Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Kännsch - Adaptive Keyboard for iOS

Bachelor Thesis

Andres Konrad

konradan@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Laura Peer, Philipp Brandes
Prof. Dr. Roger Wattenhofer

July 27, 2016

Abstract

Texts that are written on smartphones are highly individual. The words and expressions typed in these mostly casual conversations differ from person to person.
In Switzerland, there is a high diversity of dialects and no fixed dictionary that could capture this vastness of words. At the time of writing this paper, there exists no virtual keyboard for iOS that ships a Swiss German dictionary as a basis for word completion or correction.
In this thesis we built an adaptive keyboard for the mobile operating system iOS. Basis dictionaries for English, German and Swiss German were constructed and added to the keyboard back-end. We implemented several mechanisms that enable the starting dictionary to expand and deliver accurate suggestions for word completion and correction. To measure the effectiveness of our algorithm, we conducted several statistical tests. Additionaly, we are gathering the inputs from the users and their locations in a form that they can be used for future dialect research.

Contents

Chapter 1

Introduction

> *"Language is a fascinating living thing"* - Ellie from Apple's App Review Board

Language is indeed fascinating and living, always evolving and allowing people to express themselves in a personal matter. When writing short text messages people often tend to deliberately ignore orthography in order to give the digital messages a personal touch or to approximate the phonetic transcription.
A virtual keyboard should consequently be as adaptive as possible, learning all of the user's behavior and reacting accordingly. Common, successful virtual keyboards usually provide the user with different keyboard layouts, three suggestions for word completion and may include a mechanism to overwrite a (mis-)typed word, called auto-correction. Although the built-in virtual keyboard in iOS devices features these functionalities, they can still be improved, especially when it comes to (Swiss) dialects with high variance in language usage.
In this thesis, our focus was to develop a user-friendly keyboard application that suggests up to three word completions or corrections based on the user's input. Our keyboard features the same functionalities as the built-in version and is designed to match its key layout.

1.1 Related Work

This bachelor thesis is a direct follow-up of the master's thesis by Laura Peer, who developed Kännsch for the mobile operating system Android [1]. Her work mainly focuses on data collection and dialect studies, whereas this thesis tries to center around the prediction algorithm and implementation, which was already provided to Laura Peer by Google's Open Source Keyboard LatinIME [2].
A year before this thesis was completed, a first attempt to create Kännsch for iOS has been made [3]. The topic of this thesis was published again to improve on the results.
Most of the server back-end was developed and evaluated by Marcel Bertsch in

his master's thesis [4]. His dictionary creation algorithm and server implementation is now used by both main mobile operating system versions of Kännsch. When accessing Apple's App Store [5], countless other keyboards appear ready to download, the most successful ones being maybe Swype [6] and SwiftKey [7]. Depending on which application you pick, it may feature different appearance themes, offer additional symbol inputs or allow you to input your word using swiping gestures.
The long paper on a survey for text prediction by Nestor Garay-Vitoria and Julio Abascal [8] provided an overview on different prediction approaches. The paper discussing the 'Effects of ngram order and training text size on word prediction' [9] describes the importance of bigrams in text prediction, which are a big part of our concept (see Section 2.2).
We compared a number of different custom keyboard templates for iOS out of the internet, one of them being a template from BHJ Studios [10]. It already provided a lot of useful functions and was written in the desired programming language Objective-C [11]. Our application is built on top of this template.

1.2 Outline

In Chapter 2 we explain the different concepts that have been developed during the course of this thesis. The specific implementation and features are explained in Chapter 3. In Chapter 4 we discuss the prediction results on statistical tests. Chapter 5 derives a conclusion to the thesis and gives an outlook for possible future work on this project.

Chapter 2

Concept

This chapter tries to give an overview of the most important concepts that were developed for the keyboard application. Sections 2.1 and 2.2 build up to the explanation of the prediction algorithm in Section 2.3 and the correction algorithm in Section 2.4. We explain what user information we retrieve in the section about logging (see 2.5).
These concepts are the cornerstones to the Implementation Chapter 3.

2.1 Rank

In the context of word prediction, the words 'rank' and 'frequency' are often used interchangeably. We will continue to use 'rank', because 'rank' implies a relative order using natural numbers, whereas 'frequency' does not.
The concept of ordering words based on its rank is one of the simplest [8]. When a new word is added to the dictionary, it receives an initial rank (in this case 1), which gets increased for every reoccuring usage of the exact same word. The word that is used the most will have the highest rank.

Table 2.1: Top 10 Swiss words ordered by descending rank

word	rank
ich	149
du	110
de	107
das	106
und	88
so	77
i	60
isch	53
am	51
no	46

In Table 2.1 we see an example of the concept of rank, where Swiss words are

ordered by descending rank. This data was retrieved from Marcel Bertsch's analysis of the words the Android users typed using Kännsch [4] and is used as a starting point for the dictionary implemented in our application.

Text prediction systems work the best when they predict the correct content with as little information as possible. Because the rank symbolises how frequent a word is used, the most probable correct word is the word with the highest rank, when there is no other information accessible.
Wrong words will be learned in this described schema just like any other words. The data structure has to decrease in size in order to delete words that are not used as much as others and making the scanning of the whole data structure faster. We can safely assume that mistyped words will not be typed and stored as often as the correct words.
A concept widely used in different parts of computer science is called Additive Increase Multiplicative Decrease, in short AIMD. As its name suggests, it increases the rank of a content additive, in our case by 1. When reaching a predefined limit, it decreases the rank by a factor of 2. AIMD ensures that the rank we associate by the content is always distributed in a certain bound and relative order.
The first time a user types in her name for example, the dictionary probably has to store it anew with rank 1. Typing it again a second time will update the rank of the name to 2. The rank will be increased for every occurence of the name, up until the rank of the name reaches the predefined limit of 255, where the name and all the other words in the dictionary get divided by 2. After this multiplicative decrease phase, the name has the rank 127.
When a word with rank 1 undergoes a multiplicative decrease phase, it will be deleted from the database. By applying this concept of AIMD, we ensure that unused data will eventually be deleted from the dictionary.
When a new word is entered, we have to tell the database to increase the rank. If it is not yet in the database, a new entry has to be created. We also provide a functionality to decrease the rank when the user deletes a word that had been typed previously.

2.2 Bigrams

Bigrams are ordered word pairs, forming a set. They are a special case of ngrams, where ngrams are a set of word groups with n ordered words, $n \in \mathbb{N}$. The emphasis in this definition lies in the word 'ordered': the bigram "is it" is not the same as "it is". In Section 2.1 we talked about storing words, but since words are in fact unigrams, we will refer to them as unigrams from now on.
The concept of bigrams are commonly used in text prediction algorithms. Their construct allows to grasp the grammatical constructs in a simple and effective

manner. When a user types a new word, the database containing the bigrams can be scanned for the previous word and the first letter(s) of the current word, called the prefix. This result can be ordered again by rank to retrieve the most probable words, given the current context.
Bigrams are stored the same way like unigrams: with a corresponding rank and AIMD to regulate the rank.

Table 2.2: Top 15 Swiss bigrams ordered by descending rank

first word	second word	rank
das	isch	111
ich	bin	103
und	ich	77
mit	em	70
i	ha	69
bi	dir	66
ich	han	65
i	de	64
hast	du	64
uf	de	61
in	der	59
au	no	57
am	i	54
ich	ha	54
mit	de	53

Similar to the unigram table in the previous section (see 2.1), Table 2.2 visualises the concept of bigrams. As before, the data was retrieved from Marcel Bertsch's study [4] and is used in the implementation of the app.

2.3 Prediction

The task of the prediction algorithm is to find the three most probable words given its current context. The context contains the previous word (if there is one) and the prefix.
As described in the previous sections 2.1 and 2.2, unigrams and bigrams can be used to find the most probable word by scanning for matching contexts and analysing its rank. Since the number of stored bigrams is limited but their occurance has a high variance, the relevance of a bigram is much higher than a single word if we indeed find a bigram matching a given context. That is the reason why we chose to prioritise every found bigram over a unigram.
In other words: the algorithm first scans the bigram table and tries to match the previous word and the prefix. Those who are found will be ordered by decreasing

rank and the second words will appear as suggestions. If there are less than three suggestions found, the table with the unigrams will be scanned for a matching prefix and ordered and displayed accordingly.
Suppose a user wants to write “Hey, what’s up?” and has already typed “Hey, w”. To produce the three suggestions, the algorithm scans the bigrams that match the first word completely and have a second word that starts with a “w”. Because this particular user often types this phrase, two bigrams “hey what” and “hey what’s” are found after scanning. “what” and “what’s” get suggested to the user. To fill the third suggestion with a meaningful word, the unigrams that start with a “w” get scanned . The unigram “who” gets picked, because it has the highest rank of all the unigrams. The user is now able to pick the correct suggestion between “what”, “what’s” and “who”.
It may occur that after running this described algorithm, less than three words are found for a given context. The current word may have been mistyped, so that is when the correction algorithm sets in. This algorithm is explained in Section 2.4.
To improve the performance of this algorithm, the results of the last prediction algorithm are still accessible in the new run. If the results from the last run still satisfy the new criteria, for example a longer prefix, they will be used again.

Keeping these tables as compact and informative as possible is of utmost importance. Redundant data has to be avoided. By enabling case insensitivity, we are making sure that a bigram “It is” collapses to “it is” and supports the rank of the latter. Usually, the words and bigrams get detected by parsing everything in between two spaces. Punctuation characters have to be deleted before inserting every parsed unigram or bigram in the database. The sentence “Hey, what’s up?” produces the unigrams “hey”, “what’s”, “up” and bigrams “hey what’s”, “what’s up”.

2.4 Correction

The goal of the correction algorithm is to find correct words from slightly misspelled ones. In our particular environment, the usual typing error occurs when a user does not hit the correct button, but a button lying close to the intended. That means we are mainly interested in how many substitutions we would have to perform to reach a meaningful word and if these substitutions of typed keys have occurred in close proximity. Unintentional deletion and insertion have to be considered as well. Furthermore, we want to prioritise words with high ranks, because they have a higher probability to be mistyped, due to their higher occurance.
We decided to use an adjusted form of the Levenshtein distance [12] to solve this task. Given a word, the Levenshtein distance is calculated for every word in the dictionary. The Levenshtein distance is the sum of predefined insertion, deletion

and substitution costs that have to be paid according to the number of minimal transformations that have to be applied to the first word to get the second one. The runtime of calculating the Levenshtein distance for a word in the dictionary would be $O(nm)$, where n and m are the lengths of the two words. This runtime would still have to be multiplied by the total number of words in the dictionary, which would eventually be too slow.
We are only going to calculate the Levenshtein distance on a subset of all the words in the dictionary. How we choose this subset is explained in Section 3.2.2. The subsitution cost is adapted based on the proximity of the keys on the current keyboard layout. Adjacent keys have a substitution cost of a third of the default substitution cost.

2.5 Logging

For every life cycle of the keyboard, a log file gets created, that holds the most important information about the application's state. The log files include information about:

- unique user ID
- iOS version number
- application build number
- every typed key
- the layout of every keyboard button
- xy-coordinates of touches on keyboard button
- statistics of seconds spent, characters typed and saved and the total number of predictions used
- memory warnings
- suggestion array with last word, prefix and elapsed time for prediction function call
- location of the device (if enabled by user)

Every statement above gets associated with a timestamp. Log files are not kept longer than a week on the device.
iOS automatically exchanges our keyboard with the built-in keyboard for secure text input fields. This ensures that our logs do not hold passwords.
The unique user ID is generated once with a randomly seeded hash function and is stored on the device for later uploads.
Crash reports are sent automatically by Apple and can be inspected in Xcode.

Chapter 3

Implementation

As an amuse-bouche and introduction to this rather technical chapter, we show in Figure 3.1 the final user interface of the keyboard that suggests three words.

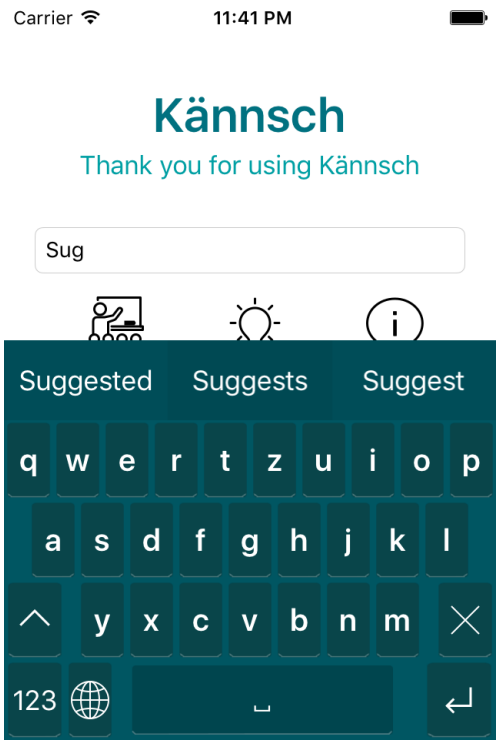

Figure 3.1: Keyboard suggesting three different suggestions in the test environment of the companion application.

3.1 iOS Implementation Environment

When writing an iOS application, there is not much choosing for the tools to work with. An iOS application is written on an Apple Computer, using the coding environment Xcode [13]. A programmer can choose between two different programming languages, Objective-C [11] and Swift [14]. We chose not to work with Swift, because it is fairly new and hence may not be as mature as Objective-C.
Since iOS 8 was introduced in 2014, it is possible to design and integrate keyboard extensions for Apple's iPhones [15]. A keyboard extension replaces the standard keyboard on an iOS device for every text input on the device, except for secure text input. Xcode offers a variety of tools that aided this project, such as checks for memory leakage, interface builder or test environments. By using this variety of functionalities, it was easy to design the user interfaces and logic for the keyboard and the companion application. Designing these lightweight iOS extensions does not come without restrictions. We noticed that a keyboard should not use more than 30MB RAM, otherwise the keyboard would trigger a memory warning and crash eventually.

3.2 Dictionary Structure

The general schema for creating the data structures elaborated in Chapter 2 is already given by the concept they are describing. We need scannable tables of unigrams and bigrams with their rank and have to be able to perform distance calculations. The tables have to be able to insert, delete and modify unigram and bigrams and allow prefix scanning with fast performance.
Apple offers multiple storage options for large data structures [16]. We chose to use SQLite, because it reduces the memory usage significantly and has other benefits elaborated in Section 3.2.1.

3.2.1 SQLite

SQLite is a library implementing an SQL database engine [17]. It allows to outsource the logic of storing and retrieving data to the sophisticated implementation of an SQL database.
The library offers a table extension called FTS, which stands for Fast Text Search [18]. As the name implies, the tables created using FTS are able to significantly increase the performance of prefix scanning through the whole table. The only cost of using the FTS is a slightly larger database, due to new shadow tables that are needed for the fast text search algorithm. All of our tables were constructed using FTS (Version 4) to benefit from this performance gain.

3.2.2 Spellfix

Spellfix is another extension of SQLite [19]. Together with FTS4, it is able to scan for words that are potentially misspelled.
By enabling spellfix in SQLite, one is capable of querying the data structure as described in Section 2.4. Every word gets mapped to a large subset of all words using a hash-function. This hash-function implemented by spellfix maps characters to phonetic groups and is called phone-hash. Although it would have been better to map them to a character group based solely on close proximity of the dynamic keyboard layout, this simple hash-function already showed satisfying results. Minor generalizations were needed to ensure that the phone-hash implementation works well for any language.
Only on this phonetic subset, the Levenshtein distance gets calculated.
The proximity of the keyboard layout keys does however play a significant role for the substitution costs. Close buttons on the view will have corresponding character substitution costs of a third of the default substitution cost.
In Figure 3.2 we see that the final application is able to correct slightly misspelled words.

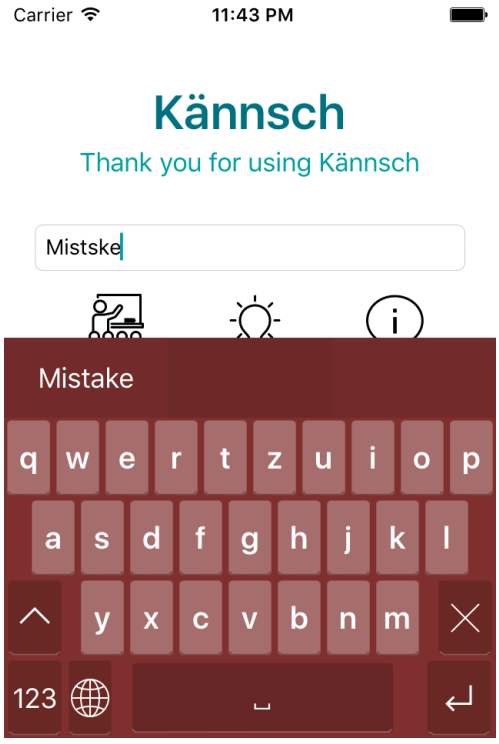

Figure 3.2: Keyboard suggesting a correction for a mistake.

Spellfix is usually loaded dynamically. Apple does not allow to load any code

on its iOS devices dynamically. To circumvent these restrictions, we had to insert the code statically and rewrite an entry function.

3.3 Server Interaction

Since the server back-end was already created as part of the corresponding Android project [4], all we had to do is communicate with our server in the same way as an Android device would have done. Provided the user grants the application full access, the log files mentioned in Section 2.5 are uploaded to our server on a regular basis. After having received a large enough amount of words from a particular user, our server analyses all the texts the user has typed. A specialised algorithm running on the server adds the user to a group of users that type similarly. For this and every other group, the algorithm creates a dictionary of combined unigrams and bigrams with their complementary ranks. When the keyboard establishes its next connection to the server, the keyboard receives this personalised dictionary and updates the local database.

Chapter 4

Results

4.1 Analysis of the Prediction Algorithm

To ensure the correctness and effectiveness of our prediction algorithm, several tests were conducted on two different data sets.
During all of these tests, the server interaction was disabled and no additional dictionary was received that could influence the prediction outcome in either way. This decision was made because it would introduce a non-deterministic factor and the prediction should adapt as well when the user does not want to send or receive data to or from the server.
We are mainly interested in the ratio of the characters saved to the total number of characters, in short CSR. This value is obtained by testing every word of the data set as if a diligent user types them in with our keyboard and presses on the suggestion whenever the correct suggestion appears. If the user wants to write 'Kännsch' and the suggestion appears after the second keystroke, she saved 5 characters with a CSR of $\frac{5}{7} \approx 71.4\%$. The CSR is very meaningful because it represents how early the prediction algorithm can predict what the user wants to type.
The company behind the successful prediction keyboard SwiftKey state in their blog that their users save 33% of their characters [20].

The first data set contains the English Harry Potter books from J.K.Rowling. These books were also used in the previous publication of this thesis [3], because they are easily accessible, most probably have no typing errors and form a special dictionary due to the high number of invented phrases.
In Figure 4.1 we see a first promising glimpse of the effectiveness of the prediction algorithm. We see that there is a big difference of the CSR if the prediction algorithm is allowed to adapt or not. As described in Chapter 2, adaption means learning and deleting unigrams and bigrams with AIMD.
When the prediction algorithm is prohibited to adapt and only uses the provided dictionary to come up with three suggestions, it shows moderate results with a CSR fluctuating close to 36.8%,
In our application the algorithm is allowed to adapt. By enabling adaption

around half of all the characters can be saved over the course of typing all the Harry Potter books.

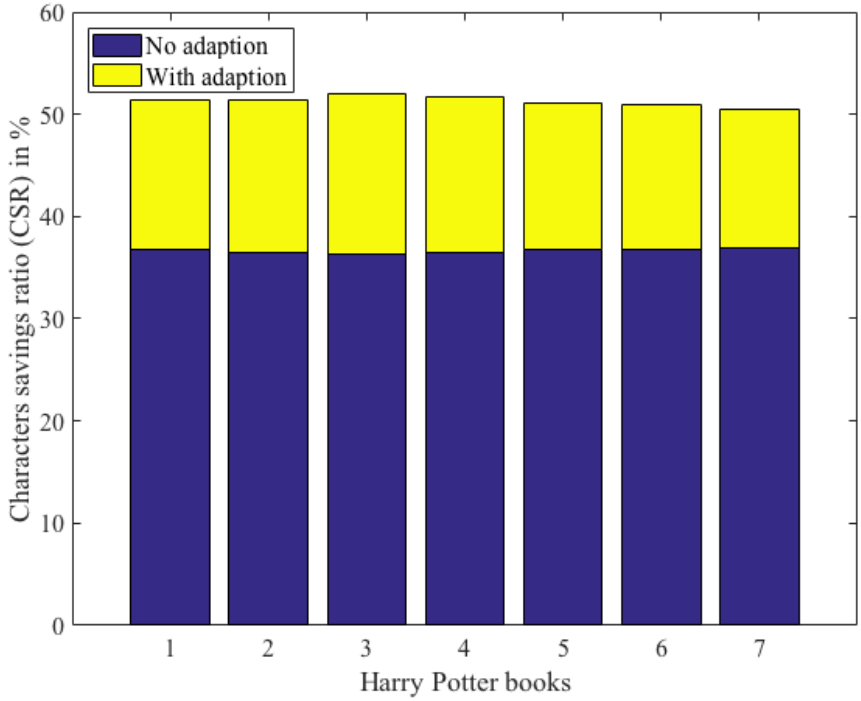

Figure 4.1: Test result of measuring CSR after every Harry Potter book. In blue, we see how the prediction would have performed using only the basis English dictionary. In yellow, we see the increase of the CSR for each book when the algorithm is allowed to adapt.

By looking at Figure 4.1 it seems like the algorithm adapted the dictionary almost completely in the first Harry Potter book. This leads us to our next test shown in Figure 4.2, where we are inspecting the adapting process during the first Harry Potter book.

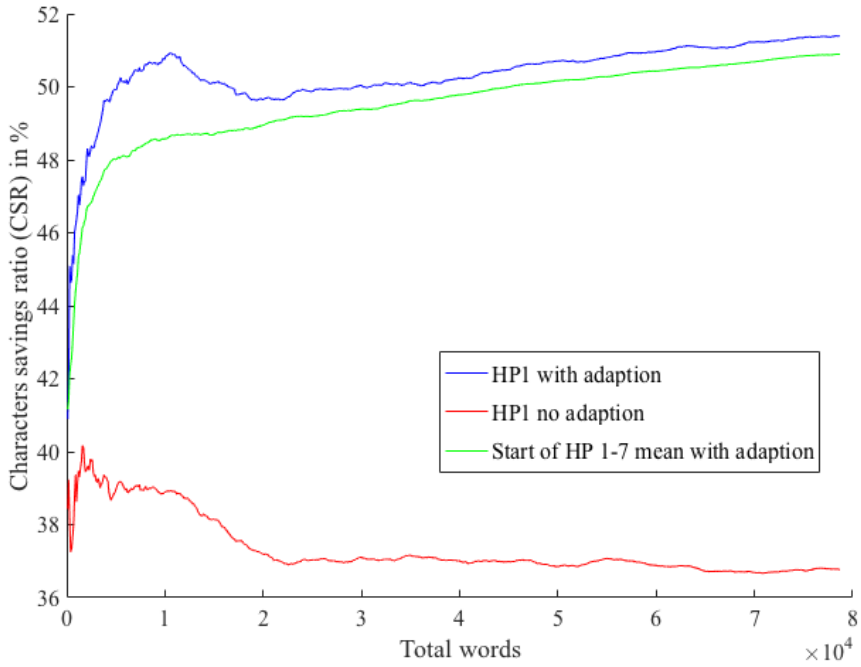

Figure 4.2: CSR evolution (on the left) while typing the first Harry Potter book with and without adaption (blue and the latter red) and the mean of the start of all Harry Potter books 1 through 7 (green).

At the start of the test, we are seeing a CSR of less than 41%. The blue line increases heavily during the first 5.000 words, which means the dictionary adapts very quickly to the writing style of J.K.Rowling. After around 13.000 words the word usage of the book changes, because both of the CSR with and without adaption decrease significantly, whereas the mean over the start of all books shows no such behaviour. The decrease is followed by a steady increase phase to reach the final 51.4%.

The second data set contains Swiss German texts written by 100 users using the Kännsch Android version. This data set is very representative, because the iOS users are expected to have similar language patterns as the Android users. In Figure 4.3 we can clearly see that the dictionary does indeed adapt itself for every one of these 100 users.

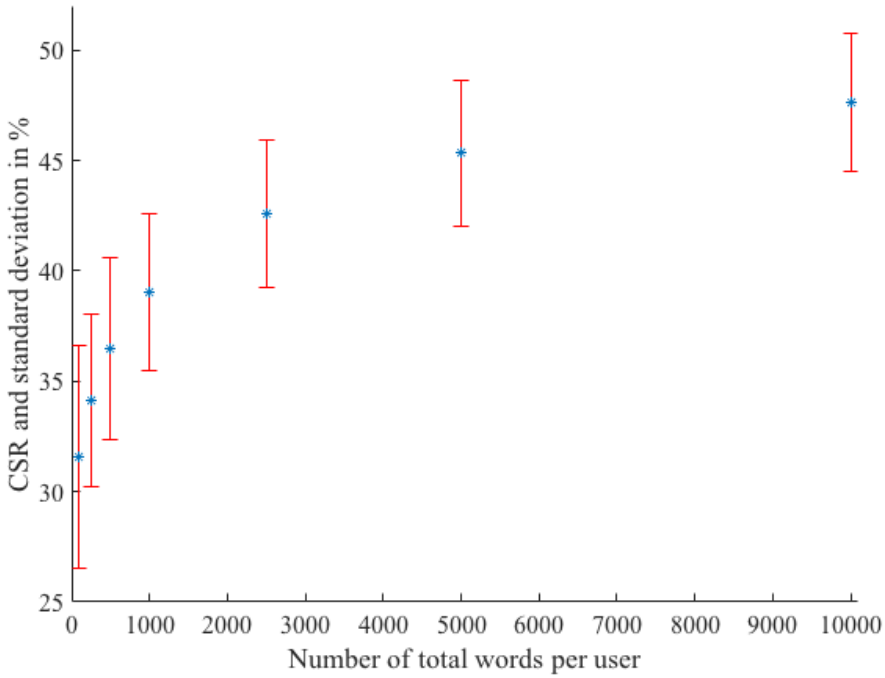

Figure 4.3: Test result of measuring CSR values of 100 users texts with increasing total number of words per user. In blue we see the trend of CSR depending on the number of words each user was allowed to type. In red, the standard deviation of these 100 users is visualised for each measuring point.

In Figure 4.3 we can see that the average Swiss user is able to save around 31% of the characters when the keyboard has not yet learnt anything. The standard deviation shrinks from around 5% to 3% with increasing word counts, which means that there might still be users who will have a significantly lower or higher CSR while using our keyboard in the beginning, but slowly converge to the CSR of the average user. The average Swiss user profits from the adapting algorithm: After writing her first 10.000 words, the average CSR is 47.7% with an ascending trend.

Chapter 5

Conclusion and Outlook

We conclude that over all our product performs well. The application is working as expected and got accepted by Apple to be published in the App Store. The application was downloaded by a few hundreds of users and received sparse but mostly postive reviews.
Regarding the analysis in Section 4.1, we can say that our promised 'adaptive keyboard' is indeed adaptive and able to learn. Especially the steep learning rate the prediction algorithm produces in the beginning may hopefully encourage the users to stick with our application.
To improve the current application, a broader language support could be added. Right now, only English, German and Swiss German are supported languages, the same holds for the keyboard layouts. Our target audience would enlarge by localizing the strings, layouts and basis dictionaries for languages like e.g. Spanish or French.
Timo Bräm worked on the addition of swiping gestures as an alternative input method for Kännsch [21].
Since we have published the application in the App Store, we will continue to receive log files and crash reports from a wide range of users that will ensure a constant quality of the software when these reports are analysed and problems fixed accordingly. These log files can also be used for future work on dialect studies.
Half a year ago, SwiftKey released the world's first virtual keyboard that works with neural networks to overpower the effectiveness of ngram prediction algorithms [22]. It would be an interesting challenge to implement the prediction system for Kännsch using neural networks in a future project.

Bibliography

[1] Peer, L.: Kännsch - a swiss german keyboard for android. Master's thesis, ETH Zurich (September 2014)

[2] Google: Latinime git. https://android.googlesource.com/platform/packages/inputmethods/LatinIME Accessed on July 27, 2016.

[3] Hüsser, M.: Kännsch - swiss german keyboard for ios. Bachelor's thesis, ETH Zurich (September 2015)

[4] Bertsch, M.: Kännsch - improving swiss german keyboard. Master's thesis, ETH Zurich (August 2015)

[5] Apple: App store website. https://itunes.apple.com/ch/genre/ios/ Accessed on July 27, 2016.

[6] Nuance: Swype website. http://www.swype.com Accessed on July 27, 2016.

[7] SwiftKey: Swiftkey website. https://swiftkey.com/ Accessed on July 27, 2016.

[8] Nestor Garay-Vitoria, J.A.: Text prediction systems: a survey. Long Paper (2006)

[9] Gregory W. Lesher, Bryan J. Moulton, J.H.: Effects of ngram order and training text size on word prediction (1992)

[10] BJH-Studios: ios custom keyboard template. https://github.com/bjhstudios/iOSCustomKeyboard Accessed on July 27, 2016.

[11] Apple: About objective-c. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html Accessed on July 27, 2016.

[12] Levenshtein, V.: Binary codes capable of correcting deletions, insertions, and reversals (1965)

[13] Apple: Xcode. https://developer.apple.com/xcode/ Accessed on July 27, 2016.

[14] Apple: Swift. https://developer.apple.com/swift/ Accessed on July 27, 2016.

[15] Apple: Custom keyboard. https://developer.apple.com/library/ios/documentation/General/Conceptual/ExtensibilityPG/CustomKeyboard.html Accessed on July 27, 2016.

[16] Apple: Persistent store types and behaviors. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/PersistentStoreFeatures.html Accessed on July 27, 2016.

[17] SQLite: About sqlite. http://www.sqlite.org/about.html Accessed on July 27, 2016.

[18] SQLite: Sqlite fts3 and fts4 extensions. http://www.sqlite.org/fts3.html Accessed on July 27, 2016.

[19] Sqlite: Spellfix documentation. http://www.sqlite.org/spellfix1.html Accessed on July 27, 2016.

[20] SwiftKey: 10 things every swiftkey user should know! https://blog.swiftkey.com/10-things-every-swiftkey-user-should-know/ Accessed on July 27, 2016.

[21] Bräm, T.: ios swipe. Bachelor's thesis, ETH Zurich (2016)

[22] SwiftKey: Introducing the worlds first neural network keyboard. https://blog.swiftkey.com/neural-networks-a-meaningful-leap-for-mobile-typing/ Accessed on July 27, 2016.