Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Kännsch - Updates and Improvements to the Swiss German Keyboard

Bachelor Thesis

Valentin Trifonov

vatrifon@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

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

August 16, 2016

Acknowledgements

I would like to thank my supervisors Philipp Brandes and Laura Peer for their help and support for this project. Thanks to Professor Roger Wattenhofer and the Distributed Computing Group at ETH for giving me the opportunity to work on such an interesting and practical project for my bachelor thesis.

Abstract

In the German speaking parts of Switzerland, it is common among young people to write in their dialect when communicating in a non-formal setting. As there are masses of Swiss German dialects, it is very difficult to build a mobile keyboard application that can suggest Swiss German words taylored to the user's typing style. One project which attempts to fill this gap is *Kännsch*, a mobile keyboard for Android smartphones, developed over two master thesis at the ETH. In this project we refine some of the application's internal mechanisms to provide more accurate word suggestions, and we update the application to a newer, more visually appealing and feature-rich version. Finally, we build a framework that uses the accumulated usage data to estimate the effectiveness of the different variants of the new algorithms.

Contents

Chapter 1

Introduction

Over the last few years, smartphones have become very popular and a central part of our everyday lives. Typing with a smartphone's mobile keyboard, with the thumbs on the touchscreen, is by far not as comfortable as typing on a computer keyboard. To ease typing, modern keyboard applications such as A.I. Type [1], SwiftKey [2] or similar, include functionality such as suggesting words for completion or correction in a bar above the keys, while the user is typing. A lot of research has gone into optimizing these algorithms and improving the user experience. Some keyboards even have features like memorizing common word combinations that a user types and trying to predict the next word entirely, or allowing typing by dragging the finger across the keyboard roughly over the letters of a word, known as swyping [3] or gesture typing [4].

A major part of how young people use their phones is for informal texting among friends or on social media. In the German speaking part of Switzerland, many dialects have emerged. Unlike other countries, Swiss people like to type in their dialect when in an informal setting.

All the typing aid features of modern keyboard applications are based on static dictionaries containing words and metadata of the used languages. And therein lies the dilemma: the lack of a common dictionary makes accurate suggestions difficult for informal Swiss German conversations.

1.1 Contributions

In this project, we are adding improvements to an existing Android mobile keyboard which tries to tackle the problem of typing in Swiss German, *Kännsch*. We will call the original version of Kännsch, i.e. the precursor of our application, *Kännsch-KitKat*.

Kännsch-KitKat was first released in August 2014, and since then the user interface has not been adapted, which gives the Keyboard an outdated feel. In particular, it does not comply with *Material Design* [5], Google's design guidelines for Android applications, which they introduced with the release of Android version Lollipop.

We therefore chose for this project to re-implement the functionality of the Kännsch project in a fork of the current version of the open-source keyboard Kännsch-KitKat is based on. Not only does the new version comply with Material Design, but it includes refined algorithms for more accurate word suggestions, as well as handy new features.

Furthermore, we decided to work on some of the internal mechanics as well. The keyboard includes improvements for our features for allowing quick switching between the different input languages, and better generation of personalized dictionaries.

1.2 Outline

In Chapter 2 we have a look at the currently available Android mobile keyboards, as well as at the precursor of our application, the approaches for solving the addressed problems, and the open-source application it is based on. In Chapter 3 we explore the functionality of our Android application and give an overview of the architecture of the relevant components. In Chapter 4 we have a look at the algorithms used at our server backend for generating personalized dictionaries. Chapter 5 summarizes the results from the evaluations of our different approaches for said algorithms. Finally, Chapter 6 concludes the project and gives suggestions for future work.

Chapter 2

Related Work

2.1 Mobile Keyboards

There is a countless number of mobile keyboards available for Android mobile phones. The *Google Keyboard* [6], *A.I. Type* [1], and *SwiftKey* [2] are a few examples.

The *Android Open Source Project* [7] includes a mobile keyboard called *LatinIME* [8]. The most notable related pieces of work and precursor projects continued in this thesis are *Kännsch - a Swiss German Keyboard for Android* [9] by Laura Peer, and *Kännsch - Improving Swiss German Keyboard* [10] by Marcel Bertsch. The keyboard developed in these theses is based on LatinIME. We will go into these projects in the next two Sections.

Since mobile keyboards are rendered on a touchscreen, the user cannot actually feel the keys. In addition to that, users usually type using only their thumbs. Therefore, typing is significantly more tedious and takes more time to get used to it, and it is much more likely to mistype. Nonetheless, nowadays smartphones are ubiquitous and thus mobile keyboards are used constantly. For that reason, it is taken for granted that a modern mobile keyboards include features to assist the user as he is typing. More often than not, mobile keyboards include a *suggestion bar* above the keys which shows a few (usually three) words, that the user is likely to have in mind as he is typing. These include *corrections* of the previously typed word, *suggestions* (or *completions*) for the currently typed word, both combined (a suggestion which also corrects the typed prefix), or *predictions* for the next word.

Another feature that seems to have become compulsory for mobile keyboards is the support for what is known as *swype* [3] input or, for Google's keyboards, *gesture typing* [4]. Instead of having to tap each key for separately, *swyping* allows the user to drag the finger over the keys and roughly draw the shape of the word he is trying to type. The keyboard then analyses the motion, and tries to guess which word was most likely the intended word by the user, based on the brushed keys and their order. The keyboard then inserts the most probable word and displays the alternatives in the suggestions bar, so that the user can quickly correct the text, if need be.

The algorithms used to compute these typing aids are based on *dictionaries*, which usually include a collection of words and associated metadata, such as how common each word is. LatinIME uses a number between 0 and 255, called the *word frequency*, as a word commonness measure. In addition, it stores for each word if it can be classified as an offensive word, if it is often used for the beginning of a sentence, and a timestamp indicating when it was added. It also stores *bigrams*, expressing a combination of two words which are commonly used together. The new version uses *n-grams*, a generalization to include combinations of more than two words which belong together. Like the words, bigrams and n-grams are associated with a frequency.

None of these keyboards focuses on people typing informally in Swiss German, as static dictionaries for languages simply do not suffice for accurate predictions, given the many different dialects of the language.

Most of the mentioned keyboards have some limited means of adaptation to a users typing style by storing and maintaining what LatinIME refers to as a *user history dictionary*. This dictionary is a collection of commonly typed words and word sequences which are not in the main dictionary and thus taylored to the user. This approach does not work good enough for a user typing in a dialect: they would have to type their dialect-specific words and phrases multiple times until the keyboard application memorizes enough of them and assigns them high enough frequencies to give reasonable suggestions. For users typing in an Swiss German dialect, it would take a very long time for the keyboard to adapt, and as such it is unsuited for them.

Most competing keyboards have other focuses. For example, some keyboards, like the *Go Keyboard* [11], allow personalizing the keyboard by using different themes, background pictures or sound effects. *Kika Keyboard* [12] provides, for supported applications, searching and sending animated GIF images directly from the keyboard. Almost all keyboards have a view for inserting *emoji* [13] to the typed text.

2.2 Kännsch

For Laura Peer's master thesis [9], she developed the Android mobile keyboard Kännsch, a fork of Google's open source keyboard LatinIME, made with Swiss German dialects in mind.

The project adds support for multiple simultaneously active input languages by combining the suggestions and corrections of multiple dictionaries. This allows the user to switch between Swiss German, Standard German, English, French, Spanish and Italian at will. Standard German and French are Switzerlands most used national languages, therefore Swiss users have to switch frequently between these languages.

LatinIME uses *input method subtypes*, i.e. combinations of a language for the used dictionary, and a keyboard layout. Since it does not support automatic switching for the former, it requires the user to select another subtype from the Android notification area, which is somewhat cumbersome.

Kännsch improves this by using a mechanic referred to as the *Language Detector* to estimate the currently used language based on the last five typed words. It generates weights for the different dictionaries roughly based on if they will likely be used next. Each dictionary weight is used to skew the score of its generated suggestions, such that for the top words in the suggestions bar dictionaries likely to match the user's current input language are preferred. We will cover the algorithm in greater detail in Chapter 3.

Kännsch ships with the *Swiss German Core Dictionary*: a collection of 990 words used in almost all Swiss German dialects, obtained by crawling Swiss German groups on Facebook.

Further, Kännsch applies an aggressive word learning scheme: each word a user types is added to Android's *user dictionary* [14]. This simple way of memorizing words speeds up the process of adapting to the users typing style. The words in the user dictionary are included in the suggestions and remembering a word only requires it to be typed once. We will discuss the advantages, drawbacks and alternatives of this approach in Chapter 3.

The app uses LatinIME's built-in logger to store a history of events used to analyse the keyboard usage. In particular, it can be used to recover a user's typed text, which in turn can be used for language research and as a way to collect data for Swiss German dialect dictonaries. Kännsch logs this data, stores it in files and sends them to our evaluation server for further analysis. It is important to note, that for password fields no data is logged, that all digits in the logs are replaced with zeros, and that each user is assigned a unique and non-reversible hash used as the ID to work and associate the logs with. These measures ensure that the user's privacy is respected. Users of the keyboard are informed about the data that is being collected.

2.3 Improving Kännsch

With Marcel Bertsch's master thesis, *Kännsch - Improving Swiss German Keyboard* [10], the Kännsch project is continued. The Swiss German Core Dictionary is extended to include 6161 common Swiss words. The new version of the application periodically synchronizes with our server. Additionally to submitting research data, it receives updates for the Swiss dictionary.

This approach allows us to work on the algorithms for personalised dictionaries on the server-side, and the client app only needs to download and to apply them. That way, Swiss users do not rely solely on their own typed dialect words, but recieve updates of words they will likely use once their dialect can be estimated well enough.

The update application uses an HTTPS connection to the server, and an opt-out option for users that do not want to upload any data.

To create new dictionaries, *collaborative filtering* techniques are applied and evaluated. In particular, *K-Nearest Neighbors* with various similarity measures, *K-Means Clustering* and *Latent Semantic Analysis* are explored. Refer to the thesis [10] for a detailed description and evaluation results of the algorithms.

Chapter 3

Männsch Mobile Keyboard

Our application is based on LatinIME [8], an open source implementation of an Android mobile keyboard. For this project we chose to re-implement the functionality of Kännsch starting from the Marshmallow release of LatinIME, i.e. the source code as it was included in the Marshmallow release of the Android Open Source Project.

This release includes new features that were not available in *Kännsch-KitKat*. Most notably, the application uses *Material Design* [5]. This improves the user experience as the keyboard design complies with the uniform design of the Android OS and applications, instead of having an outdated feel. In addition, the internal functionality has been further elaborated. Static language dictionaries now work with n-grams instead of bigrams, which allow for more accurate word predictions, since they are no longer based on only the preceding word.

The following Section includes an overview of the implementations for the relevant components of Kännsch.

3.1 Architecture Overview

3.1.1 Dictionaries

As previously mentioned, LatinIME works with static dictionaries. It includes implementations for the *BinaryDictionary*, a dictionary supporting serialization to a compact binary format. A BinaryDictionary stores words with associated metadata. Most notably, each word has a *frequency*, a value between 0 and 255, representing how commonly a word occurs. In addition to words, the BinaryDictionary can also store n-grams, i.e. sequences of words which belong together. Examples of n-grams could include “How are you” or “Happy birthday” or “What the hell”. Like words, n-grams also have an associated frequency.

LatinIME’s dictionaries support a fuzzy search over the words, used to generate suggestions as well as corrections. The outputs are assigned a *score* value.

A high quality match and a high word frequency both increase the score. Thus, the words with the highest score are most likely to be the word a user meant to type.

While these dictionaries work well in countries like Germany or Britain, for Swiss people they are very unsuitable.

The first reason is that there are a great many different Swiss dialects. It is very difficult to classify and define them, let alone develop and ship designated dictionaries for each one of them individually.

The second reason is that Swiss people commonly need to switch languages. The same dictionary cannot be reused for German and Swiss German, but depending on the formality of the setting and the recepient, the users might choose or prefer to write in either one of them. We will take a look at how Kännsch handles these problems in the next Section.

3.1.2 Languages

We chose to reuse LatinIME's robust dictionary implementation, but we also wanted to add multilingual support.

Kännsch has options in the application settings where the user can activate any combination of the languages English, German, French, Spanish and Italian. Swiss German is always active. For all lanugages other than Swiss German, LatinIME provides dictionaries, which we ship with Kännsch. For Swiss German we initially use the *Swiss German Core Dictionary*, a collection of Swiss German words common among all dialects, originally containing 990 words obtained by crawling Swiss German Facebook groups (see Laura Peer's master thesis [9]). It has later been improved using the accumulated usage logs and now it contains a total of 6161 words (see Marcel Bertsch's master thesis [10]). Kännsch adapts the Swiss German dictionary over time to match the user's dialect (described in Section 3.1.3).

Modifying LatinIME's source code allows us to combine the suggestions and corrections of all selected language dictionaries into the suggestion set, that LatinIME uses for its single input language. The words used to display the top suggestions to the user in the suggestion bar, and to apply autocorrections, thus come from a mixture of language dictionaries.

The user will likely want suggestions in the language he is using at a given moment, not all languages he is using in general. For this reason, Kännsch is using the *Language Detector*, a scheme used to estimate the the most likely current input language. Please refer to [9] for a detailed description of the language detector mechanism. Here we will give a brief summary of the algorithm.

As already mentioned, a dictionary's suggestions come with a score value attached. The different language dictionaries' score values need to be scaled to produce a final ordering for the suggestions, where relevant, currently used input languages are preferred over other active but unused languages. “Scaling” here means, that score values need to be set higher or lower, depending on which language’s dictionary they were generated from.

The language detector uses a sliding window of size n (in our application set to 5). When estimating the input language, the last n typed words are considered. Each language is assigned a *boost factor*, initially 1. For each word in the sliding window which can be found in the language’s dictionary, this language’s boost factor is incremented. For example, if the last 5 typed words were “hey what’s up, wie geht’s?”, then English has the boost factor 4 because of the words “hey”, “what’s” and “up” and German has the boost factor 4 because of the words “hey”, “wie” and “geht’s”. Thus, the formula for a language’s boost factor is given as:

$$b(l) = 1 + \sum_{i=1}^{n} \mathbb{1}_{D_l}(w_i) \tag{3.1}$$

Where:

$$\begin{aligned} l \in \{\text{English}, \text{German}, \cdots\} &= \text{a possible input language} \\ b(l) &= \text{the boost factor for language } l \\ \mathbb{1} &= \text{the indicator function} \\ D_l &= \text{the set of all words in the language } l \\ w_i &= \text{the } i\text{-th word in the sliding window} \end{aligned}$$

The higher a language’s boost is, the more likely we consider it to be the language the user is using for his current input. Kännsch-KitKat used the language’s boost factor to multiply the score value of all suggestions from the corresponding dictionary (we call this the *kitkat-ld* algorithm):

$$scale_{kk}(s, l) = b(l) \cdot s \tag{3.2}$$

Where:

$$\begin{aligned} s &= \text{the dictionary score value} \\ l &= \text{the language of the dictionary the suggestion came from} \\ scale(s, l) &= \text{the scaled score value for score } s \text{ and language } l \end{aligned}$$

We evaluate the language detector performance using the *performance evaluator*, which gives us the ratio of saved characters to total characters for a piece of text. We describe the performance evaluator in greater detail in Section 5.1.

In addition to keeping track of the saved characters, we modified the performance evaluator to log the score values of all suggestions generated by LatinIME’s Dictionary implementation. Figure 3.1 shows a histogram of the distribution of all suggestion scores.

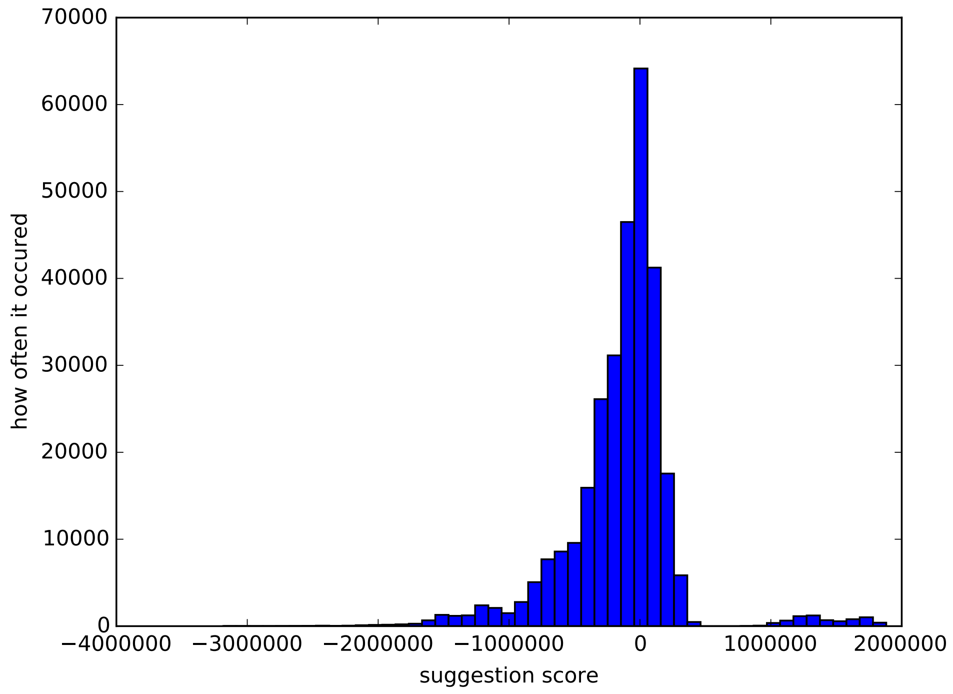


Figure 3.1: Distribution of suggestion scores for the first 1000 words of J.K. Rowling's *Harry Potter and the Sorcerer's Stone*. Note that the mass of suggestion scores lie around 0 and that the scores of this group do not exceed 800'000, and that there is a second group of values roughly between 1'000'000 and 2'000'000.

The LatinIME version, on which we based our update of Kännsch, uses both positive and negative values for suggestion scores. Accordingly, multipying with the boost factor to increase a suggestion's score has the opposite effect for negative scores, which is why kitkat-ld can give bad results. We therefore chose to deveolp some alternative algorithms for transforming suggestion's score values to rank input languages.

We introduce the notion of a normalized boost factor, i.e. a boost factor scaled to be between 0 and 1:

$$n(l) = \frac{b(l)}{\max_{l' \in \mathcal{L}} b(l')} \tag{3.3}$$

Where:

$$\begin{aligned} n(l) &= \text{the normalized boost factor for language } l \\ \mathcal{L} &= \text{all active input languages} \end{aligned}$$

By normalizing the boost factor we can hinder suggestions of unused languages to rank high among all suggestions, instead of boosting suggestions of desired languages. This avoids drowing suggestions from auxiliary dictionaries, such as the dictionary containing the names of the user's contacts, in boosted suggestions from the language dictionaries.

One simple fix to avoid the problem with negative scores is our *zero-center-ld* algorithm:

$$scale_{zc}(s, l) = \begin{cases} s \cdot n(l) & n(l) \geq 0 \\ \frac{s}{n(l)} & \text{otherwise} \end{cases} \tag{3.4}$$

While zero-center-ld does not scale negative scores in the wrong direction, it does have a flaw. The output scores around 0 are scaled a lot less than suggestions which inheritantely have high (positive or negative) scores.

For the next strategy, *grouped-ld*, we distinguish between *group 1 suggestions* and *group 2 suggestions*. The score values of group 1 suggestions lie mostly between $-1'000'000$ and $800'000$ (which we define as g_1^- and g_1^+, respectively). As seen on Figure 3.1, they make up for the majority of suggestions, corrections and predictions. Group 2 suggestions lie between $800'000$ and $2'000'000$ (g_2^- and g_2^+) and are a special type of corrections which are considered very likely to be picked, where the typed and suggested word are almost identical and the letters differ only in terms of accents or capitalization. Grouped-ld clamps the scores between the bounds of their group and normalizes them along the lower bound before scaling:

$$scale_{gr}(s,l) = \begin{cases} n(l) \cdot (s_1 - g_1^-) + g_1^- & \text{for } s \leq g_1^+ \\ n(l) \cdot (s_2 - g_2^-) + g_2^- & \text{otherwise} \end{cases} \tag{3.5}$$

Where:

$$\begin{aligned} s_1 &= \min(\max(s, g_1^-), g_1^+) \\ s_2 &= \min(\max(s, g_2^-), g_2^+) \\ g_1^- &= -1'000'000 \\ g_1^+ &= 800'000 \\ g_2^- &= 800'000 \\ g_2^+ &= 2'000'000 \end{aligned}$$

While grouped-ld works good, it still has one bothersome flaw. Scaling both suggestion groups seperately has the side effect that no matter how low the boost factor of a language is, i.e., no matter how unlikely it is to be the desired language, the group 2 suggestions will always dominate the group 1 suggestions, since we forced them to lie between g_2^- and g_2^+ (g_1^- and g_1^+, respectively), even after applying the scaling.

A modification to grouped-ld yields our *group-1-ld* algorithm. The intuition here, is that if we are unsure about a language, group 2 suggestions should be forced to lie in a lower inteval (h^- to h^+), chosen such that they only show up if there are no group 1 suggestions with high scores available.

$$scale_{g1}(s,l) = \begin{cases} h^- + n(l) \cdot (h^+ - h^-) & \text{for } s > g_1^+ \wedge n(l) < 1 \\ scale_{gr}(s,l) & \text{otherwise} \end{cases} \tag{3.6}$$

Where:

$$\begin{aligned} h^- &= -400'000 \\ h^+ &= 100'000 \end{aligned}$$

See Section 5.2 for performance evaluations of the different language scaling algorithms.

3.1.3 Research Logging and Dictionary Updates

The Kännsch project follows the following evolution cycle: the user uses the keyboard and types words. The keyboard logs the whole input, but scrapes away information that might invade the user's privacy such as numbers (which might contain credit card or phone numbers), e-mail addresses, inputs from fields marked for passwords, and the original string used to generate the user's identity.

The logged data is sent to our server, where we can analyze it and develop new algorithms. In return, the server sends back a dictionary update, i.e., a set of words and bigrams with associated frequencies. The dictionary update is automatically composed and aims to estimate the user's dialect and thus match his typing style. The communication is secured over https.

The update data from the server is inserted into the keyboard's Swiss German dictionary (described in 3.1.2) to increase its precision.

To collect said keyboard usage data, the Kännsch project came to rely on the logs generated by the *Research Logger*, a program construct that comes with LatinIME-KitKat. While a simple boolean flag made it very easy to activate research logging in Kännsch-KitKat, it has been removed from LatinIME in the meantime. We ported it from Kännsch-KitKat for the current version of Kännsch.

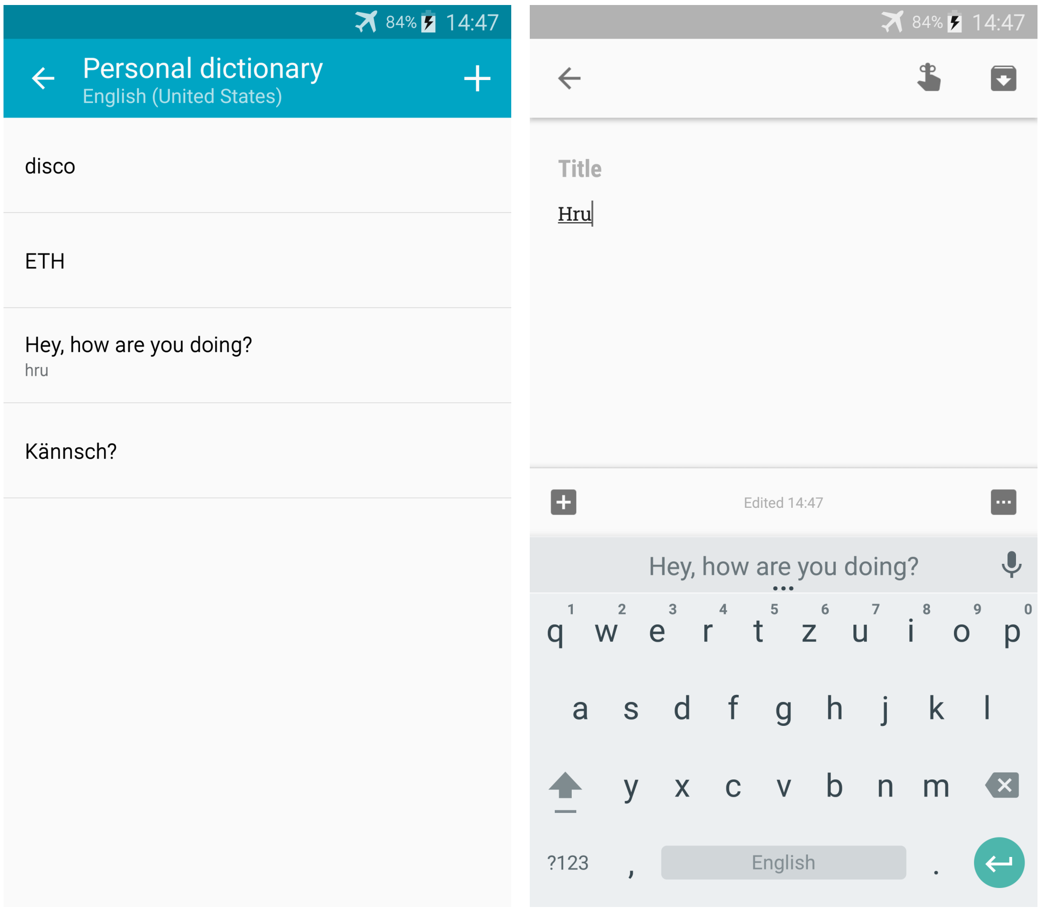
3.1.4 Personalization

With our project, personalization in terms of typing style comes first. Additionally to adapting to a user's dialect as described in the last Section, our keyboard provides other means of adaptation as well. One of them is Android's *user dictionary* [14].

The user dictionary is a service provided by the Android OS, which maintains a dictionary for user-specific words and frequencies. The user dictionary is shared among all applications used on the phone and can be displayed and edited in LatinIME's settings. It allows the user to manually add words and phrases, which can be accessed by all keyboards and spellchecker applications. These words can then be suggested, and autocorrecting them can be avoided. One feature that comes with the user dictionary are shortcuts: a user can define a word or piece of text to act as a shortcut for another word or phrase. When the user then types the shortcut text, Kännsch suggest the expanded phrase. Figure 3.2 shows a screenshot of a suggestion with a user-defined shortcut.

Another piece of software heavily used by the keyboard is the *user history dictionary* (not to be confused with the user dictionary). The user history dictionary has the same purpose as the user dictionary, i.e., to store user-specific words for suggestions and corrections. We discuss the key differences in the next Section.

The user history dictionary is not an Android service, but part of LatinIME. As such, it is managed and used only by the keyboard itself. There is no settings screen where a user can see the stored words and define shortcuts, and the fequencies are entirely managed and adapted by the keyboard. LatinIME uses this construct to memorize falsely autocorrected words. It is stored in the same format as the static language dictionaries.


84% 14:47

Personal dictionary
English (United States)

disco

ETH

Hey, how are you doing?
hru

Kännsch?

84% 14:47

Title

Hru

Edited 14:47

Hey, how are you doing?

(a) Defining entries (b) Suggestions

Figure 3.2: The user can defining entries with shortcuts in the user dictionary, and the keyboard will suggest the phrase when it encounters the corresponding shortcut.

Kännsch-KitKat used the user dictionary to aggressively learn Swiss German words, and stored every typed word in the dictionary with a constant frequency value. This approach has many downsides: since every typed word is memorized, every typogrphic error and every word only used once will be suggested and corrected to in the future. Furthermore, since the frequencies are set to a constant value without ever being adapted, all words are regarded as equally important and common.

For the current version of Kännsch, we chose a different approach for the user-specific words. We modified the user history dictionary to store typed words in any case (in LatinIME it is only used when autocorrection is enabled). This has the advantage that the keyboard maintains and balances word frequencies internally, reusing the algorithms from the user history dictionary.

Further, with every dictionary update the user history dictionary is cleared, and the user-specific words are merged into the dictionary update by the server. That way, the user history dictionary acts only as a cache for user-specific words an update containing them is available. Managing the user-specific words happens on the server side, we refer to that as the *server user history dictionary*.

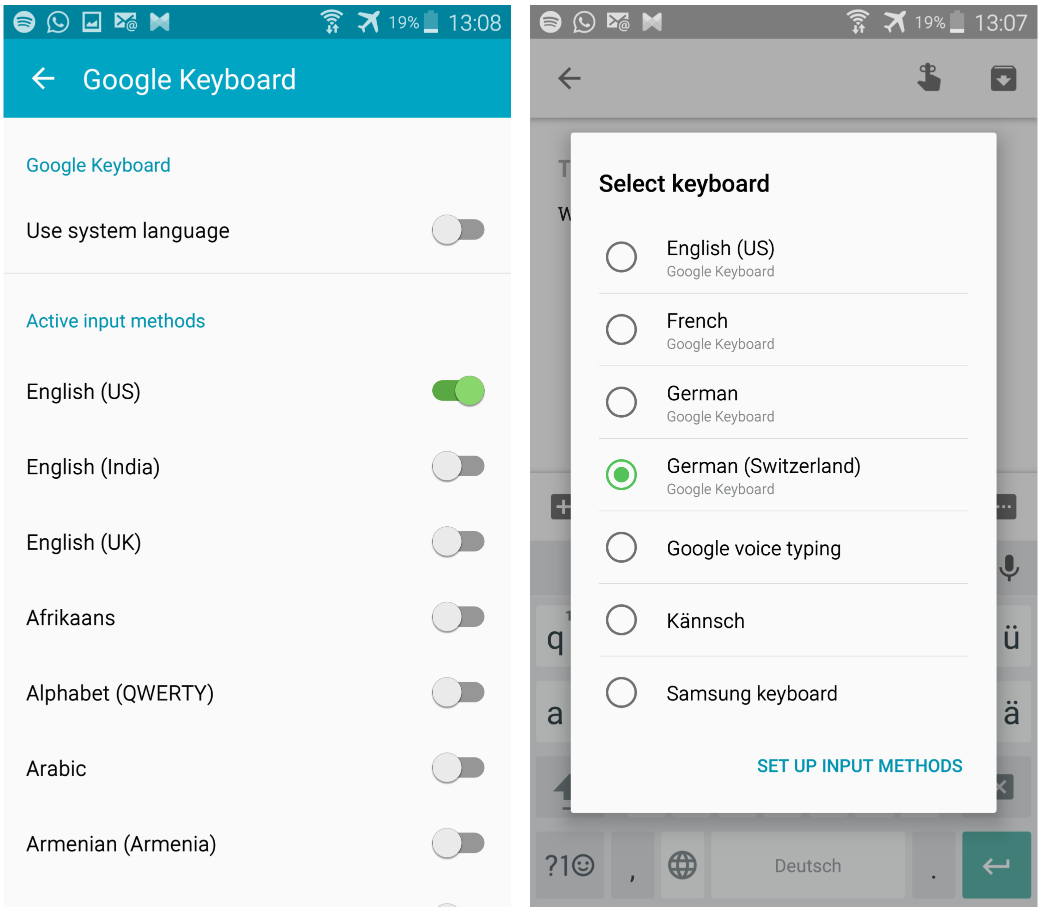
With this approach, a user's history dictionary is in sync between all his devices. A user can reinstall the keyboard application and it will rapidly adapt to his typing style with the next dictionary update, using only his ID. In addition, we can develop algorithms for balancing user-specific and dialect words, i.e. server user history dictionary and *cluster dictionary* words on our server. We will come back to that in Chapter 4. Any updates to the algorithms maintaining the server user history dictionary can be rolled out by updating the server only.

3.1.5 Other

A few other minor adaptations to LatinIME were necessary to provide the best user experience for Kännsch.

For one, we wanted to make sure that there would be no major problems with the application which go unnoticed or do not occur on our testing devices. For that reason we used *acra* [15], a framework which allows automatic crash reports to be sent to our server and is very easy to integrate into the existing application.

Another thing which seemed out of place for our application is the Android *input method subtype* API, which does not make sense with our language model. LatinIME uses input method subtypes, i.e. user-selected language models, consisting of an input language and a keyboard layout. When the user wants to switch the language used for autocorrections, suggestions, and the keyboard layout, he has to select the subtype from Android's notification bar (see Figure 3.3), or cycle through them with a language switching key on the keyboard.



19% 13:08

Google Keyboard

Google Keyboard

Use system language [off]

Active input methods

English (US) [✓ on]

English (India) [off]

English (UK) [off]

Afrikaans [off]

Alphabet (QWERTY) [off]

Arabic [off]

Armenian (Armenia) [off]

19% 13:07

Select keyboard

- English (US) — Google Keyboard
- French — Google Keyboard
- German — Google Keyboard
- [✓ German (Switzerland) — Google Keyboard]
- Google voice typing
- Kännsch
- Samsung keyboard

SET UP INPUT METHODS

(a) selecting active input method subtypes (b) subtypes work like self-contained keyboards

Figure 3.3: For keyboards which make use of Android's input method subtype interface, the user has to manually switch subtypes when he wishes to type in another language.

As mentioned earlier, Swiss users have to switch commonly between languages, and we chose to combine all language's suggestions and corrections to be active at all times with our language detector ranking them. The subtype selection model therefore does not make sense for Kännsch and we removed the corresponding interface. Instead, we added a preference in the application settings where the user can select the active input languages, and one where the user can select a keyboard layout. The options for the latter are *qwerty* (the standard keyboard layout), *qwertz* (the Standard German keyboard layout, with the 'y' and 'z' keys swapped), and *swiss* (qwertz with keys for the German umlaut letters).

Chapter 4

Kännsch Server Backend

As described in Section 3.1.3, our application synchronizes periodically with our server. It provides us with logged usage data, and relies on the returned dictionary updates. Our application components for estimating a user's dialect and providing them with dialect words reside on the server-side. Additionally, the server maintains sets of distinct per-user words and phrases.

4.1 Architecture Overview

We call our web application *Kännsch-server*. Kännsch-server accepts requests from Kännsch applications, receives and stores the usage logs. It checks for a new available dialect dictionary for the user, merges user-specific words into a dictionary update and sends it to the client. The stored logs are used for further analysis and for the creation of personalized dictionary updates.

Additionally, our *daily data aggregation* application runs on the server once a day. It looks for new logs and processes them. It reconstructs the user's typing history and records all typed words. It then updates our *words and bigrams* database, a store for all words and bigrams typed by each user. It also keeps track of all associated *occurences*, i.e. how often a user has typed a word or bigram in total.

The application further runs a clustering algorithm, to associate groups of users with similar dialects. The output is stored as a set of words and bigrams (with frequencies) corresponding to the users' dialects. It is used for dictionary updates, delivered by the Kännsch-server. In the next Section we look at the algorithms used for finding said dialect groups.

Please refer to our precursor project, Marcel Bertsch's master thesis [10], for more details on the Kännsch-server and the daily data aggregation process.

4.1.1 Clustering

To define dialects, we use the unsupervised learning technique of *clustering.* The application builds *word vectors*, i.e. vectors containing a user's word frequencies for all typed words our application has encountered and which we consider not to be typographic mistakes. Clustering means that we try to divide the users' word vectors, lying in a high-dimensional euclidian space, into groups. Each group, called *cluster*, is a collection of word vectors, such that they are as similar as possible within the cluster. "Similar" in this case is defined as the euclidian distance. A cluster *centroid* is the mean vector of a cluster's vectors. Each word vector has a *cluster membership* variable, assigning it to a cluster. This problem is referred to as *K-Means.* In our case, the cluster centroids are the different dialects we are trying to distinguish, and the cluster memberships assign a dialect to each user.

One algorithm to find an approximation for the optimal clusters and memberships is called *Lloyd's heuristic.* It is an iterative algorithm and works by randomly initializing cluster centroids and memberships, and then alternatively updating centroids as the mean cluster vector, and memberships as the nearest cluster centroid, until the memberships have converged. Please refer to Marcel Bertsch's thesis [10] for a more detailed description of the algorithm.

In this thesis, we chose to evaluate another algorithm for clustering. It is called the *expectation maximization* or *Soft-EM* algorithm [16]. It is an iterative algorithm and follows the same outline as K-Means, but there are a few important differences. Soft-EM models a cluster's samples as a Gaussian distribution. The analogous to K-Mean's cluster centroid would be the mean value of Soft-EM's cluster. Additionally, Soft-EM models clusters such that they can have different variances per-cluster and per-dimension (the frequency of specific words in our case). This model of the data corresponds to a *Gaussian Mixture Model.* Also, with Soft-EM, the samples are not mapped to clusters with a many-to-one relation (*hard assignments*, as in K-Means), but instead for every sample a vector of cluster membership probabilities is maintained (*soft assignments*). These are calculated according to the modeled Gaussian distributions.

The Soft-EM algorithm is a generalization of K-Means/Lloyd's heuristic, which can be seen as a Soft-EM model where the variances are restricted to be equal across all dimensions and clusters, and the memberships probabilities are restricted to equal one for one cluster and zero for all others.

To choose a number of clusters, we used the following heuristic: we run the algorithm for 30 clusters, and then we remove all clusters with too few (less than 5) users. Then we run the algorithm again (using the current values instead of randomly intializing) with the remaining number of clusters.

We aggregated the data to clusters with both algorithms, and used our performance evaluator to compare the performance. The results are in Section 5.3.

4.1.2 Personalization

As mentioned earlier, we decided to manage the user-specific words on the server-side, and deploy them with our dictionary updates. We refer to the user-specific words stored at the server as the *server user history dictionary.*

The applications we described above aggregate a user's typed words, along with their occurences, to the words and bigrams dataset. To mix them into the dictionary update we need to scale a word's occurences to a frequency value, as it is used by the keyboard. Figure 4.1a is a histogram of the word occurence distribution for 9 random users from our dataset. As can be seen on the plot, some few words (such as words for "me", "you", "no", "the", etc.) have very high occurences, but most words occur only rarely. For that reason, linear scaling is not a possibility. One option that can be used instead is *logarithmic scaling*:

$$f_{log}(w) = f^{+} \cdot \frac{\log(o(w)+1)}{\max_{w' \in \mathcal{W}} \log(o(w')+1)} \tag{4.1}$$

Where:

$$\begin{aligned} f(w) &= \text{frequency of word } w \text{ for the keyboard dictionary} \\ o(w) &= \text{occurences of word } w \text{ in the user's typed text} \\ \mathcal{W} &= \text{the set of all words typed by the user} \\ f^{+} &= \text{the maximum output frequency (set to 230)} \end{aligned}$$

Figure 4.1b shows the distribution of the frequencies after logarithmic scaling has been applied to their occurence values. It is noticeable that the distribution is somewhat better balanced, but most frequency values are hardly ever used.

Another option is *order scaling.* This can be understood as keeping only the ranking of the words and chosing the frequencies such that they are uniformly distributed in a certain range:

$$f_{order}(w) = f^{-} + (f^{+} - f^{-}) \cdot \frac{i(w, \mathcal{W})}{|\mathcal{W}|} \tag{4.2}$$

Where:

$$\begin{aligned} i(w, \mathcal{W}) &= |\{w' \in \mathcal{W} \mid o(w') < o(w)\}| + 1 \\ f^{-} &= \text{the minimum output frequency (set to 100)} \end{aligned}$$

$i(w, \mathcal{W})$ can be understood as the index of the word w, in the list of $\mathcal{W}$, sorted by word occurences. Figure 4.1c shows the distribution of the frequencies after order scaling has been applied. The histogram shows straight lines because we synthetically distributed the frequencies such that they are uniform.

Logarithmically scaled frequencies are badly distributed, but order scaled frequencies disregard word occurences too much. *Mixed scaling* attempts to combine the best of both worlds, and is essentially the weighted average of both:

$$f_{mixed,\alpha}(w) = \alpha f_{log}(w) + (1-\alpha) f_{order}(w) \tag{4.3}$$

Where:

$$\alpha \in [0,1] = \text{parameter for log versus order frequency preference}$$

Figure 4.1d shows the mixed scaled frequencies, with α set to 0.5. Finally, Figure 4.1e is a combined plot of all scaling algorithms with data from 4 users. Note that the y-axis uses a logarithmic scale to fit all values into one plot.

The final dictionary update the server returns is the user's cluster dictionary, merged with the server user history dictionary. For words occuring in both dictionaries, one has to decide on a frequency value. One option is to use the weighted average:

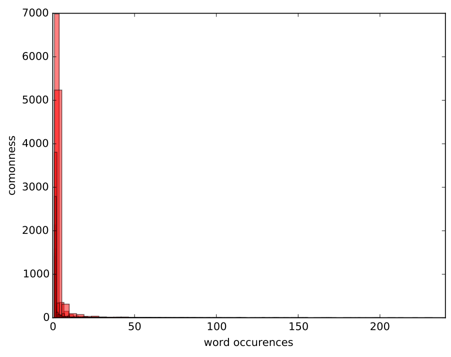
$$m_{avg,\beta}(f_c, f_u) = \beta f_c + (1-\beta) f_u \tag{4.4}$$

Where:

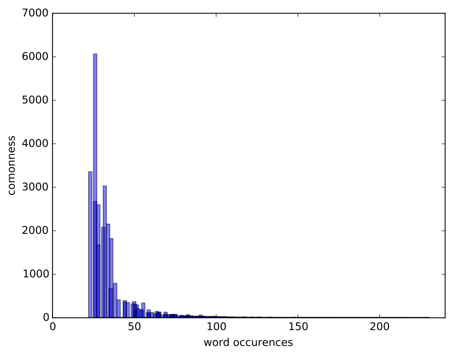
$$\begin{aligned} f_c &= \text{the cluster dictionary frequency} \\ f_u &= \text{the server user history dictionary frequency} \\ m(f_c, f_u) &= \text{merged final frequency for dictionary update} \\ \beta \in [0,1] &= \text{parameter for cluster versus user frequency preference} \end{aligned}$$

Another possibility is using the maximum frequency:

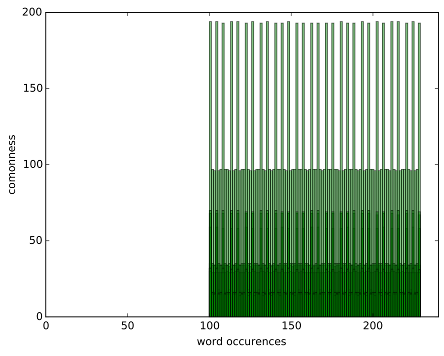
$$m_{max}(f_c, f_u) = \max(f_c, f_u) \tag{4.5}$$



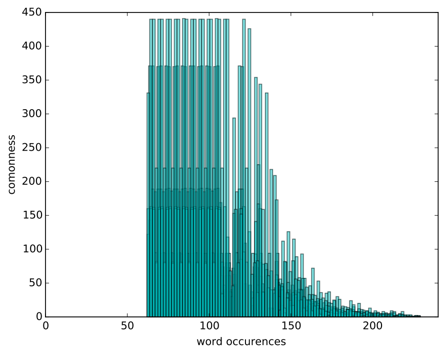
(a) raw occurence values



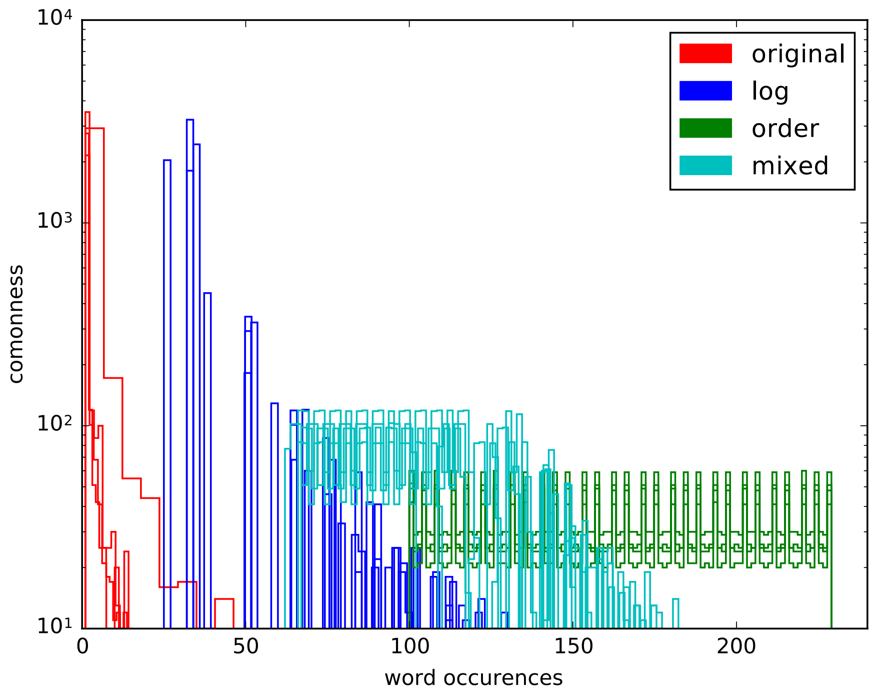
(b) log scale



(c) order scale



(d) mixed scale



(e) all

Figure 4.1: Word occurence value historgrams.

Chapter 5

Results

5.1 Evaluation

To assess how well the different algorithms and parameters work and how high the quality of the suggestions is, we developed the *performance evaluator* for Kännsch. The performance evaluator is essentially an instrumentation test for Android that uses a stubbed-out version of the keyboard, providing only a programatic interface instead of an actual UI.

The performance evaluator receives a list of words as an input, and simulates typing these words with the keyboard character by character. Before each character is typed, the suggestions generated by the keyboard are retrieved. If the currently typed word is equal to one of the top three suggestions, it counts as a match and the rest of the characters of the word count as saved. For example, if the intended word is "Evaluation", the user types "Eval" and the keyboard displays "Evaluate", "Evaluation" and "Evaluated" in the suggestion bar, then that counts as 10 total, 4 typed and 6 saved characters. Note that *predictions* are considered as well. A prediction is a word suggestion based solely on the previous words, e.g. if the evaluation text is "how are you", then after the user has typed the word "how" the keyboard suggests the word "are". After that has been picked, it suggests "you", therefore it counts as 9 total and 6 saved characters.

During the performance evaluation, the total number of characters, the number of saved characters, the total number of words, the number of completed words, and the number of predictions are kept track of. We use the ratio of saved characters to total characters for our primary measure of dictionary quality.

To evaluate how well our cluster and personalization algorithms and settings work, we constructed an evaluation framework using the data from the accumulated application usage logs and the performance evaluator. The first step is to obtain a dataset of typed text. We wrote a program to parse the logs reconstruct a *history* of all words the user has typed, in the correct order.

We picked random users, and used their first 4000 words to compute sample dictionary updates. We picked the number 4000, because that is enough words for the server user history dictionary to memorize an important part of the user's typing style and to estimate their dialect cluster. At the same time it leaves enough new words the user has not yet typed, to test the cluster dictionary's performance.

We evaluated our server-side settings for 20 randomly picked users (with sufficiently long histories). For each user, we simulated said dictionary update on an Android phone. The application data has been wiped befor each evaluation. We used the next 500 words from the user's history as our validation input and ran the performance evaluator to obtain the number of saved characters.

Figure 5.1 shows the amount of distinct words, plotted against the total number of typed words, constructed from the user histories. Only users with over 4500 typed words are considered.

5.2 Language Detection

As an input text for the language detector evaluations, we used the first 1000 words of J. K. Rowling's *Harry Potter and the Sorcerer's Stone*, in English and in German. The keyboard uses the default settings, with all languages activated (Swiss German, Standard German, English, French, Spanish and Italian). Before each evaluation the application data is wiped and no dictionary updates are being performed. Note that the user history dictionary is active, i.e. typed words are suggested throughout the rest of the text.

We evaluated 5 different algorithms. *No-ld* does not use the language detector and simply combines the suggestions of all dictionaries without ranking them in any way. *Kitkat-ld*, *zero-center-ld*, *grouped-ld* and *group-1-ld* are defined as in Section 3.1.2.

Figure 5.2a shows the evaluation results. As expected, the group-1-ld algorithm works best, with over 30% saved characters for the English text. This is a clear improvement over not using the language detector, which yields a ratio of just over 20% saved characters. The dashed lines are the reference values, i.e. the evaluation ran with the same input but all other languages deactivated and thus the best result we can hope for. Note that for the reference values, Swiss German (which overlaps with both English and German and is usually always active) is deactivated as well.

Figure 5.2b shows the part of the words that are completed or entirely predicted by the suggestions, for the English input text. Note that predicted words do not count as completed and are a separate set of words. Also note that the number of suggested words is not as expressive for a measure as the saved character ratio, as it does not take into account what part of the word had to be

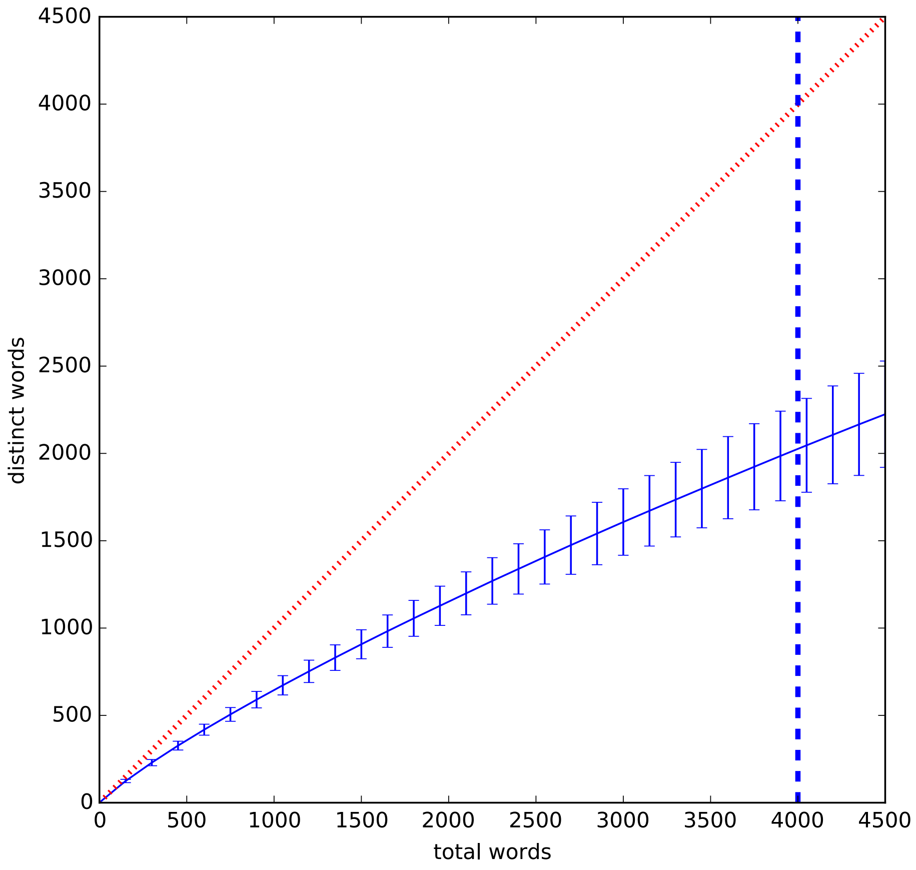
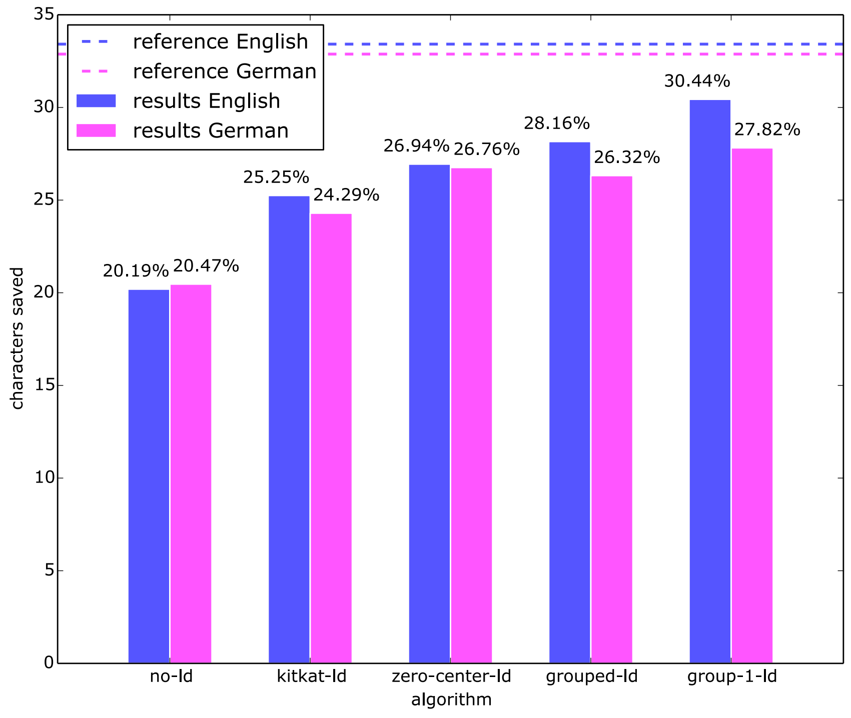
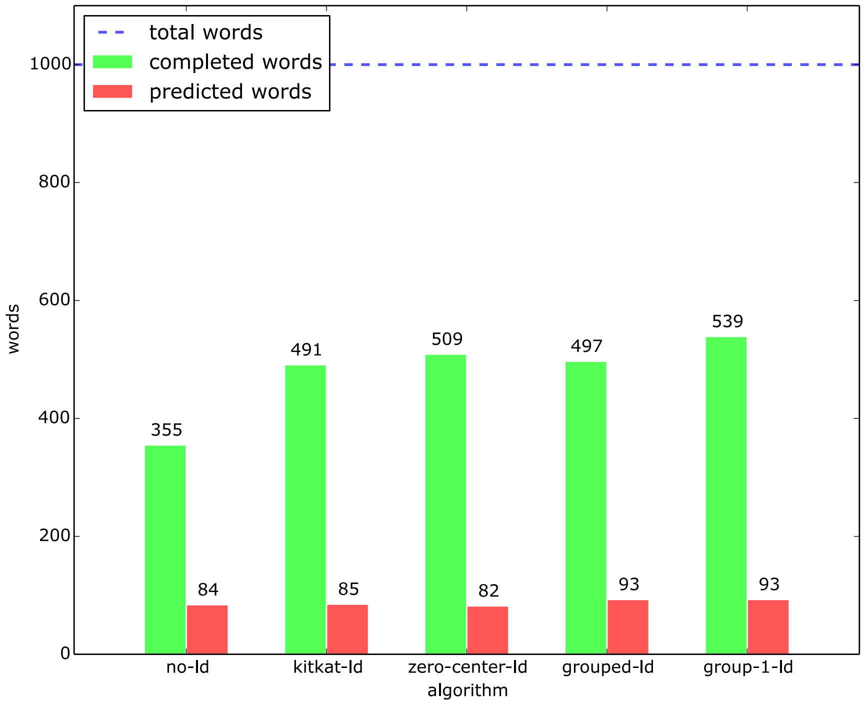


Figure 5.1: Number of distinct word plotted against the total number of words. The dotted line is the reference value, of a user who does not type words a second time. The dashed line is the margin of our dataset split: the left part, i.e. the first 4000 typed words, correspond to the part used to estimate the user's dialect and/or fill the user history dictionary, The right part, i.e. the 500 words after that, is used to analyse the suggestions and evaluate the performance.



(a) Language detector algorithms for German and English input text



(b) Saved, predicted and total words

Figure 5.2: Language detector algorithm performance results. With well tuned settings, about 30% of all characters can be saved.

typed (e.g. words for which only the last character was completed count just like all other completions). Again, the group-1-ld algorithm works best, with more than half of the words found among the suggestions and just under a tenth of the words predicted.

5.3 Cluster Dictionaries

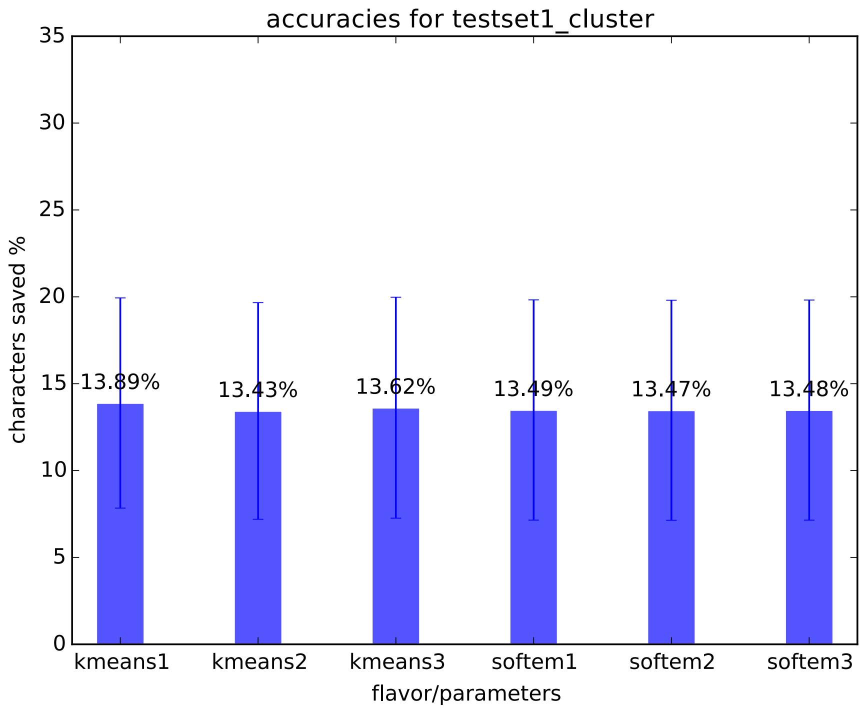
An important thing to consider for both the Soft-EM and the K-Means algorithm is that they improve the solution in each step, but only calculate a local minimum of the cost function, meaning that the final solution may not always be optimal and that it depends a lot on the initialization of the cluster centroids and memberships. For that reason, we ran both the K-Means and the Soft-EM algorithms three times each, to incorporate the effect of different randomly initialized parameters.

Figure 5.3a shows the average rate of saved characters. On average, all initializations and both algorithms perform equally well, with about 13.5% saved characters. Figure 5.3b shows an overlay of for a few notable rates of individual users. Regarding the individual performances, the K-Means algorithm depends a lot more on the initialization and there is higher fluctuation across the users, while the Soft-EM values are virtually identical across the three random initialization parameter sets.

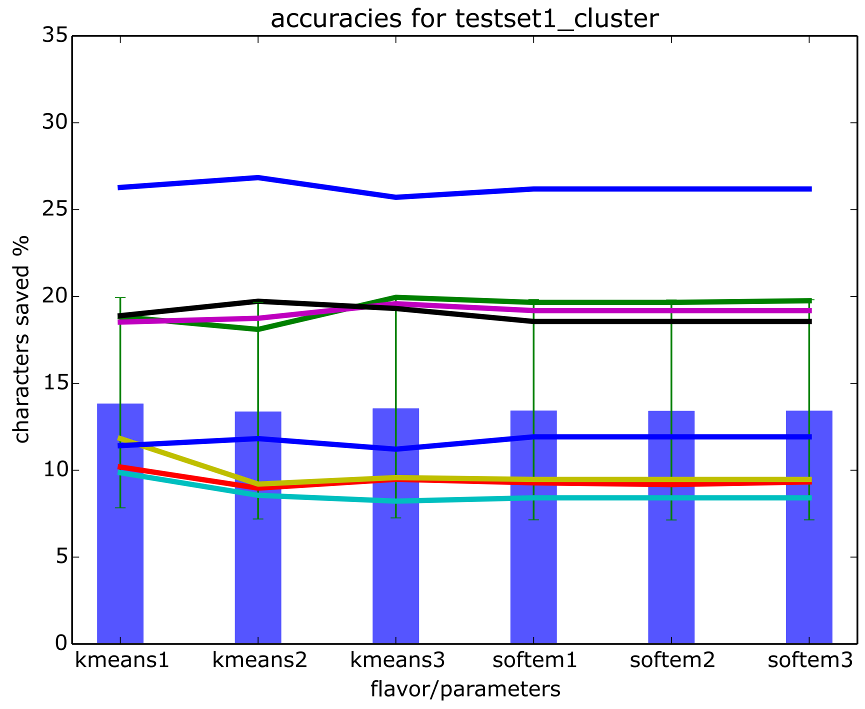
5.4 Server User History Dictionaries

We evaluated five different algorithms for obtaining dictionary frequencies: logarithmic scaling, order scaling, and mixed scales with weight $\alpha \in \{0.25, 0.5, 0.75\}$. As seen on Figure 5.4a, order scaling (on average about 14% saved characters) performs slightly better than logarithmic scaling (about 13% saved characters), but a mixed scale with order scaling preferred yields the best performance (almost 15% saved characters).

Figure 5.4b shows a server user history dictionary (obtained with a mixed scale with $\alpha = 0.75$) merged with a Soft-EM cluster dictionary. For the averaging strategy we chose the values 0.5, 0.25 and 0.75 for β (corresponding to mix, mostly_suhd, and mostly_cluster, respectively). All combinations perform equally well with just above 15% saved characters and none of them is a significant improvement over the others.

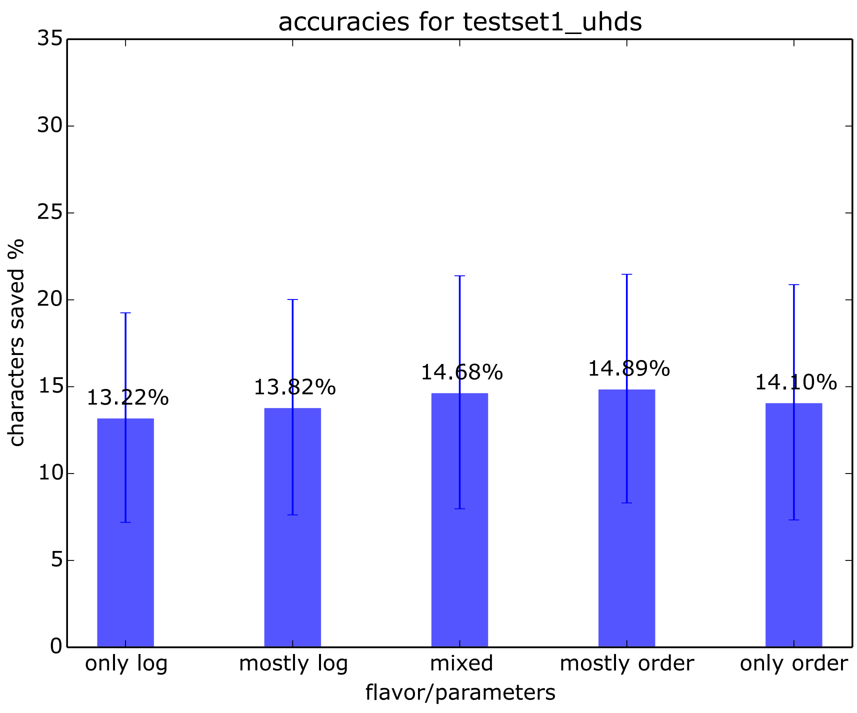


(a) Average performance values

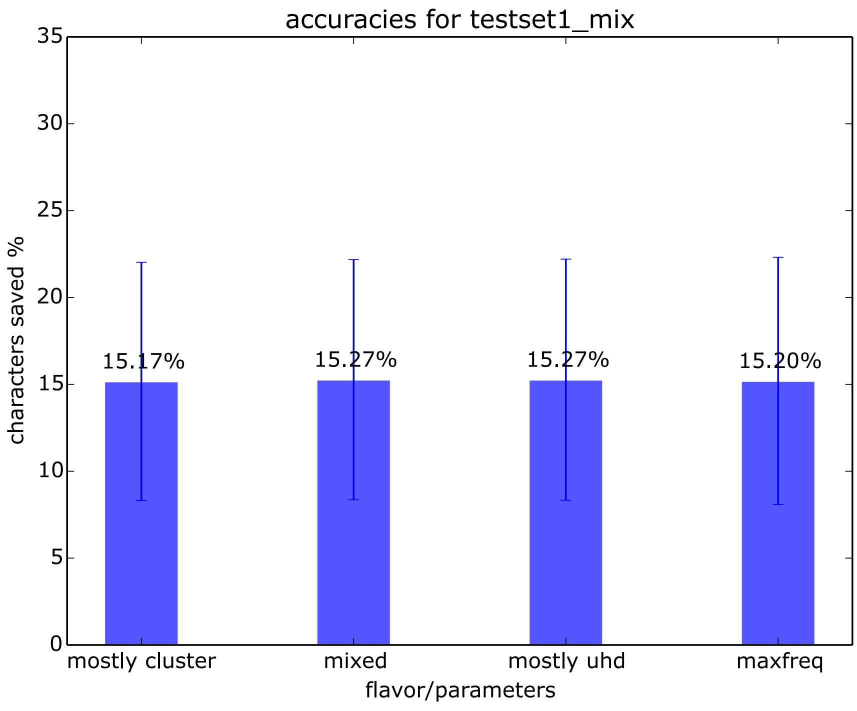


(b) User-individual performance values

Figure 5.3: Cluster dictionary performance values



(a) User history dictionary frequency algorithms



(b) Merging cluster and user history frequencies

Figure 5.4: Cluster dictionary and merged dictionary performances

Chapter 6

Conclusion and Future Work

In this thesis we develop an update to the Kännsch Swiss German mobile keyboard application. It is based on the Android Marshmallow release of LatinIME and includes its features and updates. We improved the language detection scheme to make it more accurate. We also implemented an alternative clustering algorithm for dialect dictionaries and evaluated its performance and added a user-specific, server-side word store to our application.

This project is by no means concluded and there is a lot of potential for future work. The language detector could be improved further to consider only one or two active languages at a time, to have some tolerance for random anglicisms occuring in the input text, or maybe, when certain, to switch immediately to another language without the user having to type five words first. The Swiss German language used in an informal setting inherently overlaps with the Standard German language. For that reason, and because our clustering methods are based on user typed words (and not filtered based on languages), most of our generated Swiss German dictionaries overlap considerably with the other language dictionaries. Our dictionary-based language detection scheme thus prefers Swiss German as usually many German, English or French words can be found in it. Future work could tackle that problem and develop a system to better distinguish these languages. Furthermore, our language detection scheme could be applied on the server-side when analyzing the data, to build purely Swiss German dictionaries and thus not consider other languages as an influence to the user's dialect, therefore improving the dialect analysis precision. Our algorithms for analyzing and reconstructing user-typed words from usage logs need more work as well. One could consider the events for switching of text fields and text selection to accurately reconstruct the user's typing history and to reduce the number of random words, and hence decrease history noise and increase overall dictionary and evaluation accuracy.

The probabilistic nature of the Gaussian mixture model the EM algorithm uses can be used to find an optimal number of clusters: each algorithm output can be analyzed in terms of probability to occur under the current model/cluster parameters. These probabilities, together with a well tuned regularization parameter, could be used to find a good number of dialects to distinguish. A better method to automatically estimate a sensible dialect count could become increasingly important, as our application has lately awaken interest in Austria as well, and there might be more than a total of 30 dialects to distinguish.

Words could be assigned an universal ID and could be matched across dialects. One could try to automatically generate Swiss German dialect translations. Once dialect translations are accurate enough, one could build and optimize a single universal word prediction algorithm and translate it to the correct Swiss German dialect. One could experiment with Recurrent Neural Networks [17], which are great for natural language processing and machine translation. One could build a next-generation word suggestions engine, similar to *SwiftKey Neural* [18].

Bibliography

[1] ai.type: ai.type - home. http://aitype.com/ Accessed: 2016-07-14.

[2] TouchType Ltd.: Swiftkey - smart prediction technology for easier mobile typing. https://swiftkey.com/ Accessed: 2016-07-14.

[3] Swype: Swype - type fast, swype faster. http://www.swype.com/ Accessed: 2016-07-14.

[4] Google: Use gesture typing - nexus help. https://support.google.com/nexus/answer/2811346?hl=en Accessed: 2016-07-14.

[5] Google: Introduction - material design - google design guidelines. https://material.google.com/ Accessed: 2016-07-14.

[6] Google: Google-keyboard. https://play.google.com/store/apps/details?id=com.google.android.inputmethod.latin Accessed: 2016-07-14.

[7] Google: Android open source project. https://source.android.com/ Accessed: 2016-07-14.

[8] Google: platform/packages/inputmethods/latinime - git at google. https://android.googlesource.com/platform/packages/inputmethods/LatinIME/ Accessed: 2016-07-14.

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

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

[11] GO Dev Team: Go keyboard. https://play.google.com/store/apps/details?id=com.jb.gokeyboard Accessed: 2016-07-14.

[12] Kika Keyboard Team: Kika keyboard - emoji, gifs. https://play.google.com/store/apps/details?id=com.qisiemoji.inputmethod Accessed: 2016-07-14.

[13] Unicode, Inc.: Faq - emoji & dingbats. http://www.unicode.org/faq/emoji_dingbats.html Accessed: 2016-07-14.

[14] Android Developers: Userdictionary - android developers. https://developer.android.com/reference/android/provider/UserDictionary.html Accessed: 2016-07-14.

[15] Gaudin, K.: Acra - know your bugs. http://www.acra.ch/ Accessed: 2016-07-13.

[16] Dempster, A. P. and Laird, N. M. and Rubin, D. B.: Maximum likelihood from incomplete data via the em algorithm. In: Journal of the Royal Statistical Society, Series B. (1977)

[17] Andrej Karpathy: The unreasonable effectiveness of recurrent neural networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/ Accessed: 2016-07-21.

[18] TouchType Ltd.: Swiftkey debutes alpha keyboard powered by neural networks. https://blog.swiftkey.com/neural-networks-a-meaningful-leap-for-mobile-typing/ Accessed: 2016-07-24.