8-1-2018

Improving the Accuracy of Mobile Touchscreen QWERTY Keyboards

Amanda Kirk
University of Denver

Improving the Accuracy of Mobile Touchscreen QWERTY Keyboards

A Thesis

Presented to

the Faculty of the Daniel Felix Ritchie School of Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Amanda Kirk

August 2018

Advisor: Dr. Nathan Sturtevant

Author: Amanda Kirk
Title: Improving the Accuracy of Mobile Touchscreen QWERTY Keyboards
Advisor: Dr. Nathan Sturtevant
Degree Date: August 2018

Abstract

In this thesis we explore alternative keyboard layouts in hopes of finding one that increases the accuracy of text input on mobile touchscreen devices. In particular, we investigate if a single swap of 2 keys can significantly improve accuracy on mobile touchscreen QWERTY keyboards. We do so by carefully considering the placement of keys, exploiting a specific vulnerability that occurs within a keyboard layout, namely, that the placement of particular keys next to others may be increasing errors when typing. We simulate the act of typing on a mobile touchscreen QWERTY keyboard, beginning with modeling the typographical errors that can occur when doing so. We then construct a simple autocorrector using Bayesian methods, describing how we can autocorrect user input and evaluate the ability of the keyboard to output the correct text. Then, using our models, we provide methods of testing and define a metric, the WAR rating, which provides us a way of comparing the accuracy of a keyboard layout. After running our tests on all 325 2-key swap layouts against the original QWERTY layout, we show that there exists more than one 2-key swap that increases the accuracy of the current QWERTY layout, and that the best 2-key swap is $i \leftrightarrow t$, increasing accuracy by nearly 0.18 percent.

Acknowledgements

I, firstly, would like to thank my Thesis Advisor, Dr. Nathan Sturtevant, for his patience, flexibility, inspiration, and mentorship throughout this process. To the Computer Science Department staff and faculty at the University of Denver: I appreciate your support, guidance, and for allowing me this opportunity in the first place. My experience in your graduate program was life changing. To my colleagues Andy, Zach, Andrew, Sally, and the rest – I could not have gotten through without your companionship, nor would my experience in graduate school been as exceptional. Thanks for keeping my stress levels down and bringing laughter to an otherwise serious academic environment. To my partner, thank you for all your love and support, especially in moving our family to Denver, allowing me to follow my dreams. Finally, to my family, thank you for always believing in me, sacrificing so much for me, and pushing me to be the best I can be.

Table of Contents

List of Tables

List of Figures

Chapter 1

Problem Statement

1.1 Introduction

With computing device usage growing rapidly throughout the world and already prevalent in most of it [27], it makes sense that much research has gone into improving human-computer interaction [38]. There are now various input methods available that enhance and enable our interactions with computers, such as speech recognition [25], muscle movement detection [2], character recognition [16], and camera-based sensors [50]. Though to date, the most widely used and central component to human-computer interaction is the keyboard, with text-entry being the main way we accomplish work through and communicate with computational devices. While the QWERTY keyboard layout [64], shown in Figure 1.1, is known as a well-suited configuration for desktop systems, the same has not been shown to be true for mobile devices, although being widely adopted by them. QWERTY keyboards pose a definite problem for text-entry in mobile computing [34]. Where QWERTY keyboards may have been ergonomic and efficient for desktop users, the large keyboard configuration doesn't fare well on such small devices as the ones we are growing accustomed to using today. The advantages of QWERTY are also compromised when considering the fact that users tend to type on mobile devices using a single hand, whereas QWERTY was intended for two-handed typing [22]. Despite the concerns surrounding the usage of QWERTY keyboards on mobile devices, QWERTY continues to dominate in the mobile realm because of familiarity, preexisting hardware and software, and training available.

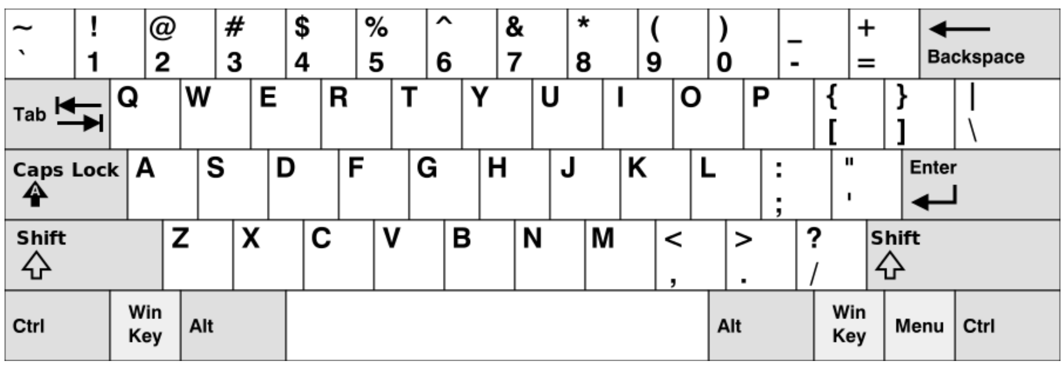

Figure 1.1: Example of QWERTY Keyboard Layout

According to the Pew Research Center [13], more than 75 percent of Americans now have smart phones, which typically employ virtual, touchscreen technology. Nearly all of these virtual touchscreen keyboards utilize the QWERTY layout. Most mobile device users are aware of the typing errors, or typos, that come with typing on a mobile touchscreen keyboard and have experienced the headache that comes with having to correct errors frequently or, in extreme cases, when mistakenly sending an invalidly typed message over the airwaves. Much time and energy is spent correcting these errors [66][48] and is often a main pain point for mobile users. These typos can only have been increased as a result of squeezing the large QWERTY layout onto small mobile devices, making the infamous fat finger problem [52] more prevalent than ever.

1.2 Approach

In our research, we explore alternative keyboard layouts in hopes of finding one which results in increasing the accuracy of text input on mobile devices. We do so by carefully considering the placement of keys, exploring a specific vulnerability that occurs within a keyboard layout. Namely, that the position of specific keys close to certain other keys may increase typos. In particular, it is well known that the most common type of error when typing is the substitution of an adjacent letter for the intended letter [15], e.g., typing *sprlling* on the QWERTY keyboard when you intended to type *spelling*. In this case, typing an *r*, which is adjacent to *e*, instead of the intended *e*. We call any key that is adjacent to a target key, a **neighbor** of the target key. For example, the neighbors of the key *q* are *w*, *a*, and *s* (Figure 1.2).

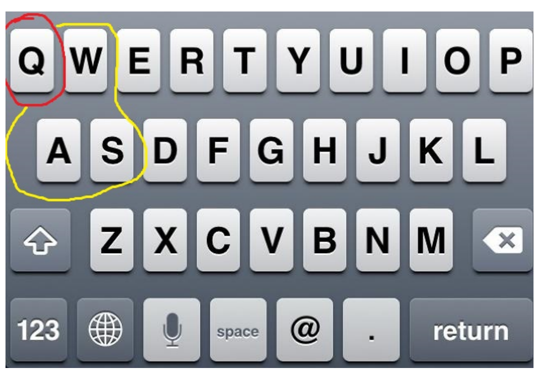

Figure 1.2: Neighbors of Q

When a typo results in a non-dictionary word, such as *sprlling*, it is quite easy for even a simple autocorrector to detect this as an error and attempt to correct it. For errors that result in a well-spelled word, such as attempting to type *dog* but typing *dig* instead, autocorrectors typically will not try to correct the word, as it may not determine a word that is already well-spelled and within reasonable context to be erroneous. This problem has been improved upon by utilizing bi-grams and Hidden Markov Models [4], but remains one of the major issues facing autocorrection performance today. It is easy to see, then, that a keyboard layout influences mistypings, substitution errors in particular. Consider if i and u were not adjacent on the keyboard we were typing on. Say we swapped z with i on the QWERTY layout so that u and i were no longer neighbors. The aforementioned typo would have resulted in the string *dzg*, rather than *dig* and the autocorrector would sprint into action to assist the user, since *dzg* is not well-spelled. As it is much more likely to press a neighbor of the intended key than a non-neighbor, we should question the arrangement of keys on the keyboard layout we are using.

Consider the fact that, on the QWERTY keyboard, the keys i and o are next to one another. One can quickly see that there are many well-spelled words that result from this type of substitution error. Many words, when swapping out an i for an o or an o for an i, will still be well-spelled, resulting in the keyboard not being able to recover from a great many of errors. Examples of such substitutions include: *does* $\iff$ *dies*, *dog* $\iff$ *dig*, *moss* $\iff$ *miss*, *fond* $\iff$ *find*, *hot* $\iff$ *hit*, *pot* $\iff$ *pit*, *son* $\iff$ *sin*, *rode* $\iff$ *ride*, *lock* $\iff$ *lick*, *lost* $\iff$ *list*. Thus, one can conclude that the keyboard layout itself, or having i and o next to one another, in particular, decreases the accuracy of the keyboard due to dampening the effectiveness of the autocorrector.

This "Problem With Neighbors" is amplified further on MT keyboards as the keys are even closer to one another than on traditional keyboards. There are many examples of these types of problematic key placements on the QWERTY keyboard. Our intuition is that separating certain keys, such as vowels, making them non-neighbor, would result in better autocorrect performance, and thus, higher keyboard accuracy overall. Our research aims to answer the question: *Can we significantly improve the accuracy of a mobile touchscreen (MT) QWERTY keyboard by swapping just two keys?*

We will investigate alternative layouts which involve a single swap of two keys (2-key swap) on the original QWERTY layout, aiming to separate those keys that, when close together, would have increased the probability of error. We test all 325 2-key swaps, comparing their accuracy to that of the original QWERTY layout. Our aim is to aide in improving the text-entry problem, increasing accuracy with a very small change to the ever-pervasive QWERTY layout. Given that QWERTY is likely here to stay, a single 2-key swap would be a small, doable change, with a large payoff. The familiarity of QWERTY would not be jeopardized, as navigating a small change to QWERTY would be an easy adaptation. The preexisting software and hardware would also not be disrupted, but by a small degree.

Our approach involves modeling the keyboard, constructing an error model and an autocorrector so that we can simulate typing on the keyboard to measure it's accuracy. Then, using our models, we measure accuracy of the original QWERTY layout as a baseline. We then measure the accuracy of all 2-key swap layouts, analyzing which layouts show better performance than

QWERTY. Our results show that there exist more than one 2-key swaps that would increase the accuracy of the current QWERTY layout, and we provide the best 2-key swap derived from the QWERTY layout, which is shown to increase accuracy by nearly 0.18 percent.

1.3 Thesis Overview

The remainder of this thesis is organized as follows: In Chapter 2, we provide the audience with historical context on QWERTY and the text-entry problem. We will explore related work and the state of the art on improving the accuracy of MT QWERTY keyboards. In Chapter 3, we begin to construct our error model, simulating the error that happens when typing on a QWERTY keyboard. In Chapter 4, we describe how we accomplish constructing a simple autocorrector for our keyboard model. In Chapter 5, we discuss our methods of using the total keyboard model in conjunction with a large corpus of test strings to measure accuracy of a keyboard layout. We then test all 325 2-key swaps, comparing each layout's accuracy to the original QWERTY configuration, and present our results. In Chapter 6, we summarize our contributions and provide suggestions for future work.

Chapter 2

Background

In this chapter, we provide the audience with historical context on QWERTY and the text-entry problem. We will also explore related work and the state of the art on improving the accuracy of MT QWERTY keyboards.

2.1 History

Today's modern keyboards are, not surprisingly, direct descendants of the typewriter. Though two lesser mentioned technologies that had a direct hand in deriving today's keyboards were the teleprinter, which was used to type and transmit stock market text data from keyboard to stock ticker machines, and the keypuncher, which was used as an early data entry device, punching holes into paper to record and verify data. As typewriters, keypunches, and teleprinters became more electromechanical, they began to employ what we know today as keyboards. The development of the earliest computers integrated electric typewriter keyboards. This included the ENIAC [41] computer, which used a keypunch device as both the input and output device, and the BINAC [56] computer, which also made use of an electromechanically controlled typewriter for both data entry and data output.

The QWERTY layout was devised and created in the early 1870s by Christopher Latham Sholes, with the layout becoming popular with the success of the Remington No.2 typewriter of 1878. The first computer terminals, such as the Teletype [44], were typewriters that could produce and be controlled by various computer codes. These early computer typewriters used the QWERTY layout. Due to the large amount of familiarity, preexisting hardware, and training available, the QWERTY layout translated seamlessly to the computational sector, with nearly all electronic computer keyboards utilizing the QWERTY layout.

While keyboards remain the central input device for computational devices, there have been many other input methods introduced to enhance human-computer interaction. One of the most commonly used input devices, second to the keyboard, is the pointing device [54]. Most often times, a pointing device takes the form of a mouse, but other examples include touchscreen, the pointing stick, and the joystick. Pointing devices allow the user to input spatial data to the computer, which is often utilized to create a simple and intuitive way to navigate a computer's graphical user interface (GUI). Other common inputs include audio [11] (e.g. microphone, etc.) and video input [19] (e.g. digital camera, scanner, etc.). More advanced forms of input have also been introduced. Optical character recognition [16] allows the conversion of text from an image to editable text on the computer. Speech recognition [25] translates audible vocalizations to machine-editable text on the computer.

Muscle-movement and camera-based sensors [50] allow gestures to be translated to the computer. A popular example of this input type is the Microsoft Kinect [67], which enables the user to control and interact with their gaming console or computer through a user interface using physical gestures.

With the introduction of smart mobile devices came the rise of software, or virtual, keyboards [43], which often take the form of computer programs that display an image of a keyboard on the screen. By the click of a pointing device or tapping of the finger on a specific letter from the virtual keyboard, software writes the respective letter on the respective spot. Virtual keyboards are commonly used as an on-screen input method in devices with no physical keyboard such as a pocket computer, personal digital assistant (PDA), tablet computer or touchscreen-equipped mobile phone. Software keyboards have become very popular on touchscreen enabled mobile devices due to the cost and space requirements of hardware keyboards. QWERTY has been widely adopted by both mobile devices with hard keyboards as well as those with virtual touchscreen keyboards.

2.2 Related Work

The problem of text-input accuracy has been around since the time of typewriters, and was one of the reasons why the QWERTY layout was developed in the first place. Since then, much research has gone into increasing accuracy on text-input mechanisms, such as the keyboard [55]. With human-computer interaction being more prominent than ever, bettering the text-input methods we currently employ would have drastic effects in overall productivity, both for persons and businesses alike. We now use computational devices for numerous tasks, both in our daily lives and in our workplaces, allowing us to automate or speed up much of what we do. With text-entry being a main way we accomplish work and boost productivity and users now turning to their mobile devices to accomplish many of these tasks, increasing the accuracy of one of our most commonly used methods of text-entry, the mobile touchscreen keyboard, is a meaningful path of research to pursue.

There are various directions that research has gone in in order to tackle the text-input problem on mobile devices [39]. Text entry evaluation focuses on two quantities: speed and accuracy. While much work has been done on speed [30], given its relative ease to evaluate and measure, accuracy is much more complex to model and measure. One method of improving the accuracy of text-input to mobile devices has been to investigate alternative keyboard layouts entirely [60], moving away from QWERTY. Another research direction has been to consider new input methods for text-entry altogether [12], rather than typing as we traditionally do. Thirdly, researchers have proposed improving accuracy via software that utilizes predictive techniques, probabilistic methods, and/or machine-learning to decrease error by correcting and/or preventing typos [39].

2.2.1 Alternative Keyboard Layouts

Several alternative keyboard layouts have been developed over the years, claimed by their designers and users to be more efficient, intuitive, ergonomic, and accurate than QWERTY. The most widely used alternative is the Dvorak Simplified Keyboard [46] (Figure 2.1). This layout places the most commonly used letters in the home row, where they're easy to reach, and the least commonly used letters on the bottom row, where they're hardest to reach. While QWERTY results in most of the typing being performed with the left hand, Dvorak results in most typing being performed with the right hand. Another increasingly popular alternative is Colemak [17] (Figure 2.2). Colemak is more similar to the QWERTY layout, so its easier to switch to from a standard QWERTY keyboard, with only 17 changes made from the QWERTY layout. Like Dvorak, it's designed so the home row of keys is used more frequently. It also claims to reduce how far your fingers need to move while typing. Another, more extreme, example of an alternative layout is Maltron [37] (Figure 2.3). Rather than a single rectangular grouping of letter-based keys, Maltron includes two square sets of letters, both of which flank a number pad in the middle.

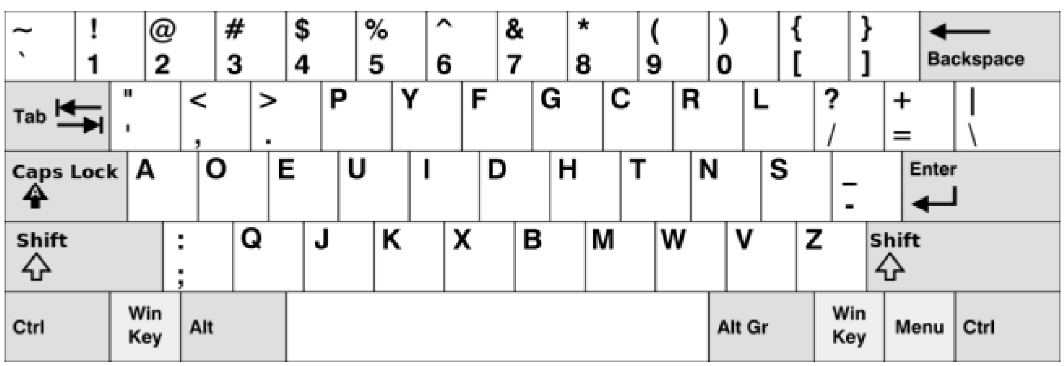

Figure 2.1: Example of Dvorak Simplified Keyboard Layout

The advantages and disadvantages of these alternative layouts have long been debated [61], some arguing that they perform better than the QWERTY configuration. Nonetheless, these alternatives have not yet seen widespread adoption.

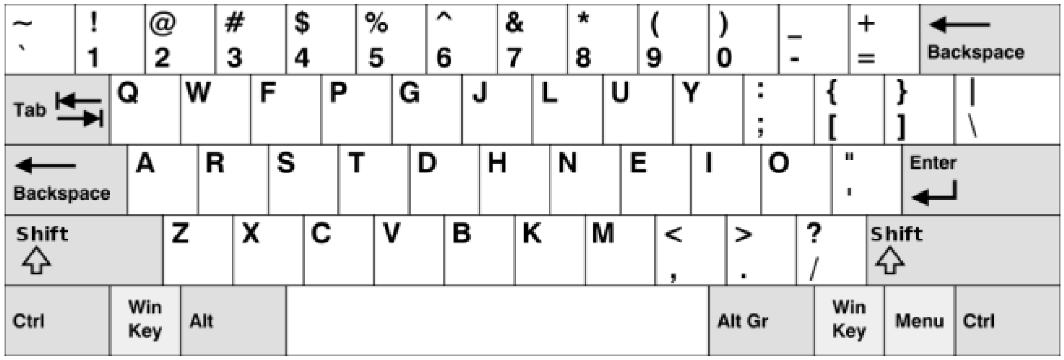

Figure 2.2: Example of Colemak Keyboard Layout

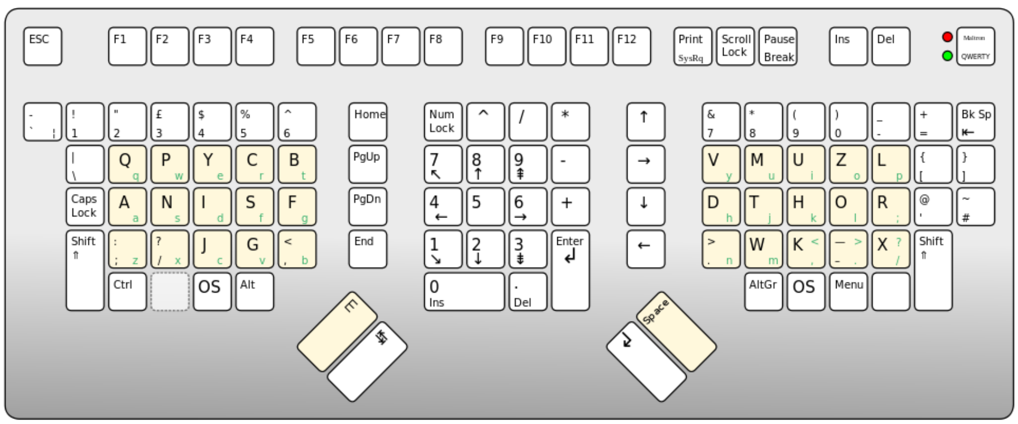

Figure 2.3: Example of Maltron Keyboard Layout

2.2.2 Alternative Input Methods

Another direction that research has gone in an effort to improve the accuracy of MT keyboards is to investigate alternative touch screen input, altogether, moving away from traditional typing. Below, we share some popular examples of alternative input methods for MT keyboards.

- **8Pen** [1] (Figure 2.4): The 8Pen Keyboard features a small wheel which is used by dragging your finger around the different segments to select each letter. The layout claims to allow for the most common letter sequences to be produced with swift, intuitive, and fluid gestures, which help eliminate typos.

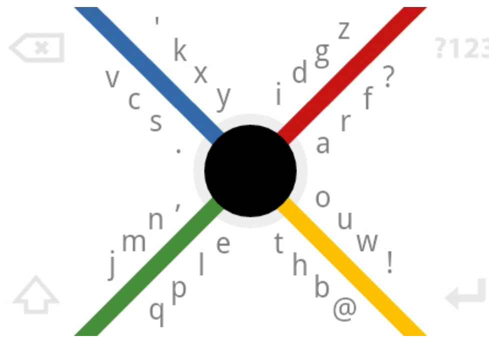

Figure 2.4: Example of 8Pen Keyboard

- **MessageEase** [45] (Figure 2.5): The MessageEase keyboard has a tap and swipe design, where the user taps common letters and swipes outward for lesser used letters. MessageEase claims to use Fitts' Law, letter frequency tables, and bi-gram frequency data to minimize the distance a finger travels while texting. Also, with fewer keys compared to a regular keyboard, MessagEase keys are 3.5 times larger, diminishing the fat finger problem.

- **Thumbly** [59] (Figure 2.6): The Thumbly keyboard is designed to be used with one hand, or more specifically, with a single thumb. Thumbly claims that it's keys are ergonomically arranged and are easily reached with one thumb using a comfortable side-to-side motion, also allowing for automatic detection of switching hands during use.

- **SwiftKey** [58] (Figure 2.7): The SwiftKey keyboard uses a gesture-based input method known as SwiftKey Flow, where the user swipes or drags their finger across all letters they want to by typed. The designers claim

Figure 2.5: Example of MessageEase Keyboard

Figure 2.6: Example of Thumbly Keyboard

that the swipe method is much better for thumb typing and predictive text capabilities.

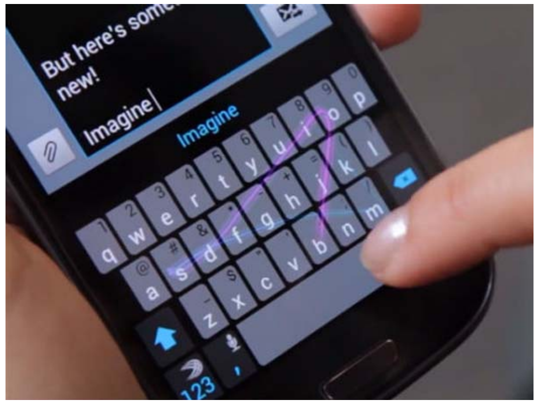

Figure 2.7: Example of SwiftKey Keyboard

While the above keyboards have been well researched and implemented, none of these alternative input methods have seen widespread adoption. Alternative inputs, such as gesture-based keyboard input, face many challenges of their own [53], making the question of if they are any better than QWERTY even more difficult to answer and further minimizing their chance for mainstream adoption.

2.2.3 Software Solutions

The third and most diverse area of research attempting to improve the accuracy of MT QWERTY keyboards has been to create smart software which can reduce errors while typing. There has been much innovation and progress with this approach [31], as scientists aim to make smart phones even smarter via mechanisms such as text prediction and autocorrection. The problem of reducing MT QWERTY keyboard input error with software can be approached form various directions, as error while typing can occur for many reasons including the fat finger problem [52], where a user's fingertip touches a larger area than intended, or when the touch location on the screen is different than the user intended altogether [26]. The user could also not know the correct spelling to begin with, which would result in a typo needing to be corrected. Also, with mobile devices, error can be introduced due to the movement of the person typing, as users are now often walking, jogging, or altering their body positions while doing so.

With each of these problems comes an array of work in the literature that attempts to make the chance of error due to them less likely. For the touch offset problems, research has gone into reducing the fat finger issue and making the target key more likely to be registered than an accidental key. Similar methods have also been devised to improve pressure input issues. Goodman et al. [21] used a language model and a key press model to select the most probable key sequence of the user, rather than the sequence dictated by strict key boundaries. This led to a significant overall error rate reduction. Kristensson and Zhai [32] proposed a geometric pattern matching technique to be used either as an enhanced spell checker or as a way to enable users to escape the Fitts' law constraint in stylus typing. Henze et al. [23] collected millions of data points from touchscreen data and modeled the offsets using polynomials to improve typing accuracy. Weir et al. [62] presented a machine learning approach for learning user-specific touch input models to increase touch accuracy on mobile devices. The model was based on flexible, non-parametric Gaussian Process regression and was learned using recorded touch inputs, showing the importance of user specificity in offset models. Bi et al. [6] derived an expansion of Fitts law for finger touch input, conducting three experiments in 1D target acquisition, 2D target acquisition, and touchscreen keyboard typing tasks. These experiments showed that the derived law was more accurate than Fitts law in modeling finger input on touchscreens. Later, Bi and Zhai [7] conceptualized finger touch input as an uncertain process and derived a statistical target selection criterion called Bayesian Touch Criterion from combining the basic Bayes rule of probability with the generalized dual Gaussian distribution hypothesis of finger touch, which showed to be significantly more accurate than the commonly used Visual Boundary Criterion. Rudchenko et al. [49] studied key-target re-sizing, which dynamically adjusts the underlying target areas of the keys based on their probabilities, which showed to significantly reduce errors. Goel et al. [20] used accelerometer data to decrease errors due to walking while typing, constructing a classification model using the displacement and acceleration of the device and inference about the user's footsteps. Hoffmann et al. [24] developed the TypeRight keyboard, which prevents errors by increasing the resistance of keys for words that are not well-spelled. Before each keystroke, the resistance of keys that would lead to a typing error according to dictionary and grammar rules is increased momentarily to make them harder to press, thus avoiding typing errors rather than indicating them after the fact. Stewart et al. [57] conducted a series of user studies to understand the fundamental characteristics of pressure in user interfaces for mobile devices, providing insight to clarify a longstanding discussion on mapping functions for pressure input by looking at how holding a device influences target acquisition time.

Another software-oriented approach for increasing accuracy on MT QWERTY keyboards is the utilization of autocorrection and text prediction. These approaches largely depend on language modeling and statistical properties of text [55] to determine the most probable sequence of keys that the user intended to type in order to prevent or correct any typos the user may have made. The study of natural language processing [29] has contributed greatly to modeling language, which is pivotal to many autocorrection and text prediction methods. The study of typographical errors [35] has also played a major role in the effectiveness of autocorrection and text prediction, allowing for precise error models to be utilized. Kukich [33] surveyed documented findings on spelling error patterns, providing descriptions of various non-word detection and isolated-word error correction techniques, and reviewed the state of the art of context-dependent word correction techniques. Levenshtein [36] orginally approached error correction in data entry as a combinatorics and coding theory problem, investigating the construction of optimal codes capable of correcting deletions, insertions, and reversals. Shannon [51] did fundamental work on the well-known *noisy channel*, a model which has been successfully applied to a wide range of problems, including spelling correction. Brill and Moore [8] derived an improvement to noisy channel spelling correction via a more powerful model of spelling errors by learning generic string to string edits, along with the probabilities of each of these edits, resulting in significant improvements in accuracy. Brown et al. [10] addressed the problem of predicting a word from previous words in a sample of text, using n-gram models based on classes of words, which formed the foundation of many context driven methods of spelling correction used today. Mayes et al. [40] created a real world spelling corrector which considered the context of the words being typed, using a tri-gram-based noisy channel model to correct errors that would otherwise be skipped by traditional autocorrectors. Yin et el. [65] proposed a new approach for improving text entry accuracy on touchscreen keyboards by adapting the underlying spatial model to factors such as input hand postures, individuals, and target key positions. Whitelaw et al. [63] devised a way to use the web for language independent spell checking, requiring no annotated data-set, as most systems using statistical models do, modeling the web as a large noisy corpus, instead.

Chapter 3

Error Model

In this chapter, we construct our error model, simulating the errors that occur when typing on an MT QWERTY keyboard.

3.1 Motivation

Our goal is to measure the accuracy of the QWERTY layout against variants of the QWERTY layout on MT keyboards. We first need a complete model of the keyboard itself, which includes simulating the action of typing on the keyboard. The first major step towards measuring the accuracy of a particular keyboard layout is to quantify the likelihood that words typed by the user are correctly output to the device screen so that if a user intends to type a word and attempts to type it, that intended word is output to the screen correctly. In other words, given a corpus and a particular keyboard layout, we need to measure the probability that each word in the corpus is accurately output to the screen, matching the intended text, when a user attempts to type each word in the corpus. There are two main components to modeling the accuracy of an MT keyboard layout. First, something that could harm the accuracy of the keyboard is *error*. Thus, an **error model** needs to be constructed to demonstrate the errors that can occur when a user attempts to type a word. Second, errors can sometimes be reversed by an *automatic spelling corrector*, or **autocorrector**, which are nearly guaranteed on modern mobile devices. A model of the autocorrector is therefore needed, as it can reduce errors and increase accuracy. Both of these components contribute to whether the correct word is output to the device screen and, as such, we must construct a model for each in order to measure the total accuracy of MT keyboard layouts. We construct both models generally, so that they can be applied to any keyboard layout. This will allow us to easily compare the accuracy of different layouts against one another. In this chapter, we start by constructing an error model for MT keyboards.

3.2 Noisy Channel Model

The noisy channel model [51] is a framework used for problems such as speech recognition [5], machine translation [9], spell checkers [8], and general language models [10]. In this section, we show how we can model error under the noisy channel framework.

When modeling the scenario of typing on an MT keyboard it can be useful to think of the process as an input string being sent over a noisy channel so that the output string may either be correct or incorrect due to the various mishaps, or noise, that can occur during its travel over the channel. When viewing the

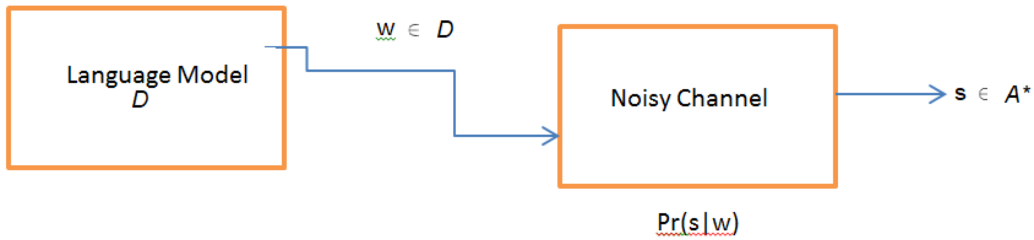

Figure 3.1: Noisy Channel Error Model

act of typing on an MT keyboard within a noisy channel framework, the input string is represented by the word the user attempts to type, and the output string represents the word output to the device screen. The noise in our model represents the typographical errors that could have occurred and effected the correctness of the word being passed over the channel.

To translate the problem of modeling error as a noisy channel, we first formally describe the scenario. Given an alphabet of letter characters A, let A^* be the set of finite strings over A. For our purposes, $A = \{a, \cdots, z\}$, the English alphabet. Let D be the dictionary of legal English words such that $D \subseteq A^*$. Suppose a string $s \in A^*$ is output to the device screen when the word w was intended. We have two possibilities: Either s is correctly output to the screen, matching the intended word ($s = w$) or some noise has occurred causing a different and incorrect word to be output to the screen ($s \neq w$). Figure 3.1 shows an illustration of this noisy channel model example. Given a word $w \in D$, which a user intends to type, the word travels over the noisy channel, representing errors that can occur, and a word $s \in A^*$ is output to the device screen. Here, s may or may not equal w. You can also see that s is output to the screen with the probability $P(s|w)$.

There are two main components to the noisy channel framework when applying it to our problem: the language model and the noisy channel model. A user attempts to type the word w which comes from the language model D, in this case, the English dictionary of words. The word w then goes through the noisy channel with word s finally output to the screen. If $s \neq w$, then s has been erroneously output to the screen according to the probability distribution $P(s|w)$, the probability that s is output to the screen given the user intended w. This probability should represent real-world noise and accurate presumptions about typographical errors. For example, $P(the|the)$ should be very high and $P(tge|the)$ should be relatively high, while $P(loveliness|the)$ should be very low. In an effort to model error, we are clearly more interested in the case where $s \neq w$, as this is when some sort of noise, or error, has occurred.

In order to fully quantify the uncertainty involved with typing on an MT keyboard, we need to build a model which describes all of the possible output we can expect to get during the act of typing. Very simple error models using the noisy channel framework have been used [40], as well as improved upon or made more complex by using, for example, context-based prediction [14]. We construct an error model building off of basic principles which enumerates the various outcomes that can occur when a user types a single character on an MT keyboard. We then extend this model to whole words. The next sections provide details of the error model we have defined to simulate the noisy channel view of typing on an MT keyboard.

3.3 Modeling Typographical Errors

Typographical errors, commonly referred to as typos, are mistakes which occur when a user is typing on the keyboard. Many factors contribute to the occurrence of typographical errors such as mechanical failure, the slip of a finger, or the lack of vocabulary knowledge on behalf of the user. Of course, typos can also be a result of many other factors, particularly on MT keyboards, including keyboard size, key arrangement, movement while typing, whether two handed typing or one handed typing is used, and the infamous fat finger problem [52]. The most common typographical errors that appear while typing a word include adding duplicate or additional letters (insertion), mistakenly typing letters different from those intended (substitution), or missing letters entirely (deletion). Much research has been done in the realm of studying these errors, what causes them, and how to quantify them [55] [35] [42] [18].

Working towards a character error model, we consider four outcomes that can occur when a user types a single character on an MT keyboard. For each of these outcomes, we give an example using a single character and also an example using that single character outcome within a word. The outcomes we will consider are:

- **Outcome 1 : Correct** - The user intends to hit a specific key, and hits the intended key correctly.
  - Example: $h \rightarrow h$ or $hello \rightarrow hello$

- **Outcome 2 : Substitution** - The user intends to type a specific key, but hits a different key by mistake.

  - Example: $e \rightarrow x$ or $hello \rightarrow hxllo$

- **Outcome 3 : Insertion** - The user intends to a hit a specific key, but mistakenly inserts an additional key before or after the intended key.

  - Example: $l \rightarrow lx$ or $hello \rightarrow helxlo$

- **Outcome 4 : Deletion** - The user intends to hit a specific key, but mistakenly misses hitting any key whatsoever.

  - Example: $e \rightarrow null$ or $hello \rightarrow hllo$

These outcomes are traditionally referred to in terms of edit operations [36], which will be explained and used in our autocorrector model (Chapter 4). First, to finish the error model, we need a probability distribution for the outcomes described above. Since error models under the noisy channel framework have been shown to be significantly improved by associating specific probabilities to the individual outcomes that are possible [14], we explain how we distribute probabilities over all of these different scenarios in the following sections.

3.4 Weighted Outcomes

We know that the probability of the four outcomes outlined in the previous section will vary (i.e., each outcome should have its own probability of occurring).

For example, from experience, we know that actually hitting the key we intend is much more probable than making an error, or in other words, we type correctly more often than typing incorrectly. We also know that if we make an error, it is most likely that of the substitution type since hitting the wrong key is the most frequent error over all error types considered [35]. Thus, we need to decide on what probability value to assign to each of our outcomes.

An important distinction comes into play when considering distance between keys in relation to substitution errors. As described in section 1.2, when discussing the "Problem with Neighbors", hitting a key that is right next to the one you are intending to press is more likely than hitting a key that is farther away on the keyboard. This fact is also evident on traditional keyboards (used with desktops or laptops), but is made worse on MT keyboards by the size constraints of the mobile device which force the keys to be smaller and much closer to one another than they are on the larger, traditional keyboards, making the boundaries of keys more difficult to feel. For example, when we attempt to type *the* on the QWERTY layout, it is more probable that we make a substitution error which results in the word *rhe* than making a substitution error that results in the word *zhe*. This is because r is closer to t on the keyboard than z is. This tells us that the probability of a substitution error should not be distributed evenly over all of the keys on the keyboard, rather, each key has a probability of being substituted relative to the distance between itself and the intended key. The closer a key is to the intended key, the higher probability it should have of being substituted by mistake. In our error model, we use the distance between keys to impact both the weight of a substitution error occurring relative to other error types and we also weigh certain keys as more likely to be hit on accident than others depending on their distance to the intended key.

3.5 Neighborhood Definition

Before defining the weighted probability distribution for the different outcomes that can occur when typing, we must integrate the distance between two keys as a factor into our probability of substitution errors since, as revealed in the previous section, it is not uniformly distributed over all keys. To do this, we distinguish two important types of substitution errors: **neighbor** substitution and **non-neighbor** substitution. We define a *neighbor* of the key δ as any key β, $\beta \neq \delta$ such that $distance(\delta, \beta) = 1$ and a *non-neighbor* of δ to be any key that is more than distance one away from δ on the keyboard or $distance(\delta, \beta) > 1$. Each key on a keyboard layout then has a static set of neighbors and a static set of non-neighbors. Let A be our alphabet of character keys on the keyboard, i.e., $A = \{a, b, c, \cdots, y, z\}$. We denote N_δ as the set of neighbors for key δ so that the set of non-neighbors for δ, NON_δ, is the set difference $A - \{N_\delta, \delta\}$. We define the *distance* function using regular euclidean distance, by assigning each key on the keyboard a coordinate location, as shown in Figure 3.2. We represent the keyboard as a grid of keys in a 2-dimensional array, where $grid[i][j]$ represents the key with coordinate location (i, j).

_	1	2	3	4	5	6	7	8	9	10
1	Q	W	E	R	T	Y	U	I	O	P
2	A	S	D	F	G	H	J	K	L	
3	Z	X	C	V	B	N	M			

Figure 3.2: Defined Grid of Keys for QWERTY Layout

Using Euclidean distance, the distance between two keys δ and β is then defined to be

$$distance(\delta, \beta) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2},$$

where the location of δ in the keyboard grid is (x_1, y_1) and the location of β on the keyboard grid is (x_2, y_2). In order to avoid vertical/horizontal bias, we additionally define any two keys that are both distance 1 away horizontally and distance 1 away vertically to be a total distance 1 away from each other. That is, keys that are directly diagonal to a given key are considered neighbors of that key. Thus, adding to the definition above, if $|x_1 - x_2| = 1$ and $|y_1 - y_2| = 1$ we set $distance(\delta, \beta) = 1$.

With our grid locations for keys in place for a particular layout, it is easy to compute the distance between keys and further, the neighbors of a particular key. For example, the key q on the QWERTY layout has the neighbor set $N_q = \{w, a, s\}$ and non-neighbor set $NON_q = A - \{N_q, q\} = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\} - \{w, a, s, q\} = \{b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, r, t, u, v, x, y, z\}$.

Now that we have a way to compute the neighbors of any key on our keyboard, we can construct the "neighborhood" of keys for a particular layout using our defined grid locations for the keys on the keyboard. See Table 3.1 for a list of neighbor sets for each key in the QWERTY layout, along with the total number of neighbors for each key. We call this comprehensive list of neighbors the **neighborhood** of a keyboard layout. The neighborhood of any keyboard layout can now be easily calculated by simply defining the grid for the keys, since distance only depends on grid locations. As we will see, the neighborhood calculation will be very important when measuring the accuracy of keyboard layouts. Next we define the probability distribution for the different outcomes that can occur when typing on an MT keyboard, each with their own weight.

3.6 Probability Distribution

We assign the total probability of error to be $\alpha = 0.1$. This means that we will hit the correct key 90 percent of the time. Now, to distribute α over the different errors that can occur, we weigh the probability of substitution to be more likely than insertion or deletion. We then weigh the probability of substituting a neighbor to be higher than substituting a non-neighbor. Table 3.2 shows a summary of the values we have assigned to our probability distribution. This probability distribution is used directly under our noisy channel framework for modeling error and is referenced furthermore simply as our error model.

Key	**Neighbors**	**Number of Neighbors**
A	S Q Z W X	5
B	N V G H F	5
C	V X D F S	5
D	F S E C R V X W	8
E	R W D F S	5
F	G D R V T B C E	8
G	H F T B Y N V R	8
H	J G Y N U M B T	8
I	O U K L J	5
J	K H U M I N Y	7
K	L J I O M U	6
L	K O P I	4
M	N J K H	4
N	M B H J G	5
O	P I L K	4
P	O L	2
Q	W A S	3
R	T E F G D	5
S	D A W X E C Z Q	8
T	Y R G H F	5
U	I Y J K H	5
V	B C F G D	5
W	E Q S D A	5
X	C Z S D A	5
Y	U T H J G	5
Z	X A S	3

Table 3.1: Neighborhood Definition for QWERTY Layout

Notice that the probability of insertion is distributed uniformly over all keys, so that any of the 26 keys are equally likely to be inserted. This probability is further split evenly between inserting either before or after the intended letter, so that the probability of inserting before or after the intended letter will be 0.005, respectively. Also, the probability of substituting a neighbor is distributed evenly over all neighbors of the intended key, while the probability of substituting a non-neighbor is distributed over all non-neighboring

Outcome	Total Probability of Outcome
Correct Key	0.9
Substitution(Neighbor)	0.05
Substitution(Non-Neighbor)	0.03
Insertion	0.01
Deletion	0.01

Table 3.2: Defined Error Model

keys of the intended key. Let us take the key a for example. When typing a, we sample over our error model (Table 3.2) to see the corresponding outcomes. Below is the list of probabilities for the possible outcomes that can occur when trying to type the letter a on the QWERTY keyboard, according to our error model. Letting A be our alphabet of character keys on the keyboard, i.e., $A = \{a, b, c, \cdots, y, z\}$. we have the following outcomes when attempting to type a:

- **Outcome 1 - Correct** : We type a correctly with probability 0.9
- **Outcome 2 - Insert Before** : We type σa with probability $\frac{0.005}{26}$, where $\sigma \in A$ and is chosen uniformly over A. There are 26 different letters that could be inserted before the intended character, which makes the total probability of this outcome 0.005.
- **Outcome 3 - Insert After** : We type $a\sigma$ with probability $\frac{0.005}{26}$, where $\sigma \in A$ and is chosen uniformly over A. There are 26 different letters that could be inserted after the intended character, which makes the total probability of this outcome 0.005.
- **Outcome 4 - Deletion** : We type $NULL$, the empty string, with probability 0.01.

- **Outcome 5 - Neighbor Substitution** : We type σ with probability $\frac{0.05}{|N_a|} = \frac{0.05}{5}$, where $\sigma \in N_a$ and is chosen uniformly over a's set of neighbors, N_a. There are 5 different neighboring letters that could be substituted in place of the intended character, which makes the total probability of this outcome 0.05.

- **Outcome 6 - Non-Neighbor Substitution** : We type σ with probability $\frac{0.03}{|NON_a|} = \frac{0.03}{20}$, where $\sigma \in NON_a$ and is chosen uniformly over a's set of non-neighbors, NON_a. There are 20 different non-neighboring letters that could be substituted in place of the intended character, which makes the total probability of this outcome 0.03.

3.7 Extending the Error Model to Whole Words

Now that we have an error model for characters, we can simulate what happens when a user types a whole word on an MT keyboard according to our character error model. We extend our character error model to accomplish this. For each character in the intended word, we sample our probability distribution to see the resulting character that actually gets registered to the display. Concatenating these results, we get the resulting word that is displayed on the screen. To showcase what the resulting outcomes would look like when typing a whole word using our extended character model, we show the set of possible outcomes when attempting to type the word *the* on the QWERTY keyboard given only a single error can occur:

- **Outcome 1 - Correct**: the

- **Outcome 2 - Deletion**: he, te, th
- **Outcome 3 - Insertion**: athe, bthe, cthe, dthe, ethe, fthe, gthe, hthe, ithe, jthe, kthe, lthe, mthe, nthe, othe, pthe, qthe, rthe, sthe, tthe, uthe, vthe, wthe, xthe, ythe, zthe, tahe, tbhe, tche, tdhe, tehe, tfhe, tghe, thhe, tihe, tjhe, tkhe, tlhe, tmhe, tnhe, tohe, tphe, tqhe, trhe, tshe, tuhe, tvhe, twhe, txhe, tyhe, tzhe, thae, thbe, thce, thde, thee, thfe, thge, thie, thje, thke, thle, thme, thne, thoe, thpe, thqe, thre, thse, thte, thue, thve, thwe, thxe, thye, thze, thea, theb, thec, thed, thef, theg, theh, thei, thej, thek, thel, them, then, theo, thep, theq, ther, thes, thet, theu, thev, thew, thex, they, thez
- **Outcome 4 - Substitution**: ahe, bhe, che, dhe, ehe, fhe, ghe, hhe, ihe, jhe, khe, lhe, mhe, nhe, ohe, phe, qhe, rhe, she, the, uhe, vhe, whe, xhe, yhe, zhe, tae, tbe, tce, tde, tee, tfe, tge, tie, tje, tke, tle, tme, tne, toe, tpe, tqe, tre, tse, tte, tue, tve, twe, txe, tye, tze, tha, thb, thc, thd, thf, thg, thh, thi, thj, thk, thl, thm, thn, tho, thp, thq, thr, ths, tht, thu, thv, thw, thx, thy, thz

To tie the concept of generating outcomes directly from our error model, let us show how we can extend our character model to calculate $P(s|w)$ for whole words, where w is the whole word we intend to type and s is the whole word that is output to the screen after going through the noisy channel. Here, $P(s|w)$ is the probability that, given we tried to type w, s was output to the screen. When dealing with characters, this was simple to compute directly from our error model. To extend to whole words, we assume that the individual probabilities of making an error for each character in the word are independent of one another. The following are examples from the previous outcome set, attempting to type the word *the* and allowing up to one error to occur:

- **Example: the $\rightarrow$ the.** We calculate the probability of this outcome by extending our character error model and assuming independence to yield the quantity $P(the|the) = P(t|t)P(h|h)P(e|e) = (0.9)(0.9)(0.9)$.

- **Example: the $\rightarrow$ he.** A deletion has occurred. Following the same method as before, $P(he|the) = P(NULL|t)P(h|h)P(e|e) = (0.01)(0.9)(0.9)$.

- **Example: the $\rightarrow$ they.** An insertion has occurred. Thus, $P(they|the) = P(t|t)P(h|h)P(ey|e) = (0.9)(0.9)(\frac{0.005}{26})$.

- **Example: the $\rightarrow$ toe.** A non-neighbor substitution has occurred since h and o are not neighbors. Thus, $P(toe|the) = P(t|t)P(o|h)P(e|e) = (0.9)(\frac{0.03}{17})(0.9)$, where 17 is the number of non-neighbors of h.

- **Example: the $\rightarrow$ thw.** A neighbor substitution has occurred since e and w are neighbors. Thus, $P(thw|the) = P(t|t)P(h|h)P(w|e) = (0.9)(0.9)(\frac{0.05}{5})$, where 5 is the number of neighbors of e.

3.8 Simulating Typos

When simulating the action of typing a word, we sample each letter, using our defined error model to see what is actually typed, character by character, concatenating these results together to get the total resulting word. The results are based on the weighted probabilities we assigned in our error model. Put simply, this means we will get the correct result most of the time and if error occurs it is most likely to be that of a neighbor substitution. Put more strictly, the results will comply and agree with our error model completely, which means if we sample a letter many times we will see the correct result around 90 percent of the time, a substitution around 8 percent of the time, a deletion around 1 percent of the time, and an insertion around 1 percent of the time.

Chapter 4

Autocorrector Model

In this chapter, we describe how we accomplish constructing an autocorrector for our keyboard model.

4.1 Motivation

In the previous chapter, we successfully modeled error when typing on an MT keyboard so we could simulate real-world typos. Yet simulating error when typing on the MT keyboard is only one component of our complete model of the MT keyboard. To go in the other direction is to model the automatic spelling corrector, or **autocorrector**. Given a typed word, which may match the intended word or be a result of an error, an autocorrector attempts to map back to the intended word correctly so that the word displayed matches the word that was intended. In other words, we want to model the keyboard's ability to reverse or prevent the errors that can occur when typing. The autocorrector obviously plays a major role in the accuracy of an MT keyboard since it can maintain accuracy, or even increase it, by reversing error. In this chapter, we use Bayesian methods to construct an autocorrector which we can use to simulate the keyboards ability to recover from typos, completing our model of the MT keyboard and allowing us to measure the total accuracy of a keyboard layout.

4.2 Methodology

The problem of autocorrection is as follows: given a word that the user has typed, let us try to find the best candidate to suggest as a "correction" for that word. The "correct" word is one which we find the most probable that the user actually intended. To do this, we first generate a set of candidate words. We then calculate a rank for each of these candidates, with a higher rank signifying a better candidate. Finally, we suggest the candidate with the highest rank as the word the user intended. Note that the word we suggest may be the original word typed, itself, if we determine it to be the word most likely to be intended by the user. Similar to spam filters, email classifiers, and other spell checkers, we employ Bayesian probability and language models as the foundation of our autocorrector. As even the most basic autocorrectors have been shown to perform relatively well [14], using these methods in our work will be sufficient to model an MT keyboard.

To autocorrect a word, our autocorrector first determines the list of candidate suggestion words. For each of the candidates, the autocorrector then calculates a rank, which is the probability that the user intended the candidate word.

Described formally, the rank of a candidate c for word w is

$$Rank(c) = P(c|w),$$

or the probability the user intended c, given w was typed to the screen, possibly by error. The autocorrector calculates the rank for the entire set of candidates C of w and ends by suggesting the candidate with the highest rank, or

$$Autocorrect(w) = argmax_{c \in C} Rank(c) = argmax_c P(c|w).$$

A summary of our method of autcorrecting a word follows. Given word w is typed to the screen, our autocorrector will go through the process below of suggesting a word:

1. **Step 1:** Find Set of Candidates

$$FindCandidates(w) = C \tag{4.2.1}$$

2. **Step 2:** Calculate Rank $\forall\ c \in C$

$$Rank(c) = P(c|w) \tag{4.2.2}$$

3. **Step 3:** Return Candidate with Largest Rank

$$Autocorrect(w) = argmax_{c \in C} Rank(c) \tag{4.2.3}$$

We will further explain how to generate the set C from Step 1 (Equation 4.2.1) in Section 4.2.1. To calculate Step 2 (Equation 4.2.2), we apply Bayes' Theorem, which is defined as

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}, \tag{4.2.4}$$

where $B \neq 0$, to get

$$Rank(c) = P(c|w) = \frac{P(w|c)P(c)}{P(w)}.$$

Since $P(w)$ is the same across all candidates c, it is safe to ignore it as the term would simply scale all ranks by the same factor. Thus, the definition of rank (Equation 4.2.2) turns into

$$Rank(c) = P(w|c)P(c). \tag{4.2.5}$$

This further transforms Equation 4.2.3 into

$$Autocorrect(w) = argmax_c Rank(c) = argmax_c P(w|c)P(c). \tag{4.2.6}$$

We describe how to calculate the terms $P(w|c)$ and $P(c)$ in sections 4.2.2.1 and 4.2.2.2, respectively. We will also end this chapter with a full example of our autocorrect method, tying all of the steps in this section together.

4.2.1 Generating Candidates

In this section, we seek to formalize how to accomplish the first step in our autocorrect process (Equation 4.2.1), how to generate a list of candidate suggections, C, for a word typed, w, or $Candidates(w) = C$. It is common in language modeling to quantify how dissimilar two words are by **edit distance** [55], which is the number of edits it would take to transform one word into another. An edit can be a deletion (removal of a letter), a substitution (the alteration of one letter for another), or an insertion (adding a letter). The edit distance we use is sometimes referred to as Levenshtein distance, from his work on self-correcting binary codes [36], and is defined as $eddist_{a,b}(|a|, |b|)$ for the two strings a and b, or

$$eddist_{a,b}(i,j) \begin{cases} max(i,j) & if min(i,j) = 0, \\ min \begin{cases} eddist_{a,b}(i-1,j)+1 \\ eddist_{a,b}(i,j-1)+1 \\ eddist_{a,b}(i-1,j-1)+1 \end{cases} & otherwise. \end{cases} \tag{4.2.7}$$

Here, $eddist_{a,b}(i, j)$ is the distance between the first i characters of a and the first j characters of b. For example, the edit distance between "kitten" and "sitting" is 3 which can be seen as follows: kitten → sitten (substitution of s for k), sitten → sittin (substitution of i for e), sittin → sitting (insertion of g at the end).

To generate our candidate set C for word w, we want to generate a set of all words $c \in C$ that, when attempting to type the word c, could have led to, possibly by error, the word w being output to the device screen. We want to utilize our error model so that we consider all possible errors within our error model that could have led to the output, w. Notice that we can easily enumerate the set C by using the definition of edit distance. For example, considering all words within edit distance one of w would give us all words that are within one substitution, one insertion, or one deletion away from w, which is exactly in line with our error model when simulating typing the word w and allowing up to one typo to occur. Thus, to generate our set C, we will calculate all words c such that $eddist(w, c) = 1$. The resulting set C can be very large. For a word of length n, there will be n deletions, $26n$ substitutions, and $26(n + 1)$ insertions for a total of $53n + 26$ candidates. For example, the word *something* has 503 candidates. To help reduce the number of candidates we generate, we can remove any duplicates and also remove any words that are not well-spelled, since the autocorrector will not consider correcting to a word that doesn't exist within the dictionary of well-spelled words. For our autocorrector, we cross-checked all candidates with a stored English dictionary [28] to ensure we only considered candidates who were well-spelled. Due to the large number of candidates and the constraints of our computational resources, we limit our work to only consider up to one typo while typing a word, as going up to two typos would be too computationally expensive for the autocorrection that we require. Additionally, most mistakes in typing have corrections within edit distance one [47], so considering candidates that are edit distance one away from the typed word on our keyboard will suffice for our autocorrector. This explains why we only consider words that are edit distance one away when generating the set C.

4.2.2 Calculating Rank

Once we have the candidate list C for the word w, we need to calculate the rank for each c, which is defined as (Equation 4.2.5) $Rank(c) = P(w|c)P(c)$. There are two parts to calculating our expression for rank. First, the term

$P(w|c)$ is the probability a user would type the word w if c were intended, possibly by mistake. We can use our error model from Chapter 3 to calculate the probability of this error occurring. The second term in the expression, $P(c)$ can be derived from a language model, and seeks to answer how likely it is that the word c appears in English text, overall. We outline how to calculate both of these quantities in the following subsections.

4.2.2.1 Calculating $P(w|c)$

We calculate $P(w|c)$, that is, the probability we typed w, given we intended to type c. This is where our error model (Table 3.2) comes in. Recall Chapter 3, where we defined the likelihood of an error occurring when typing a word. We constructed a probability distribution for the outcomes involved when typing a word on an MT keyboard. Given we try to type the character a, our outcomes are summarized below.

- **Outcome 1 - Correct** : We type a correctly with probability 0.9
- **Outcome 2 - Insert Before** : We type σa with probability $\frac{0.005}{26}$, where $\sigma \in A$ and is chosen uniformly over A. There are 26 different letters that could be inserted before the intended character, which makes the total probability of this outcome 0.005.
- **Outcome 3 - Insert After** : We type $a\sigma$ with probability $\frac{0.005}{26}$, where $\sigma \in A$ and is chosen uniformly over A. There are 26 different letters that could be inserted after the intended character, which makes the total probability of this outcome 0.005.

- **Outcome 4 - Deletion** : We type $NULL$, the empty string, with probability 0.01.

- **Outcome 5 - Neighbor Substitution** : We type σ with probability $\frac{0.05}{|N_a|} = \frac{0.05}{5}$, where $\sigma \in N_a$ and is chosen uniformly over a's set of neighbors, N_a. There are 5 different neighboring letters that could be substituted in place of the intended character, which makes the total probability of this outcome 0.05.

- **Outcome 6 - Non-Neighbor Substitution** : We type σ with probability $\frac{0.03}{|NON_a|} = \frac{0.03}{20}$, where $\sigma \in NON_a$ and is chosen uniformly over a's set of non-neighbors, NON_a. There are 20 different non-neighboring letters that could be substituted in place of the intended character, which makes the total probability of this outcome 0.03.

Recall that our error model was a character model that we extended to whole words by calculating error independently per character and concatenating the results for the entire string.

Thus, we need to calculate $P(w|c)$ independently per character or

$$P(w|c) = \prod_i P(w_i|c_i), \tag{4.2.8}$$

for $0 \leq i \leq length(c)$, where w_i and c_i are the ith character of w and c, respectively.

For example, let $w =$*tre* and $c =$*the*, so that we intended to type *the* but *tre* was output to the screen. Then we would calculate $P(tre|the)$ as follows:

$$P(tre|the) = P(t|t)P(r|h)P(e|e)$$

From our error model, we can see that the first term represents a "correct" outcome, the second term represents a non-neighbor substitution, and the third also represents a correct outcome. Then, the above equation turns into

$$P(t|t)P(r|h)P(e|e) = (0.9)(\frac{0.03}{17})(0.9),$$

since the total defined probability of a non-neighbor substitution is 0.03 and the number of non-neighbors (Section 3.5) of h is $|NON_h| = 17$. Notice that the above scenario could also be a result of a different combination of errors such as $P(t|t)P(null|h)P(re|e)$, where the user incurs a deletion error and also an insertion error. Though, with our definition of edit distance, we will always choose the scenario with minimum edits. Since the minimum number of edits to transform *the* to *tre* is one, i.e. a single substitution, we would use this scenario over an alternative that involves more than one edit.

4.2.2.2 Calculating $P(c)$

To calculate $P(c)$ we need to construct a language model, so that we can answer the question of how likely c is to appear in English text, overall. To do this, we want to analyze how frequently c appears over all of the English language relative to other words, which will tell us how likely, or probable, it is. To construct our language model, we process a large English corpus [3] and use a hash map to track the frequencies of all words in the corpus. We call this hash map our **frequency map**. The corpus we use was provided by the American National Corpus and contains 14,623,927 words derived from spoken data, written data, and switchboard transcriptions including conversations, narratives, and interviews. We directly use our frequency map to calculate $P(c)$. Let $freq(c)$ be the total number of times c appears in our large corpus and $totalCorpWords$ be the total number of words in our corpus. For example, the word *the* had a frequency of 79403 in our frequency map, while the word *humble* had a frequency of 13. We then define the probability of c to be

$$P(c) = \frac{freq(c)}{totalCorpWords}. \tag{4.2.9}$$

4.2.3 Full Example

Now, we provide a full example to tie together all of the sections of this chapter and review the total process of our autocorrector. An updated summary of our autocorrector's process is below. Given word w is typed to the screen, our autocorrector will go through the process below of suggesting a word:

1. **Step 1:** Find Set of Candidates (Section 4.2.1)
    - Generate all words c, such that $eddist(w, c) = 1$ (Equation 4.2.7).
    - Remove any duplicates from the candidate set, C.
    - Remove any candidates $c \in C$, such that $c \notin D$, where D is the English dictionary of well-spelled words.
2. **Step 2:** Calculate Rank $\forall\ c \in C$ (Section 4.2.2).

- $Rank(c) = P(c|w) = P(w|c)P(c)$ (Equations 4.2.8 and 4.2.9).

3. **Step 3:** Return Candidate with Largest Rank.

   - $Autocorrect(w) = argmax_{c \in C} Rank(c)$ (Equation 4.2.6).

For our example, the autocorrector detects the word *tre* has been typed on the MT keyboard. The autocorrector will now follow the steps above to provide a suggestion.

1. **Step 1:** Find Set of Candidates

   The autocorrector generates the candidate set, or any word that is within edit distance one of the word *tre*, removes duplicates and checks each of these candidates against the dictionary, removing any words that are not well-spelled. The resulting set of candidates, C for the word tre is :

$$\mathcal{C} = \{\, re, are, ere, ire, ore, pre, tee, the, tie, toe,$$
$$tue, try, tare, tire, tore, tree, true, trek \,\}.$$

2. **Step 2:** Calculate Rank $\forall\ c \in C$ (Section 4.2.2).

   Using the equation for Rank (Equation 4.2.2), the autocorrector calculates $Rank(c)$ for all candidates in C. The resulting ranks for the candidates of tre are summarized in Table 4.1.

3. **Step 3:** Return Candidate with Largest Rank

   We can see that the candidate $c = the$ is clearly the winner, with the largest rank among the set C. Thus, the autocorrector would return the

Candidate	**Rank**
the	1.0442850471902756e-4
are	4.0300120346243876e-6
true	1.375011040939805e-6
tree	2.6158746632513367e-7
tore	1.207326767654463e-7
try	9.613898335026279e-8
toe	2.8745875420344356e-8
tire	2.012211279424105e-8
tie	1.6768427328534208e-8
tee	7.452634368237426e-9
trek	6.7073709314136835e-9
ore	3.193986157816039e-9
tue	1.1178951552356138e-9
pre	9.720827436831423e-10
re	6.369772964305492e-10

Table 4.1: Ranks for Candidates(*tre*)

word *the* as a suggestion. To see how the rank broke down a bit further, we provide the values of the two terms used to determine rank, namely $P(w|c)$ and $P(c)$. First, a summary of the values calculated for $P(tre|c)$ for each of the candidates $c \in C$ are provided in Table 4.2. Then, in Table 4.2, you can see there were many words other than *the* that had a higher probability of being meant to be typed, given a user typed *tre* to the screen. Because it is more probable, per our error model, to have a neighbor substitution, you can see that $P(tre|tee)$ is more probable than $P(tre|the)$, since e is a neighbor of r, while h is not. This exemplifies why our language model is so important, as bringing in the term $P(c)$ will weigh these probabilities according to real life frequency data. The values for $P(c)$, over all $c \in C$, are provided in Table 4.3.

Candidate	P(tre|c)
re	1.730769230769231e-4
are	0.0012150000000000002
ere	0.0012150000000000002
ire	0.0012150000000000002
ore	0.0011571428571428572
pre	0.0010565217391304347
tee	0.008100000000000001
the	0.0014294117647058824
tie	0.0012150000000000002
toe	0.0011571428571428572
tue	0.0012150000000000002
try	0.0012150000000000002
tare	0.007290000000000001
tire	0.007290000000000001
tore	0.007290000000000001
tree	0.007290000000000001
true	0.007290000000000001
trek	0.007290000000000001

Table 4.2: $P(tre|c)$ for Candidates(tre)

From Table 4.3, it is clear that the word *the* is much more probable than the word *tee* in our frequency map, thus our total ranking (Equation 4.2.4) for *the* ends up being higher when ranking the candidates, and *the* is inevitably chosen over *tee* as the suggested word to correct to.

Candidate	$\mathbf{P(c)}$
re	3.680313268265395e-6
are	0.003316882333024187
ere	9.200783170663487e-7
ire	9.200783170663487e-7
ore	2.760234951199046e-6
pre	9.200783170663487e-7
tee	9.200783170663487e-7
the	0.07305697861001928
tie	1.380117475599523e-5
toe	2.4842114560791416e-5
tue	9.200783170663487e-7
try	7.912673526770599e-5
tare	9.200783170663487e-7
tire	2.760234951199046e-6
tore	1.6561409707194277e-5
tree	3.58830543655876e-5
true	1.8861605499860147e-4
trek	9.200783170663487e-7

Table 4.3: $P(c)$ for Candidates(*tre*)

Chapter 5

Experimental Results

In this chapter, we discuss our methods of testing using the total keyboard model in conjunction with a large corpus of test strings, allowing us to measure the accuracy of a keyboard layout. We then test all 325 2-key swaps, comparing each layout's accuracy to the original QWERTY configuration. We also present our results, showing which 2-key swap performed the best.

5.1 Measuring Keyboard Layout Accuracy

With our total keyboard model in place, we are able to proceed with measuring the total accuracy of an MT keyboard layout. We begin with the QWERTY layout to have a baseline to measure against. Once our baseline accuracy measurement is achieved, for all 325 2-key swaps, we iterate through swapping two keys at a time on the QWERTY layout. For each 2-key swap, we measure the accuracy of the layout, allowing us to decide which 2-key swaps perform better than the original QWERTY layout and by how much.

In measuring the accuracy of a keyboard layout, what we are trying to determine at a high level is how accurate the keyboard is at outputting to the screen what the user intended to type. To do this, we use a large set of testing words, T, which includes hundreds of thousands of misspelled words. The autocorrector is then ran on each $w \in T$, suggesting a word s. We determine the accuracy of this suggestion by calculating the probability that the user intended s in the first place, $P(s|w)$, which provides us the likelihood that what the autocorrector model suggested was correct. In other words, we are measuring the probability that s, our suggestion, was indeed the word intended, given w was (perhaps erroneously) output to the screen, or $P(\textit{intended } s|\textit{typed } w)$. Notice that we can again apply Bayes' Rule (Equation 4.2.4) to get

$$P(s|w) = \frac{P(w|s)P(s)}{P(w)}.$$

Since $P(w)$ is the same over all s, we ignore it as a scaling factor, so that what we truly need to calculate is

$$P(s|w) = P(w|s)P(s). \tag{5.1.1}$$

Equation 5.1.1 above gives us a measurement of how accurate our keyboard was in its autocorrection of w to s. The first term, $P(w|s)$, is the probability of making the errors that would result in w being to the screen, given s was intended in the first place, or $P(w \textit{ typed}|s \textit{ intended})$, which we can determine directly from our error model (Table 3.2). Multiplying by $P(s)$ allows us to weigh this probability by the frequency of s over the English language (see Section 4.2.2.2).

We construct T to include all mispellings of s within edit distance one, so that from our tests, we can measure the total probability that we autocorrected to s correctly over all $w \in T$. In this way, we get a weighted total probability of autocorrecting back to s from all of the possible erroneous words that could be typed by mistake (within edit distance one of s). Though, this only provides the accuracy of a keyboard layout for a single word, s. To get an overall weighted accuracy for a keyboard layout, we will repeat this process for many words, to see how the keyboard performs in getting a large corpus correctly output to the screen.

We begin testing a layout by configuring our grid of keys, as described in Section 3.5. When we swap 2 keys on the QWERTY keyboard, we update the locations of the two letters in the grid (Figure 3.2) which causes the neighborhood of keys (Table 3.5) to be updated. Then, to generate a large set of testing words, we derive the set T from our frequency map F. Recall that F was provided by the American National Corpus and contains 14,623,927 words derived from spoken data, written data, and switchboard transcriptions including conversations, narratives, and interviews. We derive T by enumerating all possible errors, within edit distance one, that could occur when typing each word in F. We generate T in line with our error model (Table 3.2), so that it contains thousands of misspelled words for us to test on. For each word, $w \in T$, we run our autocorrect algorithm (Chapter 4). We then calculate Equation 5.1.1 for this autocorrection, giving us the probability that we autocorrected correctly. Using the above method, for each word $s \in F$, we measure the total probability that we output s correctly to the screen, over all $w \in T$. This total probability is what we call the *Weighted Accuracy Rating* (WAR rating) for s, so that for each word $s \in F$, we will have a WAR rating. Let J be the set of words that autocorrect to some $s \in F$ such that $J \subset T$. We define the WAR rating for s to be

$$WAR(s) = \sum_{w \in J} P(w|s)P(s). \tag{5.1.2}$$

This gives us an accuracy rating for a single word. To get the total measure of accuracy for a keyboard layout, we want to tally this rating over a dictionary of words. We call this total probability over all testing words the *Overall WAR Rating* for a keyboard layout or

$$OverallWAR = \sum_{s \in F} WAR(s). \tag{5.1.3}$$

That is, to determine how well a keyboard layout performs, we test it by typing all possible errors (T) for thousands of words from the English corpus (F), and then observe the totality of how well we autocorrected, calculating the likelihood that we intended the suggested autocorrections. The Overall WAR rating will be the accuracy metric we use to determine and compare performance of keyboard layouts.

We ran our tests on both the QWERTY layout, as well as the 325 2-key swap layouts so that for each keyboard layout, each word in F received a WAR rating. Further, we were able to use Equation 5.1.3 to determine an Overall WAR rating for each keyboard layout itself, which tells us how accurate the layout was at outputting all of the words from our large corpus correctly to the screen. Next, we present our results, providing the Overall WAR ratings for all of the layouts we tested on and comparing their performance to the original QWERTY layout.

5.2 2-Key Swap Results

Recall from Section 5.1 that the WAR rating (Equation 5.1.2) for a word $s \in F$ on a particular keyboard layout is the total probability that we output s correctly to the screen, over all $w \in T$, where F is our language model and T is our set of testing words. This gives us an accuracy rating for a single word. An Overall WAR rating provides us this total probability over an entire dictionary of words, $s \in F$, as defined in Equation 5.1.3, so that the higher the Overall WAR rating, the better the accuracy of the keyboard layout. As these ratings are total probabilities, they will remain in the range of zero and one, inclusive, with a WAR rating of one meaning 100% accuracy for the keyboard layout. We started by running our tests on the original QWERTY layout, which yielded an Overall WAR rating of 0.85294, or 85.294% accurate. We have summarized our test results for all 2-key swaps in Table 5.1. For each 2-key swap, we provide the relative increase or decrease in WAR rating relative to the original QWERTY layout. In Table 5.2 we provide a comprehensive list of the swaps that performed better than QWERTY, along with their Overall WAR ratings. In Table 5.3, we provide a summary of the best swap per letter with their associated Overall WAR ratings. In Table 5.4 we provide a list of the top 10 swaps that performed better than QWERTY and their Overall WAR ratings, followed by Table 5.5 which provides the bottom 10 swaps which had the lowest accuracy ratings, overall, along with those WAR ratings.

A few interesting notes of observation include the fact that nearly half of the 2-key swaps performed better than QWERTY, with 157 out of 325 alternative layouts having a better WAR rating. We also observed that 13 out of 26 letters' best swap was with the letter i, while all but one of the overall top ten swaps was a swap with i. We can also see that the best overall 2-key swap layout is when the keys i and t are swapped, with the $i \leftrightarrow t$ swap having an overall WAR rating of 0.8544658504333, which is an accuracy increase of approximately 0.1785%. This means that if we attempted to type 100,000 words on the original QWERTY layout we would expect to see about 85,294 of the words correctly output to the device screen, while if we typed these same words on the $i \leftarrow t$ layout we would see 85,447 words typed correctly. This would result in over 150 words which we wouldn't have to fix manually, adding up to a lot of time saved for the user. It is easily suspected that the best place to place a vowel such as i would be away from other vowels, to avoid our "Problem with Neighbors", as exemplified in Section 1.2, and that the placement of i, o, and u so closely together on the QWERTY layout may be the largest culprit of this problem since it is the biggest cluster of vowels on QWERTY. Since i lay in between o and u it would make sense if it was more susceptible to this problem than o and u do by themselves since it is adjacent to not one, but two vowels, compounding the problem. Thus, it makes sense that we see a trend in which many of the best swaps involve i, removing it from it's place between o and u. Note that there are only 7 keys that are not neighbors of any vowel: t, g, c, v, b, n, and m. Four of these, when swapped with i, place in the top ten swaps. The others' best 2-key swap involve a swap with themselves and i. This further hints that our intuition is correct, as placing i away from any vowels seems to help in boosting accuracy more prominently than swaps involving other keys. Notice that any 2-key swap has a double effect in that we must consider how the update of the neighborhood for each other the 2 keys will effect accuracy. For example, we may wonder why the swap between i and t, in particular, is the best swap rather than, say, swapping i with any of the other keys that would isolate it from vowels (i.e., g, c, v, b, n, m). We suspect that this is due to the fact that the increase in accuracy due to i being placed around non-vowels is only one factor in the overall increase in accuracy. We also must consider the increase (or decrease) in accuracy that comes with swapping the other key with i. For example, notice that t is originally surrounded by letters such as f and r. We can immediately think of words that are problematic within this placement (i.e., $right$, $fight$, $tight$, etc), which enhance the "Problem with Neighbors" for t. Moving t away from this placement into the old position of i would reduce some of these problems. This could be a factor in why $i \leftrightarrow t$ performs better than the other 2-key swaps involving i. Overall, we can see that it would be best to reconfigure i's placement on the QWERTY layout and that the swap of $i \leftrightarrow t$ would have the best performance increase of nearly 0.18 percent.

Table 5.1: All 2-Key Swap Relative WAR Ratings

Swap	+/- WAR
A↔B	0.00034
A↔C	0.00013
A↔D	0.00050
A↔E	-0.00017
A↔F	0.00058
A↔G	0.00056
A↔H	0.00054
A↔I	-0.00031
A↔J	0.00166
A↔K	0.00204
A↔L	0.00199
A↔M	0.00079
A↔N	0.00047
A↔O	0.00029
A↔P	0.00117
A↔Q	-0.00019
A↔R	0.00059
A↔S	0.00007
A↔T	0.00098
A↔U	0.00111
A↔V	0.00018

Swap	+/- WAR
A↔W	0.00075
A↔X	-0.00006
A↔Y	0.00074
A↔Z	-0.00035
B↔C	-0.00022
B↔D	-0.00049
B↔E	-0.00029
B↔F	0.00029
B↔G	-0.00018
B↔H	0.00000
B↔I	-0.00097
B↔J	0.00031
B↔K	0.00034
B↔L	0.00043
B↔M	0.00010
B↔N	0.00052
B↔O	-0.00044
B↔P	0.00008
B↔Q	0.00000
B↔R	-0.00017
B↔S	0.00023
B↔T	0.00032

Swap	+/- WAR
B↔U	-0.00008
B↔V	-0.00013
B↔W	0.00018
B↔X	0.00002
B↔Y	0.00011
B↔Z	-0.00013
C↔D	-0.00025
C↔E	-0.00005
C↔F	-0.00028
C↔G	-0.00013
C↔H	-0.00018
C↔I	-0.00108
C↔J	0.00003
C↔K	0.00006
C↔L	0.00012
C↔M	0.00007
C↔N	0.00059
C↔O	-0.00076
C↔P	-0.00006
C↔Q	0.00001
C↔R	-0.00001
C↔S	-0.00033

Swap	+/- WAR
C↔T	0.00071
C↔U	-0.00034
C↔V	0.00003
C↔W	-0.00006
C↔X	-0.00002
C↔Y	0.00031
C↔Z	-0.00008
D↔E	0.00009
D↔F	0.00013
D↔G	0.00000
D↔H	0.00024
D↔I	-0.00102
D↔J	-0.00019
D↔K	-0.00036
D↔L	-0.00025
D↔M	-0.00009
D↔N	0.00034
D↔O	-0.00044
D↔P	-0.00047
D↔Q	-0.00037
D↔R	0.00031
D↔S	0.00010

Swap	**+/- WAR**
D↔T	0.00083
D↔U	-0.00044
D↔V	-0.00051
D↔W	-0.00002
D↔X	-0.00042
D↔Y	0.00016
D↔Z	-0.00042
E↔F	0.00004
E↔G	0.00001
E↔H	0.00025
E↔I	-0.00038
E↔J	0.00044
E↔K	0.00095
E↔L	0.00113
E↔M	0.00008
E↔N	0.00053
E↔O	-0.00088
E↔P	0.00068
E↔Q	0.00040
E↔R	0.00006
E↔S	0.00085
E↔T	0.00126

Swap	**+/- WAR**
E↔U	-0.00006
E↔V	-0.00025
E↔W	0.00065
E↔X	0.00053
E↔Y	0.00003
E↔Z	0.00036
F↔G	0.00045
F↔H	0.00011
F↔I	-0.00096
F↔J	0.00026
F↔K	-0.00023
F↔L	0.00006
F↔M	0.00083
F↔N	0.00077
F↔O	-0.00035
F↔P	-0.00043
F↔Q	-0.00038
F↔R	0.00017
F↔S	0.00117
F↔T	0.00079
F↔U	-0.00029
F↔V	-0.00044

Swap	+/- WAR
F↔W	0.00019
F↔X	-0.00032
F↔Y	0.00038
F↔Z	-0.00046
G↔H	0.00015
G↔I	-0.00100
G↔J	-0.00029
G↔K	-0.00026
G↔L	0.00008
G↔M	0.00012
G↔N	0.00094
G↔O	-0.00016
G↔P	-0.00020
G↔Q	-0.00042
G↔R	0.00015
G↔S	0.00127
G↔T	0.00042
G↔U	-0.00055
G↔V	-0.00022
G↔W	0.00026
G↔X	-0.00032
G↔Y	0.00007

Swap	+/- WAR
G↔Z	-0.00043
H↔I	-0.00067
H↔J	-0.00015
H↔K	0.00030
H↔L	0.00034
H↔M	-0.00010
H↔N	0.00058
H↔O	-0.00065
H↔P	0.00001
H↔Q	-0.00001
H↔R	-0.00016
H↔S	0.00134
H↔T	0.00025
H↔U	-0.00072
H↔V	-0.00017
H↔W	0.00022
H↔X	-0.00031
H↔Y	-0.00029
H↔Z	-0.00043
I↔J	-0.00060
I↔K	-0.00002
I↔L	-0.00019

Swap	+/- WAR
I↔M	-0.00104
I↔N	-0.00110
I↔O	-0.00016
I↔P	0.00010
I↔Q	0.00030
I↔R	-0.00132
I↔S	-0.00029
I↔T	-0.00152
I↔U	-0.00061
I↔V	-0.00116
I↔W	0.00007
I↔X	-0.00007
I↔Y	-0.00087
I↔Z	0.00025
J↔K	0.00002
J↔L	0.00033
J↔M	0.00016
J↔N	0.00030
J↔O	0.00053
J↔P	0.00015
J↔Q	-0.00004
J↔R	-0.00005

Swap	+/- WAR
J↔S	0.00049
J↔T	0.00023
J↔U	0.00001
J↔V	0.00006
J↔W	0.00034
J↔X	-0.00001
J↔Y	0.00007
J↔Z	-0.00003
K↔L	0.00008
K↔M	0.00005
K↔N	0.00022
K↔O	0.00011
K↔P	0.00006
K↔Q	-0.00005
K↔R	-0.00014
K↔S	-0.00005
K↔T	-0.00016
K↔U	0.00010
K↔V	0.00003
K↔W	0.00014
K↔X	0.00000
K↔Y	0.00006

Swap	+/- WAR
K↔Z	-0.00007
L↔M	0.00011
L↔N	-0.00003
L↔O	0.00000
L↔P	-0.00009
L↔Q	0.00007
L↔R	-0.00026
L↔S	-0.00046
L↔T	-0.00039
L↔U	0.00023
L↔V	0.00014
L↔W	0.00012
L↔X	0.00019
L↔Y	0.00013
L↔Z	-0.00001
M↔N	-0.00008
M↔O	-0.00043
M↔P	-0.00016
M↔Q	0.00014
M↔R	-0.00014
M↔S	0.00005
M↔T	0.00012

Swap	+/- WAR
M↔U	-0.00026
M↔V	0.00019
M↔W	0.00031
M↔X	0.00022
M↔Y	0.00003
M↔Z	-0.00003
N↔O	-0.00056
N↔P	-0.00036
N↔Q	0.00028
N↔R	0.00060
N↔S	0.00013
N↔T	0.00031
N↔U	-0.00023
N↔V	0.00038
N↔W	0.00081
N↔X	0.00047
N↔Y	0.00042
N↔Z	0.00023
O↔P	-0.00099
O↔Q	-0.00008
O↔R	-0.00001
O↔S	-0.00019

Swap	**+/- WAR**
O↔T	-0.00099
O↔U	0.00030
O↔V	-0.00071
O↔W	0.00056
O↔X	-0.00010
O↔Y	-0.00039
O↔Z	-0.00015
P↔Q	0.00006
P↔R	-0.00049
P↔S	-0.00089
P↔T	-0.00092
P↔U	0.00000
P↔V	0.00005
P↔W	-0.00024
P↔X	0.00008
P↔Y	-0.00014
P↔Z	0.00000
Q↔R	-0.00043
Q↔S	-0.00097
Q↔T	0.00145
Q↔U	-0.00004
Q↔V	-0.00006

Swap	**+/- WAR**
Q↔W	-0.00019
Q↔X	0.00000
Q↔Y	0.00007
Q↔Z	0.00000
R↔S	0.00086
R↔T	0.00025
R↔U	-0.00063
R↔V	-0.00023
R↔W	0.00013
R↔X	-0.00024
R↔Y	0.00026
R↔Z	-0.00051
S↔T	0.00043
S↔U	-0.00015
S↔V	-0.00009
S↔W	-0.00009
S↔X	-0.00049
S↔Y	0.00118
S↔Z	-0.00111
T↔U	-0.00071
T↔V	0.00024
T↔W	0.00119

Swap	**+/- WAR**
T↔X	0.00102
T↔Y	-0.00016
T↔Z	0.00096
U↔V	-0.00033
U↔W	0.00025
U↔X	-0.00005
U↔Y	-0.00031
U↔Z	-0.00006
V↔W	0.00005
V↔X	-0.00001
V↔Y	0.00013
V↔Z	-0.00007
W↔X	0.00003
W↔Y	0.00091
W↔Z	-0.00017
X↔Y	0.00026
X↔Z	0.00000
Y↔Z	-0.00004

Table 5.2: All 2-Key Swaps Outperforming QWERTY

Swap	WAR
A ↔ E	0.85311
A ↔ I	0.85325
A ↔ Q	0.85313
A ↔ X	0.85300
A ↔ Z	0.85329
B ↔ C	0.85316
B ↔ D	0.85343
B ↔ E	0.85324
B ↔ G	0.85312
B ↔ H	0.85295
B ↔ I	0.85391
B ↔ O	0.85338
B ↔ R	0.85311
B ↔ U	0.85302
B ↔ V	0.85307
B ↔ Z	0.85307
C ↔ D	0.85320
C ↔ E	0.85300
C ↔ F	0.85322
C ↔ G	0.85307
C ↔ H	0.85312
C ↔ I	0.85403
C ↔ O	0.85370
C ↔ P	0.85301
C ↔ R	0.85295
C ↔ S	0.85327
C ↔ U	0.85329
C ↔ W	0.85300
C ↔ X	0.85297
C ↔ Z	0.85302
D ↔ G	0.85294
D ↔ I	0.85396
D ↔ J	0.85313
D ↔ K	0.85330
D ↔ L	0.85319
D ↔ M	0.85303
D ↔ O	0.85338
D ↔ P	0.85341
D ↔ Q	0.85331
D ↔ U	0.85338
D ↔ V	0.85345
D ↔ W	0.85297
D ↔ X	0.85336

Swap	**WAR**
D $\leftrightarrow$ Z	0.85337
E $\leftrightarrow$ I	0.85332
E $\leftrightarrow$ O	0.85382
E $\leftrightarrow$ U	0.85300
E $\leftrightarrow$ V	0.85320
F $\leftrightarrow$ I	0.85390
F $\leftrightarrow$ K	0.85317
F $\leftrightarrow$ O	0.85329
F $\leftrightarrow$ P	0.85337
F $\leftrightarrow$ Q	0.85333
F $\leftrightarrow$ U	0.85323
F $\leftrightarrow$ V	0.85339
F $\leftrightarrow$ X	0.85326
F $\leftrightarrow$ Z	0.85340
G $\leftrightarrow$ I	0.85394
G $\leftrightarrow$ J	0.85324
G $\leftrightarrow$ K	0.85320
G $\leftrightarrow$ O	0.85310
G $\leftrightarrow$ P	0.85314
G $\leftrightarrow$ Q	0.85336
G $\leftrightarrow$ U	0.85349
G $\leftrightarrow$ V	0.85317

Swap	**WAR**
G $\leftrightarrow$ X	0.85326
G $\leftrightarrow$ Z	0.85337
H $\leftrightarrow$ I	0.85362
H $\leftrightarrow$ J	0.85309
H $\leftrightarrow$ M	0.85304
H $\leftrightarrow$ O	0.85360
H $\leftrightarrow$ Q	0.85296
H $\leftrightarrow$ R	0.85310
H $\leftrightarrow$ U	0.85367
H $\leftrightarrow$ V	0.85311
H $\leftrightarrow$ X	0.85326
H $\leftrightarrow$ Y	0.85323
H $\leftrightarrow$ Z	0.85337
I $\leftrightarrow$ J	0.85354
I $\leftrightarrow$ K	0.85296
I $\leftrightarrow$ L	0.85313
I $\leftrightarrow$ M	0.85399
I $\leftrightarrow$ N	0.85404
I $\leftrightarrow$ O	0.85310
I $\leftrightarrow$ R	0.85426
I $\leftrightarrow$ S	0.85323
I $\leftrightarrow$ T	0.85447

Swap	WAR
I ↔ U	0.85355
I ↔ V	0.85410
I ↔ X	0.85301
I ↔ Y	0.85381
J ↔ Q	0.85298
J ↔ R	0.85300
J ↔ X	0.85295
J ↔ Z	0.85298
K ↔ Q	0.85300
K ↔ R	0.85308
K ↔ S	0.85299
K ↔ T	0.85310
K ↔ X	0.85294
K ↔ Z	0.85301
L ↔ N	0.85297
L ↔ P	0.85303
L ↔ R	0.85320
L ↔ S	0.85340
L ↔ T	0.85333
L ↔ Z	0.85295
M ↔ N	0.85303
M ↔ O	0.85338

Swap	WAR
M ↔ P	0.85310
M ↔ R	0.85308
M ↔ U	0.85321
M ↔ Z	0.85298
N ↔ O	0.85350
N ↔ P	0.85330
N ↔ U	0.85317
O ↔ P	0.85393
O ↔ Q	0.85302
O ↔ R	0.85296
O ↔ S	0.85313
O ↔ T	0.85393
O ↔ V	0.85365
O ↔ X	0.85304
O ↔ Y	0.85333
O ↔ Z	0.85309
P ↔ R	0.85344
P ↔ S	0.85383
P ↔ T	0.85386
P ↔ U	0.85295
P ↔ W	0.85318
P ↔ Y	0.85308

Swap	WAR
V $\leftrightarrow$ X	0.85295
V $\leftrightarrow$ Z	0.85301
W $\leftrightarrow$ Z	0.85311
Y $\leftrightarrow$ Z	0.85298

Table 5.3: Best Swap per Letter

Letter	Best Swap	WAR
A	A$\leftrightarrow$Z	0.85329
B	B$\leftrightarrow$I	0.85391
C	C$\leftrightarrow$I	0.85403
D	D$\leftrightarrow$I	0.85396
E	E$\leftrightarrow$O	0.85382
F	F$\leftrightarrow$I	0.85390
G	G$\leftrightarrow$I	0.85394
H	H$\leftrightarrow$U	0.85367
I	I$\leftrightarrow$T	0.85447
J	J$\leftrightarrow$I	0.85354
K	K$\leftrightarrow$D	0.85330
L	L$\leftrightarrow$S	0.85340
M	M$\leftrightarrow$I	0.85399
N	N$\leftrightarrow$I	0.85404
O	O$\leftrightarrow$P	0.85393
P	P$\leftrightarrow$O	0.85393
Q	Q$\leftrightarrow$S	0.85391
R	R$\leftrightarrow$I	0.85426
S	S$\leftrightarrow$Z	0.85405
T	T$\leftrightarrow$I	0.85447
U	U$\leftrightarrow$H	0.85367
V	V$\leftrightarrow$I	0.85410
W	W$\leftrightarrow$P	0.85318
X	X$\leftrightarrow$S	0.85343
Y	Y$\leftrightarrow$I	0.85381
Z	Z$\leftrightarrow$S	0.85405

Table 5.4: Top Ten 2-Key Swaps

Swap	WAR
I $\leftrightarrow$ T	0.85447
I $\leftrightarrow$ R	0.85426
I $\leftrightarrow$ V	0.85410
S $\leftrightarrow$ Z	0.85405
I $\leftrightarrow$ N	0.85404
C $\leftrightarrow$ I	0.85403
I $\leftrightarrow$ M	0.85399
D $\leftrightarrow$ I	0.85396
G $\leftrightarrow$ I	0.85394
O $\leftrightarrow$ P	0.85393

Table 5.5: Worst Ten 2-Key Swaps

Swap	WAR
A $\leftrightarrow$ K	0.85091
A $\leftrightarrow$ L	0.85095
A $\leftrightarrow$ J	0.85128
Q $\leftrightarrow$ T	0.85149
H $\leftrightarrow$ S	0.85160
G $\leftrightarrow$ S	0.85167
E $\leftrightarrow$ T	0.85169
T $\leftrightarrow$ W	0.85175
S $\leftrightarrow$ Y	0.85177
A $\leftrightarrow$ P	0.85177

Chapter 6

Conclusion

In this thesis we explored alternative keyboard layouts in hopes of finding one that increases the accuracy of text input on mobile touchscreen devices. We did so by carefully considering the placement of keys, exploring a specific vulnerability that occurs within a keyboard layout, namely, what we call the "Problem with Neighbors". We provided history and related work on the text input problem, specifically in regards to the use of MT keyboards. We described our approach of simulating the act of typing on an MT QWERTY keyboard, beginning with modeling the keyboard and the typographical errors that can occur. We then constructed a simple autocorrector using Bayesian methods, describing how we autocorrect user input to evaluate how well a keyboard corrects the input. Then, using our models, we provided a method of testing the accuracy of a keyboard layout, defining the overall WAR rating metric.

We ran our testing methods on the QWERTY layout as a baseline. We then measured the accuracy of all 325 2-key swap layouts, analyzing which layouts showed better performance than QWERTY. We showed that there exists more than one 2-key swap that increases the accuracy of the current QWERTY layout, and we provided the best 2-key swap derived from the QWERTY layout, $i \leftrightarrow t$, which was showed to increase accuracy by nearly 0.18 percent. We believe this swap would be a small, doable change, with a large payoff. The familiarity of QWERTY would not be jeopardized, as navigating a small change to QWERTY would be an easy adaptation. The preexisting software and hardware would also not be disrupted, but by a small degree.

There are several areas for future research:

1. **Improving our Error Model**: There are more complex models of typographical errors, particular when touchscreens are involved [8], that could be worked into our research to make things more robust. There have also been studies on the effect of movement [20] that could be integrated. A physical model of fingers could be utilized in our work, including size and orientation of individual fingers, especially the thumb, and which keys are obscured by fingers as they hover over the keyboard. These suggestions could expand our error model to make it more accurate.

2. **Improving our Autocorrection Model**: We constructed an autocorrector based on Bayesian principles. This model could be improved by using many of the various enhancements to autocorrectors that exist in the literature [39]. One suggestion would be to employ n-grams [10]. The autocorrector in this work could be improved by utilizing context and other predictive techniques.

Another obvious improvement would be to consider words farther than edit distance one away.

3. **Improving our Neighborhood Definition**: We used a grid definition for our keyboard layouts which is how we derived our neighborhood of keys, based on distance from one key to any other. There have been more specific approximations [7][6][65] and work done on distance between keys in research on touch pressure issues that could be incorporated to get an even more accurate grip on the neighborhood issues that exist.

4. **Improving our Language Model**: Wherever a language model was used within our work, we could have used a more varied or larger corpus. Specifically for our frequency map, which we used to model word frequency over the English language, getting a corpus of mobile text would be much more preferable. At the time of our research, mobile text data was not readily available due to privacy issues. The consequence of this is that our autocorrect is based more on written and spoken English, rather than mobile text. For mobile use, we would likely want to weigh words a bit differently (eg: the word "hi" or "hey" could be much more frequent over texting data than in written data such as transcriptions or even emails). Additionally, the WAR ratings we calculate are dependent on the size of the dictionary, so a larger corpus would result in a more precise accuracy measurement.

5. **Study Ethical Implications**: Any research that involves data from only a portion of the population needs to be investigated for adversely effecting that portion of the population. For example, the data we

used for our language model could be representative of only certain demographics of the overall population and therefore the benefits of our work may benefit one portion of the population while adversely effecting another. Thus, the ethical implications of our research should be investigated in future work.

Bibliography

[1] 8Pen, 2014. [Online; accessed 17-April-2018].

[2] S. Amendola, L. Bianchi, and G. Marrocco. Movement detection of human body segments: Passive radio-frequency identification and machine-learning technologies. *IEEE Antennas and Propagation Magazine*, 57(3):23–37, June 2015.

[3] anc American National Corpus. Oanc contents. [Online; accessed 23-April-2018].

[4] Sbastien Aupetit, Nicolas Monmarch, and Mohamed Slimane. Hidden markov models training by a particle swarm optimization algorithm. *J. Math. Model. Algorithms*, 6:175–193, June 2007.

[5] L. R. Bahl, F. Jelinek, and R. L. Mercer. A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(2):179–190, March 1983.

[6] Xiaojun Bi, Yang Li, and Shumin Zhai. Ffitts law: Modeling finger touch with fitts law. In *Proceedings of the SIGCHI Conference on Human*

Factors in Computing Systems (CHI 2013), pages 1363–1372, New York, NY, USA, 2013.

[7] Xiaojun Bi and Shumin Zhai. Bayesian touch - a statistic criterion of target selection with finger touch. In *Proceedings of UIST 2013 The ACM Symposium on User Interface Software and Technology*, pages 51–60, New York, NY, USA, 2013.

[8] Eric Brill and Robert C. Moore. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, ACL '00, pages 286–293, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[9] Peter F. Brown, John Cocke, Stephen A. Della Pietra, Vincent J. Della Pietra, Fredrick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roossin. A statistical approach to machine translation. *Comput. Linguist.*, 16(2):79–85, June 1990.

[10] Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, December 1992.

[11] C. J. C. Burges, J. C. Platt, and S. Jana. Extracting noise-robust features from audio data. In *2002 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages I–1021–I–1024, May 2002.

[12] Lee Butts and Andy Cockburn. An evaluation of mobile phone text input methods. In *Australian Computer Science Communications*, volume 24, pages 55–59. Australian Computer Society, Inc., 2002.

[13] Pew Research Center. Mobile fact sheet. Technical report, Pew Research Center, Washington, D.C., February 2018.

[14] Kenneth W. Church and William A. Gale. Probability scoring for spelling correction. *Statistics and Computing*, 1(2):93–103, December 1991.

[15] James Clawson, Alex Rudnick, Kent Lyons, and Thad Starner. Automatic whiteout: Discovery and correction of typographical errors in mobile text input, 2007. [Online; accessed 19-April-2018].

[16] A. Coates, B. Carpenter, C. Case, S. Satheesh, B. Suresh, T. Wang, D. J. Wu, and A. Y. Ng. Text detection and character recognition in scene images with unsupervised feature learning. In *2011 International Conference on Document Analysis and Recognition*, pages 440–445, September 2011.

[17] Colemak. Colemak keyboard layout, 2018. [Online; accessed 18-April-2018].

[18] Fred J. Damerau and Eric Mays. An examination of undetected typing errors. *Information Processing & Management*, 25(6):659 – 664, 1989.

[19] I. Essa, S. Basu, T. Darrell, and A. Pentland. Modeling, tracking and interactive animation of faces and heads//using input from video. In *Computer Animation '96. Proceedings*, pages 68–79, June 1996.

[20] Mayank Goel, Leah Findlater, and Jacob O. Wobbrock. Walktype: using accelerometer data to accomodate situational impairments in mobile touch screen text entry. In *CHI*, 2012.

[21] Joshua Goodman, Gina Venolia, Keith Steury, and Chauncey Parker. Language modeling for soft keyboards. In *Proceedings of the 7th International Conference on Intelligent User Interfaces*, IUI '02, pages 194–195, New York, NY, USA, 2002. ACM.

[22] D. Gopher and D. Raij. Typing with a two-hand chord keyboard: will the qwerty become obsolete? *IEEE Transactions on Systems, Man, and Cybernetics*, 18(4):601–609, July 1988.

[23] Niels Henze, Enrico Rukzio, and Susanne Boll. Observational and experimental investigation of typing behaviour using virtual keyboards on mobile devices. In *Conference: CHI Conference on Human Factors in Computing Systems, CHI '12*, May 2012.

[24] Alexander Hoffmann, Daniel Spelmezan, and Jan O. Borchers. Typeright: a keyboard with tactile error prevention. In *Conference: Proceedings of the 27th International Conference on Human Factors in Computing Systems*, pages 2265–2268, April 2009.

[25] J. Holmes and W. Holmes. *Speech Synthesis And Recognition*. Taylor & Francis, 2nd edition, 2001.

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

[27] ICT. Facts and figures: The world in 2015. Technical report, ICT, February 2016.

[28] SIL International. English wordlist. [Online; accessed 23-April-2018].

[29] D. Jurafsky and J.H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice Hall series in artificial intelligence. Pearson Prentice Hall, 2009.

[30] Robin Kinkead. Typing speed, keying rates, and optimal keyboard layouts. *Proceedings of the Human Factors Society Annual Meeting*, 19(2):159–161, 1975.

[31] Per Ola Kristensson, Stephen Brewster, James Clawson, Mark Dunlop, Leah Findlater, Poika Isokoski, Benot Martin, Antti Oulasvirta, Keith Vertanen, and Annalu Waller. Grand challenges in text entry. In *Extended Abstracts on Human Factors in Computing Systems*, CHI '13, pages 3315–3318, April 2013.

[32] Per Ola Kristensson and Shumin Zhai. Relaxing stylus typing precision by geometric pattern matching. In *International Conference on Intelligent User Interfaces, Proceedings IUI*, pages 151–158, January 2005.

[33] Karen Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, December 1992.

[34] Luis A. Leiva1, Alireza Sahami, Alejandro Catal, Niels Henze, and Albrecht Schmidt. Text entry on tiny qwerty soft keyboards. In *The 33rd Annual ACM Conference*, pages 669–678, April 2015.

[35] D. D. Lessenberry. Analysis of errors. Technical report, Smith and Corona Typewriters, Syracuse, New York, 1928. Reprinted in: Dvorak, Merrick, Dealy & Ford (1936).

[36] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, 10(8):707–710, February 1966.

[37] PCD Maltron Ltd. Maltron history, 2018. [Online; accessed 18-April-2018].

[38] Jos A. Macas, Antoni Granollers Saltiveri, and Pedro Miguel Latorre Andrs, editors. *New Trends on Human-Computer Interaction: Research, Development, New Tools and Methods*. Springer, 2009.

[39] Scott MacKenzie and William Soukoreff. Text entry for mobile computing: Models and methods, theory and practice. *Human-Computer Interaction*, 17:147–198, September 2002.

[40] Eric Mays, Fred J. Damerau, and Robert L. Mercer. Context based spelling correction. *Information Processing & Management*, 27(5):517 – 522, 1991.

[41] Scott McCartney. *ENIAC: The Triumphs and Tragedies of the World's First Computer*. Walker & Company, 1999.

[42] R. Morris and L. L. Cherry. Computer detection of typographical errors. *IEEE Transactions on Professional Communication*, PC-18(1):54–56, March 1975.

[43] Takao Nakagawa and Hidetake Uwano. Usability evaluation for software keyboard on high-performance mobile devices. In *HCI International 2011 Posters Extended Abstracts*, volume 173, pages 181–185, July 2011.

[44] R. A. Nelson and K. M. Lovitt. History of teletype development. Technical report, Teletype Corporation, 5555 West Touhy Avenue Skokie, Illinois, October 1963. [Online; accessed 17-April-2018].

[45] Saied B. Nesbat. A system for fast, full-text entry for small electronic devices. In *Proceedings of the 5th International Conference on Multimodal Interfaces*, ICMI '03, pages 4–11, New York, NY, USA, 2003. ACM.

[46] Donald Norman and Diane Fisher. Why alphabetic keyboards are not easy to use: Keyboard layout doesn't much matter. *Human Factors*, 24:509–519, January 1984.

[47] Peter Norvig Norvig.com. How to write a spelling corrector. [Online; accessed 23-April-2018].

[48] Sherry Ruan, Jacob Wobbrock, Kenny Liou, Andrew Ng, and James Landay. Speech is 3x faster than typing for english and mandarin text entry on mobile devices, August 2016.

[49] Dmitry Rudchenko, Tim Paek, and Eric Badger. Text text revolution: A game that improves text entry on mobile touchscreen keyboards. In *Pervasive Computing: 9th International Conference, Pervasive 2011, Proceedings*, pages 206–213, January 2011.

[50] S. Saha, B. Ganguly, and A. Konar. Gesture based improved human-computer interaction using microsoft's kinect sensor. In

2016 International Conference on Microelectronics, Computing and Communications (MicroCom), pages 1–6, January 2016.

[51] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, January 2001.

[52] Katie A. Siek, Yvonne Rogers, and Kay H. Connelly. Fat finger worries: How older and younger users physically interact with pdas. In *Human-Computer Interaction - INTERACT 2005*, pages 267–280, Berlin, Heidelberg, 2005.

[53] Brian A. Smith, Xiaojun Bi, and Shumin Zhai. Optimizing touchscreen keyboards for gesture typing. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3365–3374, New York, NY, USA, 2015. ACM.

[54] R. William Soukoreff and I. Scott MacKenzie. Towards a standard for pointing device evaluation, perspectives on 27 years of fitts law research in hci. *International Journal of Human-Computer Studies*, 61(6):751 – 789, 2004.

[55] Robert William Soukoreff. *Quantifying Text Entry Performance*. PhD thesis, York University, 2010.

[56] N. Stern. The binac: A case study in the history of technology. *Annals of the History of Computing*, 1(1):9–20, January 1979.

[57] Craig Stewart, Michael Rohs, Sven G. Kratz, and Georg Essl. Characteristics of pressure-based input for mobile devices. In *Conference*

on Human Factors in Computing Systems - Proceedings, volume 2, pages 801–810, January 2010.

[58] SwiftKey, 2018. [Online; accessed 17-April-2018].

[59] Thumbly, 2014. [Online; accessed 17-April-2018].

[60] Horabail Venkatagiri. Efficient keyboard layouts for sequential access in augmentative and alternative communication. *Augmentative and Alternative Communication*, 15(2):126–134, 1999.

[61] Horabail Venkatagiri. Efficient keyboard layouts for sequential access in augmentative and alternative communication. *Augmentative and Alternative Communication*, 15(2):126–134, 1999.

[62] Daryl Weir, Simon Rogers, Roderick Murray-Smith, and Markus Lchtefeld. A user-specific machine learning approach for improving touch accuracy on mobile devices. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 465–476, October 2012.

[63] Casey Whitelaw, Ben Hutchinson, Grace Y. Chung, and Gerard Ellis. Using the web for language independent spellchecking and autocorrection. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 2 - Volume 2*, EMNLP '09, pages 890–899, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[64] Wikipedia, The Free Encyclopedia. Qwerty. [Online; accessed 16-April-2018].

[65] Ying Yin, Tom Ouyang, Kurt Partridge, and Shumin Zhai. Making touchscreen keyboards adaptive to keys, hand postures, and individuals - a hierarchical spatial backoff model approach. In *Conference on Human Factors in Computing Systems - Proceedings*, pages 2775–2784, April 2013.

[66] Shumin Zhai, Per-Ola Kristensson, and Barton Smith. In search of effective text input interfaces for off the desktop computing. *Interacting with Computers*, 17:229–250, May 2005.

[67] Z. Zhang. Microsoft kinect sensor and its effect. *IEEE MultiMedia*, 19(2):4–10, February 2012.