POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

AdKey: a Personalized Situation-Aware Soft Keyboard

Internal Supervisor: **Prof. Pier Luca Lanzi**
Politecnico di Milano

External Supervisor: **Prof. Piotr J. Gmytrasiewicz**
University of Illinois at Chicago

Master Thesis by:
Angeleri Michael Leo Idek
Student ID 755907

Academic Year 2012-2013

To Brendon ...

Abstract

With the advance of technology, mobile devices allow users to perform on the go tasks that were delegated to the desktop. One disadvantage is the difficulty of text input due to the lack of a physical keyboard.
In our work we investigated how the external context induced situational impairments that changed the users behavior. We believed that being able to model such changes would reduce the effort required by the users to adapt to different situations.
We developed a soft keyboard that was able to sense the external variables that define the situation and infer the typing dynamics of the user. Using this information we adjusted its shape and reduced the error rate caused by one or more situational impairments.
Our final prototype reduced the error rate to half the values observed on a non adaptive keyboard, letting the user type about 1.2 times faster.

Sommario

Con lo sviluppo della tecnologia, i dispositivi mobili permettono di svolgere operazioni che erano prima relegate al desktop. Uno svantaggio è la difficoltà di inserire testo, dovuta alla mancanza di una tastiera fisica.

In questo lavoro abbiamo esplorato in che modo il contesto esterno induca disabilità situazionali, che influenzano il comportamento dell'utente. Siamo convinti che, modellando questi cambiamenti, si possa ridurre lo sforzo richiesto dall'utente per adattarsi alla situazione avversa.

Abbiamo sviluppato una tastiera adattiva in grado di rilevare le variabili esterne che definiscono la situazione e di dedurre le modalità di digitazione dell'utente. Usando queste informazioni per modificare la configurazione dei tasti, abbiamo ridotto il margine di errore dell'utente causato da una o più disabilità situazionali.

Il nostro prototipo definitivo riduce il margine di errore a circa la metà, permettendo inoltre agli utenti di guadagnare un fatttore di 1.2 in velocità di digitazione.

Acknowledgements

I want to thank my parents that supported me throughout this experience, financing me spiritually and monetarily. – I love my grandma and I want to thank her saying ... "t'è vist'? t'a gà un nivùd ingegnier!" – Thanks to my friends that supported me through all this process, either they came on holiday with me, enjoyed Sulella's, Baghdad (now Sofi), patios, secretaries ... – Special thanks to whom paid taxes for me. – Thanks to Chicago, I'm leaving a piece of my hearth there, thanks to a special person that lives there. – Thanks to my advisors, without them I wouldn't graduate!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A new form of computing devices has appeared on the market, and is gaining huge portions of the electronics market share. With the rapid increase of their power, these small devices are almost reaching the computing capabilities of their desktop counterparts. Whether they are smartphones, tablets, or even watches, they permit the users to perform a huge variety of tasks on the move; thanks to their advanced connectivity, they let you check the e-mails while waiting in line to pay the grocery or write an important document while on the bus.

But portability comes at a price: the limited size of these devices forces the producer to make compromises on the hardware they can use. In particular, the majority of the devices does not to carry a physical keyboard, that has been replaced by a soft keyboard rendered on a touch screen. This choice leaves more space for interaction when there is no need for a keyboard, but becomes limiting when performing text input: the reduced size of the keys, combined with the lack of haptic feedback that the flat surface of the screen can not provide, make typing on mobile an area open to improvement.

In our work, we investigated how the external context induced situational impairments that changed the users behavior, and we defined a model that aims both to detect a change in the context and to automatically adapt in order to reduce the effects of the diminished performance.

We performed an exploratory study in which we asked subjects to type on a soft keyboard in an observed environment, while performing different tasks that would in our opinion lead to situational impairments. With the information retrieved, we had insights on how these situations affected both the user performance and their typing dynamics.

These findings brought us to the development of an adaptive soft keyboard that, being aware of the user patterns and of the situation he was

typing in, adjusted its shape with the aim of reducing the errors performed.

Simulations on data retrieved showed us the importance of keeping in count the situation when designing a typing interface, and a final evaluation session with subjects confirmed these findings.

Our final product reduced the error rate up to a factor of two, by means that the average error rate of users dropped to half the value that we observed in the retrieval session with a standard, non adaptive, keyboard.

We approached our problem as a machine learning instance, in which we had to perform all the steps: retrieve the data, select the variables, analyze them with different techniques, create a model that best performs on the data, eventually test the model on the field.

The data retrieval is in some case an implicit process, because data is already available from other sources. This was not the case for our problem: we specifically wanted to observe the behavior of users while typing in different conditions; while there had been previous works on retrieving user typing dynamics, none of them dealt with the conditions in which they were typing, but asked them to type in an observed environment, while seating or standing.

We decided to build a prototype that would have been our tool to perform data retrieval, as this approach had already been performed in other researches similar to ours. Our prototype would need to behave, from a user point of view, as a smartphone keyboard with which he could type; in addition to that, it should have recorded all the information provided by the phone built-in sensors that was pertinent to typing.

We recruited participants to our experiment from the student population, through the campus mailing list, asking them to participate in our research. They were not compensated monetary, since our research was not funded, but they were offered some appetizers brought by the authors. They agreed to participate in our research after receiving proper information about it and signing a user consent.

Participants population was composed by eleven grad students, with average age of 24.3 years; all of them declared to possess a smartphone that used regularly, even if few of them specified that thumb typing on soft keyboard was not their usual input method, preferring other techniques to it.

During the training session, users were asked to type a set of phrases while performing different tasks that were proposed on the screen of the device. The tasks were: type as usual, as fast as you can, while walking and with a glare on the screen.

In order to have access to the data from a variety of platforms, we loaded

them on a persistent database. We associated each keystroke to the key that was meant to be typed, by comparison with the proposed phrases. Such classification was carried out with human supervision, because we observed that in some tasks it would have been hard for a software to correctly classify the majority of the strokes.

We performed several analyzes on the data, in order to select the correct variables for the learning algorithm. In this phase, a model for inferring the user subjective typing speed was also defined: we observed that keystroke pressure durations were almost constant given the user, independently from the speed, and that its average had a good linear correlation with the user preferred typing speed.

Three different machine learning classifiers were trained on the data, aiming to classify the user intended key. Tree classifiers and neural networks were discarded for a probabilistic approach in which the keystroke location was modeled as a bivariate normal distribution, and the most likely key was selected as the intended one.

Simulations on this model gave us evidence on our hypothesis that the situation in which one types does affect the typing dynamic, and the best performing classifier is the one learned from data acquired in the same situation. Furthermore, they confirmed other works hypotheses that personalizing the shape of the keyboard on the user provides better performance than an aggregated model.

In order to put this acquired knowledge into a practical application, we developed a Bayesian network model that was responsible of quantifying the variables representing the external situation and accordingly adjust the probability distribution over the keys. Given knowledge about the user and the situation, we were able to adapt the shape of the keyboard to the one that from our simulations performed best.

The choice of probabilistic modeling permitted us to extend what we had defined, defining a hidden Markov model that added knowledge about the language. We developed a very simple language model, just to prove that this possibility was available, but leaving out complexity of a well-defined one because our goal was focusing on the effects of the situation.

With the obtained results, we implemented an Android application, namely AdKey, that exploited all the findings described above, to offer a better typing experience with reduced error rate. Since the models parameters were personalized on the user, we invited the subjects that had participated to the data retrieval, and we gave them the opportunity to test our application; seven of them agreed to participate to this second session, and confirmed the trend observed with simulations.

Results showed that our adaptive soft keyboard let the user type doing about half the errors they did with a non adaptive keyboard, and this affected their typing confidence by slightly improving also their typing speed. Nevertheless, we observed that the improvement was decreasing when the language model was added to the classifier; we thus suggested that a first order Markovian process is unreliable to model the language.

The thesis is structured in the following way.

In chapter two we denote some historical facts about typing on digital devices, and then we outline the state of the art of the research on mobile typing.

In chapter three we analyze the reasons that led us into the topic and we state the purpose that we aim to.

In chapter four we describe how we approached the design of the work from a logical perspective.

In chapter five we define the components of our model and specify the implementation choices.

In chapter six we illustrate the performance of our application and outline the goals and their evaluation.

In appendix A we report the graphical models of our components.

In appendix B we show the materials used and report some statistics on them.

In appendix C we provide the documentation of the project.

In appendix D we make accessible the source code.

Chapter 2

State of the Art

In this chapter we outline the state of the art in mobile typing. We take a brief historical overview to define what brought the research to analyze the current problematics, and then we cite the most recent works that addresses the problematic of typing on mobile, with a special attention on the ones that aim to keep in count the device external context.

2.1 Text input in digital devices

Text input is one of the most frequent user tasks that is performed during interaction with digital devices. Text writing can be carried out to reach various goals, including write email and chat messages, typing commands, taking notes and coding programs.

The desktop computer has accustomed us to the *pointing device-keyboard-display* paradigm, where the keyboard is a flat device with mechanical buttons whose pressing induce an interrupt to the operative system, which in turn takes care of notifying the inputted text to the active application.

It is interesting to point out that the actual design of the keyboard that all the world is used to, commonly referred to as *QWERTY* layout, was defined years before the advent of the computer by a typewriter company. It was specifically designed to reduce the jams in the typebars when typing at fast speed, by placing commonly subsequent characters such as "th" in non adjacent positions [48].

Thus, even if the *QWERTY* layout was designed to improve typing speed, the advent of digital devices overcame the issue of jams, leading to critics about its effectiveness. The *DVORAK* layout was proposed, which

Figure 2.1: The 12-key keypad of a Nokia 3310

permits to type a higher ratio of English words without moving the fingers from the central row; despite this performance oriented design, there has been no proof that this could provide significant advantages [38, 47]. The *QWERTY* has been so widely adopted as the *de facto* standard layout that has been analyzed in the economics as one of the most important cases of open standard adoption [6].

Earlier mobile phones did not require text input, since they adopted a numeric keypad to compose telephone number identifiers. Only the advent of SMS (Short Message Service) introduced the issue of how to compose text with this small devices.

First solutions were based on the same 12-key layout, but with keys labeled with multiple characters, as the one showed in Figure 2.1, in which multitap was performed [17]; this could be achieved mainly in two different ways: either by tapping the key showing the intended character a number of times corresponding to the position of it, or by two presses, the first of which identifies the intended key and the second the position [5].

A great improvement in terms of keystrokes per character (KSPC) [39] occurred with the introduction of character disambiguation. The most known technology in this field is undoubtedly T9 by Tegic Communications [20], which seeks to match to a given key sequence the most likely word selected from a dictionary.

With disambiguation, the user did not need anymore to perform multi-presses, except when selecting from alternative words. This was proven to lead to more than twice faster typing speed with reduced error rate, once the user has gained a proper level of experience [29]. Other techniques were proposed, using probabilities of letter sequences to guess the intended letter

[40], rather than exploiting different layouts to reduce the need for disambiguation [14].

2.2 The advent of touchscreen

In the last decade, mobile phones became pervasive in our society, and a new typology, capable of performing a wider selection of functions other than calling and sending short texts, emerged.

We refer to them as smartphones, and are portable devices running a complete operating system, permitting the development and installation of third-party applications. The intrinsic capabilities extensibility, combined with their advanced connectivity, made it possible to perform many tasks on this devices that were previously confined to desktop computers, such as navigating the internet and writing documents.

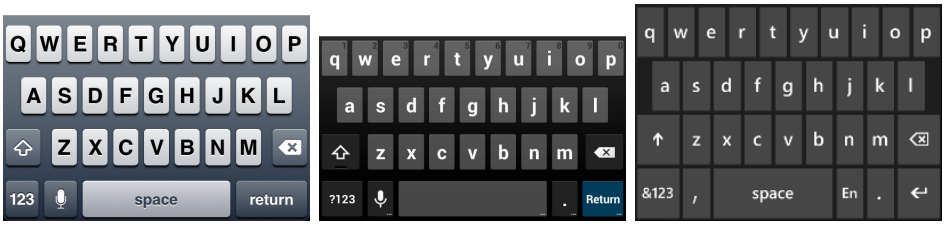
The key factor that expanded this market was the release of touch screen equipped devices. This devices provided an innovative way of interaction with them, based on touching with one or more fingers the screen surface.

Despite the technology was invented in the forties, attempts to create consumer products did not succeed until 2006, with the release of Apple iPhone. A detailed analysis of the factors that determined the success of iPhone is beyond the scope of this thesis, but we would like to outline one aspect, that is the design of an innovative and intuitive user interface, capable of transmitting to the user the idea that he is actually interacting with what is showed on the screen [36].

When gathering information on experiments performed with touchscreen devices, one should almost always distinguish whether they were performed with a stylus rather than with direct finger, because they provide a different way of interaction. While stylus has been proven to be more effective both for speed and accuracy [28], it is less convenient for users, who have to take it out every time they want to interact with the device, with the risk of losing it. In the following, we will focus our findings on fingers interaction.

The lack of accuracy was attributed for a long time to the so called *fat finger problem*. That is, users are not able to point precisely on a touch screen because the softness of the skin leads to a big, hardly controllable touch area and that the target is not visible because it is occluded by the finger [55].

Solutions to this has been proposed such as showing a copy of the occluded screen area in a non-occluded location. This approached appeared to be useful especially to acquire very small targets [56].



(a) (b) (c)

Figure 2.2: The QWERTY soft keyboards of (a) iPhone, (b) Android, (c) Windows Phone

A recent work from *Holtz et al.* suggested that inaccuracy should instead be attributed to the failure in modeling the *perceived input point*, which varies among users, how they are holding the device, and the different angle at which they point to the screen [26]. In a follow-up work they showed that touch accuracy can be incremented if the mental model of the user is known [27].

Other research confirmed this finding, showing that analyzing the user pattern and subsequently building a personalized model increases accuracy [10]. Indirectly, this brought researchers to invert the process and analyze user patterns in order to identify them, as it can be useful for security applications [51, 33, 59].

2.3 Touch typing on soft keyboards

In the majority of the devices in this category, text input is performed via a *QWERTY* software keyboard, that is, a keyboard rendered on the display with which the user can interact by touching over the keys on the screen. The reduced size of such keyboards, and the lack of a tactile feedback over the boundaries of the keys makes this method less precise than its desktop parent. Figure 2.2 shows how this keyboards are rendered for the actual three most widespread mobile operative systems.

Data on user performance are scarce, but one evaluation claims that texting on iPhone took twice as long as texting on a physical phone-sized *QWERTY* physical keyboard [25].

To overcome the lack of feedback, different techniques have been proposed, some of which are applied in current systems. One solution is to display the character associated to the key with a bigger font in an area visible to the user when he is keeping a finger on the screen; this lets the

user correct the finger position before lifting the finger when entering text slowly and to see the entered key within their visual focus on the keyboard to help notice errors when entering text rapidly.

Lack of tactile feedback can be slightly reduced by providing an haptic feedback through the built-in mobile phone vibration actuator [4]. This guarantees that one key has been pressed, but cannot deliver the information on *which* was pressed; nevertheless, it has been proven to slightly increase speed while typing [25], and is currently applied in most of the available smartphones.

The *QWERTY* layout was designed for ten fingers typing, and researches pointed out that this may not be the optimal layout for typing with either one or two fingers.

Fitts' law [11] describes the relation between travel time T and distance between two keys D in the form of

$$T = a + b \log_2 \left(1 + \frac{D}{W}\right)$$

where W is the size of the key that has to be reached, a and b are empirical parameters. Experiments showed that this model, while designed for more general interactions, might also be advantageous in touch screen typing.

Novel layouts were proposed, designed with the specific aim of reducing the finger travel time with the aid of automatic techniques to reach the optimal layout, such as Metropolis [58] and *OPTI* [40]. Unfortunately, while these layouts showed improvements in text typing, the need of long training phases to achieve optimal results seems to leave this works in the theoretic modeling realm, since the users prefer to stick with the already mastered classical layout.

The progress in touchscreen devices brought to the development of new categories of devices, such as tablets and tabletops; the former is basically a bigger smartphone, that maintains its portability characteristics, while the latter is equipped with an even bigger touchscreen, usually with more advanced touch recognition, e.g., the Microsoft PixelSense has a 30 inches display. In this devices the space available for adaptation is bigger, so they were the first on which experiments on personalizing soft keyboards were performed.

Findlater et al. implemented an on-line adaptive keyboard capable of reshaping itself knowing the user intended keystrokes [9], while a similar project used more data coming from the sensors to create a probabilistic model [7]. However, the increased dimension of the screen allows both for more accurate typing and more space for adaptation than a handheld device.

2.4 Adapting the layout

Since soft keyboards receive input as a continuous variable, representing the position of the point of touch, rather than discrete events such as keypresses, a probabilistic approach can be exploited to infer the most likely key to associate with the registered position given some model, usually based on the language.

In its simplest connotation, of an early exploratory work, this was achieved by relaxing Fitts' law on the assumption that some bigrams, i.e., sequences of adjacent characters, are very unlikely to appear in English language. This suggests that they should be replaced with a more likely character in the proximities of the one pressed [34].

Exploiting the same observation, *Faraj et al.* implemented a visually adaptive keyboard, that expands the keys as a function of their letter probability of entry [8]. This is also based on the Fitts' law, specifically on the assumption that enlarging keys will reduce acquiring time. This solution provided a better accuracy, though the approach of visually resizing keys is still debated, with some work showing that it might distract the user [24, 9] by requiring him to adapt to the changing layout.

A more accurate language model was provided by *Goodman et al.*, thanks to his studies on language model [18], he formalized the interaction between language and keyboard models [19]; using Bayes' law, the most likely sequence of characters given a sequence of touch points becomes

$$\begin{aligned} k_1^*, \dots, k_n^* &= \arg\max_{k_1,\dots,k_n} P(k_1, \dots, k_n | l_1, \dots, l_n) \\ &= \arg\max_{k_1,\dots,k_n} \frac{P(k_1, \dots, k_n)\ P(l_1, \dots, l_n | k_1, \dots, k_n)}{P(l_1, \dots, l_n)} \\ &= \arg\max_{k_1,\dots,k_n} P(k_1, \dots, k_n)\ P(l_1, \dots, l_n | k_1, \dots, k_n) \end{aligned}$$

in which the last passage is thanks to the fact that the denominator is a constant given the sequence of keys on which we are maximizing. We have thus expressed the probability of the word in relation to the language probability (first term) and the touch probability (second term).

This relationship has an important drawback also on our work, and will be deeply analyzed in Chapter 4, but basically it permits to define a probabilistic model of the keyboard, that does not have to necessarily coincide with the standard one, i.e., with the mean values placed in the center of the keys. A subsequent work expanded this findings by placing *anchors* on the keys to avoid excessive reshaping [21].

Different approaches have been attempted in order to define what can be the *actual* keyboard model, and it has been discovered that this does not necessarily coincide with the standard model we defined above. *Henze et al.* deployed an Android application on the market that had many thousands of installations, which asked the users to enter text with a scoring function to engage them and encourage them to type faster. They observed that, on aggregated data, the keypresses were shifted towards the bottom of the screen, and proposed a modified model that was able to better classify the input [23]. A similar work reshaped the keyboard using both aggregated and personalized acquired information, showing that the input error rate can jump from 78.2% to 82.4% when resizing the keys [52].

Since the works mentioned above were performed on a very large amount of users, there was no control over the external variables, because the experiment was not observed. Opposed to that, experiments executed in an observed environment, such as a research laboratory, permitted to understand how the keyboard model can change widely when the user is asked to type in different modalities, such as using one or both thumbs [1].

The other key factor while is also the language model, by means of a model that assigns to a sequence of words a probability of it to appear. The most simple model in this case considers each words independent from the other words in the sequence, and for this reason is called unigram model; in this case

$$P(w_1.w_2.w_3) = P(w_1)P(w_2|w_1)P(w_3|w_1.w_2) = P(w_1)P(w_2)P(w_3)$$

where . is the concatenation operator.

More complex models keep in count the relationship between words, thus considering the second term of the formula above, but limited to a n long sequence of words.

In this case problems arise on how to estimate the probability of these *n-grams*, because simply counting their occurrences, even from huge text corpora such as the Google *n-gram* database, will assign zero probability to sequences that are not present in that corpus, but that could appear in a newly observed text. Smoothing techniques, such as Katz [31] and Kneser-Ney [46], are designed to deal with this issue, taking out probabilities of unlikely sequences and redistributing them over unseen trigrams, under the assumption that if the observed sequence was unlikely in the analyzed corpus, other similar sequences should have almost the same probability to appear even if they did not.

2.5 Keeping into account the situation

Mobile computing introduced a new paradigm of how user interact with their digital devices. What was previously intended as a typical desktop activity, became something that could be performed in any kind of situations. While this can be seen as revolution for users, that can perform tasks such as checking their account balance or writing an e-mail on the move, it introduces a new variable that should be kept in count when designing a mobile interface: we will refer to it in this dissertation as the situation, meaning all the features of the external environment and the user current state that hypothetically influence the interaction with the device.

We think that the best way to describe this concept is by reporting an image from *Jacob O. Wobbrok*.

> "A person using a mobile device on the beach in San Diego may struggle to read the device's screen due to glare caused by bright sunlight, while a user on an icy sidewalk in Pittsburgh may have gloves on and be unable to accurately press keys or extract a stylus."
>
> *The Future of Mobile Device Research in HCI* [57]

While this quote probably addresses only a few elements that can influence the usage, it shows how important it can become, and thus that it should be kept in count while designing a mobile application.

In psychology, it is referred to as *situationally-induced impairments and disabilities*, that is the difficulty in accessing computers due to the context or situation one is in, as opposed to a physical impairment. *Sears et al.* defined a three dimensional context space which categorized these impairments by their primary cause, that can be human, environmental or given by the application itself [54].

When *Wobbrok* in his exploratory work underlined the problem of how mobile devices increased the amount of personal computing done away from the desktop, human-computer interfaces evaluations were performed in laboratories, where users were observed in an *ideal* situation; no works on how this could impact on the usability had been performed [57].

Different researches addressed how walking makes the user less accurate in acquiring a target. Walking through a given pattern reduced the capability of reading [2] and of acquiring a target on the screen [53]. In particular, it has been showed, by usage of a treadmill to perform the test, that the speed of walking is a relevant variable, since even walking at a very low

pace reduces the capacity of acquiring a target, and this increments almost linearly with the speed, with a local optimum at roughly the 80% of the subject preferred walking speed [3].

Interacting with a mobile device in different light conditions or with a glare on the screen, even if evidenced as a potentially influencing factor by different papers, e.g., [49, 57], has been explored in a lesser extent. As far as we know, only one work included in its use scenarios one where changes in lightning conditions are performed, showing that this can increase the response time of the user [2].

To the best knowledge, no work has been performed on the influence of typing speed on accuracy, as this has only been evaluated as a variable for estimating the goodness of text interfaces.

If understanding how these factors affect the interaction is important, at the same time we need to make the device capable of recognizing them in order to eventually adapt. Many devices are equipped with a set of sensors that, if correctly exploited, can carry information on some of the external factors, and other could be inferred directly from the interaction dynamics.

By reading the information provided by the built-in accelerometer, it is possible to infer what is the user activity, such as walking, running or climbing stairs, while the phone is kept inside the pocket [35], and the way the user is holding the device [16]. *Bergstrom et al.* proposed a simple way to read the user walking speed while he is interacting the device by means of standard deviation of the acceleration measured over the y-axis of the phone [3].

Chapter 3

Motivations and Goals

In this chapter we describe the process that led us to the development of the thesis. We start from the motivations that brought us into the topic, then we describe the research work that inspired the actual title and the early assessment of the research by defining the concept of context. Finally, we make explicit considerations that set the key points of the development.

3.1 Preliminary phase

We started inspecting the topic of mobile typing even before the development of this thesis. We knew, from personal experiences and feedbacks from different sources, how this is an issue that would need to be improved, and we had already inspected the current approaches to achieve that.

While having informal conversations with smartphone users on this topic, the main impressions that we were getting varied from the disappointment on the performance that can be reached, to the frustration for the incorrect behavior of the automatic correction tools. These facts triggered our interest in the topic, leading us to the development of the thesis being discussed in this paper.

Furthermore, if one types into an internet search engine the words "mobile typing", the results that he will face will regard either applications that promise a better typing experience, or articles on online magazines describing the issues of this task, and how to improve it by the use use of such applications. This can by no means be interpreted as another evidence that even the public opinion is aware of this issue, and the software industry is working to offer products that aim to reduce it.

It is interesting to point out what are their most recurrent features: apart from the ones that boast of the aesthetic customization, which is out of our scope, either they offer new keyboard layouts supposed to improve typing accuracy, or they exploit artificial intelligence techniques to predict the intended entered text.

For the time being, the only mobile platform available on the market which permits the installation of a personalized keyboard from a choice available on the Internet is Google Android. This can be explained through the fact that Android is an open source operative system, permitting the development of applications that run at a lower level. We are waiting for the release of Ubuntu Phone, that is expected later this year, to see whether it will bring this feature, being another open source system.

We confide that this option will progressively become available on more devices, due to the fact that typing can not be considered anymore as the same task for every user. New typing techniques have been developed and in general user should be able to choose the system that better meets their needs; one example is the so called swiping, that lets input text by sliding the finger over the keys.

We could easily imagine that most of these aspects had already been explored by researches in the human computer interaction field. This was an issue that influenced so many users worldwide, and whose improvement would increase productivity and even opening possible usage scenarios for smartphones (imagine a journalist, capable of taking notes on his touchscreen device while performing an interview, as they would do on a laptop).

Nevertheless, we assumed that, because the spread of touchscreen devices is a relatively recent fact, there could exist areas that had not been completely explored yet . This reasoning led us to the choice of what would have become the main topic of our thesis.

We would also like to outline how the main author of this dissertation has been interested in the mobile environment since its very beginning, observing with particular care the evolution process that it has followed during the last recent years.

As owner of one of the earliest smartphones, carrying a Symbian system, then developer of an application for Microsoft Windows Mobile platform for his bachelor thesis, and currently possessor of a Google Android equipped phone, he directly experienced this kind of devices. He also had the opportunity to study them during his everyday life, and deepening his knowledge in the field through specialized sources. These aspects undoubtedly influenced his choice for this thesis, and will hopefully lead his future career into this area.

3.2 Topic definition

A natural way to start a research project is to look for other works related to the topic, and this is how we proceeded. As expected, a great number of papers had addressed this topic, dealing with it in many different ways; while a comprehensive description of these works can be found in Chapter 2, in this section we will discuss how these works influenced ours.

At the beginning of our investigation, we were focusing more on the study of a technique to improve word prediction and correction systems. This is a category of softwares that help the users to type by proposing them, usually on the upper part of the keyboard, a selection of words based on the last keystrokes, that are supposed to interpret what was meant to type. They usually exploits artificial intelligence techniques, specifically in the field of natural language processing, to interpret the user input as a noisy signal and try to understand the word that was meant to be typed, even if errors were performed and the word has not been completely entered.

Our expectation was to find a way to improve this category of softwares by applying some refinement to the models they use, but it soon became apparent that this could be a conspicuously hard task, due to the amount of research done in this field since the late 1940s, that apply not only to the specific problem of mobile typing, but more in general to all those that deal with natural language, e.g., speech recognition, typing with disabilities and text translation.

Thanks to these studies, that are categorized as part of the artificial intelligence, but take their cue also from psychology and linguistics, emerged that natural language can be analyzed at several levels of abstraction; the lower levels are the ones that have been widely implemented, mainly through statistical approaches, and that perform at acceptable standards for most applications [37].

In conclusion, the methods that are still under research investigation are the ones that deal with very large texts and keep in consideration specific targeted knowledge; they wouldn't then fit with a task like mobile typing, in which the sentences entered are usually short and the topic can vary often.

Given that going at a higher level would not have helped, it seemed reasonable to us to see what could be done descending through the degree of abstraction. This drove our research to topics that were more closely related to the problem of typing on touchscreen devices, and here appeared to be more space for intervention.

While papers and articles from some year ago would cover issues that were mostly resolved or did not appear to be of concern due to the devel-

opment of the matter, there appeared to be an active area of research that was proposing new approaches, some of which could potentially lead us to real improvements in the execution of the task we were interested in.

Apart from articles measuring the performances of the different available text input methods, we could define two main categories that were attempting instead to improve them through the study of specific techniques.

Redesigning the layout. This could be interpreted as the direct approach; given that typing on a touchable surface is a different task from typing on a physical keyboard (just to mention a few, only one or two fingers are utilized and the typing area is usually small sized) attempts were made in order to rethink the layout basing on these factors in order to provide a more effective one.

Adapting the layout. This is a more subtle approach, that takes advantage from the fact that a software keyboard does not need to have a static layout; keys can be rearranged, moving their centroids or enlarging certain keys using heuristics. These heuristics can exploit either models inherent to the language or personalized to the user. The adaptation can occur even online, by mean that it happens while the user is typing.

The latter category immediately aroused our interests, because it could potentially deal with artificial intelligence aspects. In fact, adaptation of the interface to the user is a topic which researchers in the AI have already dealt with; in a broad view, this means understand what the user is performing within the application, and predict what he will do in order to propose him a narrowed set of options, thus permitting him to achieve his tasks in an easier and faster way. A very famous, even if despised, example is *Clippy*, the user assistant for earlier versions of Microsoft Office, which offered advice based on Bayesian algorithms.

3.2.1 Improving typing through adaptation

The *Findlater et al.* work [9] particularly inspired us, because it was showing a new approach that could be better defined. We will analyze it deeply in this section, showing what were the points that most attracted our interest, and what in our opinion could be extended to gain even a better improvement. Finally, we will observe in the next section how we adapted their work, that relates to ten finger typing, to the same task on handheld devices, where usually typing happens through the use of one or two fingers.

Figure 3.1: Layout of the Leah Findlater [9] keyboard after adaptation

The main point of this work was to generate a keyboard personalised to the users' dynamics, retrieving entry points while they were typing and using their distribution to assess the position of the keys. Three sessions took place and performance were compared in term of WPM (Word per Minute) showing significant improvements in this typology of keyboard.

An important contribution to the field is the fact that they outlined what are, according to their analysis, the dimensions of an adaptive keyboard, that we report here since they pose the basis for our work:

Key-press classification model. This model associates a touch event on the screen to a given character; in the standard applications, this is a static function from every point of the keyboard to a letter, while in an adaptive one, the touch point is processed with other information, in this case the previous keystrokes, by a classifier algorithm.

Visual key layout. The layout resulting from the adaptation can then be showed to the user, thus giving him a visual clue of where the keys currently reside. Since classifier algorithms can result in an arbitrary layout with very irregular bounds between the keys, it is opportune, if one chooses to implement this dimension, to apply a different algorithm that maintains a visually acceptable layout.

Keyboard positioning and orientation. This dimension refers to the positioning on the screen of the keyboard itself, rather than that of the individual keys. While this seems more something to deal with on larger screens, interesting results showed that even in mobile phones a keyboard placed right below the input text can perform better than the one we are

used to placed in the lower part of the screen [44].

The reasons why we did not include adaptation at the visual level are based on various observations.

Firstly, the screen of a smartphone is considerably smaller than one of a tabletop, thus leaving fewer space for adaptation, that could have resulted in a messy layout confusing for the user; in addition, as stated in the previous paragraph, we aimed to obtain a model that could adapt not only to the user keypresses, but also to the external factors that influence the typing experience. This could have resulted in extremely rapid changing in the layout that would disorient the users.

In previous works that experimented with this aspect, results did not show considerable improvements in terms of quantitative measures, and even the work being reviewed in this section showed the same trend, performing worse than the standard non adaptive keyboard. Also qualitative measures acquired by interviewing the users reported a generally negative trend.

The position of the keyboard, as we noted previously, has been explored in other works with interesting results, but appeared to us out of the scope of our thesis.

Then, having chosen a way to explore, we started focusing on what could have been improved when performing such approach, and we noticed that some assumptions had been taken during the research, based on intuitive observations but implemented in a naive way.

For example, they point out how, in their adaptive keyboard with static visual layout, it appeared necessary to them to deal with this specific event: the user is looking carefully at the keyboard, so that he knows exactly where to position his finger.

The problem of an adaptive keyboard in a situation like this is that, having adjusted its layout, it may not output what the user expect when he touches the surface, and this could lead the application to behave unpredictably: an interface should respond to an action of the user with what the user expects to happen. They overcame to this issue by simply setting a timeout of one second; after measuring such time of inactivity, the soft keyboard would disable its personalized layout to let the user tap with more accuracy its first letter. The same assumption and relative solution were applied to the similar situation of when the user pressed the first key after having performed a correction.

This was the first clue that suggested us that this was an early approach to the matter, and that a naive approach like that could have been better defined putting more analysis on it.

3.2.2 Situational impairments and their influence on touch typing

If it is true that every user has its own pattern and it is a good approach to try to learn it, on a mobile environment there are other variables that influence the way the user types. We have previously summarized this concept with one term: the situation. In fact, while a tabletop is supposed to be placed in a given environment, and the way the user interact with it is always the same, owners of smartphones use their devices in many different surroundings.

By situation, we intend all the variables that could affect in a certain extent the user behavior while typing. Our analysis, based on the one from *Barnard et al.* [2] splits these variables in two main categories:

External environment. The world influences our actions, and so can influence the typing experience. Probably the most conditioning variable is the light, that can interact with the typing experience in many ways: a strong source of light will cause to the user difficulty to discern where he is touching and what is he typing, and if that light is pointed directly to the screen, it will be almost impossible to discern what is shown on it. But this is not the only environmental cause, lower temperatures affect the sensibility of the fingers, thus causing a less sensible feedback of touching.

In this category are included also causes that are not strictly natural, as the fact of being traveling on a car or on a train, whose vibrations will decrease the user accuracy in pressing the intended key.

Human environment. The user itself can influence his typing in different ways. With this category we identify all the factors that depend from the user choices, attitude and social environment.

If he needs to complete the task of writing while being in a hurry, and he does not need to write an error free document, he will type faster, disregarding of the loss of accuracy.

It is also possible that he is accomplishing different tasks not related with the phone; one meaningful example is to walk while writing a text, thanks to the mobility feature of the device, or it may be the case that he is driving, even if in most of the countries this attitude is illegal, or talking with an interlocutor. This subcategory is often referred to as *divided attention impairments*.

The way the user holds the phone, or the number of fingers used, are just a user choice based on how he feels more comfortable, but could change

depending on the device they have in hand.

Also the mood takes its place in this category; if a user is very tired he might be unable to put the same attention to its task than he usually does, or if he is distracted by some external factor, e.g., he is watching a movie, he may be tempted to type without staring at the screen the whole time.

All the factors that we outlined, and many more, take their part in influencing the act of typing, but they do it in different ways, so as first approach we tried to define a variable that could summarize them, but we left this task to be accomplished when we would have had some data to analyze.

Another aspect that the reader could have already noticed is that most of these ambient variables that define the situation are not directly known neither to the device nor to the typing application; for example, is impossible to know whether the user is tired without explicitly asking him, but this approach would imply to make him fill a long survey before he could start typing even just a few words, leading to an overly complex system that would not be judged as usable.

Fortunately, smartphones come equipped with a variety of different sensors that, with the opportune software, can be exploited to measure variables of the external situation, and subsequently infer aspects from it.

For the internal situation, analyses on typing dynamics could potentially lead to models capable of inferring it.

3.3 Improving text input through adaptation to the situation

Our goal, from a high level perspective, is to improve the text input on touchscreen devices. But there may be different ways of perceiving such improvement; does it mean that the user is capable of typing faster? Or that he may perform less errors, thus feeling more accurate? Or could it be simply enough to make him feel like he is achieving these results without being there an effective and measurable improvement?

In our opinion, all of these aspects have their weight when dealing with typing enhancement; the first mentioned measure is probably the easiest to quantify, and the one that would seem more appealing to aim, but it would not be enough without keeping in count that is very likely that a higher speed, if we don't take proper countermeasures, will result in a higher error rate. Finally, one has to keep in count also how the user will perceive its experience, being this one of the key principles of the design of a human

computer interface.

Speed of typing can be measured by mean of characters per second (cps) or words per minute (wpm). Speed can not be seen as an user independent parameter because, besides of the situation, it can vary enormously from one user to another. Bigger thumbs will force the user to be slower, and studies have been performed to see if even the age of the subject can influence its speed [55].

We stated as hypothesis that the error rate, besides being user independent, is also influenced by the situation, as defined in Section 3.2.2. Furthermore, we believed that the higher the error rate is, the less the user will feel whether an automatic correction tool is aiding him.

It is also noticeable that the state of the art correction tools already perform well in aiding the user by interpreting his touch at word level, making it seem like our work would not be useful.

We are of the opinion that a tool such ours, working at the keystroke level instead of the language one, could be used concurrently with a word prediction system to improve its performance; additionally, while it is true that the mentioned tools correct the typing, they do it only when the user has typed the whole word, thus showing its incorrect spelling while he is still typing it, instead our work would permit to show more likely characters even during the act of inserting the word.

While it may seem that correcting the user input is a good practice, it may not always be the case, because, since it is impossible to always infer the correct letter (in which case, the system could write in place of the user), the system would eventually correct typos that were not actually performed, and this could lead the user to frustration because he would feel that the system is not behaving as he expected.

We assumed that, if the user is typing in a more inattentive way, he will simultaneously perform a greater number of errors, and he will notice less that the system is correcting him with letters that in its "opinion" are wrong.

Asserting our goal in a single statement, would then sound as: build a platform capable of inferring the user intended keypresses without letting him notice it.

Chapter 4

Design of the Work

In this chapter we analyze our research from a logical perspective. We describe the design choices that brought us to intermediate results, and how these influenced the following steps. For all the techniques that we exploited, we will give a short theoretical introduction, and then we will describe how we set them to solve specific parts of our work, to finally show the results that we obtained.

4.1 Ethical aspects in research involving human subjects

During the design process we had to deal with the ethical issues related to performing a research involving human subjects. In the United States, this kind of research is important and is dependent on the approval from the Office for Human Research Protection (OHRP).

This process is necessary after the Belmont report, [12] where it defines the ethical issues that must be kept when performing clinical and social/behavioral research conducted on humans. The main belief of this act is that any research should treat human subjects with respect, beneficence, and justice. Not overstep their rights despite of the improvements it could bring to a much larger population.

Since some investigators who performed research were biased, and to further their quest for knowledge, they often posed their test subjects to risks, as a result, specific offices were instituted who are responsible of evaluating and assessing the risks of participants, with particular attention to vulnerable populations such as children and prisoners.

In our specific case, after taking the required course for social/behavioral

research investigators, we designed our research so that we would not involve hazardous situations while performing the tests. Special attention was given to the retrieval of data in order to keep the anonymity and all the personal information was kept in a secure place.

Another relevant point was designing the subject recruitment in a way that it would not focus to specific populations involved in the research, as it would have been discriminating.

A protocol of our research was submitted to the Institutional Review Board who gave us the permission to proceed with our work, as it was exempted from risks involving the subjects.

4.2 Data retrieval session

A crucial part of the research was the designing of the first prototype that would be used to retrieve the data. It evaluated the aspects that we wanted to explore and that, once fixed, could not be changed.

We opted to develop our sample application for the Android platform, because it is the only one as of the date of publication, that permits us to substitute the software keyboard from the one provided by the system. For our research, we built a simple study application, able to produce text only within itself and therefore not usable as an everyday keyboard. We liked the concept and consider the possibility for future enhancements in order to be a usable application.

For testing, we had available a Sony-Ericsson Xperia Neo device, running the Android 4.0.1 Ice Cream Sandwich operative system. Its key features were a Qualcomm Snapdragon MSM825 1 GHz processor, an Adreno 205 GPU and 512 MB RAM. The screen was a 3.7 inches capacitive touch screen with a resolution of 480x854 pixels; even if the screen size was higher than that of the iPhone (3.5 inches), it was a wide screen resolution with a ratio of 16:9, thus bringing the screen width very close to the iPhone one, that carries a 3:2 aspect ratio screen: 57 vs. 58.6 millimeters.

In order to provide a consistency with this selection, we decided to implement the same layout of the standard Android keyboard, thus keeping the same key disposition and proportions. This layout differs very slightly from the iPhone one, that is a little higher, thus covering the most commonly used keyboards (even if iPhone users pointed out that they were feeling a change in the layout that could have affected their typing).

We wanted to simulate a typing experience similar to the real one. For this reason, we used the feedbacks that are commonly used on devices: a visual feedback was displayed over the currently pressed button, a label

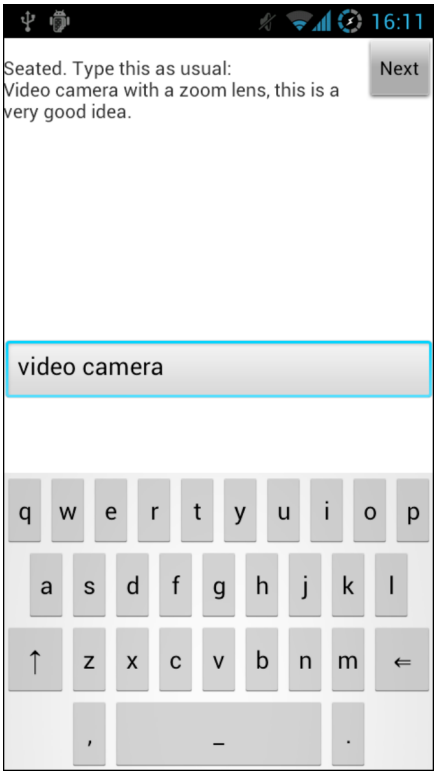


Figure 4.1: A screenshot of the data retrieval application

showing the character, and the haptic feedback was performed when the screen was pressed to validate the user touch. The keyboard was behaving exactly as a standard, non adaptive keyboard, and users could correct their input through the backspace key.

The test sentences were displayed on top of the screen, and the user could continue through different sentences and tasks by pressing the Next button. Phrases were sampled from *Scott MacKenzie et al.* set [39], especially designed to evaluate text input techniques. Its key features were ability to remember the sentences, moderate length and representativeness of the target language. This choice permitted us to achieve two goals: first, it would have been harder to ask the user to type free text, since we needed to know what was the intended input in order to subsequently classify it, and second we could avoid secondary effects such as pondering.

A crucial point was the choice of the tasks that user would perform. We had available a wide variety of different possible contexts in which the user could find themselves typing in normal smartphone usage, so we accurately chose them by evaluating the following features:

Relevance. Often users can find themselves typing in very different environments, such as riding a train, driving a car or walking. They also have different postures such as standing, sitting or even laying in bed; we wanted to evaluate the most common ones. Furthermore, we had to restrict our analysis to the ones that could have reproduced in an observed environment, such as our laboratory.

Quantifiability. Even though it would have been interesting to analyze the dynamics of typing in contexts that had never been explored, such as users affected by divided attention, e.g., carrying out a conversation while typing, we decided to limit our test cases to the ones that we had a chance to measure, either by typing dynamics or by exploiting the phone's sensors.

We chose four test cases, that would compose the retrieval experiment sessions:

1. **Normal condition.** Users were asked to type normally while seating on a chair, taking care of the correctness of the input and eventually performing corrections.

2. **Walking.** Users walked through the laboratory, without following a given path but being asked to perform variations in their trajectory. As already showed by previous works [15, 3], walking can be detected through the device built-in accelerometer.

3. **At high speed.** Users were asked to type at the maximum speed they could achieve. We decided to observe this case because we expected that user would type faster on an adaptive keyboard, so we wanted to anticipate the effects of this factor.

4. **With a glare on the screen.** A flashlight was pointed directly on the screen, thus limiting the visibility of the rendered keyboard. As far as our knowledge, only few works dealt with this variable [2], and no one inspected the effects of typing in such condition.

A factor that we did not directly keep in count was how the user was holding the phone while typing. While this has been showed to be a determinant element of the user pattern [1], and can be inferred through sensors [16], we aimed to observe the natural user behavior, so we instructed them to hold the phone the same way they were used to in most common condition, and we subsequently adapted the model to their choice.

4.3 Selection of the relevant variables

Our application registered all the information regarding each key press of our evaluation study in a log file stored on the device.

The raw information stored was: the position and timestamp in which the user pressed the screen, intermediate positions if the finger was slided during the keypress, the position and timestamp of when the finger was released, and for convenience the selected key. Additionally, the touch size was registered but was subsequently discarded since it was presenting too much noise due from the inaccurate sensor that was more suited for longer presses, where one can average through measurements.

We loaded this information on a persistent database, we classified it by assigning to each keystroke the intended letter. This step was performed manually, by scanning the data and interpreting it; only keystrokes that hit a key next to the intended one instead of the correct one were classified, and the others were discarded. We opted for a manual approach because we observed that the impairments that we were experimenting on introduced a big variety of errors not depending on the accuracy, such as missed and swapped strokes that an automatic system would have had difficulty to spot.

Since in the next subsections we analyze the data discriminating it within the different observed situations, we define the set of our use cases as

$$S = \{n, h, w, g\}, s \in S$$

where n stands for *normal condition*, h for *in a hurry*, w for *walking*, g for *with glare* and s is the generic session.

4.3.1 Assessing the typing speed model

As mentioned in Section 2.5, no work has been performed that keeps in account the user typing speed to model its accuracy, so we needed to denote a model capable of inferring the speed at which the user was typing, in order to relate it to his accuracy.

The average typing speed of a subject in each session can be measured in characters per second as

$$v_s = \frac{1}{\bar{t}_s + \bar{p}_s}$$

where $\bar{t}_s$ is the average time elapsed between two subsequent strokes, referred to as inter-stroke time, and $\bar{p}_s$ the average time the finger was touching the screen for each keypress, referred to as pressure time.

Figure 4.2 shows the typing speed for each subject. The lowest three values were retrieved by users typing with only one thumb, while the remaining ones are from users typing with both thumbs.

Even if we can infer that typing using only one finger is slower, one can easily see that there is a big discrepancy between the values, showing that the typing speed is a property relative to the user, as it was suggested by *Robinson et al.* when analyzing physical keyboard dynamics [51].

We needed a way to model this relative speed, thus we observed that the pressure time $\bar{p}_s$ was roughly constant over all sessions, and could then be interpreted as a user static property. Performing a regression over it would have permitted us to estimate the user preferred speed.

Figure 4.3 shows the pressure times of the subjects in relation to their inter-stroke time observed when they were typing in normal condition; there is an observable correlationship between the two values, that we decided to estimate with a linear regression technique, specifically Weka's linear regression, that computes the least squared error regression line [22]. In our opinion, while it is true that this might not be the most accurate model, given the small amount of data at our disposal, we believe that a more advanced regression would have had the risk of overfitting the data.

The resulting linear function that we obtained was

$$\bar{t}_n = 1.20\bar{p}_n + 93.85\,.$$

Thanks to this model we were able to estimate simply from the typing dynamics, and specifically by its timings, whether the user was typing at his average speed, the one that he reputed to be more convenient when he was asked to type normally. Since the pressure time was observed to be constant for a given user, if we observed a value of the inter-stroke times that was smaller than the one predicted by the linear regression equation, we could infer that he was typing faster than is usual, and we could consequently adapt the keyboard to that situation.

Defining an analogous model for the subjects typing with only one thumb was not straightforward, because we only had three observation. We observed that one hand typists were showing inter-stroke times about 1.5 times greater than two hands typists, so we multiplied the previous equation by that factor, obtaining

$$\bar{t}_n = 1.80\bar{p}_n + 140.87\,.$$

This equation is based on an assumption made on the few observations that we had at our disposition, and has no statistical meaning; nevertheless, we needed a model also for the subjects typing with one hand, and this one

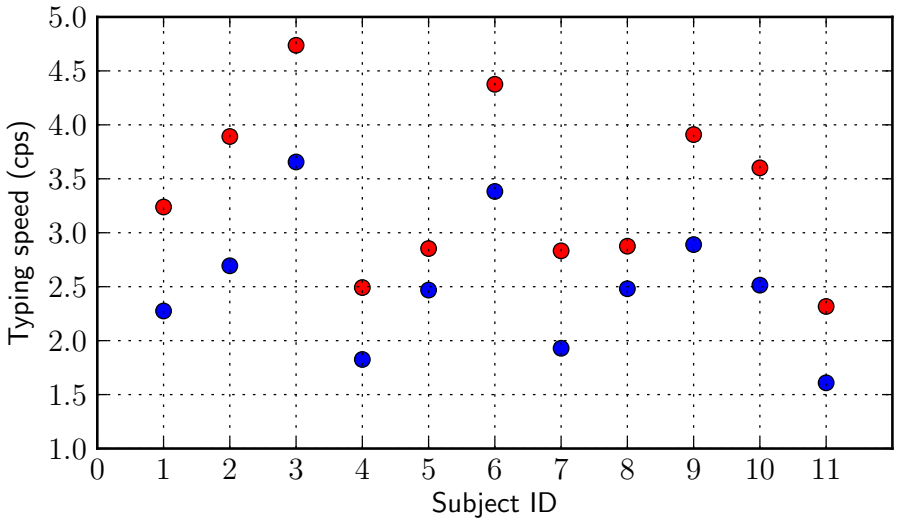


Figure 4.2: The subjects average speed for (blue) normal, (red) in a hurry sessions

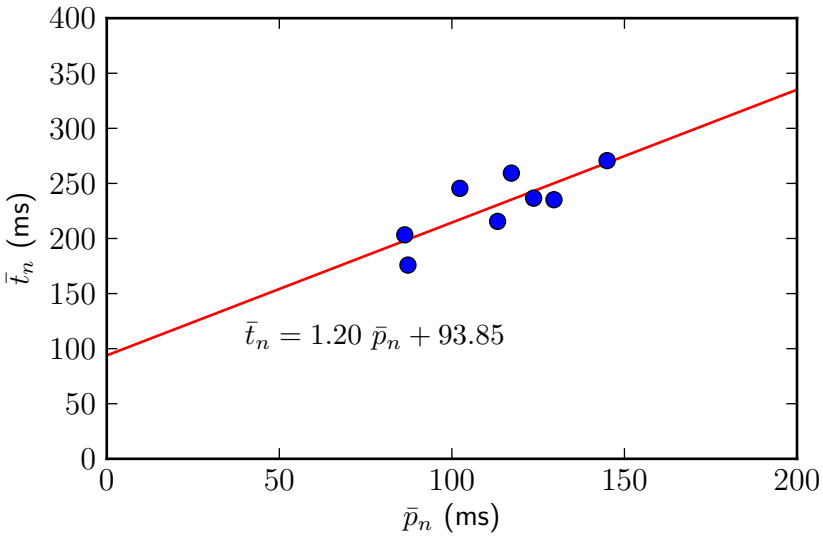


Figure 4.3: Subjects average pressure and inter-stroke times for users typing with both thumbs in normal condition and the relative regression line, with $\rho = 0.80$, $MSE = 17.46$

showed to correctly fit our data. In order to produce a generalizable model, we would need to observe more one hand typist dynamics.

4.3.2 Intra-stroke movement

On a mobile system, when the user hits the display on a given point what usually happens is that the graphic element underlying the touch point is selected; then, if the user slides his finger outside the boundary of that element, it will become deselected and releasing the finger from the screen will not produce any action. As an analogy, one can think of the same behavior to happen when clicking a button with the mouse pointer on a desktop environment: if the user did not intend to press that button, he can still move the mouse away from it before releasing the click physical button, and the action will be canceled.

While this fact is true for many interfaces, it does not hold when interacting with a soft keyboard, where this behavior would be frustrating for the user, that does not receive any output after having pressed a key, eventually leading to a series of missing keystrokes.

All the (non-adaptive) keyboards that we inspected select as character for the text input field the one that was under the perceived touch point in the moment the user ends it by releasing his finger. From our hypotheses, this could have been either a good choice, permitting to the user that realizes to have just hit the wrong location to *adjust the shot* and slide his finger to the adjacent one, or a bad feature, because involuntary sliding of the finger, that can be as well caused by the excessive sensitivity of the touch sensor, could lead to wrong key classification.

We decided to explore this factor taking at first a naive approach. We restricted the analysis to the keystokes k in which a the location of first touch $\mathbf{l}_k^f$ differed from the one of release $\mathbf{l}_k^r$

$$k \; s.t. \; \mathbf{l}_k^f \neq \mathbf{l}_k^r$$

and for each of them we simulated which would have been the misclassification rate if instead of using the last retrieved touch point, we would have used the first; results from this simulation showed that the number of errors would increase, and even relaxing this assumption by using the travel midpoint

$$\mathbf{l}_k^m = \frac{\mathbf{l}_k^f + \mathbf{l}_k^r}{2}$$

would lead to a smoother, but still negative effect. Results of this simulation are shown on Table 4.1, where for each session is reported the ratio of

	Error rate using		
Session	Release	Middle	First
n	0.04	0.08	0.10
h	0.12	0.16	0.19
w	0.18	0.16	0.14
g	0.20	0.22	0.26

Table 4.1: The misinterpreted character rate on keystrokes in which finger sliding occurred, when classifying them with the the release point, medium travel point and first touch point, for each session

	Errors		
Session	Initial	Fixed	Introduced
n	110	48	219
h	334	161	387
w	324	144	256
g	306	101	184

Table 4.2: The number of errors present classifying the keystrokes with the last retrieved location, and the ones that get correctly interpreted versus misinterpreted using the first retrieved location

erroneous keypresses over presses experiencing finger travel; the first column can be interpreted as the error rate caused by involuntary sliding the finger over an adjacent key with respect to sliding that ended on the intended key.

Further inspection showed that these simulations, despite misclassifying a greater number of errors than the normal model, were nevertheless correcting a conspicuous set of mistaken presses, thus confirming our hypothesis that, while it is the case that many errors are performed due to unnoticed finger sliding over a neighbor key, this feature is more often used to select the correct key when the user realizes he posed his finger on the wrong one (as it is in effect designed any point-and-click interface).

As shown in table 4.2, almost half of the errors are caused by this wrong behavior, problem is that blindly applying a different classification that uses the either the first or medium point instead of the last would also include a bigger number of errors than the one corrected.

If we build a model that is capable of classifying the involuntary finger slides from the voluntary ones, this could decrease the error rate, but with the time and resources at our disposition we were not able to find a feature and a technique that could aid in spotting this behavior.

Despite we think this is an interesting problem that would need deeper

Session	Error rate	S_x (px)	S_y (px)
n	0.05	15.11	13.94
h	0.12	22.20	17.60
w	0.14	19.37	16.45
g	0.23	26.77	20.82

Table 4.3: The aggregated error rate over the sessions, with relative sample standard deviations with respect to the center of the key (in pixels)

inspection, we decided to leave it for future works, because we did not have the necessary resources and time to do that, and our main goal was to build an adaptive keyboard.

We nevertheless made use of the results of this analysis, selecting the touch release location as the valid variable to use in input for our classifier.

4.3.3 Error rate as a function of the situation

Probably the most strong hypothesis that we posed when defining our work was that varying the situation in which the user was typing, he would behave differently, showing a different pattern that could have been exploited to build a better classifier model.

While in the previous sessions we showed that this should be the case, with different experiments showing consistently different values, we wanted to deeper inspect how our superimposed situational impairments would impact on the user performance. The most direct way to measure it is undoubtedly the error rate, i.e., the ratio of keystrokes that the user hit on a character. This analysis was run independently from the fact that the user corrected or not the mispelled character, thanks to the fact that we were keeping trace also of deleted insertions.

Table 4.3 shows the observed error rates, and their relative sample variances over the two axis. As one can see, variances over the horizontal axis are higher than over the vertical, thus expecting that most of the error will select a key laterally adjacent to the intended one, rather than one on a different row.

The last session was probably the most critical, with almost one mispressed keystroke over four, showing that the inability to observe the screen makes the user type very loosely.

Furthermore, it is easy to spot that the error rate increases when introducing situational impairments, partially confirming the hypothesis of different patterns caused by variation in the user experience; specifying that

this is a partial result was needed, because we had still to investigate *how* these patterns changed, and whether it was possible to model such changes.

This made our hypothesis of treating the different situations differently, handling the ones that showed a higher error rate with a stronger correction algorithm.

Notation

In the following sections are discussed several techniques that are based on probabilistic rules; we define here the notation that will be used throughout the next sections.

Given a stochastic variable X, we denote by $\mathbf{x}$ the possible values it can assume, by x a specific value. For $P(X\dots)$ we mean a specific probability value, and for $p(X\dots)$ a probability distribution[1] over the variable(s) specified within the parenthesis.

To keep track of the time, we introduce the subscript $._t$ to indicate the tth instance of the variable. Generally, the cardinality of the timesteps is identified by k, thus $._k$ is the last element of the sequence.

We denote with α the normalization constant. This appears in those expressions in which a constant term should appear, and is instead stripped off thanks to the fact that the values resulting from the distribution must sum up to 1. α can be computed knowing all the values of the distribution after the expression has been evaluated.

Finally, we mean by *prior* distribution the distribution over a variable in absence of any evidence, and by *posterior* the one given a set of evidence (observed) variables.

4.4 Shaping the keyboard

Once that we proved that the situation was effectively influencing the users typing dynamics, and we had some insight on how that was happening, we needed to define a technique that would be able to define an improved keyboard.

Formally speaking, a keyboard can be seen as a function f that, given a sequence of strokes, returns an equivalently long sequence of characters according to its internal model. For simplicity, we will consider for now a static keyboard, i.e., classifying one stroke at a time independently from the

[1] a probability distribution is a function whose summation of the values it assumes over the range it is defined (the integral if continuous) is one

others, and the stroke as the last retrieved point over the touchscreen

$$f(\mathbf{s}) = c$$
$$\mathbf{s} \in \mathbb{R}^2, \quad c \in \{a, b, \dots\}.$$

To be precise , the domain of $\mathbf{s}$ should be over integers, since measures of it are provided by the system in pixel unit measure, the highest available resolution; but we will see shortly how modeling it as a real numbers pair permitted to define more sophisticated models.

This static model can be also seen as a mapping between the locations on the screen and the available keys. Our goal is to search over the defined function space the one that minimizes the number of discrepancies between the classified character and the user intended one i

$$f = \arg\min_f \#\big(f(\mathbf{s}) \neq i\big)$$

where # is the function counting the number of occurrences.

Unfortunately, this is just a theoretical lower bound; since the problem of stating which keyboard will perform best in any circumstance is undecidable, we will again make a simplification and state that we are looking for a function that is the result of a compromise between its ability of fitting the data we have available and its capacity of generalizing over unseen data.

In order to have quantifiable results, in the following we will compare the performance of our models to the non-adaptive keyboard with classification bounds correspondent to the visual ones, referred to as classical keyboard, or $\hat{f}$, in terms of improvement in the error rate

$$\frac{\#\big(\hat{f}(\mathbf{s}) \neq i\big)}{\#\big(f(\mathbf{s}) \neq i\big)}.$$

The reader may note that the inverse of this expression is not the percentage of corrected errors, because new errors may have been introduced by the model that were not present with the classical keyboard.

4.4.1 Choice of the learning algorithm

Once we defined our problem as an instance of the general problem of learning a function, many machine learning techniques were available for this scope. Some of them are really general, and can be applied to any problem, others require refinements over the assumptions in order to be executed; in this section we will show results achieved testing some of the techniques

within the first category, and then motivate the choice for a more specific model.

The models that we selected were the following three, each with its own characteristic features that we will briefly explain.

Tree classifier. Use a decision tree as a predictive model which maps observations about an item to conclusions about the item's target value. In these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Tree classifier are simple to learn: at each step, the feature that permits to discriminate a greater number of instances within their class is selected, and then the nested subtree is constructed greedily. The output model is also lightweight, in that it classifies an instance in logarithmic time with respect to the number of nodes in the tree.

Neural network. Are composed of artificial neurons with weighted interconnections, aiming to abstract away the complexity of how biological neurons work, still maintaining a similar behavior. Usually it implements one or hidden layers of neurons between the input and the output layer, and learns the weights with a back-propagation algorithm, that takes non-polynomial time. Even if it is in general hard to abstract from the obtained model the properties of the system in study, they perform well on a large range of scenarios.

Multivariate normal classifier. This is a probabilistic model that describes the classes with a multivariate normal distribution, with parameters to be determined. Learning of such model usually requires exploiting statistical techniques, such as parameter estimation. Once each class has its own distribution fully specified, the one showing higher probability given the instance to be classified is selected.

Table 4.4 shows the performance improvement obtained by these three algorithms when asked to estimate the function from the touch location to the intended key, with respect to the classical keyboard.

Results of tree and neural network classifiers were retrieved using Weka algorithms, respectively J48 and Multilayer Perceptron, while the last column was computed using the algorithm that we will define in the next sessions.

As one can see, tree classifier was the best performing, but we had to discard it because it was showing one of its greatest issues, that is it was

Session	Tree	NN	MultiGauss
n	1.36	1.07	1.21
h	1.25	1.13	1.16
w	1.25	1.19	1.12
g	1.28	1.21	1.10

Table 4.4: Performance of the three tested algorithms in classifying the intended key over each session

overfitting the function by estimating it pixel by pixel. Since the whole keyboard was approximately 300×400 pixels, it was very likely that one pixel had only one observation, so the algorithm was choosing the key correspondent to that observation even if all the surroundings were attributed to a different key.

Having to choose between neural networks and a probabilistic model, we opted for the second for two important reasons: first of all neural networks are defined as black boxes models, where is difficult to evaluate the choice it makes because of its internal complexity, on the other hand, probabilistic models permit to exploit all the mathematical properties of the probability framework, thus letting us expand our model as we will show in the next sections.

To further enhance the achievable improvement, we also decided to consider a personalized model, that is, training the model separately for each user to get different parameters defined for each of them. From now on, all the results shown are the average over the different model of each subject, rather than the performance of a single model that tries to classify them all.

4.4.2 Bayesian inference

The goal now was to estimate consistent parameters for the latter model described in previous section. We assumed that the touch location L was generated from a bivariate Gaussian distribution over the two possibly correlated aleatory variable X and Y

$$L \sim \mathcal{N}_2(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are respectively a 2×1 vector and a 2×2 positive definite[2] matrix representing the mean and covariance of L

$$\boldsymbol{\mu} = \begin{bmatrix} \mu_X \\ \mu_Y \end{bmatrix}, \quad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2 \end{bmatrix},$$

$$\rho = \frac{\operatorname{cov}(X,Y)}{\sigma_X\sigma_Y}.$$

The density function of the p-variate is basically a generalization of the univariate using matrices

$$f(\mathbf{x}) = \frac{1}{(\sqrt{2\pi})^p}|\boldsymbol{\Sigma}\,|^{-1/2}\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right).$$

The most used estimator for the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ is the maximum likelihood estimator.

It is obtained by maximizing the likelihood function, that represents the joint probability to observe the data, given the parameters. The idea underlying this process is to choose as estimate the parameters that provide the highest likelihood to the data $\mathbf{x}$ representing our observed training keystrokes (not to be confused with $\mathbf{x}_i$, which represents a single observation); this provides unbiased estimators for many families of probability distributions, as it is the case for the multivariate normal, where the likelihood is given by

$$p(\mathbf{x} \mid \theta) = \prod_{i=1}^{n} f(\mathbf{x}_i \mid \theta) = \mathcal{L}(\theta \mid \mathbf{x}).$$

The logarithm of the likelihood can be expressed in closed form as

$$\begin{aligned} \ln\mathcal{L}(\theta \mid \mathbf{x}) &= \sum_{i=1}^{n} \ln f(\mathbf{x}_i \mid \theta) \\ &= -\frac{p}{2}\ln(2\pi) - \frac{1}{2}\ln|\boldsymbol{\Sigma}| - \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \end{aligned}$$

and we are now interested in obtaining the value that maximizes it

$$\hat{\theta}_{MLE} = \langle \widehat{\boldsymbol{\mu}}_{MLE}, \widehat{\boldsymbol{\Sigma}}_{MLE} \rangle = \arg\max_{\theta} \ln\mathcal{L}(\theta \mid \mathbf{x}).$$

Maximizing $\mathcal{L}(\theta \mid \mathbf{x})$, that is the same that maximizing its logarithm, thanks to the monotonicity property of the same, gives the two unbiased

[2] an $n \times n$ real matrix $\mathbf{M}$ is said to be positive definite if $\mathbf{z}^{\mathrm{T}}\mathbf{M}\mathbf{z}$ is positive, for any non-zero column vector $\mathbf{z}$ of n real numbers.

estimators for mean and covariance

$$\widehat{\boldsymbol{\mu}}_{MLE} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i\,,$$
$$\widehat{\boldsymbol{\Sigma}}_{MLE} = \frac{1}{n-1}\sum_{i=1}^{n}(\mathbf{x}_i-\overline{\mathbf{x}})(\mathbf{x}_i-\overline{\mathbf{x}})^{\mathrm{T}}\,.$$

Unfortunately, these estimators have a good confidence over the values that are estimating only for big samples (cardinality is usually set over 50) because they depend only on the retrieved data.

In our case, we had to estimate the distribution over each key and each subject over each session. Each of these sets contained roughly 500 stroke instances, so an average of 16.1 for each key, but unevenly distributed because of the property of the English language of showing with higher frequences certain letters, e.g., vowels, with respect to other letters such as q and z, of which we had respectively an average of 1.7 and 1.5 occurrences per set; Table **??** in Appendix B shows this information for all the keys.

To overcome this issue, we decided to put a prior knowledge over our data; in fact, maximum likelihood estimator are designed to estimate parameters that are completely unknown. In a keyboard, we would expect that users will try to hit a key roughly at its center, with a variance inversely proportional to their accuracy.

A prior knowledge $p(\theta)$ is represented as a probability distribution over the parameters of the density function; from Bayes' theorem

$$P(A \mid B) = \frac{P(B \mid A)\,P(A)}{P(B)}$$

we can thus express the distribution over our parameter given the observed data, usually referred to as posterior distribution, as product of the likelihood function and the prior, normalized (divided) by the probability of the data $p(\mathbf{x})$

$$p(\theta \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \theta)\,p(\theta)}{p(\mathbf{x})}\,.$$

usually, the denominator p(x$p(\mathbf{x})$ does not need to be computed explicitly and can be treated as a normalization factor, since it is independent from θ, so that it can be determined after the nominator has been computed as the value that makes the function integrate to one

$$p(\mathbf{x}) = \int_{-\infty}^{+\infty} p(\mathbf{x} \mid \theta)\,p(\theta)\,\mathrm{d}\theta\,.$$

Now, since the prior distribution is conjugate prior of the likelihood [42], we do not need to integrate to calculate the resulting posterior $p(\theta \mid \mathbf{x})$, because it will be in the same form with parameters update. In our case, since the likelihood $p(\mathbf{x} \mid \theta)$ has multivariate normal distribution, its conjugate prior is the normal-inverse-Wishart distribution

$$p(\theta) \sim \mathrm{NIW}(\boldsymbol{\mu}_0, \kappa_0, \boldsymbol{\Psi}, \nu_0)\,.$$

The parameters of this distribution will be referred to as hyperparameters, to distinguish them from parameters of the underlying model $\boldsymbol{\mu}\,, \boldsymbol{\Sigma}$. $\boldsymbol{\mu}_0$ represents the mean of the prior, $\boldsymbol{\Psi}$ its covariance matrix, and $\kappa_0\,, \nu_0$ their respective degrees of freedom; intuitively, they can be seen as the number of pseudo-observations that defined $\boldsymbol{\mu}_0$ and $\boldsymbol{\Psi}$, or also as how much informative our prior is.

As previously mentioned, the posterior probability of the parameters after observing a sequence of samples $\mathbf{x}$ of length n is still a normal-inverse-Wishart with updated hyperparameters

$$p(\theta \mid \mathbf{x}) \sim$$
$$\mathrm{NIW}\left(\frac{\kappa_0\boldsymbol{\mu}_0 + n\bar{\mathbf{x}}}{\kappa_0 + n}\,, \kappa_0 + n\,, \boldsymbol{\Psi} + \mathbf{C} + \frac{\kappa_0 n}{\kappa_0 + n}(\bar{\mathbf{x}} - \boldsymbol{\mu}_0)(\bar{\mathbf{x}} - \boldsymbol{\mu}_0)^{\mathrm{T}}\,, \nu_0 + n\right)$$

where

$$\bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i\,,$$
$$\mathbf{C} = \sum_{i=1}^{n}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^{\mathrm{T}}$$

are the sample mean and sample covariance.

After the update, we can use as estimation of our parameters the mode of the obtained distribution, as it represents the maximum a posteriori

$$\hat{\theta}_{MAP} = \langle \widehat{\boldsymbol{\mu}}_{MAP}\,, \widehat{\boldsymbol{\Sigma}}_{MAP} \rangle = \arg\max_{\theta} p(\mathbf{x} \mid \theta)\, p(\theta)\,.$$

Since normal-inverse-Wishart encodes a normal, and the mode of a normal coincides with the mean, we have

$$\widehat{\boldsymbol{\mu}}_{MAP} = \mathrm{Mode}(\boldsymbol{\mu}) = \mathbf{E}(\boldsymbol{\mu}) = \frac{\kappa_0\boldsymbol{\mu}_0 + n\bar{\mathbf{x}}}{\kappa_0 + n}$$

while the covariance can be computed as

$$\widehat{\boldsymbol{\Sigma}}_{MAP} = \mathrm{Mode}(\boldsymbol{\Sigma}) = \frac{\boldsymbol{\Psi} + \mathbf{C} + \frac{\kappa_0 n}{\kappa_0 + n}(\bar{\mathbf{x}} - \boldsymbol{\mu}_0)(\bar{\mathbf{x}} - \boldsymbol{\mu}_0)^{\mathrm{T}}}{\nu_0 + n + p + 1}$$

where p in our case is 2, because we are analyzing the bivariate case.

4.4.3 Model validation

In the previous section, we defined a technique to estimate the parameters of a the bivariate normal that would model our keyboard. As mentioned, we opted per an a posterior estimator because we did not have enough data, but we could formulate assumptions over the distribution.

These assumptions came from a previous work of *Shiri Azenkot and Shumin Zhai*, who investigated the patterns of different users typing on a keyboard similar to ours (only with a few key missing), asking them to type sentences from the same set using different hands postures [1].

Compared to ours, they had ten times more subjects, and their experiment included more sentences, but this is explainable by the fact that they aimed to provide generalizable knowledge, rather than a user specific improvement.

They published a paper showing aggregated skews and variances of each key for the different postures, basically proving that there exist methodical skews when the user was typing, and that they are influenced by the posture; two handed typers tend to move their fingers to the relative edges of the screen, while one thumb typers struggle to reach the opposite edge. On both cases distributions are slightly skewed towards the bottom of the keyboard.

The part of their results that is of our interest is showed in Figure 4.4. We used the information encoded in that image to define our prior knowledge for the mean of each key, while for the variance we used their average standard deviation for all key, assuming a zero covariance, even if in their work, and as we will show later also in ours, there appeared to be some light form of correlation between the touch coordinates.

Having defined $\boldsymbol{\mu}_0$ and $\boldsymbol{\Sigma}_0$, we needed to assign a value to κ_0 and ν_0, that would reflect how much confident we were on the previous values.

Since we aimed to reshape the keyboard with few observations available, we decided to put a weak prior on the mean. It would still have been useful to have the prior since it would not allow keys with with fery few observations (in the worst case, only one observation and much distant from the center of the key) to influence too much on the model, and at the same time would let keys with many observations reshape according to them. Furthermore, we were aware, from how we built our dataset, that data could not be excessively noisy, because only keystrokes falling in the adjacencies had been considered.

An opposite consideration was made for the prior to set for the variance. We did not any have control on how the strokes were distributed within a single key; when there were only few observations, whether they were close this would have resulted in a very low, eventually close to zero, variance

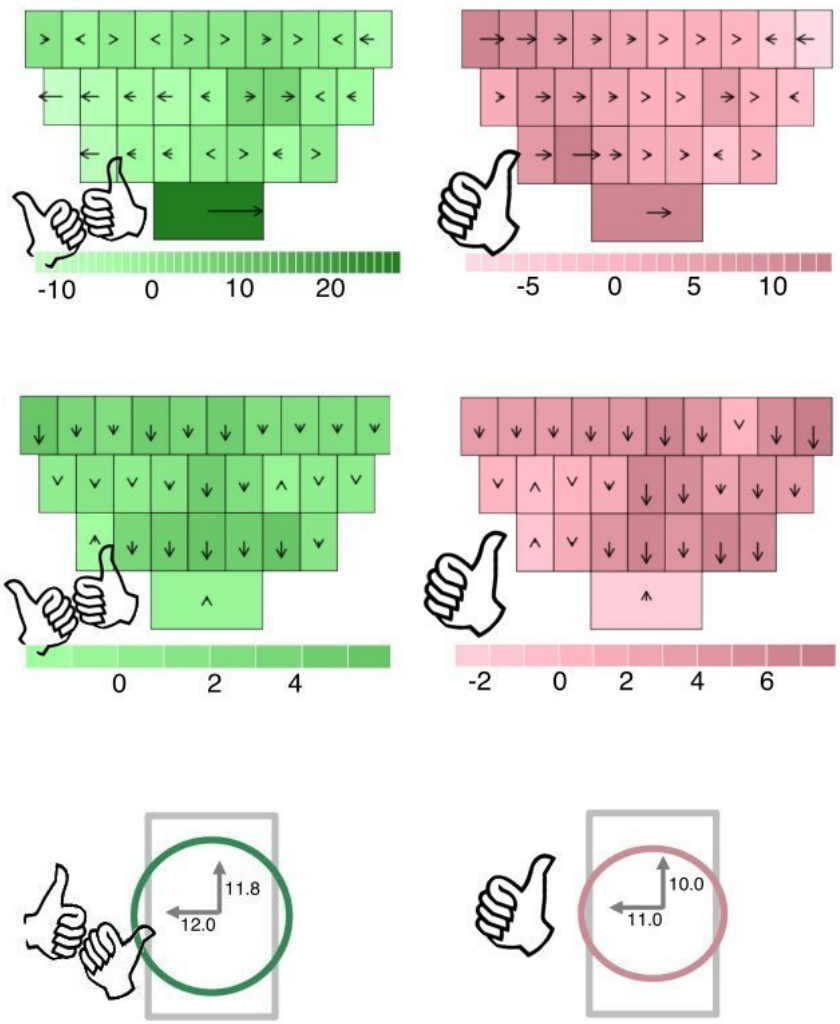


Figure 4.4: Azenkot and Zhai [1] results on inspecting touch behavior for (left) both hands typing, (right) right thumb typing, showing horizontal and vertical skews and the average standard deviation

and on the contrary when two keystrokes were at the opposite side of the key we would have measured high variances. This unpredictable behavior would have risked to compromise our keyboard, because keys showing low variance values would have been almost impossible to select.

Running a local optimization over these parameters was not feasible (each simulation over the full dataset required about ten minutes on our machine), but empirically testing of some different values confirmed our hypothesis, and eventually showed that there was a trade-off on the value to assing to ν_0, with higher values favoring the normal session, thanks to its higher regularity, against sessions with less accuracy.

Finally, the two parameter were set to

$$\kappa_0 = 2$$
$$\nu_0 = 500$$

and the variances on Figure 4.4 were normalized to

$$\mathbf{\Psi}_{both\_t} = (\nu_0 - 2 - 1) \begin{bmatrix} 12.0 & 0.0 \\ 0.0 & 11.8 \end{bmatrix}$$
$$\mathbf{\Psi}_{one\_t} = (\nu_0 - 2 - 1) \begin{bmatrix} 11.0 & 0.0 \\ 0.0 & 10.0 \end{bmatrix}$$

We performed a K-fold cross-validation [32] with $K = 9$, for each fold selecting 8 sentences from the section for the training, and testing over the remaining; we chose this deterministic folding in order not to break the integrity of the typed sentences, as it would have become useful when testing the language model. We repeat this procedure in pseudo-code, for clarity:

```
for s in sessions:
  for u in users:
    for k in (1,9):
      model[s][u][k] = learn(train_sample[s][u],
          prior[u])
```

For each fold, we performed the Bayesian learning described in the previous section separately for each session and user, and we stored personalized parameters. Then, we used the obtained distributions to simulate the performance of our model over the test data; we performed this step simulating also how each model learned in a certain situation performed in a different situation.

	Test sample			
Model	n	h	w	g
n	**2.23**	1.32	1.36	1.04
h	1.42	**1.79**	1.41	1.15
w	1.39	1.49	**1.70**	1.08
g	1.14	1.48	1.23	**2.22**

Table 4.5: Performance improvement, w.r.t. the classical keyboard, of each model when simulated with different test samples

```
for s1 in sessions:
  for s2 in sessions:
    for u in users:
      for k in range(9):
        performance[u][k] = simulate(model[s1][u],
            test_sample[s2][u][k])
      res[s1][s2] = average(performance)
```

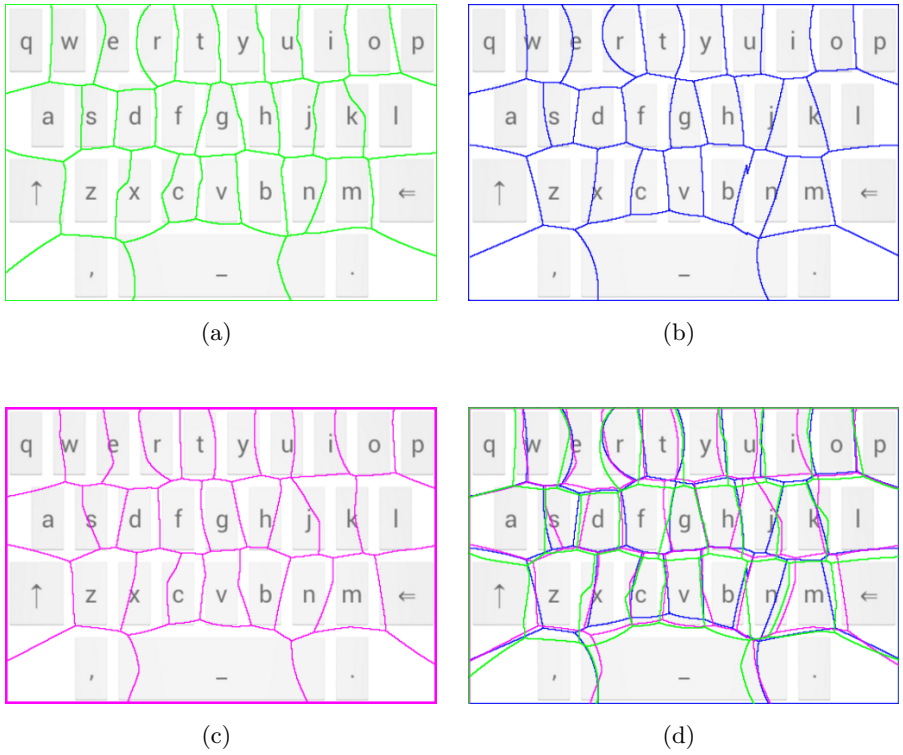
Results of this simulation are showed in table 4.5; they exhibit the expected trend: the highest values of each column and row were on the diagonal, thus proving that there is an improvement in considering the current situation with respect to using the same model, e.g., the one in the first row, to classify keystrokes in every situation.

Figure 4.5 shows how the resulting distributions determined the shape of the keyboard for one subject, by selecting for each pixel the most likely key, that is the one with highest probability of being the intended key. Figure 4.5(d) overlaps the three images to highlight their difference.

4.5 Putting it all together

In the previous section, we showed how we performed the learning of the shape of the keyboard. After this process, we had a model of the keyboard of each participant over the sessions; we also concluded that the final model should have kept in count the current situation and adapt according to it. We needed a technique that made possible to put together these pieces of information in a single model.

Having chosen a probabilistic approach, we could exploit the properties of a probabilistic framework, that permit to easily calculate values of, e.g., joint or posterior probabilities using proved axioms of probability theory. Probabilistic reasoning is a large set of techniques in an artificial intelligence



(a) (b) (c) (d)

Figure 4.5: Shape of keyboard for subject 6 while (a) normal, (b) walking, (c) glare on screen, (d) overlapped

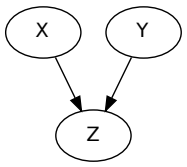


Figure 4.6: X and Y are conditionally independent given Z

for inference when the data is uncertain.

We opted for modeling the various components of our application with a Bayesian network, this permitted to use all the information we had without having to convert it, in order to have a smooth effect on the output.

4.5.1 Bayesian networks

Bayesian network [13] is a probabilistic graphical model that represents a set of random variables and their conditional dependencies via a directed acyclic graph[3], where nodes represent random variables and edges represent conditional dependencies between them.

Conditionally independence is the property of two variables X and Y to be independent given some evidence on a third variable Z; in other words, given knowledge that Z occurs, knowledge of whether X occurs provides no information on the likelihood of Y occurring, and knowledge of whether Y occurs provides no information on the likelihood of X occurring

$$P(X \cap Y \mid Z) = P(X \mid Z)P(Y \mid Z)\,.$$

In a Bayesian network, two variables that are not directly connected are conditionally independent given knowledge on all the variables on the common paths that start from them . Eventually, if there does not exist such a path, they are unconditionally independent.

In case all variables are discrete, each node is associated to a conditional probability table that defines the probability distribution over the values that it can assume given all the possible combination of values of its parents. As one can note, the values of nodes other than the parents does not directly influence the variable represented by the node, but they still might do it indirectly if there is a path that connects them to one of its parents.

[3]a directed acyclic graph is a directed graph in which there is no way to start at some vertex v and follow a sequence of edges that eventually loops back to v again

Extending this framework to handle continuous variable is obtained by adding two features: if the variable represented by the node is continuous, then the probability distribution is described by a density function, and if the parent is continuous it must be defined a generic function from the domain of the parent to the parameters of the distribution of the child.

Bayesian networks encode a joint probability distribution over all the variables in the model, so it is always possible to rewrite them as a joint probability table. The improvement they introduce is in the fact that they permit to simplify the description of this joint distribution by considering only the parameters that are evaluated to be relevant for the problem, and disregarding unnoticeable or negligible effects of some variables on others. The reduction in parameters ensures both a easier development of the model, and more lightweight implementations.

Thanks to this property, it is possible to retrieve the joint probability of any set of variables, given knowledge on any (disjoint) set of other variables.

4.5.2 AdKey underlying model

First step of the process of defining a Bayesian network was to determine which were the variables that were playing a role in determining the user typing pattern. From previous analysis, emerged that the situation had to be kept in count, so we modeled each situation as an independent variable, that altogether defined the abstract concept of *context*. The values these variables could assume were reals comprised between zero and one, representing how much that situation was influencing the context.

Once these values were known, and this could be accomplished exploiting the device sensors and the speed model outlined in Section 4.3.1, they would define how the user would type. We modeled the variable *context* as a set of four weights, one for each distribution that we had learned (refer to Section 4.3.1), and then defined a function that reflected the interaction between the three observed situations.

Since the normal situation contrasted with the other three, or in other words when there was a situational impairment the situation was not *normal* anymore, its correspondent weight was determined as one minus the value of hurry situation minus the maximum value between walking and hurry; then all values were normalized to sum up to one.

Clearly, such distribution would not be useful if it did not carry with it some information, specifically, which key the user intended to type when hitting that position on the screen, so we introduced the variable *key*. This would be the variable which we would like to perform inference on, to select

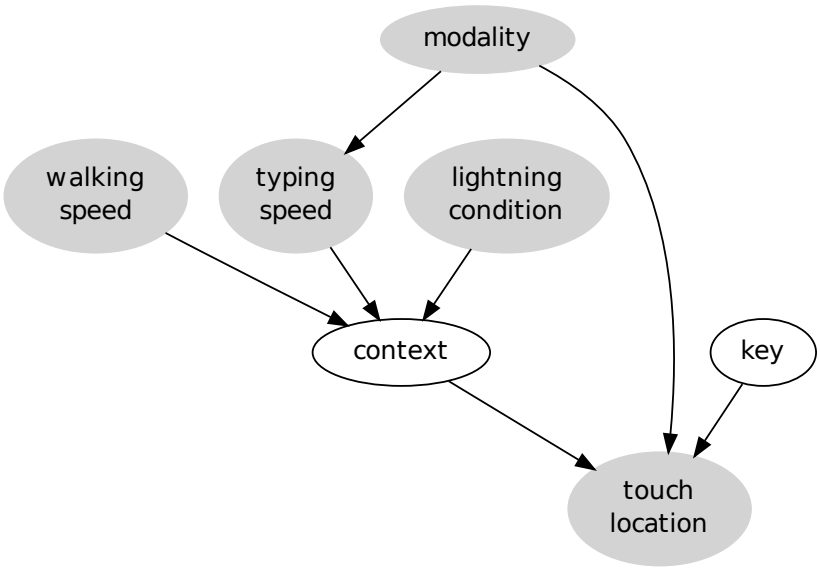


Figure 4.7: Bayesian network representation of the AdKey classifier model

the key that is most likely.

The function between *context* and *touch location* was straightforward: it was enough to multiply the learned distributions for the respective weight to obtain one mixture of Gaussians for each *key*, that is a discrete variable.

Finally, we have already showed how the *modality*, that is, how the user is holding the phone while typing, conditions both the *typing speed* (Section 4.3.1) and the *touch location* (Section 4.4.3). *Modality* is another discrete variable representing whether the subject used one or both thumbs; this value would change the parameters of the linear model to infer the relative speed of the user, and also the distribution over the screen. In our case it affected only the prior, so the arrow towards *touch location* has been put just to show this relation, implemented by an identity function; in an online model this would have actually an effect.

Figure 4.7 shows how we modeled these relations in a Bayesian network.

Our task with the network was to select the key that was most likely to be pressed given the context and the observed touch locations. In the figure the variables that are known at runtime are represented by a shadowed node.

In order to compute the value of the context we simply used the function described above, that was defined by us and represented the most likely

Figure 4.8: Shape of the keyboard for subject 10 when typing at very high speed while walking and with a glare on the screen

value the context should have assumed (another approach could have been to learn it, but we decided to put our knowledge about how the situations were influencing each other.

Finally, the key k was from the mixture of Gaussians obtained distribution as the one having the highest probability

$$k = \arg\max_{key} P(key \mid touch\_location\,, context)\,.$$

Figure 4.8 shows how the three situation models combined to form the shape depicted, in a simulation of a very unfortunate situation in which the subject had to type fast while walking and with a glare on the screen (we did not actually ask anyone to do that). The weights were respectively $0, 0.33, 0.33, 0.33$.

4.6 Extending it with the language

As we mentioned at the beginning of this chapter, probabilistic models have the nice property that can be combined together, as long as the axioms of probability theory are respected. We will here show how it is possible to include in the model proposed in the last section another one, that adds knowledge about the language.

4.6.1 Hidden Markov Models

A Hidden Markov Model (HMM) is a model for probabilistic reasoning over time. HMMs were developed as a technique in the mid '80s [50], and were subsequently showed to be a special case of Dynamic Bayesian Networks (DBN) [43]. A representation in form of DBN of a sample HMM is shown in Figure 4.9, where the state is denoted as Z_k and the corresponding evidence as X_k. We will use this notation through all this section.

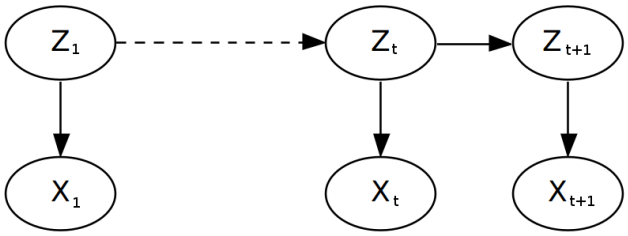


Figure 4.9: A Dynamic Bayesian Network graph of a HMM

HMMs permit us to deal with processes that vary over time, modeling the time flow as a Markov Chain, thus the evolving of the state Z of the model (that must necessarily be discrete) is regulated by the Markov Assumption, that is, the current state depend on a finite fixed number of previous states. In case of a first-order Markov Process

$$p(Z_t \mid Z_{0:t-1}) = p(Z_t \mid Z_{t-1})\,.$$

What differentiates HMMs from a Markov Decision Process is the fact that the state is not directly observable. What we observe is an evidence of it, or in other words what the agent's sensors are capable of measuring from the state.

Under the Markov Sensor Assumption, the evidence X depends only on the current state, that is

$$p(X_t \mid Z_{0:t}, X_{0:t-1}) = p(X_t \mid Z_t)\,.$$

Given a HMM, several inference tasks can be performed; the most interesting for the scope of language modeling are:

Filtering. Filtering means computing the belief state, given all evidence up to date, that, thanks to the Markov assumptions, becomes

$$p(Z_{t+1} \mid \mathbf{x}_{1:t+1}) = \alpha p(\mathbf{x}_{t+1} \mid Z_{t+1}) \sum_{\mathbf{z}_t} p(Z_{t+1} \mid \mathbf{z}_t) P(\mathbf{z}_t \mid \mathbf{x}_{1:t})$$

where the first element is specified by the sensor model, and the summation *weights* the transition model for the current belief state.

Thus, the belief state is updated at every timestep given the observation, and this becomes the *message* for the next timestep.

Prediction. This is the task of computing the posterior probability over the future states, before observing the new evidence, it turns out to be a very similar computation, without the contribute from the evidence

$$p(Z_{t+k+1} \mid \mathbf{x}_{1:t}) = \sum_{\mathbf{z}_{t+k}} p(Z_{t+k+1} \mid \mathbf{z}_{t+k}) P(\mathbf{z}_{t+k} \mid \mathbf{x}_{1:t}) \,.$$

Smoothing. This is somehow a derivation of the filtering task, that computes the posterior distribution over a past state k for $0 \leq k < t$ given all the evidence up to the present t. Smoothing provides better estimation of the belief state, because it incorporates more evidence.

This probability can be decomposed in the two terms

$$p(Z_k \mid \mathbf{x}_{1:t}) = \alpha p(Z_{1:k} \mid \mathbf{x}_{1:k}) p(\mathbf{x}_{k+1:t} \mid Z_k) \,.$$

The first term has the same value of filtering up to the state k, while it turns out that the second term can be computed in a similar way as a recursive message, but starting from t.

$$p(\mathbf{x}_{k+1:t} \mid Z_k) = \sum_{\mathbf{z}_{k+1}} P(\mathbf{x}_{k+1} \mid \mathbf{z}_{k+1}) P(\mathbf{x}_{k+2:t} \mid \mathbf{z}_{k+1}) p(\mathbf{z}_{k+1} \mid Z_k)$$

where the first term is retrieved from the sensor model and the third from the transition model, while the second is the *recursive call* from the previous step. The first step, that corresponds to the last observation available, is

$$p(\mathbf{x}_{t+1:t} \mid Z_t) = p(\ \mid Z_t) = \mathbf{1}$$

because $\mathbf{x}_{t+1:t}$ is an empty sequence.

Most likely sequence. In this case, given a sequence of observations, we want to find the corresponding most likely sequence of states that generated those observations, that is $\arg\max_{\mathbf{z}_{1:t}} P(\mathbf{z}_{1:t} \mid \mathbf{x}_{1:t})$.

This task is solved by the Viterbi algorithm [45], and is very useful in many applications, in particular speech recognition where the aim is to find the most likely sequence of words, given a series of sounds.

The algorithm achieves the result by at each step (for each observation) computing the probability for each state of generating that message and then selecting the most likely to compute next joint probability

$$\begin{aligned} &\max_{z_1 \dots z_t} p(\mathbf{z}_1, \dots \mathbf{z}_t, Z_{t+1} \mid \mathbf{x}_{1:t+1}) \\ &\quad = \alpha p(\mathbf{x}_{t+1} \mid Z_{t+1}) \max_{x_t} \left(p(Z_{t+1} \mid \mathbf{z}_t) \max_{\mathbf{z}_1 \dots \mathbf{z}_{t-1}} P(\mathbf{z}_1, \dots \mathbf{z}_t \mid \mathbf{x}_{1:t}) \right) . \end{aligned}$$

Since at every step is stored the probability of the most likely sequence until that state, once the most likely final state is known, we can reconstruct recursively backward which was the sequence of states that brought to that state.

4.6.2 A simple language model

In our model, the variable on which we were performing inference on was the *key*, which is defined a discrete variable, so we could assume it to be the hidden state of a HMM. This implied that the classification of the key would become a process that involved both the context and the time flow, sampled at each keystroke.

Many language models have as underlying description a HMM, because it simple to estimate the transition probability of a word being after another by observing the frequency with which this sequence happen on a large text corpus, e.g., the Google n-gram database [41] with respect to the total count of the first word in the sequence

$$P(w_{k+1} \mid w_k) = \frac{P(w_k.w_{k+1})}{P(wk)}\,.$$

Since this process will assign probability zero to any sequence that has not been observed in the corpus, this estimation is usually adjusted by a smoothing procedure. There are many available, such as Katz [31] and Kneser-Ney [46], some of them are more accurate than others but what they basically all do is to take away some probability from the observed sequences, and redistribute it to the unobserved.

The same process can be performed with frequencies of characters within words, so we proceeded by taking the estimated bigram frequencies over a large English text corpora [30] and computing with the formula above the conditional probabilities of any character given observation of the previous character. This probability table was equivalent to a transition matrix between the states of the HMM, thus specifying an initial distribution, that we assumed equally distributed over the keys, we had a completely specified HMM.

We opted for the filtering algorithm, that gave us a real-time estimation of which was the probability of each key. The probability of each key being pressed became

$$k = \arg\max_{k_t} P(k_t|touch\_location, context) \sum_{keys} P(k_t \mid k_{t-1}) P(k_{t-1})$$

We are aware that this model is overly simplified, but our purpose was just to show that our model could be extended with other knowledge, in our example about the language. Our main goal was still to prove that there exists a correlation between the typing patterns and the situation in which they are performed.

It is easy to extend this model with a more predictive one, for example just by considering strings of length greater than one, or, as more advanced statistical language processors do, by using dictionaries with words tagged with the grammatical function they assume in the sentence.

Finally, it would be possible to combine it with other algorithms described in previous section, like for example Viterbi, that performing a backward analysis of the sequence likelihood is able to correct misclassified characters when new evidence appears that contradicts them.

Chapter 5

Description of the Architecture

In this chapter we describe how we implemented the modules that composed our work. We start by listing them and further go into details.

5.1 The components

In order to obtain the final product, we developed several software modules that aided as tools to, starting from the definition of the problem, reach a working prototype of a situation-aware adaptive keyboard, namely AdKey.

At certain points during the research, these modules had to communicate with each other, so we had also to design them in order they could transmit pieces of information using some sort of standard language.

We introduce here the main components of our work, that are described later in this chapter.

The data acquisition application. As mentioned in Section 4.2, we needed to collect data on people typing in different observed conditions. Many different keyboards are available for the Android platform, but we needed the feature of registering touch events with related information, in order to subsequently analyze them, so we developed our keyboard clone, working as an Android application but simulating a soft keyboard.

The database. Data was retrieved on an Android handheld device, but needed to be analyzed in a desktop environment. When considering how to

transfer the log from one to another we evaluated to store it in a platform that would ensure both security for our data and an interface to make it available to many analysis tools; we opted for a relational database management system.

The learning process. Once it was clear that a specific model needed to be developed for our problem, because standard machine learning tools were not providing expected results, we needed to develop an application specific learning tools, which included also methods for testing the obtained models. We developed this as a series of Python modules, because of its versatility that permitted to test various models writing short and well-readable code. All the graphs in this essay are also printed in Python.

AdKey, the final prototype. Again, the final application needed to run on Android, to test its performances on actual people typing, while exploiting the device's sensors to adjust the shape of the keyboard. We call this a prototype because, despite being a completely working keyboard, it does not let input type in other applications, such as e-mails, but only in the application itself for evaluation purposes.

5.2 Keyboard for data acquisition

This was an Android application which rendered a soft keyboard on the screen and proposes some phrases to be inputted by the subject participating in the experiment. It registered information about the subject interaction with the touch screen and it saved it persistently on the device external storage, in order to be later accessed.

The layout was specified, following Android instructions, in an XML file that the Android SDK automatically parsed to generate the effective UI components. The file `/res/layout/activity_main.xml` contained the declaration of the graphical elements, that were essentially a `TextView` to show the phrase to be inserted, a `Button` to flow between sentences, an `EditText` to show the user the text he typed, and the keyboard.

The keyboard was described by a `RelativeLayout` that contained the information on where to render each key, which itself was a `Button`. The trick we set up for detection the touch locations was to pose a transparent `TextView` over the rendered keys, that would intercept user touch. The logic would then manage to inform the correspondent underlying key that it had been pressed.

The `MainActivity` activity was the class that was called when the application was started. The `OnCreate` method contained the procedure to initialize the graphical interface, including all the event listener associated to them. The core of the application was the event listener associated with the `TextView` overlying the keyboard, `onTouch` that was activated whenever the user touched the screen in the keyboard area. This method was responsible of discriminating what was the event that generated the call. It could be `ACTION_DOWN`, meaning the user has posed his finger on the screen, `ACTION_POINTER_2_DOWN` meaning the user has posed a second finger on the screen, `ACTION_MOVE` that indicated a sliding of the posed finger, `ACTION_POINTER_1_UP` meaning that the user released his first finger but there was still one finger touching the screen, `ACTION_UP` to indicate that all finger had been released from the screen.

All this events were translated into a portion of XML containing the related information (x, y, timestamp, size, pointer id, key selected) that was concatenated to the `log String`, which was flushed to file with a `BufferedWriter` when the subject had finished typing the sentence and hits the `next Button`.

At each event, the `getButtonAt` method was also called, that received in input the coordinates and selected the key correspondent to that coordinates; if the event was of category down or move, the returned `Button` was selected, a `label TextView` visualized over it and a 30 milliseconds vibration performed, to guarantee visual and haptic feedback; when it was instead of category up the correspondent character was added to the `form EditText`, except for the *backspace* key, that eliminated the last inserted character, and the *shift* key, that switched all the keys to capital letters.

The `PhraseGenerator` class encoded all the sentences that the subjects were asked to type, retrieved through the `getPhrase` method when the `next Button` is pressed.

5.3 How we stored the data

Data retrieved from the experiment was stored in an XML file on the device external memory (micro SD) with the following DTD structure, where basically `event` is just a container for all the events related to a given touch, from when the first finger touched the screen to when the last released it, stored as single `TouchEvent`s.

```
<!ELEMENT root (phrase)+>
<!ELEMENT phrase (event)+>
```

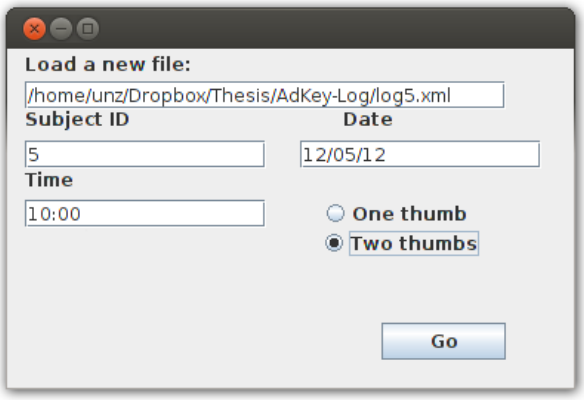


Figure 5.1: The application that parsed the XML retrieved during the experiment and loaded it into the database

```
<!ELEMENT event (touchEvent)+>
<!ELEMENT touchEvent EMPTY>

<!ATTLIST touchEvent
    x       CDATA   #REQUIRED
    y       CDATA   #REQUIRED
    type    CDATA   #REQUIRED
    time    CDATA   #REQUIRED
    size    CDATA   #REQUIRED
    pointer CDATA   #REQUIRED>
```

We set up a MySQL Server version 5.5.29, with graphical interface provided by phpMyAdmin 3.4.10.1 running on Apache 2.2.22 web server, that lets you visualize and manipulate the database content in a web-browser environment.

We built a Java application for loading these files into the database, that had a basic graphic interface showed in Figure 5.1 to permit the insertion of additional information about the experiment; the subject ID, in order to subsequently identify it for the evaluation session (no personal information was stored on the database), date and time and how the experiment had been performed.

This application was composed of a `XMLParser` that exploiting the Java `DocumentBuilder` returned an `ArrayList` of `Event` objects, each of them containing another `ArrayList` of associated `TouchEvent` objects.

This data structure was then iterated by the `DBLoader` and inserted in the database with standard `INSERT` commands using the Java database con-

troller `MySQL-connector` version 5.1.22. Care was taken in order to maintain a reference between the `Event` and `TouchEvent` through its field `EventKey`, in order to be able to later compute relevant information on `Event`.

The schema of the database is showed on Figure A.1.

5.4 Model learning and testing

This was the critical part of the process, where data was analyzed, used to train and test the final model described in Chapter 4 and finally loaded on the device to be used for the final prototype. All of these steps were performed in Python, a general-purpose, high-level programming language.

The main drawback of this language is its feature of being interpreted, rather than compiled and executed, but this was overcome by the great extensibility given by its modularity and the ease of coding with a high-level interface. Furthermore, execution times were not a priority, given that we were simulating.

We can split this part of the process in various steps.

Classify the data. Once data was loaded onto the database it still did not included the most important piece of information we needed: the *intended* key. This is not in fact something a system could know by itself, but we had to insert this knowledge in order to proceed. This is something that always happens in supervised learning, as it was ours where the algorithms need a pre-classified set of data from which to infer the generalizing model.

We performed this process by hand, scrolling through the database spotting the errors and correcting them. As previously mentioned, we opted for a manual approach because we noted that the situational impairments we were experimenting introduced a big variety of errors; not only users were pressing keys adjacent to the intended one, but were also performing errors such as missed or reversed keystrokes, that would have been difficult to spot for an automated technique.

We kept the information on the actual key pressed in the field `PressB`, and the corrected one in `IntentB`, in order to subsequently analyze difference in performance.

Compute additional information. Raw data at this point included only basic information, such as the touch coordinates and the timestamps (in milliseconds) in which they happened, other than experiment related data such as the subject who generated them `Subject` and the relative `Session`.

Subject	Session	FirstX	FirstY	LastX	LastY	Dist
1	1	153	148	153	148	0
1	1	246	38	242	37	4.12
1	1	110	86	98	81	13
1	1	80	38	78	28	10.19
1	1	269	36	262	36	7

PDist	Time	Ptime	AccXInt	AccYInt	PressB	IntentB
-1	172	-1	-7	11	v	v
144.04	206	441	-15	10	o	i
140.80	215	434	2	-2	d	d
46.61	235	230	-2	1	e	e
191.16	225	261	10	-10	o	o

Table 5.1: Some sample data from `Event` table

In order to have analyzable data, we needed it to be at a higher descriptive level.

For each keystroke `Event` we added the sequent fields and computed their relative values: the location of laying, `FirstX` and `FirstY` and release `LastX` and `LastY`, including the eventual sliding distance `Dist` (simple euclidean distance between the previous two values), the duration of the pressure `Time` and its time distance from the previous stroke `PTime` whether they were coinciding and in negative case.

For shaping the keyboard, we opted to give the learner algorithm a relative coordinate systems such as the center of the intended key, so that he had not to distinguish between keys and at the end we could simply sum up the offset given by the key center position to reobtain the absolute coordinates. The distance from the center of the key, that can be interpreted as the accuracy in hitting that key, was stored in `AccXInt` and `AccYInt`.

Create the folds. Now the data was almost ready to be parsed by the learning algorithm. We just needed to split it in order to have a training and a test set of samples. So we replicated the data over nine different tables `EventTest-`i each containing samples retrieved from typing one of the sentences in each session. The remaining sentences used to train the i-th fold were replicated in `EventFold-`i.

Choice of nine over the classical k-folding value of ten was due to the fact that we had exactly nine sentences for each session, and in this way we avoided splitting them between the sets.

Learn the user distribution. We implemented a Python object `MultiNormalBayesLearner` that retained information on the prior distribution `mean_prior`, `cov_prior`, `nu_0` and `kappa_0`. The `train` received in input a list `train_sample` of all the keystokes which the intended each user in each session.

Thanks to `numpy`, a library for mathematical calculus in Python, all the computation described in formulas in Section 4.4.2 was expressible in a few and intuitive lines of code.

All the estimation performed were stored and subsequently averaged to find what were the parameters of our *personalized situation-aware* model. Each user had thus $n_{keys} \times n_{params} \times n_{sessions} = 32 \times 5 \times 4 = 640$ parameters.

Simulate the obtained model. With the retrieved parameters, we ran a simulation of how a keyboard implementing our probabilistic model with these parametric quantities would have performed. The simulation involved loading all the keystrokes in the testing folds and classifying them using the models we learned at previous step.

Each keystroke was classified with a model of the user who generated it, but we iterated over all the models that we learned for that user, in order to compare them and select the one that was performing better. As showed in Table 4.5, these were the model corresponding to the session in which the keystrokes had been collected.

The classifier itself was straightforward: it retrieved the keystroke release location, `LastX` and `LastY` and computed the probability of that keystroke being under the probability distribution of each key. The one with highest probability was selected and compared to the intended one; if they were coincident, the classification was accepted, otherwise an error counter was increased. The final value of the error counter was compared with the previous error occurrence, that is the number of keystrokes in which `PressB != IntentB`.

5.5 The final adaptive soft keyboard

Simulation results proved that a personalized situation-aware keyboard could increase typing accuracy by a factor of roughly two, which means that the number of errors performed would be halved. In order to test this on a real environment, and to produce a tangible realization of what observed, we proceeded adapting the keyboard that we had built for the data retrieval session enhancing it with the adaptive-model.

We kept the graphical interface we had developed but we extended it with the logic derived from our research, so that the classification of the keystrokes would happen as a function also of the user and the external situation, rather than only on the touch position.

While for the learning process we used a description that was mainly sequential, we will here describe our application in a more structural way, listing the main components that together worked to classify the user keystrokes online, i.e., while the user was actually typing. Please refer also to Figure A.2 on page 84 for the class diagram.

Graphical elements. As we said, we maintained the same graphical structure of the data retrieval application; the reader can refer to Section 5.2 for the details.

Sensor models. They were three modules, namely `WalkingQuantifier`, `SpeedQuantifier` and `LightQuantifier`, that were responsible of monitoring the context in which the user was typing, either by receiving information from sensors or inferring it from user typing dynamics. The state of the situation was then read by the stroke classifying process to parametrize the distribution over the keys.

Classifier. The classification was performed in the class `Model`, which was the core of the application; when a stroke was performed, it read the values from the sensor models and computed the relative weights to be associated to the distributions stored in `UserModel`, eventually coupled with the `LanguageModel` ones.

We will now analyze in detail how these components are defined and how they interact with each other.

5.5.1 Graphic interface and sensors management

The Android system defines the applications as composed by `Activity` classes, that are responsible of communicating with the system to acquire or handle its resources. Our application had two activities, `UserSelect` and `Keyboard`; the former was a very simple window permitting to insert the subject ID, needed to load its personalized distribution, users were informed on which was their subject ID so that they could personally type it.

Then, the `Keyboard` was started, rendering on screen the same layout described in Section 5.2. This was the main activity, responsible of instanti-

ating all the objects needed by the application. Furthermore, since only the activities are designed to communicate with the system, and in particular with the touch screen and the accelerometer, it was responsible with opportune event listener of intercepting hardware interrupts and forwarding them to the indicated components.

The number of keys was defined as a static property of this class `nKeys`, but accessible within all the project. All the reference to the keys were stored in an array, that for our scope worked as a map between integers and real keys; in this way we could abstract the classifier from knowing on which key it was working, and referring to it as an index. When the correct index was selected, it was possible to perform the correspondent action on the key in constant time.

The activity needed to implement the `OnTouchListener` method `onTouch` to retrieve the events related to the touches on the screen. Switching over the type of event was performed, so that when the screen was pressed the classified key was selected, a magnified version of it was displayed over it to provide visual feedback and a small vibration of thirty milliseconds was performed. When the touch was released instead, we retrieved the character represented by the key and we appended it to the `EditText` form designed to contain the inserted text.

Another implemented class was `SensorEventListener`, needed to retrieve values measured by the accelerometer through the overridden method `onSensorChanged` and sending them to the walking speed model.

Finally, the label showed when over the key when touch was performed was a simple `TextView` that was set to visible only when a finger was touching the screen, showing the same text that was present on the underlying key and shifted in order to appear over it.

5.5.2 Situational quantifiers

These modules were responsible of keeping track of the situation in which typing was performed, in order to let it accessible to the classifier when needed. We had three of these quantifier, one for each situation we were considering.

An important design choice is that we instantiated them in the `Keyboard` activity and then we passed the references to the classifier, so that they could be updated from the activity, responsible of reading the values from the hardware sensors, and accessed directly within the `Model`. This property is showed in the class diagram on page 84 by the common arrows between the two, we are here specifying that those arrows represent shared objects.

`WalkingQuantifier` was responsible of estimating the speed at which the user was walking. This was achieved by measuring the standard deviation of the measured acceleration over the y-axis of the device, over the last 50 observations (at a sampling frequency of 14 Hertz, this meant approximately over the last 4 seconds). This approach had already been used by [3], which showed how this value estimates the walking speed accurately enough. Observations were stored in a `Queue` (FIFO servicing), of fixed length, and the value `sum` was updated at each step in order to have the mean immediately available; when requested, the standard deviation was computed as average quadratic distance from that value.

The value was limited to 1.6, which appeared to be the value of a user walking at normal speed, and then normalized to 1.

`SpeedQuantifier` worked in a similar manner, but in order to estimate the typing speed used the model that we defined in Section 4.3.1. In this case the values of inter and intra stroke time were smoothed over the last ten observations, stored also in a `Queue`; to retrieve the effective times the object took as input the timestamp in which the stroke was generated and subtracted it to the previous observed one, stored respectively in `LastPress` and `LastRelease`.

Then it computed the distance of the newly observed keystroke times from the linear model developed in Section 4.3.1, which parameters were depending on the way the user was holding the device. If the inter measured inter stroke time was higher or equal to the one predicted by the model, then a value of zero was returned, meaning the user was typing slowly; the more this time was lower than the expected, the more the inferred speed was inferred to be higher, till the value of one third was reached.

Finally, `LiqhtQantifier` did not receive any input from the sensor, because we realized that the device, despite being equipped with a light sensor, this was not available at API level, or the system was not able to read its value. We solved this issue by manually setting the value to one when we knew that there was a light pointed to the screen, that is, in the last session of the experiment.

5.5.3 Underlying models

There were two classes `UserModel` and `LangaugeModel` responsible respectively of handling the keyboard model, personalized on the user that was currently typing, and the language model. The former stored the four distributions that we had learned in the previous section, accessible through the method `getProbabilities` that returned the probability distribution over

the keys given a certain position on the screen, while the latter behaved as an HMM, storing the distribution computed after the last keystroke and returning the predictive probability distribution for the next stroke, retrieved using the internal representation of the language `transitionMatrix`.

The `Model` class was responsible of handling all the logic of the classification process. As we said, it had access to the quantifiers instantiated by the `Keyboard` by sharing their references; furthermore, he had control over the two specific models `UserModel` and `LanguageModel`.

When a touch event was detected, the method `classifyThis` received in input its coordinates and type of event, then interrogated the situation quantifiers about their status, which was in all the three cases a value comprised between zero and one. The function `computeWeights` was responsible of adapting these three values to the four weights correspondent to the distribution.

The reason why there were three situations and four models is to attribute to the fact that the normal situation is complementary to all the others: when the user is walking, he is not in a normal situation anymore, and so it is when writing at high speed. In the opposite, the other three situations can manifest together, so the weights between them should be balanced in that case. This was achieved by removing a certain amount to the normal situation weight, and redistributing it proportionally to the other weights.

Once the weights were note, it was straightforward to calculate the probabilities, by weighting each distribution given by the different situations. One approximation that we made was of computing the probability from the density function; this was necessary because the multivariate normal distribution has no closed form of the cumulative function, so the other approaches would have been random sampling over the distribution, or numerical integration, but we considered them too much computationally expensive.

Furthermore, we believed the density function to be a good approximation, since we needed to integrate over an area of one, and with that we were basically assuming that the mean over that area coincided with the value of the function, that is the median.

Chapter 6

Results and Conclusions

In this chapter we show the performance of our final prototype on the field, and compare it to similar works. Next, we draw the conclusions of our work and list possible future scenarios.

6.1 Evaluation on field

We performed a second experimental session in which we recontacted a subset of subjects who had participated to the first session and gave them the opportunity to test AdKey with personalized model.

We had to contact specifically subjects that participated to the first session because our model was personalized on their behavior during the first; six users agreed to participate to this second session.

We reduced the amount of typing required by them, primarily because we did not need the data retrieved to perform other testings, but just to verify what our simulation showed, and also because we received complaints about the length of the test during first experiment.

Subjects were asked to type four sentences for each of the same situations we had retrieved data, but this time the keyboard was not preprogrammed to know the situation, but was instead *sensing* it thanks to the models we developed. The first two sentences of each session used just the user models, while the other two exploited also the language model.

Table 6.1 shows the improvements obtained by the users over the sections, compared as usual to the non adaptive keyboard, but this time compared with the same sentence typed at distance of some day. Reported values have no statistical meaning, because they were acquired on a very small sample of six users typing twenty sentences; nevertheless, we believe

	Language	
Session	Without	With
n	2.53	1.29
h	2.11	1.81
w	2.50	2.79
g	1.95	1.56

Table 6.1: Observed improvement w.r.t same sentence typed on the classical keyboard with and without the language model

they confirm the trend showed in the simulation, with values that are similar to the one in Table 4.5.

We compared also the values of the speed, that showed an average increase of 20% in both the first two sessions, specifically

1.21 1.22

A separate discussion should be made for the values observed with the language model active. While they are still greater than one, thus showing better performance over the classic keyboard, the language model seemed to have less predictive power than the one based only on the situation.

We assume that this can be attributed to the fact that a first order Markovian process is not accurate enough to model the language complexity.

Our work was not the first that shaped a soft keyboard for smartphone with a probabilistic model. An approach similar to ours is from *Goodman et al.* [19], who adopted the same multivariate Gaussian approach, but putting more emphasis on the language model, that was based on a seven order Markov process. Their results were very similar to ours, showing a 1.87 improvement with respect to the standard keyboard.

We hypothesize that combining our keyboard model to their language model, which appears to more accurately reflect the complexity of the language, could lead to higher improvements.

6.2 Contributions to the state of the art

First of all, we believe that our work confirmed our hypothesis that typing dynamics are influenced by different variables. Some of them had already been explored, and we confirmed them by adding new experimental evidence, and other were completely unexplored and we set up for them some hypothetical models, that will eventually need to be confirmed by other observations.

In particular, regarding the first category of already explored facts, we think that we confirmed with new evidence how important is, when designing mobile interfaces, to keep in count how the user personally perceives the interaction; this applies not only to soft keyboard but to any mobile interface in general. Nevertheless keyboards, as interfaces for an important task such as text input, need particular care.

As mentioned in Section 2.2, for years inaccuracy observed in touch screen interaction has been attributed to the so said *fat finger* problem, that is the inability to point accurately a target showed on the screen because of the softness of the skin and the occlusion of what is being touched. Only recent works suggested that the accuracy can be improved if the perceived input model of the user is known; we took this fact as an hint, and in fact we proved that having a model that is personalized on what observed by a single user increases accuracy up to a factor of two (refer to Table 4.4 on page 38).

Our hypothesis was that the patterns exhibited while interacting are a variable not that only depends on the user himself, but also the external environment takes part into it. Our work proved this to be true, showing that a model that does not keep in count the situation performs worse when there is a shift in the situation in which the user is interacting (refer to Table 4.5 on page 45).

Even if it takes a secondary part in our research, the model we developed for inferring the user typing speed analyzes a new aspect of typing that as far as we know had never been explored. The typing speed is clearly a variable that depends on the user, its level of experience and also his hurry to perform the task. Being able to infer whether a user is typing at his average speed, rather than faster, permits to adjust the model parameters in different way: in our work we focused in adjusting the shape of the keyboard to the one we observed when typing faster, but this information could be exploited even at higher levels, as we point out in Section 6.3.

Lightning condition has been supposed to be a factor of influence in mobile typing, but has not been inspected deeply. In our work, we showed how lighting conditions, and in particular a glare showing off on the screen, particularly influence the interaction dynamic, being the context that performed worse when simulated with model learned in different situations (refer to table 4.5 on page 45). We suggest that this has to do with the fact that the inability to look at the screen makes the user type following even more his own patterns, because he can not rely anymore on a visual feedback.

As one can see from the picture in Figure 6.1, glare was one of the toughest situational impairments we considered. The number of errors was still

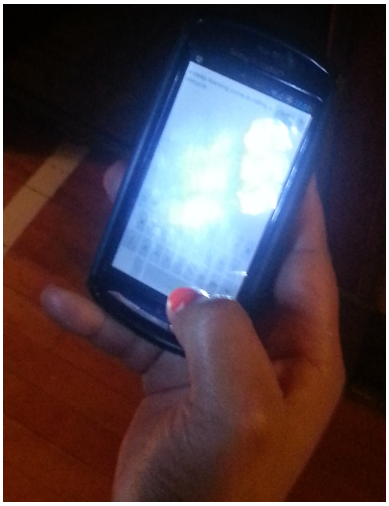


Figure 6.1: Picture showing the glare on the screen

high even with the adaptive keyboard, but the performance improvement maintained on line with the other sessions.

From a higher perspective, we think we contributed to diminish the level of effort users need to put when interacting with devices; we firmly believe that the more a system knows or is able to infer about the user, the easier will be for him to accomplish his tasks with the machine.

6.3 Possible follow-ups

We conducted an exploratory work, mainly focused on how induced situational impairments can be modeled to produce a keyboard that lets the user perform less errors. The keyboard that we developed is far from being usable as a real product, because of many simplification we introduced in our work.

First of all, the lack of an on-line learning process, with the need of external supervision to classify the intended keystrokes, makes this keyboard not designed for the final user, who wants a product working out of the box.

Secondly, we introduced a very basic language model, based only on the previous keystroke, thus modeling the language as a first order Markovian process. This can be extended to longer sequences in order to provide better predictions that are actually based on a dictionary, and can go even further, by modeling probabilities of words within phrases; most advanced techniques might classify label words by their grammatical function within the sentence, in order to predict words even more accurately by discarding part of speech

that are unlikely, e.g., in common English, a noun is very rarely followed by an adjective.

The two aspects we mentioned go along well, in fact having a good language model is the first step to implement an automated technique capable of learning the distribution over the keys. Imagine a user typing "vodeo", if the language model corrects it to "video" and the user leaves it unchanged, then the system knows that the "o" has to be attributed to a wrong keypress, and if this happens to be frequent, it can adjust its distribution over the correspondent key, so that next out of that key could be interpreted correctly, or at least given an higher probability of the intended key being the adjacent one.

On the other hand, further analysis is necessary to better inspect the typing speed model we introduced in Section 4.3.1. While our model appear to fit good our data, with a correlation coefficient $\rho = 0.80$, we really had too few observations to produce reproducible knowledge. It might also be the case that the real underlying relation is not linear, but has some quasi-linear shape, e.g., a slow exponential, but as we said also in the definition of the model, we did not consider opportune to try better fitting models with so little data.

Nevertheless we believe that knowing the user relative had an important part in our work, and could be exploited at higher levels, for example to infer even more about the context. In a hypothetical application, when the user appears to be in a hurry the system could automatically propose him predefined messages such as "I'm going to work" or "Talk to you later".

It would also be interesting to further investigate how the users adapt to interfaces that let them type with less accuracy, because the system automatically corrects them. In our work, we implemented an adaptive model that adjusts its parameters to the user, but there was no observation on how the user would react to this adaptation, by counter-adapting himself.

We would like to close this essay with an open question: will the predictive capabilities of the machines become so powerful that there will be no need for the user to interact with them?

Bibliography

[1] Shiri Azenkot and Shumin Zhai. Touch behavior with different postures on soft smartphone keyboards. In *14th international conference on Human-computer interaction with mobile devices and services*, page 251, New York, New York, USA, 2012. ACM Press.

[2] Leon Barnard, Ji Soo Yi, Julie A. Jacko, and Andrew Sears. Capturing the effects of context on human performance in mobile computing systems. *Personal and Ubiquitous Computing*, 11(2):81–96, 2006.

[3] Joanna Bergstrom-Lehtovirta, Antti Oulasvirta, and Stephen Brewster. The effects of walking speed on target acquisition on a touchscreen interface. In *13th International Conference on Human Computer Interaction with Mobile Devices and Services*, pages 143–146, 2011.

[4] Stephen Brewster, Faraz Chohan, and Lorna Brown. Tactile feedback for mobile interactions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 159–162, 2007.

[5] Lee Butts and Andy Cockburn. An Evaluation of Mobile Phone Text Input Methods. In *Third Australasian conference on User interfaces*, volume 24, pages 55–59, 2002.

[6] Tony Casson and Patrick S. Ryan. Open standards, open source adoption in the public sector, and their relationship to microsoft's market dominance. *Standards Edge:Unifier or Divider?*, 2010.

[7] Jörg Edelmann, Philipp Mock, Andreas Schilling, Peter Gerjets, and Wolfgang Rosenstiel. Towards the Keyboard of Oz: Learning Individual Soft-Keyboard Models from Raw Optical Sensor Data. In *2012 ACM international conference on Interactive tabletops and surfaces*, pages 163–172, 2012.

[8] Khaldoun Al Faraj, Mustapha Mojahid, and Nadine Vigouroux. BigKey: A Virtual Keyboard for Mobile Devices. In *Human-Computer*

Interaction. Ambient, Ubiquitous and Intelligent Interaction, volume 5612, pages 3–10. 2009.

[9] Leah Findlater and Jacob O. Wobbrock. Personalized input: improving ten-finger touchscreen typing through automatic adaptation. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 815–824, 2012.

[10] Leah Findlater, Jacob O. Wobbrock, and Daniel Wigdor. Typing on flat glass: examining ten-finger expert typing patterns on touch surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2453–2462, New York, NY, USA, 2011. ACM.

[11] Paul M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47(3):381–391, 1954.

[12] National Commission for the Protection of Huamn Subjects of Biomedical and Behavioral Research (US). The belmont report: Ethical principles and guidelines for the protection of human subjects of research, 1979.

[13] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.

[14] Vadim Fux, Michael G. Elizarov, and Sergey V. Kolomiets. Handheld electronic device with text disambiguation, 2006.

[15] Mayank Goel, Leah Findlater, and Jacob O. Wobbrock. WalkType: Using Accelerometer Data to Accommodate Situational Impairments in Mobile Touch Screen Text Entry. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 2687–2696, 2012.

[16] Mayank Goel, Jacob O. Wobbrock, and Shwetak N. Patel. GripSense: Using Built-In Sensors to Detect Hand Posture and Pressure on Commodity Mobile Phones. In *User Interface Software and Technology*, pages 545–554, 2012.

[17] Jun Gong and Peter Tarasewich. Alphabetically constrained keypad designs for text entry on mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, pages 211–220, New York, NY, USA, 2005. ACM.

[18] Joshua T. Goodman. A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403 – 434, 2001.

[19] Joshua T. Goodman, Gina Venolia, Keith Steury, and Chauncey Parker. Language modeling for soft keyboards. In *Proceedings of the 7th international conference on Intelligent user interfaces*, IUI '02, pages 194–195, New York, NY, USA, 2002. ACM.

[20] Dale L. Grover, Martin T. King, and Clifford A. Kushler. Reduced Keyboard Disambiguating Computer, 1998.

[21] Asela Gunawardana, Tim Paek, and Christopher Meek. Usability guided key-target resizing for soft keyboards. In *15th international conference on Intelligent user interfaces*, page 111, New York, New York, USA, 2010. ACM Press.

[22] Mark Hall, Frank Eibe, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. In *Intelligent Information Systems,1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, volume 11, pages 357–361, 2009.

[23] Niels Henze, Enrico Rukzio, and Susanne Boll. Observational and experimental investigation of typing behaviour using virtual keyboards for mobile devices. In *ACM annual conference on Human Factors in Computing Systems*, pages 2659–2668, New York, New York, USA, 2012. ACM Press.

[24] Johan Himberg, Jonna Häkkilä, Petri Kangas, and Jani Mäntyjärvi. On-line personalization of a touch screen based keyboard. In *8th international conference on Intelligent user interfaces*, page 77, New York, New York, USA, 2003. ACM Press.

[25] Eve Hoggan, Stephen A. Brewster, and Jody Johnston. Investigating the Effectiveness of Tactile Feedback for Mobile Touchscreens. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 1573–1582, 2008.

[26] Christian Holz and Patrick Baudisch. The generalized perceived input point model and how to double touch accuracy by extracting fingerprints. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 581–590, New York, NY, USA, 2010. ACM.

[27] Christian Holz and Patrick Baudisch. Understanding touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2501–2510, New York, NY, USA, 2011. ACM.

[28] Andreas Holzinger, Martin Holler, Martin J. Schedlbauer, and Berndt Urlesberger. An investigation of finger versus stylus input in medical scenarios. In *Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on*, pages 433–438, 2008.

[29] Christina L. James and Kelly M. Reischel. Text input for mobile devices: comparing model prediction to actual performance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '01, pages 365–371, New York, NY, USA, 2001. ACM.

[30] Michael N. Jones and Douglas J. K. Mewhort. Case-sensitive letter and bigram frequency counts from large-scale English corpora. *Behavior Research Methods, Instruments, & Computers*, 36(3):388–96, August 2004.

[31] S. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 35(3):400–401, 1987.

[32] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1137–1143. Morgan Kaufmann, 1995.

[33] Sarah Martina Kolly, Roger Wattenhofer, and Samuel Welten. A Personal Touch - Recognizing Users Based on Touch Screen Behavior. In *Third International Workshop on Sensing Applications on Mobile Phones*, 2012.

[34] Per-Ola Kristensson and Shumin Zhai. Relaxing stylus typing precision by geometric pattern matching. In *10th international conference on Intelligent user interfaces*, page 151, New York, New York, USA, 2005. ACM Press.

[35] Jennifer R. Kwapisz, Gary M. Weiss, and Samuel A. Moore. Activity Recognition using Cell Phone Accelerometers. In *ACM SIGKDD Explorations Newsletter*, volume 12, pages 74–82, 2010.

[36] John Laugesen and Yufei Yuan. What factors contributed to the success of apple's iphone? In *Mobile Business and 2010 Ninth Global Mobility Roundtable (ICMB-GMR)*, pages 91–99, 2010.

[37] Elizabeth D. Liddy. Natural Language Processing. In Marcel Decker, editor, *Encyclopedia of Library and Information Science*. 2nd edition, 2001.

[38] Stan J. (North Carolina State Univeristy) Liebowitz and Stephen E. (North Carolina state University) Margolis. The Fable of Keys. *Journal of Law and Economics*, 33(1):1–25, 1990.

[39] I. Scott MacKenzie. KSPC (keystrokes per character) as a characteristic of text entry techniques. In *Human Computer Interaction with Mobile Devices*, number i, pages 195–210. 2002.

[40] I. Scott MacKenzie, Hedy Kober, and Derek Smith. LetterWise: prefix-based disambiguation for mobile text input. In *14th annual ACM symposium on User interface software and technology*, volume 3, pages 111–120, 2001.

[41] Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K Gray, Joseph P Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, et al. Quantitative analysis of culture using millions of digitized books. *science*, 331(6014):176–182, 2011.

[42] Kevin P Murphy. Conjugate bayesian analysis of the gaussian distribution. *def*, 1($2\sigma 2$):16, 2007.

[43] Kevin Patrick Murphy. *Dynamic bayesian networks: representation, inference and learning*. PhD thesis, University of California, 2002.

[44] Takao Nakagawa and Hidetake Uwano. Usability Evaluation for Software Keyboard on High-Performance Mobile Devices. In *HCI International 2011 - Posters' Extended Abstracts*, pages 181–185. 2011.

[45] David L. Neuhoff. The Viterbi algorithm as an aid in text recognition (Corresp.). *IEEE Transactions on Information Theory*, 21(2):222–226, 1975.

[46] H. Ney, U. Essen, and R. Kneser. On the estimation of 'small' probabilities by leaving-one-out. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(12):1202–1212, 1994.

[47] Donald A. Norman and Diane Fisher. Why Alphabetic Keyboards Are Not Easy to Use: Keyboard Layout Doesn't Much Matter. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 24:509–519, 1982.

[48] Joan Noyes. The qwerty keyboard: a review. *International Journal of Man-Machine Studies*, 18(3):265 – 281, 1983.

[49] Tim F. Paymans, Jasper Lindenberg, and Mark Neerincx. Usability trade-offs for adaptive user interfaces: ease of use and learnability. In *Proceedings of the 9th international conference on Intelligent user interfaces*, IUI '04, pages 301–303, New York, NY, USA, 2004. ACM.

[50] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257 – 286, 1989.

[51] J.A. Robinson, V.W. Liang, J.A.M. Chambers, and C.L. MacKenzie. Computer user verification using login string keystroke dynamics. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 28(2):236–241, 1998.

[52] Dmitry Rudchenko, Tim Paek, and Eric Badger. Text Text Revolution: A Game that Improves Text Entry on Mobile Touchscreen Keyboards. In *Pervasive Computing, Lecture Notes in Computer Science*, volume 6696, pages 206–213. 2011.

[53] Bastian Schildbach and Enrico Rukzio. Investigating selection and reading performance on a mobile phone while walking. In *12th international conference on Human computer interaction with mobile devices and services*, page 93, New York, New York, USA, 2010. ACM Press.

[54] Andrew Sears, Min Lin, Julie Jacko, and Yan Xiao. When computers fade: Pervasive computing and situationally-induced impairments and disabilities. In *Proceedings of HCII 2003*, pages 1298–1302, 2003.

[55] Katie A. Siek, Yvonne Rogers, and Kay H. Connelly. Fat Finger Worries: How Older and Younger Users Physically Interact with PDAs. In *Human-Computer Interaction*, volume 3585, pages 267–280, 2005.

[56] Daniel Vogel and Patrick Baudisch. Shift: a technique for operating pen-based interfaces using touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 657–666, New York, NY, USA, 2007. ACM.

[57] Jacob O. Wobbrock. The Future of Mobile Device Research in HCI. In *Human-Computer Interaction*. 2006.

[58] Shumin Zhai, Michael Hunter, and BA Smith. The metropolis keyboard-an exploration of quantitative techniques for virtual keyboard design. *13th annual ACM symposium on User interface software and technology*, 2:119–128, 2000.

[59] Nan Zheng, Kun Bai, Hai Huang, and Haining Wang. You are how you touch: user verification on smartphones via tapping behaviors. Technical report, College of William & Mary - Department of Computer Science, 2012.

Appendix A

Architecture Diagrams

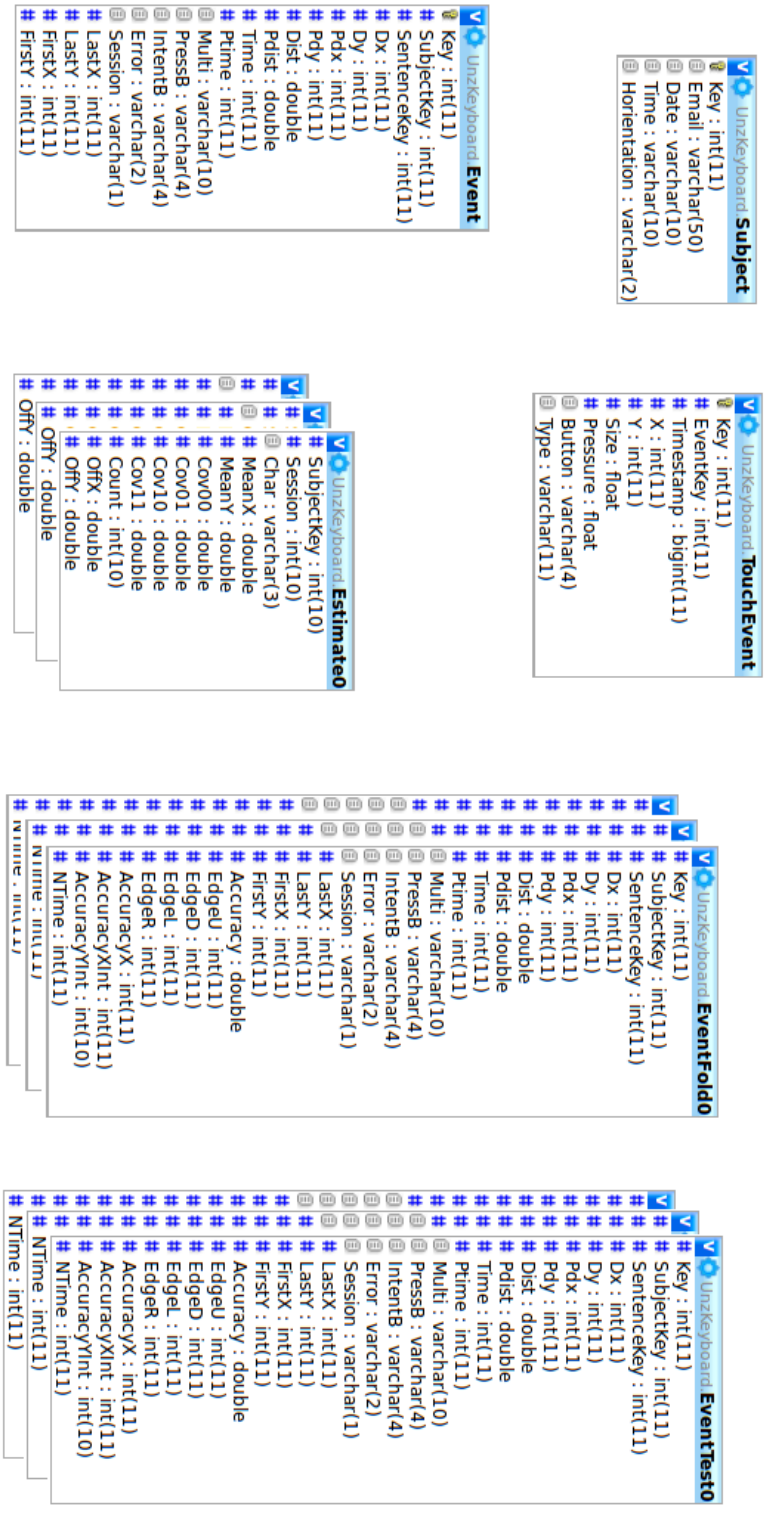


Figure A.1: The database schema

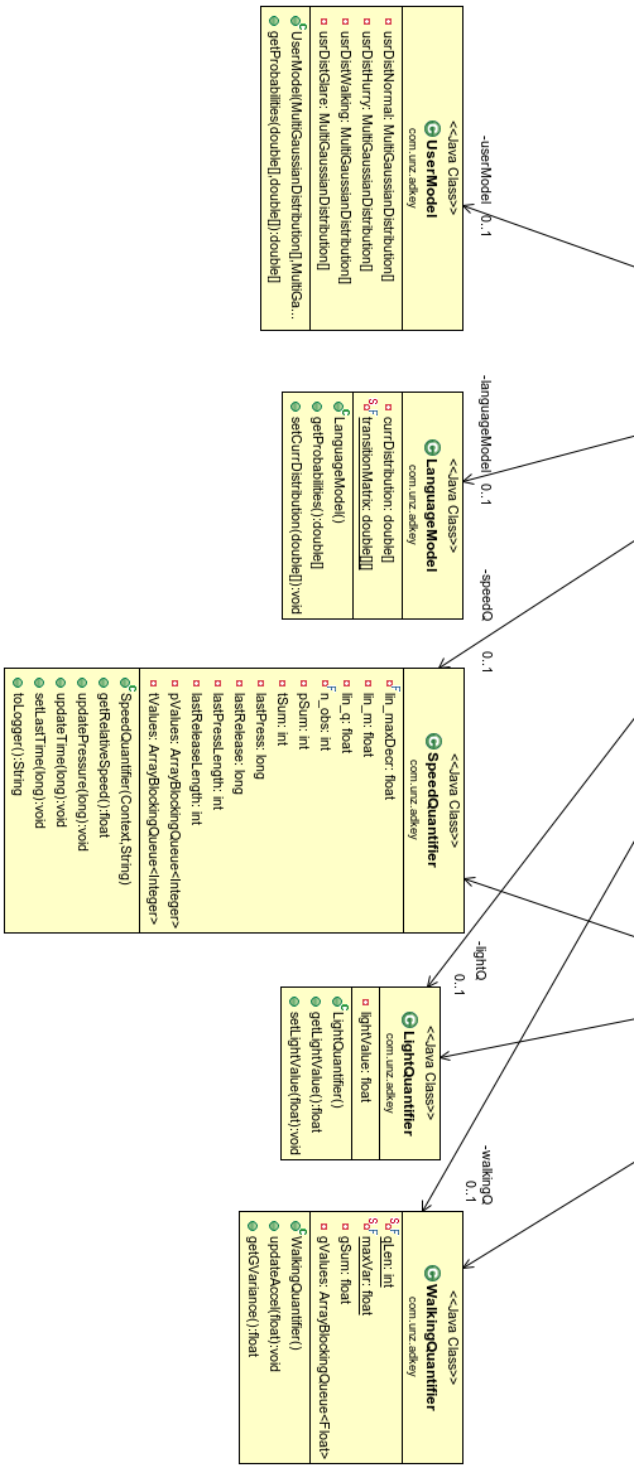


Figure A.2: AdKey class diagram

<<Java Class>>
UserSelect
com.unz.adkey
EXTRA_MESSAGE: String
UserSelect()
onCreate(Bundle):void
startKeyboard(View):void
<<Java Interface>>
SensorEventListener
android.hardware
onSensorChanged(SensorEvent):void
onAccuracyChanged(Sensor,int):void
<<Java Interface>>
OnTouchListener
android.view
onTouch(View,MotionEvent):boolean
<<Java Class>>
Model
com.unz.adkey
useLanguage: boolean
weights: double[]
Model(Context,String,SpeedQuantifier,LightQuantifier,WalkingQuantifier)
classifyThis(double[]):int
setUseLanguage(boolean):void
computeWeights():void
toLogger():CharSequence
-model
0..1
<<Java Class>>
Keyboard
com.unz.adkey
mKeys: int
sensorManager: SensorManager
accelerometer: Sensor
vibrator: Vibrator
keyArr: Button[]
form: EditText
preview: TextView
label: TextView
capitalize: boolean
currSentence: int
Keyboard()
onCreate(Bundle):void
onTouch(View,MotionEvent):boolean
onSensorChanged(SensorEvent):void
onAccuracyChanged(Sensor,int):void
onResume():void
onPause():void
nextPressed(View):void
handleKeystroke(int):void
drawLabel(int):void
mapKeys():void
-speedQuantifier
0..1
-lightQuantifier
0..1
-walkingQuantifier
0..1
-phraseGen
0..1
-logger
0..1
<<Java Class>>
Logger
com.unz.adkey
logFile: File
log: String
Logger()
appendLog(String):void
writeLog():void
<<Java Class>>
PhraseGenerator
com.unz.adkey
phrases: Vector<Vector<String>>
index: int
set: int
currentSentence: int
PhraseGenerator()
getCurrentSentence():int
getPhrase():String

Appendix B

Materials and Statistics

Used sentences

All the sentences used to retrieve data in the different sessions:

Type the following phrases

video camera with a zoom lens, this is a very good idea.
a steep learning curve in riding a unicycle
the first time he tried to swim, he almost flooded
important news always seems to be late
I put garbage in an abandoned mine
Mario, you are a capitalist pig
dormitory doors are locked at midnight
if you come home late, the doors are locked
most judges are very honest

Type the following phrases at the maximum speed you can achieve don't care too much of errors!

the insulation is not working
a steep learning curve in riding a unicycle
Mario, you are a capitalist pig
meet tomorrow in the lavatory
if you come home late, the doors are locked
I put garbage in an abandoned mine
I like baroque and classical music

the first time he tried to swim, he almost flooded
important news always seems to be late

Type the following phrases while walking

video camera with a zoom lens, this is a very good idea.
our life expectancy has increased
if you come home late, the doors are locked
I put garbage in an abandoned mine
the location of the crime
important news always seems to be late
a steep learning curve in riding a unicycle
the first time he tried to swim, he almost flooded
the accident scene is a shrine for fans

Type the following phrases with a glare on the screen (wait for the PI)

bring the offenders to justice
a steep learning curve in riding a unicycle
important news always seems to be late
a coupon for a free sample
the first time he tried to swim, he almost flooded
I put garbage in an abandoned mine
if you come home late, the doors are locked
Mario, you are a capitalist pig
the plug does not fit the socket

Key	Occurences
a	36.31
b	5.12
c	13.86
d	16.90
e	51.79
f	6.45
g	9.10
h	11.85
i	35.48
j	1.65
k	3.23
l	16.55
m	16.17
n	24.76
o	32.48
p	7.55
q	1.70
r	23.62
s	22.90
t	31.36
u	9.79
v	3.81
w	6.12
x	1.55
y	7.74
z	1.48
↑	2.33
⇐	23.38
,	4.68
_	79.31
.	1.67

Table B.1: Average occurrences of keys in each session

Key	MeanX	MeanY	VarX	VarY	CovXY	Count
a	-3.65	-0.61	81.20	46.18	7.30	13024
b	-2.18	3.41	24.66	16.35	1.99	2068
c	-0.54	2.40	50.41	29.92	-2.38	5544
d	0.60	1.76	33.61	29.44	1.144	6765
e	3.50	2.24	55.29	52.66	6.72	20834
f	1.14	2.05	26.10	20.62	0.62	2596
g	-1.53	2.75	29.86	26.26	2.14	3729
h	-0.61	2.23	29.50	23.98	0.28	4708
i	-1.63	2.14	52.11	45.65	-0.46	14366
j	-0.82	-0.13	19.00	13.65	-0.94	319
k	-3.19	1.83	19.37	15.25	0.15	1144
l	1.47	-0.17	52.20	35.71	-2.26	6457
m	-3.94	2.14	37.00	35.72	-1.90	6556
n	-4.73	3.29	57.61	36.76	-1.69	9900
o	-2.75	1.51	40.85	39.82	-0.15	13178
p	2.33	2.59	28.06	21.58	-2.84	3058
q	0.51	0.67	18.00	12.41	0.37	165
r	0.33	2.72	34.84	31.46	-0.25	9614
s	1.35	1.19	44.12	33.19	1.02	9306
t	0.67	3.34	43.87	38.56	1.91	12738
u	-0.86	2.95	31.56	25.45	-0.99	4026
v	0.24	1.89	19.75	15.33	0.35	1529
w	5.83	2.50	23.82	19.22	0.96	2464
x	2.80	0.16	17.64	13.06	0.20	165
y	-2.70	3.78	39.28	22.57	1.38	3179
z	0.42	1.30	18.05	12.93	-0.16	308
↑	-2.41	2.11	20.12	13.64	0.05	484
⇐	14.38	0.88	37.99	31.55	-2.22	8470
,	-1.46	5.43	22.69	21.73	0.03	1419
_	17.68	7.67	149.88	105.28	4.97	31977
.	0.34	3.88	17.68	13.48	-0.07	198

Table B.2: Aggregated parameter estimation over keys

Appendix C

AdKey Documentation

Package com.unz.adkey

Package Contents *Page*

Classes

C.1 Class Keyboard

The main activity of AdKey, an adaptive situation-aware keyboard. This class loads the GUI defined in `/res/layout/activity_keyboard.xml` and handles the sensor models SpeedQuantifier (in C.7, page 100), LightQuantifier (in C.3, page 95) and WalkingQuantifier (in C.10, page 105). It also handles the Model (in C.5, page 97) where the logic resides.

C.1.1 Declaration

public class Keyboard
extends Activity

C.1.2 Field summary

nKeys Number of keys of the keyboard.

C.1.3 Constructor summary

Keyboard()

C.1.4 Method summary

nextPressed(View) Gets the new sentence from the PhraseGenerator (in C.6, page 99) and updates the label `preview`.
onAccuracyChanged(Sensor, int)
onCreate(Bundle) Instantiates the GUI and the logic.
onPause() Unregister the accelerometer when the activity is paused.
onResume() Register the accelerometer when the activity is resumed.
onSensorChanged(SensorEvent) Handles the events retrieved from the accelerometer, updating the WalkingQuantifier (in C.10, page 105).
onTouch(View, MotionEvent) Handles a touch event on the sensitive area of the keyboard.

C.1.5 Fields

- public static final int **nKeys**
  - Number of keys of the keyboard.

C.1.6 Constructors

- **Keyboard**
  `public` **Keyboard**`()`

C.1.7 Methods

- **nextPressed**
  `public void` **nextPressed**`(View target)`
  - **Description**
    Gets the new sentence from the PhraseGenerator (in C.6, page 99) and updates the label `preview`. Finally, communicates to the Logger (in C.4, page 96) to flush the log to file.
  - **Parameters**
    * `target` –
- **onAccuracyChanged**
  `public void` **onAccuracyChanged**`(Sensor sensor, int accuracy)`
- **onCreate**
  `public void` **onCreate**`(Bundle savedInstanceState)`
  - **Description**
    Instantiates the GUI and the logic
- **onPause**
  `protected void` **onPause**`()`
  - **Description**
    Unregister the accelerometer when the activity is paused.
- **onResume**
  `protected void` **onResume**`()`
  - **Description**
    Register the accelerometer when the activity is resumed.

- **onSensorChanged**
  `public void` **onSensorChanged**`(SensorEvent event)`
  - **Description**
    Handles the events retrieved from the accelerometer, updating the WalkingQuantifier (in C.10, page 105).

- **onTouch**
  `public boolean` **onTouch**`(View v, MotionEvent event)`
  - **Description**
    Handles a touch event on the sensitive area of the keyboard. Model (in C.5, page 97) is responsible of classifying the retrieved touch, while SpeedQuantifier (in C.7, page 100) is updated. Finally, it updates the content of the `form`.

C.2 Class LanguageModel

Implements the language model for the keyboard. Logically, it stores the current belief state of the Markov Model, described as a probability distribution over the keys, and when it receives new evidence updates the belief state.

C.2.1 Declaration

public class LanguageModel
extends java.lang.Object

C.2.2 Constructor summary

LanguageModel() Sets the initial distribution over the keys as uniform.

C.2.3 Method summary

getProbabilities() Perform prediction on the next state given the current belief state.

setCurrDistribution(double[]) Updates the current belief state.

C.2.4 Constructors

- **LanguageModel**
  `public` **LanguageModel**`()`

– **Description**

Sets the initial distribution over the keys as uniform.

C.2.5 Methods

- **getProbabilities**
  `public double[]` **getProbabilities**`()`

  – **Description**

  Perform prediction on the next state given the current belief state.

  – **Returns** – the predicted probability distribution over the keys.

- **setCurrDistribution**
  `public void` **setCurrDistribution**`(double[]` **newDistribution**`)`

  – **Description**

  Updates the current belief state. It is also responsible of normalizing it to one.

  – **Parameters**

  * `newDistribution` – the new belief state.

C.3 Class LightQuantifier

Stores the value of the quantified level of light. No sensor handling because the API of the phone did not give us access to it.

C.3.1 Declaration

public class LightQuantifier
extends java.lang.Object

C.3.2 Constructor summary

LightQuantifier()

C.3.3 Method summary

getLightValue() Gets the quantified level of light.
setLightValue(float) Sets the quantified level of light.

C.3.4 Constructors

- **LightQuantifier**
  `public` **LightQuantifier**`()`

C.3.5 Methods

- **getLightValue**
  `public float` **getLightValue**`()`
  - **Description**
    Gets the quantified level of light. A value between 0 and 1.
  - **Returns** – light level.
- **setLightValue**
  `public void` **setLightValue**`(float` **lightValue**`)`
  - **Description**
    Sets the quantified level of light. Value must be between 0 and 1.
  - **Parameters**
    * `lightValue` – light level

C.4 Class Logger

Stores the log and handles writings to file. This class permits to retain the log of the occurred keystrokes in memory until they are flushed to file.

C.4.1 Declaration

public class Logger
extends java.lang.Object

C.4.2 Constructor summary

Logger() Creates the log file.

C.4.3 Method summary

appendLog(String) Appends the new log to the currently stored one.
writeLog() Flushes the content of the stored log to file.

C.4.4 Constructors

- **Logger**
  `public` **Logger()**
  - **Description**
    Creates the log file. To ensure file unicity, new file will have as name `System.currentTimeMillis()`.

C.4.5 Methods

- **appendLog**
  `public void` **appendLog**`(java.lang.String` **log)**
  - **Description**
    Appends the new log to the currently stored one. It appends it in a new line.
  - **Parameters**
    * `log` – the new log.
- **writeLog**
  `public void` **writeLog()**
  - **Description**
    Flushes the content of the stored log to file.

C.5 Class Model

The logical model of the application. It is responsible of classifying a touch location as a key, given knowledge about the UserModel , the current state of the LanguageModel (in C.2, page 94) and the state of the sensors. In order to know the latter, it shares a reference of a SpeedQuantifier (in C.7, page 100), WalkingQuantifier (in C.10, page 105) and LightQuantifier (in C.3, page 95) with the activity Keyboard (in C.1, page 92) that is responsible of updating them. Weights between them are computed.

C.5.1 Declaration

public class Model
extends java.lang.Object

C.5.2 Constructor summary

Model(Context, String, SpeedQuantifier, LightQuanti-

fier, WalkingQuantifier) Takes the reference of the sensor models and loads the user distributions.

C.5.3 Method summary

classifyThis(double[]) Given a stroke, returns the most likely key.

setUseLanguage(boolean) Enables or disables the language model.

toLog() Returns the state of the model in a format valid for the Logger (in C.4, page 96).

C.5.4 Constructors

- **Model**
  `public` **Model**`(Context` **context**`, java.lang.String` **userId**`, SpeedQuantifier` **sq**`, LightQuantifier` **lq**`, WalkingQuantifier` **wq**`)`
  - **Description**
    Takes the reference of the sensor models and loads the user distributions. User specific distribution are loaded from `res/values/usr_distributions.xml` and used to construct the UserModel . References to sensor models are stored as attributes.
  - **Parameters**
    - `context` – the application context.
    - `userId` – the user identifier.
    - `sq` – the speed quantifier.
    - `lq` – the light quantifier.
    - `wq` – the walking quantifier.

C.5.5 Methods

- **classifyThis**
  `public int` **classifyThis**`(double[]` **pos**`)`
  - **Description**
    Given a stroke, returns the most likely key. Computes probabilities of each key to be pressed given the coordinates, eventually joint with the language probabilities if the language model is enabled, and returns the index of the most likely key.

  - **Parameters**
    - `pos` – size 2 array containing the x and y coordinates in pixels.
  - **Returns** – the most likely key.

- **setUseLanguage**
  `public void` **setUseLanguage**`(boolean` **useLanguage**`)`
  - **Description**
    Enables or disables the language model.
  - **Parameters**
    - `useLanguage` – the language model selector.

- **toLog**
  `public java.lang.CharSequence` **toLog**`()`
  - **Description**
    Returns the state of the model in a format valid for the Logger (in C.4, page 96).
  - **Returns** – the log.

C.6 Class PhraseGenerator

Stores the sentences to be used in the experiment. The sentences are returned to the caller one by one.

C.6.1 Declaration

public class PhraseGenerator
extends java.lang.Object

C.6.2 Constructor summary

PhraseGenerator() Initialize the set of sentences.

C.6.3 Method summary

getCurrentSentence() Gets the counter over the current sentences.
getPhrase() Gets the next sentence.

C.6.4 Constructors

- **PhraseGenerator**
  `public` **PhraseGenerator**`()`
  - **Description**
    Initialize the set of sentences.

C.6.5 Methods

- **getCurrentSentence**
  `public int` **getCurrentSentence**`()`
  - **Description**
    Gets the counter over the current sentences. Used by the Logger (in C.4, page 96) to assign a unique ID to each sentence.
  - **Returns** –
- **getPhrase**
  `public java.lang.String` **getPhrase**`()`
  - **Description**
    Gets the next sentence. Gets the next sentence for the experiment and updates the counters.
  - **Returns** – a sentence.

C.7 Class SpeedQuantifier

Extimates the user writing speed. It uses a linear model between inter- and intra-stroke times to estimate the user relative typing speed. Both times are smoothed over the last 20 observations lower than 500 ms.

C.7.1 Declaration

public class SpeedQuantifier
extends java.lang.Object

C.7.2 Constructor summary

SpeedQuantifier(Context, String) Sets up the linear model parameters.

C.7.3 Method summary

getRelativeSpeed() Gets the estimated typing speed.
setLastTime(long) Updates the last release timestamp.
toLog() Returns the state of the quantifier in a format valid for the Logger (in C.4, page 96).
updatePressure(long) Updates the smoothed value of pressure times.
updateTime(long) Updates the smoothed value of inter-stroke times.

C.7.4 Constructors

- **SpeedQuantifier**
  `public` **SpeedQuantifier**`(Context` **context**`, java.lang.String` **userId**`)`
  - **Description**
    Sets up the linear model parameters. Paraeters are selected depending on whether the user is typing with one or both tumbs, as retrieved from `res/values/user_modality.xml`.
  - **Parameters**
    * `context` – the application context.
    * `userId` – the user identifier.

C.7.5 Methods

- **getRelativeSpeed**
  `public float` **getRelativeSpeed**`()`
  - **Description**
    Gets the estimated typing speed. The estimation is performed in terms of distance from the linear model of last observations, smoothed to 1. Value returned is between 0 and 1. It should be called before updating the model with the last touch, in order to get results dependent only on previous touches.
  - **Returns** – the estimated typing speed.
- **setLastTime**
  `public void` **setLastTime**`(long` **tTimeStamp**`)`

  - **Description**

    Updates the last release timestamp. This is called when the spacebar is pressed, because intra-words times do not count in the model in order to have more smoothed value, but is necessary to know when they happened to calculate the next inter-stroke time.

  - **Parameters**

    * `tTimeStamp` – the timestamp at which the spacebar was released.

- **toLog**
  `public java.lang.String` **`toLog()`**

  - **Description**

    Returns the state of the quantifier in a format valid for the Logger (in C.4, page 96).

  - **Returns** – the log.

- **updatePressure**
  `public void` **`updatePressure(long pTimeStamp)`**

  - **Description**

    Updates the smoothed value of pressure times. The intra-stroke time is updated only when pressure was not greater than 500ms. Also lastRelease is updated with the current timestamp.

  - **Parameters**

    * `pTimeStamp` – the timestamp at which the touch release occurred.

- **updateTime**
  `public void` **`updateTime(long tTimeStamp)`**

  - **Description**

    Updates the smoothed value of inter-stroke times. The inter-stroke time is updated only when pressure was not greater than 500ms. Also lastPressure is updated with the current timestamp.

  - **Parameters**

    * `pTimeStamp` – the timestamp at which the touch is initiated.

C.8 Class UserModel

Stores the user model and provides probability of events. User model is a collection of bivariate gaussians over the keys for each situation, when a touch is collected, given the weights of the situation is possible to retrieve the probability of each key being the intended key.

C.8.1 Declaration

public class UserModel
extends java.lang.Object

C.8.2 Constructor summary

UserModel(MultiGaussianDistribution[], MultiGaussianDistribution[], MultiGaussianDistribution[], MultiGaussianDistribution[]) Initializes the user model distributions.

C.8.3 Method summary

getProbabilities(double[], double[]) Gets the probability distribution over the keys.

C.8.4 Constructors

- **UserModel**
  `public` **UserModel**`(distribution.MultiGaussianDistribution[]` **usrDistNormal**`, distribution.MultiGaussianDistribution[]` **usrDistHurry**`, distribution.MultiGaussianDistribution[]` **usrDistWalking**`, distribution.MultiGaussianDistribution[]` **usrDistGlare**`)`
  - **Description**
    Initializes the user model distributions. Receives a series of MultiGaussianDistribution that composes the user model.
  - **Parameters**
    - `usrDistNormal` – distribution in normal condition.
    - `usrDistHurry` – distribution while in a hurry.
    - `usrDistWalking` – distribution when walking.
    - `usrDistGlare` – distribution when there there is glare on the screen.

C.8.5 Methods

- **getProbabilities**
  `public double[] getProbabilities(double[] weights, double[] pos)`

  - **Description**

    Gets the probability distribution over the keys. When a touch is performed, this method computes the probability of the touch of belonging to each key, as a mixture of bivariate gaussians of weighted by `weights`.

  - **Parameters**

    * `weights` – the weight to assign to each distribution.
    * `pos` – the location where touch was performed.

  - **Returns** – probability distribution over the keys.

C.9 Class UserSelect

The application starting activity. This activity is created when the application is run, and it serves to select the user performing the experiment, that is communicated to the created Keyboard (in C.1, page 92) activity.

C.9.1 Declaration

public class UserSelect
extends Activity

C.9.2 Field summary

EXTRA_MESSAGE

C.9.3 Constructor summary

UserSelect()

C.9.4 Method summary

onCreate(Bundle) Initialize the graphic elements.
startKeyboard(View) Starts the Keyboard (in C.1, page 92) activity.

C.9.5 Fields

- public static final java.lang.String **EXTRA_MESSAGE**

C.9.6 Constructors

- **UserSelect**
  `public` **UserSelect**`()`

C.9.7 Methods

- **onCreate**
  `public void` **onCreate**`(Bundle` **savedInstanceState**`)`

  – **Description**

  Initialize the graphic elements.

- **startKeyboard**
  `public void` **startKeyboard**`(View` **view**`)`

  – **Description**

  Starts the Keyboard (in C.1, page 92) activity. Passing to it as a message the content in the `editText`.

  – **Parameters**

  * `view` – the button to start the activity.

C.10 Class WalkingQuantifier

The model that quantifies the user walking speed. This stores readings of values from the device y-axis accelerometer and uses them to compute their standard deviation as an estimate of the user walking speed.

C.10.1 Declaration

public class WalkingQuantifier
extends java.lang.Object

C.10.2 Constructor summary

WalkingQuantifier() Class constructor.

C.10.3 Method summary

getGVariance() Return the variance over the last measured acceleration values.

updateAccel(float) Updates the acceleration values Get the y-acceleration as measured by the Keyboard activity and updates the stored values (only y-axis considered)

C.10.4 Constructors

- **WalkingQuantifier**
  `public` **WalkingQuantifier**`()`

  - **Description**

    Class constructor.

C.10.5 Methods

- **getGVariance**
  `public float` **getGVariance**`()`

  - **Description**

    Return the variance over the last measured acceleration values. The mean and standard deviation of the 50 last measured values are computed and the latter is normalized to one.

  - **Returns** – the variance over acceleration normalized to one.

- **updateAccel**
  `public void` **updateAccel**`(float value)`

  - **Description**

    Updates the acceleration values Get the y-acceleration as measured by the Keyboard activity and updates the stored values (only y-axis considered)

  - **Parameters**

    * `value` – the values acquired from the accelerometer

Package com.unz.adkey.distribution

Package Contents *Page*

Interfaces

Classes

This package implements various pseudo-random distributions.

C.11 Interface MultiRandomDistribution

This interface must be implemented by all the package's classes implementing a multi-variate random distribution.

C.11.1 Declaration

public interface MultiRandomDistribution

extends java.io.Serializable

C.11.2 All known subinterfaces

MultiGaussianDistribution (in C.14, page 111)

C.11.3 All classes known to implement interface

MultiGaussianDistribution (in C.14, page 111)

C.11.4 Method summary

dimension() Returns the dimension of the vectors handled by this random distribution.

generate() Generates a pseudo-random vector.

probability(double[]) Returns the probability (density) of a given vector.

C.11.5 Methods

- **dimension**
  `int` **dimension**`()`
  - **Description**
    Returns the dimension of the vectors handled by this random distribution.
  - **Returns** – The generated vectors' dimension.
- **generate**
  `double[]` **generate**`()`
  - **Description**
    Generates a pseudo-random vector. The vectors generated by this function must follow the pseudo-random distribution described by the object that implements it.
  - **Returns** – A pseudo-random vector.
- **probability**
  `double` **probability**`(double[] v)`
  - **Description**
    Returns the probability (density) of a given vector.
  - **Parameters**
    * `v` – A vector.
  - **Returns** – The probability of the vector `v`.

C.12 Interface RandomDistribution

This interface must be implemented by all the package's classes implementing a mono-variate random distribution. Distributions are not mutable.

C.12.1 Declaration

public interface RandomDistribution
extends java.io.Serializable

C.12.2 All known subinterfaces

GaussianDistribution (in C.13, page 110)

C.12.3 All classes known to implement interface

GaussianDistribution (in C.13, page 110)

C.12.4 Method summary

generate() Generates a pseudo-random number.
probability(double) Returns the probability (density) of a given number.

C.12.5 Methods

- **generate**
  `double` **generate**`()`
  - **Description**

    Generates a pseudo-random number. The numbers generated by this function are drawn according to the pseudo-random distribution described by the object that implements it.
  - **Returns** – A pseudo-random number.

- **probability**
  `double` **probability**`(double n)`
  - **Description**

    Returns the probability (density) of a given number.
  - **Parameters**
    - `n` – A number.

C.13 Class GaussianDistribution

This class implements a Gaussian distribution.

C.13.1 Declaration

public class GaussianDistribution
extends java.lang.Object
implements RandomDistribution

C.13.2 Constructor summary

GaussianDistribution() Creates a new pseudo-random, Gaussian distribution with zero mean and unitary variance.
GaussianDistribution(double, double) Creates a new pseudo-random, Gaussian distribution.

C.13.3 Method summary

generate()
mean() Returns this distribution's mean value.
probability(double)
variance() Returns this distribution's variance.

C.13.4 Constructors

- **GaussianDistribution**
  `public` **GaussianDistribution**`()`
  - **Description**
    Creates a new pseudo-random, Gaussian distribution with zero mean and unitary variance.
- **GaussianDistribution**
  `public` **GaussianDistribution**`(double` **mean**`, double` **variance**`)`
  - **Description**
    Creates a new pseudo-random, Gaussian distribution.
  - **Parameters**
    * `mean` – The mean value of the generated numbers.
    * `variance` – The variance of the generated numbers.

C.13.5 Methods

- **generate**
  `double` **generate**`()`
  - **Description copied from RandomDistribution** (**in C.12, page 109**)
    Generates a pseudo-random number. The numbers generated by this function are drawn according to the pseudo-random distribution described by the object that implements it.
  - **Returns** – A pseudo-random number.

- **mean**
  `public double` **mean**`()`
  - **Description**
    Returns this distribution's mean value.
  - **Returns** – This distribution's mean value.

- **probability**
  `double` **probability**`(double n)`
  - **Description copied from RandomDistribution** (**in C.12, page 109**)
    Returns the probability (density) of a given number.
  - **Parameters**
    * `n` – A number.

- **variance**
  `public double` **variance**`()`
  - **Description**
    Returns this distribution's variance.
  - **Returns** – This distribution's variance.

C.14 Class MultiGaussianDistribution

This class implements a multi-variate Gaussian distribution.

C.14.1 Declaration

public class MultiGaussianDistribution
extends java.lang.Object
implements MultiRandomDistribution

C.14.2 Constructor summary

MultiGaussianDistribution(double[], double[][]) Creates a new pseudo-random, multivariate gaussian distribution.

MultiGaussianDistribution(int) Creates a new pseudo-random, multivariate gaussian distribution with zero mean and identity covariance.

C.14.3 Method summary

covariance() Returns (a copy of) this distribution's covariance matrix.

covarianceDet() Returns the covariance matrix determinant.

dimension()

generate() Generates a pseudo-random vector according to this distribution.

mean() Returns (a copy of) this distribution's mean vector.

probability(double[])

C.14.4 Constructors

- **MultiGaussianDistribution**
  `public` **MultiGaussianDistribution**`(double[] mean, double[][] covariance)`
  - **Description**
    Creates a new pseudo-random, multivariate gaussian distribution.
  - **Parameters**
    - `mean` – The mean vector of the generated numbers. This array is copied.
    - `covariance` – The covariance of the generated numbers. This array is copied. `covariance[r][c]` is the element at row `r` and column `c`.
- **MultiGaussianDistribution**
  `public` **MultiGaussianDistribution**`(int dimension)`

- **Description**

  Creates a new pseudo-random, multivariate gaussian distribution with zero mean and identity covariance.

- **Parameters**

  * `dimension` – This distribution dimension.

C.14.5 Methods

- **covariance**
  `public double[][] covariance()`

  - **Description**

    Returns (a copy of) this distribution's covariance matrix.

  - **Returns** – This distribution's covariance matrix.

- **covarianceDet**
  `public double covarianceDet()`

  - **Description**

    Returns the covariance matrix determinant.

  - **Returns** – The covariance matrix determinant.

- **dimension**
  `int dimension()`

  - **Description copied from MultiRandomDistribution** (**in C.11, page 107**)

    Returns the dimension of the vectors handled by this random distribution.

  - **Returns** – The generated vectors' dimension.

- **generate**
  `public double[] generate()`

  - **Description**

    Generates a pseudo-random vector according to this distribution. The vectors are generated using the Cholesky decomposition of the covariance matrix.

  - **Returns** – A pseudo-random vector.

- **mean**
  `public double[] mean()`

  - **Description**

    Returns (a copy of) this distribution's mean vector.

  - **Returns** – This distribution's mean vector.

- **probability**
  `double` **probability**`(double[] v)`

  - **Description copied from MultiRandomDistribution** (**in C.11, page 107**)

    Returns the probability (density) of a given vector.

  - **Parameters**

    * `v` – A vector.

  - **Returns** – The probability of the vector `v`.

Appendix D

Code Snippets

Learning process

The sequent code snippets were used for the training and evaluation session

D.1 Bayesian learner

A reusable class to perform Bayesian learning of a multivariate normal

```
1  import numpy as np
2
3  class MultiNormalBayesLearner:
4    """Computes the maximum a posteriori of a multivariate
         Normal given its prior and the observation"""
5
6    def __init__ (self, mean_prior, cov_prior, kappa, nu):
7      self.mean_prior = mean_prior.astype(float)
8      self.cov_prior = cov_prior.astype(float)
9      self.kappa = float(kappa)
10     self.nu = float(nu)
11
12   def train (self, train_sample):
13     n = len(train_sample)
14     if n != 0:
15       p = len(train_sample[0])
16     else:
```

```
17        p = 0
18      mean_sample = np.zeros((p,1)).astype(float)
19      cov_sample = np.zeros((p,p)).astype(float)
20      for i in range(n):
21        mean_sample += train_sample[i]
22      mean_sample /= n
23      for i in range(n):
24        cov_sample += (train_sample[i] - mean_sample) * ((
              train_sample[i] - mean_sample).transpose())
25      self.mean_estimated = (self.kappa * self.mean_prior + n *
            mean_sample) / (self.kappa + n)
26      const = self.kappa * n / (self.kappa + n)
27      self.cov_estimated = (self.cov_prior + cov_sample + const
            * (mean_sample - self.mean_prior) * "
28        ((mean_sample - self.mean_prior).transpose())) / (self.
              nu + n + p + 1)
```

D.2 Training algorithm

The selection of training samples and storage of the retrieved distribution

```
1  import learner as ln
2  import numpy as np
3  import MySQLdb as mdb
4  import random
5  from scipy.stats import norm
6  import math
7
8  def train(fold):
9    centx = (('q',16),('w',16+32),('e',16+32*2),('r',16+32*3),(
         't',16+32*4),('y',16+32*5),('u',16+32*6),('i',16+32*7),(
         'o',16+32*8),('p',16+32*9),
10   ('a',32),('s',32*2),('d',32*3),('f',32*4),('g',32*5),('h'
         ,32*6),('j',32*7),('k',32*8),('l',32*9),
11   ('shf',24),('z',32*2),('x',32*3),('c',32*4),('v',32*5),('b'
         ,32*6),('n',32*7),('m',32*8),('bks',32*9),
12   (',',48+16),('_',160),('.',256))
13   centy = (('q',27.5),('w',27.5),('e',27.5),('r',27.5),('t'
         ,27.5),('y',27.5),('u',27.5),('i',27.5),('o',27.5),('p'
         ,27.5),
14     ('a',82.5),('s',82.5),('d',82.5),('f',82.5),('g',82.5),('
           h',82.5),('j',82.5),('k',82.5),('l',82.5),
15     ('shf',137.5),('z',137.5),('x',137.5),('c',137.5),('v'
           ,137.5),('b',137.5),('n',137.5),('m',137.5),('bks'
           ,137.5),
16     (',',192.5),('_',192.5),('.',192.5))
17
18   kappa = 2
```

```
19    nu = 500
20    prior_mean = np.zeros((2,1))
21    prior_cov = np.matrix([[16.8065*(nu-3),0],[0,12.2008*(nu-3)
          ]])
22    nbl = ln.MultiNormalBayesLearner(prior_mean, prior_cov,
          kappa, nu)
23
24    con = mdb.connect('localhost', 'root', 'inz', 'UnzKeyboard'
          )
25    with con:
26      cur = con.cursor()
27      cur.execute("TRUNCATE TABLE Estimate%s", fold)
28
29      cur.execute("SELECT MAX(`Key`) FROM Subject")
30      vaaa = cur.fetchone()[0]
31      for subj in range(1,vaaa + 1):
32        cur.execute("SELECT MAX(Session) FROM Event2 WHERE
              SubjectKey = %s", subj)
33        for sess in range(1,6):
34          chars = map(chr, range(97, 123))
35          chars.extend(['shf','bks',',','-','.'])
36          for char in chars:
37            cur.execute("SELECT AccuracyXInt, AccuracyYInt FROM
                  EventFold%s WHERE Session = %s AND IntentB = %s
                  AND LastX <> 0 AND Multi = 'N'", (fold, sess,
                  char))
38            strokes = cur.fetchall()
39            strokesarr = []
40            for stroke in strokes:
41              strokesarr.append(np.matrix(stroke).transpose())
42            if len(strokesarr) != 0:
43              nbl.train(strokesarr)
44              est_mean = nbl.mean_estimated
45              est_cov = nbl.cov_estimated
46            else:
47              est_mean = prior_mean
48              est_cov = prior_cov / (nu-3)
49            cur.execute("INSERT INTO Estimate%s VALUES (%s,%s,%
                  s,%s,%s,%s,%s,%s,%s,%s,NULL,NULL)", "
50              (fold, subj, sess, char, est_mean.item(0),
                    est_mean.item(1), est_cov.item(0), est_cov.
                    item(1), est_cov.item(2), est_cov.item(3), len
                    (strokesarr)))
51      for char in chars:
52        for i in range(len(chars)):
53          if centx[i][0] == char:
54            cur.execute("UPDATE Estimate%s SET OffX = %s, OffY
                  = %s WHERE `Char` = %s", (fold, centx[i][1],
                  centy[i][1], char))
```

D.3 Evaluation algorithm

The simulation of the training set over the obtained model.

```
def multivariate_pdf(x,y,mx,my,vx,vy,cov):
  mean = np.array([mx,my])
  cov = np.array([[vx,cov],[cov,vy]])
  vector = np.array([x,y])
  quadratic_form = np.dot(np.dot(vector-mean,np.linalg.inv(
      cov)),np.transpose(vector-mean))
  return np.exp(-.5 * quadratic_form)/ (2*np.pi * np.linalg.
      det(cov))

def test(fold, ses,use):
  con = mdb.connect('localhost', 'root', 'inz', 'UnzKeyboard'
      )
  with con:
    cur = con.cursor()
    if use == 4:
      cur.execute("SELECT LastX, LastY, SubjectKey, Session,
          IntentB , PressB, `Key` FROM EventTest%s WHERE
          Session = %s  AND SubjectKey > 6", (fold,ses))
    else:
      cur.execute("SELECT LastX, LastY, SubjectKey, Session,
          IntentB , PressB, `Key` FROM EventTest%s WHERE
          Session = %s  AND SubjectKey <> 1 AND SubjectKey <>
          5", (fold,ses))
    strokes = cur.fetchall()
    count, error, ok , olderror, corrected, wronged = 0, 0,
        0, 0, 0, 0
    for stroke in strokes:
      cur.execute("SELECT `Char`, MeanX, MeanY, Cov00, Cov11,
          OffX, OffY, Cov01 FROM EstimateMean WHERE
          SubjectKey = %s AND Session = %s","
        (stroke[2], use))
      pdfs = cur.fetchall()
      currmaxval = 0.0
      currchar = 'a'
      for pdf in pdfs:
        probability = multivariate_pdf(stroke[0]-pdf[5],
            stroke[1]-pdf[6],pdf[1],pdf[2],pdf[3],pdf[4],pdf
            [7])
        if probability > currmaxval:
          currmaxval = probability
          currchar = pdf[0]
      if currchar != stroke[4]:
        error +=1
      else:
        ok +=1
```

```
33          count+=1
34          if stroke[4] != stroke[5]:
35            olderror += 1
36          if stroke[4] != stroke[5] and stroke[4] == currchar:
37            corrected += 1
38          if stroke[4] == stroke[5] and stroke[4] != currchar:
39            wronged += 1
40        return (olderror, error)
```

AdKey

Main classes of AdKey

D.4 Class Model

```
1  package com.unz.adkey;
2
3  import android.content.Context;
4  import android.content.res.Resources;
5  import android.content.res.TypedArray;
6  import android.view.MotionEvent;
7
8  import com.unz.adkey.distribution.MultiGaussianDistribution;
9
10 /**
11  * The logical model of the application.
12  * It is responsible of classifying a touch location as a key
        , given knowledge
13  * about the {@link #userModel UserModel}, the current state
        of the {@link #LanguageModel LanguageModel} and the state
14  * of the sensors. In order to know the latter, it shares a
        reference of a {@link #SpeedQuantifier SpeedQuantifier},
15  * {@link #WalkingQuantifier WalkingQuantifier} and {@link #
        LightQuantifier LightQuantifier}
16  * with the activity {@link #Keyboard Keyboard} that
17  * is responsible of updating them. Weights between them are
        computed.
18  * @author Michael Angeleri - michael.angeleri@mail.polimi.it
19  *
```

```
20  */
21 public class Model {
22
23   private boolean useLanguage = true;
24
25   private UserModel userModel;
26   private LanguageModel languageModel;
27
28   private SpeedQuantifier speedQ;
29   private LightQuantifier lightQ;
30   private WalkingQuantifier walkingQ;
31
32   private double[] weights;
33
34   /**
35    * Takes the reference of the sensor models and loads the
          user distributions.
36    * User specific distribution are loaded from <code>res/
          values/usr_distributions.xml</code> and used to
37    * construct the {@link #userModel UserModel}. References
          to sensor models are stored as attributes.
38    * @param context the application context.
39    * @param userId the user identifier.
40    * @param sq the speed quantifier.
41    * @param lq the light quantifier.
42    * @param wq the walking quantifier.
43    */
44   public Model(Context context, String userId,
          SpeedQuantifier sq, LightQuantifier lq,
          WalkingQuantifier wq) {
45     //initializing the user model
46     Resources res = context.getResources();
47     TypedArray tmx = res.obtainTypedArray(R.array.meanx);
48     TypedArray tmy = res.obtainTypedArray(R.array.meany);
49     TypedArray tvx = res.obtainTypedArray(R.array.variancex);
50     TypedArray tvy = res.obtainTypedArray(R.array.variancey);
51     TypedArray tcov = res.obtainTypedArray(R.array.covariance
           );
52
53     int offset = Integer.parseInt(userId) * Keyboard.nKeys *
           4;
54     MultiGaussianDistribution[] usrDistNormal = new
           MultiGaussianDistribution[Keyboard.nKeys];
55     MultiGaussianDistribution[] usrDistHurry = new
           MultiGaussianDistribution[Keyboard.nKeys];
56     MultiGaussianDistribution[] usrDistWalking = new
           MultiGaussianDistribution[Keyboard.nKeys];
57     MultiGaussianDistribution[] usrDistGlare = new
           MultiGaussianDistribution[Keyboard.nKeys];
```

```
58      for (int i = 0; i < Keyboard.nKeys * 4 ; i++) {
59        double[] mean = {(double)tmx.getFloat(i + offset , 0), (
              double)tmy.getFloat(i + offset , 0)};
60        double[][] cov = {{(double)tvx.getFloat(i + offset , 0),
               (double)tcov.getFloat(i + offset , 0)},
61                    {(double)tcov.getFloat(i + offset , 0), (
                          double)tvy.getFloat(i + offset , 0)}};
62        if (i < Keyboard.nKeys) {
63          usrDistNormal[i] = new MultiGaussianDistribution(mean
                , cov);
64        } else if (i < Keyboard.nKeys *2 ) {
65          usrDistHurry[i - Keyboard.nKeys] = new
                MultiGaussianDistribution(mean, cov);
66        } else if (i < Keyboard.nKeys * 3) {
67          usrDistWalking[i - Keyboard.nKeys * 2] = new
                MultiGaussianDistribution(mean, cov);
68        } else {
69          usrDistGlare[i - Keyboard.nKeys * 3] = new
                MultiGaussianDistribution(mean, cov);
70        }
71      }
72      userModel = new UserModel(usrDistNormal, usrDistHurry,
            usrDistWalking, usrDistGlare);
73      languageModel = new LanguageModel();
74      speedQ = sq;
75      lightQ = lq;
76      walkingQ = wq;
77    }
78
79    /**
80     * Given a stroke, returns the most likely key.
81     * Computes probabilities of each key to be pressed given
            the coordinates, eventually
82     * joint with the language probabilities if the language
            model is enabled, and returns
83     * the index of the most likely key.
84     * @param pos size 2 array containing the x and y
            coordinates in pixels.
85     * @return the most likely key.
86     */
87    public int classifyThis(double[] pos, int type) {
88      computeWeights();
89      double[] usrProb = userModel.getProbabilities(weights,
            pos);
90      if (useLanguage) {
91        double[] langProb = languageModel.getProbabilities();
92        for (int i = 0; i < Keyboard.nKeys; i++) {
93          usrProb[i] *= langProb[i];
94        }
```

```
95        if (type == MotionEvent.ACTION_UP || type ==
              MotionEvent.ACTION_POINTER_1_UP) {
96          languageModel.setCurrDistribution(usrProb);
97        }
98      }
99      double max = 0;
100     int index = -1;
101     for (int i = 0; i < Keyboard.nKeys; i++) {
102       if (usrProb[i] > max) {
103         max = usrProb[i];
104         index = i;
105       }
106     }
107     return index;
108   }
109
110   /**
111    * Enables or disables the language model.
112    * @param useLanguage the language model selector.
113    */
114   public void setUseLanguage(boolean useLanguage) {
115     this.useLanguage = useLanguage;
116   }
117
118   private void computeWeights() {
119     double w = walkingQ.getGVariance();
120     double l = lightQ.getLightValue();
121     double s = speedQ.getRelativeSpeed();
122     double n = 1 - s;
123     double takeAway = Math.max(w, l);
124     double effTakeAway;
125     if (takeAway > n) {
126       effTakeAway = n;
127       n = 0;
128     } else {
129       effTakeAway = takeAway;
130       n -= takeAway;
131     }
132     w += w * effTakeAway / (w + l);
133     l += l * effTakeAway / (w + l);
134     double sum = w + l + s + n;
135     weights = new double[] { n / sum, s / sum, w / sum, l /
            sum };
136   }
137
138   /**
139    * Returns the state of the model in a format valid for the
           {@link #Logger Logger}.
140    * @return the log.
```

```
141     */
142   public CharSequence toLog() {
143     String log = "";
144     log += speedQ.toLog() + ",";
145     for (int i = 0; i < weights.length; i++) {
146       log += String.valueOf(weights[i]) + ",";
147     }
148     return log;
149   }
150 }
```

D.5 Class UserModel

```
1  package com.unz.adkey;
2
3  import com.unz.adkey.distribution.MultiGaussianDistribution;
4
5  /**
6   * Stores the user model and provides probability of events.
7   * User model is a collection of bivariate gaussians over the
        keys for each situation,
8   * when a touch is collected, given the weights of the
        situation is possible to
9   * retrieve the probability of each key being the intended
        key.
10  * @author Michael Angeleri - michael.angeleri@mail.polimi.it
11  *
12  */
13 public class UserModel {
14   private MultiGaussianDistribution[] usrDistNormal;
15   private MultiGaussianDistribution[] usrDistHurry;
16   private MultiGaussianDistribution[] usrDistWalking;
17   private MultiGaussianDistribution[] usrDistGlare;
18
19   /**
20    * Initializes the user model distributions.
21    * Receives a series of {@link #MultiGaussianDistribution
        MultiGaussianDistribution} that composes the user model
        .
22    * @param usrDistNormal distribution in normal condition.
23    * @param usrDistHurry distribution while in a hurry.
24    * @param usrDistWalking distribution when walking.
25    * @param usrDistGlare distribution when there there is
        glare on the screen.
26    */
27   public UserModel(MultiGaussianDistribution[] usrDistNormal,
        MultiGaussianDistribution[] usrDistHurry,
```

```
28        MultiGaussianDistribution[] usrDistWalking,
              MultiGaussianDistribution[] usrDistGlare) {
29      this.usrDistNormal = usrDistNormal;
30      this.usrDistHurry = usrDistHurry;
31      this.usrDistWalking = usrDistWalking;
32      this.usrDistGlare = usrDistGlare;
33    }
34
35    /**
36     * Gets the probability distribution over the keys.
37     * When a touch is performed, this method computes the
            probability of the touch of
38     * belonging to each key, as a mixture of bivariate
            gaussians of weighted by
39     * <code>weights</code>.
40     * @param weights the weight to assign to each distribution
            .
41     * @param pos the location where touch was performed.
42     * @return probability distribution over the keys.
43     */
44    public double[] getProbabilities(double[] weights, double[]
          pos) {
45      double[] p = new double[Keyboard.nKeys];
46      for (int i = 0; i < Keyboard.nKeys; i++) {
47        p[i] = usrDistNormal[i].probability(pos) * weights[0] +
              usrDistHurry[i].probability(pos) * weights[1] +
48            usrDistWalking[i].probability(pos) * weights[2] +
                  usrDistGlare[i].probability(pos) * weights[3];
49      } /*
50      for (int i = 0; i < Keyboard.nKeys; i++) {
51        p[i] /= sum;
52      } */
53      return p;
54    }
55  }
```

D.6 Class LanguageModel

```
1  package com.unz.adkey;
2
3  import java.util.Arrays;
4
5  /**
6   * Implements the language model for the keyboard.
7   * Logically, it stores the current belief state of the
        Markov Model, described as a probability
8   * distribution over the keys, and when it receives new
        evidence updates the belief state.
```

```
 9  * @author Michael Angeleri - michael.angeleri@mail.polimi.it
10  *
11  */
12 public class LanguageModel {
13   private double[] currDistribution = new double[Keyboard.
         nKeys];
14
15   //smoothed language model
16   private final static double[][] transitionMatrix =
17     { { 0.00027900242013 , 0.0171700770224 , 0.0345762037089
           , 0.0312858509044 , 0.00180978815082 ,
           0.00698088235388 , 0.0189758609283 , 0.00145240564405
           , 0.0422314519293 , 0.00145240564405 , 0.0104142112833
            , 0.0850436152805 , 0.0256147035389 , 0.171256725142
           , 0.000420363122443 , 0.0155361361754 ,
           0.000493286080146 , 0.10387248762 , 0.0769506532858 ,
           0.126870124229 , 0.0115094745871 , 0.0189758609283 ,
           0.00631657157969 , 0.00267298216645 , 0.0283086165684
           , 0.00177395361215 , 0.031551261219 , 0.031551261219 ,
            0.031551261219 , 0.031551261219 , 0.031551261219 }
           ... };
18
19   /**
20    * Sets the initial distribution over the keys as uniform
21    */
22   public LanguageModel() {
23     Arrays.fill(currDistribution, 1 / (double) Keyboard.nKeys
           );
24   }
25
26   /**
27    * Perform prediction on the next state given the current
         belief state.
28    * @return the predicted probability distribution over the
         keys.
29    */
30   public double[] getProbabilities() {
31     double[] result = new double[Keyboard.nKeys];
32     double totalSum = 0;
33     for (int i = 0; i < Keyboard.nKeys; i++) {
34       double sum = 0;
35       for (int j = 0; j < Keyboard.nKeys; j++) {
36         sum += transitionMatrix[j][i] * currDistribution[j];
37       }
38       result[i] = sum;
39       totalSum += sum;
40     }
41     for (int i = 0; i < Keyboard.nKeys; i++) {
42       result[i] /= totalSum;
```

```
43      }
44      return result;
45    }
46
47    /**
48     * Updates the current belief state.
49     * It is also responsible of normalizing it to one.
50     * @param newDistribution the new belief state.
51     */
52    public void setCurrDistribution(double[] newDistribution) {
53      double sum = 0;
54      for (int i = 0; i < Keyboard.nKeys; i++) {
55        sum += newDistribution[i];
56      }
57      for (int i = 0; i < Keyboard.nKeys; i++) {
58        currDistribution[i] = newDistribution[i] / sum;
59      }
60    }
61  }
```

D.7 Class SpeedQuantifier

```
1  package com.unz.adkey;
2
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  import android.content.Context;
6  import android.content.res.Resources;
7
8  /**
9   * Extimates the user writing speed.
10  * It uses a linear model between inter- and intra-stroke
        times
11  * to estimate the user relative typing speed. Both times are
        smoothed over
12  * the last 20 observations lower than 500 ms.
13  * @author Michael Angeleri - michael.angeleri@mail.polimi.it
14  *
15  */
16 public class SpeedQuantifier {
17   //linear model parameters
18   private static final float lin_maxDecr = (float) 0.50;
19   private float lin_m;
20   private float lin_q;
21
22   private static final int n_obs = 20;
23
24   private int pSum, tSum;
```

```
25    private long lastPress, lastRelease;
26    private int lastPressLength, lastReleaseLength;
27    private ArrayBlockingQueue<Integer> pValues = new
          ArrayBlockingQueue<Integer>(n_obs);
28    private ArrayBlockingQueue<Integer> tValues = new
          ArrayBlockingQueue<Integer>(n_obs);
29
30
31    /**
32     * Sets up the linear model parameters.
33     * Paraeters are selected depending on whether the user is
           typing
34     * with one or both tumbs, as retrieved from <code>res/
           values/user_modality.xml</code>.
35     * @param context the application context.
36     * @param userId the user identifier.
37     */
38    public SpeedQuantifier(Context context, String userId) {
39      lastPress = lastRelease = System.currentTimeMillis();
40      Resources res = context.getResources();
41      int[] usr_mod = res.getIntArray(R.array.urs_modality);
42      int usrId = Integer.parseInt(userId);
43      switch (usr_mod[usrId]) {
44      case 0:
45        lin_m = (float) 1.8706;
46        lin_q = (float) 44.4696;
47        break;
48      case 1:
49        lin_m = (float) 0.0;
50        lin_q = (float) 0.0;
51      }
52    }
53
54    /**
55     * Gets the estimated typing speed.
56     * The estimation is performed in terms of distance from
           the linear model
57     * of last observations, smoothed to 1. Value returned is
           between 0 and 1.
58     * It should be called before updating the model with the
           last touch, in
59     * order to get results dependent only on previous touches.
60     * @return the estimated typing speed.
61     */
62    public float getRelativeSpeed() {
63      float tMean, pMean;
64      try {
65        tMean = tSum / tValues.size();
66        pMean = pSum / pValues.size();
```

```
67      } catch (ArithmeticException e) {
68        //thrown at the first keystroke
69        return (float) 0;
70      }
71      float t_model = (float) ((lin_m * pMean + lin_q) * 1);
72      if (t_model - tMean < 0) {
73        return 0;
74      } else  if ( t_model - tMean > lin_maxDecr * t_model) {
75        return 1;
76      } else {
77        return (t_model - tMean) / (lin_maxDecr * t_model);
78      }
79    }
80
81    /**
82     * Updates the smoothed value of pressure times.
83     * The intra-stroke time is updated only when pressure was
          not
84     * greater than 500ms. Also lastRelease is updated with the
          current
85     * timestamp.
86     * @param pTimeStamp the timestamp at which the touch
          release occurred.
87     */
88    public void updatePressure(long pTimeStamp) {
89      lastPressLength = (int) (pTimeStamp - lastPress);
90      lastRelease = pTimeStamp;
91      if (lastPressLength < 500) {
92        if (pValues.size() == n_obs) {
93          pSum -= pValues.poll();
94        }
95        pSum += lastPressLength;
96        pValues.add(lastPressLength);
97      }
98    }
99
100   /**
101    * Updates the smoothed value of inter-stroke times.
102    * The inter-stroke time is updated only when pressure was
          not
103    * greater than 500ms. Also lastPressure is updated with
          the current
104    * timestamp.
105    * @param pTimeStamp the timestamp at which the touch is
          initiated.
106    */
107   public void updateTime(long tTimeStamp) {
108     lastReleaseLength = (int) (tTimeStamp - lastRelease);
109     lastPress = tTimeStamp;
```

```
110       if (lastReleaseLength < 500) {
111         if (tValues.size() == n_obs) {
112           tSum -= tValues.poll();
113         }
114         tSum += lastReleaseLength;
115         tValues.add(lastReleaseLength);
116       }
117     }
118
119     /**
120      * Updates the last release timestamp.
121      * This is called when the spacebar is pressed, because
           intra-words times
122      * do not count in the model in order to have more smoothed
           value, but
123      * is necessary to know when they happened to calculate the
           next inter-stroke
124      * time.
125      * @param tTimeStamp the timestamp at which the spacebar
           was released.
126      */
127     public void setLastTime(long tTimeStamp) {
128       lastRelease = tTimeStamp;
129     }
130
131     /**
132      * Returns the state of the quantifier in a format valid
           for the {@link #Logger Logger}.
133      * @return the log.
134      */
135     public String toLog() {
136       String log = lastPressLength + "," + lastReleaseLength;
137       return log;
138     }
139 }
```

D.8 Class WalkingQuantifier

```
1  package com.unz.adkey;
2
3  import java.util.Iterator;
4  import java.util.concurrent.ArrayBlockingQueue;
5
6  /**
7   * The model that quantifies the user walking speed.
8   * This stores readings of values from the device y-axis
        accelerometer
```

```
 9  * and uses them to compute their standard deviation as an
       estimate
10  * of the user walking speed.
11  * @author Michael Angeleri - michael.angeleri@mail.polimi.it
12  *
13  */
14 public class WalkingQuantifier {
15
16   private static final int qLen = 50;
17   private static final float maxVar = (float) 1.6;
18
19   private float gSum;
20   private ArrayBlockingQueue<Float> gValues = new
         ArrayBlockingQueue<Float>(qLen);
21
22   /**
23    * Class constructor.
24    */
25   public WalkingQuantifier() {
26   }
27
28   /**
29    * Updates the acceleration values
30    * Get the y-acceleration as measured by the Keyboard
         activity and
31    * updates the stored values (only y-axis considered)
32    * @param value the values acquired from the accelerometer
33    */
34   public void updateAccel(float value) {
35     if (gValues.size() == qLen) {
36       gSum -= gValues.poll();
37     }
38     gSum += value;
39     gValues.add(value);
40   }
41
42   /**
43    * Return the variance over the last measured acceleration
         values.
44    * The mean and standard deviation of the 50 last measured
         values are computed
45    * and the latter is normalized to one.
46    * @return the variance over acceleration normalized to one
         .
47    */
48   public float getGVariance() {
49     float mean = gSum / gValues.size();
50     float gSqrSum = 0;
```

```
51      for (Iterator<Float> i = gValues.iterator(); i.hasNext();
            ) {
52        gSqrSum += Math.pow(i.next() - mean, 2);
53      }
54      float var = gSqrSum / gValues.size();
55      //normalizing the value to 1
56      if (var > maxVar) {
57        var = maxVar;
58      }
59      return var / maxVar;
60    }
61  }
```