with these pages, we refer you to the documentation chapter of this document (see 5.2).

- "/": the homepage. That's the first page our users will see when they load the application. This page gives navigation links to all the other pages of the application. This page is also the place where our users must upload their gesture set(s) in order to start making tests.
- "/login" and "/signup": the login and sign-up pages. These pages only contains a form, made of two fields and a button : an input field for an email address and another input field for a password. This is where the user is expected to provide its credentials to login or to create a new account. There is no redirection link in those pages. However, upon successful connection or sign-up, the user will be

automatically redirected to "/". Also, the access to these pages is restricted : a user can't access any of these two if he/she is already logged in.

- "/benchmark": the principal page, divided in two different parts : the left part, dedicated to the tests setup (test(s) configuration(s) and gesture set(s) configuration). The right part, dedicated to results display : our charts, confusion matrices and results saving feature. To access this page, the user must have at least the first gesture set (which is, the mandatory one) uploaded.

- "/results": the page where users can see their previously saved results. This page also has a navigation link leading back to the home page.

All these pages, without exception, interact with the server for some specific tasks. These interactions will be described in the next section, as well as the structure of the server. Every page is not made of one and only one React component, but of many. We will limit ourselves here to describe the different kinds of components that our application is made of, without precisely describing each one of them. For more technical details and graphs, we refer you to the "documentation" chapter of this document. Here is, roughly, how our client sources are divided :

- tests: is a folder containing our three different Jest test suites. One is dedicated to test our Redux store and thus the general dataflow between React components and Redux. The second one is dedicated to our consistency model : given correctly formed parameters, those tests ensure that either an error is reported to the user in case of conflict in our user's configuration or either nothing is reported if there are no conflict. The last one is dedicated to the benchmark procedure itself : we test here our different scenario with different parameters to ensure that the correct procedure, correct recognizers and correct parameters are used to run the tests.

- benchmark: is a folder containing all scripts related to the benchmark procedure: transformation of raw gesture sample into template gestures readable by a recognizer, instantiation of a recognizer, train and test phase according to the selected scenario, formatting results... Note that all files contained in this folder are not React components : they are pure JavaScript.

- action and reducers : are the folders containing all Redux related boiler plates. A reducer is a concrete object stored in our Redux store. An action is an event triggered to update values stored in our reducers.

- components: is the folder containing all React components. Pages, forms, buttons, alert messages, modals, input fields... In other words, everything our users see (the GUI) is defined in this folder. It is probably the most consequent source folder of the application. For the reader who wishes to have a general idea of what our application is made of, the "page" decomposition explained above is enough.

- style: is a folder containing all custom CSS we defined for our application style. Note that, thanks to react-bootstrap library, styling is a really tiny part of the application since we didn't have to define most styling properties.

- index.js: is the file initiating the render phase of the application when a user loads it.

- App.js : is the root component of our application. It defines our five pages and their respective root component.

3.6.2 Server

The server is divided in three different ExpressJS's "routes". An ExpressJS route is simply a middle ware that associates three things. First, an URL that should be queried by the client (the set of route's URLs actually defines the client-server *API*). Second, a request type (GET, POST, ...). Third, a function to handle the request. Those three routes interact with two different mongoose models. Those two models, in their turn, interact with our MongoDB database. Here is an overview of our server's routes and models:

- "auth": is the route that will provide users signed token after verification of credential. This route handles only one request of type "POST", sent from client's "/login" page.
- "users": is the route that will register new users in our database. It also handles only one POST request, sent from client's "/signup". This request will, provided that the email address in the request's payload is not already associated to another user, save a new user in our database. Like "auth" if the user registration is successful, it will respond with a signed user token.
- "results": is the route that handles results records from our users. This route handles two different requests. A POST request sent from the "/benchmark" client page. It adds results and bind them to the authenticated user that sent the request. Another request of type GET. It will return all results records associated to the authenticated user. This request is sent from client's page "/results".
- "User": is a mongoose schema giving a data model for a user. A user has an e-mail, a password hash, a function to set its password and a function to convert a user record into a signed token.
- "Result": is a mongoose schema for users' saved results. A result has an owner (the email address of the user who saved these results), a name (a unique identifier for the results) and a data object, which is the JSON object containing all results as they are sent from client page "/benchmark".

3.6.3 General View

The following schema shows a general view of the application. We see how both client and server work, what are there main components. It is however, not detailed enough to have a deep understanding of the used tools. If the reader of this document wishes to have more informations, we suggest to start by reading the provided references, reading the official documentations of the used tools and finally, reading chapters two and five of this document to have a even more informations about our software and the algorithms it supports to perform tests.

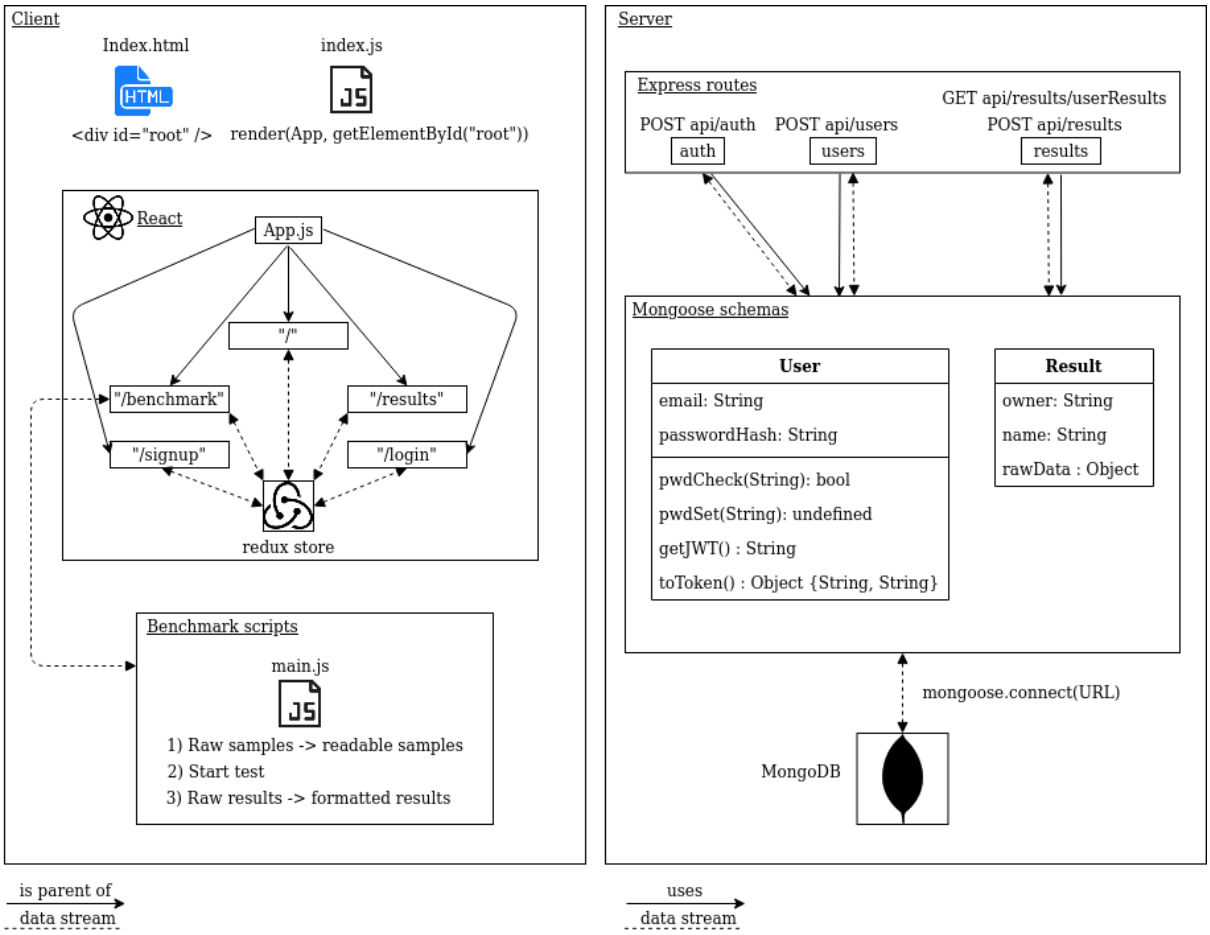


Figure 3.5: Simplified architecture of the application

Chapter 4

Benchmarking of Gesture Recognizers

4.1 Overview of this chapter

This chapter will contain detailed definitions and descriptions of the suggested benchmark methodology. We will explain here what is the purpose of a benchmark, how useful it can be and how can we apply a benchmark procedure in stroke gesture recognition. We will also perform some tests on our supported *recognizers* to give examples of results our application is able to produce.

4.2 Benchmark

4.2.1 Criteria

Let's start by giving a very minimalist definition of our benchmarks objectives : compare different gesture recognizers to know how they perform in different contexts. Here is a list of criteria we will take into account in our definition of what is a "good" recognizer for us :

- It should be accurate : it must recognize correctly input gestures it has never seen before. Aim : the highest possible recognition rate.
- It should be fast : you can't afford to wait forever for a gesture recognizer to perform a classification. Aim : the lowest possible recognition time.
- It should be polyvalent : we want a recognizer to have the least possible variation in terms of both recognition rate and time when we change the context.

4.2.2 Parameters

To assess the performance of gesture recognizers based on the previously defined criteria, we can now define a set of parameters that we should try and vary in order to correctly test our recognizers.

- How does the recognizer reacts when we add or remove one or multiple gesture *class(es)* from the gesture set ? This implies that we should look at the recognition rate per gesture class.

- How does the recognizer reacts when we modify the number of samples per gesture class ? This implies that we should test our recognizers with different parameters and data during the training phase.
- How does the recognizer reacts when we modify the number of sampling points of its input (i.e, when the input data becomes more or less complex).
- How does the recognizer reacts when we change the origin of its input samples and test samples : our four defined scenarios (3.5.1).

4.2.3 Objectives

What is the purpose of a benchmark? Following Camp's definition [5], a benchmark is "the continuous process of measuring products, services and practices against the toughest competitors..." In general, the purpose of a benchmark is to analyze a process, by measuring a set of parameters that should reflect its performance. By analyzing those parameters, we should then be able to determine possible actions to improve the performance of our process. For our field of interest, stroke gesture recognition, this benchmark definition corresponds to the fact that we want, for a set of candidates recognizers, to know how well they perform, according to the aforementioned criteria.

4.2.4 Procedure : a 12 Steps Approach

The benchmark procedure that we will explain in this section follows the 12 stages methodology defined by Robert Camp [5] for benchmarks. Since this methodology was initially defined for processes in management, we need to redefine the steps here to make this methodology suits our problem.

Step 1 : selecting subjects. This is the very first step of our benchmark : selecting our subjects. Here, it corresponds to the different gesture recognizers we selected and integrated in Gester. Remember, our focus is on simple and portable algorithms, which is why our subjects, here, are mostly based on pattern recognition (particularly, *1-NN*). Our subjects are: $1, $N, $N-Protractor, $C, $P, $P+, $Q, FTL, NFTL, Penny Pincher, Rubine and JackKnife. All technical informations about our subject can be found in the chapter two of this document (see, 2.3.5).

Step 2 : define the process. Our "process" here, is the recognition of 2D stroke gestures. More precisely, it is the evaluation, in terms of recognition time and recognition rate, of different gesture recognizers under different contexts. The process is thus concretely defined by both the selected candidates and the context of use. Remember that a context, in our case, is always defined by the three different aspects, as mentioned in the introduction chapter : users, platforms, environment. Our applications should thus support context changes from the user. Any change in one of these three aspects inevitably means a change in the context. This phase is also the moment to make assumptions and hypothesis on expected results, with the help of gesture recognizers' literature or maybe prior knowledge gathered through other tests and benchmarks.

Step 3 : identify potential partners. In our process, this step correspond to result comparison between relevant candidates. If you refer to 2.7, you can see that some of our subjects are "extension" of other subjects. If it is true that a comparison between all recognizers is interesting, it is even more interesting to see to what extent successors are better or worse than their predecessors : are they always better ? Analyzing and comparing performances of algorithms having such a "predecessor / successor" relation is more interesting. For example, analyzing results by comparing such related algorithms could open new research questions, or simply verify assumptions and conclusions made by other researchers ? Note that, in 2.7, our relation between recognizers is just a suggestion. By no mean we do wish to force anyone to restraint researches by limiting observations and tests to recognizers being related with respect to the relation defined in our figure.

Step 4 : identify data source. This means that we need to gather input data to actually test our recognizers. In our case, the only relevant data source are the gesture samples. More precisely, the user they belong to, their origin platform, their class and stroke path(s) (ordered set(s) of 2D coordinates modeling the gesture of a user): this is our modeling of a gesture and the only relevant data source we need to run our benchmark. We then need to gather gesture samples of one or several users collected on the same platform and group them together : there is our gesture set.

Step 5 : collect data and select all partners. This phase refers to the data you are actually going to use to evaluate your selected subjects. Collecting data, here, refers to the results of your tests. That is the step where our application, Gester, is crucial. The "data" here, is the results produced thanks to the data source identified and gathered during previous phase. Selecting partners correspond to selected candidates from our recognizer pool defined in step 1, following (or not) the relation defined in step 3.

Step 6 : determine the gap. In our process, this phase correspond to the analysis of actual differences in the obtained result gathered in step 5. Concretely, we have to confront results of the test(s) obtained for each one of our selected recognizers, confront those results with our hypothesis and assumptions made in step 2, then repeat this process for all selected recognizers. At this point, we can already state the following facts: for each test result, is the result consistent with our hypothesis and assumptions from phase 2? This is the first step of analysis.

Step 7 : establish process differences. A process difference means that you have to find a possible reason explaining why you have differences with respect to your hypothesis in your results and how you can explain them. In our case, it means that we have to take the facts that we stated during the previous phase (is the test's results consistent with our hypothesis), conclude if our hypothesis is acceptable or if it needs refinement. This is the second step of analysis, where the main focus is results' justification.

Step 8 : target future performance. Can we get better results than those obtained? How? These are the main concerns of this phase. We have to setup goals in terms of performance we want our recognizers to achieve. What do we have, at this point ? We have initial hypothesis, an experimental setup, tests results and conclusion with respects to our hypothesis. Now, the next thing we have to do, is to target future performance

: can we get better results from our evaluated recognizers ? This means, changing the contexts, the parameters, input data, setup of our testing environment to see if we can achieve better results in terms of recognition rate and time. This is, of course, not always possible to have better performance. In that case, we should at least make sure that our conclusion and results are sound.

Step 9 : communicate. Talk about the results you got with your peers. Communication is showing obtained results, given a specific context. The objective of communication (again, in our case) is to give the possibility to our peers to reproduce and/or replicate our tests to see if they came to the same conclusion(s) or no. With an application like GESTER, this is possible.

Step 10 : adjust goals. Now that we communicated about our results, we have to refine our goals. Is the "future performance" we defined in step 8 plausible ? Does it make sense to refine our tests parameters to obtain better results if those results are not consistent with results you will have in a real life situation ?

Step 11 : implement. Depending on our conclusions gathered during previous steps, we may or may not be able to define potential prospects for improvement in terms of performances. If our conclusions gave the possibility to detect such a prospect, we have to implement it in our process. A benchmark can potentially reveal that, if some specific changes are made in our contexts of use, we can have better results that still represent a real life situation. But those changes might not be currently possible to make in our application. This is also why, one of the main focus of Gester is its maintainability and flexibility : to make it possible to add new features to further increase the range of our contexts.

Step 12 : review and recalibrate. Concretely, it is the "feedback loop" stage. During this final step, we analyze what has been done during the eleven previous steps. More precisely, we have to determine what kind of results we got, what kind of conclusion or new hypothesis are implied from these results. Then, and only then, shall we refine our tests and restart the whole benchmark procedure. This last step shows that the suggested procedure is rather continuous and iterative than punctual.

4.2.5 Summary

We gave, in the first section of this chapter, a suggestion of a benchmark methodology to apply in the area of 2D stroke gesture recognition. We want to insist that, this methodology can be adapted (the same way we did, since this methodology originally comes from business processes analysis) and that there may be other methodologies. We chose to suggest this methodology since it is a concrete application of Deming's quality management concepts [11] (Plan Do Check Act: *PDCA*). Gester, our application, was developed in hope that it could help its users to perform benchmark with as much flexibility as possible by intervening in different steps of the suggested methodology.

4.3 Examples of Tests

In this section, we are now going to give a demonstration of concrete examples of tests executed using Gester : the test configurations shown as examples in the following sections were all processed by our application. All tests results presented in this section were gathered using Gester. However, some of the graphs you will see come from ML-ModelExplorer [1], an application showing different visuals only based on confusion matrices. Since our application has a feature to export the results data (including thus our confusion matrices) in CSV, we can easily export results obtained from Gester and import them in ML-ModelExplorer. We will consider three different test cases in this section. The first test will be the study of a single recognizer in different contexts of use, with the same gesture set. We will analyze how the selected recognizer behaves when we change its number of training templates per class and how it behaves when we change the scenario. The second one will be the comparison of two different recognizers, both tested with the same parameters, the same context, the same gesture set. The last one will be the study of the impact of the platform on the recognition rate, for a specific recognizer. For this last test, we will thus use two gesture sets containing gestures from the same classes, the same participants, but gathered from two different platforms.

4.3.1 Test Case 1 : $P in Different Scenarios

Test setup. We will test $P (see 2.4.2) with the following parameters :

- Number of training samples per gesture class : ranging from 1 to 3.
- A fixed number of sampling points per gesture : 16
- A set of typical gesture classes : 16 different classes, with a total of 1920 gesture samples [3].
- Two different scenarios : user dependent, user independent (see 3.5.1).

The gesture set is made of 24 distinct participants, each having a total of 80 samples, 5 samples of each of the 16 evaluated gesture classes. The results, as we already stated in the introduction of this document, will be expressed in percent for the recognition rate and in milliseconds for the recognition time. All gesture samples of all participants were selected as potential candidates for the testing procedure, no exception. Since all used gesture samples come from the same gesture set, this test is platform dependent. The total number of trials made to obtain these results is 2 (the number of tested scenarios) times 3 (the number of distinct values for the number of training samples per class) times 24 (the number of participants from our input gesture set) times 16 (the number of gesture classes from our gesture set) times 100 (the number of repetition of the whole procedure), which is equal to a total of **230 400** trials.

Test results. The recognizer performs better, in terms of recognition rate, in the user dependent scenario than in the user independent scenario. We suppose thus that there are different possible geometric interpretations for gestures who belong to the same class and that these geometric interpretations are linked, one way or another, to the participant who made the gestures. What we can also observe, is that the delta in the recognition

[1] https://ml-and-vis.shinyapps.io/ML-ModelExplorer/_w_a29af428/ (last consulted 3rd of June, 2020)

rate of both scenarios decreases as the number of training templates per class increases. From those results, we can formulate the new hypothesis : the geometric difference between gestures from the same class tend to diminish as the number of training templates grows. From the point of view of the recognition time however, it seems that there is no "significant" difference between the two scenarios. The table and charts below shows a summary of the results, scenario versus scenario, for the different values of the number of training templates per class.

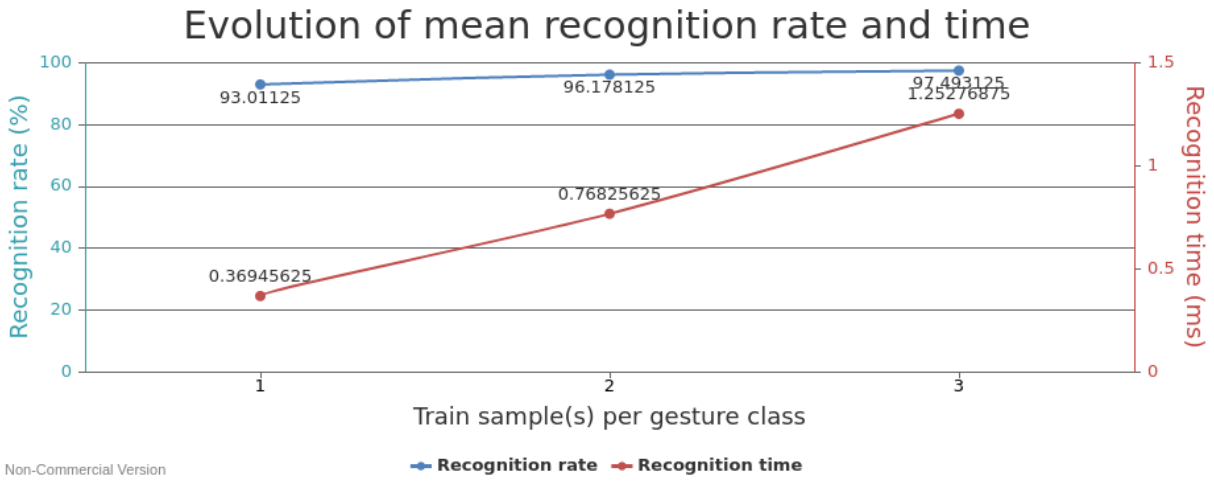


Figure 4.1: Spline chart showing the evolution of mean recognition rate and time for $P, user dependent scenario

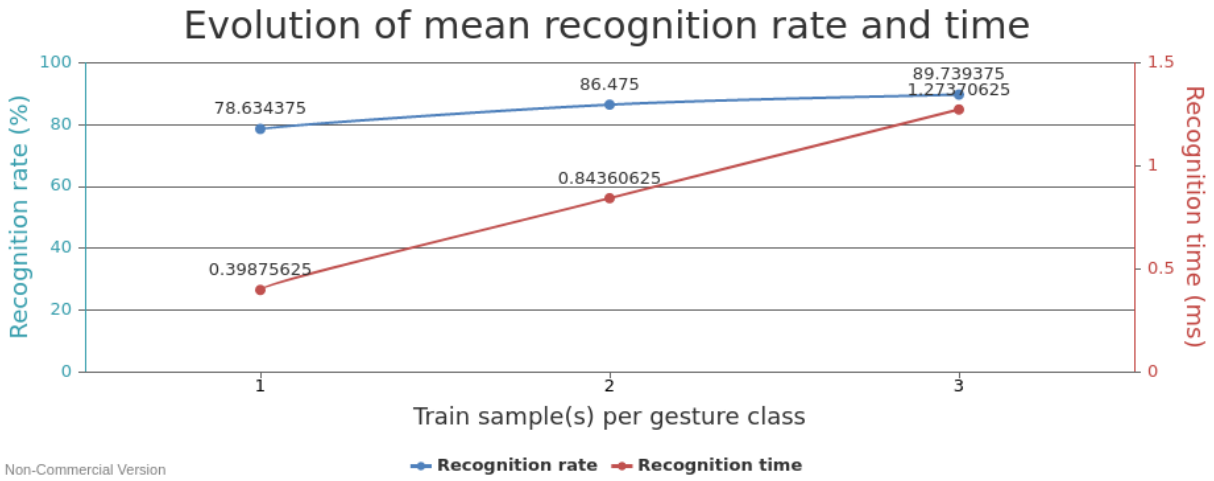


Figure 4.2: Spline chart showing the evolution of mean recognition rate and time for $P, user independent scenario

Scenarios	**UD**			**UI**		
# of template(s) per class	1	2	3	1	2	3
Mean recognition rate (%)	93.01	96.18	97.49	78.63	86.48	89.74
Mean recognition time (ms)	0.37	0.77	1.25	0.4	0.84	1.27

The previous table also shows that there is much likely a directly proportional relation between the recognition rate and the number of training samples per gesture **class**. There

are in fact, much less miss classification within the different gesture class when the model gets more template of each distinct gesture during the training phase. The previous table also shows, unfortunately, that this directly proportional relation also holds for the recognition time. We will use the charts and matrices shown below to analyze the impact of the number of training samples per class (in the user dependent scenario) on the recognition rate per gesture class.

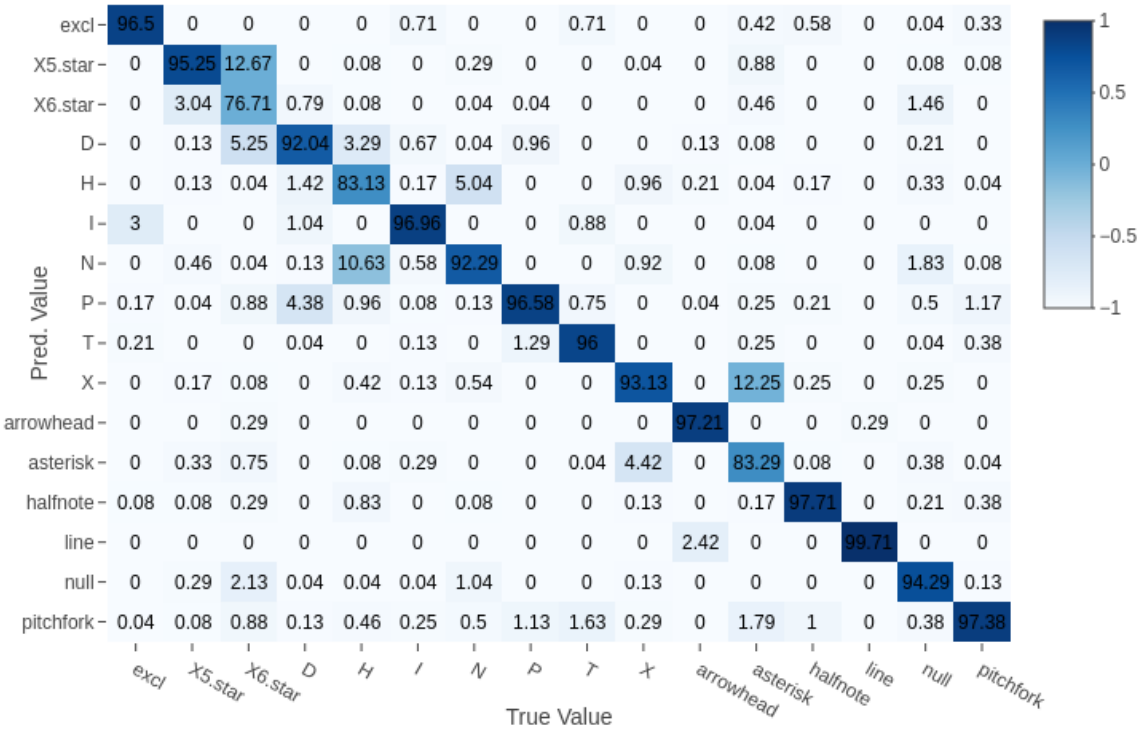


Figure 4.3: Confusion matrix, $P, user dependent test with one template per gesture class

We clearly see that, for our tested recognizer, three gesture classes are much more problematic than the others : the “six branch star”, often confused by a five star branch and the letter D. The letter “H” often confused by the letter “N”. The “asterisk”, often confused with the letter “X”. We can also observe that the miss classification between gesture is not always reciprocal. For example, in 4.3, the “six branch star” is often confused with the letter “D”, but much lesser the other way around.

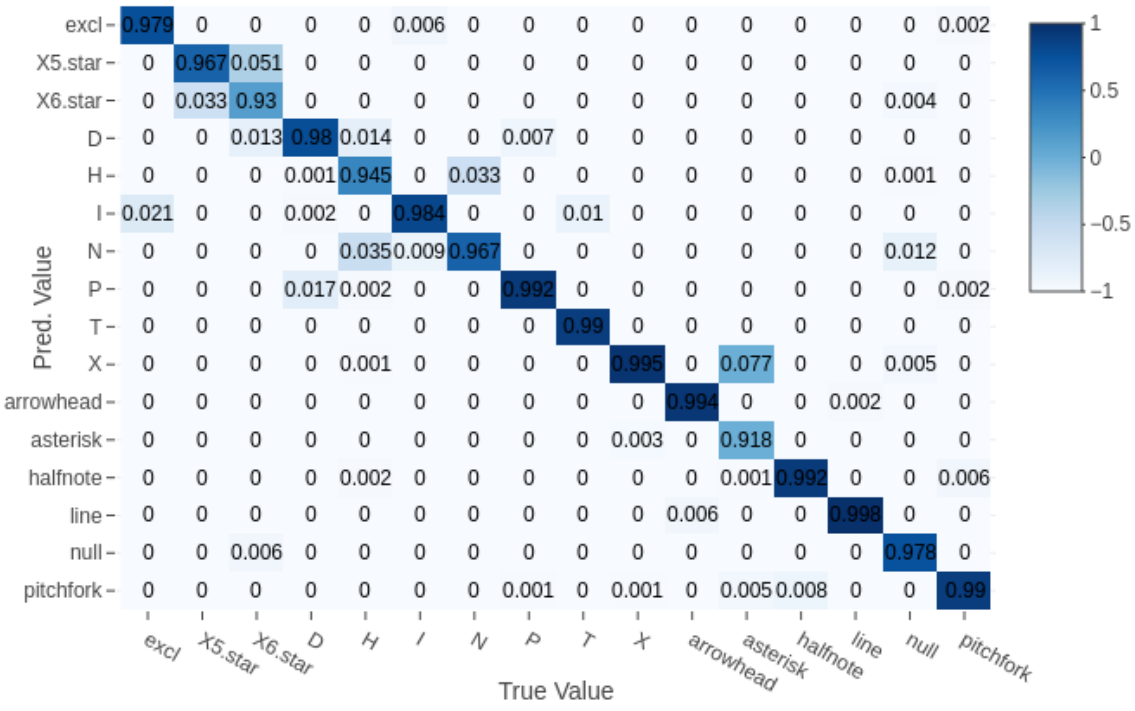
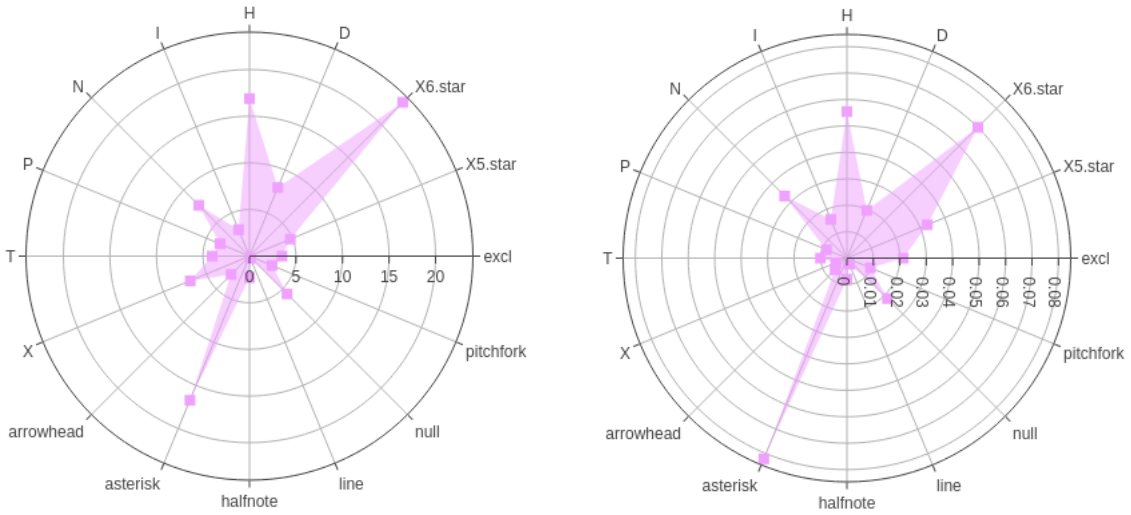


Figure 4.4: Confusion matrix, $P, user dependent test with three templates per gesture class

We clearly see that there is a lot less miss classifications when the number of training templates is greater, but also that the recognizer "confuses" gesture much less : there are less miss classifications, and less distinct classes among the remaining miss classifications !



(a) One template per class

(b) Three templates per class

Figure 4.5: Radar error charts for $P, user dependent scenario

4.3.2 Test Case 2 : Comparing $N-Protractor and Penny Pincher

Test setup. We will now confront two different recognizers under the same context of use. We will test both of them with the following parameters :

- Number of training samples per gesture class : 1 and 2.
- A fixed number of sampling points per gesture class : 16.
- The same set of gesture classes as the one used for our previous test case.
- One single scenario : user dependent, platform dependent.

The gesture set and the gesture set configuration are both exactly the same than those of our previous test case. The test here is just an example, to show what kind of test can be done using our application. However, our choice to compare $N-protractor and Penny Pincher (see 2.4.2 and **??**) is quite relevant. We choose these two, because they serve the same purpose (classifying stroke gestures) using the same strategy (nearest neighbour classification) but with different comparison technique (Penny Pincher compares distance of vectors between points, $N-protractor compares points between points distance). The total number of trials for this test is 2 (the number of testes algorithms) times 2 (the number of different values for the number of training templates per class) times 24 (the number of participants in our gesture set) times 16 (the number of classes in our gesture set) times 100 (the number of time the whole test procedure is repeated), which is equal to **153 600** trials.

Test results. The table below summarizes the results, in terms of recognition rate and time, of the test :

Algorithm	**$N-Protractor**		**Penny Pincher**	
# of template(s) per class	1	2	1	2
Mean recognition rate (%)	91.5	95.94	89.51	94.37
Mean recognition time (ms)	0.34	0.73	0.012	0.016

We see that, for this test configuration, in this specific context of use, with our gesture set, $N-protractor is superior to Penny Pincher in terms of recognition rate. It is, however, far much slower (up to **45** times) than Penny Pincher to classify gestures (fig. 4.6).

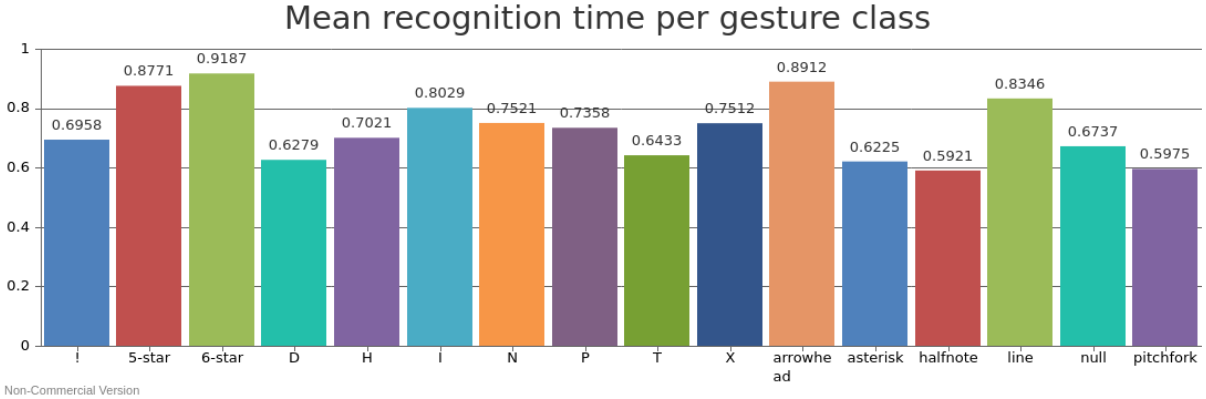


Figure 4.6: Mean recognition time for $N-protractor, two training templates per class

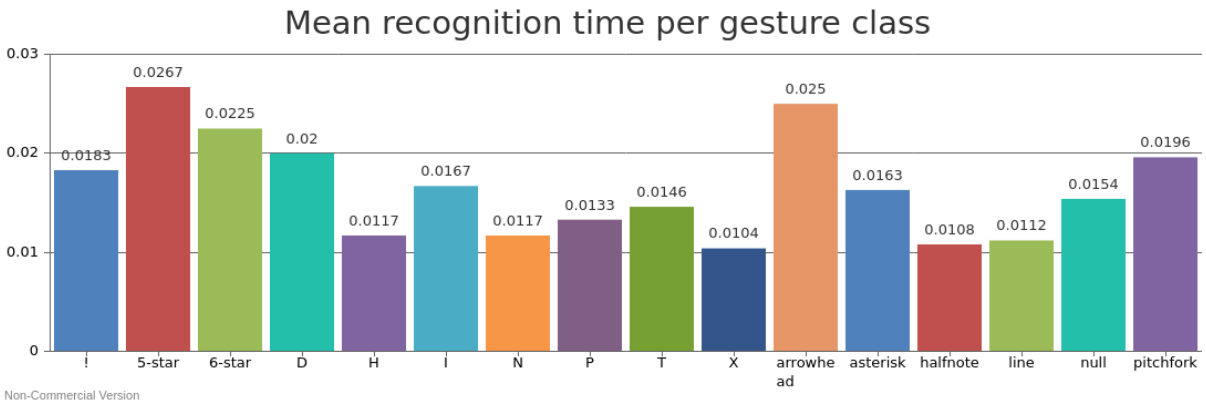


Figure 4.7: Mean recognition time for Penny Pincher, two training templates per class

Looking at the box plot below, we see that $N-protractor has another advantage compared to its opponent : it has less dispersion in its results. We can, however, note that Penny Pincher gap closes $N-protractor : the difference between the two algorithms is proportionally lesser when we use two templates per class instead of one.

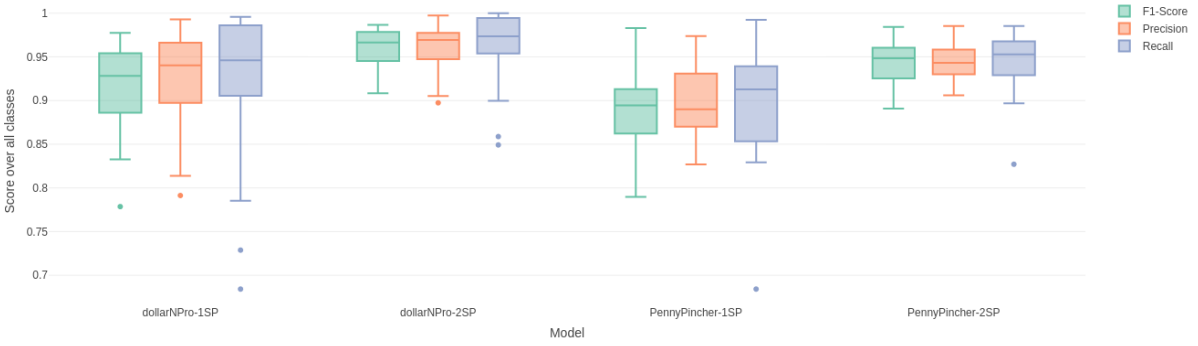


Figure 4.8: Box plot showing F1-score, precision and recall of $N-protractor and Penny Pincher, with one and two templates per class

The delta confusion matrix below contains, in each cell, the difference between the confusion matrix of a reference model (here, it is $N-protractor) and a comparison model (here, Penny Pincher). Cells where the comparison model performs better are indicated with shades of greens, whereas cells with shades of red indicate that the basis model performs better. For example, if you look at the top-left cell (“excl”/“excl”), it means that the basis model is 2% less accurate than the compared model for the gesture class “exclamation mark”. The compared model is thus superior to the basis model when it comes to the recognition rate of the “exclamation mark” : the cell is colored in green. If we look at the diagonal, we see that for 9 gesture classes out of 16, $N-protractor performs better than its opponent, not really an overwhelming victory for $N-protractor... However, what is more interesting is to look at the cells outside the diagonal : there are, overall, much more red cells than green cells. What it means in our case, is that our basis model has much less dispersion in its classifications. Overall, the basis model is better.

Figure 4.9: Delta confusion matrix of $N-protractor (reference model) and Penny Pincher (compared model), one template per gesture class

The second delta confusion matrix below helps us to highlight an advantage of Penny Pincher : it seems like a “fast learner”. The number of red cells has significantly decreased, and for almost every remaining red cell, the delta is smaller compared to the previous matrix. What it means in our case, is that, with 2 templates per class instead of one, Penny Pincher improves itself better compared to its opponent. It is still, overall, less accurate.

Figure 4.10: Delta confusion matrix of $N-protractor (reference model) and Penny Pincher (compared model), two templates per gesture class

Our previous supposition about Penny Pincher is reinforced by the radar plot presented below. This plot gives, for each model and for each gesture class, the miss rate expressed on a scale from 0 to 0.3, where 0.3 means 30% of miss rate. We see, in fact, that Penny Pincher has a lot of distinct classes with a big miss rate, compared to $N-protractor. We

also see that these miss rates drastically decrease when we give two training templates per class to Penny Pincher, instead of one.

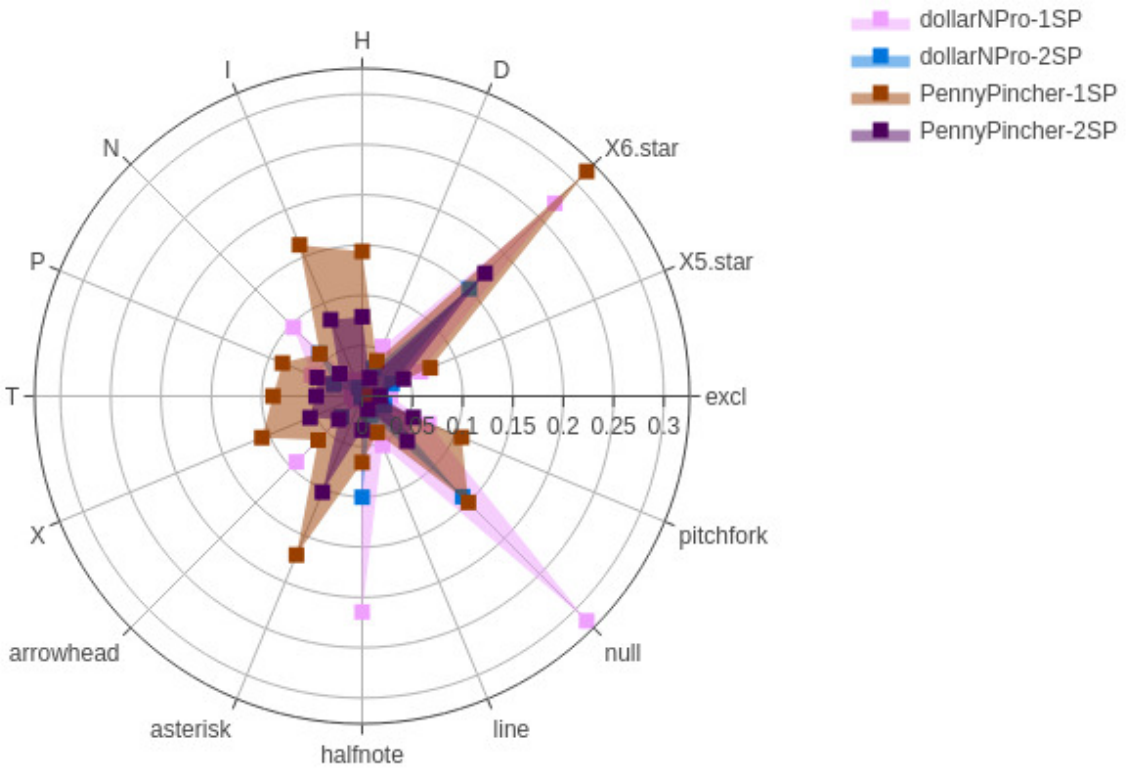


Figure 4.11: Radar plot showing error rate per gesture class

Another useful information we get from this plot is the set of classes that are "problematic" for our models. We see, for example, that the "six branches star" is often miss classified by both recognizers, even with two templates per gesture class used during the training phase. We also see that the "null" and "halfnote" characters are often miss classified by $N-protractor, but they are only a minor problem for Penny Pincher.

4.3.3 Test Case 3 : Platform (In)dependent Tests for $Q

Test setup. This test case will be performed using the following parameters :

- Number of training samples per gesture class : 1, 2 and 3.
- A fixed number of sampling points per gesture class : 16.
- The same set of 16 distinct gesture classes used than the two previous test cases.
- 2 different scenarios : platform dependent and independent.

We will use, here, two different gesture sets : the first one contains samples collected from a smartphone, the second one from a tablet. Each gesture set contains 1920 distinct samples, 80 per participant, 5 per gesture class per participant. The objective here, is to observe if, yes or no, there is an impact on the performance of $Q (see 2.4.2), in terms of recognition rate. The number of trials for this test is 2 (the number of scenarios) times 3 (the number of distinct values for the number of training samples per class) times 16 (the

number of gesture classes) times 24 (the number of participants) times 100 (the number of repetition of the whole tests : 100 times for the platform dependent scenarios, 50 times per gesture set for the platform independent scenario. Since the platform independent test will use both gesture sets in turn for training and test, it's actually 50 times 2, thus 100 too). The total is **230 400** trials.

Test results. The results of this test are summarized in the table below :

Scenarios	**Platform dep.**			**Platform indep.**		
# of template(s) per class	1	2	3	1	2	3
Mean recognition rate (%)	93.52	96.9	97.68	90.25	94.91	96.13
Mean recognition time (ms)	1.03	1.17	1.29	0.99	1.14	1.28

If we look at mean recognition time results, we can see that there is no significant difference between the two scenarios : we can't conclude anything about the impact of the platform on the recognition time. It seems, but it is just an hypothesis, that there is no impact. The results about the mean recognition rate are more interesting. There is a small difference between the two scenarios : the platform dependent always perform better than the platform independent. We also see that this difference is proportionally lesser when the number of training samples per gesture class increases. An interesting question could be : why is there such a difference, even if it is small? $Q resamples and translates every gesture it receives [20], so even if samples are of different size, which is something we could expect since samples come from two different platforms, there should be no difference. But there is a difference... We could make the following assumption : there is a geometric difference between gestures gathered from different platforms, even if they come from the same human being, even if they are of the same class.

Figure 4.12 shows that platform independent scenario has also a higher dispersion than the platform dependent scenario, regardless of the number of training samples per gesture class. This is yet another interesting track to explore if you wish to study the impact of platform in the process of stroke gesture recognition.

More detailed results are presented here in four different figures : two delta confusion matrices and two radar error charts, respectively showing the difference in terms of recognition rate between $Q in a platform dependent (basis model) scenario and $Q in a platform independent scenario (comparison model), with 1 and 3 samples per gesture class. We see in figure 4.13 that the basis model is overall better than the comparison model. We see, however, that the gap between the two models is smaller in figure 4.15. But still, the basis model is slightly better. Finally, the radar charts show what classes are the most problematic for our models : we see that, for both scenarios, the most problematic classes are "H", "X", "asterisk" and the "six branches star". Those radar plots show another way to display the results from our delta confusion matrices : they also highlight the fact that $Q is overall better in a platform dependent scenario than in a platform independent scenario.

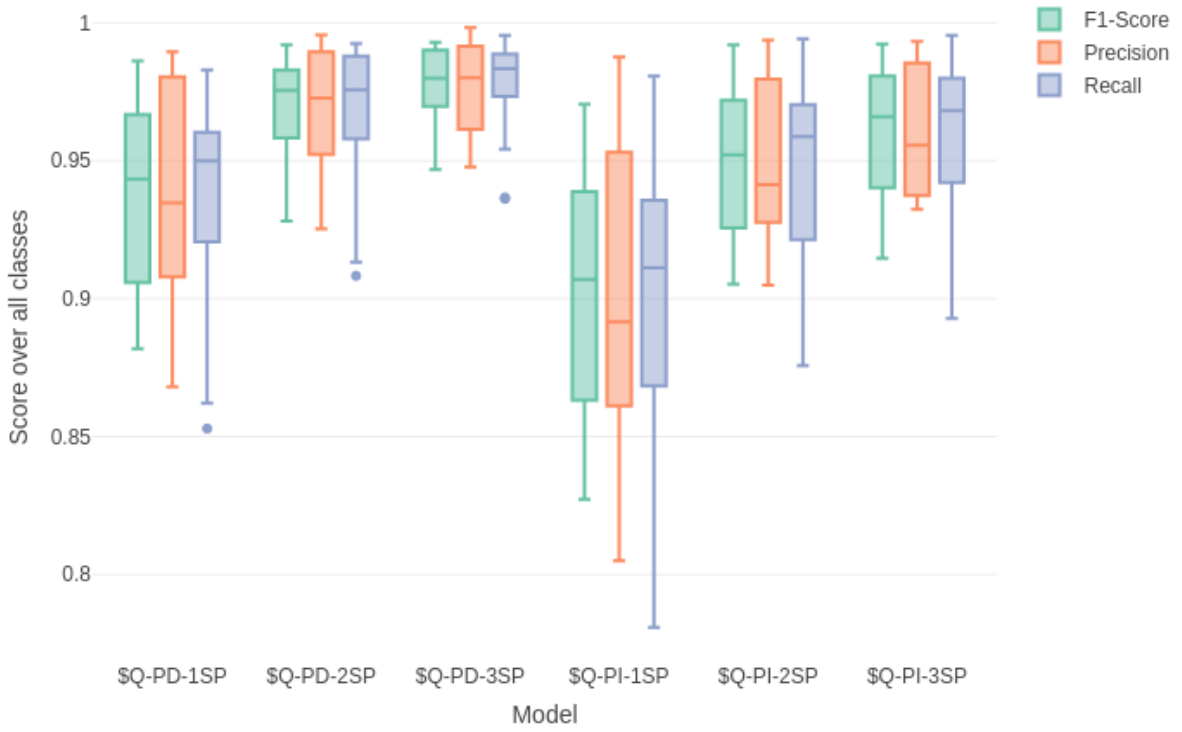


Figure 4.12: Box plot showing F1-score, precision and recall of $Q

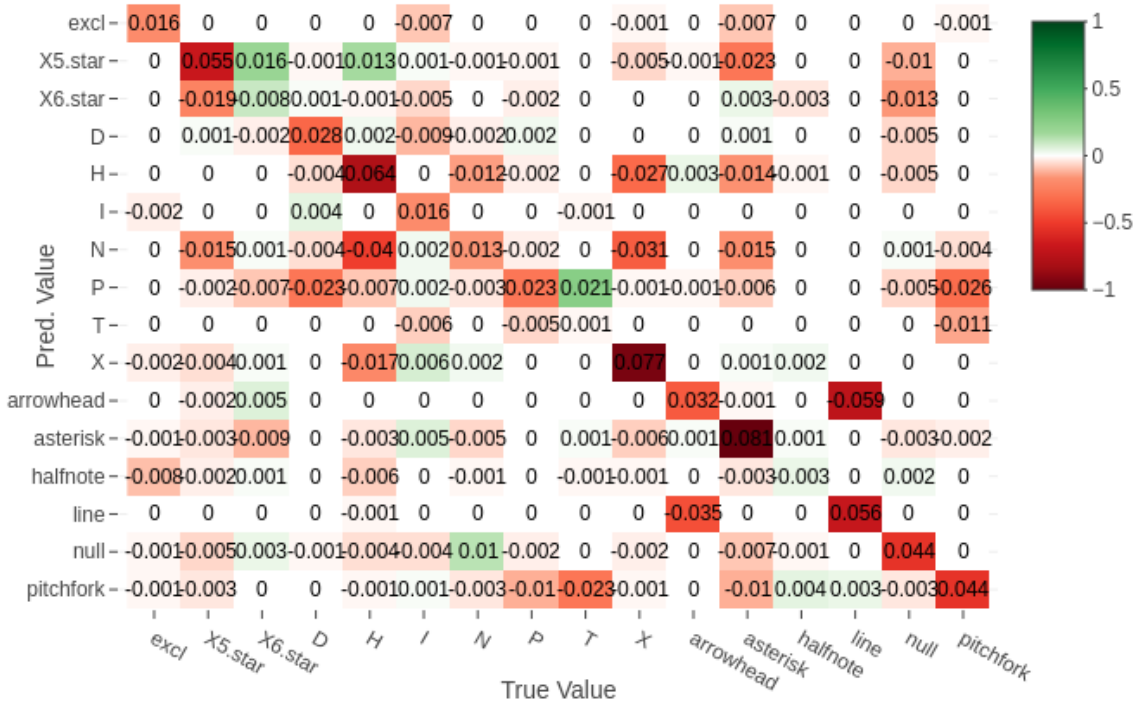


Figure 4.13: Delta confusion matrix of $Q in platform dependent scenario (basis) and platform independent scenario (comparison), 3 samples per class

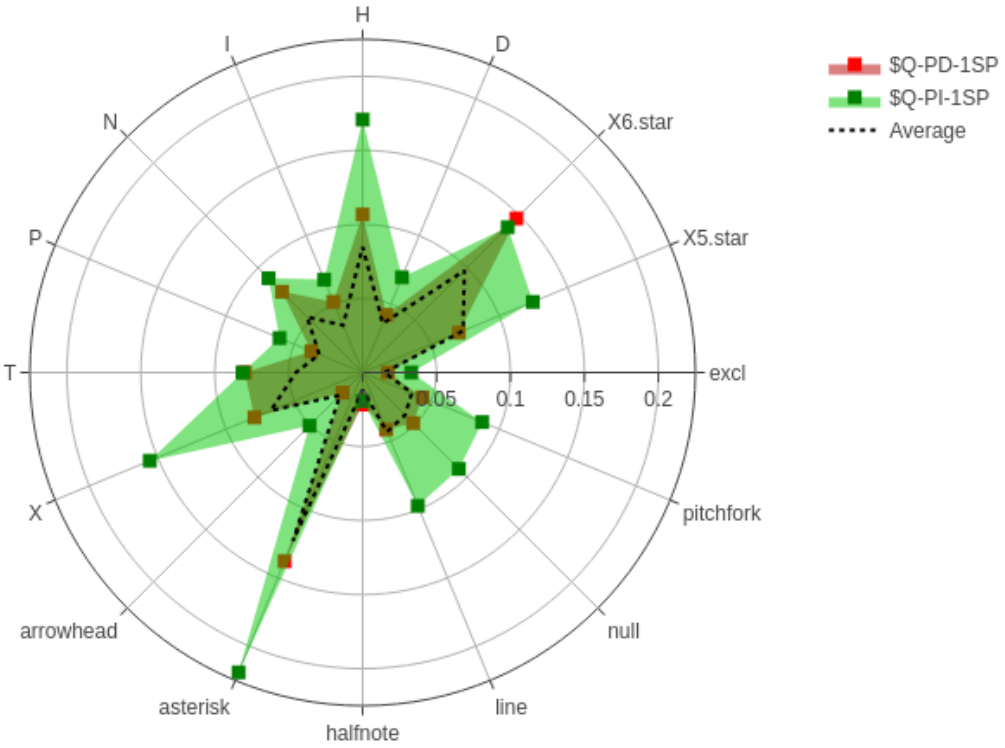


Figure 4.14: Radar plot showing error rate per gesture class for $Q, 1 training sample

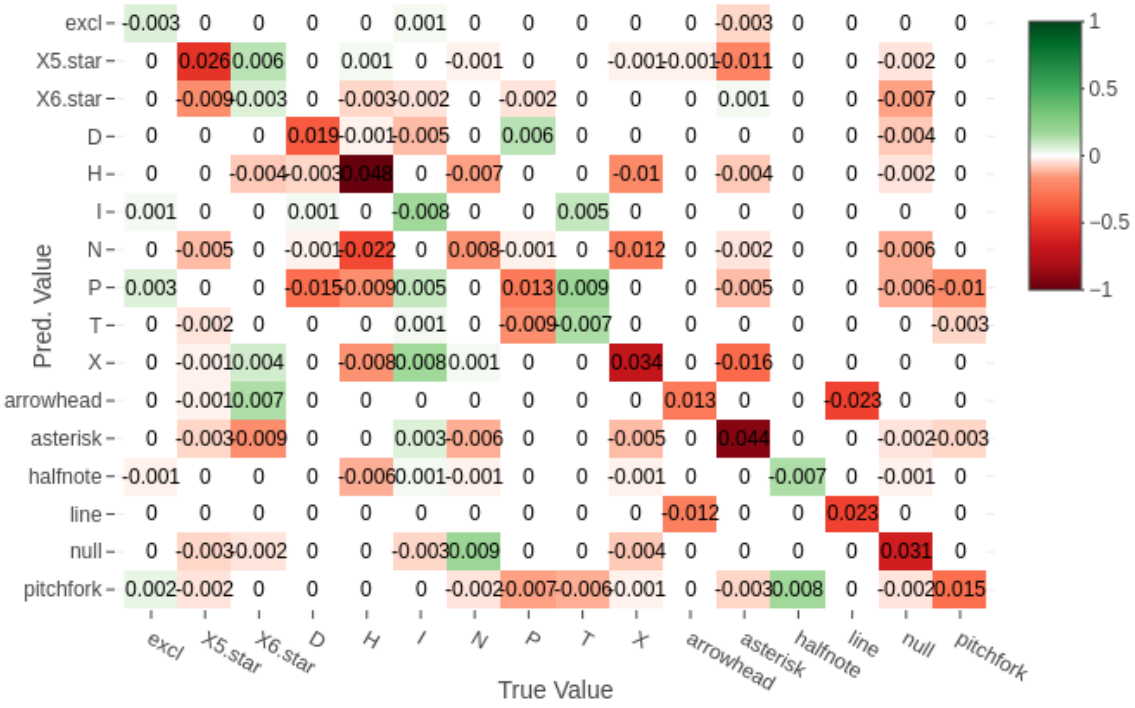


Figure 4.15: Delta confusion matrix of $Q in platform dependent scenario (basis) and platform independent scenario (comparison), 1 sample per class

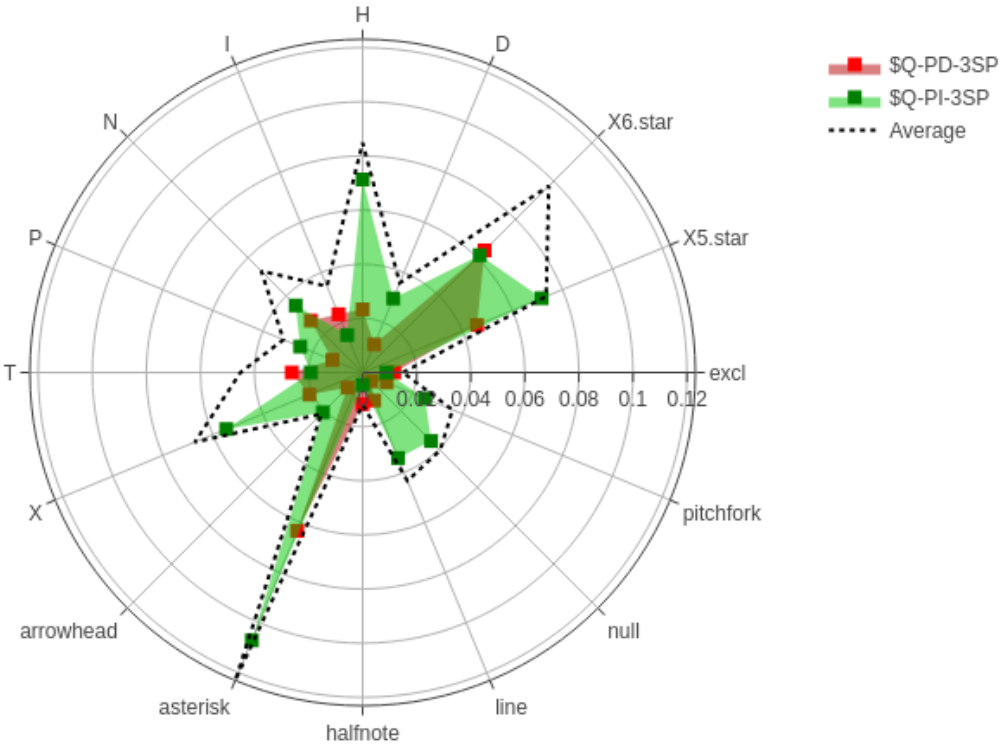


Figure 4.16: Radar plot showing error rate per gesture class for $Q, 3 training samples

Chapter 5

Documentation and Deeper Insights of Gester

5.1 Overview of this chapter

This chapter will be dedicated to the documentation of Gester. We will provide, here, two different kind of documentations : the first one will be purely user oriented : how to use the application, what are its constraints. The second one will be more technical and intended for developers : we will describe all client-server interactions, the interactions between React components and our Redux store. We will also have a look at the typical execution flow of the benchmark procedure integrated in the application. We will conclude this developer documentation by explaining how to integrate new scenarios, new *recognizers* and new *GUI* components.

5.2 User documentation

This section will be dedicated to the GUI of Gester. The purpose of this section is to show how the interface look like and how to use it. Screenshots of the interface in its current state can be found in the appendix C of this document.

5.2.1 Home Page

The Homepage is the first page where the user is brought when the application loads. He/she can upload his/her gesture sets, change the color theme of the interface. Depending on if the user is Logged in or not, he/she will have access to the login / sign up pages or to the result page, to see his/her previously saved results. To upload a gesture set, the user just has to click on the "configure" button and then click one of the two "upload" buttons C.1. Gester can handle two gesture sets. The first one is mandatory, the second one is only used if the user wishes to performs platform independent tests. In other words, if the user choose platform dependent tests, they will run on the first gesture set, but if the user choose platform independent tests, they will run on both gesture sets. For this reason, the first gesture set is the only one that is mandatory in order to proceed to the configuration. Once this mandatory (and the optional) gesture set(s) is/are uploaded, the user can proceed to the "benchmark page" to configure the tests, run them and see the results of his/her tests C.2.

The input format. The user should be aware of the expected input format that the gesture sets should have. A gesture set is a folder containing as many sub folders as there are distinct participants. Each sub folder represents thus a participant and all the gesture samples that belong to this participant should be grouped in the appropriate sub folder. All gesture samples should be in their own file and have the JSON format as shown in figure 5.1.

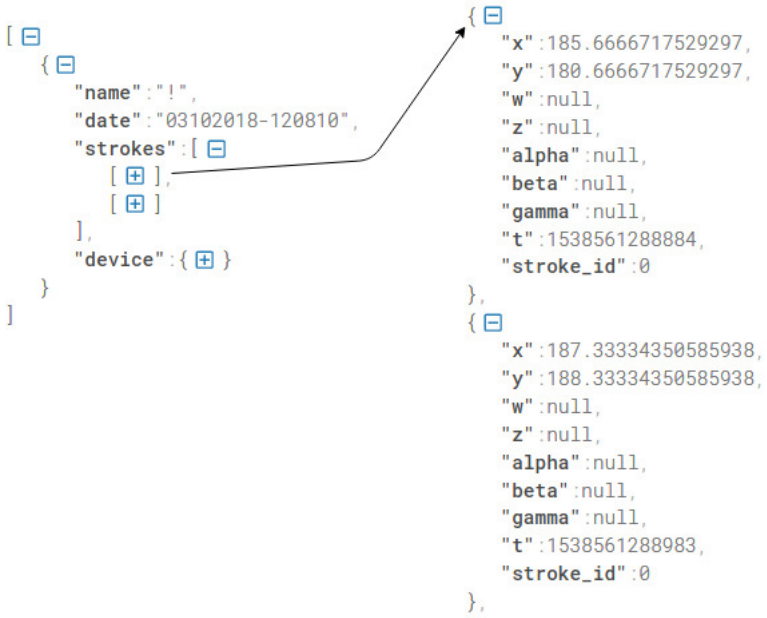


Figure 5.1: Example of a gesture sample

The important part of the sample, required by our application are : "name", a string that gives the *class* which the sample belongs to. A 2D array "strokes", whose sub-arrays each contains one or several object(s), each object representing a *sampling point*. A sampling point is expected to contain at least the following things : two coordinates, namely "x" and "y", "t" a timestamp, "stroke_id" the stroke to which this sampling point belongs to. Figure 5.2 depicts the expected shape of a gesture set.

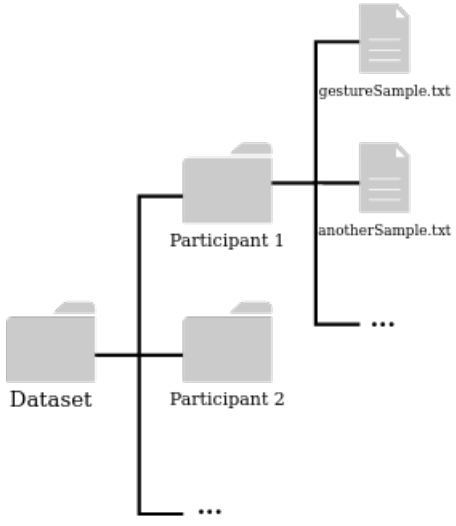


Figure 5.2: Expected format of a gesture set

5.2.2 Login and Sign up Pages

Those pages are pretty straightforward and both look exactly the same. Both have the same form, containing two fields : the first one for your email address, the second one for your password. The user should of course provide an email that is syntactically correct. The password should be at least six characters long. If the provided input is wrong, or something bad happened server side, a red banner with a message will appear above the form. If everything went well, the user should be redirected to the home page once logged or registered.

5.2.3 Benchmark Page

This page is divided in two parts : left pane and right pane. The left pane contains everything related to the tests and gesture set configurations while the right one contains everything related to the results obtained from the tests.

The left pane contains, top to bottom, the following elements :

- Test case selection (see C.3) : there are 3 different fields, one for each parameters the user has the possibility to customize. Those 3 parameters are the number of sampling points for the input data, the number of training templates per gesture class that will be used during the training phase of a recognizer and the number of repetition of the test procedure. These 3 fields come with validation, the user can't put any value he/she wishes. For example, you can't put a negative number as the number of repetitions... In addition to those 3 fields, there are two drop down menus : one to select the algorithm, one to select one the four built-in scenarios. Once the test configuration is complete, the user will be able to click the "Add test" button. Below this form, there is a table that summarize all the test configurations selected by the user. The user also has the possibility to delete a test configuration by clicking the "x" button, located on the right of each test configuration in the table.

- Gesture class selection : the first step in the gesture set(s) configuration is to select the different gesture classes, automatically extracted from the uploaded gesture set(s). Each gesture class has a check box that the user can check or uncheck. All gesture classes are selected by default. They are, by default, all checked. If the user uncheck one or several gesture classes, they will be ignored during the test procedure.

- Selecting gesture samples (see C.4) : the second step is to select all the gesture samples that can be used during the test(s) execution(s). Like gesture classes, each sample has a check box. They're also all selected by default. Gesture samples are displayed "per participant". To view the samples of another participant, change the currently displayed participant in the drop down menu above the samples' check boxes. On the right side of each check box, the user will see a "View" button. If he/she clicks it, the sample will be drawn and displayed in a canvas.

- Running the test (see C.5) : when both test(s) and gesture set(s) configurations are done, the user will be able to click the "Run Tests" button, the last element at the bottom of this left pane. However, if something is wrong in the user's configuration, a red alert box will displayed, containing detailed informations about what's wrong :

this is what we called the "consistency" model. The user will be able to click the "Run Tests" button if and only if there is no error reported by the consistency model.

The right pane contains, top to bottom, the following elements :

- Selecting the result to display : each result is associated to a tab. By clicking on a tab, the user will display all the result associated to the test. The summary of the test configuration is displayed just below the tabs to help the user remember which result correspond to which test.
- Spline chart (see C.6) : if the currently displayed test has different values for the parameter "number of training samples per class", a spline chart containing the evolution of mean recognition rate and time will be displayed.
- Confusion matrices (see C.7) : it contains in its row the expected (the true class) of the gesture, the actual (predicted) class in its column. Each cell displays thus a number, corresponding to the number of time the recognizer classified a gesture of the "expected" class as a gesture of the "actual" class. Note that there will be as many matrices as there are different values for the parameter "number of training samples per class". Each matrix can be displayed in percentage or in absolute value.
- Bar charts (see C.8) : those plots always come in pair. The first one displays the recognition rate, the second one the recognition time. The results of both bar charts can be displayed either "per gesture class" or "per participant".
- Save the results for later (see C.9) : the last thing in this right pane is a small form to save the results. After filling in the "name" field, the user should be able to click the "save" button. The results will be bound to the user's account. If something went wrong, a red alert message will be displayed. If the result were successfully saved, a green alert message will be displayed. In either case, the user will be warned about the outcome. There is also a red button beside the "save" button : the "wipe" button. This button is used to remove all results. The user should be careful : results that were not saved will be definitely lost when deleted.

If there is no result to display at all, either because no test has been executed or because the user has deleted all results, the right pane will contain nothing but a small information message, indicating that results will appear once the tests are done. All plots can be exported as images (either in PNG or JPEG), printed or exported as CSV. Confusion matrices can only be exported as CSV.

5.2.4 The Result Page

The sole purpose of this page is to give users the possibility to retrieve all their previously saved set of results. The users can select the result on the left menu, by clicking on the name they selected back when they saved them. When they click the name, the attached results will be displayed. The results are displayed exactly as they would in the right pane of the benchmark page. It means that the display on the result page has the same features : change the unit of the results and export them.

5.3 Developer documentation

5.3.1 Redux store

Every important informations our React components need to have are stored in Redux. Every test configuration, the gesture set configuration, the user token... Any major element needed by components to render properly are stored in Redux. Of course, that does not mean that we don't use React props and state (the "natural" React dataflow). Props are used when some specific interaction between parent and child component need to occur, or sometimes to pass down data from the Redux store to a child component [1]. States are still used for particular styling and rendering properties that do not depend on anything else than the component itself or if the component expect a return value from an external script (the results of a test, the response from the server...).

Concretely, the Redux store is composed of two things : reducers and actions. Reducers are object that describe the content of our store, actions are functions called when we need to change the value of what's in your reducers. Actions are exclusively fired from our React components, except one : the user login action. This action is re-fired every time the applications load if and only if there is a user token persistently stored on the client's device. We need to do this simply because the Redux store is not persistent : if the user press F5 to refresh the app, the Redux store will be entirely flushed. The different actions are defined in `src/types.js` and are used in functions defined in the folder `src/actions`. React components, in their turn, will import (if needed) functions defined in `src/actions` and call them. The second thing we need do define are reducers : we will now list them and detail their respective composition.

User. The "user" reducer is initially an empty object. It will contain the user web token of an authenticated user. This is, by far, our smallest reducer. You can, however, add any information you wish in this object if you want to further customize the GUI depending on who is connected (display the name of your user, for example).

Dataset. It is probably the heaviest one, in terms of size. This reducer holds the following data :

- dataset(2) : both are arrays, both serve the same purpose. They hold the raw samples uploaded by the user. Since Gester can use one or two gesture sets, we need two different object properties to differentiate them when they are stored.
- participants(2) : both are instances of JavaScript built-in `Map` data structure. They contain the set of distinct participants present in the associated gesture set and also informations about where are their respective samples located in the gesture set. Since our scenarios rely strongly on the origin of the samples, those `Map`s are crucial.
- displayed: is an object that contain all points of the sample our user wishes to display on screen. Its only purpose is to help displaying samples in canvas (see the feature defined in section 3.5.2).

[1] React props are much more efficient, in terms of execution time, compared to fetching data from the Redux store, this is why we sometimes pass down some piece of data from the Redux store as props

This reducer is most likely the one that will contain the biggest volume of data. We tested our application with gesture sets having sizes of more than 40Mb. It seems that it is not a problem for Redux to handle relatively quickly such volume of data.

Config. This is the reducer that holds all informations related to the user configuration : test(s) configuration(s) and gesture set(s) configuration. The configuration is composed of all the following elements :

- gestureClasses: an array containing all distinct gesture *class* that are going to be evaluated.
- SelectedSamplesDataset(1/2): both are arrays, defining a one to one relation between every sample of the gesture sets and the user decision to select the sample or not.
- nbSelectedSamples(1/2): total number of selected samples in gesture set 1 / 2. This number is here to improve the efficiency of our consistency model (see section 3.5.2).

Result. It is the reducer containing all informations related to the the results of tests ran by the user. It is made of two different arrays :

- results: the collection of all results data from all tests executed by the user.
- infos: the collection of all informations related to all tests executed by the user (its parameters).

Note that, for this last reducer, there is an index correspondence between elements from "results" and elements from "infos" (i.e, `results[0]` contains the results of the test configuration from `infos[0]`, etc...). So when the user clicks on another tab, all we have to do to change the results that are displayed is to change the index.

5.3.2 Client-server interactions

There are a couple of client-server interactions in Gester. In the end, from both client and server point of view, sending / handling a request is just equivalent to call a function. For the client, we use Axios, an open-source asynchronous HTTP client. We use functions defined by Axios to make our request to the server. For the server, we use Express routes, which maps a function for every type of supported request by the server. For conciseness and maintainability reasons, we decided to define all possible requests for the client in a single file called `api.js`, defined in the client's root source folder. This file contains two JSON objects : user and result, one for each of our mongoose schemas. Each function contained in each of these objects represents one query that is supported by the server. Each of the following requests are modeled by sequence diagrams in the appendix A of this document.

User registration. This request is always sent from our React component `<SignupPage />`. The client will send a POST request with user credentials in its payload. Credentials are made of two strings : an email address and a password. The server will then extract credentials from the payload and create a new instance of our `User` mongoose schema and finally save it in the database. There are two possible outcomes to this request. Either everything went as expected, and the user is redirected to the `<HomePage />` with a signed JWT token, or an error message is displayed if the email address is already bound to another user account. A.1

User authentication. This request is always sent from our React component `<LoginPage />` and is similar to user registration. It is also a POST request containing credentials in the payload. There are three different outcome to this request. The first possible outcome, is that everything went as expected : the provided credentials are correct. The email address is bound to an existing user and the provided password match the password hash stored in the database. The user will then, like in the registration process, be redirected to the home page of the `<HomePage />` with its JWT token. A second possibility is that the email address is correct (i.e, mongoose returned a `User` record from the database) but the provided password does not match. The last possibility is the email not being correct : the database returned no record. From the client's point of view, the second and third case are similar : the same message "invalid credentials" will be displayed to the user. A.2

Fetching saved results. This request is sent from `<ResultContainer />`, a child component of our `<ResultPage />`. The request is of type GET and should contain the authenticated user token in its payload. There are four possible outcomes to this query. The "everything went well" scenario is, the token is correctly verified and an email could be extracted from it, this email is attached to an existing user and this user has some records of `Result` linked to it. In that case, all the results will be returned to the user and he/she will then be able to choose which one to display. The two next possible outcomes mean that something went wrong. The first possible error is if the user token is wrong (happens if `jwt.verify()` fails). The second possible error is the fact that the user token is correctly formatted, but the email it contained is not bound to an existing user (this error should obviously never happen unless you manually remove users from the database, or maybe someone managed to falsify a token). In both cases, an error message will be reported to the user and the response will contain a 401 error code. The last possible outcome is that everything went well, but the user has no previously saved results. In that case, the database will return an empty record and the server will return a 404 error code, stating that no result were found A.3.

Save new results. This request is sent from `<SaveResultsForm />`, a child component of `<BenchmarkPage />`. This is a POST request, where an authenticated user sends all the data of his/her test results to retrieve them later (see the previous request). The "everything went well" scenario is the following: the user sends the request with the data to save, his/her JWT token, and a name in the payload. The token is then verified by the server and the email address of the user who sent the request is extracted from the token. This email and the data are then used to create a new `Result` (an instance of our mongoose schema). This instance is then saved in the database, and the server notifies the user that everything went as expected. A possible error that could occur is the fact that user wants to save his/her results with a name he/she already attributed to another set of saved results. We impose a uniqueness constraint on our users email address and the chosen name of the results. The two other possible errors are the same than the previously defined request : a token is not valid, or a token is valid but the extracted email is not bound to any existing user. A.4

5.3.3 Benchmark Execution

Once your gesture sets are uploaded, your configuration is done and consistent, you can click the button "Run tests" defined in `<RunTestsButton />`. What happens when you click

it? The `run` function from `src/benchmark/main.js` will be called. This function is the core entry point of the benchmark procedure. Here are the different important steps of the execution flow :

- The first step is to convert gesture sets, still in their raw form, into gesture samples readable by the selected recognizer. To fulfill this purpose, each gesture set **and** the configuration of gesture set will be passed as parameter to the `prepareDataset` function. This function will return a 4 dimension array. The first dimension is the whole "readable" gesture set. The second dimension is the set of all samples of one participant. The third dimension is the set of all samples that belong to the same class. The fourth dimension is the actual points of the samples : the actual data used by our recognizers.

- Once our gesture sets are ready, the second very important phase is a warm up phase. Since the benchmarks scripts are run client side, the code will be executed in your browser's JavaScript engine. Since **every** modern JavaScript engine (V8 for Chrome and NodeJS, Rhino for firefox...) uses *JIT* compilation, it is important to have a warm up phase to let compiler tag the functions as "hot". For your information, if you repeat the execution of a function a several time or if there is a pattern in the execution flow of your code, the JIT compiler will notice it and automatically optimize it. A function tagged as "hot" by the compiler will execute faster. You understand now why this warm up phase is crucial : without this phase, reported execution time will be false, because at the beginning of the execution, function are not tagged by the compiler and thus slower. Without a warm up, there will be a really big variance between reported execution time for gesture classification at the beginning of the process (slower) and at the end of the process (faster).

- The third step, starting once and only once the warm up is done, is the execution of the test. We gather our users test configuration, chose the correct scenario to execute (one of the four defined scenarios defined in 3.5.1) from the appropriate function defined in its own source file. We then execute the scenario. All our scenarios have (roughly) the same execution flow : instantiate a new recognizer, train it with randomly selected templates from all selected gesture classes, test it with one gesture sample for every selected gesture class. Repeat this process for as many times as the user decided to, for each different value for the number of training samples per class specified by the user.

- the fourth and final step, is to convert raw results gathered during the previous phase into formatted results. The raw results have the same "shape" than the gesture sets : a four dimension array. The three first dimensions are exactly the same. The only difference is the last one : instead of having a set of readable points representing our gesture samples, we have JSON objects containing the result of the classification function from the recognizer. Every recognizer returns a different data structure with different field names and properties who hold the results. To have a standard, we suggest that every result returned is transformed into a JSON object containing at least two fields : name, the attributed gesture class and time, the recognition time. We also suggest to keep this standard if the developer wishes to integrate new algorithms in the application (we will further discuss integration in the next section). All those "raw" results will be passed to the function `prepareResults`, which will

return all the different elements that will be displayed once the benchmark procedure is done.

The return value of the "run" function is, precisely, the following: the set of gesture classes evaluated in this test. Then, the set of distinct participants evaluated in this test. Third, confusion matrices, one for each different value of the number of training samples per class. Fourth, results per gesture class (both recognition rate and mean recognition time). Last, the results per participants (also, recognition rate and mean recognition time). For your information, all the above explanations about the benchmark procedure are synthesized in a flow chart (see appendix B).

5.3.4 Extending Gester

In this section, we will briefly describe how our current algorithms and different scenarios are integrated in the application. This will be very helpful for a developer who wants to add another currently unsupported algorithm or a new scenario in the application.

Add a new recognizer. Every algorithm is different from the others. This difference can be small or huge, whether in the way you train them, test them, instantiate them or all at once... We need thus one standard to rule them all. For extendability and evolution, it is important that the benchmark process is the same for all supported algorithms. For this reason, we built an abstraction that encapsulate a recognizer. This abstraction is defined in `src/benchmarks/recognizer.js`. It is nothing but a factory. It takes a mandatory string and an optional "tote-bag" object as parameters. The string should be an identifier to represent the algorithm that will be used behind the instance of `Recognizer`. A `Recognizer` has three methods: `train(points, gestureName)`, the method you call to train your recognizer. `recognize(sample)` is the method you call when you want to submit an unknown gesture sample for classification. The last one, `instantiate()` is the method that will decide which algorithm we need to instantiate. It depends thus on the mandatory string you passed as parameters when you called `new Result(...)`. To sum it up, our abstraction `Recognizer` serves as a standard, uniform API for your tests scenarios. Thanks to this abstraction, there are only a few things you have to do if you want to integrate a new algorithm in Gester :

- Choose a unique "token" to represent the new algorithm, and add a new choice in that your user will be able to select from the GUI. When the user choose this new algorithm, the token you chose needs to be passed to the `main.js` file in the benchmark scripts folder.
- Define a new method in `prepareDataset.js` that will return a readable gesture for your new algorithm.
- Extend the three existing method from `recognizer.js` to support your new algorithm.

Add a new scenario. If you want to add another scenario, it is even simpler. Here are the things we suggest you to do :

- The same way than if you add a new algorithm, choose a unique token that will identify your new scenario. Add the possibility for your user to select this new scenario, directly from the GUI.

- Add your scenario's script. It should use our `Recognizer` abstraction as interface between the scenario and supported algorithms. This way, your new scenario will directly be compatible with all supported algorithms.

- Import your scenario in `main.js`, map it to the token you chose so when the method "run" is called with your token you know that your scenario needs to be executed.

- optional : if your scenario has particular rules and you want Gester to automatically verify consistency between user's tests and user's gesture set configuration, you will have to define them in `consistency.js`, the script responsible for the consistency verification process.

Extend, modify the GUI. Adding new components or changing the current look of the interface should be easy, thanks to our choice of tools. React components are loosely coupled between each other. Modifying a component will much likely have no impact on other components, unless you modify the data flow between parent and children components. Even then, changing those components should not be difficult : we focused on keeping them clean (thanks to ESLint[2] and Prettier[3]), as short as possible, as readable as possible, as simple as possible. The only thing that may require more work is if the developer wants to modify the current reducers of our Redux store. We insist on the verb modify here, since adding new data to existing reducers will much likely be as easy as modifying our React components. However, since some piece of data from some reducers (the datasets, for example) are used by different components, the developer could expect changes to be a little harder if there are modifications in current piece of data from our current set of reducers. Yet, thanks to the declarative nature of React and the structured work flow of the redux store, no refactor or no change should be difficult.

5.3.5 Testing Gester

To ease further development, we included three different test suites in a separate folder "tests" in the client. These suites were build with Jest, the most common testing library for React applications. The three test suites cover different critical aspects of our application :

- The Redux store : the central point of data store for our client. It defines the shape of most components, since many of them adapt their behaviour depending on the content of the Redux store. This is why we absolutely wanted to provide a dedicated test suite. This test suite emulates our store, fires the different supported actions and inspect that everything is placed correctly in the store. This test suite is defined in `tests/redux.test.js`. Note that, if you modify existing reducers or add new reducers, you should write new tests and add them in this suite.

- The consistency model : it ensures that the test run with Gester are all consistent with their respective configuration. This ensures not only that the application is a minimum defensive, but also that results produced have a standard. For these reasons, we wanted to include a dedicated test suite for this feature : we create artificial test cases and gesture sets configurations (some are wrong, some are correct) and pass them to the verification mechanism. We then compare the output and

[2]https://eslint.org/ (last consulted June 2020)
[3]https://prettier.io/ (last consulted June 2020)

see if the verification mechanism has detected inconsistencies when there are some, but does not detect any inconsistency when everything is correct. This test suite is defined in `tests/consistency.test.js`

- The benchmark procedure : this test suite checks another simpler yet essential thing: does the actual benchmark procedure matches the user choice? In other words, this last test suite verifies that, if the user select some algorithms, with some scenarios and specific parameters, the test that is actually run is the one selected by the user. This test suite ensure that the correct procedure, with the correct parameters and correctly instantiated recognizers are used to run the test, according to what the user specified. This last test suite is defined in `tests/benchmark.test.js`

Chapter 6

Conclusion and future works

6.1 Review

What has been done in this thesis ? We started by formulating a problem in our introduction chapter : there are many different techniques to perform automated 2D stroke gesture recognition. Yet, to the best of our knowledge, there is no tool giving the possibility to evaluate them all. Convinced that benchmarking of stroke gesture *recognizers* has not yet been thoroughly performed, we built a software, Gester, described in chapter three and detailed in chapter five of this document. This software is able to test, under the same conditions, a set of relevant stroke gesture recognition algorithms, some considered as state-of-the-art, some considered as pioneers. We focused on providing a tool that is able to evaluate the algorithms under equal conditions, but that is also able to evaluate them with a variety of parameters and scenarios. In fact, every test made in the past were performed on specific constraints with specific gesture sets. Here, we offer the possibility not just to analyze the performances of stroke gesture recognizers, but also to analyze their behaviour when the use-case changes. We also suggested, in chapter four, a conceptual definition of a procedure to perform benchmarks in our field of interest, based on a famous procedure originally meant to evaluate business processes. We also showed examples of use of our application, by showing results obtained in three different test cases to show the usefulness of our tool: the first one, analyzing a recognizer in different contexts. The second one, analyzing two different recognizers in the same context of use. The third one, showing results when changing the platform of used samples. There are (much likely) still many things to do, since Gester is probably the first application to fulfill this specific purpose. Therefore, we were committed to offer a tool that is easy to extend, understand and maintain with an extensive documentation. We hope that our tool will be useful for researchers and that it will encourage future works and researches.

6.2 Critical analysis

In this section, we are going to identify a set of possible limits of the work presented in this document.

Threat to UX. JavaScript is, by nature, single threaded. Everything runs in the same thread. The benchmarking procedure can be really heavy if you work with large gesture sets, if you have a high number of repetitions of the test procedure or if the computational

complexity of the recognizer is high. Since, in every modern browser, the application has to share its thread with other processes (like the DOM rendering, the CSS interpreter...) if you have a very big function that takes all the computation resources, nothing else will be executed meanwhile : your application freezes. This is exactly what happens in GESTER if you run a heavy benchmark. This is absolutely not a problem with respect to the outcome of the benchmark, but it can be frustrating for our users not to be able to interact with the application when it's running heavy computations.

Hard constraints on the input data. As mentioned in chapter three and chapter five of this document, GESTER expects a very specific format of input data. The application can't be used if the user tries to upload a wrongly formatted gesture set.

Repeatability, replicability and reproducibility. Those three terms are artifacts suggested by the *ACM*. Those artifacts are important to fully establish experimental results. In other words, no result shall be considered as valid unless it is possible to reproduce it, according to the three aforementioned artifacts [1]. Fortunately, our application can be used to reproduce experiments made by others. But there is currently no feature that is specifically designed to ease the implementation of these artifacts.

6.3 Prospects for Improvement, Future Works

Threat to UX : what we tried. There is actually a simple and elegant solution to the problem identified in the previous section. It is possible to run code in a separated thread, by using a Web Worker [2]. Unfortunately, we want to measure the recognition time of a recognizer during the benchmark procedure. The problem is that the time measurement functions uses monotonic clocks [3]. In other words, if we run our benchmark in a separated thread, the reported recognition times will be wrong, since they will be equal to the wall clock time instead of the CPU time. The figure below represents this problem. There is thus, to the best of our knowledge, no simple solution to prevent the GUI from freezing that preserves the correctness of our application's reported results.

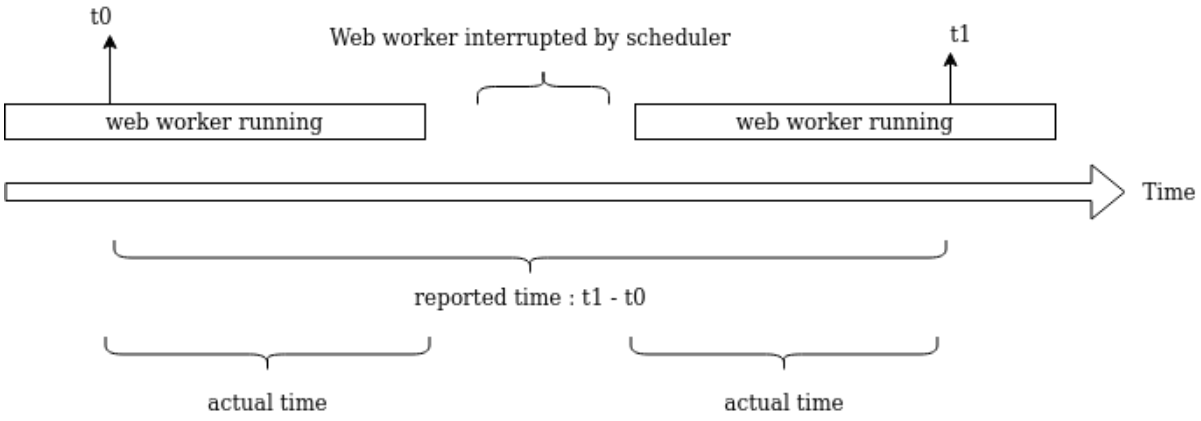


Figure 6.1: Reporting execution time in a multi threaded context in JavaScript

[1] https://www.acm.org/publications/policies/artifact-review-badging
[2] https://developer.mozilla.org/fr/docs/Web/API/Web_Workers_API
[3] https://www.w3.org/TR/hr-time-2/sec-monotonic-clock

Hard constraints on the input data : our suggestion. It is possible to "soften" this constraint. It is even easier than what one may imagine. The easiest solution is to modify the data directly when it is uploaded and then push it to the Redux store while preserving the current structure and content of the "dataset" reducer (see 5.3.1). As explained in the previous chapter of this document : modifying the current structure of the Redux store can be difficult, modifying / adding components is not. Therefore, we suggest the developer who wishes to modify the expected input data format not to touch the Redux store, but instead modify the way the raw data is handled before sending it to the Redux store.

Repeatability, replicability and reproducibility : a couple of ideas for future features. To ease the implementation of these artifacts, we should commit to provide a feature that eases the communication between teams of researchers. Here are a couple of feature ideas:

- Sharing results : a feature that makes it possible to share results. For example, we could imagine a form where the user has to put the email of another registered user. The other user will then be able to see the results shared with him/her. This could be actually fast and easy to implement since there is already a similar feature : users already have the possibility to save their results, they will be bound to their email address. Instead of binding results to one email address, we could, for example, bind them to "n" email addresses.

- Exporting / Importing test configurations : to ease the reproduction of specific tests case, there could be a feature that allows the user to export or import their configurations, to publicly display them, share them with other researchers...

Other possible features. There are also a couple of possible tracks that could be interesting to explore to perform benchmarks of stroke gesture recognizers. Since the benchmark procedure is a highly parellelizable process, can we make it much faster by running it on a GPU ? It could be interesting to give it a try, especially since GPUs tend to be more and more efficient and their popularity continuously rose these last few years. There are JavaScript libraries that makes it possible to run pure JS code on a GPU [4]. It is thus, in theory, possible to include such a feature in Gester. Another possible future work, is to study the feasibility of a mobile version of Gester. Even though the application, in its current state, has a responsive interface, it is surely not perfect and was not designed on a mobile device, nor for a mobile device. There is, however, a mobile version of React, whose focus is to provide mobile applications. This framework is called "React native" and it uses the same patterns and logic as React. An interesting work would be the evaluation of the implementation costs of such an application.

[4] https://gpu.rocks/#/

List of Figures

Appendix A

Sequence Diagrams

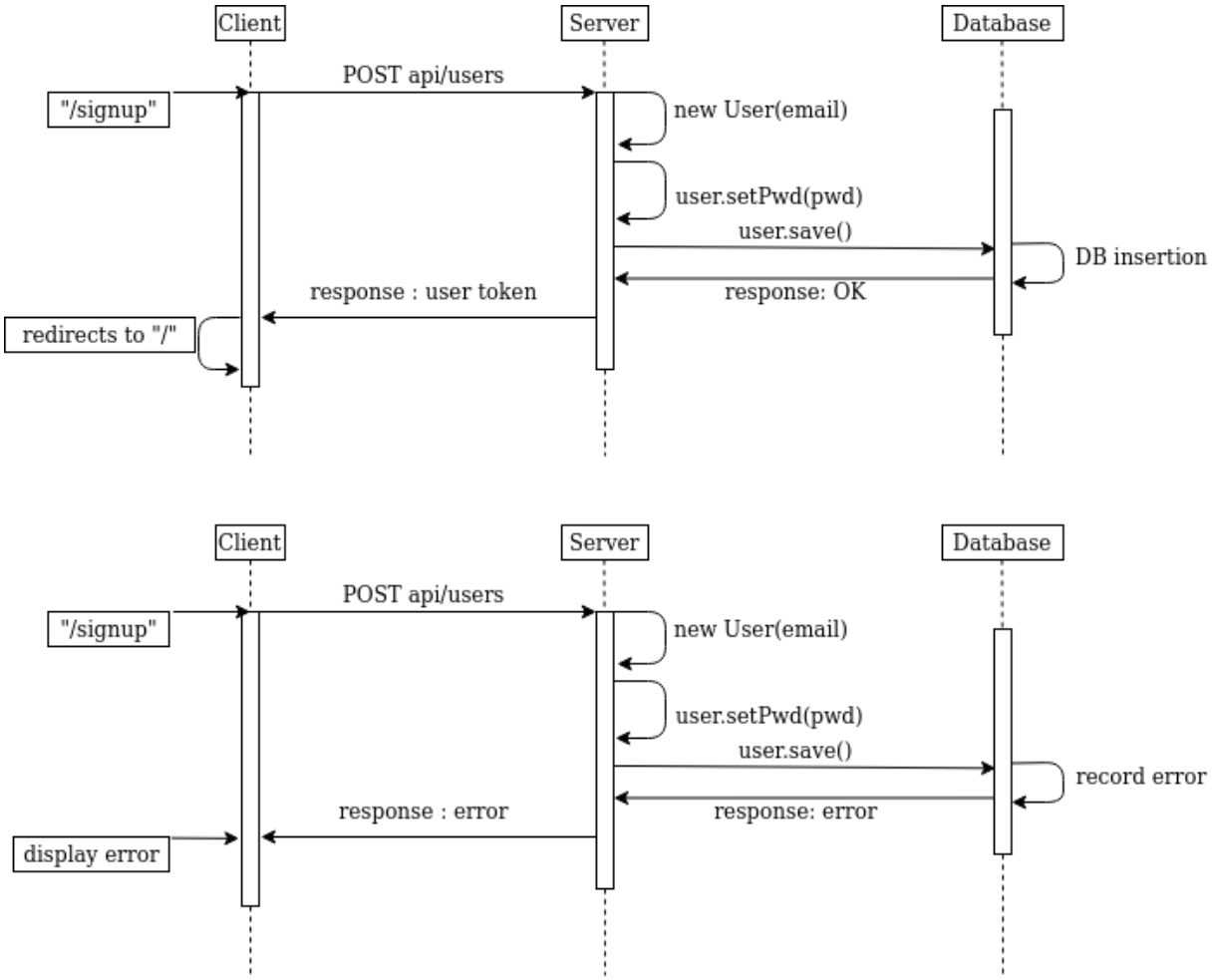


Figure A.1: Sequence diagrams for user registration

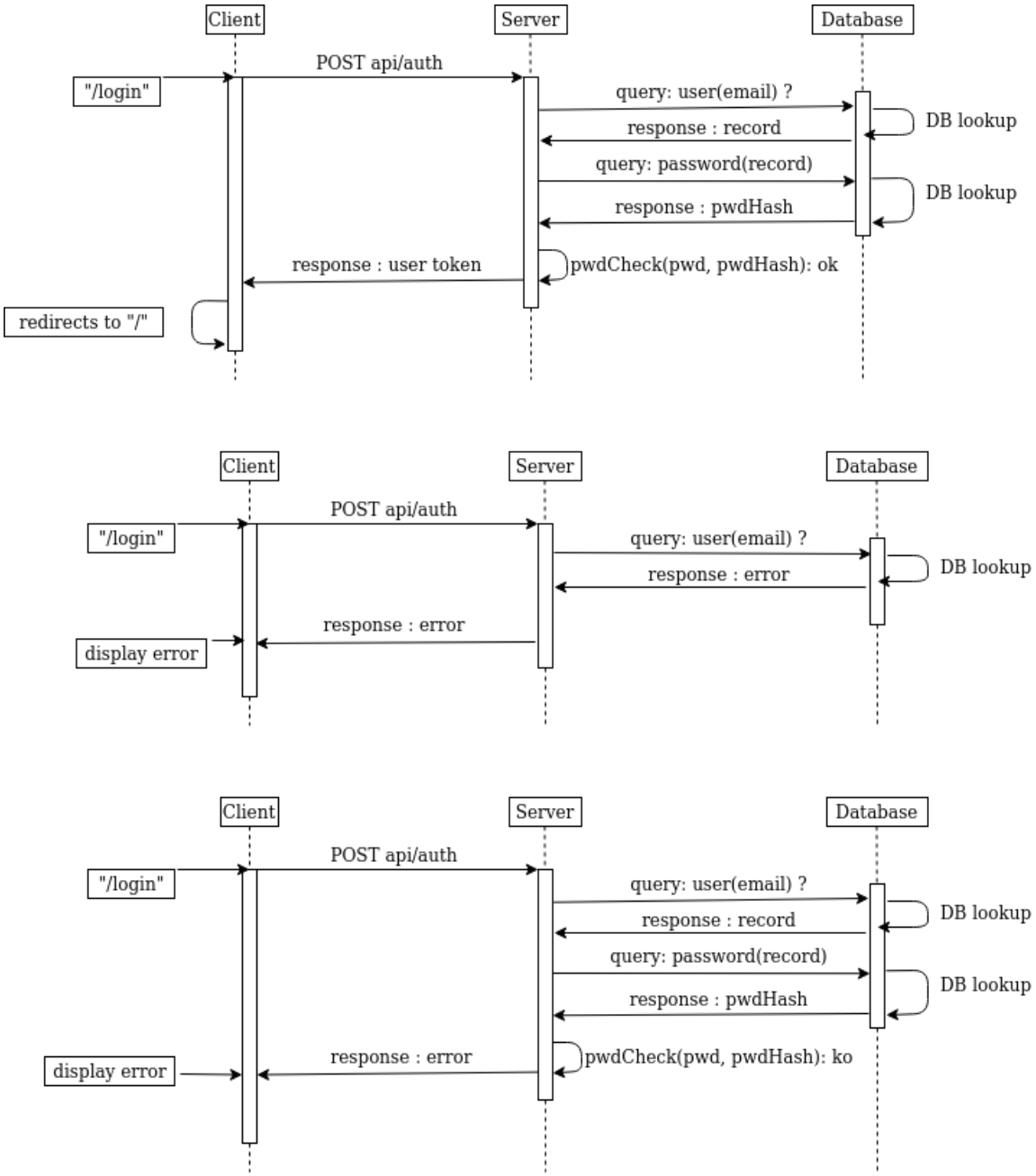


Figure A.2: Sequence diagrams for user authentication

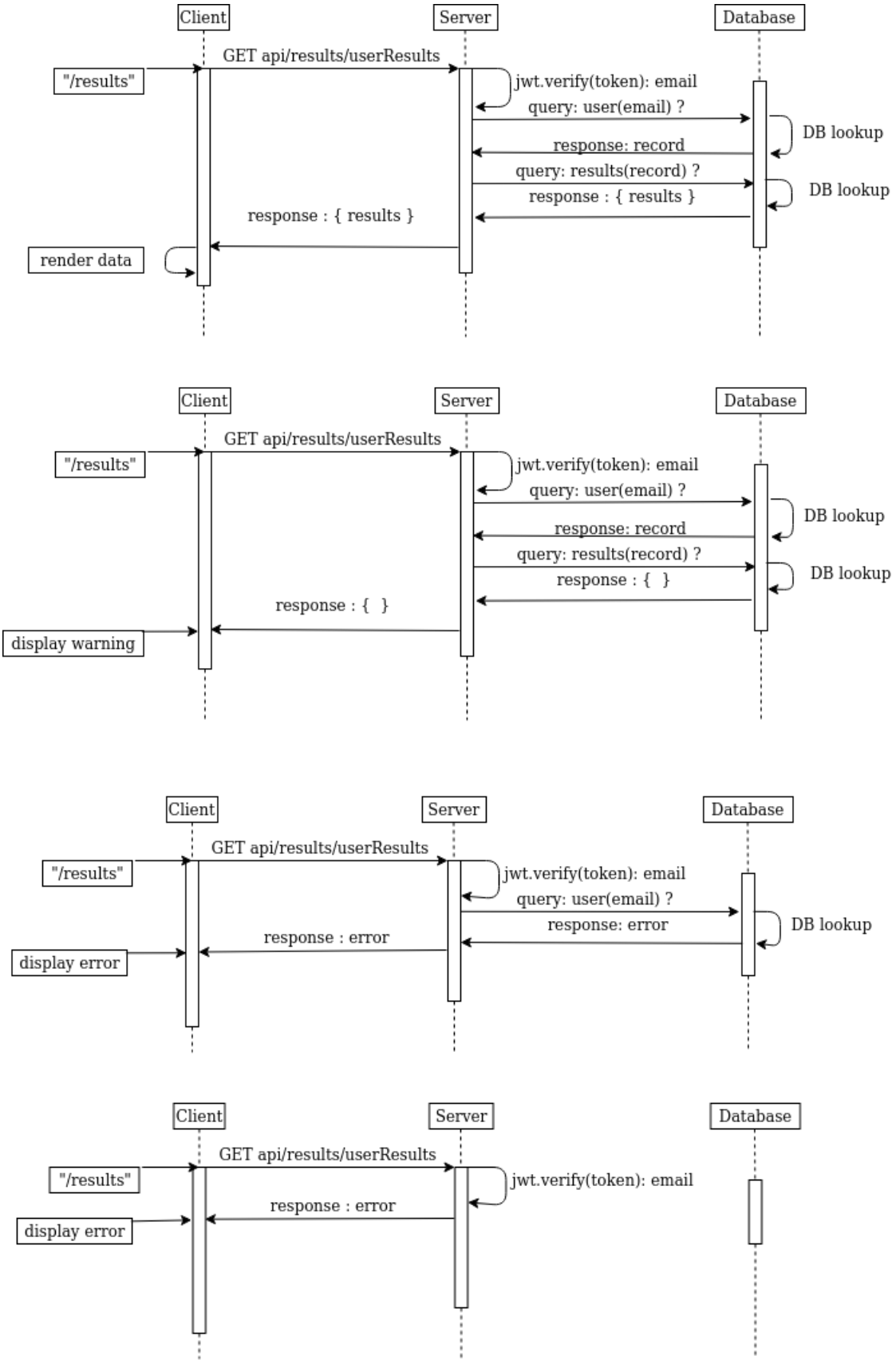


Figure A.3: Sequence diagrams for results fetching

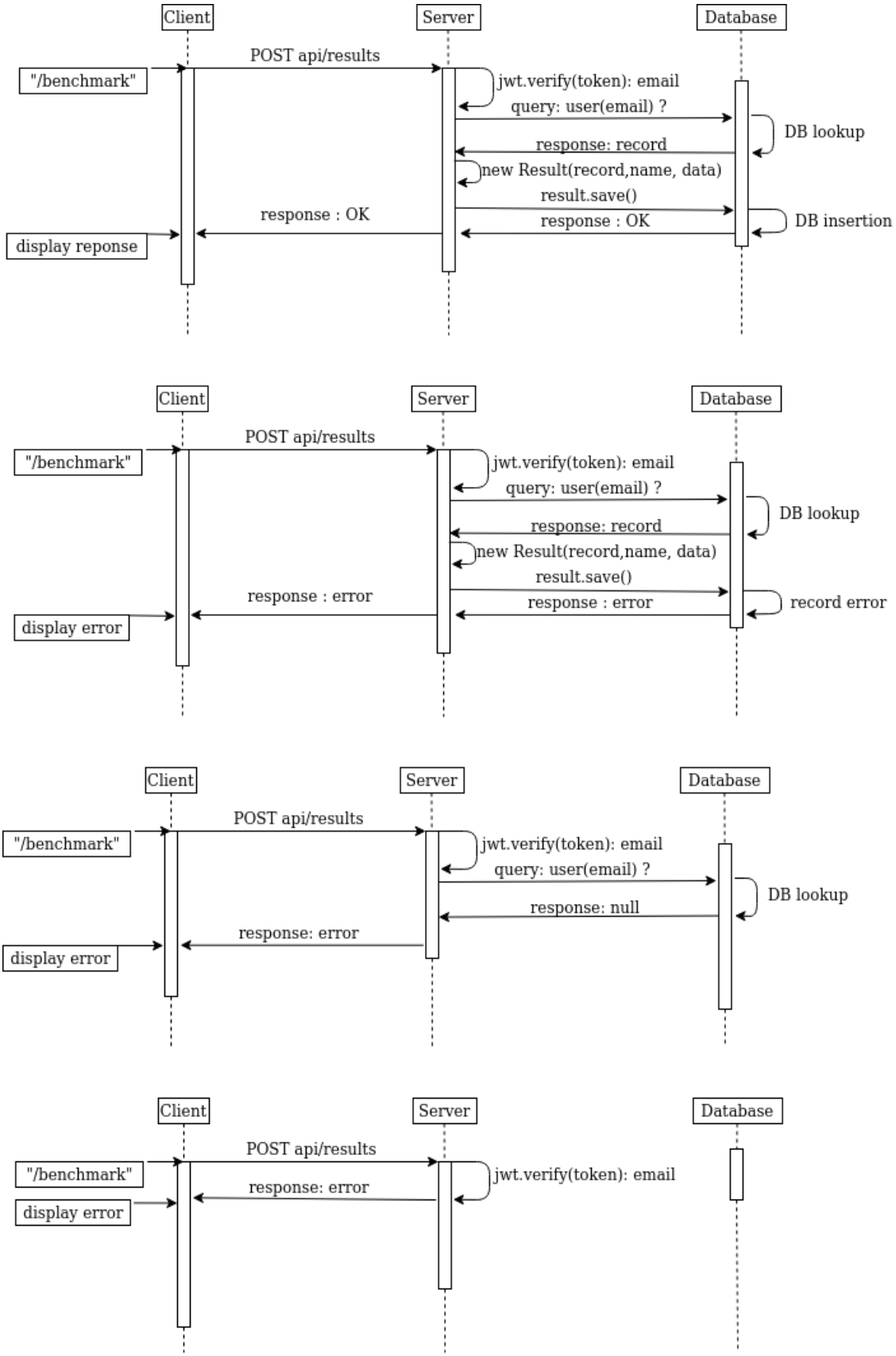


Figure A.4: Sequence diagrams for results saving

Appendix B

Flow Chart

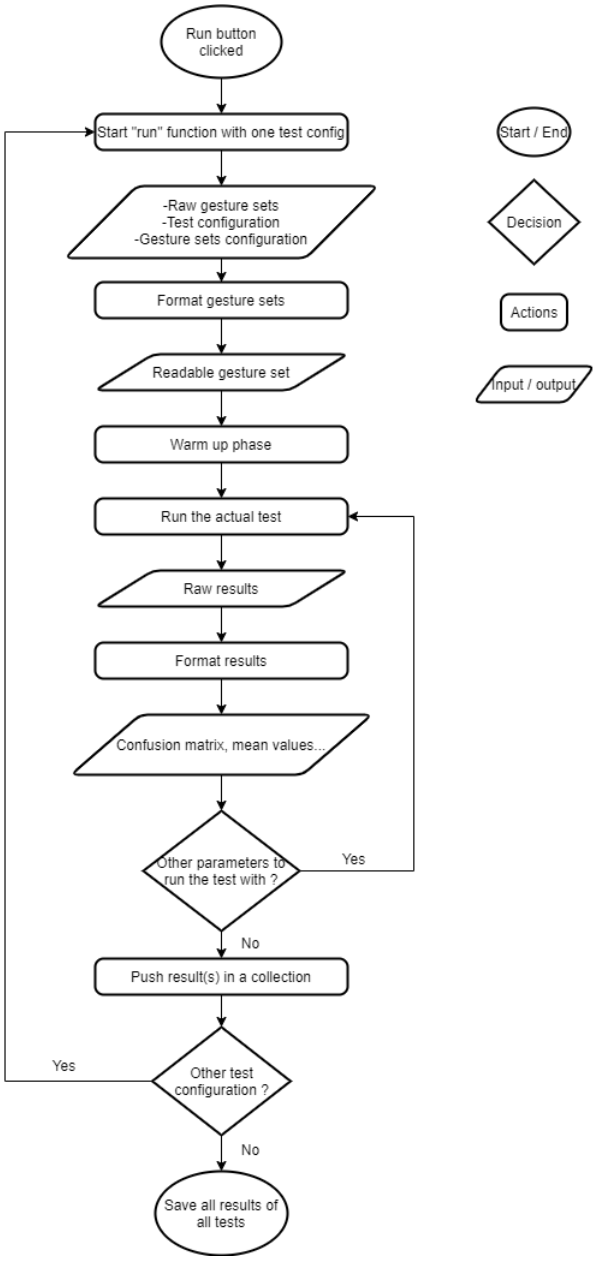


Figure B.1: Flow chart of the execution of a benchmark

Appendix C

Gester user interface

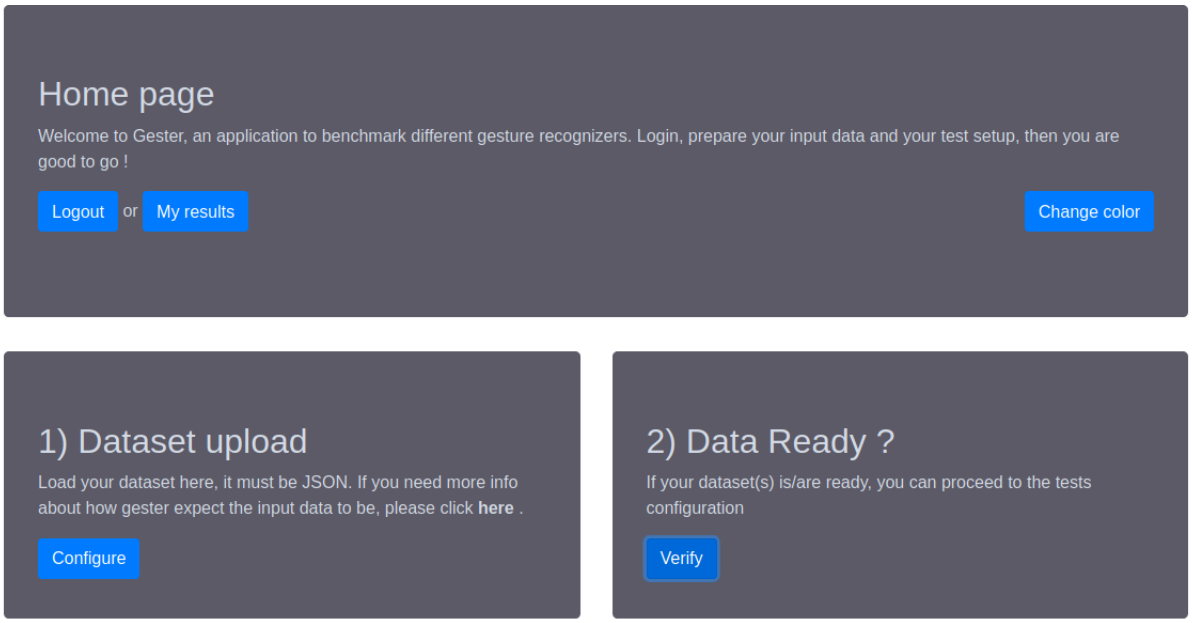


Figure C.1: Home page of Gester

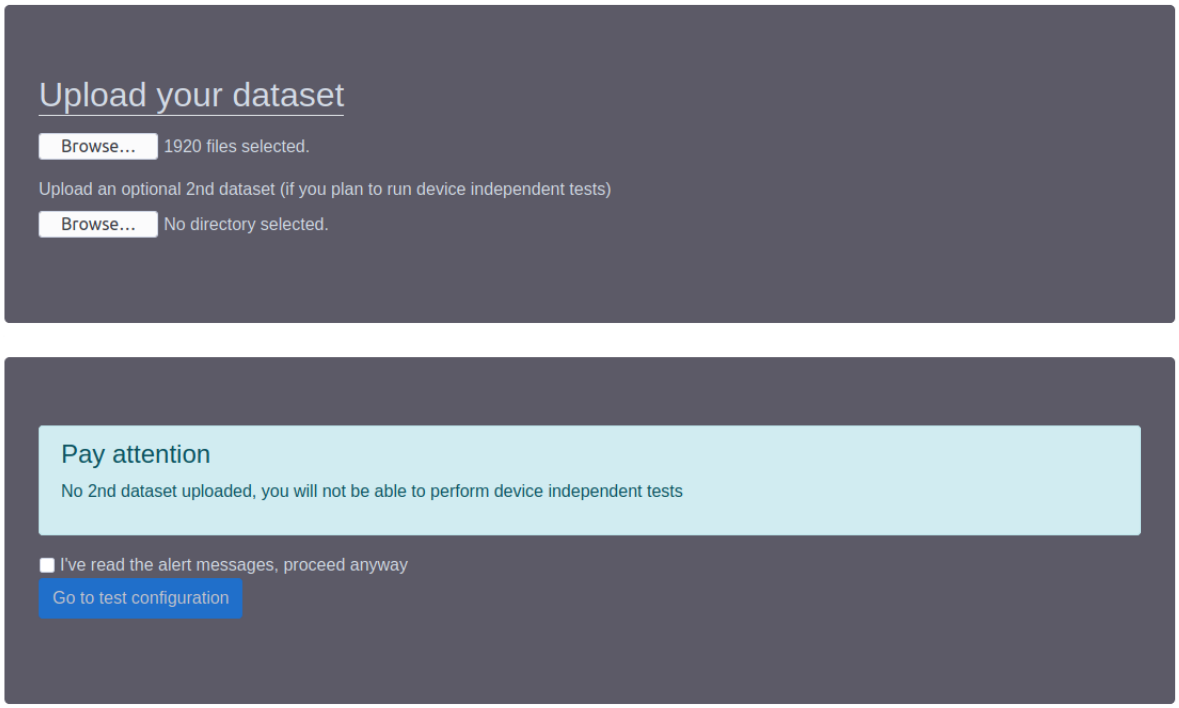


Figure C.2: Uploading a gesture set

Figure C.3: Managing test configurations

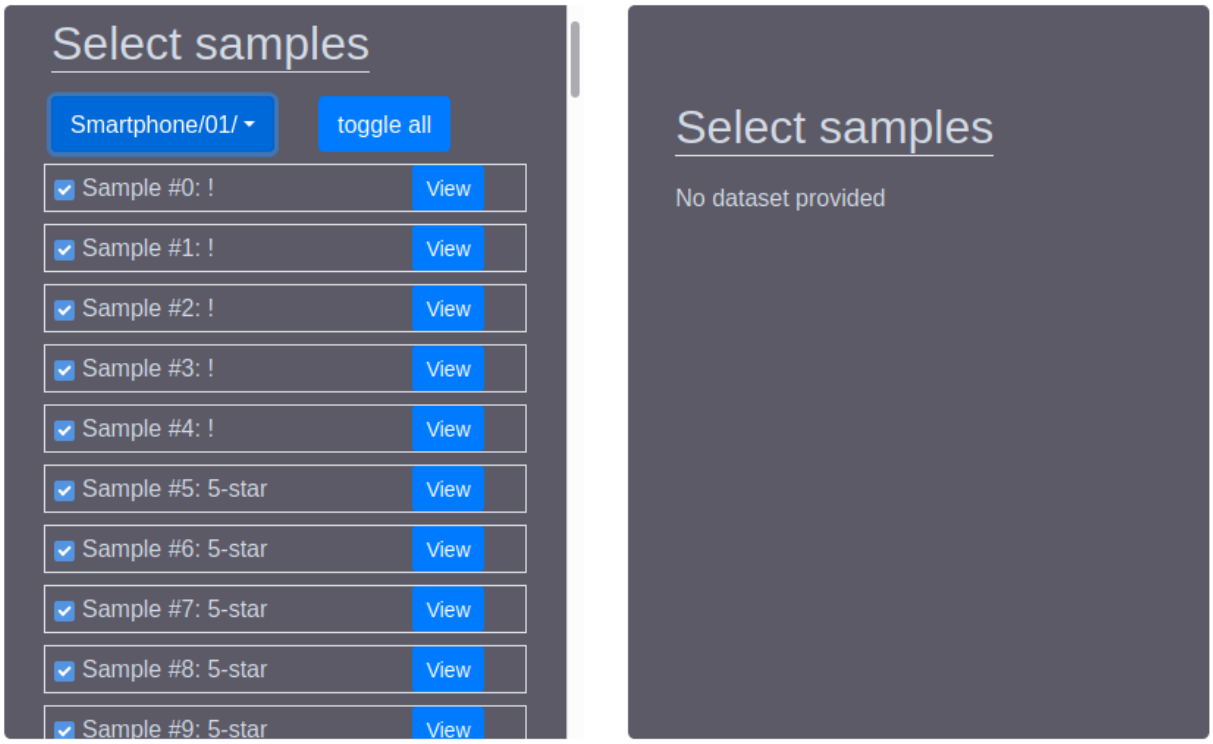


Figure C.4: Selecting gesture samples

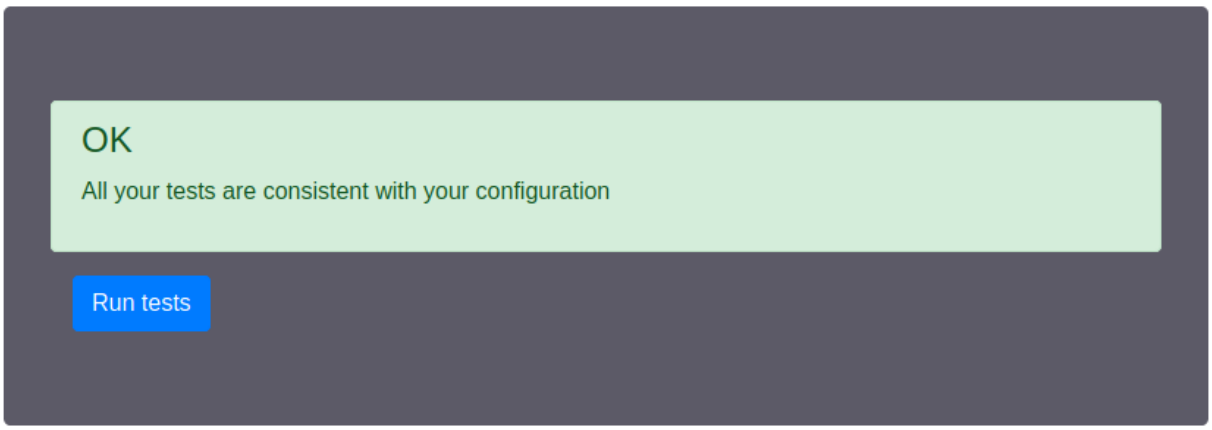


Figure C.5: Automatic verification and start of benchmark procedure

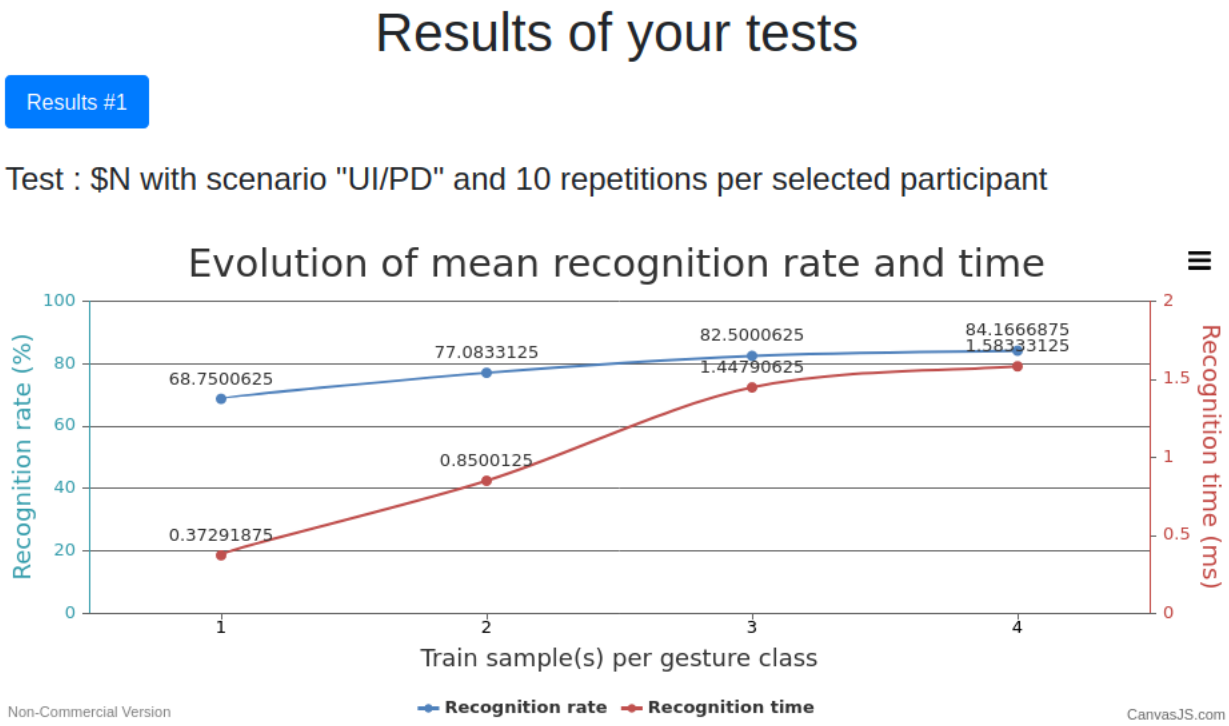
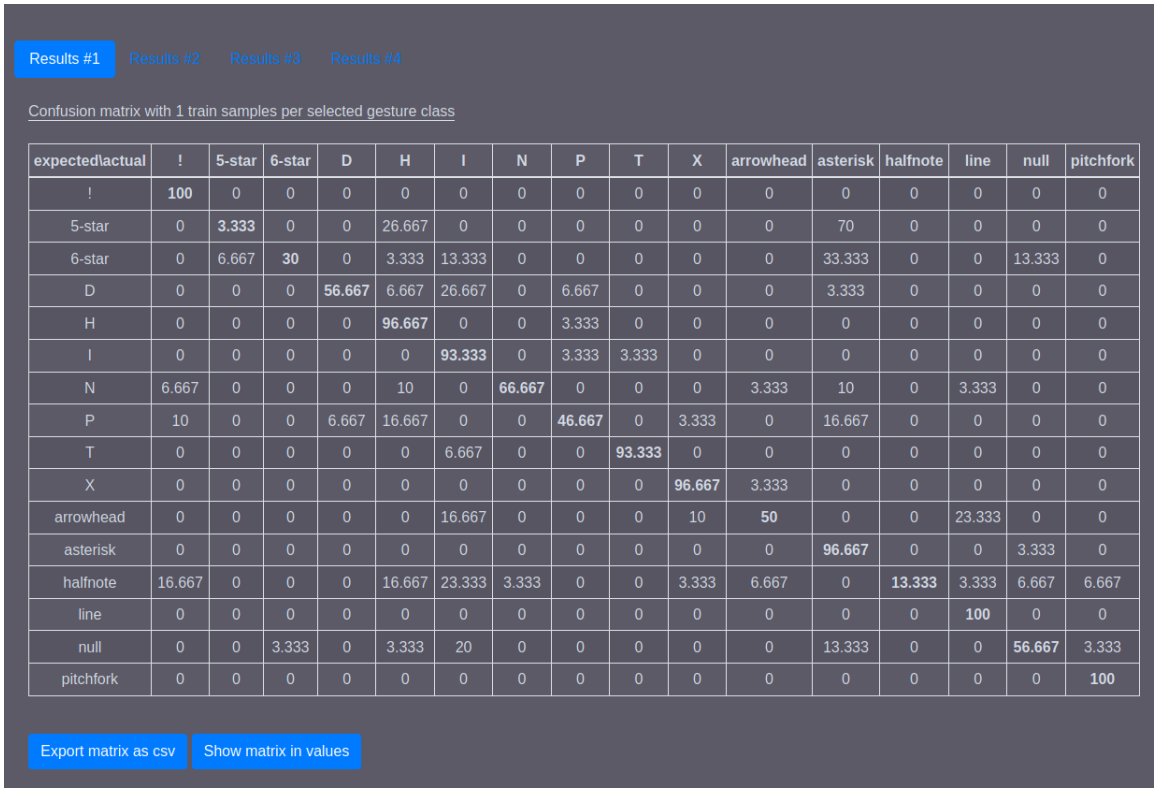


Figure C.6: Example of spline chart returned from a test



expected\actual	!	5-star	6-star	D	H	I	N	P	T	X	arrowhead	asterisk	halfnote	line	null	pitchfork
!	**100**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5-star	0	**3.333**	0	0	26.667	0	0	0	0	0	0	70	0	0	0	0
6-star	0	6.667	**30**	0	3.333	13.333	0	0	0	0	0	33.333	0	0	13.333	0
D	0	0	0	**56.667**	6.667	26.667	0	6.667	0	0	0	3.333	0	0	0	0
H	0	0	0	0	**96.667**	0	0	3.333	0	0	0	0	0	0	0	0
I	0	0	0	0	0	**93.333**	0	3.333	3.333	0	0	0	0	0	0	0
N	6.667	0	0	0	10	0	**66.667**	0	0	0	3.333	10	0	3.333	0	0
P	10	0	0	6.667	16.667	0	0	**46.667**	0	3.333	0	16.667	0	0	0	0
T	0	0	0	0	0	6.667	0	0	**93.333**	0	0	0	0	0	0	0
X	0	0	0	0	0	0	0	0	0	**96.667**	3.333	0	0	0	0	0
arrowhead	0	0	0	0	0	16.667	0	0	0	10	**50**	0	0	23.333	0	0
asterisk	0	0	0	0	0	0	0	0	0	0	0	**96.667**	0	0	3.333	0
halfnote	16.667	0	0	0	16.667	23.333	3.333	0	0	3.333	6.667	0	**13.333**	3.333	6.667	6.667
line	0	0	0	0	0	0	0	0	0	0	0	0	0	**100**	0	0
null	0	0	3.333	0	3.333	20	0	0	0	0	0	13.333	0	0	**56.667**	3.333
pitchfork	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**100**

Figure C.7: Example of confusion matrix, here in percent

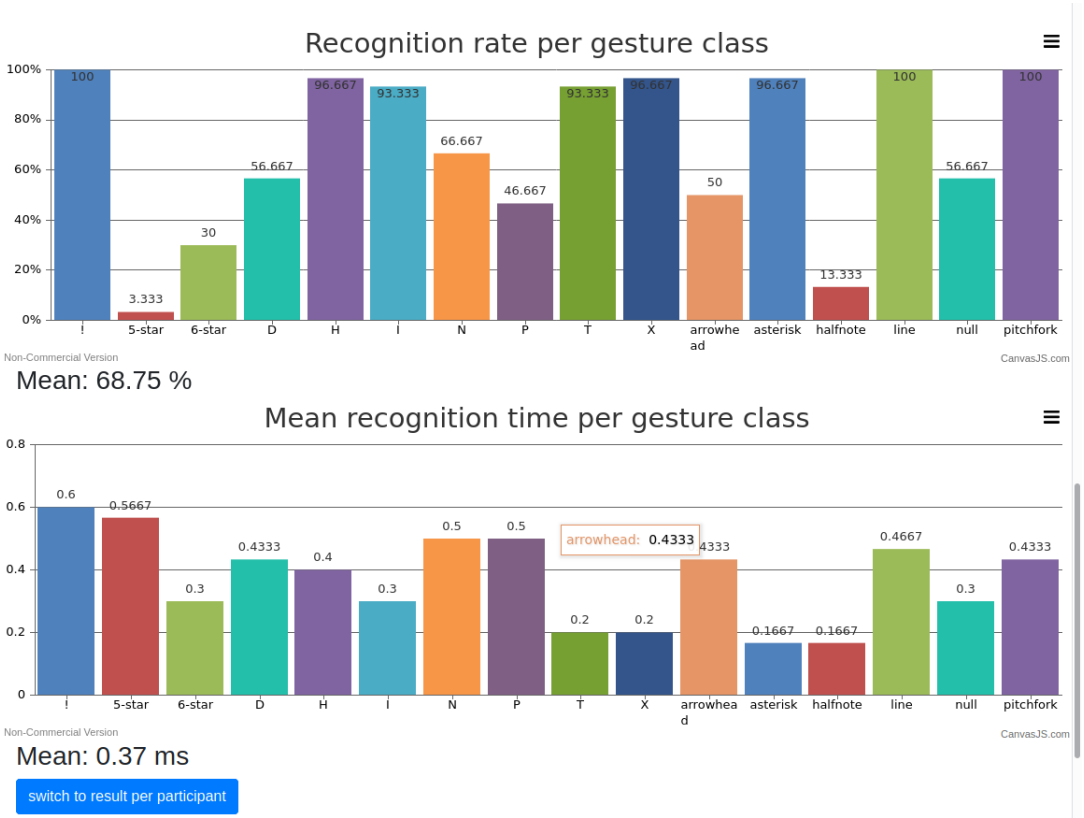


Figure C.8: Examples of bar charts, showing recognition rate and mean recognition time

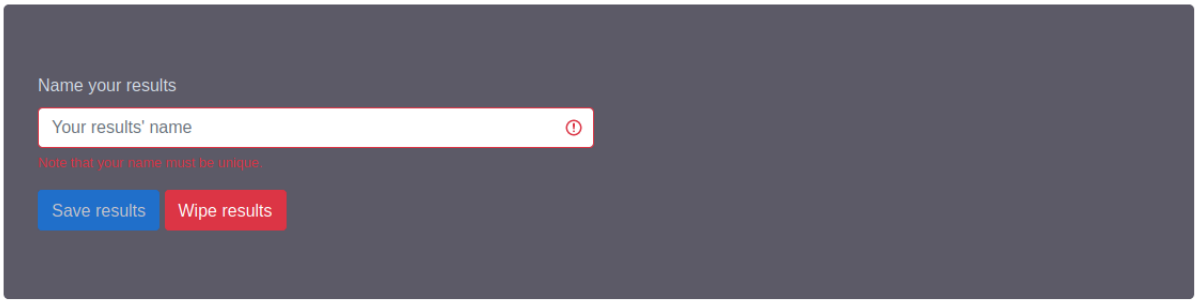


Figure C.9: Saving results for later

Glossary

DOM	Document Object Model.
FSM	Finite State Machine.
Gesture Recognizer or Recognizer	Algorithm whose objective is to interpret human gestures..
GUI	Graphical User Interface.
HCI	Human-Computer Interaction.
HMM	Hidden Markov Model.
HTML	HyperText Markup Language.
JIT	Just-In-Time (compilation). A compilation performed at run time..
JSON	JavaScript Object Notation.
k-NN	k-nearest neighbor.
MERN	MongoDB, Express, React, Node.
PD/PI	Platform Dependent / Independent.
Recognition Rate	Percentage of correctly classified gestures by a recognizer.
Recognition Time	Time required by a recognizer to classify a gesture, expressed here in milliseconds.
Sampling Point(s)	Refers to a (set of) coordinate(s). Those coordinates, butted together, give a visual representation of a gesture.
SLOC	Source Line of Code.
UCLouvain	Universite catholique de Louvain.
UD/UI	User Dependent / Independent.
UX	User eXperience.

Bibliography

[1] Roland Aigner, Daniel Wigdor, Hrvoje Benko, Michael Haller, David Lindbauer, Alexandra Ion, Shengdong Zhao, and JTKV Koh. Understanding mid-air hand gestures: A study of human preferences in usage of gesture types for hci. *Microsoft Research TechReport MSR-TR-2012-111*, 2:30, 2012.

[2] Lisa Anthony and Jacob O. Wobbrock. A lightweight multistroke recognizer for user interface prototypes. In *Proceedings of Graphics Interface 2010*, GI '10, pages 245–252, Toronto, Ont., Canada, Canada, 2010. Canadian Information Processing Society.

[3] Lisa Anthony and Jacob O Wobbrock. $n-protractor: a fast and accurate multistroke recognizer. In *Proceedings of Graphics Interface 2012*, pages 117–120. 2012.

[4] Alex Banks and Eve Porcello. *Learning React: functional web development with React and Redux*. " O'Reilly Media, Inc.", 2017.

[5] Robert Camp. *The search for industry best practices that lead to superior performance*. Taylor & Francis, 1989.

[6] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[7] Pengyu Hong, Matthew Turk, and Thomas S Huang. Gesture modeling and recognition using finite state machines. In *Proceedings Fourth IEEE International Conference on Automatic Face and Gesture Recognition (Cat. No. PR00580)*, pages 410–415. IEEE, 2000.

[8] Adam Kendon. *Gesture: Visible action as utterance*. Cambridge University Press, 2004.

[9] Yang Li. Protractor: a fast and accurate gesture recognizer. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 2169–2172, 2010.

[10] Sushmita Mitra and Tinku Acharya. Gesture recognition: A survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(3):311–324, 2007.

[11] Ronald Moen and Clifford Norman. Evolution of the pdca cycle, 2006.

[12] Miguel A. Nacenta, Yemliha Kamber, Yizhou Qiang, and Per Ola Kristensson. Memorability of pre-designed and user-defined gesture sets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, page 1099–1108, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Orasa Patsadu, Chakarida Nukoolkit, and Bunthit Watanapa. Human gesture recognition using kinect camera. In *2012 ninth international conference on computer science and software engineering (JCSSE)*, pages 28–32. IEEE, 2012.

[14] Dean Rubine. *The automatic recognition of gestures*. PhD thesis, Citeseer, 1991.

[15] Eugene M Taranta and Joseph J LaViola Jr. Penny pincher: a blazing fast, highly accurate $-family recognizer. In *Proceedings of the 41st Graphics Interface Conference*, pages 195–202. Citeseer, 2015.

[16] Jean Vanderdonckt, Paolo Roselli, and Jorge Luis Pérez-Medina. !ftl, an articulation-invariant stroke gesture recognizer with controllable position, scale, and rotation invariances. In *Proceedings of the 20th ACM International Conference on Multimodal Interaction*, ICMI '18, pages 125–134, New York, NY, USA, 2018. ACM.

[17] Radu-Daniel Vatavu. Characterizing gesture knowledge transfer across multiple contexts of use. *Journal on Multimodal User Interfaces*, 11(4):301–314, 2017.

[18] Radu-Daniel Vatavu. Improving gesture recognition accuracy on touch screens for users with low vision. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 4667–4679, New York, NY, USA, 2017. ACM.

[19] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. Gestures as point clouds: A $p recognizer for user interface prototypes. In *Proceedings of the 14th ACM International Conference on Multimodal Interaction*, ICMI '12, pages 273–280, New York, NY, USA, 2012. ACM.

[20] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. $q: A super-quick, articulation-invariant stroke-gesture recognizer for low-resource devices. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '18, pages 23:1–23:12, New York, NY, USA, 2018. ACM.

[21] Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, pages 159–168, New York, NY, USA, 2007. ACM.

[22] Junji Yamato, Jun Ohya, and Kenichiro Ishii. Recognizing human action in time-sequential images using hidden markov model. In *CVPR*, volume 92, pages 379–385, 1992.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**