Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

# Kännsch - Swiss German Keyboard for iOS

Bachelor Thesis


Melanie Hüsser

huesseme@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich


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

September 3, 2015

# Abstract

Many software keyboards for mobile devices these days offer various support to assist users when writing text. Algorithms for prediction and correction of typed input have been developed and tuned in speed and accuracy. Sadly, those algorithms only work for languages with clear orthography and a defined dictionary. In this thesis, we developed a keyboard application for Apple iPhone users that supports Swiss German. Similar to the already existing Android application Kännsch it is designed to work with various dialects and accesses the same resources. The application reaches reasonable performance results that can compete with similar applications.

# Contents

**Abstract** i

**1 Introduction** 1

1.1 Related Work . . . 2

1.2 Outline . . . 2

**2 Design** 3

2.1 Prediction . . . 3

2.1.1 Word Completion . . . 3

2.1.2 Word Prediction . . . 4

2.2 Correction . . . 4

2.2.1 Auto-Correction . . . 4

2.2.2 Levenshtein Algorithm . . . 5

2.2.3 Adjusted Levenshtein Algorithm . . . 5

2.3 Frequencies . . . 6

2.4 Trade off between Completion and Correction . . . 6

2.5 Format of Dictionary . . . 8

2.5.1 Core Dictionary . . . 8

2.5.2 Personal Dictionary . . . 9

2.5.3 Top list . . . 10

**3 Implementation** 11

3.1 Swift Programming Language . . . 11

3.2 Framework and Libraries . . . 11

3.3 Retrieval of Local Dictionaries . . . 11

3.3.1 Research Logger . . . 12

3.3.2 Kännsch Server . . . 12

**4 Results** **13**

4.1 Typing Error Analysis . . . . . 13

4.2 Prediction Benchmark . . . . . 14

4.3 Correction Benchmark . . . . . 16

**5 Conclusion and Outlook** **19**

**Bibliography** **20**

Chapter 1

# Introduction

Swiss German is not a full language by regular standards. For example, there is only one form of past tense. It is only used for talking but there is no clear orthography and no complete dictionary. In fact, every region developed its very own version of Swiss German. These dialects are very diverse such that they can be differentiated based on the choice of words only. Official documents and letters are commonly written in standard German.

Nowadays, with the huge popularity of mobile devices, most communication over distance takes place by SMS and chat applications. These ways of communication feel much more direct than letters and e-mails and more similar to talking face-to-face so people started to write those in Swiss German. Typing on a mobile device can be quite cumbersome and typing errors occur frequently. Software keyboards are designed to overcome this problem. Prediction algorithms reduce the amount of keystrokes which is needed to write a word while auto-correction counteracts the mistakes made by the user. But these algorithms highly depend on a well-defined and complete dictionary. Such a dictionary does not exist for Swiss German. Therefore Swiss German natives cannot benefit from these advantages because those algorithms need a collection of words to operate on.

A first attempt to improve the situation was made with Kännsch [1], an application for Android [2]. It is a software keyboard design for individual language. The application comes with a preinstalled list of common Swiss German words. Kännsch got further improved [3]. The data that was collected by the application got analyzed and was used to create region specific dictionaries. These dictionaries are then sent back to each user based on their writing style.

In this thesis, we will develop a similar application for Apple iPhone [4] users. Since the original keyboard [5] is closed source and inaccessible, a new keyboard must be designed from scratch. The main parts of the thesis are the design and implementation of prediction and correction algorithms and corresponding data structures. The application is tuned to adapt to the user's writing style. Furthermore, it will send the same data as the Android counterpart to benefit from the local dictionaries as well.

## 1.1 Related Work

This thesis will transfer the work of *Kännsch - a Swiss German Keyboard for Android* [1] to iOS [6], the operating system for Apple iPhone [4]. Kännsch is a software keyboard designed to improve writing in Swiss German. The application is based on Google's *Android Open Source Project LatinIME* [7]. LatinIME contains a large amount of standard dictionaries in different languages which are referred to as *Google dictionaries* in this thesis. The German dictionary is used as a basis for our word collection.

*Kännsch - Improving Swiss German Keyboard* [3] added more features to Kännsch. Logs created by users of the application are used to create region based dictionaries. Our application will use the same web server to fill the word collection with words adjusted to the user's dialect.

There already exist a couple of different keyboards for iOS. One of our goals is to give the user the same experience that he is accustomed to while using Apple's standard keyboard *QuickType* [5]. Other notable keyboards are *Swype* [8] and *SwiftKey* [9]. Both keyboards are capable of filling the dictionary with words based on the user's SMS history as well as learning new words. In addition, they offer the possibility to use swipe gestures as alternative input: Instead of typing every key individually, the user may sweep over the intended keys in one stroke. *Tasty Imitation* [10] is an plain open source keyboard. It offers nothing but a close resemblance to the standard keyboard. We will use it as a template for our project.

The *Levenshtein distance* [11] is an algorithm that compares the similarity between two words. In Section 2.2.2 we will use it to calculate the likelihood of dictionary words if the user misspelled the input. An open source implementation [12] of the algorithm will serve as a basis for us.

*ZipArchive* [13] is a library to compress and decompress files on iOS. We will use it in Section 3.3.1 to compress the log files before sending them to the web server. An implementation for JSON in Java [14] is used to parse the dictionary and log files for testing.

## 1.2 Outline

In Chapter 2, we discuss the features that a keyboard application should offer and which algorithms and data structures were chosen to accomplish them. Chapter 3 gives an overview over implementation details and used tools. In Chapter 4, we set up test benchmarks to measure our algorithms in accuracy performance and calculation speed. Chapter 5 summarizes the state of the application and gives an outlook on future improvements.

Chapter 2

# Design

This chapter discusses every feature a keyboard should offer and which algorithm was chosen to fulfill a particular purpose. The parameters used in the implementation are determined with test benchmarks to tune the performance as shown in Chapter 4. We can assume that we already have a dictionary and know the frequency of every word.

## 2.1 Prediction

Word prediction is one of the two major features a mobile keyboard should offer. Its purpose is to reduce the number of keystrokes a user must type. We differentiate between two sub areas: Word prediction is the process of suggesting words that are often used together based on the current context while word completion only tries to figure out how the word that the user started to write will end.

### 2.1.1 Word Completion

Word completion is the process of suggesting the intended ending of a user input. Every word in the dictionary is sorted by its frequency and the most probable ones will be suggested by the application. The algorithm is designed to be case and umlaut insensitive since most people do not want to take the detour. Therefore the input and the dictionary word are adjusted into a comparable version by omitting umlauts and change all to capital letters. As an example, consider a dictionary that contains the words "ich", "hallo" and "Haus" with decreasing frequency. If the user types the letter "h", word completion will suggest the word "hallo" even though the word "ich" has higher frequency because the prefix does match the input.

### 2.1.2 Word Prediction

Word prediction is all about suggesting words based on the current context rather than the beginning of the input. A simple approach is the use of bigrams. Bigrams are pairs of words which often occur together. We know how often each pair occurs inside a typical text. That way the suggestions can be sorted by probabilities. For instance, the dictionary may contain the word pair ("ich", "bin"). If the last written word was "ich", word prediction will suggest the word "bin". If the user continues with the letter "h", word prediction will withdraw its suggestion and word completion takes over.

Additionally, the bigrams can be used to improve the word completion. This means that whenever the first word of the bigram is written the second one will be shown as a suggestion. As soon as the user begins to type the beginning of a word which matches the second one, it will be prioritized over other possible words in the dictionary.

## 2.2 Correction

Word correction is the other major part of a keyboard application. Since mobile keyboards are quite small, errors are common. Our system needs to distinguish errors from intended input and correct the detected mistakes.

### 2.2.1 Auto-Correction

Auto-correction is the process of eliminating typing errors made by the user on-the-fly. If the user writes a word and hits space, the typed word is exchanged with a corrected version. The user can decline a correction by selecting another suggestion or revert an applied correction by hitting backspace.

For every word in the dictionary, the algorithm uses an adjusted version of the Levenshtein distance as explained below to count the amount of mistakes that were made if the user intended to write that specific word. If a word in the dictionary is very similar to the input, it is very likely that the user wanted to write this word but made an typing error. We can neither believe that our dictionary is complete nor that our algorithm is perfect. This is why the auto-correction is only applied if the distance between both words does not surpass a certain limit.

For example, consider a dictionary with the words "ich", "habe" and "hallo". If the input of the user is "hllo", it will be corrected to "hallo" because only one error would have been made (the user forgot the letter "a"). Three errors are needed to reach the word "habe" (the user mistyped "llo" for "abe") and four for "ich" (the user mistyped three letters and added one).

### 2.2.2 Levenshtein Algorithm

The Levenshtein distance calculates the similarity of two words. The algorithm alters one word into the other by applying different modification steps (like inserting, deleting, and modifying letters). For example, to transform the word "Stuhl" into "Stil", we need two modification steps. The letter "u" is changed into an "i" and the letter "h" is omitted. These modification steps can be considered as the amount of mistakes that are made if one word is the input and the other is the actual intention. The algorithm uses dynamic programming to run in $O(nm)$ time where $n$ and $m$ are the lengths of the two words.

The algorithm in its original form is not suitable for our purpose because of its runtime. The correction algorithm will run after every keystroke to improve the result. Running a full Levenshtein algorithm on every word in question will take too long and keeping the intermediate results (the two dimensional matrices used for dynamic programming) is too memory expensive. Here are some properties we want to have:

- $O(nm)$ is too costly. We need a near linear approximation since we need to check a lot of words in our dictionary every time the user types.
- The algorithm should be able to reuse the last result since we will run it after every key stroke.
- We are interested in more diverse modification steps. For example, it is quite reasonable to distinguish whether the mistyped key was adjacent to the intended one or not.
- Some mistakes are more likely than others. The algorithm should use this to improve the result.
- We do not actually need the correct Levenshtein distance. A good approximation is sufficient.

### 2.2.3 Adjusted Levenshtein Algorithm

The algorithm we use is considering one input letter after the other. It will run on every word in the dictionary that is considered to be the correct word. A pointer to the corresponding letter position in the dictionary word is kept. Normally, this pointer gets increased after each operation. Both letters, the one from the input and the one at the current pointer position at the dictionary word, are compared according to the following criteria in the given order:

1. Is the input letter exactly the same as the dictionary letter? If so, add a copy operation (distance untouched).

2. Is the input key adjacent to the dictionary key on the keyboard? If so, add a near operation (distance raised by 1).

3. Is the input letter the same as the previous dictionary letter? If so, decrease the dictionary pointer, revert the last operation and add an insert (distance raised by 2).

4. Is the input letter the same as the next reference letter? If so, increase the reference pointer by an additional step and add a delete operation (distance raised by 7).

5. If nothing fits, add a modify operation (distance raised by 2).

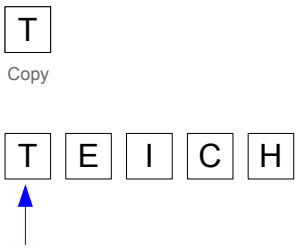
Figure 2.1 illustrates an example calculation of the algorithm. Each operation is weighted according to the probabilites of occurrences of this particular error in the results of the typing error analysis in Section 4.1. The current distance and reference pointer as well as the last action and input word is temporarily saved for every word that is computed. The last action is needed for reverting in case of an insert and the last input word is needed to ensure that the intermediate result is still fresh and suitable.

## 2.3 Frequencies

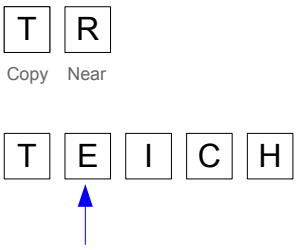
Using frequency numbers is a crucial part of any prediction system. Naturally, some words and word compositions occur more often than others. Every word in the dictionary gets associated with a number within a given limit that should represent its average appearance within a text. Since we have highly individual and probably incomplete dictionaries, we need a system that adapts its frequencies dynamically to the user. Increasing the frequency of a word is easy. Its number is simply raised each time it is written. More difficult is decreasing the frequency of all other words without needing too much computation time. Without decrease, words will eventually end up with very similar numbers. We will divide the frequency of all words by two as soon as one word reaches the upper limit. This way, all frequencies will still remain within the given limit. Additionally, the numbers will keep up the diversity because the system acts similar to AIMD (additive increase/multiplicative decrease) used in TCP to distribute data streams of different throughput.

## 2.4 Trade off between Completion and Correction

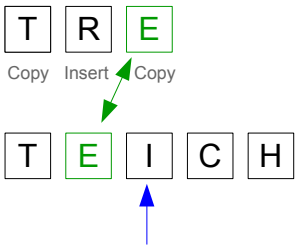
The keyboard application is trying to find out the most probable words the user was intending to write. Word prediction and correction are two different ways



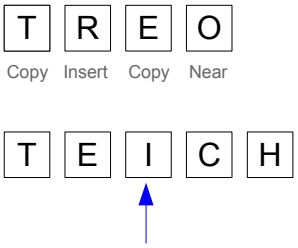

(a) At first, the pointer is at the beginning of the word. The first input is a "T" which matches the dictionary word. Therefore, a copy operation is used and the pointer moves one letter to the right.




(b) "R" does not match "E". But the two keys are adjacent on the keyboard. Therefore a near operation is used and the pointer moves one letter to the right.




(c) The input letter and the letter at the pointer are neither the same nor adjacent keys. But the algorithm detects that the input letter matches one letter previous to the pointer. It assumes that the "R" was inserted by accident. The previous operation gets reverted and the pointer stays at the same point.




(d) Because of the insert operation, the input letter and the dictionary letter are shifted by one. The pointer keeps track of these shifts. In the next step, "O" and "I" are compared which are neither the same nor adjacent. In addition

Figure 2.1: Example of four possible calculation steps of the adjusted Levenshtein algorithm. The upper letters ("Treo") resembles the input word and the lower letters represent the dictionary word ("Teich" in this case). The blue arrow indicates the current pointer position at the dictionary word. In each step, the last input letter is compared with the letter at the pointer.

to do that and both are needed for the best result. The problem is finding a suitable combination of the two approaches leading to one single result. Both algorithms will calculate a list of words. First, we observe that the word completion algorithm usually only finds a few results because it searches for exact matches. But showing the suggestions of the word completion first and then append the results of word correction is no option since this scales badly with dictionary size, short inputs and high error rate. Instead, we use a heuristic to compare the probability of correct words with mistyped words. We alter the frequency of misspelled word based on the correction algorithm. Then the altered frequency is used to perform the prediction algorithm together with all correct words. We can use the calculated Levenshtein distance and multiply it with a penalty factor. The result is subtracted from the frequency of the word.

For example assume that the word "habe" is much more frequent than "heben". When user types "hebe", the prediction algorithm will suggest "heben" since both share the same prefix. But the Levenshtein distance to "habe" is only 1. It is likely that the frequency number of "habe" after the reduction is still higher than the frequency number of "heben". Therefore the word will be corrected to "habe".

## 2.5 Format of Dictionary

The dictionary should have a practical data structure, which supports all operations that are mentioned above in reasonable time.

### 2.5.1 Core Dictionary

The core dictionary holds all words from the original Google dictionary. It contains the most frequent words of the German language. The words match the rules of the German orthography and already have frequency numbers assigned. The words are in a separated dictionary because we want these words merely as a basis. If a suitable word in Swiss German is found, it has precedence.

The words are stored in an asymmetric tree. Every node has 27 children, one for every letter in the latin alphabet and one for numbers and special characters. If a child contains more than 200 words, it gets split up into another 27 nodes up to a maximum tree depth of 3. Each leaf of the tree is a linear list containing words, which share a certain prefix. The implementation for prediction and correction traverses the tree based on the prefix and applies its algorithm on the list in the found leaf.

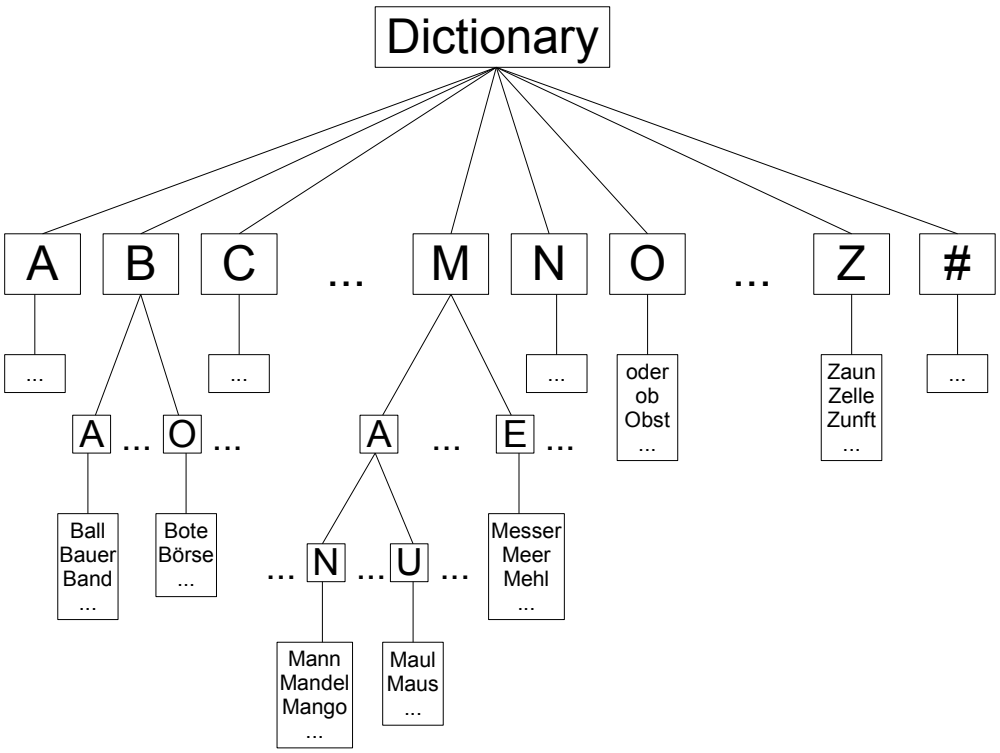



Figure 2.2: Schematic illustration of the asymmetric dictionary tree. Depending of the number of stored words each node either splits up into 27 additional nodes or points to a list of words. Words in the same list share the same prefix.

### 2.5.2 Personal Dictionary

The personal dictionary has the exact same structure as the core dictionary but it contains words that are added by the Kännsch server. Additional words are added whenever a user types a word that is not yet listed in either dictionary. Those words are first copied into a temporary buffer and then inserted into the personal dictionary as soon as they are typed a second time. Words that were not used for a long time are forgotten whenever the frequency of all words are decreased since this dictionary contains seldom used words and typing errors.

### 2.5.3 Top list

The tree structure creates an additional problem: Since only words in the corresponding leaf are compared, the correction algorithm can only suggest words which share the same prefix. But the user's type errors might also occur in the beginning of the word. Assume that the user wrote "ivh" but intended to write "ich". The algorithm will traverse the tree and end up in a list of words with the prefix "iv". The auto-correction algorithm will not consider the word "ich" since it is stored in a different list. To cope with this, an additional linear list of the thousand most frequent words overall is kept. The algorithm will be performed on these words as well and is able to detect a large part of typing errors. Practice has shown that, it is more accurate to multiply their Levenshtein distance by a higher penalty factor. This is because the list contains words with very high frequency numbers.

Chapter 3

# Implementation

## 3.1 Swift Programming Language

The Kännsch application for iOS is programmed in Xcode with a language called Swift. This is a new programming language designed by Apple to offer an alternative to Objective-C. It is fully compatible with Objective-C and offers the same features but with a more readable syntax.

## 3.2 Framework and Libraries

In this section, a list of open source projects that are used to realize this application is given.

Tasty Imitation [10] is a simple keyboard implementation completely written in Swift. It is specifically designed to look like the standard keyboard by Apple. This keyboard was used as a framework and forms the major part of the project. Some minor changes were made to adjust the keyboard layout and a custom banner is added. This is the part above the actual keyboard, which is used to show several word suggestions based on the input of the user.

An implementation of the Levenshtein algorithm [12] was used as a basis for the word correction algorithm. The algorithm was changed to match our requirements.

ZipArchive [13] is a library to compress and decompress files. It is used to bundle the log files in a zip archive to send it to the Kännsch server. The library is written in Objective-C.

## 3.3 Retrieval of Local Dictionaries

The design and implementation of the Kännsch server infrastructure was a part of the master thesis of Marcel Bertsch. Its purpose will be explained briefly as

well as all mandatory parts that the client application needs to fulfill.

### 3.3.1 Research Logger

The research logger is that part of the application which is responsible for collecting data about the user. Many parts of it are designed to imitate the function of the Android counterpart. The logger is called at several input operations. In particular, it will log the following events:

- Every keystroke of the user.
- Every explicitly picked suggestion together with the originally typed input.
- Every automatically corrected word together with the originally typed input.
- Every reverted correction together with the originally typed input. A correction is reverted when the user presses backspace after an auto-correction took place.

All events are stored in a buffer and after several minutes saved as a JSON file. Every five hours, all logged files are bundled in a Zip archive and sent to the server together with a hashed user ID.

### 3.3.2 Kännsch Server

The main task of the server is to provide a regional dictionary based on the user's dialect. Every user input will be logged and sent to the server. It will then analyze the log and compare it to the data of other users. After a user has produced enough data, a list of words and frequencies that should represent the user's dialect is sent back to the application. The logging and sending of data continues on a regular basis. The whole process is designed so that the user should not notice any of these processes.

Chapter 4

# Results

## 4.1 Typing Error Analysis

This section describes a method to measure the distribution of various typing errors made by an average user. We are interested in what mistakes are being made and how often they occur. This information is used in two different ways. First, the resulting numbers are used as weights in the Levenshtein algorithm. Second, we can use the distribution to create test cases for evaluating the autocorrection algorithm.

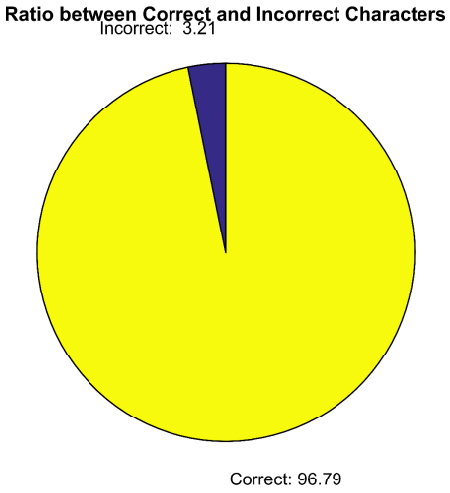
Our analysis is based on the data logs that were collected by the Kännsch Android application. The logs gather all keystrokes, picked suggestions and autocorrected words. These log entries are reinterpreted as a stream of deletion and insertion events. Whenever a sequence of letters was removed, it considers the same amount of letters that were inserted to replace them. A full Levenshtein algorithm (not the adjusted version) is calculated on the two words. The Levenshtein algorithm calculates a way to alter one word into the other. The used modification steps are summed up over the whole log.

For example, if the log states that the word "elefsnt" was corrected into "Elefant", seven deletion events ("t", "n", "s", "f", "e", "l", "e") and seven insertion events ("E", "l", "e", "f", "a", "n", "t") registered. The process detects that the sequence "elefsnt" is replaced with the input "Elefant". The Levenshtein algorithm computes that the first letter was capitalized and the fifth was replaced.

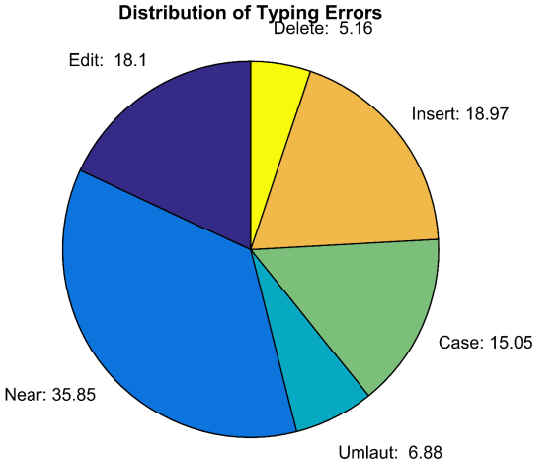
Some of the detected substitutions may be uncorrelated because the log files do not track cursor movements. This may lead to a problem whenever the user deletes a word, moves the cursor and continues to write. We will only accept the result of a substitution if at least one third is unchanged.

The following modifications are distinguished:

- Copy: A letter is unchanged.
- Near: A letter is exchanged with an adjacent key on the keyboard.




(a) Correct typed letters compared to misspelled letters.




(b) The distribution of errors made by an average user.

Figure 4.1: Figure 4.1a shows how many errors are made overall. Figure 4.1b splits up the blue part of Figure 4.1a to show which error occurred how often.

- Umlaut: E.g. if the letter ”ä” is exchanged with ”a”.
- Case: A letter is capitalized or vice versa.
- Edit: A letter is exchanged with a letter that does not fit into any of the first four categories.
- Insert: A letter occurs in the second word which is missing in the first.
- Delete: A letter was omitted in the second word.

3957 log files are used for the analysis. The results are illustrated in Figure 4.1. More than three percent of all typed letters are not intended. Hitting a key adjacent to the intended key occurs about twice as often than hitting a completely different key. Omitting a letter is the least frequent mistake.

## 4.2 Prediction Benchmark

The prediction benchmark is used to measure the performance of word completion and word prediction. It counts the amount of saved keystrokes using our application compared to writing every single letter of a text separately.

The benchmark splits a given text into a list of words and scans every word letter by letter while performing the algorithm used in the application to suggest words. If the original word is among the suggested words, the amount of letters

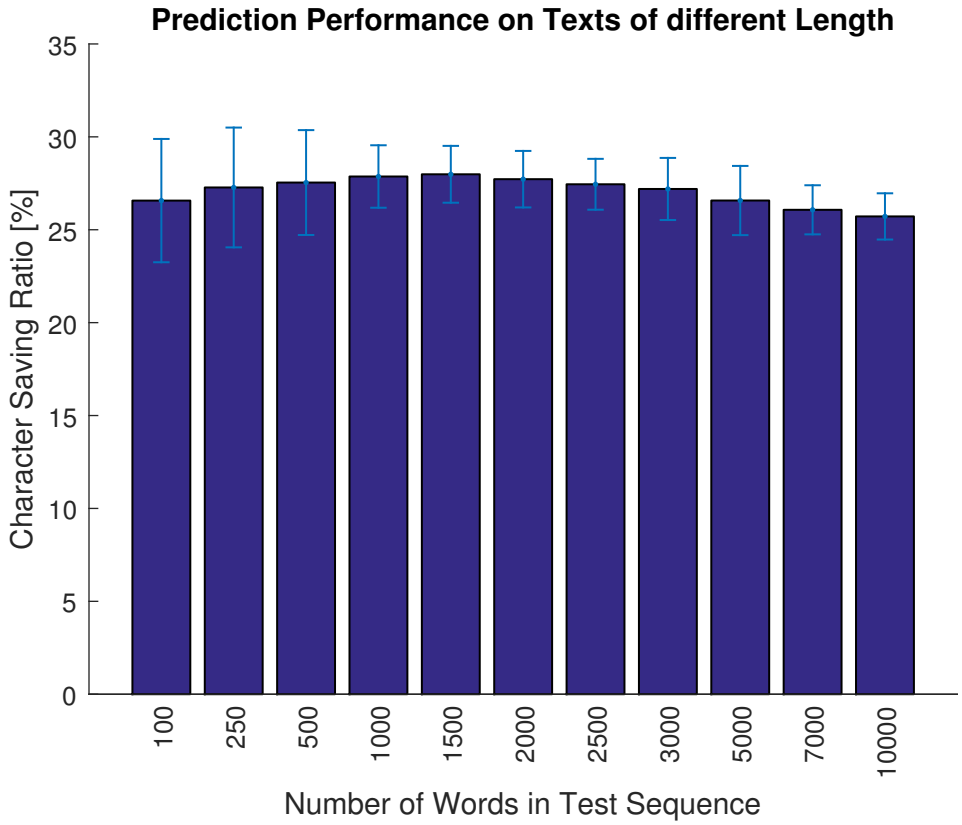



Figure 4.2: Performance of word prediction and word completion. The algorithms were performed on texts of various length.

that did not need to be typed are added to the result. This number is called *Saved Characters*. For example, if we intend to write the word "Elefant" and the suggestion shows up after typing "Ele", we saved four characters. We are interested in the *Character Saving Ratio* which is the result of dividing the *Saved Characters* by the total amount of characters in the text.

The algorithms were performed on parts of the Harry Potter books by J.K. Rowling. The books offer a large amount of orthography checked text passages. Some of the words and names are made up which are an additional challenge. The results in Figure 4.2 show that short input sizes perform worst since the dictionary has no chance to adapt to the writing style. Longer texts help the dictionary to adapt to the writing style. The standard aberration is smaller after long inputs which indicates that the dictionary gets more accurate. The overall performance settles between 26% which is about the same as the algorithm used in Google's LatinIME [3]. Figure 4.3 shows the performance while frequency adaption is disabled. That means all frequencies stay the same over the whole test run. While the results do not change much over time, the average performance

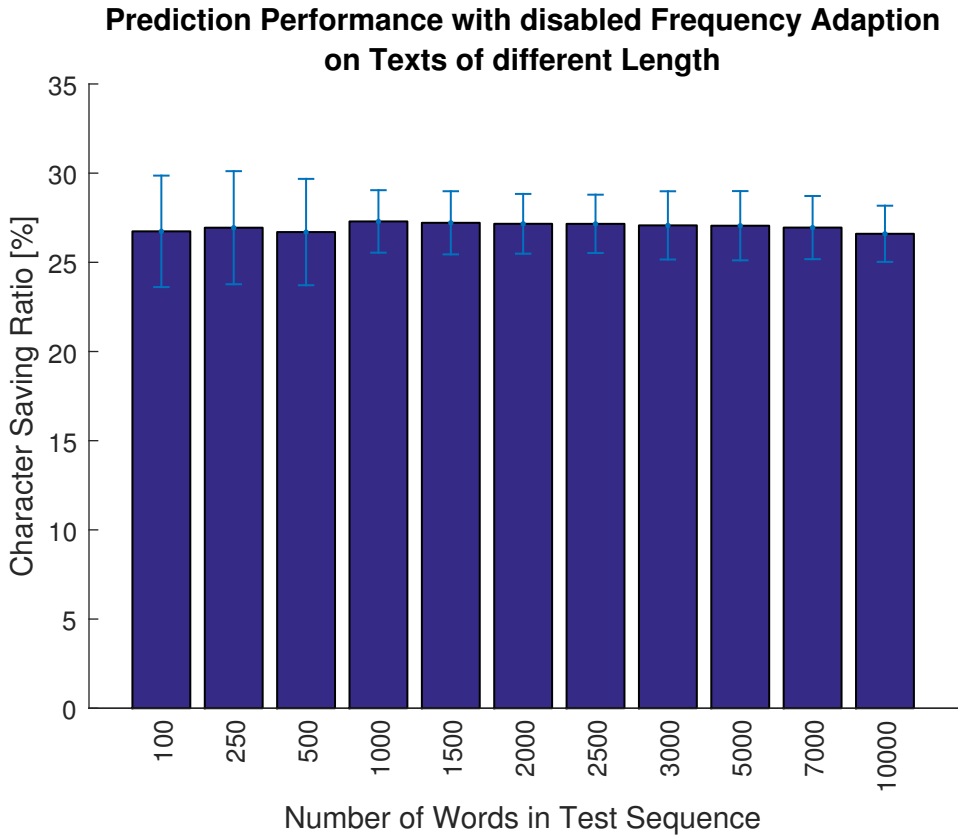



Figure 4.3: Performance of word prediction and word completion with unchanging frequency. The algorithms were performed on texts of various length.

is about the same as before.

## 4.3 Correction Benchmark

The correction benchmark will test the accuracy of the auto-correction algorithm. The benchmark puts mistakes into a text with correct spelling. It will feed our algorithm with the altered text letter by letter similar to the prediction benchmark. But it expects that the correct, original word is among the suggestions. We are interested in how many words can be corrected and how often the correction failed and suggested a wrong word instead. The *Corrected Words Ratio* is the division of the number of successfully corrected words divided by all words that contained at least one error.

Again, the Harry Potter books were used as input for the benchmark. The texts were altered according to the distribution results of Section 4.1 with a fixed random seed. The results are illustrated in Figure 4.4. The sample size for short

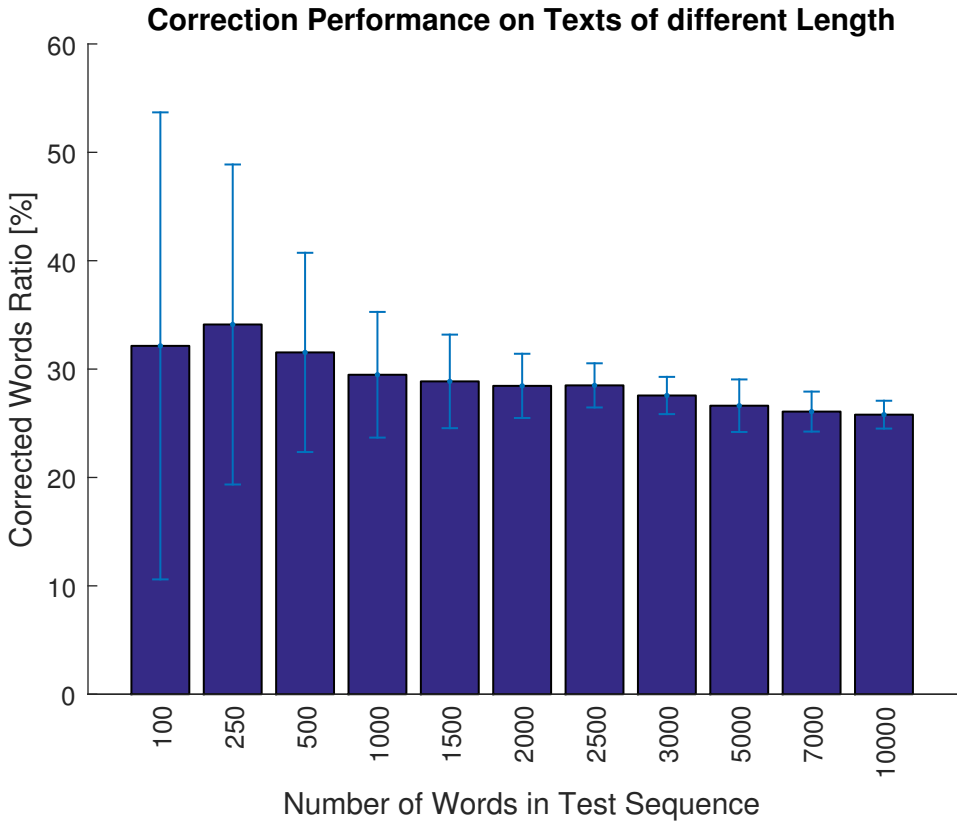



Figure 4.4: Performance of word correction. The algorithm was performed on texts of various length.

texts are too small because not more than 4% of all letters contain an actual error. This is the reason why tests with less than 500 words have very high standard aberrations. The correction performance remains above 25% even with high input sizes while the standard aberration gets smaller over time. Google's algorithm achieves a significantly higher performance. Depending on the test case, it reaches between 28% and 40% [3]. Figure 4.5 shows the performance while frequency adaption is disabled. While the standard aberration looks similar, the performance slightly increases over time instead of decreasing.

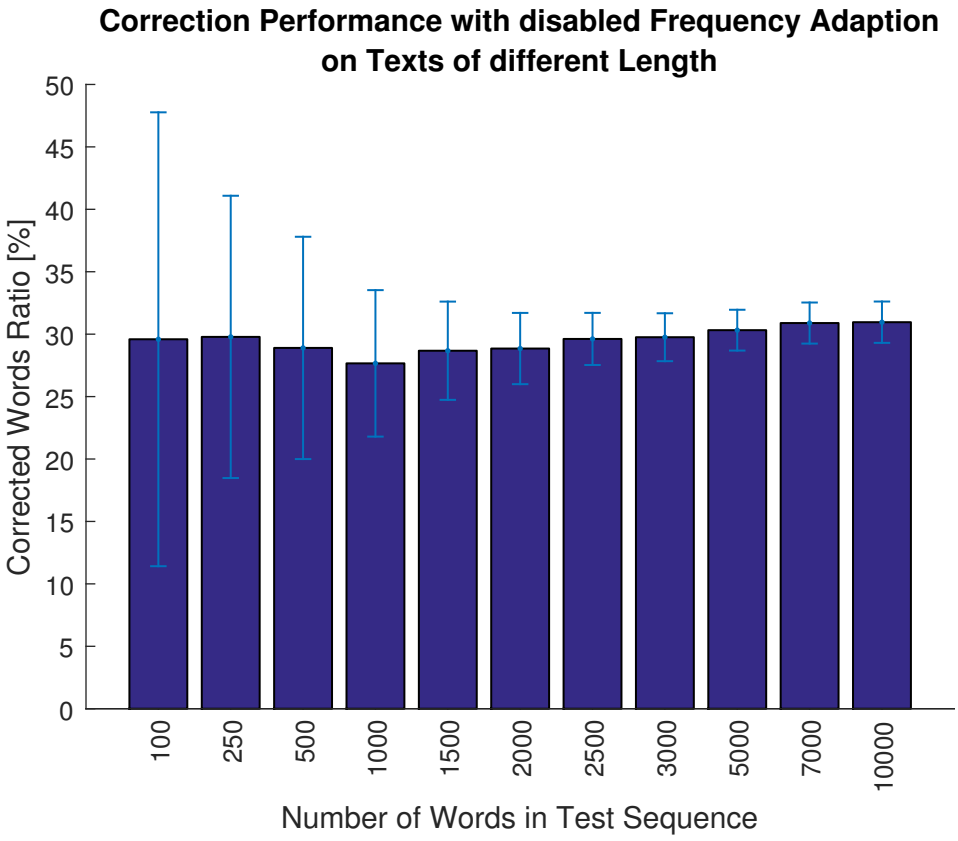



Figure 4.5: Performance of word correction. The algorithm was performed on texts of various length.

Chapter 5

# Conclusion and Outlook

The Kännsch keyboard application for iOS is a first step to give Apple iPhone users the possibility to profit from a Swiss German dictionary just like Android users already do. The keyboard offers word suggestions with prediction, completion and correction mechanics. It offers reasonable accuracy and speed that can compete with similar applications. It is designed to access the Kännsch server, which means it is based on the same resources as its Android counterpart.

Besides further improvements on speed and accuracy of the proposed algorithms to match other keyboards, there are more aspects that can be improved in the future. The application has no support for multiple languages. It may be extended so that it holds various dictionaries for more languages or dialects and chooses the right one based on context and user input. The keyboard layout may be more user friendly by adding more umlauts, special characters and emoticons like Emoji. Many keyboards such as Swype offer the possibility to swipe over the intended keys instead of typing them one after the other. Our application could be expanded with an algorithm that detects swipe gestures and matches them with words in the dictionary.

# Bibliography


[1] Peer, L.: Kännsch - a Swiss German Keyboard for Android. Master's thesis, ETH Zürich (2014)

[2] Google: Android. https://www.android.com/ Accessed 2015/09/03.

[3] Bertsch, M.: Kännsch - Improving Swiss German Keyboard. Master's thesis, ETH Zürich (2015)

[4] Apple: iPhone. http://www.apple.com/iphone/ Accessed 2015/09/03.

[5] Apple: QuickType. http://www.apple.com/ios/whats-new/quicktype/ Accessed 2015/09/03.

[6] Apple: iOS. http://www.apple.com/ios/ Accessed 2015/09/03.

[7] Google: Android Open Source Project LatinIME. https://android.googlesource.com/platform/packages/inputmethods/LatinIME/ Accessed 2015/09/03.

[8] Swype: Swype. http://www.swype.com/ Accessed 2015/09/03.

[9] SwiftKey: SwiftKey. http://www.swiftkey.com/ Accessed 2015/09/03.

[10] Baboulevitch, A.: Tasty Imitation Keyboard. https://github.com/archagon/tasty-imitation-keyboard Accessed 2015/09/03.

[11] Levenshtein, V.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady **10** (1966) 707

[12] Riegler, M.: Levenshtein - Swift. https://gist.github.com/kyro38/50102a47937e9896e4f4 Accessed 2015/09/03.

[13] ZipArchive: ZipArchive. https://github.com/ZipArchive/ZipArchive Accessed 2015/09/03.

[14] Crockford, D.: JSON in Java. https://github.com/douglascrockford/JSON-java Accessed 2015/09/03.