Incremental Methods for Efficient and Personalized Text Entry in Noisy Input Environments

Andrew H. W. Fowler
M. A. Mathematics, Washington State University, 2006

Presented to the
Center for Spoken Language Understanding
within the Oregon Health & Science University
School of Medicine
in partial fulfillment of
the requirements for the degree
Doctor of Philosophy
in
Computer Science & Engineering

March 2020

Center for Spoken Language Understanding
School of Medicine
Oregon Health & Science University

CERTIFICATE OF APPROVAL

This is to certify that the Ph. D. dissertation of

Andrew H. W. Fowler

has been approved.

Steven Bedrick, Thesis Advisor
Associate Professor, OHSU

Brian Roark
Senior Staff Research Scientist, Google

Melanie Fried-Oken
Professor, OHSU

Peter Heeman
Associate Professor, OHSU

Xubo Song
Professor, OHSU

Shumin Zhai
Principal Scientist, Google

Acknowledgements

The two people most directly responsible for the success of my work at CSLU have been Steven Bedrick and Brian Roark, both of whom served as my advisors. I owe them each a deep well of thanks and gratitude for their wisdom, unfailing kindness, and perseverance. Their guidance has been a consistent reminder of the power of hard work and of asking the right questions. To my other committee members I also owe many thanks—to Melanie Fried-Oken for her clinical insights, to Peter Heeman for having my back on multiple occasions, to Xubo Song for graciously volunteering to join my committee on short notice, and to Shumin Zhai for his mentorship at Google and his deep expertise.

Many thanks are also due to the CSLU faculty and staff that have guided and helped me over the years, in particular Richard Sproat, Kyle Gorman, and Pat Dickerson. I also will miss the RSVP Keyboard team at OHSU and NEU, most notably Barry Oken, Betts Peters, Glory Noethe, Aimee Mooney, Deniz Erdoğmuş, Umut Orhan, and Greg Bieker (who made it all possible).

CSLU is a small program in a small graduate school but its students are of the highest quality. I will never forget my fellow graduate students, both current and former, for their friendship and esprit de corps. Of special significance to me have been Meysam Asgari, Russ Beckley, Nate Bodenstab, Shiran Dudy, Aaron Dunlop, Maider Lehr, Meg Mitchell, Masoud Rouhizadeh, Golnar Sheikhshab, Emily Tucker Prud'hommeaux, and Mahsa Yarmohammadi.

To Wally Tseng, Brice Tebbs, and Sabrina Burleigh I wish to offer my deep gratitude for allowing me to work full-time jobs while still finishing my PhD. There is no way I could have finished without their accommodation and kindness and I hope they realize how much it has mattered. To my wife's relatives I offer my thanks for reminding me every Christmas, Thanksgiving and Easter dinner just how excited they were that I was getting my PhD. In particular I want to give a very special thanks to Al Burmester for being my biggest cheerleader (and for a hundred cans of grape soda).

I want to thank my parents, who in the middle of the Great Recession said to me "why don't you go back to school?" and without whom I quite literally would not be here. Thank you to Peter and Ben for being the best brothers, and thank you to my kids for reminding me that sometimes going outside is more important than algorithms. Finally, I want to thank my wife Kaci for her

endless and unwavering support these many years. People act like having a job and kids while going to graduate school is some kind of juggling act, but it really isn't: It's about having somebody willing to cover for you, to sacrifice for you, to pick up the slack for you in thousand ways both large and small. Kaci has been that person for me, and for that I can only offer my love and gratitude.

Contents

List of Tables

List of Figures

Abstract

Incremental Methods for Efficient and Personalized Text Entry in Noisy Input Environments

Andrew H. W. Fowler

Doctor of Philosophy
Center for Spoken Language Understanding
within the Oregon Health & Science University
School of Medicine

March 2020
Thesis Advisor: Steven Bedrick

The global rise of text entry modalities that are not desktop QWERTY keyboards—and therefore exhibit significant uncertainty in input actions—has created enormous opportunities for language technology to improve the quality and efficiency of these tools. In addition to the near-total ubiquity of smart phone touchscreen technologies, specialized communication tools for people with disabilities have exhibited a similar growth in research and adoption. While language technology is known to be a crucial feature in these text entry tools, its specific application is not always deeply studied. In particular, the *incremental* aspect of many of these technologies—the fact that words are often entered letter-by-letter and cannot easily be revised by the system after input—is an important limitation of many such tools that is poorly explored in the research literature.

In this dissertation, I will explicitly examine the constraints and opportunities afforded by incrementality as they apply to language technology for text entry. In particular, I will explore user-directed text *revision and error correction* in these tools. By exploiting the fact that the *backspace* action allows for revision without violating incrementality, I develop and test two algorithms for accurately incorporating backspace into text entry. Both of these methods show large improvements over existing baselines, both in terms of raw text entry efficiency and in situations where text revision is required. Of particular interest is the application of these algorithms as elements of assistive technologies for people with disabilities, as these domains often represent situations where the speed of language input is extremely important. Further, I develop a system for testing language model personalization in text entry, a technology that is in wide use but has until now never been measured empirically. With these results, I hope to show that thinking about text entry in terms of its incrementality is not only a useful exercise, but can lead to quantitatively superior algorithms and methods, particularly when text revision or user personalization is needed.

Chapter 1

Introduction

Incremental processing can loosely be defined as any algorithmic task which produces discrete, low-latency output directly corresponding to an input data stream. It is a topic with a rich body of literature, yet only represents a small proportion of research in Natural Language Processing (NLP) applications. This discrepancy is primarily because incrementality is not a required design attribute of most NLP tasks. Parsing, for instance, typically takes an entire sentence as its input, and incremental parses of partial sentences are rarely required. Machine Translation (MT) presents a similar scenario: The input is usually a full sentence or utterance, and incremental translations are not needed in most applications. The same can be said for Automatic Speech Recognition (ASR), where latency in the transcription output is large enough to allow a less strict decoding modality that explores many paths through a potential lattice of possible outputs.

Text entry is a notable exception to this rule. Because input of letters and words in a text entry environment is typically expected to occur similarly to handwriting or keyboard typing, there is little room for latency. Once a text entry system commits to a letter and outputs it into a typed string, re-evaluation of that hypothesis is less straightforward than it is in other input modalities. This property is particularly notable in text entry applications for Augmentative and Alternative Communication (AAC). Due to the often severe physical limitations of the users of such systems, typing throughput can be very slow, increasing the importance of immediate and incremental output.

Incremental algorithms represent an interesting trade-off between latency and performance. Since looking ahead into the input stream is not possible, one not only has less data to work with, but one is required to act sooner on that data. This places a high burden on the current best hypothesis being correct as often as possible, making high-performance algorithms more difficult to implement. Incrementality requires modifying pipeline strategies (one processing step, then another, etc.) as well. Every step in a pipeline must not only *itself* be incremental, but must be configured to run simultaneously and in parallel with each of the other steps. Interesting questions can also be explored if one wishes to *abandon* strict incrementality in rare cases, such as allowing a user to revise a typed string.

This thesis seeks to address three problems in the existing research: 1) Text entry literature typically fails to explicitly recognize the unique challenges and opportunities posed by incrementality. 2) Error correction strategies such as backspace are rarely empirically examined or even considered in the text entry literature or the AAC literature. 3) Even with state-of-the-art algorithms, *evaluation* of text entry methods can be difficult, because truly extrinsic system evaluation requires that user input be *noisy*, which is not trivial on a large scale.

1.1 Research Objectives

The goal of this thesis is to explore three techniques for improved text entry, each addressing one or more of the problems listed above.

The first technique is called Full History Fusion (FHF). This method is used in text entry scenarios where the input signal is noisy, and each symbol input event can be expressed as a probability distribution over the characters in an alphabet. This occurs in the Rapid Serial Visual Presentation (RSVP) paradigm, for instance. In this paradigm, a language model is combined with the character input probabilities to give a final posterior probability distribution over characters. However, since the input is noisy, errors in *output* will inevitably occur. This poses a challenge, because the only way to correct such an error is to revise the output, technically a violation of incrementality. One solution is to transfer the burden of revision to the user, in the form a *backspace* key. This re-frames the decision to revise as a type of *input*, making it more natural in a text entry setting and maintaining its incremental nature. Unfortunately, including backspace as a possible input creates a new question: What is its probability? Certainly the language model does not assign probability mass to backspace. FHF seeks to solve this problem by calculating, dynamically in a principled way, the posterior probability of the backspace key. Using the full history of noisy input data as its input, FHF allows for for better performance on backspace actions as well as on other typed characters.

The second technique considers the backspace key more directly. By re-framing backspace as just another symbol in an input stream, it is possible to consider backspace prediction as a sequence modeling problem. Accurately predicting the probability of a backspace event can then be accomplished by a range of sequence modeling techniques. In particular, we attempt to demonstrate that Long Short-Term Memory (LSTM) models are not only capable of accurately predicting backspace events, but also learning some of the basic *semantics* of backspace. These semantics include common-sense reasoning, such as the fact that a just-deleted letter should have lower probability than it did before backspace was used to delete it.

The third technique seeks to understand the actual impact of language model *adaptation and personalization* for text entry. While language model adaptation is a well-established topic in language modeling, its benefits are difficult to evaluate. In particular, improvements in word-level accuracy are hard to measure. Since human text entry trials are impractical at large scales, one of the only ways to empirically study the potential benefits of adaptation is through simulation. This requires the simulation of noisy text entry, the decoding of that signal into symbols, and a means to evaluate the resulting output. Since information-theoretic measures like perplexity do not always correlate with commensurate (or any) improvements in actual performance, using simulation to compute word error rate is crucial for evaluating text entry. The adaptation method used for this work is simple linear interpolation of adapted and background language models. Importantly, the adapted language model changes with each new word typed, itself a form of incrementality.

By exploring these three techniques this thesis accomplishes the following aims:

Aim 1: Full History Fusion Implement FHF for Rapid Serial Visual Presentation Keyboard$^{\text{TM}}$. Evaluate Brain Computer Interface (BCI) application using simulation. Evaluate button-press application with human subjects.

Aim 2: Neural Modeling of Backspace Explore the possibility of backspace prediction as a sequence modeling problem, using neural networks. Compare to a traditional ngram sequence model. Compare to FHF under simulated RSVP trials.

Aim 3: Simulated Noisy Text Entry For Evaluation of LM Personalization Evaluate LM personalization using large-scale text entry simulation. Determine the impact of out-of-vocabulary (OOV) words in personalization performance.

1.2 Contributions

This thesis contributes significantly to the existing body of research, both in terms of algorithmic development and empirical evaluation. While incremental methods have been studied before, the arguments for doing so are typically based on computation speed (incremental parsing tends to be faster), very specific domain requirements (e.g. dialogue systems), or parallels to biological processes (human sentence processing is largely incremental). Text entry provides a more concrete reason to study incrementality, not only because it is usually implemented incrementally, but because text entry is itself ubiquitous (smartphones outnumber humans in some countries). By focusing on broad applications of language models in text entry, this thesis represents a large contribution to the literature, in the following concrete ways:

- Development of the Full History Fusion method, the first algorithm for calculation of posterior probabilities in the RSVP protocol that utilizes all past user evidence *and* allows for dynamic backspace probability
- Concrete results, from both a simulation of a BCI environment and from human trials, showing significant empirical gains in text entry performance when FHF is implemented
- The first analysis of the backspace symbol as a sequence modeling problem; in particular the application of neural networks to the prediction of dynamic backspace probabilities, and a demonstration that neural networks can "learn" the semantics of the backspace action
- The first large-scale demonstration of the quantitative benefits of language model personalization in text entry, measured via extrinsic evaluation in a large simulation experiment over a significant real-world corpus

The contributions of this work to AAC are particularly appealing, as this work has the potential to significantly improve typing speed and accuracy in a domain where speed can be crucial.

1.3 Organization of the Thesis

In Chapter 2, I will describe the current state of the literature as it relates to incremental processing, in particular when applied to language models for text entry and AAC. Chapter 3 will describe, in greater technical detail, technologies particularly applicable to this thesis. Chapter 4 will be devoted to the algorithmic description of the Full History Fusion technique, and arguments for its theoretical validity. Chapters 5 and 6 will describe the results of two major experiments testing the relative performance of FHF compared to more simple baselines. The first is a large-scale simulation experiment of RSVP typing in a BCI setting. The second is a study conducted with human subjects using the RSVP typing paradigm under a noisy button-press input modality. Chapter 7 will examine the possibility of predicting the backspace key as part of a sequence modeling problem, comparing various traditional and neural network-based techniques for doing so. Chapter 8 will cover language model adaptation and personalization for text entry, including a large simulated experiment showing the empirical benefits of adaptation when applied to personalized text corpora. I will conclude with a chapter summarizing the concrete contributions of this thesis and outlining possible future work.

Chapter 2

General Background

Incremental methods have been studied across a wide range of language-related technologies and fields. In this chapter, I will give an overview of the research that has been done in this regard, and describe various methods in particular that are important context to my work. This is not intended to be an exhaustive survey of all related work, rather a sampling of the kinds of questions that are being asked about incrementality and how they connect to this thesis.

2.1 Incremental Methods & Analysis

Incremental language processing consumes input in a letter-by-letter (or word-by-word) manner and generates increasingly complete output [Beuck et al., 2011]. In the strictest sense, the output structure cannot be revised, only added to. In addition, the k^{th} output must correspond in some way with the k^{th} input segment[1] As such, there is no latency allowed in the processing (under a strict definition of incrementality). Look-ahead is similarly disallowed; only the single next unit of input in the sequence can be processed at any given time.

2.1.1 Psycholinguistics

Psycholinguistics is one area with significant research into incremental processing, particularly incremental parsing. A primary driver of this research is the hypothesis that the human brain processes sentences in a largely incremental manner (though the extent to which this is true is the subject of significant debate). Psycholinguists attempting to model human sentence parsing are therefore very interested in incremental parsing models. The question of how this parsing occurs, given the cognitive limits of the brain, is a noteworthy one. Abney and Johnson [1991]

[1] It is possible, for the output and input to consist of a different number of constituent parts, e.g. a machine translation where the input and output sentence have a different number of words. However, this can still be accomplished by allowing (for instance) the possibility of null outputs and multi-part outputs (though such techniques do violate incrementality in the strictest sense).

approaches the problem by examining the relationship between local ambiguities and short-term memory limits in the brain. [Sturt and Lombardo, 2005] uses eye gaze analysis during sentence reading to suggest that models of human parsing involving bottom-up methods alone are insufficient, as certain embedded structures more amenable to tree-adjoining grammars are evident from the processing disruptions evident in the eye gaze. Another active area of research is the extent to which a human anticipates or predicts upcoming words, a strategy which is certainly allowed even in a strict incremental context. Kamide et al. [2003] examines this prediction phenomenon by using eye gaze data to explore whether thematic dependencies can be evaluated *before* the linguistic data is observed that unambiguously signals such a dependency. They find that this is indeed the case in studies of English and Japanese. Van Berkum et al. [2005] studied anticipatory effects by looking for an event-related potential (ERP) in a brain scan, postulating that an unexpected upcoming word would elicit this event. They found evidence that certain words had been anticipated by observing an absence of ERP in certain contexts. This relates to the surprise aspect of ERP as described in the description of RSVP Keyboard in Section 3.2. Language *production* can also be modeled incrementally. Ferreira and Swets [2002] looked into this phenomenon using an experiment in which subjects were asked to produce utterances that required arithmetic calculations. They found that subjects under a timed deadline exhibited differences in latency and duration dependent on the difficulty of the math problem. This suggests possible incrementality in language production, as it means that speakers sometimes speak and plan simultaneously. Although some of the work studying incrementality in psycholinguistics uses brain scans, and it obviously is focused on language, it is distinct from my work in that the incremental effects of text entry (and text revision) are not covered in this literature.

2.1.2 Parsing and Tagging in NLP

Non-incremental dynamic programming algorithms have been extensively studied in NLP. Methods like CYK [Cocke and Schwartz, 1970, Younger, 1967, Kasami, 1965], a syntactic parsing algorithm, and the Viterbi algorithm [Viterbi, 1967], used to find likely sequence of hidden states given a set of observed events, are executed only after the entire sentence is known. There has nonetheless been significant work on incremental analysis, particularly with regard to parsing. A parser which does not "back up" (i.e. amend output) during its analysis is called a *deterministic parser*[2]. A classic example is LR parsers [Knuth, 1965], which are useful for compiling programming languages. LR

[2] It is important to avoid confusion here with the concept of a *deterministic context-free grammar (DCFG)*, which is a related but distinct idea. Indeed, practically any interesting stochastic grammar representing natural language is *not* a DCFG (though it can be used for deterministic parsing).

parsers are effective for deterministic context-free languages and operate in linear time. The ambiguity of natural language precludes the direct use of LR parsers, but strategies for incremental parsing have long been studied in NLP. The GLR algorithm of Lang [1974] and the PARSIFAL system of Marcus [1980] were early attempts to algorithmically tackle the problem of deterministic parsing of natural language. GLR is related to LR parser, but it executes a breadth-first search to process all interpretation of a given input. It then uses a stack mechanism to fork in transitions where conflicting interpretations are possible. Although it is deterministic and incremental, its worst case time complexity is $\mathcal{O}(n^3)$. PARSIFAL uses a combination of a stack (for incomplete constituents) and a constituent buffer (for complete constituents of unknown higher-level grammatical function). It also violates strict incrementality by using a three-word look-ahead.

More recent work on incremental parsing often gives other arguments for the benefits of incrementality. Abney [1990] cites parsing speed as a significant reason to pursue deterministic parsing, since incremental methods are typically less computationally complex. Collins and Roark [2004] use incremental parsing as a testbed for discriminative modeling (though the use of beam search makes the method not strictly incremental).

Dependency parsing is another area in which incremental methods have been successful. In contrast to constituency parsing, which aims to construct a parse tree based on a grammar that subdivides a sentence into a phrase structure, dependency parsing relies instead on a grammatical intuition that words are connected to each other by directed links. These links have their basis in the finite verb of the sentence. Dependency grammars were first studied in detail by Lucien Tesnière, who argued for their usefulness [Tesnière, 1959]. The reason dependency parsing lends itself to incremental processing has been the introduction of so-called *transition-based* parsing methods, which decompose parsing into a sequence of simple moves, each building a part of the parse structure. These algorithms proceed through a sentence in an incremental fashion, not requiring that the entire sentence be known before parsing can begin. This allows for the possibility of fast, incremental, deterministic parsing.

Transition-based dependency parsers, including *shift-reduce* parsers [Nivre, 2003, Yamada and Matsumoto, 2003], have produced competitive results at much faster computational speeds than the algorithms underpinning constituency parsing methods. Although the current state of the art dependency parsing algorithm is not incremental [Mrini et al., 2019], very nearly comparable results have been obtained under the strict incrementality constraints of transition-based methods [Fernández-González and Gómez-Rodríguez, 2019]. That paper uses pointer networks in a transition-based setting. Pointer networks are a neural models that yield a succession of pointers to elements in the input series, similar to attention-based methods (see Section 7.2.2). Although

Nivre [2004] points out that incremental dependency parsing has certain theoretical limits, these appear to have not yet been reached.

Part-of-speech (POS) tagging is often performed as a preprocessing step to parsing and other algorithms. However, in a truly incremental environment such a pipeline approach is impossible; one sees the input once and only once. This poses a significant problem for parsing, as accurate POS tags are extremely important features in the development of competitive parsing algorithms. Unfortunately, unlike standard POS tagging, which boasts state-of-the-art English language accuracy of nearly 98% [Bohnet et al., 2018], *incremental* POS tagging is not a solved problem, and Beuck et al. [2011] discusses three properties of incrementality that can be violated in specific ways: *timeliness*, *monotonicity*, and *decisiveness*. Timeliness requires that each an input be fully processed before the next one. Monotonicity is the requirement that the output can only be augmented, never modified later on. Decisiveness requires that an algorithm commit to one analysis at a time. One potential solution is to model POS tags and dependency parses *jointly*, which has been accomplished with some success [Hatori et al., 2011, Bohnet and Nivre, 2012].

The connection of these parsing and tagging papers to this thesis is notable insofar as these papers demonstrate some of the trade-offs involved with incrementality (such as speed vs. performance), and illustrate that incrementality is not an insurmountable system requirement.

2.1.3 Dialogue Systems & Speech Recognition

Automatic speech recognition (ASR) typically has relatively loose incrementality requirements, as the output is usually not required or expected by the user until after each sentence or utterance is completed. One notable exception is in dialogue systems. Dialogue systems are unlike other ASR applications in that partial transcriptions of an utterance are significantly more useful, since these transcriptions can be used to inform or invoke various actions (or reactions) in the dialogue system. As such, so-called Incremental Speech Recognition (ISR) may be necessary. Selfridge et al. [2011] examines the utility of incomplete speech recognition results in a dialogue setting, noting the possible instability of these outputs and proposing a method for determining, on the fly, whether such an intermediate result is stable or not. Baumann et al. [2009] takes a similar approach, developing a set of measures for determining the relative value of an incomplete ASR hypothesis, and looking at the timeliness trade-off by examining downstream results under varying amounts of latency. Even without considering speech input, dialogue systems are inherently incremental at least in part; no system can wait until a dialogue is complete before processing it. This makes these systems fruitful ground for research into incremental algorithms. Schlangen and Skantze [2009] lays out a general conceptual topological framework for specifying incremental architectures, while Skantze and Hjalmarsson [2010] looked at incremental processing from the standpoint of user perception, comparing incremental and non-incremental methods. The latter paper showed that their incremental method produced longer utterances but with a shorter response time; users ultimately perceived this combination as a comparative increase in system efficiency. Finally, Baumann et al. [2017] lays out a particularly compelling analysis of the special requirement of incrementality in dialogue systems and how it affects the technology.

2.1.4 Machine Translation

Some research into incremental methods for statistical machine translation (MT) has been done, and the arguments for it are varied. Hassan et al. [2009] strives mainly for processing speed, and uses an incremental combinatory categorial grammar (CCG) parser to provide fast, linear-time decoding for translation. Sankaran et al. [2010] describes the benefits of incremental MT in specific applications: interactive language learning, real-time speech-speech translation, and translation during interactive chats. Yarmohammadi et al. [2013] attempts to model the incremental behavior of human translators, who use various strategies to maintain low latency during translation.

2.2 General Language Model Adaptation

The essential problem that LM adaptation seeks to solve is the problem of *domain mismatch.* LMs can be trained on enormous text corpora, but those corpora rarely match the domain of the target text one wishes to model. In-domain text is usually small in quantity and not enough to build a robust model. LM adaptation algorithms attempt to combine the benefits of a large, out-of-domain model with a small in-domain model. *Personalization* can be seen as the ultimate adaptation problem, since one is adapting to a single individual (for which in-domain text is almost certainly limited). LM adaptation has been studied extensively. An excellent review of various algorithms can be found in Bellegarda [2004] and DeMori and Federico [1999].

Cache-based LM adaptation keeps a continuously-updated cache of words, and use this cache to build the adaptation/personalization model. In the context of model adaptation, the model generated from the cache is the in-domain model. A cache varies slightly from other adaptation techniques in two ways: First, the adaptation model must be updated on the fly because new words are typed continuously (the technique is commonly called dynamic cache-based adaptation to reflect this fact). Second, the user cache may begin in a completely empty state, or be primed with a small amount of in-domain text. Cache-based LM adaptation was used by Kuhn and De Mori [1990] to improve the performance of a speech recognition system.

Cache-based LMs are further explored in Clarkson and Robinson [1997], which describes a method in which the relative weights of words in the cache are made to decay exponentially, with "older" words having lower weights. This reflects the common-sense notion of *recency*, which suggests that, in general, more recently-used words in a text will re-occur with higher probability than words used in the more distant past. The results of Clarkson and Robinson [1997] indicate that using an exponentially-decaying cache results in improvements in perplexity, but as shown in Chapter 8, extrinsic evaluation of text entry appears to indicate that these gains may be negligible in certain applications.

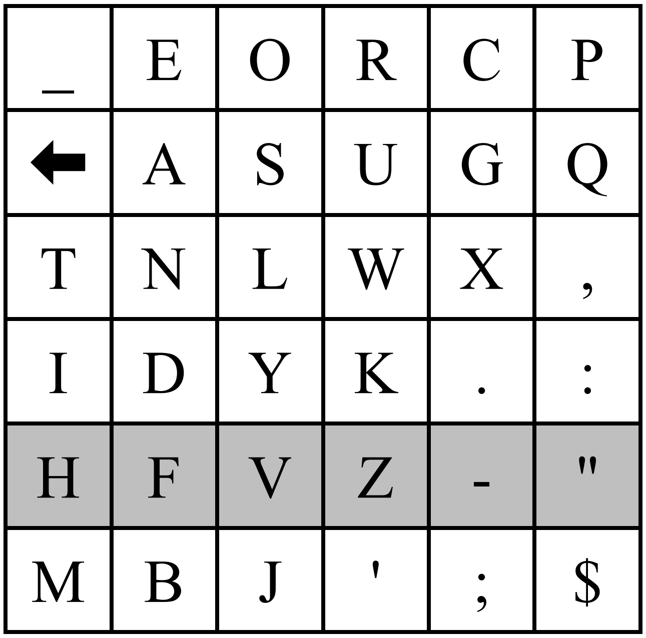
There are two principle shortcomings of this previous work. First, the adaptation is not used for personalization, i.e. adapting to an individual. Second, the evaluation methods are typically model-intrinsic (e.g. perplexity) and not focused on text entry. This means that the true benefits of these methods in text entry are difficult to ascertain. My LM adaptation work [Fowler et al., 2015] addressed both of these shortcomings by simulating many personalized models in a true simulated text entry task (see Chapter 8).

2.3 Interfaces for AAC Text Entry

Text entry is a central component to many AAC technologies. Among the most useful and widely used methods of facilitating text entry in this domain are scanning interfaces and brain computer interfaces (BCI).

2.3.1 Scanning Interfaces

Scanning interfaces typically feature a grid of symbols that are highlighted in a specific order in order to determine which symbol the user wishes to type. The symbols in question are typically the letters of an alphabet or some superset thereof. These are also known as binary typing systems, because the user selects a symbol or group of symbols by actuating a binary switch (physically or through a brain interface). Typically the binary choice in binary typing is between *select* and *do not select*. Furthermore, *select* is typically accomplished via an active intention and *do not select* is accomplished passively, i.e. absence of selection. Figure 2.1 shows a simple row/column scanning grid, in which more probable symbols are located in the upper left corner (and can therefore be selected more quickly). Lesher et al. [1998] outlines multiple variants on scanning interfaces, including different ways of selecting, modifying and rearranging symbols. Roark et al. [2010] investigates other scanning options, in particular those which incorporate language models into the selection algorithm. Lesher and Rinkus [2002a] show the benefit of adding possible predicted words



_	E	O	R	C	P
⬅	A	S	U	G	Q
T	N	L	W	X	,
I	D	Y	K	.	:
H	F	V	Z	-	"
M	B	J	'	;	$

Figure 2.1: A row/column scanning interface with the fifth row selected.

to a scanned grid, particularly when the selection of those words is informed by a language model. Beckley and Roark [2011] and Roark et al. [2013a] describe a sophisticated form of scanning called *Huffman scanning*, in which Huffman codes [Huffman, 1952] are used to create theoretically optimal symbol subsets during selection. Notably, scanning methods can be (and often are) accomplished without any computer interface whatsoever. This is done by human communication partners or assistants, who are trained to ask a series of binary questions that verbally correspond to a scanning protocol.

2.3.2 Brain Computer Interfaces

Brain computer interfaces can be used in AAC for text entry when the user has a serious disability that precludes or inhibits voluntary physical movement. One such disability is locked-in-syndrome, a condition in which the thoughts and consciousness of a person are completely intact but their ability to move is largely or entirely absent. While BCI technology is still in its infancy, it is now possible to collect certain brain signals and transform them into external actions via carefully constructed interfaces. The principal BCI method associated with this thesis is RSVP Keyboard^{TM} [Orhan et al., 2012], a typing system that uses electroencephalogram (EEG) signals to detect recognition in the brain of target symbols presented in a rapid letter sequence to the user. The initialism RSVP stands for Rapid Serial Visual Presentation, a technique related to scanning interfaces in which the scanning proceeds very quickly one character at a time rather than across groups of characters. RSVP Keyboard also makes use of a character-based language model to improve its performance (see Chapter 3 for technical details). The specific brain signal, or event-related potential (ERP), that indicates symbol selection is known as the P300 signal, due to its average time delay of 300 milliseconds after stimulus onset. Use of the P300 for spelling originates with Farwell and Donchin [1988], in which the user focuses on their target letter in a grid and the system flashes rows and columns in such a way as to elicit a P300 response. Kübler et al. [2001] and Wolpaw et al. [2002] provide overviews of BCI techniques. A notable contribution of my work absent from these BCI-based typing systems is the specific focus on text revision as a first-class system attribute, in particular my principled consideration of the backspace key and its role as an incremental input.

Rapid Serial Visual Presentation itself has been a technique used in psycholinguistics since at least Kolers and Katzman [1966], who studied the ability of subjects to identify rapidly presented letters and their position within a sequence (outside of a BCI context). It is also used with non-linguistic targets, as in Matran-Fernandez et al. [2013], where the RSVP protocol was used in a BCI setting to perform image classification (in this case in a collaborative setting, where multiple subjects were asked to look for aerial photographs containing an airplane).

2.4 Language Models for Text Entry

Language models have long been used to improve text entry. MacKenzie and Soukoreff [2002b] describe the general theory of these methods for mobile devices, for instance. Further, modeling language is of particular interest in AAC applications. One of the reasons for this is that communication with AAC devices can be significantly slower than written or spoken language, or communication using text entry modalities outside of assistive technology. As a result, both users and designers of AAC technologies often place great value on improving communication speed and accuracy. Predictive language models are a natural fit for this problem in two ways: First, they can retrieve likely continuations of a message in a way that requires fewer gestures (keystrokes) to select than if they were produced letter-by-letter. Second, they can help to assign probability mass to inputs in noisy environments, bringing to bear statistics about words and letters that can be useful in making better interpretations of user input.

The simplest LM is a lexicon, i.e. a list of permissible words. A lexicon can already provide quite powerful constraints in, for example, decoding ambiguous 12-button-based dialpad input found on traditional phones, which dates back to the early 1970's [Smith and Goodwin, 1971a]. Since entering logographic text such as Chinese with a Roman alphabet is highly ambiguous, applying some form of model that maps Roman letter-based phonetic input to logographic characters has also been necessary and common in East Asian countries for a long time [Tanaka-Ishii, 2007].

One way to leverage language models is *word completion*, which is widely used and studied in text entry. Stocky et al. [2004] describes a "commonsense" approach to word completion to be used in contrast with statistical approaches. van den Bosch and Bogers [2008] introduces two methods, one involving a character buffer containing recent keystrokes and one that includes the previous word as an input feature. Li and Hirst [2005] seeks to augment ngram-based methods by incorporating semantic information. This is accomplished by using pointwise mutual information to create a list of semantically related words, then using this information to better select possible word completions given a specific context. Word completion is also a component of almost every commercial AAC typing system in use today, and this has been the case since the early days of such systems [Higginbotham et al., 2012].

More mathematically rigorous word completion involves using an ngram language model trained on a large corpus to help predict the next word (or current word in progress) based on the last few words of context in the typed string. Very often a specific area of the screen is reserved specifically for a set of probable next words, which can be selected directly by the user in order to type one of them without taking time on the intervening letters, or selected via scanning that region in a keyboard emulation system. Higginbotham [1992] showed that such word completion lists can provide about 40-50% keystroke savings[3] over systems without word completion. One important caveat of word completion is that more cognitive effort is required on the part of the user to keep track of dynamically changing word predictions. Koester and Levine [1996] demonstrated this fact using a text entry task performed by both able-bodied and spinal cord injured subjects, finding that word completion involves such a great cognitive overhead that it actually decreases user performance somewhat in such systems. Quinn and Zhai [2016] examined this effect in the context of mobile text entry, finding similarly that typing speed on soft keyboards decreases when word suggestions are provided, even when only highly-probable suggestions are shown.

Another way to use language models in text entry is with *error correction*, a term which describes a variety of methods for counteracting input noise and/or ambiguity. Error correction is the subject of various text entry papers, ranging from early work with alphanumeric keypads [Smith and Goodwin, 1971b], to gesture input techniques [Zhai and Kristensson, 2003, Kristensson and

[3] Keystroke savings is the percentage savings in keystrokes versus typing letter-by-letter; see Section 3.1.3 for details.

Zhai, 2004], to more theoretical investigations of the trade-offs involved in error correction [Kano et al., 2007, Bi et al., 2014b]. Goodman and colleagues [Goodman et al., 2002] drew inspiration from speech recognition for stylus keyboard tapping and applied a character ngram model to correct stylus taps, achieving a reduction in *character-level* error rate from 2.40% to 1.39%. Klarlund and Riley [2003] applied ngram language modeling to a cluster keyboard (i.e. a keyboard with reduced number of keys), and showed large improvements over unigram frequency models.

Language modeling can also be used to modify the interface itself, in particular in binary switch scanning systems of the sort described in Section 2.3. For row/column scanning methods, likely symbols can be moved to the upper left corner of the grid so that they require less scanning to select. The TIC system cited above [Crochetiere et al., 1974, Foulds et al., 1975] organized the grid based on overall symbol frequency so that the grid remained static. Lesher and Rinkus [2002b] described a technique for dynamically rearranging the grid based on contextual language models, but showed that the large cognitive load involved in keeping track of an ever-changing letter grid made this method dispreferred.

While statistical rigor has been applied to text entry language models, and text revision has been considered as a design attribute, it is notable that these two have not been combined, considering the importance of user-directed error correction in text entry. Merging these two halves is an important contribution of this thesis. Further, framing text revision as an incremental input activity external to the typing of individual letters (and potentially subject to the interaction cost trade-offs found in Koester and Levine [1996] and Quinn and Zhai [2016]) allows it to be examined empirically as an alternative to these other methods known to create large cognitive overhead.

2.4.1 Model Adaptation in Text Entry

Adaptation techniques are the topic of a small body of research in the text entry field. The most applicable paper to my thesis is Tanaka-Ishii [2007], which explores various language model adaptation techniques for text entry specifically. Harbusch et al. [2003] investigates the extent to which language domain affects text entry. Harbusch and Kühn [2003] attempts to adapt the behavior of a user interface based on user actions. Weir et al. [2014] seeks to adapt to the individual touch patterns on a mobile device, while Azenkot and Zhai [2012] examines the feasibility of adapting to different hand postures on mobile devices. Wandmacher and Antoine [2006] is a preliminary investigation of adaptation for AAC. Adhikary et al. [2019] contributes more to this topic, investigating the possibility of using ASR to recognize speech of the non-AAC user and leveraging this transcript into an adapted language model. Dasher, a gesture-based text entry system developed by Ward et al. [2000], contains a simple language model based on prediction by partial match (PPM). PPM is a text compression algorithm [Bell et al., 1990], which can be used to determine letter probabilities given context. Notably, Dasher further allows for user-typed text to be fed back into the algorithm, allowing for a level of adaptation to be achieved. Dasher was further investigated empirically by Rough et al. [2014], who replaced its PPM-based language model with a more standard ngram-based model and evaluated its performance on human subjects (though they did abandon the adaptive functionality of the LM for these experiments).

The aformentioned gesture-typing interfaces [Zhai and Kristensson, 2003, 2012] also leverage language regularities in the form of a lexicon that could further be adapted to the individual user by extracting a vocabulary from the user's past documents [Kristensson and Zhai, 2004], or by moving passive words into an in-use active vocabulary [Kristensson and Zhai, 2008].

Unfortunately, the papers cited here represent an exception to the rule. Model adaptation in text entry is typically done in a relatively rudimentary fashion, both for mobile text entry and for AAC. Most systems simply keep a list of words recently entered by the user, and use various heuristic methods to make those words easier to type (either by adding them to a suggestion list or boosting their probability). Empirically investigating the more principled approach, of using established NLP language model adaptation, is a meaningful contribution of my work.

Chapter 3

Technical Background

In this chapter I will explore in greater detail some of the existing techniques, models, and algorithms that are particularly central to my work. Some, like language modeling, are essential building blocks for the technologies I introduce in this thesis. Others, like the RSVP Keyboard, are existing technologies that I hope to augment with the methods I have developed in this work.

3.1 Language Modeling

A language model, as it is usually formulated, is a means by which one can estimate the probability of a given sequence of words or characters. Language models have wide-ranging applications across Natural Language Processing (NLP), including parsing, machine translation, document classification, sentiment analysis, and information retrieval. Knowing the probability of a word sequence may not be of high utility in and of itself, but a good language model allows for the *comparison* of competing hypothesis in the space of utterances, which is an incredibly useful and important aspect within a multitude of language technologies.

As described in Section 2.4, language models have long been used in text entry and AAC applications. The ability to give more weight to likely word strings is crucial in these domains, as it allows one to overcome the noise inherent to many such input modalities by bringing to bear prior knowledge of language itself. This can significantly improve both the accuracy of these systems as well as the speed with which users can generate text [Trnka et al., 2007].

3.1.1 ngram Language Models

A widely used construct in language modeling is the ngram language model [Jurafsky and Martin, 2008]. In this formulation, we compute the probability of a string not *in toto* but by chaining together a series of conditional probability values. Underlying ngram language models is the concept of an ngram. An ngram is a contiguous sequence of words or characters of a specific length, for instance “gravy boat” or “greetings”. In an ngram language model, we seek to calculate the probability of some word or character coming next in the context of a specific prefix, for instance $P(\text{hat} \mid \text{the cat in the})$, which can be interpreted as ”the probability of the word *hat* following the sequence *the cat in the*”. More technically, we want to compute the probability of a given word or symbol occurring at a specific position in an utterance, conditioned on a prefix of a specific length:

$$P(w_n \mid w_1^{n-1}) \tag{3.1}$$

Here, we wish to know the probability of some word/character w_n given its prefix w_1^{n-1}. The combined sequence is an ngram of length n. Provided we can compute this conditional probability, we can now compute the probability of any given sequence of characters, using the chain rule:

$$P(w_1^n) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1^2)...P(w_n \mid w_1^{n-1}) \tag{3.2}$$

Unfortunately, computing the constituent values in Equation 3.2 is not always possible. For example, a long utterance might require computation of the conditional probability of a word given a very long prefix context. If we seek to calculate this probability by observing ngrams in the wild, we will likely encounter a situation where the prefix sequence has never been attested, even in a very large corpus. We can solve this problem by making a *Markov assumption.* The Markov assumption, in an ngram language model, is to estimate the right side of Equation 3.2 by only considering prefixes up to a certain length $k - 1$. The value k is called the *order* of an ngram language model. Under this simplifying assumption, we get the following:

$$P(w_1^n) \approx P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1^2)...P(w_{n-1} \mid w_{n-k}^{n-2})P(w_n \mid w_{n-k+1}^{n-1}) \tag{3.3}$$

Interestingly, although the Markov assumption allows us to compute conditional probabilities from data in a way that is less sparse, it turns out that even low-order ngram models often require the calculation of probabilities where the prefix (or its completion in a word w_n) has rarely or never been observed in data. For instance, suppose we wish to know $P(\text{stegosaurus} \mid \text{toothpaste})$. It may be that this bigram[1] has never been used by anybody in the history of recorded language. Yet it is incorrect to estimate this conditional probability simply by counting how often the word *toothpaste* appears in some corpus and counting how often that word is followed by the word *stegosaurus.* In such a simplistic model, one could take the quotient of these values to get a conditional probability (this is a form of Maximum Likelihood Estimation (MLE)). In this example we would get a conditional probability of zero. However, it stands to reason that this bigram (or

[1] A *bigram* is an ngram of length 2, in this case “toothpaste stegosaurus”.

any ngram) is not *impossible* to write or utter; it should be afforded some nonzero probability mass in any useful language model. This is particularly important when one considers the chain rule formulation itself. Under MLE, any string of characters with a substring of length k that does not appear in the training corpus will be given a probability of zero.

The solution to this problem is to abandon MLE for more sophisticated ways of computing conditional probabilities. As the goal is to make the probability distribution of a model less peaked, these techniques are called *smoothing* methods. Multiple ngram smoothing methods exist. One such technique is *Laplace smoothing*, in which we count ngrams and take quotients as we would in an MLE model, but we add a constant value to all counts, even zero counts. This ensures that all ngrams are given some probability. Another technique is *Kneser-Ney smoothing* [Kneser and Ney, 1995], in which words that appear as continuations in a greater number of ngrams are afforded more probability mass in the smoothing process. An improved version known as *modified Kneser-Ney smoothing* is described in Chen and Goodman [1999]. Witten-Bell smoothing [Witten and Bell, 1991], which is used as the character language model for many parts of this thesis, similarly considers counts of counts, adjusting probability mass based on the diversity of ngrams containing given words.

3.1.2 Neural Language Models

Over the past several years, neural network-based language models have begun to surpass the performance of traditional ngram models. In particular (and of particular interest to this thesis), the Recurrent Neural Network (RNN) architecture has been used to create models that perform extremely well on language modeling tasks.

An RNN model is a neural network that takes as its input a sequence of vectors $\mathbf{x_1}...\mathbf{x_n}$ and outputs a vector sequence $\mathbf{y_1}...\mathbf{y_n}$. When used for language modeling, the input vectors correspond to n words/characters used as input to a language model, and the output vectors correspond to predictions about possible next symbols. For instance, output vector $\mathbf{y_j}$ can be interpreted (under certain formulations) as a probability distribution over words/characters given the input vector sequence $\mathbf{x_1}...\mathbf{x_j}$. The output vectors typically contain $|V|$ entries, each corresponding to a word or character in the LM alphabet/vocabulary.

There is also an initial state vector $\mathbf{s_0}$ and a sequence of state vectors $\mathbf{s_1}...\mathbf{s_n}$, the latter of which are intermediate outputs of the RNN. These state vectors have two purposes: First, the state vector $\mathbf{s_j}$ is used to compute the output vector $\mathbf{y_j}$. Second, each successive state vector is fed back into the neural network cell as the string of inputs is consumed by the model. This recursion is what gives RNNs their name, and also their power. Ultimately an RNN is a deep neural network, capable in theory of maintaining information from any point in the input. This means that it is no longer subject to the Markov assumption described in Section 3.1.1.

Borrowing notation from Goldberg [2017], an RNN is defined by

$$RNN(\mathbf{s_0}, \mathbf{x_{1:n}}) = \mathbf{s_{i:n}}, \mathbf{y_{i:n}}, \tag{3.4}$$

where $\mathbf{x_{1:n}}$ is the input vector sequence, $\mathbf{y_{1:n}}$ is the output vector sequence, and $\mathbf{s_{1:n}}$ is the state vector sequence. Further, we describe the computation of the output and state vectors using the functions R and O:

$$\begin{aligned} \mathbf{s_i} &= R(\mathbf{s_{i-1}}, \mathbf{x_i}) \\ \mathbf{y_i} &= O(\mathbf{s_i}) \end{aligned} \tag{3.5}$$

The exact calculation of R and O depends on the particular RNN formulation being used. In the basic RNN formulation known as Simple RNN [Elman, 1990, Mikolov, 2012], they are defined as follows (again borrowing notation from Goldberg [2017]):

$$\begin{aligned} R(\mathbf{s_{i-1}}, \mathbf{x_i}) &= g(\mathbf{x_i}\mathbf{W^x} + \mathbf{s_{i-1}}\mathbf{W^s} + \mathbf{b}) \\ O(\mathbf{s_i}) &= \mathbf{s_i} \end{aligned} \tag{3.6}$$

Here, g is a non-linear activation function like $tanh$, $\mathbf{b}$ is a bias vector, and $\mathbf{W^x}$ and $\mathbf{W^s}$ are weight matrices used to perform linear combinations on the input and state vectors, respectively. The dimensions of these matrices (and $\mathbf{b}$) are selected so that the sum inside of the g function consists of vectors having the same dimension as the state vectors.

One shortcoming of Simple RNN is the vanishing gradients problem, which can cause useful information in the input sequence to be gradually lost in the state vectors of the RNN. Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997] is an attempt to address this problem. In LSTM, the basic idea is to include a more sophisticated set of state vectors, as well as a more involved R function within each recurrent cell. These state vectors are designed specifically to act as "memory" of previous states, and a complex series of "gate" operations within the R function serve to maintain useful state memory and protect it from vanishing. LSTM models have been shown to outperform RNNs in many applications.

The input vectors $\mathbf{x_1}...\mathbf{x_n}$ used in RNN-based word LMs are typically embeddings of the actual word inputs. An embedding is a real-valued vector representing a discrete value like a word, projected into a space of lower dimensionality. Embeddings have the nice property that vector arithmetic can reveal useful semantic relationships between the original words. Further, embeddings allow for RNN models with fewer parameters. For character-based RNN LMs, embeddings are less useful, as the dimensionality of the inputs is relatively small, on the order of the size of the alphabet as opposed to the size of a vocabulary.

Stacked RNNs are RNNs for which the state vector $\mathbf{s_i}$ is not transformed immediately into an output vector $\mathbf{y_i}$ but instead is fed into the i^{th} RNN cell in another "layer" of cells, as if it were an input vector like $\mathbf{x_i}$. In this formulation, only the state vectors in the top layer are used to compute output. The rest are fed as input into their respective next layers. The *depth* of a stacked RNN refers to the number of layers it has, which is typically 2 or 3 in practice.

Because there are many parameters that must be estimated in the weight matrices and vectors of an RNN, they must be trained carefully. The standard method for doing this is Backpropagation Through Time (BPTT) [Werbos, 1990]. In the case of an RNN used as an LM, we compute a local model loss value after each output vector is generated, then sum these losses at the end of the sequence and use that combined loss function during backpropagation to compute gradients and train the model parameters.

Like traditional ngram models, RNN-based LMs can suffer from overtraining. There are various ways of combating this problem. Dropout, developed by Hinton et al. [2012], is one such method. In this technique, some proportion of model weights are set to zero during training. This prevents the model from over-relying on certain weights, and also prevents correlated features from overpowering other dimensions. In stacked RNNs, dropout is used between the model layers and between model cells. In the LSTM architecture, dropout is avoided between cells, as it can interfere with the memory aspect of LSTM. A typical dropout proportion value is 0.5. Another regularization technique is early stopping, where a model being trained is evaluated on a small development set at regular intervals, and training is halted when that evaluation begins to degrade and does not recover.

3.1.3 Evaluation

Language models, and the systems in which they are used, can be judged by various evaluation techniques. *Intrinsic* evaluation metrics are typically calculated by analyzing the model itself, or its probabilistic performance on a piece of data. *Extrinsic* evaluations consider the performance of the model as it contributes to the completion of the task for which it is intended to be used. Extrinsic evaluations are typically more useful than than intrinsic ones.

Language Model Evaluation

Intrinsic evaluation of language models is typically done with the *perplexity* metric. Perplexity is an information-theoretic metric of the quality of a probabilistic model. To compute perplexity, one needs a language model and a test set $T = w_1 w_2 w_3 ... w_m$. The test set is a sequence of words/characters never seen during creation of the model. The general formulation of perplexity for a given probabilistic language model P is as follows:

$$\begin{aligned} PP(T) &= P(T)^{-\frac{1}{m}} \\ &= \sqrt[m]{P(w_1 w_2 w_3 ... w_m)} \\ &\approx \sqrt[m]{\prod_{i=1}^{m} P(w_i | w_{i-k+1}^{i-1})} \end{aligned} \tag{3.7}$$

Note the use of the chain rule and the ngram Markov assumption in the last line here.

The value output by the perplexity can be interpreted in comparison with a uniform probability distribution. A perplexity of 6, for example, means that the model is comparable (in assigning probabilities) to a sequence prediction task where there are 6 equally likely symbols.

Since character-based language models are used extensively in this thesis, the *bits-per-character* (BPC) metric is also used. This metric, also from information theory, represents the performance of a statistical model using the number of bits that would be required, on average, per character, to encode the test set. It is more common when evaluating character-based models. Its relationship to perplexity is very simple:

$$BPC(T) = \log_2 PP(T) \tag{3.8}$$

Text Entry Evaluation

Keystroke savings and word error rate are two common evaluations used in text entry research. Notably, both of these metrics are extrinsic, as they are computed based on a downstream task. Keystroke savings is a measure of *efficiency*, i.e. how much faster one can enter text given a specific system. Word error rate (WER) is a measure of *accuracy*. Keystroke savings is defined in Equation 3.9. Note that "taps" is more applicable in a mobile text entry context than in an AAC context, but any discrete input action can be considered a "tap."

$$ks = 1 - \frac{total\ taps\ required}{total\ characters\ in\ final\ output} \tag{3.9}$$

MacKenzie and Soukoreff [2003] establish a widely-used phrase set for consistency of evaluation across text entry experiments. Vertanen and Kristensson [2014] describe methods for evaluating *free text entry*, in which a user is allowed to type whatever they wish. Trnka and McCoy [2008] explore the theoretical pitfalls of keystroke savings as a metric. Higginbotham [1992] evaluates the typing efficiency of five different AAC systems.

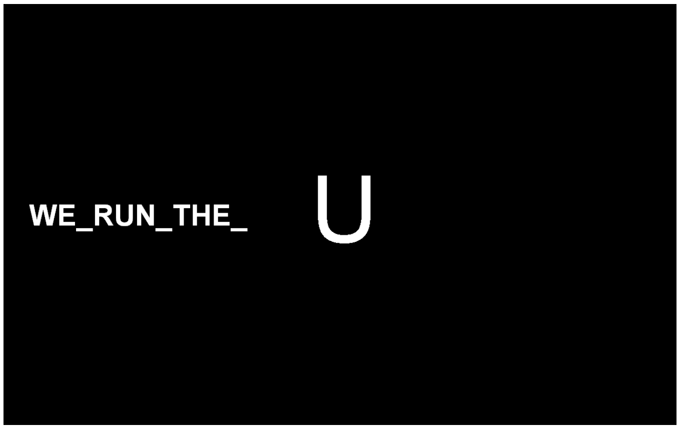


Figure 3.1: The RSVP Keyboard$^{\text{TM}}$interface, with the candidate symbol in the center of the screen, and typed message to its left.

3.2 RSVP Keyboard

The RSVP Keyboard$^{\text{TM}}$BCI typing system [Orhan et al., 2012] combines a language model prior distribution over symbols with repeated EEG observations to facilitate accurate typing. It is a key underlying technology in this work, and will be used as the basis for Chapters 4, 5 and 6. However, it is important to note that the RSVP Keyboard described in this thesis, and used as a baseline for many experiments herein, represents a basic version of the system as it existed in 2012. Improvements and augmentations made subsequently are not included in this work.

Symbol presentation follows a rapid serial visual presentation (RSVP) paradigm, which involves linear visual scanning of one symbol at a time. Figure 3.1 shows the interface of the system, with a single symbol in the center of the display rather than a grid. To perform text entry, one attends to a rapid sequence of individual symbols at the center of the screen, looking for the target symbol. When the user recognizes the target symbol an event related potential (ERP) is evoked – the well-known P300, occurring 300ms after stimulus onset – which can be detected from the EEG signals. This detection is made via classifier fusion with a language model: when the posterior probability of an event associated with a symbol is greater than a threshold, that symbol is typed; otherwise another sequence of EEG observations is obtained.

The primary input tool for RSVP Keyboard is an EEG cap of 10-20 wet electrodes for measuring brain signals on the scalp. The symbol set consists of 26 letters in the alphabet plus a space bar symbol (“_”) and a backspace symbol (“←”) that permits deletion of the previously typed symbol. All 28 symbols are presented in sequence in random order at 2.5 Hz, i.e., 400 ms on screen display for each symbol.[2] After the full sequence of symbols has been presented, a classifier is used to detect the presence of a P300 event, under the assumption that one such event has occurred (corresponding to the target symbol). A posterior probability is calculated, which combines, via naïve Bayesian fusion, the probability from each sequence presentation with the prior probability from the language model (see details below). If the posterior probability of any symbol is above a parameterized threshold value, then that symbol is typed.[3] Otherwise, another sequence of the symbol set is presented in random order.[4] Once a symbol is typed (or deleted), the display and internal context state is updated and scanning continues at the next position.

Beyond EEG signal capture and stimulus display, the RSVP Keyboard system consists of three key parts: the ERP classifier; the language model; and the fusion of the evidence for decision on the target symbol. In addition, there are several system meta-parameters that control functionality. We present each of these in turn.

3.2.1 ERP Classification

The input to the ERP classifier is a set of features extracted from the EEG signals. For each stimulus in the sequence (28 symbols), the signals are extracted from the onset of the symbol presentation until 500ms after presentation, since this window is expected to include the relevant signal components associated with the P300 ERP. Bandpass filters and linear dimensionality reduction methods (PCA) are applied to derive a feature vector to be given to the binary classifier (target/non-target).

Regularized Discriminant Analysis (RDA) [Friedman, 1989] is used to generate a likelihoods for both classes (target/non-target) for each of the 28 stimuli in the sequence. RDA is a modified version of a quadratic discriminant analysis model which is made less susceptible to singularities in the covariance matrices via regularization and shrinkage. The class feature vectors are assumed to conform to multivariate normal distributions. This model can be used to generate classification scores (via log-likelihood ratios) or a probability distribution over stimuli. For more details, see Orhan et al. [2012].

For this work, the ERP classifier remains unchanged from the current RSVP Keyboard system. For simplicity of later notation, let $ERP_{ij}(x)$ be the likelihood from sequence presentation j that x is the target symbol at position i in the typed string, as determined by the ERP classifier. Then,

[2] Presenting *subsets* of the full set of symbols is also possible (See Chapter 6) but this is not done within the standard RSVP Keyboard system.

[3] The backspace ("←") symbol is not "typed" per se, rather it causes the last typed letter to be deleted.

[4] Note that the randomness of the sequence is important, since the strength of the P300 response is partially based on the unpredictability/surprise on the part of the user when the target stimulus appears.

after k sequences have been presented at position i, $ERP_i^k(x)$ is the likelihood of symbol x at position i,

$$ERP_i^k(x) \quad = \quad \prod_{j=1}^{k} ERP_{ij}(x) \tag{3.10}$$

3.2.2 Language Modeling

The language model used in the RSVP Keyboard is a 6-gram character-based language model, i.e., the probability of symbol x at position i (denoted x_i) is conditioned on the previous 5 symbols. Let $h_i^{i-k} = x_{i-k} \ldots x_{i-1}$ be the context (or history) of length k of the typed string at position i. Then

$$\begin{aligned} \mathrm{P}(x_i \mid h_i^{i-k}) \quad &= \quad \lambda(h_i^{i-k})\ \widehat{\mathrm{P}}(x_i \mid h_i^{i-k}) \\ &\quad + \quad (1-\lambda(h_i^{i-k}))\ \mathrm{P}(x_i \mid h_i^{i-k+1}) \end{aligned} \tag{3.11}$$

where $\widehat{\mathrm{P}}(x_i \mid h_i^{i-k})$ is the raw relative frequency as found in the training corpus, and $\lambda(h_i^{i-k})$ is a mixing parameter between 0 and 1, estimated using the version of Witten-Bell smoothing [Witten and Bell, 1991] from Carpenter [2005a], as detailed in Roark et al. [2013b].

For the current RSVP Keyboard system, and unchanged in this work, the language model was trained on a subset of the New York Times portion of the English Gigaword corpus (LDC2007T07). Also included in the corpus were 112 thousand words from the CMU Pronouncing Dictionary,[5] and several repeated lines of text generated by an actual user of AAC technology. The training data was text-normalized to de-case letters, and several other procedures were performed in order to reduce the possibility of duplicate sentences or articles appearing in the corpus, as presented in Roark et al. [2013b].

Each symbol is scored by the language model using a likelihood ratio of the symbol being the target versus the symbol not being the target, and then normalized over the symbol set. Since this language model prior is not being changed for this work, for notational simplicity let $LM_i(x)$ be the prior that symbol x is the target symbol at position i, given what has already been typed.

3.2.3 Language Model and EEG Fusion

At position i in the typed string, a posterior probability P of each symbol x in the symbol set can be calculated via Bayesian fusion of the language model and ERP classifier scores, as follows

$$P_i^j(x) \quad = \quad ERP_i^j(x) \cdot LM_i(x) \tag{3.12}$$

[5] `www.speech.cs.cmu.edu/cgi-bin/cmudict`

Note that $P_i^0(x) = LM_i(x)$, and that

$$P_i^j(x) \quad = \quad P_i^{j-1} \cdot ERP_{ij}(x) \tag{3.13}$$

Once the normalized posterior $P_i^k(x)$ is greater than the decision threshold, the symbol x is typed.

3.2.4 System meta-parameters

The RSVP Keyboard relies upon several meta-parameters for operation. In this section, we introduce each of these parameters, and the settings in the current version of the system.

- **Decision threshold**: If the posterior probability of a symbol is greater than the decision threshold θ, then that symbol is typed. In the current system, that threshold is set at $\theta = 0.9$.
- **Minimum Sequences**: The minimum number of sequences to show before typing. In the current system, this parameter is set to 1, meaning that at least one sequence must be presented to the user prior to typing a symbol. If this parameter is set to 0, it means that if $P_i^0(x) > \theta$ then the symbol will be typed based solely on evidence from the language model (autotyping).
- **Maximum Sequences**: The maximum number of sequences to show before typing. If this number is reached and no symbol exceeds the decision threshold θ, the most probable symbol is typed. The current system sets this to 3.
- **Backspace Probability**: The current system uses a fixed probability $\beta = 0.05$ of the backspace symbol in every context.
- **LM Damping**: This is a damping factor used on the language model probability distribution. A value of λ means we multiply log probabilities from the language model by λ. In the current system, $\lambda = 0.5$.

Chapter 4

Full History Fusion

Full History Fusion is an augmentation of the functionality of RSVP Keyboard that seeks to address some of its shortcomings. From the standpoint of incremental processing, FHF allows for dynamic text revision in a fully incremental environment, extending the usefulness of the backspace key in RSVP Keyboard by calculating its probability in a more principled manner. This increases the validity of RSVP Keyboard as a test case for incremental methods, particularly in the AAC domain. In this chapter I will describe the motivations for the creation of Full History Fusion and the mathematics underlying its algorithm.

4.1 RSVP Keyboard Failures

As the RSVP Keyboard system was developed and began to be tested with true EEG signals as input, users of the technology began to experience certain failure modes that caused difficulty in the typing process. These failure modes were intermittent and inconsistent but did recur across multiple users.

Two of these failure modes are germane to this thesis: First, in some cases the system would begin to type incorrect letters in such a way that the user was unable to correct them. In this failure mode, the user repeatedly attempts to select the backspace key but its posterior probability never rises above the typing threshold. As such, multiple incorrect characters are typed and the user is unable to return to their intended typed message. I will refer to this as the *runaway typing problem.* The second failure mode occurs when the system types a single incorrect character, then the user deletes this character, then the system types that same error again, trapping the user in an infinite loop from which they cannot proceed. I will refer to this as the *repeated error problem.*

Further analysis tends to suggest that the runaway typing problem occurs when the language model becomes “overconfident” in a particular string of characters and creates a scenario in which evidence from the EEG[1] classifier is insufficient to overcome this overconfidence. There are a handful of possible reasons this can happen. It could be that the LM damping meta-parameter λ described in Subsection 3.2.4 may be have a value that is too high. Recall that this parameter serves to regularize the language model probabilities by attenuating peaks in its distribution and boosting low-probability characters. This is akin to LM smoothing in a very coarse sense, the argument being that an un-dampened model may be overtrained and fail to generalize to new data (or fail to work well when combined with EEG evidence). Another possibility is that runaway typing occurs when the fixed backspace probability β (set to 0.05 in vanilla RSVP Keyboard) is not high enough to handle situations where a strongly LM-favored character string is actually not the intended one. It may be that the EEG evidence for backspace is simply not strong enough to overcome the language model and meet the typing threshold, making backspace impossible to type. Finally, it could be a misconfiguration of one of the other RSVP Keyboard meta-parameters. For example, a low setting for the maximum sequences parameter might allow the language model to commit to a low-probability best guess if the EEG evidence is weak but the sequence total has been reached.

The repeated error problem is related to the runaway typing problem, but its cause is actually more subtle. In this failure mode, an errorful character is typed, then deleted, then typed again, ad infinitum. This turns out to be caused by a shortcoming of the RSVP Keyboard protocol itself. In particular, it has to do with which evidence is used by RSVP Keyboard when calculating the posterior probability of a letter. In this particular case, we have some letter L that has been typed in error (usually due to a strong LM prior) and then correctly deleted by the user, who has selected the backspace symbol. At this point, RSVP Keyboard sees that the typed sequence has changed and re-queries the language model for character probabilities. It then proceeds to collect a new set of EEG observations. Crucially, the algorithm fails to take into account two things: 1) the EEG observations that were made in this same text position previously, and 2) the fact that the letter L has just been deleted. This causes it to make the same error it made before, perpetuating the infinite loop.

4.2 Formulating a Solution

These failure modes made RSVP Keyboard an occasionally sub-optimal system, a flaw that was amplified by the fact that it was intended for use in AAC, a domain in which communication

[1] In this chapter I use the initialism EEG to refer to the output of a brain signal classifier, but (as I will show in Chapter 6) this evidence can come from any source that produces a probability distribution over characters, not just an electroencephalogram.

throughput is particularly prized. Fortunately, the root causes of these failures seemed to fall into a small set of categories:

1. **Incorrect Meta-Parameter Settings** The default RSVP Keyboard settings appeared to be partially responsible for both major typing failure modes, with evidence implicating practically all of the parameters.

2. **Fixed Backspace** The use of a fixed probability value for the backspace key (which had a default value that been chosen somewhat heuristically) appeared to be a possible contributing cause for the runaway typing failure. In addition, it stands to reason that the probability of backspace is not actually fixed in real scenarios. For example, if the posterior probability of the most recently typed symbol is very high, it makes sense for the backspace symbol to be low at the beginning of the next typing step (and vice-versa).

3. **Constraints to EEG Evidence** Since evidence collected during EEG sequence presentations could not persist beyond the particular typing step in which they occurred, the system was subject to failures like the repeated error problem. In a long typing session, RSVP Keyboard uses (at a given decision point) only a fraction of the EEG evidence it has collected. Further, system calculations *after the typing of a backspace* did not take into account that backspace, nor was the identity of the deleted character itself considered.

Full History Fusion is intended to address all three of these root causes, thereby solving both the runaway typing problem and the repeated error problem. It additionally is intended to be more principled in its calculation of character probabilities, with the goal of improving typing speed in general over that of RSVP Keyboard.

There are two key insights within FHF. The first is to collect and utilize the *full history* of EEG observations made during a typing session, as opposed to only those collected during the current typing step. The second is to consider not the probability of individual characters but the probability of *strings* of characters.

In the RSVP Keyboard protocol, once a letter is typed (or deleted), the system advances to the new position and reconsiders the 28 possible options (26 letters, space and backspace), without taking into account previous events. In FHF, once a possible string has been considered, its posterior probability is maintained even if another symbol is typed. The typed symbol at position i in the string generally has a posterior probability less than 1.0, meaning that other options retain some probability mass. Use of the backspace symbol as a deleting mechanism allows the system to go back and potentially access those alternate strings, hence we maintain their non-zero posterior probability. In fact, if we accrue additional evidence for the backspace symbol through EEG detection, that probability mass should be allocated to these alternate strings, since it is evidence that the current string is not the intended string but rather one of the alternates.

For instance, suppose the user has typed the string `cat`. The full history of EEG observations at this point will include data from *all* sequences presented to the user at each typing epoch, including at the beginning of the typing session and after each subsequent letter. FHF allows us to combine all of these observations, incorporate the language model probabilities, and calculate the probability of any string of characters. This allows for the calculation of various values. For instance, the conditional probability $P(cats|cat)$ will help us to calculate the probability of the character `s` in the current context. Further, FHF is aware of the special semantics of the backspace symbol. If `cat` has been typed and a strong EEG signal is collected suggesting that backspace is the next character, not only will FHF use this evidence to make the rightmost `t` less probable, but it will shrink the probability of every letter in `cat`, as a backspace is evidence that the word itself may be an error. Still further, that backspace will slightly *increase* the probability of strings completely different from `cat`, such as `dog` and `oldsmobile`, a phenomenon that not only informs the probability of backspace itself, but also can prove useful if `cat` is deleted and one of those alternates turns out to be the correct string intended by the user.

4.3 Details of FHF

4.3.1 Definitions

s	a string of characters
s^i	the prefix substring of s, consisting of its first i characters
s^0	the prefix substring of length zero, i.e. the empty string
s_k	the k^{th} character of s
$\|s\|$	the length of the string s
$s\phi$	the string s concatenated with the character ϕ
M_s	the set of matching strings (all strings that have s as a prefix)
N_s	the set of non-matching strings (all strings *without* s as a prefix)

4.3.2 Combining Evidence from LM and EEG

Full History Fusion is accomplished by factoring in all EEG observations in a typing session, not just those in the current epoch. The fusion consists of three components: LM evidence, positive EEG evidence, and negative EEG evidence. In FHF, each of these gives a probability/score for a *string* rather than a *character*. We define them as follows:

LM evidence gives the probability of a string s according to the language model. This can be decomposed into a product of conditional probabilities using the chain rule[2] It is defined as

$$P_{LM}(s) = \prod_{i=1}^{|s|} P_{LM}(s_i|s^{i-1}). \tag{4.1}$$

Positive EEG evidence is a score for a string s derived from all EEG observations that are consistent with the characters in that string. By *consistent* I mean direct probabilistic evidence collected for (or against) letters that are actually contained in the string. As with the LM evidence, we use the chain rule. It is defined as

$$L_{EEG^+}(s) = \prod_{i=1}^{|s|} L(s_i|s^{i-1}), \tag{4.2}$$

where $L(s_i|s^{i-1})$ is the *product* of all EEG classifier probabilities for character s_i when the typed string on the screen at the time was s^{i-1}. Note that, depending how the typing session has proceeded, there may be zero[3], one, or many such scores.

Negative EEG evidence is a score for a string s derived from all EEG observations of the *backspace* symbol that occurred on *other* strings. For instance, suppose that at some point, when a string not matching s was the typed string, EEG evidence was collected that indicated a high probability of backspace. Since this is evidence that a non-matching string could be incorrect, it should *increase* the probability of *all other strings*, including s. Conversely, low probability of the backspace action in a non-matching context should *decrease* the probability of all other strings (and s), because it is evidence that the current context is correct. Negative EEG evidence is defined as

$$L_{EEG^-}(s) = \prod_{t \in N_s} L(\leftarrow |t), \tag{4.3}$$

where $L(\leftarrow |t)$ is the product of all EEG classifier probabilities for the backspace symbol when the typed string on the screen at the time was t.

The final posterior probability Q for a string s is proportional to the product of these three scores:

$$Q(s) \propto P_{LM}(s) L_{EEG^+}(s) L_{EEG^-}(s) \tag{4.4}$$

It is important to note that positive EEG evidence for a string s does not necessarily mean that s is more probable, nor does negative EEG evidence mean that s is less probable. These notations simply refer to the letter probabilities used to calculate their respective values.

[2] Since this language model is an ngram model, the chain rule will give an approximation when $n < |s|$. This is a standard simplifying assumption in language modeling.

[3] By convention, the product of zero items is equal to one.

4.3.3 Deriving Conditional Posterior Probabilities

Now that we can calculate the posterior probability of any string, it is straightforward to generate the probability of a single character in a given typed context. If the currently-typed string is p, the conditional probability of the next character being ϕ is calculated by marginalizing over all strings having a prefix that matches p and that have the character ϕ in the next position:

$$Q(\phi|p) \propto \sum_{t \in M_{p\phi}} Q(t) \tag{4.5}$$

Backspace is a special case, requiring a slightly different formulation. In FHF, the posterior probability of backspace is calculated by marginalizing over all strings that *do not* match p:

$$Q(\leftarrow|p) \propto \sum_{t \in N_p} Q(t) \tag{4.6}$$

This is a very central insight within Full History Fusion. By marginalizing over non-matching strings, we leverage the fact that the only way to “get to” those strings in an incremental setting like RSVP Keyboard is to type the backspace key first. Therefore every non-matching string contributes probability mass to the probability of a backspace action. A very useful consequence of this formulation is that the probability of backspace will therefore be dynamic, depending on p and the EEG evidence collected so far. Note that when $|p| = 0$, N_p will be the empty set. A convenient consequence of this is that the backspace symbol will have no probability if there are no typed characters.

4.3.4 Practical Considerations for Computation

In practice, a good way to keep track of the scores required for the FHF algorithm is to keep a table with a row for each potential string, and store the LM and EEG scores corresponding to that string in the columns of the table. It is also useful to use logarithms so that we can take products using summation.

Unfortunately, the equations presented in the previous subsection are not immediately amenable to actual calculation. The main problem is that the sets M_s and N_s are (usually) infinitely large. This would make our score table infinitely large. One way to get around this issue is to only consider strings up to some maximum length. A useful maximum length is the length of the longest typed string that has been observed so far in the current session (plus one). This makes the sets M_s and N_s finite, without sacrificing any accuracy. The reason accuracy is maintained is that all EEG evidence consists of observations made with typed strings shorter than this maximum length, so we are still considering all EEG evidence with each fusion calculation.

However, even when we consider only strings below a certain length, it still results in a score table that increases in size exponentially as that length increases. This creates the risk of an algorithm that cannot actually be computed. But there are other optimizations to be made. The calculation space can be further decreased in size by observing that certain “branches” of the possibly-typed string need not be fully explored *if we have never typed anything on that branch.* For instance, if no string beginning with the letter `B` has been typed in the current session, our table of scores need not contain all strings beginning with `B`. Instead, we can keep a single row that itself represents *all strings beginning with* `B`, and only split that row into more rows if the letter `B` becomes the typed string at some time in the future. The fusion calculation is still valid, because P_{LM}(‘`B`’) correctly represents the probability of all strings beginning with `B`, and there will be no EEG evidence for any string starting with `B` (other than that of `B` itself) since we never actually typed it. Using this shortcut, I have been able to prevent the size of the scores table from growing exponentially[4].

4.4 FHF as an Algorithm

FHF can also be understood as an algorithm, which I present formally in Figure 4.1, and describe here less formally, with references to specific lines in the algorithm. To copy a given text T using a symbol vocabulary V, we provide three metaparameters: a threshold parameter θ to determine the required probability of a symbol to type it; and a minimum and maximum number of iterations of sequence presentation to gather ERP evidence for the symbols, as discussed earlier. Note that the set V includes every symbol that will be typed, i.e., not backspace, which is a special symbol denoting that a previously typed symbol should be deleted. In our case, V includes the letters `a-z` plus whitespace.

The algorithm maintains a set of context strings S, initialized with the empty string (with probability 1) on line 2 of the CopyText algorithm. The set S is the set of strings that are one symbol extensions of strings that have actually been typed at some point in the session. Each of the strings in the set is maintained with a posterior probability by the algorithm. Until the input text T is correctly typed, we must decide at each point which of the 28 symbols to choose. If each member of V is given an index from 1 to $|V|$, and we let 0 be the index of the backspace symbol, we can establish the prior probability for each of these in the current context. Scanning through all strings s (with associated probabilities p) in the set S (lines 6-15 of the CopyText algorithm in Figure 4.1), there are three possible cases for each stored string: 1) the typed string t is not a

[4] This is not proven, but appears true across many simulations.

prefix of s (line 7), in which case the posterior probability p of that string is allocated to backspace (index 0); 2) the string s is exactly the typed string t (line 9), in which case s has never been expanded before and must be expanded (see below); or 3) the string s is longer than t (line 14), meaning that the posterior probability p of s should be added to that of whatever symbol from V comes after the prefix t in s. For case 2, the string s is removed from the set S, and the set of strings resulting from concatenating each symbol $x \in V$ to s are added to S, with the appropriate probability as assigned by the language model. Finally, these prior probabilities are normalized for use by the LM and ERP fusion decision process.

The algorithm UpdSymbolProbs at the top of Figure 4.1 takes as input the symbol vocabulary, the probability array calculated as described above, the position in the string and the relevant meta-parameters, and returns an updated (normalized) posterior probability array and an ERP classifier-based likelihood array $\mathcal{L}$, which does not include the prior probabilities. After the symbol sequences have been presented the requisite number of times, the strings in set S are scanned again, to update their posterior probabilities with the new information (lines 18-22 of the CopyText algorithm in Figure 4.1). For strings s which have t as a prefix, the posterior probability is updated with the relevant symbol's likelihood at that position from the recent round of symbol sequence presentations. Otherwise, the posterior probability is updated with the likelihood of the backspace symbol. Finally, the posterior probabilities of all strings in the set are normalized.

After the posterior probabilities have been calculated for all strings in the set, the system detected symbol is determined. If that symbol is backspace, then the typed string t is shortened by removing its last symbol. Otherwise, the detected symbol is appended to t.

For this work, I extended the above algorithm by pruning extremely unlikely strings from the set S. Any string with a probability less than e^{-30} was removed from S, resulting in increased efficiency and no typing performance difference.

4.4.1 Worked Example

For this example, we will assume an alphabet of two symbols (`A` and `B`), and our observations from the EEG will be of the form

$$\text{ERP observation } j = [\mathcal{L}(\texttt{A}), \mathcal{L}(\texttt{B}), \mathcal{L}(\texttt{backspace})],$$

where $\mathcal{L}(x)$ is the normalized probability that the next symbol is x based on the EEG classifier. Table 4.1 presents the probabilities used to maintain the normalized posterior for each context string in the set S. For this example, $\theta = 0.8$ will be the decision threshold.

We begin with no letters currently typed, and no observations made. The first step is to query

typed	context	LM	ERP observation			normalized
string	string	prob.	1	2	3	posterior
ϵ	A	0.4	0.2			0.14
	B	0.6	0.8			0.86
Action:	type B					
B	A	0.4	0.2	0.1		0.03
	BA	0.4	0.8	0.7		0.85
	BB	0.2	0.8	0.2		0.12
Action:	type A					
BA	A	0.4	0.2	0.1	0.95	0.17
	BAA	0.3	0.8	0.7	0.03	0.11
	BAB	0.1	0.8	0.7	0.02	0.03
	BB	0.2	0.8	0.2	0.95	0.69
Action:	delete A					

Table 4.1: Probabilities for worked example.

the language model for priors across potential next letters. The first two rows of Table 4.1 show example LM probabilities for the two symbols in the alphabet. Since no letter has a posterior probability above the threshold value, we must now make an observation. Note that when no letters have been typed, the probability of backspace must be zero.

$$\text{ERP observation } 1 = [0.2, 0.8, 0.0]$$

We add this to our table, calculate the product across each row, then normalize to get an updated posterior probability. The probability of the string B now exceeds threshold, so we type it. We can also now calculate the posterior probability of backspace, which is the sum of all posterior probabilities that do not have our currently typed string (B) as a prefix. In this case, it is equal to 0.14.

This leads to rows 3-5 of Table 4.1. Since B has been typed, we expand the context strings starting with B and re-query the language model. We need not expand the A row, but we retain it for our calculations – it now falls in the 'inconsistent' class of context strings referred to in the CopyText algorithm in Figure 4.1. Note that score of B from ERP observation 1 is shared by both new expanded context strings BA and BB.

Since no string probability exceeds threshold based just on the LM probability and ERP observation 1, we make another ERP observation:

$$\text{ERP observation } 2 = [0.7, 0.2, 0.1]$$

When adding this observation to our table, we apply $\mathcal{L}(\texttt{backspace}) = 0.1$ to A, since it falls in the set of strings that are inconsistent with the currently typed string.

Based on posteriors calculated after ERP observation 2, `A` is the symbol above threshold that is typed, making the currently typed string `BA`. Then we expand again. Note that the probability of `backspace` – calculated by summing the posteriors of all context strings that are inconsistent with the typed sting (lines 7-8 in the CopyText algorithm in Figure 4.1) – is now $0.03 + 0.12 = 0.15$.

Suppose the next observation strongly favors typing backspace:

$$\text{ERP observation } 3 = [0.03, 0.02, 0.95]$$

The newly expanded table with ERP observation 3 can be seen in rows 6-9 of Table 4.1. The probability of `backspace` is now $0.17 + 0.69 = 0.86$, so we type it, taking our typed string back to `B`. For the next step of the algorithm after deleting the symbol, the prior probabilities for `A` and `B` would now be calculated by summing over already expanded context strings (lines 14-15 in the algorithm).

```
UpdSymbolProbs($V, \mathcal{P}, i, \theta$, min_iter, max_iter)
$j \leftarrow 0$
for $k = 0$ to $|V|$ do                          ▷ Let $v_0$ = backspace
  $\mathcal{L}[k] \leftarrow 1$                          ▷ Initialize likelihoods to 1
while $j <$ max_iter and ($\max(\mathcal{P}) < \theta$ or $j <$ min_iter) do
    $j \leftarrow j + 1$                        ▷ Present another sequence to user
    for $k = 0$ to $|V|$ do                       ▷ Let $v_0$ = backspace
      $\mathcal{L}[k] \leftarrow \mathcal{L}[k] \cdot \mathrm{ERP}_{ij}(v_k)$        ▷ Update likelihoods to return
      $\mathcal{P}[k] \leftarrow \mathcal{P}[k] \cdot \mathrm{ERP}_{ij}(v_k)$          ▷ See Equation 3.13
    $\mathcal{P} \leftarrow$ NORMALIZE($\mathcal{P}$)
return ($\mathcal{P}, \mathcal{L}$)

CopyText($T, V, \theta$, min_iter, max_iter)     ▷ Given a text $T$ to copy
$t \leftarrow \epsilon$                      ▷ Initialize copied text to empty string
$S \leftarrow \{(t, 1.0)\}$        ▷ Initialize set of context strings (string, prob)
while $T \neq t$ do
    for $k = 0$ to $|V|$ do                       ▷ Let $v_0$ = backspace
      $\mathcal{P}[k] \leftarrow 0$                   ▷ Initialize probabilities to zero
    for each $(s, p) \in S$ do                ▷ for all existing contexts
      if $s[1, |t|] \neq t$ then          ▷ $t$ is not a prefix of $s$ (inconsistent)
          $\mathcal{P}[0] \leftarrow \mathcal{P}[0] + p$     ▷ contributes to backspace probability
      else if $s = t$ then               ▷ $t$ has not been extended before
            REMOVE $(s, p)$ FROM $S$
            for each $v_k \in V$   $(1 \leq k \leq |V|)$ do
              $\mathcal{P}[k] \leftarrow \mathrm{LM}_{|t|+1}(v_k) \cdot p$     ▷ Initialize with LM prior
              $S \leftarrow S \cup \{(s \cdot v_k, \mathcal{P}[k])\}$   ▷ extend $s$ by each symbol
      else for $v_k = s[|t| + 1, |t| + 1]$        ▷ $s[0, |t| + 1] = t \cdot v_k$
            $\mathcal{P}[k] \leftarrow \mathcal{P}[k] + p$  ▷ Add probability (already extended)
    $\mathcal{P} \leftarrow$ NORMALIZE($\mathcal{P}$)
    $(\mathcal{P}, \mathcal{L}) \leftarrow$ UpdSymbolProbs($V, \mathcal{P}, |t| + 1, \theta$, min_iter, max_iter)
    for each $(s, p) \in S$ do                ▷ Update all context probs
      if $s[1, |t|] = t$ then                   ▷ $s$ is consistent with $t$
          for $v_k = s[|t| + 1, |t| + 1]$       ▷ $v_k$ is symbol following $t$
            $(s, p) \leftarrow (s, p \cdot \mathcal{L}[k])$
      else $(s, p) \leftarrow (s, p \cdot \mathcal{L}[0])$           ▷ $s$ is inconsistent with $t$
    NORMALIZE($S$)
    $\hat{v} \leftarrow \arg\max_{v_k \in V}(\mathcal{P})$         ▷ System detected target symbol
    if $\hat{v} = v_0$ then                    ▷ Backspace was selected
        $t = t[1, |t| - 1]$                ▷ Remove final symbol from $t$
    else $t = t \cdot \hat{v}$                     ▷ Type symbol $\hat{v}$ in copied text $t$
```

Figure 4.1: Improved inference algorithm for copying text. For strings s, t, $s \cdot t$ denotes string concatenation. For string $s = s_1 \ldots s_n$ let $s[i, j]$ be the substring $s_i \ldots s_j$. For simplicity, if $j > i$, then $s[i, j] = \epsilon$; and if $j > |s|$, then $s[i, j] = s[i, |s|]$.

Chapter 5

Applied Full History Fusion: BCI Simulation

In this chapter and the next I will describe the results of two experiments undertaken to test the effectiveness of the Full History Fusion method compared to the more heuristic standard approach used in RSVP Keyboard. Both experiments examine typing performance in a *copy task* environment, where the user (or simulated user) is required to exactly copy a certain amount of text using RSVP as the input modality.

Both of these experiments demonstrated a measurable improvement in sequences per character for the Full History Fusion method over default RSVP Keyboard settings. Further, these improvements persisted even when the default settings were optimized beyond their current values. Since sequences per character is expected to correlate monotonically with typing input speed, these results show that FHF is effective at improving typing performance under the RSVP Keyboard paradigm.

The first of these experiments was a large-scale analysis of RSVP Keyboard as it might be used by actual people, but the user inputs (EEG brain signals) were *simulated* rather than being collected from humans. This allowed for an experiment to be performed with orders of magnitude more typed text than could ever be practically achieved with human subjects, while still demonstrating the relative performance of FHF and the standard method.

5.1 Materials and Methods

The basic structure of this experiment is a copy task, in which a user must “type” a series of letters that perfectly match a series of target sentences. In this case, the “user” is not an actual person at all, but a simulated agent, whose brain wave outputs are generated using a model that can create artificial EEG score vectors from prerecorded EEG data. Notably, the EEG signals themselves are not directly simulated; instead, the samples are generated from the classifier outputs from EEG decoder models trained on real people. This makes them easy to use as input in the RSVP Keyboard paradigm. Specifically, these samples are made using a kernel density estimation of the target/non-target classes generated by the RDA classifier on data from real users of the system.

Multiple models are created, each corresponding to an actual RSVP Keyboard user who has previously used the system in a *non-simulated* setting, i.e. with real EEG brain signals. Data from these real sessions are used to create a separate model specifically for each individual user. During a calibration phase, these models are divided based on the area under the receiver operating characteristic curve (AUC), which is calculated using cross-validation. This value generally corresponds to the ease with which the ERP classification system can classify the EEG signals from that particular user. We therefore use the AUC score to characterize the user model in terms of the relative difficulty within the RSVP Keyboard protocol of typing for that user. In general, user data that is easier to classify results in a model with higher AUC, which subsequently results in fewer sequence presentations during RSVP, which finally results in a faster overall typing speed in simulation.

We used five different user sessions as seeds for simulation data, each with a different AUC value, ranging from quite high (0.90) to relatively low (0.71), thus simulating several diverse use scenarios.

The independent variables were the user model selected, the model paradigm (typically standard RSVP Keyboard or FHF), the test corpus, and various model meta-parameters.

5.2 Data and Evaluation

Three test sets were used. The main test set was a set of personal emails set aside for research use from a person with LIS, whom we will refer to as GB. This set contained 1,128 lines of text, about 10,000 words. We also tested two other sets, one with around 35,000 lines of New York Times newswire text, and 10,000 words from a "simulated" AAC-like corpus created through crowdsourcing [Vertanen and Kristensson, 2011b].

For each experiment performed, the simulated user carried out the task of copying the test set in question using the RSVP paradigm. The user was assumed to be perfect in intent, i.e. they always attempted to select the correct symbol (including backspace) at each decision point, even noise inherent in the model itself caused errors to occur.

The dependent variables in this experiment were computed via two evaluation metrics. The first is sequences presented per letter. This records the number of observations (sequences shown) divided by the total number of characters in the test set. It is important to note that the denominator in this formula only counts the number of symbols in the eventual correct typing action. Any incorrect letters or backspaces typed are not considered (though they are reflected in the numerator). This allows the metric to better correlate to typing speed, as the end result of typing is the true measure of how much has been produced in a typing session. A typing session where a word is typed correctly, but many errors have to be deleted, should result in a higher sequences per letter score than one in which no errors occur. The second metric is letters per minute, which is an estimated typing rate for the test set, based on an RSVP paradigm of 200 milliseconds per symbol, 28 symbols, and five seconds between sequences.

5.3 Parameter Optimization

Before examining the impact of the FHF algorithm on the efficiency of typing with the RSVP Keyboard, I first examined the impact of optimizing metaparameters of the *standard* system, along with some other adjustments suggested by my algorithm. Optimizing the baseline in this way is important because it had never actually been done before; it was strongly suspected by the developers of RSVP Keyboard that its default settings might not be optimal, but until the simulation framework was developed it was extremely difficult to attempt to find optimal settings in a systematic way. Further, this optimization allows for comparison against a number of baseline system configurations, to better judge the ultimate utility of FHF in the space of possible configurations.

For this optimization, and the optimization of RSVP Keyboard with FHF, grid search over the parameter set was used. implemented for all of the system meta-parameters (see Section 3.2.4) other than minimum sequence presentations. We tried 1960 combinations of parameters, each varying by regular intervals within their natural range, over five simulations apiece on a held-out development set, which consisted of 50 reserved lines from the GB email corpus.

5.4 Model Variants

Several RSVP Keyboard variants were simulated. The first, and the most basic baseline, is the current RSVP Keyboard system, using metaparameters as described in Subsection 3.2.4. Here, a fixed backspace value is used, along with all of the other metaparameter defaults.

The next variants involved improving this baseline. I performed a grid search for optimized metaparameter settings for each AUC level on a held-out data set, as described in the previous section. The main two variants of this model depended upon whether autotyping was allowed, e.g.

System Configuration	AUC 1.00	AUC 0.90	AUC 0.83	AUC 0.80	AUC 0.75	AUC 0.71
Current	1.00	2.34	2.87	7.77	FAILURE	FAILURE
Current Opt.	1.00	2.05	2.39	3.74	5.22	6.18
Current Opt+Auto	FAILURE	2.09	2.46	3.83	5.27	6.29
Current Opt+Back	1.00	1.98	2.32	3.64	5.10	6.14
Current Opt+Auto+Back	FAILURE	1.84	2.17	3.51	4.89	6.03
Best Configuration Pct. Improvement over Current	0.0%	21%	24%	54%	N/A	N/A

Table 5.1: Simulated Sequences per Letter on the GB email test set for various AUC values and RSVP Keyboard parameterizations, using current system inference. 100 Monte Carlo simulations in every condition.

whether the minimum sequences parameter is set to 1 or 0.

Two further variants used the current RSVP Keyboard system but with a modified *pseudo-dynamic backspace* configuration where the probability of backspace is not fixed, but rather is related to the posterior probability when the previous symbol was typed. If p was the posterior probability of that last-typed symbol (either above threshold or highest probability after the maximum number of sequences have been presented), then the probability of backspace at the next position is set to $1-p$. This intuition, that backspace can be dynamic and should be roughly inversely proportional to the confidence of the last letter typed, is related to FHF but accomplished in a heuristic way rather than by collecting a full history of observations. This variant was used with and without autotyping.

Finally, the FHF algorithm was used, both with and without autotyping allowed.

5.5 Results

In this section, evaluation results for the various baseline models as well as the FHF models are given. Significant performance improvements were observed in both the baseline variants and the FHF model.

5.5.1 Baseline Results

Table 5.1 presents the mean number of sequence presentations required per letter in the GB email testset under five configurations of the baseline system, for varying AUC values. We include AUC=1.0 (a perfect classifier, always giving all of the probability mass to the correct symbol) to illustrate behavior of the system under ideal conditions.

The top row represents the unmodified RSVP Keyboard settings. Notably, in this scenario, the simulations for users with AUC less than 0.8 *entirely fail* to type the test corpus, due to the inability to recover from errors. This is a very concrete example of the runaway typing problem described in Section 4.1. Essentially, the fixed backspace probability under default settings is likely too low to adequately recover from errors or groups of errors. This result was a powerful validation of what had been observed in practice: Sometimes the RSVP Keyboard simply did not work for certain users.

The next two rows represent versions of RSVP Keyboard with optimized parameters. The Current Opt. row does not allow autotyping, and the Current Opt+Auto row allows it. Optimizing these meta-parameters yields large improvements, particularly for the lower AUC scenarios – halving the number of sequence presentations for AUC=0.8 and providing a system configuration for the lower AUC values that permit task completion. Interestingly, autocompletion yields no gains in any condition for standard RSVP keyboard, and in fact causes failure in the perfect classifier condition (AUC=1.0). This is due to endless loops that occur when an incorrect symbol is autotyped, followed by a (correct) backspace – which again yields a state that will autotype the same incorrect symbol. This is exactly the repeated error problem described in Section 4.1, further underscoring the importance of keeping track of prior events in assigning probabilities to subsequent events (as is done in Full History Fusion).

The next two rows feature the pseudo-dynamic backspace method, with and without autotyping allowed, labeled Current Opt+Back and Current Opt+Auto+Back. Other than for the hypothetical perfect classifier, this dynamic backspace probability yields modest improvements over the standard optimized configuration; and with the dynamic backspace probability, autocompletion now yields some additional improvements.

The final row shows how much the baseline model can be improved using parameter optimization and simple heuristics, presented as a percentage improvement of the best baseline variant configuration over the current baseline system configuration. We find robust improvements for all subject-derived AUC levels, with dramatic improvements for the lower AUC levels. These optimized baselines can now be used as a stronger comparison point for the performance of models using Full History Fusion.

5.5.2 Full History Fusion Results

Table 5.2 shows the results achieved with Full History Fusion versus both the Current Opt. configuration and the best performing baseline configuration from Table 5.1 of the current RSVP Keyboard system. For all conditions other than the perfect classifier (AUC=1.0), the FHF algorithm with no autotype (minimum number of presented sequences set to 1) achieves modest improvements over the current configurations, with more gains at the lower levels of AUC. When

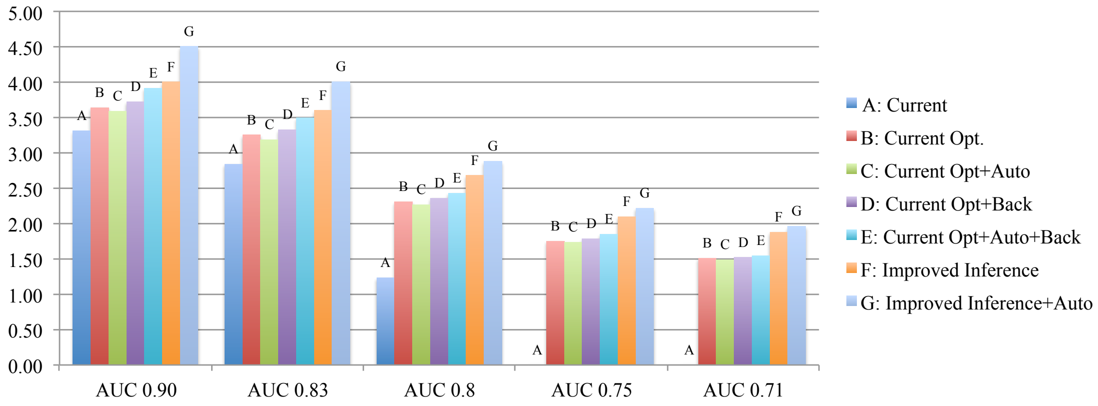


Figure 5.1: Letters typed per minute for various simulated AUC values.

we add autotyping to FHF by reducing the minimum number of presented sequences to 0, we get additional gains across the board, even for the AUC=1.0 condition. In this case, gains due to autotyping are greater for higher AUC conditions than they are for lower AUC conditions.

Figure 5.1 presents all conditions reported in Tables 5.1 and 5.2, with the sequences per letter converted to letters per minute, following the conversion as presented in Section 5.2. In this graph, it is easy to visualize the relatively higher impact of autotyping with improved inference at the higher AUC levels versus the lower levels; and the improvements due to FHF alone at those lower levels.

These improvements were observed for other test corpora, as well. Table 5.5.2 shows the performance of the best current RSVP Keyboard configuration compared to Full History Fusion with autotype for AUC=0.90, when evaluated on the different test sets mentioned in Section 5.2. Comparable results (indicating generalization across different text genre) were found for other AUC levels as well. Due to the number of simulations and the size of the test sets, all reported differences are statistically significant at $p < 0.001$.

System Configuration	AUC 1.00	AUC 0.90	AUC 0.83	AUC 0.80	AUC 0.75	AUC 0.71
Current Opt.	1.00	2.05	2.39	3.74	5.22	6.18
Current Best Configuration	1.00	1.84	2.17	3.51	4.89	6.03
Improved Inference	1.00	1.78	2.08	3.10	4.21	4.80
Improved Inference+Auto	0.67	1.48	1.78	2.82	3.93	4.56
Improvement Pct. over						
Current Opt.:	33%	28%	26%	25%	25%	26%
Current Best Configuration:	33%	20%	18%	20%	20%	24%

Table 5.2: Sequences per Letter for various AUC values and inference algorithms. Values average 100 Monte Carlo simulations.

Test Corpus	Current Opt+Auto+Back	Improved Inference+Auto	Pct. Improvement
GB	1.84	1.48	20%
AAC	2.01	1.69	16%
NYT	1.85	1.50	19%

Table 5.3: Comparison of sequences per letter across different test corpora (AUC = 0.90).

AUC	Current Opt+Auto+Back	Improved Inference	Improved Inference+Auto
0.71	18.5	31.6	37.2
0.75	14.1	28.0	34.3
0.80	13.1	24.6	32.7
0.83	10.0	18.3	28.9
0.90	9.7	16.4	29.2

Table 5.4: Percentage of symbols typed that are backspace across varying AUC values.

5.6 Analysis

In this section I perform a deeper analysis of these results, and discuss other interesting behaviors and trade-offs.

5.6.1 The Backspace Trade-off

Monte Carlo simulations on development data (and subsequent test results) show that the optimal value for the decision threshold with the improved algorithm is at 0.5. Recall that this is the probability value at which the system will type the most probable letter. Thus, if the probability is 0.5 or higher, then it is more efficient to go ahead and type the symbol than to spend the time to achieve higher certainty. One potential problem with setting the decision threshold this low is that *a higher proportion of typing actions is a backspace* than when the threshold is set at, say, 0.9, as in the current default RSVP Keyboard system settings. This increased use of backspace is seen in the data. Table 5.4 shows the ratio of backspaces to all typed symbols in different simulations. The effect is even more striking in FHF, which requires backspace more than twice as often as the fastest method under the current system, and over three times as often for the highest AUC level.

Because frequent backspaces may be confusing and frustrating, users might choose to sacrifice some performance by reducing the frequency of backspace. This trade-off can be managed by varying the decision threshold parameter; a higher threshold will be suboptimal from a typing speed standpoint, but will result in fewer backspaces. The dynamics of this trade-off are illustrated in Figure 5.2. The decision threshold values used for this figure were 0.5, 0.6, 0.7, 0.8, 0.9, 0.95,

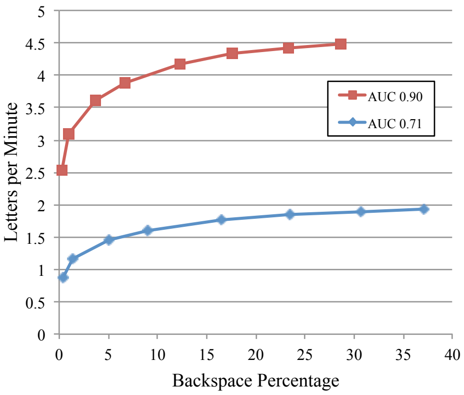


Figure 5.2: Backspace percentage versus typing speed for the improved inference with autotype at two simulated AUC values.

and 0.99. By choosing a particular operating point, it is possible to cut the backspace percentage in half and only slightly decrease typing performance. As a result, it is possible to allow for user control of backspace performance based on personal preference, while still maintaining relative gains in performance.

5.6.2 Autotyping and Autodelete

One aspect of autotyping that I have not discussed much is that the backspace symbol can be autotyped just like other symbols, which results in what I call autodeletion. In fact, complex combinations of automatic typing and deleting can lead to what I have termed "autorevision." For example, Figure 5.3 presents a sequence of autotype and autodelete events that were actually observed in the simulated trials. Once the string "my care" was typed, autotyping produced the rest of the word "carefully" with following whitespace, before reaching a state where EEG observations were required. The result of ERP detection was a backspace symbol, leading to deletion of the whitespace. Once this deletion occurred, autotyping of the backspace symbol (autodeleting) deleted the suffix of "carefully" back to "care", at which point autotyping produced a new full word candidate "caregivers." This entire sequence of actions required just a single round of ERP detection.

For AUC=0.90, we find that 41% of symbols are autotyped, and that around 18% of autotyping events complete a word. For lower AUC values, those percentages are lower, 31% autotyped and 13% of autotyping events complete a word. Table 5.5 presents statistics on the frequency of the various kinds of autotyping events for different AUC levels with the improved inference algorithm. We see that forward autotyping is the most common, from 70-80% of the autotyping events, versus

step	typed	via action
1	`my_caref`	`autotype`
2	`my_carefu`	│
3	`my_careful`	│
4	`my_carefull`	│
5	`my_carefully`	│
6	`my_carefully_`	↓
7		`observe EEG`
8	`my_carefully`	`delete`
9	`my_carefull`	`autodelete`
10	`my_careful`	│
11	`my_carefu`	│
12	`my_caref`	│
13	`my_care`	↓
14	`my_careg`	`autotype`
15	`my_caregi`	│
16	`my_caregiv`	│
17	`my_caregive`	│
18	`my_caregiver`	│
19	`my_caregivers`	↓

Figure 5.3: Example of autorevision

20-30% being autodeletions. More than half of autotyping is a single symbol.

5.7 Discussion

This experiment shows that the Full History Fusion algorithm results in significant improvements in simulated typing speed. I have shown that this gain is consistent across different text corpora, and that autotyping is both fast and well-suited to this method. I have introduced the possibility of autodeletion and autorevision, interesting varieties of autotyping that we believe to be novel to this research. I have examined the trade-offs involved with these higher typing speeds, specifically those involved with the frequency of typing backspace versus typing speed.

The example in Figure 5.3 is a particularly effective illustration of the power of Full History Fusion. Because FHF keeps track of an entire history of EEG observations, it is able to make long-range determinations about the relative probabilities of different character sequences. In this

Type of autotyping event	AUC 0.90	AUC 0.80	AUC 0.71
Forward only	77.3	73.0	68.0
Forward only (1 symbol)	40.7	39.1	39.0
Forward only (2+ symbols)	36.6	33.9	29.1
Backward only	19.6	25.2	30.7
Backward only (1 symbol)	10.7	14.2	18.1
Backward only (2+ symbols)	8.9	11.0	12.7
Forward and backward (autorevision)	3.2	1.8	1.3

Table 5.5: Frequency of different types of autotyping: forward only, backward only, and autorevision, under various AUC values. Algorithm is Improved Inference+Auto.

case, a single observation of a backspace signal is enough to effectively "wither the branch" in the character tree corresponding to the current word, and *simultaneously* boost the probability of another branch corresponding to another word. This is a powerful example of how FHF shifts the concept of textual revision into an *incremental* setting. Whereas non-incremental algorithms might simply change the current best hypothesis as new evidence is collected, this is not easily done in the incremental environment of text entry, particularly the relatively low-speed text entry that often occurs in AAC. With FHF, individual backspace observations, in combination with autocompletion and autodeletion, can accomplish a very similar revision action without violating the incrementality at the core of these modalities.

Chapter 6

Applied Full History Fusion: Button Press Experiment

The second major FHF experiment I performed sought to determine whether the results of the simulation experiment in Chapter 5 can be observed in RSVP Keyboard typing done by human subjects in the real world. In essence, the question is whether the expected improved performance of Full History Fusion can be observed empirically in actual users.

This experiment was a smaller-scale typing study with human subjects. For ease of data collection, a *button press* was used as a proxy for an EEG signal. This alternate input was made possible by the relative flexibility of the RSVP Keyboard paradigm; any measurement that can be interpreted as a probability distribution over the symbols in the RSVP sequence can be used as input. In this case, the exact button press timings were converted into a probability distribution using a specialized decoder, built in combination with established models of human reaction time.

6.1 Materials and Methods

There are three major differences between this experiment and the simulation experiment presented in Chapter 5. First, there is no simulation involved in the experiment[1], only actual human subjects. This means that the sample sizes are necessarily much lower, particularly with regard to the amount of text involved in the copy task. Second, EEG is not used for the user typing intent signal. This was deemed impractical given the number of subjects needed and the length of time required to type the copy text. Instead, the signal from a single button is used to express user intent. Users are instructed to simply press the space bar on a laptop when their target letter appears in the RSVP sequence stream. This button press is then converted into a probability distribution by a

[1] This is not entirely true, as simulation was used to optimize certain parameters before beginning the experiment proper (see Section 6.1.1).

Difficulty Level Group	Context	Target Word
1	I_DO_	NOT
	WHAT_DO_	YOU
2	HEALTH_	CARE
	SUPER_	BOWL
3	MAGIC_	WAND
	BEFORE_THE_	RUSH
4	THE_COMPOSER_	SITS
	SKYLARKS_	BOOING

Table 6.1: Copy task words for button press experiment, with context.

button press decoder model. Finally, the use of human subjects allowed for the collection of other evaluation metrics, in particular a questionnaire asking the participants to judge their subjective physical and mental state at various points during the completion of the typing task.

6.1.1 Task Data Setup

Like the simulated EEG experiment, the button-press experiment was a copy task, and was structured as a direct comparison between standard RSVP Keyboard and RSVP Keyboard with Full History Fusion.

The copy task consisted of four groups of two words. Each word group was typed twice by each user, once using standard RSVP Keyboard and once using FHF. This meant that each user typed a total of sixteen words. The words were structured in two ways. First, the word groups themselves corresponded to varying *difficulty levels*, ranging from easy to difficult. This difficulty was determined by querying the language model at each character position in a given word, and determining how well the model predicts the next character in each context. Higher difficulty words contain letters poorly predicted by the language model, and easier words have letters assigned very high prior probability mass by the LM. The exact words themselves were taken from the word groups I developed for Oken et al. [2014], in which BCI users were asked to complete a series of *mastery tasks*, copy tasks of increasing difficulty based on the LM predictive power of the constituent words. The words are shown in Table 6.1. The algorithm chosen and and the difficulty levels were the primary independent variables in the experiment.

The second way the words were structured was in their ordering. I used a Latin square design to ensure balance and to help mitigate any learning effects or other biases related to the order in which the words were presented to the subject. An equal number of subjects saw each difficulty level group at each of 4 possible positions in their ordering. The Latin square is shown in Table 6.2. Each cell in the table is a word group. Each row in the table consists of eight word groups typed by

F1	S1	F2	S2	F4	S4	F3	S3
F2	S2	F3	S3	F1	S1	F4	S4
F3	S3	F4	S4	F2	S2	F1	S1
F4	S4	F1	S1	F3	S3	F2	S2
S1	F1	S2	F2	S4	F4	S3	F3
S2	F2	S3	F3	S1	F1	S4	F4
S3	F3	S4	F4	S2	F2	S1	F1
S4	F4	S1	F1	S3	F3	S2	F2

Table 6.2: Latin square used for word ordering.

each subject. The numbers represent difficulty levels. S represents standard RSVP Keyboard, and F represents Full History Fusion. Note that half of all subjects typed each word group with FHF first, and half typed them with the standard method first. Further note that the difficulty levels were kept together so that the subjects could more easily remember their subjective experience when answering the questionnaire, e.g. F2 and S2 are always temporally adjacent for each subject.

6.1.2 Subjects and Protocol

Subjects were recruited using convenience sampling, based on the criteria that they be age 18 or over, literate in English, able to see words on a screen, and able to push a button. A power analysis of the simulation experiment in Chapter 5 showed that, assuming relative comparability of the experiments, 18 subjects were needed in order to significantly demonstrate the same FHF improvements seen in that experiment. Ultimately 24 total subjects were recruited, in order to complete three cycles through the Latin square. It is important to note that, because of certain IRB limitations, no personally identifiable information was collected about the subjects. While this protected their privacy, it prevented any evaluation or analysis that depends upon these values, e.g. results by age or gender.

Each subject was given instructions on how the RSVP Keyboard paradigm works. They were then asked to complete an initial calibration session, in which single target letters are provided to the subject directly before each RSVP sequence. The purpose of this calibration session was to estimate a set of parameters associated with that subject's button press reaction time model (See Section 6.2). This is comparable to the calibration mode used for BCI/EEG users of RSVP Keyboard, in which user-specific parameters are estimated. Eighty characters were used for calibration, which took 10-15 minutes. Subjects were then given a practice word to type, in order to gain familiarity with the full-word copy text interface. After typing this word, and being specifically instructed on the importance of the backspace symbol, subjects began the experiment proper.

The main experiment consisted of a copy task, in which the subject was asked to use the space bar in the RSVP modality to type a set of target words. Along with each target word, an ”already typed” section was provided, giving the subject context (which was also given to the language model) and placing them in the middle of an imagined typing session rather than at its beginning. A bar chart of current symbol probabilities was provided on the left side of the screen so that subjects could see how the algorithm was proceeding (if desired). Subjects were told that this was an exact copy task, and that they would not be able to proceed if any errors persisted in their typed string.

After each word group (consisting of two target words), subjects were asked to give subjective scores on how they did on that group only, based on the NASA Task Load Index (TLX) questionnaire developed for measuring human perception of workload. This subjective dependent variable was not present in Chapter 5. I modified the questionnaire slightly, changing the Likert number range and switching the polarity of one of the questions. Subjects used the number keys on the keyboard to input their scores. Each subject completed the full questionnaire eight times. The questions were as follows:

1. How mentally demanding was the task? (1 = very low, 7 = very high)

2. How physically demanding was the task? (1 = very low, 7 = very high)

3. How hurried or rushed was the pace of the task? (1 = very low, 7 = very high)

4. How successful were you in accomplishing what you were asked to do? (1 = failure, 7 = perfect)

5. How hard did you have to work (mentally and physically) to accomplish your level of performance? (1 = very low, 7 = very high)

6. How insecure, discouraged, irritated, stressed, and annoyed were you? (1 = very low, 7 = very high)

After every letter typed during the experiment, in fact after every RSVP sequence, subjects were given a ”press any key to proceed” message. This made the experiment partially asynchronous, which allowed subjects to rest or take breaks at any time. The total experiment duration ranged from 35 minutes to 60 minutes, including verbal instructions. In addition, all timing data of button presses and sequence presentations, as well as the contents of the sequences and the various metaparameters, were logged during the experiment for future analysis.

6.2 Modeling Button Presses

In order to conduct a button press experiment within the RSVP Keyboard paradigm, it is necessary to convert sequence presentation timing data—and button press timing information—into a posterior distribution over characters in a typing alphabet. For example, suppose the letters `A` through `Z` are included in the RSVP sequence, each letter persisting for 200 milliseconds. Suppose that the user presses the space bar at 1478 milliseconds into the sequence presentation. How can this information be used to construct a set of probabilities? I accomplish this by building a button press model.

6.2.1 Reaction Time Model

At the core of this larger model is a constituent model of human reaction time. Fortunately, human reaction time has been studied extensively [Rohrer and Wixted, 1994, Palmer et al., 2011]. It turns out that reaction time can be modeled using an exponentially modified Gaussian distribution, sometimes called the exGaussian distribution. The exGaussian distribution is the sum of two distributions, one normal and one exponential (in this case, exponentially decaying). Neurologists hypothesize that the normal distribution corresponds to an initial perception of a stimulus, and the exponential distribution corresponds to a longer, cognitive process in which the brain determines whether that stimulus is in fact a target to be reacted to. The exGaussian distribution is defined [Grushka, 1972] as

$$\mathrm{exg}(x;\mu,\sigma,\lambda) = \frac{\lambda}{2}e^{\frac{\lambda}{2}(2\mu+\lambda\sigma^2-2x)}(1-\mathrm{erf}(\frac{\mu+\lambda\sigma^2-x}{\sqrt{2}\sigma})), \tag{6.1}$$

where erf is the error function

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt. \tag{6.2}$$

The parameter μ is the mean of the Gaussian component, σ is the variance of the Gaussian component, and λ is the decay rate of the exponential component. In my model, I used the slightly alternate formulation $\tau = \frac{1}{\lambda}$, where τ is the exponent relaxation time.

The exGaussian is a probability density function of x, where x is the amount of time that has elapsed since the onset of a stimulus. Figure 6.1 shows a typical exGaussian distribution of human reaction time. Note the positive skew, indicative of the exponential component of the distribution.

My reaction time model included two additional parameters, both having to do with human error in the RSVP paradigm. The first is the probability that the subject will notice the target symbol in the RSVP sequence. Due to the relatively high presentation rate typical of this experiment, it is quite common for subjects to not notice the target. I call this the *notice* parameter,

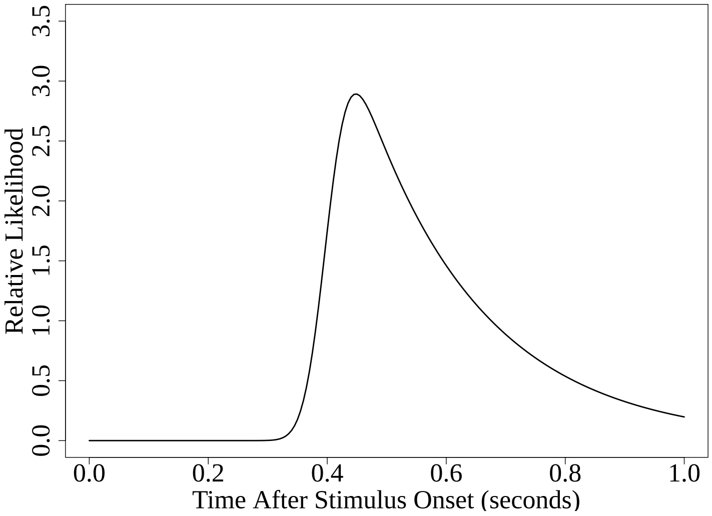


Figure 6.1: A typical exGaussian distribution ($\mu = 0.4$, $\sigma = 0.03$, $\tau = 0.2$)

denoted as p_n. The second is the probability that the subject will accidentally execute a spurious input (in this case a tap), not related to any stimulus. This is also frequently attributed in practice, both in button press and EEG modalities. I call this the *spurious tap* parameter, denoted as p_s.

The input to the button press model has two parts. First is a set R consisting of k presented symbols, each with a corresponding presentation time t_s. Second is the time t_p of any button press that occurred during sequence presentation (or, importantly, if no button press occurred). The purpose of the button press model is to take these inputs and output a probability distribution over every symbol in the typing alphabet (which usually includes backspace). For each presented symbol s, we know its presentation time t_s. If there has been a button press observed, we can compute the elapsed time between the targe symbol onset and the button press and use this as an input to the exGaussian probability density function.

6.2.2 Parameter Estimation

There are five parameters in the button press model: μ, σ, τ, p_n and p_s. In order to customize the button press experiment for each subject, we estimate these parameters for each person individually. This is accomplished using the calibration session, where the subject is asked to respond to 80 single-letter typing stimuli. In order to get better estimates of p_n and p_s, we deliberately exclude the target symbol in 40% of all presented sequences.

The parameters p_n and p_s are estimated first. The *noise tap probability* parameter p_s is estimated using the following equation, looking at sequences in calibration where the target character was not actually present:

$$p_s = \frac{\#\text{ of taps observed with no target present} + 1}{\#\text{ of sequences with no target present} + 2} \tag{6.3}$$

We use Laplace smoothing to improve the estimate. This was useful because some users are relatively good at avoiding noise taps, even avoiding them altogether during calibration. In such a situation, Laplace smoothing prevents p_s from being set to zero.

The *notice probability* parameter p_n is estimated similarly, looking instead at sequences where the target *was* present:

$$p_n = \frac{\#\text{ of taps observed with target present} + 1}{\#\text{ of sequences with target present} + 2} - p_s \tag{6.4}$$

We similarly use Laplace smoothing here, but also discount the probability using p_s. This is because we are making an independence assumption between p_n and p_s, which implies that noise taps will occur with equal probability irrespective of whether the target character is present in that sequence. Therefore we must assume that some of the taps observed in sequences with targets are noise, and discount the estimate of p_n accordingly.

Let $\mathbb{B}$ be the set of 80 observations made during the calibration session. Each element of $\mathbb{B}$ is a single observation consisting of three pieces of information: First, it records the presence/absence of the target symbol in that particular sequence. Second, the timing of any user tap that occurred is known. Third, the timing and identity of each symbol presented in that sequence is recorded.

In order to compute a Maximum Likelihood estimation of the system parameters, we need a formula for the likelihood of this set of observations. We first compute the likelihood of each observation individually. For each observation b in $\mathbb{B}$ there are four possibilities, depending on whether the target and a tap are present or absent in that observation. Given a target symbol c, we define the likelihood $L(b, c)$ as

$$L(b,c) = \begin{cases} p_s & \text{if target is absent and tap is present} \\ 1 - p_s & \text{if target is absent and tap is absent} \\ \frac{p_s}{d} + p_n \operatorname{exg}(x_c; \mu, \sigma, \lambda) & \text{if target is present and tap is present} \\ (1 - p_s)(1 - p_n) & \text{if target is present and tap is absent} \end{cases} \tag{6.5}$$

where exg() is the exGaussian function, d is the duration of a single sequence presentation, and

$$x_c = t_{tap} - t_c \tag{6.6}$$

is the latency between the display onset time of the target character c and the observed tap. The four parts of this likelihood equation can be explained as follows:

> **Target absent, tap present:** Here we simply use the noise probability p_s.
>
> **Target absent, tap absent:** This occurs when no noise is present, $1 - p_s$.
>
> **Target present, tap present:** This is a combination of the likelihood of noise *and* the target being noticed and reacted to. Note that p_s is scaled by the sequence length d so that, like exg(), it is a true probability density function.
>
> **Target present, tap absent:** This occurs when the user *both* fails to notice the target *and* does not commit a noise tap, so we use the product of their complements.

Now that we can compute the likelihood of each observation given the target symbol associated with that observation, we can compute the total likelihood T of the calibration session as follows (note that during calibration, the target symbol c is always known):

$$T = \sum_{b \in \mathbb{B}} L(b, c) \tag{6.7}$$

We can now compute the remaining button press model parameters (μ, σ, λ) using Maximum Likelihood Estimation (MLE) on the value T. I did this using the Nelder-Mead [Nelder and Mead, 1965] option within the `minimize()` function in SciPy. The use of 80 calibration observations was arrived at using simulations on a development set. Simulations using fewer than 80 calibration points resulted in occasional bad models that did not work for RSVP decoding. I also experimented with using MLE to estimate the parameters p_n and p_s as well (without manually estimating them first) but this proved difficult. None of the optimization algorithms worked well with five total parameters, and the ones that did converge did not allow for constraints on the parameters (for instance, keeping p_n and p_s between 0 and 1).

6.2.3 From Taps to Probabilities

After the MLE step, we now have a customized button press model for each user at the end of the calibration phase of the experiment. After each sequence displayed to the user, we can now use this model to generate a dynamic probability distribution over symbols, for use as input into the RSVP Keyboard algorithm. The difference now is that, during the main typing portion of the experiment, the button press model no longer "knows" the target symbol. We therefore must generate a likelihood value for each symbol in the alphabet, and use these values to compute a posterior probability for each symbol (including backspace). After each sequence presentation and a corresponding observation b, we compute the probability $P(\hat{c}|b)$ of a given symbol $\hat{c}$ as follows:

$$P(\hat{c}|b) = \frac{L(b,\hat{c})}{\sum_{\forall s \in \mathbb{V}} L(b,s)} \tag{6.8}$$

Here, $\mathbb{V}$ is the alphabet of all symbols available to be typed (including backspace) and L is the likelihood function from Equation 6.5.

This equation can produce a wide range of probability distributions depending on when a tap is observed. Figure 6.2 shows how the probability of different symbols as button press timings change. In this example, there is an alphabet of 9 symbols and five of them (corresponding to the bottom five colors in the figure) are displayed at 0.1 second intervals, starting at time zero. Four symbols (corresponding to the top five colors) are not displayed in the RSVP sequence. A vertical slice through this chart at time t gives the relative probability of each symbol being the target symbol provided that a button press was observed at that time. The parameters used for this figure are the same as those of Figure 6.1, with the addition of a p_n setting of 0.6 and a p_s setting of 0.2. There are several things of interest here. First, note that on the left side of the chart, all symbols are equally probable. This is because no human can react this quickly to stimulus. Therefore the model spreads the probability equally over all symbols. Note also that, when a button press is observed in the middle of the sequence, very little probability is allocated to the non-displayed symbols (the four top stripes). Finally, note the dominance of the final displayed letter (aqua stripe) when a tap is observed late in the sequence. This is because, unlike the first four symbols, it does not have any competition from letters appearing after it.

If no tap is observed for a given sequence, the probability distribution over symbols is still computed, but they are constant and do not depend on timing. For example, consider the 9-symbol alphabet and the parameter settings used in Figure 6.2. If no tap is observed, application of Equation 6.8 will give each of the five *displayed* symbols a probability of 0.067 and each of the four *absent* symbols a probability 0.167. This makes intuitive sense; if no tap is observed it should boost the probability of symbols not in the sequence and dampen the probability of displayed symbols.

6.2.4 Input Noise

One reason for the selection of button press in this experiment was that, since the reaction times are probabilistic, they are fundamentally uncertain. I considered this to be a good thing, because it meant that the algorithms would have to use a combination of language model and the button-press evidence in order to execute efficient typing. This is comparable to the EEG/BCI modality,

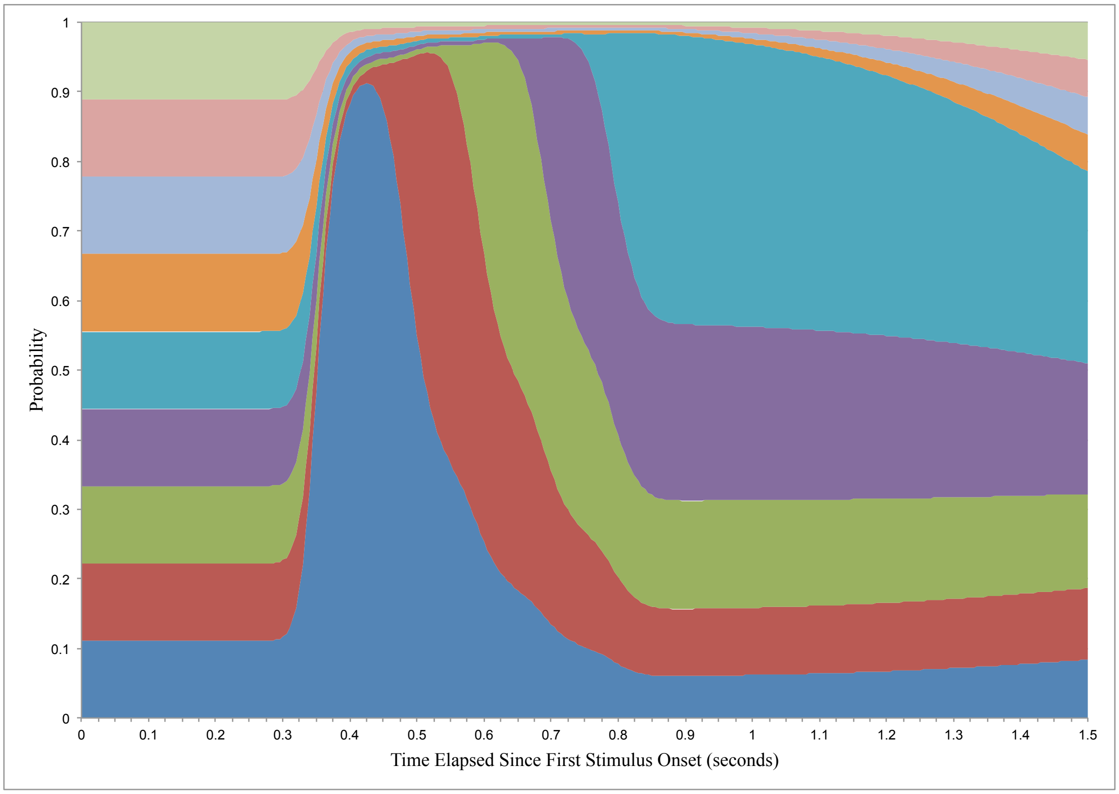


Figure 6.2: A button press probability estimate over time. Each colored stripe represents a symbol.

where scalp-measured brain signals can be very weak and difficult to classify. I hypothesized that, given a fast enough RSVP presentation time, the exGaussian PDF (in conjunction with the notice and noise tap parameters) would result in a good amount of uncertainty in the output of the button press decoder, spreading probability mass over multiple symbols. Interestingly, I found this to be somewhat inadequate. In simulated trials, I found that the backspace symbol was being selected very infrequently, which suggested that the button press model was *too good*, causing very few errors to be typed that would require backspace to repair. Even when I increased the (simulated) stimulus presentation rate, showing each letter for fewer and fewer milliseconds (down to 50), this effect persisted. This was a difficult situation, because the RSVP paradigm begins to break down and become too fast for human perception at around 100-millisecond presentation times.

I solved this problem by introducing noise directly. At each sequence presentation, I calculated the posterior character probabilities as usual. Then, with an error probability p_e, I randomly switched the probability values of the highest-probability character and a random, different, character. This allowed for a more errorful typing session, which is more similar to the BCI scenario and required the subjects to select the backspace key more often, which is central to this work. Through simulations, I found that a good value for p_e is 0.2.

6.3 Results

The results of the button press experiment parallel those of the simulated EEG experiment: FHF is faster than the standard RSVP Keyboard algorithm, both in terms of sequences per character and total typing time. In addition, the questionnaire scores provided by subjects also demonstrate that there is no significant difference in perceived mental or physical effort present for FHF compared to the standard method.

6.3.1 Sequences per Character

Recall from Section 3.1.3 that sequences per character measures how many total RSVP sequences are required, on average, to type each character in the test set. It is a primary dependent variable in this experiment. Figure 6.3 shows the sequences per character results of the button press experiment across all 24 subjects. Under the standard method, subjects required an average of 2.84 sequences to type each character, compared to 2.32 sequences using FHF. Of the 24 subjects, 20 typed the test set with fewer sequences when FHF was used for symbol probability calculation. On average, 0.52 fewer sequences were required, per subject, under FHF. This is an improvement of 18.3%. Using a one-tailed Wilcoxon signed rank test on this difference yields a p value of 0.00248,

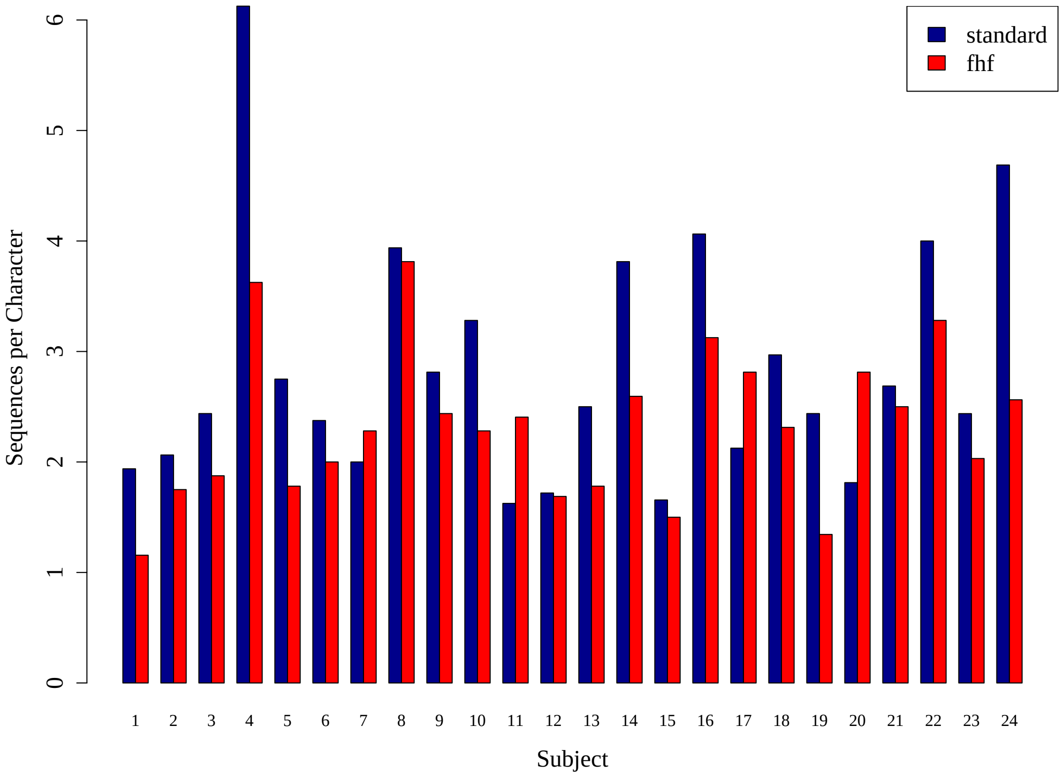


Figure 6.3: Seq/char results for all human test subjects.

a significant result.

Another nice result is that user performance on various mastery levels actually reflects those levels. The hypothesis underlying the mastery levels as developed in Oken et al. [2014] was that words that are "harder" for the LM to predict will be more difficult to type. This can be seen very clearly now empirically, as shown in Table 6.3.

Mastery Level	Standard	FHF
1	1.51	0.56
2	1.97	1.65
3	3.28	2.94
4	3.99	3.43

Table 6.3: Sequences per character, averaged across all users, for various mastery levels.

6.3.2 Typing Time

Overall typing time correlates well with sequences per character, and this is reflected in the results of this experiment. Under the standard method, subjects required an average of 13.33 minutes to type the words in the test set, compared to 11.57 minutes using FHF. Of the 24 subjects, 18 typed the test set faster when FHF was used for symbol probability calculation. On average, 1.56 fewer minutes were required, per subject, under FHF. This is an improvement of 13.2%. Interestingly, this result is slightly less strong than the sequences per character result: Using a one-tailed Wilcoxon signed rank test yields a p value of 0.0174.

It turns out that the reason for this slightly different result is that people spent slightly longer in the pauses *between* sequences when FHF was being used. I measured typing time by combining the total time of all displayed RSVP sequences and the total time spent between sequences. Because the experiment design asked the user after each sequence to press any button to proceed, it was possible for two users to have a different total typing time, even if they observed the same number of sequences. During this gap time, the user sees what letter has been typed (or not typed) by the system and observes any autotypes that have occurred. It is reasonable to assume that FHF, which exhibits a more complex set of behaviors than the standard method, might require a bit more time on the part of the user to ensure they know what is happening. This is observed very clearly in the data. When typing words under the standard method, users spent 6.21 seconds between each sequence, and with FHF they spent 6.75 seconds[2]. This difference of 0.54 seconds is significant under a one-tailed Wilcoxon signed rank test, with a p value of 0.00014. Only 4 of the 20 users spent more "between" time on the standard method than FHF. In short, users spent a little more time thinking under the FHF method, but the savings in terms of total sequences overpowered any risk that this might cause an overall decrease in typing speed.

6.3.3 Subjective Scores

Interestingly, none of the six subjective TLX scores in the user questionnaire showed any significant difference between those reported on the standard algorithm and those reported on FHF. Table 6.4 shows averaged TLX scores for standard vs. FHF, each of which failed to reject the null hypothesis with $p < 0.05$ in a two-tailed Wilcoxon signed rank test. Another interesting effect was that the subjective scores do not seem to correlate with objective metrics of typing speed. For instance, if we take the average of TLX scores across the six questions, we can use this value as a rough proxy

[2] I rounded long between-sequence times down to 20 seconds to remove the effect of users who took breaks during the test, which they were encouraged to do (and some did).

TLX Question	Full Text	Standard	FHF
Q1	How mentally demanding was the task? (1 = very low, 7 = very high)	3.51	3.34
Q2	How physically demanding was the task? (1 = very low, 7 = very high)	1.66	1.74
Q2	How hurried or rushed was the pace of the task? (1 = very low, 7 = very high)	3.46	3.38
Q4	How successful were you in accomplishing what you were asked to do? (1 = failure, 7 = perfect)	5.15	5.23
Q5	How hard did you have to work (mentally and physically) to accomplish your level of performance? (1 = very low, 7 = very high)	3.72	3.47
Q6	How insecure, discouraged, irritated, stressed, and annoyed were you? (1 = very low, 7 = very high)	3.16	2.93

Table 6.4: Average TLX scores across users by question.

for user "demand." This value does not, however, strongly correlate with the observed sequences per character values. For the standard algorithm the correlation coefficient is equal to 0.01; for FHF it is 0.24. In other words, people who were able to type faster did not report any less physical or mental exertion, or vice-versa.

There are various possible explanations for this absence of correlation. The TLX scores might not actually reflect mental and physical demands. Alternatively, those demands might not have any relationship to objective performance measures; there may be other variables driving the subjective experience of exertion. It is also possible that, although word pairs in the same mastery group were typed sequentially to allow subjects to better remember and compare them, the time between groups was just too long. Similarly, the experimental design itself, in which absolute scores but not *relative* scores were collected, might have made it more difficult for the subjects to report comparative differences. Nevertheless, it is probably meaningful that users did not report significant differences in performance between the two algorithms.

6.3.4 Analysis of exGaussian Curves

If we look at the exGaussian curve estimated via MLE for each of the 24 experimental subjects, shown in Figure 6.4, an unusual pattern emerges. Seven of the subjects had a much flatter curve, with a long tail. This appears to be a somewhat discrete phenomenon, as all subjects seem to fall in one of two general curve types with no intermediate forms. Closer inspection reveals that these seven forms all have a large estimated value for τ, the exponential component of the exGaussian curve. These subjects all have a τ value above 0.6, whereas the remaining subjects all have a value

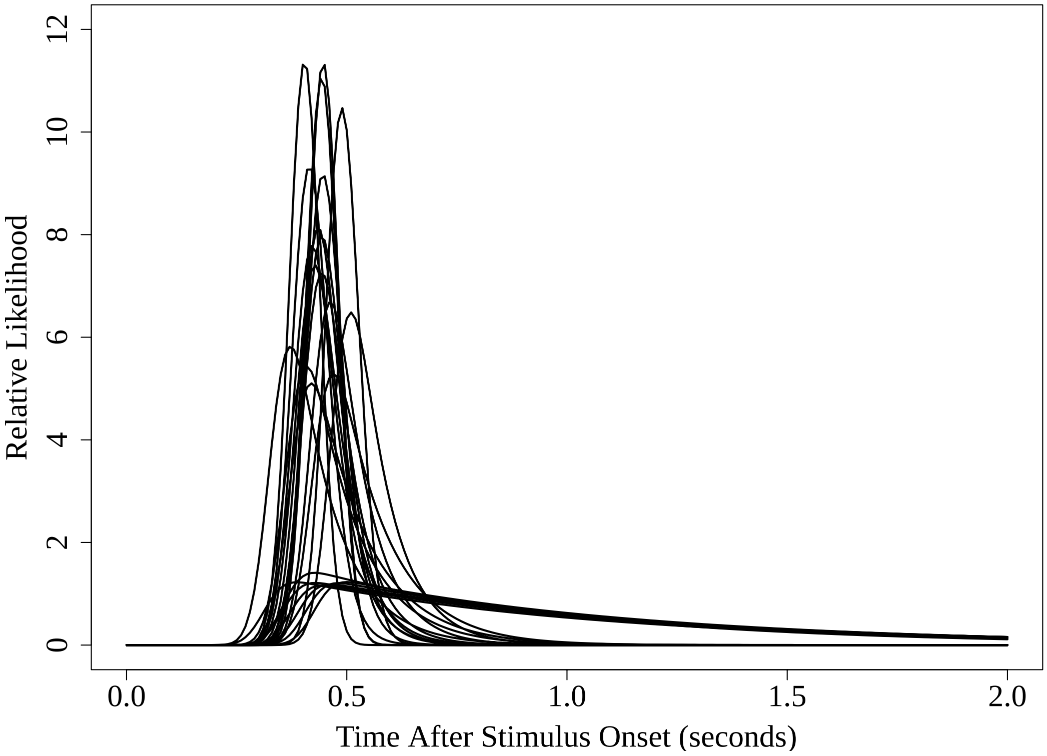


Figure 6.4: The estimated exGaussian curve for all 24 subjects.

below 0.15.

Interestingly, this phenomenon appears to be unrelated to other aspects of that subject's tap profile. For example, the seven users in the long-tail group do not exhibit significantly different values of the noise tap probability or the notice probability. Nor did visual inspection of the individual tap points and latencies reveal any noticeable patterns that would differentiate the two groups. Given the absence of a single causative factor causing this discrepancy, it is a reasonable hypothesis that this is an artefact of the Nelder-Mead optimization algorithm used in the MLE calculation, which may have gotten stuck in a local maximum for those seven subjects.

Most interesting of all is that these seven subjects all show a much more striking relative difference (3.7 times larger) between standard RSVP Keyboard performance and FHF performance. Subjects in the long-tail struggled under the standard RSVP keyboard protocol. This is actually a strong argument *for* the use of FHF, as it suggests that FHF is more robust to defective or misconfigured models, which is certainly not unlikely in this domain.

6.4 Discussion

These results, as in the simulated EEG experiment in Chapter 5, show that Full History Fusion is a superior method to the standard RSVP Keyboard for achieving faster typing speeds. Further, even if the autotyping/autodeleting/autorevision behavior of FHF creates a difficult cognitive overhead for its users (though the subjective scores from the button press experiment appear to suggest that this is not the case), it is possible to tune the algorithm to type fewer backspaces with a relatively small impact on overall typing speed. As such, Full History Fusion is a robust and experimentally validated method for typing within in the RSVP Keyboard text entry paradigm, and an important advancement of the state of the art for incremental typing within this AAC modality.

Chapter 7

Backspace as a Sequence Modeling Problem

One purpose of this thesis is to demonstrate the importance of the backspace key a a powerful tool for maintaining incrementality in a text entry setting. Backspace allows text revision/correction to be cast as a form of *user input*, as opposed to a non-incremental change to the output stream. As described in Chapter 4, one of the benefits of the Full History Fusion algorithm is that it allows for the dynamic calculation of the backspace action within the RSVP Keyboard protocol. User evidence of backspace intent is combined with other evidence in a principled way to compute the probability of backspace at any given position in a typing session. The experiments in Chapters 5 and 6 show that this algorithm is superior to heuristic methods such as a fixed backspace probability and a slightly-more-clever $1 - p$ backspace probability technique. In this chapter I will present an entirely different method of computing the probability of backspace, where backspace is introduced into the training data so that the language model itself can learn it and attempt to predict it.

7.1 Introduction

The backspace key has long been a primary error correction tool for typed text. While its original use was simply to move the typewriter carriage one character to the left[1], an additional error correction functionality has been available since at least 1973[2]. Computer keyboards and word processing software have followed suit, and the backspace[3] key is now ubiquitous on virtual and physical keyboards across languages.

[1] In this chapter I will refer to the *left* as the direction opposite the direction of typing, though this is obviously not true for some languages.

[2] The IBM Correcting Selectric II typewriter allowed the user to press a special key to go back and overtype errors with white ink (or use a special adhesive ribbon to "lift off" the existing letter).

[3] Designated, somewhat confusingly, as `delete` on Apple keyboards.

Error correction is present in many text entry applications. However, these tools typically operate on the word level: A word is completely typed (indicated by a subsequent space or punctuation mark), and based on an underlying error correction or spelling model the word may be automatically transformed into a corrected form or notated as a possible typo. Such systems do not consider backspace as a modality for system-directed error correction. This can pose a problem for *character-based* text entry, as is often used in Augmentative and Alternative Communication (AAC), because such systems are typically low bandwidth. Waiting until the end of a word to execute error correction is often impractical.

In this chapter, I explicitly consider the problem of *predicting* backspace for text entry. I approach this as a sequence modeling task, considering the backspace symbol as just another symbol that can be entered, and using previously typed symbols as model input. In order to obtain training data, I develop an *errorful typist model* to transform an existing text corpus into one that contains character-level typing errors and the backspace entries required to repair them.

I then apply various character-based language models to modified text corpora that have been generated under a range of error parameters. My results show that an LSTM-based model outperforms a traditional ngram-based language model in this task, and further that this neural model is capable of generalizing certain semantics of the backspace key itself.

It is useful to ask why such a model would be worth investigating when the Full History Fusion technique performs relatively well. There are a few reasons for this. First, FHF requires a very specific set of inputs, and is potentially difficult to adapt to other domains or modalities. Second, neural models like LSTM are versatile in that they can easily be incorporated into other systems or composed with other models. Third, FHF makes certain hard assumptions about the user it is modeling, most notably that that user will always seek to repair errors and not proceed until the output is perfect. It may be the case that a neural model is more robust to loosening these assumptions (though this final question is largely left as future work in the context of this thesis).

7.2 Model Setup

Because standard text corpora do not include backspace, they cannot be used directly to train a model that is aware of backspace. Another way of putting it: There is no backspace in the Wall Street Journal (despite the authors of that text surely having used backspace in the creation of its articles). My proposal to solve this problem is to augment a given corpus with *simulated* backspaces, as outlined in Section 7.3. The result is a training corpus that contains an alphabet of both ordinary printable characters *and* a special backspace symbol. This modified corpus is then used as input for the language models.

Note that this approach is different from full history fusion, where the language model has no concept of backspace, and the probability of backspace is computed by carefully using past user input evidence to calculate the sum of all possible strings that do not match the current one (see Equation 4.6). In this new approach, user input is not required to estimate the probability of backspace; the model learns it during training.

7.2.1 Baseline Models

I used two baseline language models. The first was a very simple baseline based on the heuristic method used in the RSVP Keyboard system. In this model, backspace is given a fixed probability p_b and the remaining symbols are assigned probabilities from an LM *not* trained on backspace. These non-backspace probabilities are scaled to sum to $1 - p_b$. Note that, while this baseline contains a fixed backspace probability like the RSVP Keyboard, it is distinct from it. Unlike the RSVP Keyboard standard method, there is not necessarily any user input collected during the testing of this baseline.

For my second baseline, I trained standard ngram language models on *error-augmented* corpora (see Section 7.3), which are created by simulating a typist who makes errors and corrects them while typing an existing non-errorful corpus. These augmented corpora contain the special symbol `*` corresponding to the backspace key, though the training algorithm is functionally unaware that this key carries special meaning. Following Carpenter [2005b], we regularize these character-based models using Witten-Bell smoothing [Witten and Bell, 1991].

7.2.2 Neural Models

The primary model used a Long Short-Term Memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997] (see Section 3.1.2 for background). Given the small vocabulary size of a character-based model, no embedding layer was needed. I simply used a one-hot encoding of each successive symbol in the training corpus as model input. I used a two-layer stacked LSTM. Regularization was achieved using a combination of early stopping, gradient clipping, and a dropout layer between the two LSTM layers.

I also explored some variants to the LSTM model. The first has to do with the *literal string*, which is the series of characters that are present on the "screen" at a given point in a typing session. For instance, if the symbols

```
m a l * * o r n i n v j x * * *
```

have been entered (`*` is the backspace key), the literal string is the following:

```
m o r n i n
```

In particular, the *rightmost* symbol in the literal string is important, because it can be informative in error contexts. In the above example, three characters have just been deleted with the backspace key. The letter `n` is the rightmost character in the literal string (though it several characters from the end of the symbol string). It is reasonable that the letter `n` should have some bearing on the probability of the letter `g` coming next, for instance. In this LSTM variant, we one-hot encode the rightmost literal character and concatenate it with the input vector during training, effectively doubling the size of the input vector.

A second variant I explored is based on the *attention* mechanism developed by Bahdanau et al. [2015] and elsewhere. The basic idea of attention is to learn a set of weights that amplify the importance of certain past hidden states in order to cause the model to "attend" to those states during training. While attention is typically used in sequence-to-sequence modeling, it can also be used in language modeling. I used the LM attention scheme described by Salton et al. [2017]. In this method, a set of attention weights is obtained after each training step, and these weights are used to compute a weighted sum of hidden states from earlier in training. This combined vector is then concatenated with the most recent hidden state and the concatenated vector is passed through a softmax layer to generate the model output.

The details of the attention mechanism are as follows. As the LSTM model progresses through a sequence of input characters (represented by one-hot input vectors $\mathbf{x}_{1 \ldots n}$), it generates a series of state vectors $\mathbf{s}_{1 \ldots n}$[4]. At time step i, we first compute $\mathbf{s}_i$ based on the standard LSTM calculation. We then generate a context vector $\mathbf{c}_i$, which is a linear combination of past state vectors, each having a scalar weight a_j:

$$\mathbf{c}_i = \sum_{j=1}^{i-1} a_j \mathbf{s}_j \tag{7.1}$$

This is where the concept of *attention* comes in; we wish to learn a way to generate these scalar weights so that past hidden states of importance to the current model decision are given higher weights. They are calculated as follows, using a softmax to ensure a sum of 1:

$$a_j = \frac{exp(\text{score}(\mathbf{s}_j))}{\sum_{k=1}^{i-1} exp(\text{score}(\mathbf{s}_k))} \tag{7.2}$$

[4] Note that, since this is an LSTM, the state vector consists of two parts, the hidden state and the cell state, but for the purposes of this attention algorithm they are concatenated into one vector.

The score function is at the heart of the attention mechanism, as it determines the relative importance of the state vector. It is calculated using the dot product

$$\text{score}(\mathbf{s}_j) = \mathbf{v}_r \odot tanh(\mathbf{W}_r \mathbf{s}_j), \tag{7.3}$$

where $\mathbf{W}_r$ is a parameter matrix and $\mathbf{v}_r$ is a vector, both with values learned during training.

Once the context vector $\mathbf{c}_i$ is calculated, it is concatenated with the most recent state vector $\mathbf{s}_i$ (which, notably, had been absent from the attention calculation so far) and multiplied by another weight matrix and adjusted by a bias vector. After passing through an activation function, this creates a new attention-aware state vector $\mathbf{s}'_i$:

$$\mathbf{s}'_i = tanh(\mathbf{W}_c[\mathbf{s}_i; \mathbf{c}_i] + \mathbf{b}_c) \tag{7.4}$$

Finally, the output vector $\mathbf{y}_i$ can be calculated as in the typical RNN formulation, using the standard weight matrix and bias vector:

$$\mathbf{y}_i = \mathbf{W}\mathbf{s}'_i + \mathbf{b} \tag{7.5}$$

All in all, the attention mechanism requires learning the following additional parameters: $\mathbf{W}_r$, $\mathbf{v}_r$, $\mathbf{W}_c$, and $\mathbf{b}_c$. It therefore takes longer to train and has a greater risk of overtraining.

Another variant I tried was including a embedding layer between the one-hot input vector and the LSTM, but the input vectors were so small (fewer than 30 entries) that I did not observe any improvements in performance.

7.2.3 Errors in Training and Test

One extremely important fact about “errorful” corpora is that not all symbols in them are useful in the context of sequence modeling. In particular, some of the symbols in an errorful corpus are the result of extremely noisy, often random processes (see Section 7.3 for details). As a result, it is not logical to train a model that is capable of predicting these error symbols, nor is it reasonable to judge a model based on its ability to predict such symbols. This led me to modify the training and testing protocols.

In LSTM model training, each training step has an input vector $\mathbf{x}_i$ (corresponding to the symbol at position i) and an output target $\mathbf{y}_i$. Since we are modeling language, $\mathbf{y}_i$ is typically equal to $\mathbf{x}_{i+i}$, i.e. the next symbol. However, in the case of an errorful corpus, we modify the target $\mathbf{y}_i$ as follows:

$$\mathbf{y}_i = \begin{cases} \text{correct\_next}(\mathbf{x}_i) & \text{if } \mathbf{x}_{i+1} \text{ is an error} \\ \mathbf{x}_{i+1} & \text{otherwise,} \end{cases} \tag{7.6}$$

where correct_next($\mathbf{x}_i$) is defined as the *correct* next symbol after $\mathbf{x}_i$, i.e. the symbol that would have followed $\mathbf{x}_i$ had an error not occurred. This allows us to remove unwanted noise from the training of our LSTM, and prevent the model from allocating probability mass to events that only occur in error (and are therefore both useless to predict and essentially impossible to predict).

Interestingly, this training modification scheme is *not* easily applicable to the standard ngram baseline model. This is because ngram model training methods are based on counts of ngrams and their prefixes. For example, suppose we encounter the ngram `thq*` during training, and we know the letter `q` is an error and should have been an `e`. A typical ngram training algorithm counts how often `thq*` occurred in training compared to the number of times its prefix `thq` occurred in training. However, suppose we performed a modification similar to that used in the LSTM model. In such a scenario we would have transformed `thq` to `the` when that trigram was being counted, but the tetragram `thq*` would be unmodified during counting since its rightmost symbol—backspace—is correct. This could cause the existence of an ngram that occurs *more often than its prefix*. In practice, this situation seriously violates the underlying assumptions of an ngram language model, and causes most smoothing algorithms to fail outright[5].

One possible way to train a standard ngram model under this modified counting scenario would be to count prefix ngrams differently depending on what part of the model calculation one is doing. For example, when calculating the conditional probability $P(*|\text{thq})$ one could use the *unmodified* count of the number of times the trigram `thq` appeared, but when calculating the probability $P(\text{q}|\text{th})$ the counts of that trigram would be decreased. However, this would require maintaining two separate counts of each ngram, and a special model architecture within a transducer-based model framework. As this was a baseline and not the focus of this work, I did not pursue this line of inquiry.

Fortunately, errors in the test set are significantly easier to handle. They are simply not included in the evaluation calculation, i.e. any predictions of a symbol that is known to have been generated by an error process are skipped over. Doing so is very simple, provided one keeps a record of the position of every errorful symbol in the test corpus. This is possible for ngram and LSTM models without complication.

7.2.4 Data Augmentation

One interesting fact when generating errorful training corpora as in Section 7.3 is that the errors are generated by a random process. This means that corpora generated with different random

[5] Notably, Kneser-Ney smoothing [Ney et al., 1994] does not fail, but it creates a model so deeply flawed as to be useless.

seeds will have errors that occur in different locations in the training set and consist of different characters. We can therefore create multiple errorful versions of a given corpus, each based on the same original data set but manifesting slightly differently. For instance, here are three versions of the same sentence, generating errors with different random seeds:

```
call_mg*e_ishmael

we**call_x*me_o*ishmael

cav*p*ll_me_it*shmaeg*l
```

I found that it is possible to leverage this fact to get improved performance. In the case of LSTM, each training epoch uses a version of the training corpus that has been generated using a different random seed. For the ngram baseline, we aggregate our counts over several different versions of the training corpus. This is akin to the data augmentation technique described in Simard et al. [2003], used in that paper for image processing but applicable for text as well. Figure 7.1 illustrates this effect on an LSTM model. Using the same errorful corpus for each training epoch results in a model that peaks at a lower performance level than the model trained on a slightly different corpus each epoch. This is a nice side effect, as it allows us to learn more about backspace without having to collect more training data. It also fits well into the LSTM training regime, since multiple epochs are typically required for training anyway.

7.3 Simulated Data Generation

The basic model for generating the augmented text corpora was an Errorful Typist Model (ETM). Under ETM, we simulate a person at a keyboard who is attempting to "type" a given text corpus. Importantly, the simulated errors made in this model are single-character stochastic errors. We are deliberately *not* considering other error types, such as misspellings based on incorrect knowledge or accidentally typing the wrong word before reconsidering and deleting it. We also exclude other reasons the backspace symbol might be pressed, such as deleting an entire sentence when one wishes to word it differently.

7.3.1 Model States

The ETM algorithm can be thought of in terms of states. In it, there are two possible model states, consistent state and inconsistent state. If there are any errors in the typed text, by definition the model is in the inconsistent state. Similarly if there are no errors, the model is in the consistent

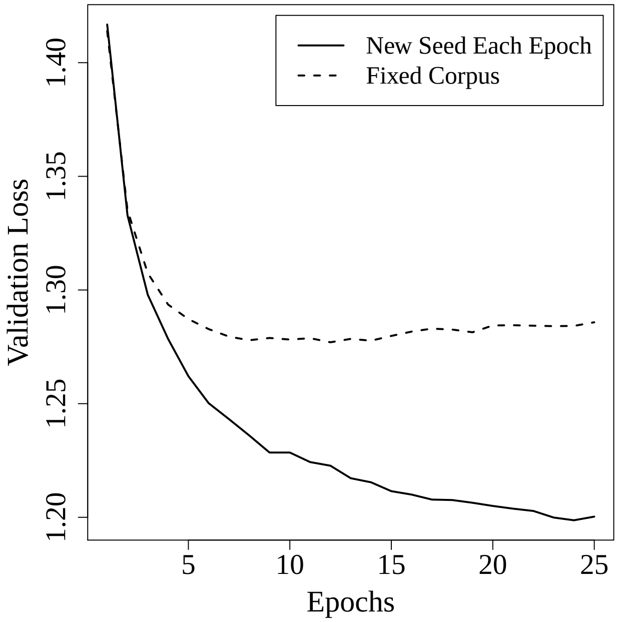


Figure 7.1: Validation performance for model with fixed corpus versus model with newly-generated corpus each epoch

state. There are two courses of action from the consistent state: Committing an error and typing a correct character. There are two courses of action from the inconsistent state: Typing *another* error and typing the backspace key.

Typing proceeds in timesteps. At each timestep the model decides what to do next based on the current state and various probabilistic parameters. If the model is in the consistent state, it commits a new error with *error probability* p_e and types the correct letter with probability $1 - p_e$. A new error takes the form of a single incorrect character being typed. If the model is in the inconsistent state, it “notices” that state with *notice probability* p_n. When this happens the backspace key is pressed and the rightmost typed symbol is removed. If the inconsistent state is not noticed, a new error character is typed.

Under this formulation of ETM the backspace key cannot be typed in error. We can remove this restriction by adding a *backspace error probability* parameter p_b. Under this paradigm, when the model is in the consistent state *and* decides with probability p_e to make an error, that error will be an (unnecessary) backspace with probability p_b. Note that backspace is never an error when the model is in the inconsistent state, so this only happens in the consistent state.

If $p_e < 0.5$ and $p_n > 0.5$, ETM will eventually generate a new errorful corpus corresponding to a simulated typist copying the source corpus and ultimately repairing all typing errors.

7.3.2 Error Modes

There are two error modes possible in ETM. The first is `uniform`, which means that error characters are sampled uniformly from the alphabet $A_{\hat{c}}$, which corresponds to every symbol in the model vocabulary except the correct symbol and backspace. The second error mode is `ngram`. In the `ngram` error mode, error characters are sampled from the same alphabet but are biased via a small ngram language model[6] conditioned on the rightmost characters in the literal string. This error mode can be considered more difficult to model, as the errors tend to be more "plausible"[7].

7.4 Experiments and Results

7.4.1 Corpora and Model Details

The primary text corpus was the Wikitext-2 corpus introduced by Merity et al. [2016], consisting of 2M words of training data and validation/test sets each about 200K words in length. I normalized the corpus to only include the letters `a-z`, the `space` character, and special `start` and `end` symbols (the latter two bookending paragraphs in the corpus). This gave us a vocabulary size of 29 (not including backspace, which was added later).

For the ETM corpus generation, I experimented with a variety of parameter combinations. I set the base error probability in the 5-15% range, which corresponds to error rates observed in user typing studies [Dhakal et al., 2018]. As a rule I also set p_n to be equal to $1 - p_e$. This made some sense and resulted in reasonable error behavior. In experiments where errorful backspaces were possible we set p_b to be equal to p_e.

All LSTM models were trained for 20 epochs with a differently-seeded version of the training corpus for each epoch. For ngram models, 3 different versions of the training corpus were created and the model processing was performed on the concatenation of these versions. The reason I did not use 20 augmented copies of the training corpus for ngram models was that it created extremely large models that were impractical to compute. Including errors in text via the ETM algorithm greatly increases the number of ngrams observed in that text, because errors and their corrective backspaces can occur anywhere within a given ngram. I also observed that ngram models did not appear to improve much in performance after 2-3 augmented copies were concatenated.

Based on parameter optimization experiments on the validation set, the LSTM parameters

[6] Specifically, a 6-gram character-based model trained on the Wall Street Journal.

[7] Notably, the `ngram` error mode mirrors errors encountered frequently in the RSVP Keyboard protocol, since all typing actions in that system are informed by an LM prior.

		LSTM				ngram baseline			
p_e	p_n	overall	bksp	non-bksp	after-bksp	overall	bksp	non-bksp	after-bksp
0.05	0.95	1.82	1.40	1.84	1.96	2.05	2.08	2.05	2.85
0.10	0.90	1.79	1.00	1.88	1.86	2.23	1.74	2.29	2.96
0.15	0.85	1.72	0.77	1.92	1.75	2.39	1.60	2.56	3.06

Table 7.1: Bits-per-character performance of models in `uniform` error mode.

were set as follows for all experiments: 1024 hidden states[8], 2 hidden layers, a sequence length of 100, a batch size of 70, and a dropout probability 0.5. Gradient clipping was performed when the combined norm of the gradients exceeded 5.

7.4.2 Evaluation

The primary evaluation metric used was bits per character (see Equation 3.8), corresponding to the cross entropy. As discussed in Section 7.2.3, I did not consider characters typed in error (including backspaces typed in error) in the evaluation calculation. I also calculated model performance on various subsets of the test corpus, including performance in situations when the correct next character is a backspace and performance when the most recently typed character was a backspace.

In order to specifically probe model performance with regard to the backspace key, I also created an additional set of metrics based on whether the model-provided probability for certain letters in specific contexts changed in expected ways (more on this in Section 7.5).

7.4.3 Primary Results and Analysis

LSTM vs ngram baseline

Bits-per-character results for corpora generated using the `uniform` error mode are shown in Table 7.1. In the LSTM models, the $\mathbf{y}_i$ value has been corrected in training as in Equation 7.6. The "bksp" columns are bpc values calculated only for test indices where the correct target is a backspace. Similarly, "non-bksp" is calculated over non-backspace target indices and "after-bksp" is calculated over indices directly to the right of backspace symbols. For all calculations, symbols typed by the ETM that are known to be errors are excluded.

There are several notable aspects to these results. The first and most prominent is that the LSTM model consistently outperforms the ngram baseline across all evaluation conditions and error rates. As I will discuss later, this is a good indication that the neural model is learning something about the behavior of backspace that simply is not captured by the ngram model.

[8] In the LSTM paradigm this corresponds to a cell state and a memory state, each having a size of 1024.

		LSTM				ngram			
p_c	p_n	overall	bksp	non-bksp	after-bksp	overall	bksp	non-bksp	after-bksp
0.05	0.95	1.91	2.78	1.87	1.82	2.07	3.37	2.00	2.24
0.10	0.90	1.92	1.92	1.92	1.77	2.20	2.59	2.15	2.38
0.15	0.85	1.90	1.53	1.97	1.73	2.31	2.18	2.34	2.50

Table 7.2: Bits-per-character performance of models in `ngram` error mode.

This difference is especially striking for the after-backspace character positions, in which knowing something about backspace (e.g. that a just-deleted character should now be less probable than it was previously) is very important.

Another thing to notice is that the LSTM models paradoxically appear to do *better* as the error rate p_e increases, the model performance actually *decreases.* This is due to two factors: First, a higher error rate means more backspaces, and in the `uniform` error mode these errors are often odd letters that are easy for the model to "notice" as candidates for deletion. This can be seen in the comparatively low bpc values at the backspace positions at the bottom row of the table. Second, under the hypothesis that the LSTM model is learning about the semantics of backspace, a higher error rate also means more after-backspace symbols, in which a "smarter" model will do better because it now knows more about what the correct letter might be than it did before a bad letter was deleted. It is notable that non-backspace symbols are still more difficult to predict under higher error rates.

Bits-per-character results for corpora generated using the `ngram` error mode are shown in Table 7.2. Recall that in this error mode, the errors are biased toward more "likely" characters using a small language model during ETM generation. Model parameters and evaluation calculations are the same as those in Table 7.1. Here we see somewhat comparable results to the `uniform` error mode results, in that the LSTM model outperforms the ngram baseline consistently across all dimensions.

There are some interesting differences in these results, however. The easiest to explain is that the models generally perform worse under the `ngram` error mode than under the `uniform` error mode. This makes sense, as the errors are biased toward real ngrams and are therefore more "devious" and harder to spot, making the prediction of the backspace symbol in particular more difficult to do. This can be seen in the striking difference in bpc for the backspace symbol. Interestingly, the ngram baseline didn't do much worse in the `ngram` error mode, actually performing slightly better under some conditions than it did in the `uniform` error mode. I hypothesize that this is because ngram models like ngrams: Since this error mode is more likely to result in common error patterns that appear repeatedly in the training corpus, an ngram model is going to encounter those

train		test		overall	backspace	non-backspace	after-backspace
p_c	p_n	p_c	p_n				
0.05	0.95	0.05	0.95	1.82	1.40	1.84	1.96
0.10	0.90	0.05	0.95	1.82	1.01	1.86	1.88
0.15	0.85	0.05	0.95	1.83	0.81	1.89	1.87
0.05	0.95	0.1	0.9	1.83	1.43	1.88	2.01
0.10	0.90	0.1	0.9	1.79	1.00	1.88	1.86
0.15	0.85	0.1	0.9	1.78	0.80	1.90	1.81
0.05	0.95	0.15	0.85	1.86	1.48	1.94	2.08
0.10	0.90	0.95	0.85	1.75	0.99	1.91	1.84
0.15	0.85	0.15	0.85	1.72	0.77	1.92	1.75

Table 7.3: Bits-per-character performance of LSTM models in `uniform` error mode, train/test different.

same patterns in the test corpus and will be able to predict them without learning anything clever.

Finally, it is notable that the overall bits per character did not improve as the error rate increased. This appears to be because backspaces did not become easier to predict to the same extent that they did in the `uniform` error mode results. As a result, the greater prominence of errors did not ultimately make the overall prediction performance any better.

Error Rate Mismatch

It is important to note that the *rows* in Tables 7.1 and 7.2 are not directly comparable in certain ways, as they represent different test sets generated under different error rates. To control for this, I performed a series of experiments in which the training error rate changed but the test set remained the same. The results are shown in Table 7.3 for the `uniform` error mode. Interestingly, it appears that models tested on a set with a certain error rate, but trained on a set with a *higher* error rate, perform relatively well. In particular, performance on backspaces and after-backspace symbols is understandably better when the training data contains more errors, but at a very small evident cost on non-backspace symbols. Further, training on an error rate that is too *low* can actually give bad results. This suggests that in real-world situations, where the true error rate is likely not known, it may be wise to err on the side of a higher error rate in training to avoid risking serious performance degradation. Similar results can be seen for the `ngram` error mode, shown in Table 7.4.

These error rate results echo the findings reported in Dong et al. [2019]. That paper asked a slightly different question, seeking to determine the *noise tolerance* of various character-based models in a BCI typing domain. The backspace key is not part of these formulations; rather the model represents a situation in which errors in the typed history persist without being corrected.

train		test		overall	backspace	non-backspace	after-backspace
p_c	p_n	p_c	p_n				
0.05	0.95	0.05	0.95	1.91	2.78	1.87	1.82
0.10	0.90	0.05	0.95	1.91	1.89	1.91	1.73
0.15	0.85	0.05	0.95	1.95	1.53	1.97	1.74
0.05	0.95	0.1	0.9	2.01	2.88	1.90	1.97
0.10	0.90	0.1	0.9	1.92	1.92	1.92	1.77
0.15	0.85	0.1	0.9	1.92	1.53	1.97	1.74
0.05	0.95	0.15	0.85	2.14	3.00	1.96	2.16
0.10	0.90	0.95	0.85	1.94	1.96	1.94	1.82
0.15	0.85	0.15	0.85	1.90	1.53	1.97	1.73

Table 7.4: Bits-per-character performance of LSTM models in `ngram` error mode, train/test different.

Building on comparable grammar correction [Xie et al., 2017] and machine translation [Belinkov and Bisk, 2017] results, they confirmed that LSTM and ngram language models trained on *clean* text performed quite poorly on test sets containing noise. They mitigated this problem by injecting noise into the training data, replacing characters with uniformly-distributed errors with a given probability. These models trained on noisy data performed much better. Interestingly, they found a somewhat different result from Table 7.3 when it comes to the *relative* error rates, showing that it is acceptable in their formulation to use training data with a *lower* error rate than the test data (they cite the example of a model trained with 10% noise performing relatively well on tests with up to 40% noise). It is unclear why this is the case, though it is probably due to their different formulation that does not include a special backspace symbol in training or test.

Other Baselines

The fixed-backspace baseline model described in Section 7.2.1 performed so poorly that it is not included in the main results above. Table 7.5 shows its performance compared to the ngram baseline. For these results, a 12-gram model was used with no augmented copies added. The `uniform` error mode was used to generate the corpora. For each model, the fixed backspace probability was set equal to the corpus error rate p_e, which is logical as the error rate and the backspace rate are typically extremely close to each other. These weak numbers makes sense, as this baseline does not consider any contextual information. Its probability of backspace is fixed and therefore cannot increase after obvious errors or decrease inside strings of text that are very likely to be correct.

All of the models described so far were trained on ETM-generated corpora in which the backspace key itself is never an error. I created several models where I allowed backspace to

p_e	p_n	ngram baseline	fixed backspace baseline
0.05	0.95	2.07	2.54
0.10	0.90	2.29	3.06
0.15	0.85	2.47	3.41

Table 7.5: Bits-per-character overall performance on the two baseline models.

be an error by setting a non-zero value of p_b, the backspace error probability. Interestingly, the performance of these models on their respective test sets was very consistent: Equal in bpc to the no-backspace-error models or very slightly worse across all evaluations. Nowhere did I observe a model with errorful backspaces that had notably different performance from its counterpart in the perfect-backspace world. This absence of a striking difference can be considered an interesting result in and of itself, as it suggests that these models are perfectly happy to allow backspace to occur in error with only a slight decrease in performance, if any.

7.5 Other Variants and Analysis

The two main model variants I investigated were the attention-based model and the literal-string model, both described in Section 7.2.2. Disappointingly, both of these models gave results that were not significantly better than those of the the standard LSTM model.

It is not immediately clear why these variants did not outperform LSTM. Attention in particular seems uniquely suited to the backspace prediction problem, as knowledge of specific previous states is particularly helpful in estimating symbol probabilities when errors and backspaces are possible. In the case of attention, it may simply be that the LSTM model's memory apparatus, combined with the large number of hidden units, was able to adequately capture something equivalent to what attention provides, making it not help much. The small prediction space of 30 characters may also have contributed to the ability of the LSTM to learn and keep useful information. An interesting avenue for future work would be to examine the contribution of attention in a weaker model, either by decreasing the number of LSTM hidden states or by switching to a Standard RNN architecture.

Another potentially useful idea would be an analysis similar to that done in Weiss et al. [2018], which examined what can actually be learned by an RNN. They were able to show that finite RNNs are capable of certain computational functions such as counting, and in particular were able to directly examine the model weights to show this effect visually. Such an analysis might be used to illustrate what the LSTM is learning in the backspace model. However, I am not entirely

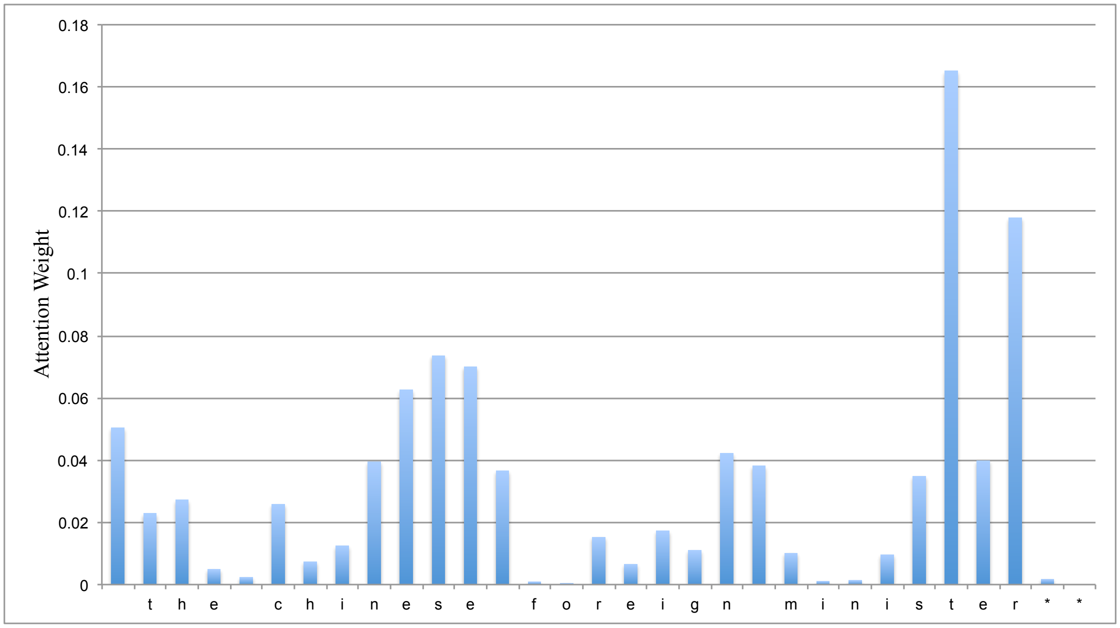


Figure 7.2: Well-behaving attention weights, after typed string `the_chinese_foreign_minister**`.

convinced that visual inspection of model weights would be useful. In particular, there are certain aspects of backspace (for instance the fact that a symbol should become less likely after it has been deleted once) that don't immediately lend themselves to visually informative model parameters. The models in the aforementioned paper consisted of very small vocabularies, so something general and contextual like "discount the symbol two letters back if the rightmost symbol is a backspace" might not be clearly evident in model weights when there are dozens of symbols. This remains an empirical question.

In a similar vein, I did an analysis of the actual attention weights $a_{1 \ldots n}$ learned by the attention algorithm, as defined in Equation 7.2. The results were mixed. In some cases, the attention weights seemed to be higher for states associated with past letters that are important in the calculation of the next symbol. An example is shown in Figure 7.2. Here the typed symbol sequence is `the_chinese_foreign_minister**`, the two `*` symbols representing two backspaces. In this case, the backspaces suggest that the last two letters in `minister` are likely to be errors, suggesting another continuation like `ministry`. We see that the attention model appears to be aware of this, focusing its greatest attention on the `t` symbol (technically its associated hidden state), a very important letter if one is to predict a new continuation of that word.

However, the attention weights did not always align themselves in such a logical way. Figure 7.3 is a good example. Here the typed symbol sequence is `the_amazo*`, the rightmost letter `o` having

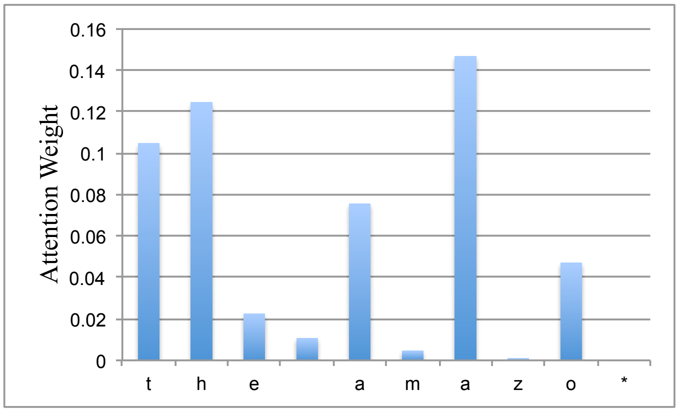


Figure 7.3: Poorly-behaving attention weights, after typed sequence `the_amazo*`.

been deleted with a backspace. We therefore have the literal string `the_amaz`, which has reasonable alternate branches like `amazing` and `amazed`. However, the attention weights are not focused on the `z` position, which contains the hidden state that would seem to be most important in this situation. I found several examples of both kinds, some where the attention model did logical things and others where it did not. One explanation for this is that it may well be more evidence that the LSTM is simply capturing most of the information that would make attention contribute to the model, so the attention weights become not very informative.

As for the literal-string variant, I am inclined to make a similar argument: It is possible that the LSTM is simply already learning the information contained in the rightmost-literal string vector (which is concatenated to the input vector in this variant), and therefore does not gain anything by adding that feature. A similar gradual weakening of the LSTM model to explore this trade-off, as discussed above regarding the attention model, is an interesting idea for future work.

Deeper Backspace Semantics

One thing I analyzed in depth was whether the models in question were learning patterns that reflect the *meaning* of backspace in a general way. The most basic of these is the idea that a character should become less likely after it is deleted with a backspace. But this can be extended further. For example, we can look at chains of backspaces like this:

```
y o l k * * * *
```

Here, four consecutive characters have been deleted, and we are in a situation where, despite the fact that the letter `y` is now eight positions back in the symbol string, it should still be less likely

model	p_e	p_n	a*	ab**	abc***	abcd****	a*b*
LSTM	0.05	0.95	78.5	50.5	41.4	–	70.0
	0.10	0.90	82.9	58.6	44.8	36.0	81.4
	0.15	0.85	84.5	66.0	55.8	49.6	86.2
ngram	0.05	0.95	69.9	24.5	20.6	–	59.1
	0.10	0.90	66.7	36.9	22.6	29.5	62.3
	0.15	0.85	66.2	46.9	25.8	24.5	65.0

Table 7.6: Percentage of predictions with decreased probability after pattern, `uniform` error mode.

model	p_e	p_n	a*	ab**	abc***	abcd****	a*b*
LSTM	0.05	0.95	97.9	78.3	64.6	55.6	98.1
	0.10	0.90	98.8	90.5	77.3	67.4	99.1
	0.15	0.85	98.6	94.0	85.6	71.1	98.6
ngram	0.05	0.95	81.9	73.8	58.6	55.6	83.1
	0.10	0.90	77.0	74.7	72.7	73.3	75.1
	0.15	0.85	71.5	69.5	71.4	73.9	68.5

Table 7.7: Percentage of predictions with decreased probability after pattern, `ngram` error mode.

than it was when it was first typed. I examined five such patterns[9]:

```
a *

a b * *

a b c * * *

a b c d * * * *

a * b *
```

In each of these cases, I expect the probability of the first character to be lower after this sequence than it was at the beginning. In the first four patterns, an increasingly long chain of characters has been deleted by backspace. In the last pattern, one character has been typed and deleted, then another.

I looked for these patterns in the test set, then compared the model's prediction of the probability of the first letter *before* the pattern was typed to its prediction *after* the pattern was typed. I then counted the proportion of predictions where the probability estimate went *down*, the expectation being that a model aware of the semantics of backspace should have a large such proportion. The results are shown in Tables 7.6 and 7.7. The LSTM and ngram models are the same as those

[9] Note that letters are representative here; the letter in position `a`, for example, can be any character except a backspace.

used in Table 7.1. Table 7.6 is missing two values because the pattern abcd**** only occurred four times in the test set.

There are many phenomena of interest here. Foremost is that the LSTM almost always outperforms the ngram baseline model at this metric. This is a good indication that the LSTM is learning something about backspace that the ngram model simply cannot. Second, the models appear to do better when there the error rate p_e is higher. This is logical, because the patterns in question are more likely to have been observed in training when there are more errors in training. Third, the models tend to perform worse as the patterns get longer. Again, this is logical; the models have to look farther back into the past to obtain the first letter that was deleted.

Of particular interest is that the results for the uniform error mode corpora are quite a bit worse, for both models, than the results for the ngram error mode corpora. I hypothesize two reasons for this. First, the corpora with ngram-based errors contains a smaller variety of errors, since they are biased toward likely characters. Therefore the patterns observed in test are more likely to have been observed in training. Second, when a “likely” error occurs, it tends to have a relatively high probability, so it is easier for the probability of that character to go down after it is deleted.

This latter concept, that high-probability error characters are easier to make less probable, is an idea I explored in some depth. It is evident from Table 7.6 that while the LSTM model tends to outperform the ngram model at this metric, it is not doing particularly well in many cases. Indeed, for certain patterns it is decreasing the character probability less than half the time, which is worse than chance! I analyzed this by charting the before and after probabilities of the characters in question, denoted by a in the above patterns. The results are shown in Figure 7.4. Each point in this figure represents an observation of the pattern ab** in the test set. The x-axis is the model probability of the character in position a before the pattern, the y-axis is that same character’s position after the pattern. This figure is in the negative log domain, so points to the lower left of the graph represent characters that are more probable. Points above the red line represent cases where the model decreased the probability of that character after observing the pattern. The model in question is LSTM, and the error rate p_e is equal to 0.1.

It is evident from this scatter plot that the model is much better at decreasing the probability of the character in question when its initial probability is high. Overall, in the 249 times the pattern ab** appears in the test set, the model only decreased the probability 58.6% of the time. But in in the 93 cases where the before-pattern probability is greater than 0.001 (points left of 3 on the x-axis in the plot), the model successfully predicts a decreased after-pattern probability 90% of the time. The story is even better when we limit to a before-pattern probability of 0.01, where

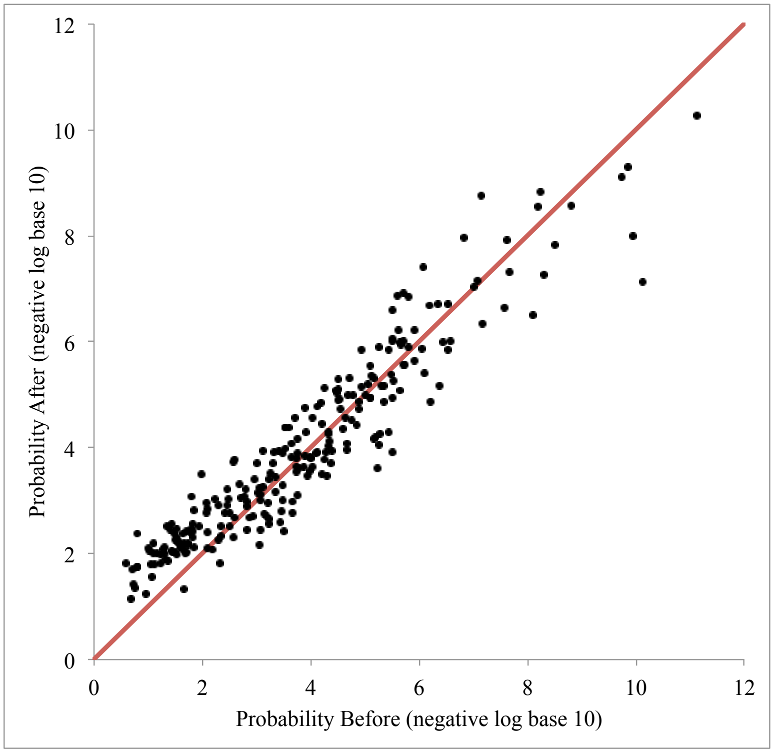


Figure 7.4: Before vs. after probabilities for pattern `ab**`, `uniform` error mode.

	error mode	
forbidden letter	uniform	ngram
e	62.4	79.0
m	40.6	36.0
z	17.3	59.4

Table 7.8: Percentage of predictions with decreased probability in LSTM model after pattern `a*` with specific characters forbidden as errors in training data.

the model succeeds 98% of the time (all but one case). This suggests that the model is really only failing to decrease probability when the initial probability is very low in the first place. In such a scenario, it is likely that decreasing the probability in question is not actually a particularly useful thing for the model to learn, as the character is very unlikely to be typed again in that position. This also explains why the numbers appear better for the `ngram` error mode corpus; there are more points that are initially likely under that error mode.

Forbidden Errors

The superior performance of the LSTM model over the ngram baseline in the decrease-probability-after-backspace metric is a good indication that the LSTM model is learning backspace semantics, but I wanted to explore a more concrete generalization. To do this, I created a special training set where I prevented certain characters from ever being typed in error. In the corresponding test set, however, I *did* allow those characters as errors. For example, suppose the letter `p` is barred from being an error in the training set, but in the test set we see the string `p*`. If a model is able to decrease the probability of `p` in this scenario, it will have truly demonstrated its ability to generalize a key semantic aspect of the backspace key.

I selected three individual letters for these experiments, forbidding them one at a time from appearing in specially-constructed training sets. I selected `e`, `m`, and `z` in order to test a range of base unigram commonality in English. The results are shown in Table 7.8. In this table, the percentage given is only calculated when the character in position `a` is the forbidden letter in question.

One very interesting result from this table is that, at least for the letter `e`, the LSTM model is apparently able to generalize a basic semantic attribute of backspace even when the letter in question has never been deleted in training. This is a very nice demonstration of the power of RNNs to learn patterns. The results for the other two letters are less conclusive, and more confusing. For the letters that did not decrease in probability, I hypothesize that the models are falling back on some kind of baseline probability value, since they are in an unseen context. The `ngram` error mode value for `z` is a bit of a mystery. It is possible, since `z` is so rarely a "likely" error (as occurs in that corpus), that when it does occur as an error it has a relatively high probability and the model is able to compute a lower probability for it after the pattern.

7.5.1 Future Work

Two paths for future work have already been described: Better analyzing the reasons why the attention-based model did not do better, and better analyzing why the literal-string-augmented models did not do better. The idea of limiting the base model until it no longer captures what is provided by these methods is an interesting possibility.

It would also be interesting to further explore how, and to what extent, the LSTM model is learning backspace semantics. This could be accomplished by creating more novel training scenarios. For instance, one could build a corpus where the *only* symbol typed in error is the backspace key. This is akin to a stack-like data structure, as any deleted symbols are removed in a Last-In First-Out manner. In this formulation, the probability of the symbol in position `a*` would be expected to *increase* rather than decrease.

I experimented briefly with limited-vocabulary models, e.g. models that contain only one character, or two characters. This would be an interesting thing to analyze, as one could better understand the memory-related aspects of backspace in a scenario where all the characters in question are ones and zeros, for instance.

Chapter 8

Personalization in Text Entry

In this chapter[1], I explore something a bit different from that of the previous chapters, but that nonetheless ties into the concept of incrementality for text entry in an interesting way. Here, the input modality is a smart phone with a so-called *soft keyboard*, i.e. a keyboard that corrects spatial errors as you tap. By creating a closed loop evaluation system that simulates these spatial errors, I am able to perform large-scale experiments that can empirically measure the extrinsic performance of various language modeling techniques as they are applied to the soft keyboard modality. Of particular interest in this chapter is language model *personalization*, or adapting a model to a specific person. In the techniques explored here, a cache-based model is used to keep track of recently used words by a user and boost their probability within the language model. Notably, such a model changes *incrementally* as the user produces text, becoming essentially a slightly different model after each word. This is an entirely different way of thinking about incrementality. Whereas in previous chapters we have considered the incrementality of the input modality itself, now we consider the incrementality of the underlying model.

8.1 Introduction

Text input methods have often been key features in shaping the paradigm shifts in mobile computing in the last decades. For example, handwriting with a modified character set (Graffiti) was central to the Palm Pilot era of PDAs. Miniature thumb keyboards dominated the design of BlackBerry and similar devices. Smart touch keyboards (STK) enabled the current generation of full touchscreen mobile devices.

[1] Significant parts of this chapter were originally published as Fowler et al. [2015]. As the primary author of that paper, my personal contribution was significant, including the design and implementation of the simulation system, preparation of corpora, and all of the subsequent analysis. However, I do not wish to minimize the contribution of my co-authors, who provided helpful guidance and technical expertise, crucial editorial feedback, and assistance with the background references and research.

STK products rely on language models, or least a dictionary, to correct touch errors, autocomplete partial letter strings to complete words, and predict what the user will type next. Surprisingly (and for methodological reasons discussed later) the quantitative effects of applying language modeling to touch-based STKs have never been formally reported in the research literature. In fact many recent studies of touch screen typing do not involve LMs at all (e.g. Findlater et al. [2011]). Of concern is that in many cases, evaluation without language-model based correction may lead to incorrect conclusions about the benefits of a particular technique. For example, Yin et al. [2013] found that improvements in spatial error modeling had little effect in practice since LM-based correction compensated for such spatial errors. For this reason, quantitatively benchmarking the effects of a standard LM to STK is one of the goals of this chapter.

Conversely, traditional metrics of language models in isolation may not accurately reflect how they perform in a deployed decoder. A model-intrinsic measure like per-word perplexity (see Section 3.1.3), though commonly used to compare LMs, can correlate poorly with downstream metrics, as is the case with word error rate in speech recognition [Chen et al., 1998]. *Extrinsic* evaluations like typing accuracy, which more closely reflect the experience of an actual user, can only be evaluated if the user's role in the decoding process is considered. This is similar to the rationale of the EEG simulation experiment presented in Chapter 5.

An especially relevant language modeling technique for mobile text entry, and one that is also difficult to measure, is *LM personalization*, in which a language model's lexicon and probabilities are made flexible and adaptive to the user. A natural way to accomplish this is to utilize a user's written history, since it stands to reason that this would best reflect that person's usage of language. However, scientific understanding of text input LM personalization is limited. The most detailed work dates from 2006, and studies previous-generation mobile devices with physical keyboards [Tanaka-Ishii, 2007]. Modern soft keyboards, with more fuzzy input on touchscreens, may rely more heavily on LMs, and therefore are more likely to benefit when personalization works, or suffer when it does not.

This chapter represents the first large-scale longitudinal simulation study of the effects of language modeling, with and without personalization, on finger-touch-based soft keyboards (as published in Fowler et al. [2015]). As a methodological contribution, I helped to create a novel integrated framework (dubbed Sketch Jr.) for testing input *accuracy*, which differs from intrinsic measures such as LM perplexity and keystroke savings, which are typically computed in isolation of the input problem. This method allows for *extrinsic evaluation* of models by incorporating large-scale simulation of human-like inputs. We can therefore evaluate the LMs in connection with sloppy user input based on empirical data, using a principled error correction and word completion model (decoder). I use the longitudinal real-world text-writing history of a large group of individuals (the Enron Corpus). Through this human simulation, I show that a background LM can reduce typing word error rate (WER) from 38.4% to 5.7%, and that personalized LMs further reduce this error to 4.6% (a 19.3% relative change compared to the non-adapted model). I also examine personalization methods based on the idea of an exponentially-decaying cache, and find that a decay time in months is optimal, but that the decay curve has little effect on the overall performance result. This work is also the first to study and quantify the trade-off between the overall error rate and *false corrections*, a subclass of errors highly visible to end-users.

For background information and related work regarding language model adaptation and its use in text entry, please see Sections 2.2 and 2.4.

8.2 Research Methodology

Two crucial questions in the experimental design were whether to primarily rely on modeling and simulation or human subjects directly, and whether to collect text from users or use an existing corpus. The final decision was to simulate using a large real world text corpus (Enron) for all experiments, for the following reasons:

- **Practicality and Scale:** Scale is important to language modeling evaluation, particularly with personalization. Variances and nuances in language use can only be revealed with longitudinal data. Further, as later confirmed, large user-to-user variance is possible and many users are needed to give meaningful results. Collecting this amount of data from individual users is possible, but impractical. In addition, we required a large development set in order to empirically optimize various system parameters. Large-scale parameter optimization of this kind without simulation is infeasible.

- **Privacy and Reproducibility:** Another problem with collecting native text from individuals is that they would have a reasonable expectation of privacy. The Enron Corpus largely avoids this issue, and its widespread availability allows for experiments that are vastly more reproducible.

- **Availability of Foundational Research:** Recent Human-Computer Interaction (HCI) research has measured crucial parameters of human motor control for text entry tasks. It has been shown that tapping on key targets may be modeled by a Gaussian distribution model whose parameters depend on hand posture (finger, thumb, or two thumbs), the portion of the keyboard (left, right, top, or bottom), and the individual user [Findlater et al., 2011, Azenkot and Zhai, 2012, Bi et al., 2014a]. This means that the tapping model used to simulate spatial errors has a grounding in the literature.

I do not wish to discount the advantages of human tests, which would provide valuable information not available to simulation, such as how users respond to personalized text input and what strategies are used to correct errors. Human tests would also validate the simulation approach. These questions are reserved as important avenues for future work; for the present work, however, this chapter asserts that combining human input models established from users' natural typing data and longitudinal real world text writing records captures the essence of smart touch keyboard evaluation and reveals the effects of language modeling at a large scale.

8.3 Data

8.3.1 Background Language Model

The background model was a Katz-smoothed [Katz, 1987] bigram LM trained on 114 billion words scraped from the publicly-accessible web in English. Notably and unlike the models used in previous chapters, this is a word-based model and not a character-based one. The model was pruned using entropy pruning [Stolcke, 1998], a technique for decreasing the size of a backoff LM. The basic method involves selecting ngrams for removal from the model by measuring the relative entropy of models with and without that ngram. A threshold is set for excluding ngrams by this criterion. The choice of Katz over the more popular Kneser-Ney [Kneser and Ney, 1995] for smoothing was dictated by the small memory footprint available for the LM on a mobile device, and the reduced difference between the two approaches after entropy pruning [Chelba et al., 2010]. The final pruned model contained about 8.5 million ngrams (8.4M bigrams and 168K unigrams).

8.3.2 Enron Corpus as an Evaluation Set

The Enron Corpus [Klimt and Yang, 2004] is a large set of emails that were collected by the Federal Regulatory Commission when the Enron Corporation was under investigation in late 2001. This corpus was made public in 2003, and consists of over 600,000 individual email messages spanning multiple years (1998 to 2002). It is one of the largest collections of email publicly available, and is a valuable and widely-used resource for natural language research.

In the field of text entry, Enron data has been used for the creation and comparison of phrase sets to be used in other text entry experiments [Kristensson and Vertanen, 2011, 2012, Vertanen and Kristensson, 2011a] . In Paek and Hsu [2011], Enron data was compared to data from Facebook and Twitter in an information-theoretic manner, with the goal of judging how "representative" various phrases are. Enron has also been used for domain adaptation in tasks other than language modeling [Chiticariu et al., 2010, Sandu et al., 2010, Zajic et al., 2008].

The Enron Corpus is well suited for text entry research for three reasons: First, it consists of human communication, making it a better match to text input applications. Second, it consists of a relatively large body of real-world usage of text communication (as opposed to news corpora [Charniak et al., 2000] or fiction writing [Ward et al., 2000]). Third, the corpus preserves a long history of communication by users with minimal privacy concerns, allowing for more in-depth research into personalization techniques.

Notably, I found the Enron corpus to be inadequate in its raw form. Preprocessing of the corpus was extensive, and included deduplication, signature text removal, name canonicalization, and attached text removal. Because the goal was LM personalization, the driving idea of these preprocessing steps was a desire to have clean text for each user that was strongly linked to that user, i.e. not generated automatically or typed by somebody else. This is particularly important when modeling for text entry, since URLs and attached messages are very unlikely to have been typed by the person doing the text entry. I found that extracting clean, human-generated text from the Enron Corpus is surprisingly nontrivial, so I have made the preprocessing details available in the Appendix of this thesis.

8.3.3 Development and Test Sets

The next step was to divide the Enron emails into test sets. We selected 90 Enron users, each with more than 1500 total words in their collected sent emails. I used a minimum word count per user for two reasons: First, I wanted a large enough volume of text to perform adaptation experiments that involved learning user words and adapting a model to them. Second, I wanted to avoid sparsity issues. I divided these 90 users into development and test sets of 45 users apiece, with a similar distribution of per-user word counts in each set. The development set contained 38,114 messages and 1,355,266 words, and the test set contained 31,740 messages and 1,214,403 words. Since the models are based on caching, there is no Enron training set *per se*; user models are trained dynamically as one proceeds through each user's portion of the test set.

8.4 Simulation

The basic workflow of the system is to generate simulated keyboard taps for a given text, use a decoder to reconstruct the text from those noisy taps in the presence of various LMs, then evaluate the output.

8.4.1 Simulated Smart Touch Keyboard Typing

Section 8.2 outlines the benefits of human simulation as a general research method in LM personalization. What follows are critical details needed to making such simulation meaningful to text input, as well as aspects of typing not simulated in the current work.

In order to study the impact of LM personalization in STK applications, I simulated each Enron user typing her or his test set in chronological order using a simple STK decoder, to be described shortly. I used the key layout from the specifications of the Nexus 5 QWERTY keyboard, which has a single-key width of 6.16 mm and a key height of 9.42 mm. Users were assumed to be perfect in *intent*; in other words, the user's target word is always the actual target word[2]. Additionally, if the decoder fails to produce the target word, it is simply counted towards the error rate but no simulation is made of any attempt on the part of the user to correct such errors with backspace or other methods.

To evaluate the power of LMs and personalized LMs in error correction, I introduced human errors in the simulated input process that reflected the main source of error in touch screen keyboards. Spatial noise was incorporated in the tapping/typing signal by sampling randomly from a 2D Gaussian distribution. The mean of the distribution for each tap was the center of the intended key, and the standard deviations were set to be 1.97 mm in the x direction and 1.88 mm in the y direction, as illustrated in Figure 8.1. These values were based on Azenkot and Zhai [2012], which determined the 2D Gaussian distribution to be representative of actual human tapping, and described typical variances. Taps landing outside of the border of the intended key and inside another key were given the identity of the new key in the literal string output (a typing error). Taps outside the border of *every* key were discarded and resampled. These settings result in a per-tap error rate of 12.8%, which translates to 38.4% of words in the test set containing at least one key error. In practice, the actual amount of spatial errors follows a speed-accuracy trade off model. The faster one types, the more errors one would make. See "FFitts law" [Bi et al., 2014a] for an in depth exploration of the speed-accuracy model of touch screen target acquisition containing both

[2] This does not exclude the possibility of spelling errors in the Enron Corpus; in this case, we simply assume that the typo is the correct target, and judge it accordingly.

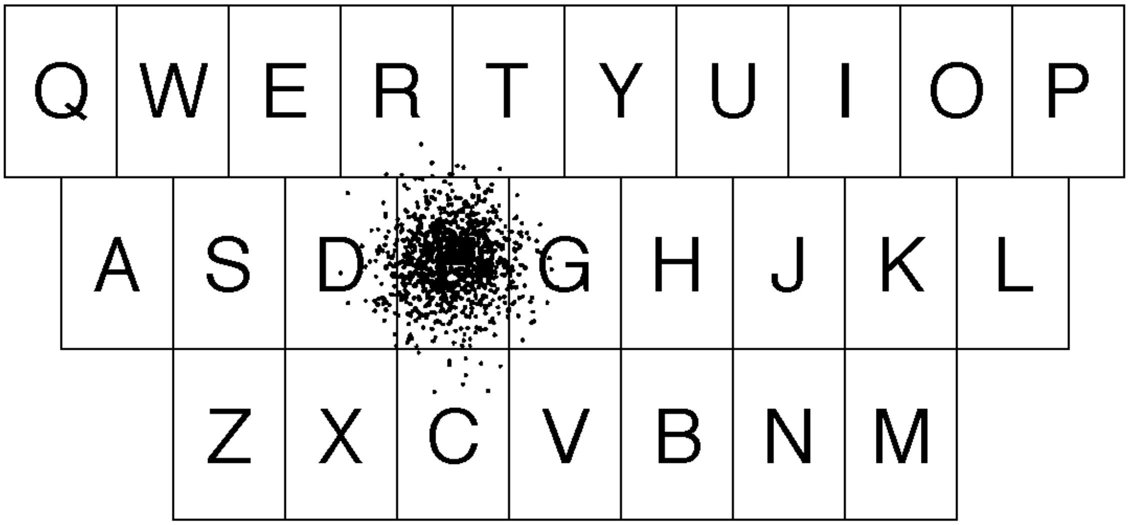


Figure 8.1: Simulated keyboard layout with 1000 randomly sampled taps on the “F” key

relative and absolute precision errors.

I deliberately chose the 2D Gaussian for modeling spatial touch, for multiple reasons: First, a simple, principled model was desired (in the spirit of Occam’s razor) that still captured the spatial variance needed to test LM contribution to decoding noisy touchscreen input. Second, 2D Gaussian is at the core of spatial models used in commercial level keyboards such as the Android Open Source Project (AOSP) keyboard. Third, while more complex models of touch have been explored in the literature (Holz and Baudisch [2011], Azenkot and Zhai [2012], and others highlight additional phenomena involved with finger touches, in particular spatial offsets that depend on finger position, visual cues, and hand posture), it is quite hard to put these models in practice because of issues such as sensor requirements. Fourth, Yin et al. [2013] showed that if a more complex hierarchical backoff model (with hand posture and key location dependencies) is used, the gain is large when considering individual letters only but small (diminishes) when connected with an LM. It is likely that further gains can still be made with improved spatial modeling, but a simple Gaussian model reflects the state of the art and is sufficient in testing LM contribution.

I made three further simplifying assumptions about the test data and the simulated user. First, since the simulated STK contains only the symbols a–z, the typing simulation only considers those letters. Other non-space symbols were stripped out before testing. Second, the simulated user always taps the *space bar* correctly, i.e. always taps it when intended and never taps it when not intended. Third, the input process is case-insensitive.

I chose to focus on 2D spatial noise exclusively as a source of error in the tapping signal.

There are other potential error sources, many of which fall under the category of *cognitive error*. An example of a cognitive error would be the user misspelling a word because they do not know the correct spelling. In such a scenario, an incorrect string is created, but it is not due to an error of typing; the user intended the string, whether or not it was correct. For instance, typing `forword` instead of `forward` would constitute a cognitive error, while `forqard` would likely be a spatial error. Other cognitive errors include selecting the wrong word from the suggestion list, or failing to tap the correct word when it appears in the selection list. Because modeling cognitive error is subtle and because we wish to be able to interpret the results more straightforwardly, it is excluded from these experiments. Similarly, errors involving inserted and deleted characters are excluded. Overall, the goal was to introduce a predominant category of error in sufficient and representative quantity to test the LM personalization effect in correcting that error category. Other error categories may follow the same pattern, but are reserved for future work.

8.4.2 A Simple Model Decoder

Once the simulator had generated a sequence of taps, I next applied a simple model decoder to represent the core function of an STK. The general idea of combining a spatial model of touchscreen behavior with language modeling for text entry has previously appeared in the literature. For example, Weir et al. [2014] applied user-customized spatial models and used touch pressure as a signal to increase the spatial model weight relative their LM weight. This work differs from Weir et al. in its use of longitudinal user data, the large scale of the simulations, and the fact that it personalizes the LMs rather than the spatial model. For the purpose of systematically evaluating the impact of LM adaptation in mobile text input applications, this decoder was not designed to be optimal, complete, or computationally efficient. Instead it was meant to be as simple as possible but still able to take advantage of a language model and effectively correct imprecise spatial input.

Suppose we have a tap sequence $T = t_1, t_2, ..., t_n$, where the t_i values are the coordinates of each tap, and a potential word w=$l_1, l_2, ..., l_m$, where the l_i values are the letters that make up that word. Note that any predicted word must have at least n letters, so we require that $m \geq n$. We define the spatial score $S(w|T)$ as

$$S(w|T) = \sum_{i=1}^{n}((t_{ix} - l_{ix})^2 + (t_{iy} - l_{iy})^2), \tag{8.1}$$

where t_{ix} and t_{iy} are the x and y coordinates of the tap t_i, and l_{ix} and l_{iy} represent the x and y coordinates of the key center of the letter l_i.

We further let $L_{LM}(w|c)$ be the negative logarithm of the LM probability of the word w given context c. The combined score $B(w|c,T)$ is equal to

$$B(w|c,T) = L_{LM}(w|c) - \gamma S(w|T), \tag{8.2}$$

where γ is the spatial model weight. The best-scoring word w_{best} under the combined model is therefore

$$w_{best} = \arg\max_{w} B(w|c,T). \tag{8.3}$$

Experiments on the development set showed that the optimal spatial model weight γ for the task is 0.00022[3].

Algorithm 1 Procedure for generating possible words matching a tap sequence

```
procedure GetPotentialWords($T, d$)
    input: list $T$ of $n$ taps; distance threshold $d$
    $wordlist \leftarrow []$
    for $t_i$ in $T$ do
        generate set $S_i$ of all letters on keyboard with centers within distance $d$ of $t_i$
    for $w_p$ in lexicon $L$ with $\geq n$ letters do
        for letter $l_i$ in $w_p$, $1 \leq i \leq n$ do
            if $l_i \in S_i$ then
                continue
            else
                break (word contains too-distant letter)
        append $w_p$ to $wordlist$
    return $wordlist$
```

An important parameter in this spatial decoding technique is a pruning criterion, used to decrease the size of the search space by limiting potential keys for each tap. Any key with a center within a specific radius from the tap may be a potential letter. Given the geometry of the simulated keyboard, this allows for between one and five potential keys for each tap. The search algorithm then recursively traverses the tree of possible words and word prefixes given these potential keys, and filters out all prefixes that do not match a known word in the LM lexicon. This is demonstrated in the procedure GetPotentialWords in Algorithm 1. We found using the development set that the pruning radius of 7.99 mm resulted in fast simulation without degrading model performance.

Figure 8.2 and Table 8.1 illustrate a simple example of the combined decoder at work. The spatial decoder has no notion of word probabilities in language, so when alone it can only function as a naive keyboard. It favors the spatially-closest words. The LM has no spatial data (other than

[3] The small value of γ is a consequence of the spatial model outputting much larger absolute values than the LM, ultimately because the spatial model uses pixel widths as a distance unit.

Algorithm 2 Procedures for typing simulation and evaluation

```
procedure CalculateAccuracy($w, n, c, d$)
    input: target word $w$ of length $n$, previous context $c$
    input: pruning threshold $d$, autocorrect threshold $\phi$
    $T \leftarrow []$
    for letter $l_i$ in $w$ do
        $t_i \leftarrow$ sample from 2D Gaussian centered at $l_i$
        add $t_i$ to $T$
    $wordlist \leftarrow$ GetPotentialWords($T, d$)
    $lit \leftarrow$ string of actual keys tapped by taps in $T$
    if $wordlist$ is not empty then
        for $w_p$ in wordlist do
            calculate score $B(w_p|c, T)$ given $T$ and $c$
        if highest score $> \phi$ then
            $w_{best} \leftarrow w_p$ with highest model score
        else (best word not good enough; use lit. string)
            $w_{best} \leftarrow lit$
    else (no suggestions; use literal string)
        $w_{best} \leftarrow lit$
    if $w_{best} = w$ then
        return $success$
    else
        return $failure$
procedure CalculateEfficiency($w, n, c$)
    input: target word $w$ of length $n$, previous context $c$
    $T \leftarrow []$
    for $i$ in 0 ... n do (iterate over number of taps)
        if $i > 0$ then
            $l_i \leftarrow i^{th}$ letter of $w$
            $t_i \leftarrow$ key center of $l_i$
            add $t_i$ to $T$
        $wordlist \leftarrow$ GetPotentialWords($T, 0$)
        $lit \leftarrow$ string of actual keys tapped by taps in $T$
        if $wordlist$ is not empty then
            for $w_p$ in wordlist do
                calculate score $B(w_p|c, T)$ given $T$ and $c$
            $w_{bestset} \leftarrow$ three $w_p$ with highest model score
        else (no suggestions; use literal string)
            $w_{bestset} \leftarrow lit$
        if $lit \notin w_{bestset}$ then
            if $w_{bestset}$ contains fewer than three words then
                append $lit$ to end of $w_{bestset}$
            else
                replace $3^{rd}$ word in $w_{bestset}$ with $lit$
        if $w \in w_{bestset}$ then
            return $i + 1$ (count of taps needed to type $w$)
```

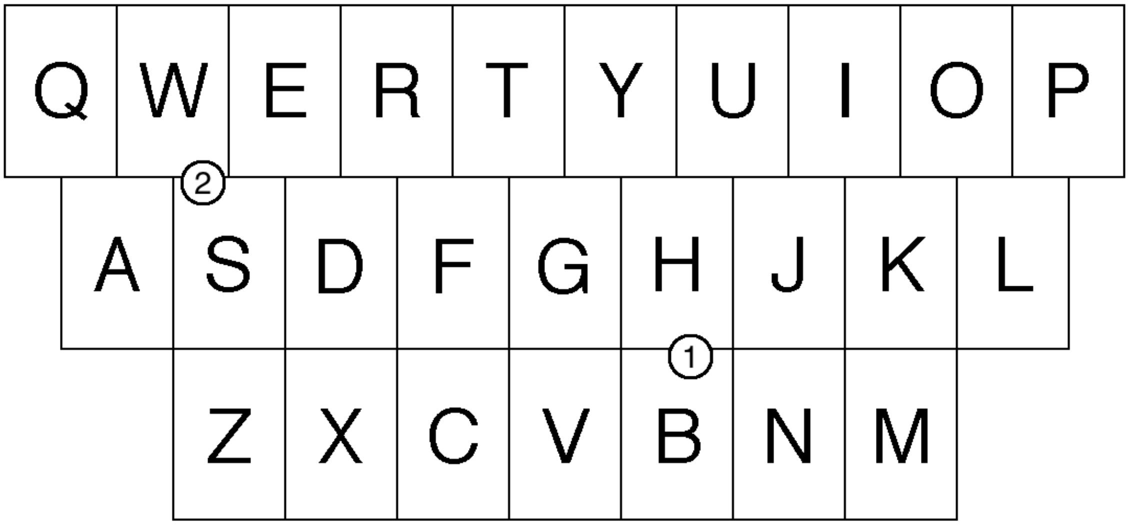


Figure 8.2: Two example taps on the simulated keyboard.

	Decoding Method		
Rank	**Spatial Model**	**Language Model**	**Combined**
1	ba	be	be
2	ha	he	best
3	be	had	bad
4	he	best	he
5	bad	bad	had

Table 8.1: Top potential words given the taps in Figure 8.2, under the spatial model only, language model only, and combined decoder.

what set of keys is nearby each tap), so it simply gives the most probable words possible given the pruning radius. The combined model synthesizes these two sources of information to generate a better word list. Note that all three models, *even the spatial model*, are constrained by the lexicon; the word '`bs`' is absent from Table 8.1 because it is not in the lexicon, despite the fact that it is the typed literal string. The top word in the spatial model, '`ba`,' is the word in the lexicon (albeit rare) with the highest spatial score. (As mentioned elsewhere, the decoding procedure has a method for including the actually-typed sequence in the suggested words list, but this is a separate process.)

8.5 Evaluation Metrics

Language models are often evaluated using the *per-word perplexity* metric across a test set described in Section 3.1.3. It is an information-theoretic measure of how well a statistical model predicts a sample. In theory, a language model better suited to the test data will result in a lower perplexity value. Many LM adaptation techniques are evaluated in this way, and report a decrease in observed perplexity as evidence of an improved model. In practice, however, and as discussed previously, even significant improvements in perplexity do not necessarily correlate with commensurate improvements in extrinsic objectives, particularly in speech recognition research, where word error rate (WER) is the dominant evaluation metric [Chen et al., 1998, Goodman, 2001]. Further, perplexity is often difficult to compare across tests, because it is lexicon-dependent, i.e. the models involved must use the same word list. This is not easy to accomplish in an adaptation/personalization setting, where adding new words to a model is of central importance.

One response to the shortcomings of perplexity and other LM-intrinsic metrics is to use LM-extrinsic evaluation, a research philosophy outlined in Jones [1994]. In this case, the extrinsic evaluation metrics are keystroke savings and WER. Keystroke savings (recalling Equation 3.9) is the de-facto evaluation standard for text entry efficiency measurement [MacKenzie and Soukoreff, 2002a]. WER is a standard way of measuring the accuracy of a word-based decoding task, and is widely used in speech recognition.

Importantly, an LM's extrinsic power in error correction (i.e. word-level accuracy) can *only* be measured in the presence of noise and a complete closed-loop system that includes a decoder and erroneous input. The development of such a system, and the ability it affords to measure error correction, are key contributions of this research. In ASR it was only when a reference recognizer was applied that language modeling studies could move from intrinsic measures such as perplexity to extrinsic measures such as WER [Chen et al., 1998]. Since "fat" finger imprecision is the main source of noise in finger-operated touch keyboard [Azenkot and Zhai, 2012], the simple model decoder as described earlier together with the spatial noise model and the real world personal history data set of Enron emails enabled us to measure the extrinsic power of the LM and its adaptation.

8.5.1 Simulating Word Correction

When simulating word *correction*, we proceed one word at a time. The simulator samples one tap (with spatial noise) for *every letter* in the target word, then returns the single most probable word from the decoder. If this word has a probability above a certain *autocorrect threshold*, the word is typed, followed by a space. If not, the uncorrected *literal string* is typed (the literal string is the sequence of visual keys tapped; in the case of Figure 8.2, the literal string is '`bs`'). A threshold value 0.7 was used for all accuracy experiments, for reasons explained in Section 8.7 below. This is demonstrated in the procedure CalculateAccuracy in Algorithm 2. The evaluation metric for word correction is word error rate. WER is calculated by dividing the number of incorrectly-decoded words by the total number of words.

8.5.2 Simulating Word Prediction

When simulating word *prediction*, we also proceed one word at a time, but there are two important differences. First, we provide the simulated user with a list of three word suggestions after each tap, potentially saving keystrokes. Second, in order to isolate the prediction effect, we exclude spatial noise. The following is demonstrated in the procedure CalculateEfficiency in Algorithm 2. For each word, we incrementally simulate perfect input for each letter, starting with zero taps. At each point, the LM (no spatial model or decoding) generates the three most probable words given the letters so far (including the case where there are not yet letters typed). If the literal string is not among these three, the third-most-probable word is removed and replaced with the literal string. Otherwise, the list remains unchanged. We assume that the simulated user will notice immediately if the target word is among these top three. If so, we count one keystroke for selecting the word. If not, we make another simulated tap and generate a new suggestion list. (Note that, in word-initial position, no taps have occurred, and all words in the LM are possible.) In the case where all n taps are needed to type an n-letter word, the simulated user makes one last check for the target word in the top-three word prediction list. If the target is there, it is selected. If not, a space key is typed. When the space key is tapped, the single most probable word in the prediction list is entered. This is the reason the literal string is always included in the list. Without the literal string the user has no way of entering words that are not in the LM lexicon.

The evaluation metric for word prediction is keystroke savings, as defined in Equation 3.9. Because we assume no spatial noise when measuring word prediction, accuracy is always 100%. Note that, when simulating either correction or prediction, each word requires a minimum of one keystroke, since either space or select-suggestion must be tapped to move to the next word.

Since the LM is a bigram model, its probabilities are conditioned on the previous word in each context. At the beginning of each line (every line in the test set is a sentence), the model is primed with the start-sentence symbol `<S>`. After each word is entered, that word becomes the previous word, *even if the typed word is incorrect and does not match the true string.* The latter scenario matches real-world use, since the typed word is all any typing system has access to, even if it is not correct. In the case of words not in the LM lexicon, the previous word is set to the unknown symbol `<UNK>`.

It is notable that both word correction and word prediction in this modality are ways of framing the revision of past text in an incremental fashion. For instance, if a given correction is computed by the algorithm and inserted into the literal string as the space key is typed, this is still an incremental process from the standpoint of the user on a word level, even though the hypothesis of the current best word can change as new taps are observed. Similarly, a word completion suggestion might include a word where corrections have been made in the already-tapped letters, but the input here is essentially incremental. This is a difference from non-incremental algorithms like traditional speech recognition, where a hypothesis can change at the scale of a sentence or larger.

8.6 Language Model Personalization

I implemented two basic types of cache-based language models: Uniform and exponentially-decaying. The uniform cache considered a sliding window of words stretching back as far as available in the user's test data, and consisted of an unsmoothed unigram LM from the words in that window. It was continuously updated with each individual user's writing history as simulation progressed. Linear interpolation was used to combine the background model with this in-domain cache model[4]. Linear interpolation is a simple technique for LM adaptation. Equation 8.4 (adapted from Bellegarda [2004]) illustrates the basic interpolation method, where $P(w|c)$ is the conditional probability of word w given context c, P_A is the adaptation model, P_B is the background model, and λ represents the relative weight of the background model.

$$P(w|c) = (1 - \lambda)P_A(w) + \lambda P_B(w|c) \tag{8.4}$$

I found through experiments on held-out data that the optimal window size is infinite, i.e. all words in the user's history should be kept in the cache. Further, the optimal λ value was found to be 0.8.

The exponentially-decaying cache was derived from Clarkson and Robinson [1997], which first described the technique. It used an unsmoothed unigram model like the uniform cache, but instead of counting words directly the weights are modified according to how far back they are in the cache. This captures the recency effect. Equation 8.5 (adapted from Clarkson and Robinson [1997]) illustrates the basic method, where P_{cache} is the conditional probability of word w_i given

[4] Note that since a LM requires large quantities of text, using the cache model alone would not be effective; even users with large text histories do not have enough text to build a useful LM without sparsity issues.

the cache w_1 through w_{i-1}, and I is a binary function such that $I(A) = 1$ if A is true, and 0 otherwise. The decay rate α describes how quickly word weights decay in the cache, and β is a normalizing constant:

$$P_{cache}(w_i|w_1, w_2, ..., w_{i-1}) = \beta \sum_{j=1}^{i-1} I(w_i = w_j)e^{-\alpha(i-j)} \tag{8.5}$$

A grid search on held-out data was used to find that the optimal decay rate α is 0.0003, which is a slow rate of decay but consistent with the values determined in Clarkson and Robinson [1997]. An α value of 0.0003 means that it takes roughly 2300 words of history before the adjusted weight decreases to half of its initial value. Since most users in the test set generate between 1000 and 2000 words per month, this decay rate can be said to take months to take effect. Note, however, that this formulation is based on word positions rather than elapsed time. A word located at position $i - 100$ may have been typed at any point in the past, depending on the user. As with the uniform cache method, the cache model is combined with the background model using linear interpolation. The optimal λ value was again found to be 0.8.

For both cache methods, I primed the cache with the first 30 days of email from each of the 45 individuals in the test set. Messages sent during this period were used only to fill the personalization cache, and not simulated for evaluation purposes. The choice of 30 days was based on an empirical study on the development set, which showed that 30 days generally provides a reasonable initial cache size. The mean size of this priming cache in the test set was 1429 words. Both cache models were updated from the 31st day onward till the end of each individual's text available in Enron.

We define the model lexicon at any given time to be all words in *either* the background model unigram list or the cache. When a word was absent from one of these two models, it received a probability of zero in that model before the interpolation defined by Equation 8.4 took place. Though the background model did contain an `<UNK>` symbol, we normalized it out when calculating word probabilities[5]. Also, no attempt was made to estimate probabilities of words not in either model, i.e. entirely absent from the model lexicon (though I did preserve the literal string, which can sometimes be absent from both models).

8.7 Results

By replicating and simulating users' natural finger touch input behavior, multi-user and longitudinal writing records, and the primary functions keyboard decoding, we are able to reveal what

[5] We still use `<UNK>` for ngram contexts; see earlier paragraph on LM conditioning.

Model	**WER %**	**OOV %**
No Language Model	38.4	n/a
Background LM Only	5.7	1.6
Uniform Cache Adaptation	4.6	0.9
Decaying Cache Adaptation	4.6	0.9

Table 8.2: Word error rates for various model setups, averaged across all simulated Enron users.

simple human subjects lab experiments could not do—quantitatively benchmarking the effects of language modeling with and without personalization at scale.

8.7.1 Word Correction

Table 8.2 shows WER results for the accuracy experiments. The first row represents the value of a naive keyboard without any language modeling and takes the literal string only, which is equivalent to decoding touch input according to its closest key centers. The word error rate of such a naive keyboard is 38.4%.

With a background LM combined with the simple decoder, both as described earlier, the error rate is drastically reduced to 5.7%. This is at the 0.7 autocorrect threshold operating point, as explained below. LM personalization further improves the error rate to 4.6%, a relative decrease of 19.3%. The exponentially-decaying cache method did not outperform the uniform cache method.

The OOV rate is calculated on a per-word basis as the human simulation progresses through the test set. In the case of cache-based models, a given word is considered OOV if it did not exist in the background model or the cache (though since all OOVs are added to the cache immediately, no word can be an OOV twice for the same user). The cache is emptied before moving to a new user in the test set.

8.7.2 False Corrections and the Autocorrect Threshold

In the word correction experiments, I considered one type of failure to be worse than the others. If the literal string is correct, but the autocorrect mechanism changes it to a word that is incorrect, we call this a *false correction*. The rationale for setting apart these failures is this: Suppose a user carefully types a word such that all taps in that word fall inside their respective target key boundaries. The user likely has a strong expectation that the STK will output the literal string, and autocorrecting to another string is particularly undesirable. We therefore may wish to decrease the rate of false corrections, even at the cost of a slight increase in overall word error rate.

It turns out that this trade-off can be managed by introducing an *autocorrect threshold* parameter to the model. It operates as follows: If the top-scoring word is not equal to the literal string,

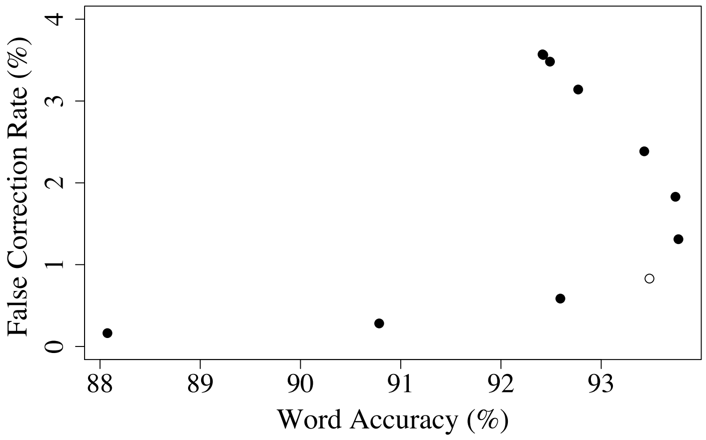


Figure 8.3: Trade-off between overall word accuracy and false correction rate, managed by varying the autocorrect threshold from 0.0 (upper right) to 0.95 (lower left). The rightmost dot is optimal from an overall word accuracy standpoint, but I chose the white dot as an operating point to bring the false correction rate below 1%.

and its probability[6] is below the autocorrect threshold, we prevent the decoder from performing a correction and simply *use the literal string.*

Figure 8.3 illustrates the trade-off, calculated on a subset of the development set. It turns out that allowing the decoder to *always* perform correction, i.e. an autocorrect threshold of 0.0, results in a high rate of false corrections, more than 3.5%. This amounts to nearly half of all failures for this test. The optimal autocorrect threshold value for maximizing *overall* accuracy is roughly 0.6, which corresponds to the typical accuracy of any given literal string[7]. I selected 0.7 as the operating point for the autocorrect threshold in all accuracy experiments. This choice caused a slight decrease in overall accuracy, but it brought the false correction rate below 1%. On the test data, the false correction rate was 0.70% for the baseline model and 0.39% for the uniform cache mixture model, representing an even larger relative WER improvement than that observed for overall failures.

8.7.3 Precision and Recall

Another way to think about the performance of the word completion model is as a trade-off between precision and recall. In this case, precision is defined as the proportion of autocorrected words that were changed to the true word. Recall is defined as the proportion of words needing correction (i.e. with incorrect literal strings) that were autocorrected to the true word. Figure 8.4 shows

[6] I converted the log-space model scores to probabilities and normalized to make this calculation.

[7] This is just the accuracy of a word in the no-model scenario, or $1.0 - 0.384$.

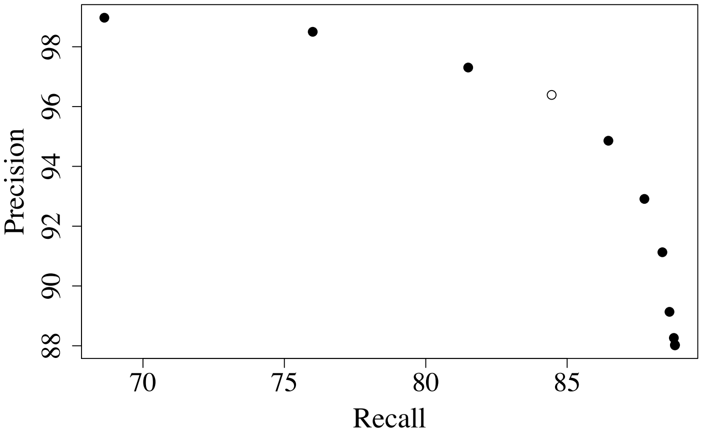


Figure 8.4: Precision vs. recall in word correction, generated by varying autocorrect threshold from 0.0 (lower right) to 0.95 (upper left). Operating point is white dot, equivalent to Figure 8.3.

Model	**Keystroke Savings %**
Background LM Only	42.1
Uniform Cache Adaptation	45.7
Exponential Cache Adaptation	45.8

Table 8.3: Keystroke savings for various model setups, averaged across all simulated Enron users. See Equation 3.9 for definition of keystroke savings.

the precision vs. recall curve. Note that, under the conditions of the model, 100% recall is not possible. This is due to the fact that sometimes a literal string (with an error) is the only word in the predicted word list. This can happen when the decoder fails to generate any candidates because the tap sequence is too far from any known word. In this situation, correction cannot occur in the model (not even a false correction), because no candidates exist.

8.7.4 Word Prediction

Table 8.3 shows keystroke savings results for the efficiency experiments. Without any language modeling (not even a lexicon), keystroke savings would not be possible. The background LM increased (potential) keystroke savings from 0% to 42.1%. When using a cache model for LM personalization, this number was further increased to 45.7%, a relative gain of 8.6%. As before, the uniform cache method performed comparably to the exponentially-decaying cache method.

8.7.5 Results by User

The results in Tables 8.2 and 8.3 represent averages over all users in the test set, but LM personalization had different effects on different users. Of the 45 users in the test set, 42 exhibited improved typing accuracy under the uniform cache adaptation method. This relative improvement in WER ranged widely, from 2.9% to 30.9%. Of the three users whose accuracy got worse, one had accuracy decrease from 91.4% to 89.3%, one decreased from 91.5% to 91.1%, and one decreased from 92.9% to 92.7%. All three with decreased performance had fewer than 1500 words in their priming emails, which suggests that it may be prudent to delay the application of cache-based adaptation until a sufficient amount of user text has been collected.

8.8 Conclusion and Discussion

Through research methodology innovation and evaluation system development, I have investigated the effects of language models, with and without personalization, for improved accuracy and efficiency of contemporary touch screen keyboards operated with sloppy finger touches. In benchmark tests, I demonstrated that it was possible to decrease word error rate due to imprecise finger touch tapping from 38.4% to 5.7% using a simple decoder and a background language model. LM personalization further improved this result to 4.6%. Language modeling also improved prediction efficiency to 42.1% (without personalization) and 45.7% (with personalization). The OOV rate of the test set improved from 1.6% to 0.9% using a personalized model. I also showed how raising the autocorrect threshold could lower the false correction rate from 3.5% to 0.7% without significantly affecting overall model performance.

These results were only observable and tuneable because of the novel evaluation methodology. First, web-scale data training and entropy pruning was leveraged to produce a state-of-the-art background LM. Second, the adoption of the Enron Corpus with long-term individual history enabled the study of personalization effects with real-world language use fidelity. Third the simple-yet-effective model decoder enabled measurement of extrinsic power in an integrated closed loop evaluation system. Fourth, I introduced human-like noisy spatial input in the evaluation based on empirical findings previously reported in the literature. Taken together, these methodological innovations allowed for the efficient performance of computational experiments over two million words of text, which otherwise would have taken months or years of calendar time to perform. Crucially, this combined closed-loop system also allowed for the reporting of *accuracy* on a large scale (whereas efficiency has been a typical metric in text entry research). I additionally found, somewhat counter-intuitively (but consistent with Tanaka-Ishii [2007]), that the exponential decay cache model did not outperform the simple uniform cache.

All of these methodological innovations have limitations, as is true of any research methodology. Although both the spatial input and the language content were based on human generated data from Azenkot and Zhai [2012] and Enron respectively, they were nonetheless simplified to a degree that efficient experiments could be run. Future work needs to replicate and expand these results.

Another opportunity for future work is to study how cognitive errors, particularly spelling errors, affect decoding performance. This would clarify whether the observed gains via LM personalization with spatial noise can be observed in a more general error framework. Along this same lines, although adding OOV words to the cache was found to be always desirable in this system, cognitive error would likely introduce the problem of learning misspellings, which may require separate techniques to manage. Other ideas, such as varying the amount of spatial noise, expanding to input methods other than QWERTY, and applying these techniques to gesture typing, are all interesting avenues for future work.

Ultimately this chapter is a bit orthogonal to the work in the previous chapters, but the presence of the cache-based personalization model, in addition to the use of word correction and word prediction, situate this work firmly within the larger examination of incremental processing for text entry. The very fact that the LM is being updated incrementally suggests ideas for future work. For instance, if a word is deleted should it be removed from the cache model? One could also study the impact word-level incrementality itself, allowing for models that can revise words farther and farther back in the literal string and examining whether such models perform better on various objective and subjective metrics.

It would also be useful to study LM personalization as described in this chapter in the AAC settings described in previous chapters, such as RSVP Keyboard. LM personalization for users of assistive technology is actually particularly promising. This because many people using AAC devices not only use text as a primary form of communication but have access, through their technology, to historical utterances they have made that could be used as input to a personalization model.

Chapter 9

Conclusion

This stated goal of this thesis was to explore incrementality as it relates to language technology for text entry, in particular in AAC. I first developed the full history fusion algorithm, an augmentation of the RSVP Keyboard text entry paradigm used in AAC. This algorithm collects all past user inputs and combines them in a principled way using the intuition that posterior symbol probabilities can be derived from full string probabilities. It also gives particular focus to the dynamic nature of the backspace key as a method for user-directed revision and error correction. Along with the mathematics of the algorithm, I developed a means by which it can be computed efficiency, which is itself nontrivial.

I then presented the results of two experiments that compared FHF to the standard setup for RSVP Keyboard. The first was a large-scale simulation that used actual EEG-derived user data as the basis for an analysis of typing performance over thousands of words of text under a wide range of system parameters. This simulation demonstrated not only that FHF is superior to RSVP Keyboard in terms of typing speed, but also that the default settings of RSVP Keyboard itself can be optimized. Further, it showed that FHF avoids certain traps that can occur in RSVP keyboard, where a user can get stuck or experience runway typing that they do not intend. This experiment also showed the validity of FHF across a range of user signal strengths, demonstrating a consistent improvement in typing speed of 20-30% across six simulated users. I also explored autodelete and autorevision, useful phenomena that can occur in FHF where its principled calculation of posterior probabilities can result in the automatic repair of a word and correction into a new word.

The second experiment examined the usefulness of FHF when used by human subjects. Through this button-press typing study, I was able to demonstrate that the improved performance of FHF observed in simulation translated to the real world, giving an average typing speed improvement in the 15-20% range. In addition, I used a questionnaire to answer one of the open questions about FHF: Does it involve more cognitive load, mental effort, or frustration to use? The questionnaire answered all of these questions in the negative; no significant difference in subjective measure of effort was reported by the users. I was also able to confirm the hypothesis underlying the *mastery task*, a series of word groupings that clustered words based on their relative "difficulty" in the context of a language model. Users of the button press modality consistently showed a positive correlation between the mastery level being typed and the time taken to type those word groups.

I then described an entirely different algorithm for the computation of backspace probabilities, this time re-framing the question as a sequence modeling problem. For this algorithm, I developed the Errorful Typist Model, a method for generating augmented text corpora as if they had been typed by a person making and correcting typing errors. By including backspace as a symbol in the text stream to be predicted, I was able to use a neural network-based architecture (LSTM) to create a model capable of predicting both the characters in the text and backspace itself in a dynamic way. I then tested this algorithm under a variety of conditions, showing that not only did the neural method outperform a traditional ngram-based baseline model, but that it appears to be learning certain semantics of the backspace key, such as the fact that a just-deleted character should now be less probable.

Finally, I presented the results of a large scale simulation studying the quantitative effects of language model personalization on text entry for touchscreen smartphones. By creating a closed-loop system for extrinsic evaluation of text entry performance under various language models, I was able to demonstrate the precise gains afforded by the language model itself, a keystroke savings of 42.1% (a result known to exist but not presented quantitatively at scale anywhere in the literature). I also showed, empirically for the first time, the improvements to these gains that can be obtained by creating a custom uniform cache language model for each person known to have typed various sections of the test set, a relative improvement of 8.6%. I was able to exploit the power of extrinsic evaluation to reject the exponential cache adaptation method as one significantly better than the uniform method for this task, despite the fact that it showed better perplexity in the literature. Language model personalization is another way of thinking about text entry in terms of incrementality, because it is a cache-based model that changes as the user progresses through a text. Here, the incrementality is an attribute of the model itself, which incrementally changes throughout a typing session.

9.1 Impact and Future Work

In terms of practical applicability, I believe this work has great potential. The most singular use case is obviously RSVP Keyboard, the paradigm used as the basis for the FHF algorithm, under which I have repeatedly demonstrated empirical typing speed improvements. But I envision the backspace work of Chapter 7 and the personalization work of Chapter 8 to be of significant value to the academic community, as these chapters present previously unreported results and unstudied methods for text entry. Further, I think AAC is a particularly appealing domain for the work of this thesis, as it not only represents an environment where even small gains in typing performance as crucial, but also exhibits incrementality in the strictest possible way by often requiring single-character inputs.

I think a necessary next step would be to perform a test of the FHF algorithm within RSVP Keyboard in which the input modality is EEG rather than button press. My experiments have provided hints that the results of such an experiment would be positive for FHF, but it remains an empirical question. Another thing that would be useful is a further analysis of what the neural models in Chapter 7 are actually learning in terms of the semantics of backspace. The results of Chapter 8 could also be tested outside of simulation and with real human subjects, which could show whether keystroke savings is actually a useful metric for this domain.

In terms of error correction, revision, and backspace, I made a research decision early in this work that I was deliberately not studying certain classes of errors like misspellings and long-term revisions. However, these represent appealing starting points for future work. In particular, the concept that certain errors will be either unnoticed or ignored by the user—the latter phenomenon having been observed particularly among AAC users—is an interesting twist to some of the assumptions of RSVP Keyboard and FHF.

Overall, the algorithms I have developed for this thesis, and the results showing their utility, represent a significant addition to the body of text entry literature. In particular, my framing of text entry in terms of incrementality has allowed me not only to explore new avenues for algorithmic improvement, but also to develop techniques related to error correction and revision having to do with the backspace key that have not been examined before.

Appendix A

Enron Preprocessing Steps

The purpose of preprocessing of the Enron Data set was to remove as much non-user-generated text as possible, and group together messages associated with the people who typed them. This was accomplished using the following steps:

1. Extracted only messages that appear in the `sent` or `sent_items` directories in the Enron Corpus directory. Visual inspection shows that these are emails generated by the senders themselves, and contain few duplicates.

2. Extracted useful information from email headers. This included message ID, sender, date, and subject line.

3. Mapped email addresses and sender names to a canonical single name for each user. This was necessary because the sender name implied by the directory structure of the corpus was not always reliable; some users had multiple email addresses and some emails were sent by Person A using the email client of Person B.

4. Removed all forwarded and replied-to messages from the bottom of emails. The corpus is not consistently formatted; I counted 21 distinct ways in which a message can end and supplementary text can begin.

5. Fixed poorly-formatted line breaks. This occurred in some older emails, where inconsistent text encoding caused junk characters to appear at the end of every line.

6. Removed duplicate emails. Many emails appeared, copied verbatim, in more than one place in the corpus. I removed all but one copy of each of these.

7. Removed emails longer than 1000 words. This was a heuristic measure to exclude non-user-generated text. Many emails in the corpus are news or magazine articles copy-pasted into the message body, and these are typically significantly longer than normal emails.

8. Converted non-ASCII characters (extremely rare) to ASCII using the `iconv` utility Drepper [2001].

9. Collapsed extra whitespace into single spaces.

10. Deleted email signatures. I searched for medium-length strings of text appearing at the ends of more than one email, then hand-selected a list of probable signature blocks from these. I then removed all of them from their respective messages.

11. Removed lines containing more than half OOV words. This was a way to exclude computer-generated text, numbers, and other non-useful text.

12. Removed URLs and other link-like text.

13. Removed all copies of emails containing duplicate substrings longer than 15 words. This was important because such text was very rarely user-generated. In addition, because of the way such messages could interleave and overlap in subtle ways, it was better not to remove just the duplicated strings but the entire emails (this amounted to only a small proportion of the total emails).

14. Performed simple rule-based sentence segmentation. This allowed the language model, which was trained on sentences, to provide more useful predictions at the start of sentences.

Bibliography

S. Abney. Rapid incremental parsing with repair. In *Proceedings of the 6th New OED Conference: Electronic Text Research*, pages 1–9, 1990.

S. P. Abney and M. Johnson. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, 20(3):233–250, 1991.

J. Adhikary, R. Watling, C. Fletcher, A. Stanage, and K. Vertanen. Investigating speech recognition for improving predictive AAC. In *Proceedings of the Eighth Workshop on Speech and Language Processing for Assistive Technologies*, pages 37–43, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-1706. URL `https://www.aclweb.org/anthology/W19-1706`.

S. Azenkot and S. Zhai. Touch behavior with different postures on soft smartphone keyboards. In *Proceedings of the 14th international conference on Human-computer interaction with mobile devices and services*, pages 251–260. ACM, 2012.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1409.0473`.

T. Baumann, M. Atterer, and D. Schlangen. Assessing and improving the performance of speech recognition for incremental systems. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 380–388. Association for Computational Linguistics, 2009.

T. Baumann, C. Kennington, J. Hough, and D. Schlangen. *Recognising Conversational Speech: What an Incremental ASR Should Do for a Dialogue System and How to Get There*, pages 421–432. Springer Singapore, Singapore, 2017. ISBN 978-981-10-2585-3. doi: 10.1007/978-981-10-2585-3_35. URL `https://doi.org/10.1007/978-981-10-2585-3_35`.

R. Beckley and B. Roark. Asynchronous fixed-grid scanning with dynamic codes. In *Proceedings of the Second Workshop on Speech and Language Processing for Assistive Technologies*, pages 43–51. Association for Computational Linguistics, 2011.

Y. Belinkov and Y. Bisk. Synthetic and natural noise both break neural machine translation. *arXiv preprint arXiv:1711.02173*, 2017.

T. C. Bell, J. G. Cleary, and I. H. Witten. *Text compression.* Prentice-Hall, Inc., 1990.

J. R. Bellegarda. Statistical language model adaptation: review and perspectives. *Speech Communication*, 42:93–108, 2004.

N. Beuck, A. Köhn, and W. Menzel. Decision strategies in incremental pos tagging. In *Proceedings of the 18th Nordic Conference of Computational Linguistics*, pages 26–33, 2011.

X. Bi, T. Ouyang, and S. Zhai. Both complete and correct? multi-objective optimization of touchscreen keyboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14. ACM, 2014a.

X. Bi, T. Ouyang, and S. Zhai. Both complete and correct?: multi-objective optimization of touchscreen keyboard. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2297–2306. ACM, 2014b.

B. Bohnet and J. Nivre. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1455–1465. Association for Computational Linguistics, 2012.

B. Bohnet, R. McDonald, G. Simões, D. Andor, E. Pitler, and J. Maynez. Morphosyntactic tagging with a meta-BiLSTM model over context sensitive token encodings. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2642–2652, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1246. URL `https://www.aclweb.org/anthology/P18-1246`.

B. Carpenter. Scaling high-order character language models to gigabytes. In *Proceedings of the ACL Workshop on Software*, pages 86–99, 2005a.

B. Carpenter. Scaling high-order character language models to gigabytes. In *Proceedings of the Workshop on Software*, pages 86–99. Association for Computational Linguistics, 2005b.

E. Charniak, D. Blaheta, N. Ge, K. Hall, J. Hale, and M. Johnson. BLLIP 1987-89 Wall Street Journal Corpus Release 1, 2000.

C. Chelba, T. Brants, W. Neveitt, and P. Xu. Study on Interaction between Entropy Pruning and Kneser-Ney Smoothing. In *Proc. Interspeech*, pages 2242–2245, Makuhari, Japan, 2010. ISCA.

S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394, 1999.

S. F. Chen, D. Beeferman, and R. Rosenfield. Evaluation metrics for language models. Technical report, Carnegie Mellon University, 1998.

L. Chiticariu, R. Krishnamurthy, Y. Li, F. Reiss, and S. Vaithyanathan. Domain adaptation of rule-based annotators for named-entity recognition tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '10, pages 1002–1012. Association for Computational Linguistics, 2010.

P. R. Clarkson and A. J. Robinson. Language model adaptation using mixtures and an exponentially decaying cache. In *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, volume 2, pages 799–802. IEEE, 1997.

J. Cocke and J. T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

M. Collins and B. Roark. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 111. Association for Computational Linguistics, 2004.

W. Crochetiere, R. Foulds, and R. Sterne. Computer aided motor communication. In *Proceedings of the 1974 Conference on Engineering Devices in Rehabilitation*, pages 1–8, 1974.

R. DeMori and M. Federico. Language model adaptation. In *Computational models of speech pattern processing*, pages 280–303. Springer, 1999.

V. Dhakal, A. M. Feit, P. O. Kristensson, and A. Oulasvirta. Observations on typing from 136 million keystrokes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 646. ACM, 2018.

R. Dong, D. A. Smith, S. Dudy, and S. Bedrick. Noisy neural language modeling for typing prediction in bci communication. In *Proceedings of the Eighth Workshop on Speech and Language Processing for Assistive Technologies*, pages 44–51, 2019.

U. Drepper. iconv: GNU C Library, 2001.

J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

L. A. Farwell and E. Donchin. Talking off the top of your head: toward a mental prosthesis utilizing event-related brain potentials. *Electroencephalography and clinical Neurophysiology*, 70 (6):510–523, 1988.

D. Fernández-González and C. Gómez-Rodríguez. Left-to-right dependency parsing with pointer networks. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 710–716, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1076. URL `https://www.aclweb.org/anthology/N19-1076`.

F. Ferreira and B. Swets. How incremental is language production? evidence from the production of utterances requiring the computation of arithmetic sums. *Journal of Memory and Language*, 46(1):57–84, 2002.

L. Findlater, J. O. Wobbrock, and D. Wigdor. Typing on flat glass: examining ten-finger expert typing patterns on touch surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2453–2462. ACM, 2011.

R. Foulds, G. Baletsa, and W. Crochetiere. The effectiveness of language redundancy in non-verbal communication. In *Proceedings of the Conference on Devices and Systems for the Disabled*, pages 82–86, 1975.

A. Fowler, K. Partridge, C. Chelba, X. Bi, T. Ouyang, and S. Zhai. Effects of language modeling and its personalization on touchscreen typing performance. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 649–658. ACM, 2015.

J. H. Friedman. Regularized discriminant analysis. *Journal of the American statistical association*, 84(405):165–175, 1989.

Y. Goldberg. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, 10(1):1–309, 2017.

J. Goodman, G. Venolia, K. Steury, and C. Parker. Language modeling for soft keyboards. In *Proceedings of the International Conference on Intelligent User Interfaces*, IUI '02, pages 194–195, New York, NY, USA, 2002. ACM. ISBN 1-58113-459-2. doi: 10.1145/502716.502753. URL `http://doi.acm.org/10.1145/502716.502753`.

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

E. Grushka. Characterization of exponentially modified gaussian peaks in chromatography. *Analytical Chemistry*, 44(11):1733–1738, 1972.

K. Harbusch and M. Kühn. Towards an adaptive communication aid with text input from ambiguous keyboards. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics-Volume 2*, pages 207–210. Association for Computational Linguistics, 2003.

K. Harbusch, S. Hasan, H. Hoffmann, M. Kühn, and B. Schüler. Domain-specific disambiguation for typing with ambiguous keyboards. In *Proceedings of the 2003 EACL Workshop on Language Modeling for Text Entry Methods*, pages 67–74. Association for Computational Linguistics, 2003.

H. Hassan, K. Sima'an, and A. Way. A syntactified direct translation model with linear-time decoding. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 3*, pages 1182–1191. Association for Computational Linguistics, 2009.

J. Hatori, T. Matsuzaki, Y. Miyao, and J. Tsujii. Incremental joint pos tagging and dependency parsing in chinese. In *IJCNLP*, pages 1216–1224. Citeseer, 2011.

D. J. Higginbotham. Evaluation of keystroke savings across five assistive communication technologies. *Augmentative and Alternative Communication*, 8(4):258–272, 1992.

D. J. Higginbotham, G. W. Lesher, B. J. Moulton, and B. Roark. The application of natural language processing to augmentative and alternative communication. *Assistive Technology*, 24 (1):14–24, 2012.

G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

C. Holz and P. Baudisch. Understanding touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2501–2510. ACM, 2011.

D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

K. S. Jones. Towards better NLP system evaluation. In *Proceedings of the Workshop on Human Language Technology*, pages 102–107. Association for Computational Linguistics, 1994.

D. Jurafsky and J. H. Martin. *Speech & Language Processing.* Prentice Hall, second edition, 2008.

Y. Kamide, G. Altmann, and S. L. Haywood. The time-course of prediction in incremental sentence processing: Evidence from anticipatory eye movements. *Journal of Memory and Language*, 49 (1):133–156, 2003.

A. Kano, J. C. Read, A. Dix, and I. S. MacKenzie. Expect: an expanded error categorisation method for text input. In *Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI... but not as we know it-Volume 1*, pages 147–156. British Computer Society, 2007.

T. Kasami. An efficient recognition and syntaxanalysis algorithm for context-free languages. Technical report, AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA., 1965.

S. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 35 number 3 of *ICASSP '87*, pages 400–01. IEEE, 1987.

N. Klarlund and M. Riley. Word n-grams for cluster keyboards. In *Proceedings of the EACL Workshop on Language Modeling for Text Entry Methods*, TextEntry '03, pages 51–58, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1628195.1628202`.

B. Klimt and Y. Yang. Introducing the Enron corpus. In *Proceedings of the Conference on Email and Anti-Spam*, CEAS '04, 2004.

R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 1 of *ICASSP '95*, pages 181–184. IEEE, 1995.

D. E. Knuth. On the translation of languages from left to right. *Information and control*, 8(6): 607–639, 1965.

H. H. Koester and S. Levine. Effect of a word prediction feature on user performance. *Augmentative and alternative communication*, 12(3):155–168, 1996.

P. A. Kolers and M. T. Katzman. Naming sequentially presented letters and words. *Language and Speech*, 9(2):84–95, 1966. doi: 10.1177/002383096600900202. URL `https://doi.org/10.1177/002383096600900202`. PMID: 5944120.

P. O. Kristensson and K. Vertanen. Asynchronous multimodal text entry using speech and gesture keyboards. In *Proc. Interspeech*, pages 581–584, 2011.

P. O. Kristensson and K. Vertanen. Performance comparisons of phrase sets and presentation styles for text entry evaluations. In *Proceedings of the International Conference on Intelligent User Interfaces*, IUI '12, pages 29–32. ACM, 2012.

P.-O. Kristensson and S. Zhai. Shark 2: a large vocabulary shorthand writing system for pen-based computers. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 43–52. ACM, 2004.

P. O. Kristensson and S. Zhai. Improving word-recognizers using an interactive lexicon with active and passive words. In *Proceedings of the International Conference on Intelligent User Interfaces*, IUI '08, pages 353–356. ACM, 2008.

A. Kübler, B. Kotchoubey, J. Kaiser, J. R. Wolpaw, and N. Birbaumer. Brain–computer communication: Unlocking the locked in. *Psychological bulletin*, 127(3):358, 2001.

R. Kuhn and R. De Mori. A cache-based natural language model for speech recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(6):570–583, 1990.

B. Lang. *Deterministic techniques for efficient non-deterministic parsers*. Springer, 1974.

G. Lesher and G. Rinkus. Leveraging word prediction to improve character prediction in a scanning configuration. In *Proceedings of the RESNA 2002 Annual Conference*, pages 90–92, 2002a.

G. Lesher, B. Moulton, and D. J. Higginbotham. Techniques for augmenting scanning communication. *Augmentative and Alternative Communication*, 14(2):81–101, 1998.

G. W. Lesher and G. J. Rinkus. Leveraging word prediction to improve character prediction in a scanning configuration. In *Proceedings of the RESNA 2002 Annual Conference*, Reno, 2002b.

J. Li and G. Hirst. Semantic knowledge in word completion. In *Proceedings of the 7th international ACM SIGACCESS conference on Computers and accessibility*, pages 121–128. ACM, 2005.

I. S. MacKenzie and R. W. Soukoreff. Text entry for mobile computing: Models and methods, theory and practice. *Human–Computer Interaction*, 17(2-3):147–198, 2002a.

I. S. MacKenzie and R. W. Soukoreff. Text entry for mobile computing: Models and methods, theory and practice. *Human–Computer Interaction*, 17(2-3):147–198, 2002b.

I. S. MacKenzie and R. W. Soukoreff. Phrase sets for evaluating text entry techniques. In *CHI'03 extended abstracts on Human factors in computing systems*, pages 754–755. ACM, 2003.

M. P. Marcus. *Theory of syntactic recognition for natural languages.* MIT press, 1980.

A. Matran-Fernandez, R. Poli, and C. Cinel. Collaborative brain-computer interfaces for the automatic classification of images. In *International IEEE/EMBS Conference on Neural Engineering*, pages 1096–1099, 11 2013. doi: 10.1109/NER.2013.6696128.

S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

T. Mikolov. *Statistical language models based on neural networks.* PhD thesis, Brno University of Technology., 2012.

K. Mrini, F. Dernoncourt, T. Bui, W. Chang, and N. Nakashole. Rethinking self-attention: An interpretable self-attentive encoder-decoder parser. *arXiv preprint arXiv:1911.03875*, 2019.

J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, 01 1965. ISSN 0010-4620. doi: 10.1093/comjnl/7.4.308. URL `https://doi.org/10.1093/comjnl/7.4.308`.

H. Ney, U. Essen, and R. Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language*, 8(1):1–38, 1994.

J. Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT).* Citeseer, 2003.

J. Nivre. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57. Association for Computational Linguistics, 2004.

B. S. Oken, U. Orhan, B. Roark, D. Erdogmus, A. Fowler, A. Mooney, B. Peters, M. Miller, and M. B. Fried-Oken. Brain–computer interface with language model–electroencephalography fusion for locked-in syndrome. *Neurorehabilitation and neural repair*, 28(4):387–394, 2014.

U. Orhan, K. E. Hild, D. Erdogmus, B. Roark, B. Oken, and M. Fried-Oken. Rsvp keyboard: An eeg based typing interface. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 645–648. IEEE, 2012.

T. Paek and B.-J. P. Hsu. Sampling representative phrase sets for text entry experiments: a procedure and public resource. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2477–2480. ACM, 2011.

E. M. Palmer, T. S. Horowitz, A. Torralba, and J. M. Wolfe. What are the shapes of response time distributions in visual search? *Journal of experimental psychology: human perception and performance*, 37(1):58, 2011.

P. Quinn and S. Zhai. A cost-benefit study of text entry suggestion interaction. In *Proceedings of the 2016 CHI conference on human factors in computing systems*, pages 83–88, 2016.

B. Roark, J. De Villiers, C. Gibbons, and M. Fried-Oken. Scanning methods and language modeling for binary switch typing. In *Proceedings of the NAACL HLT 2010 Workshop on Speech and Language Processing for Assistive Technologies*, pages 28–36. Association for Computational Linguistics, 2010.

B. Roark, R. Beckley, C. Gibbons, and M. Fried-Oken. Huffman scanning: using language models within fixed-grid keyboard emulation. *Computer speech & language*, 27(6):1212–1234, 2013a.

B. Roark, R. Beckley, C. Gibbons, and M. Fried-Oken. Huffman scanning: Using language models within fixed-grid keyboard emulation. *Computer speech & language*, 27 6, 2013b.

D. Rohrer and J. T. Wixted. An analysis of latency and interresponse time in free recall. *Memory & Cognition*, 22(5):511–524, 1994.

D. Rough, K. Vertanen, and P. O. Kristensson. An evaluation of dasher with a high-performance language model as a gaze communication method. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*, pages 169–176, 2014.

G. Salton, R. Ross, and J. Kelleher. Attentive language models. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 441–450, Taipei, Taiwan, Nov. 2017. Asian Federation of Natural Language Processing. URL `https://www.aclweb.org/anthology/I17-1045`.

O. Sandu, G. Carenini, G. Murray, and R. Ng. Domain adaptation to summarize human conversations. In *Proceedings of the Workshop on Domain Adaptation for Natural Language Processing*, pages 16–22. Association for Computational Linguistics, 2010.

B. Sankaran, A. Grewal, and A. Sarkar. Incremental decoding for phrase-based statistical machine translation. In *Proceedings of the Joint Fifth Workshop on Statistical Machine Translation and MetricsMATR*, pages 216–223. Association for Computational Linguistics, 2010.

D. Schlangen and G. Skantze. A general, abstract model of incremental dialogue processing. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, pages 710–718. Association for Computational Linguistics, 2009.

E. O. Selfridge, I. Arizmendi, P. A. Heeman, and J. D. Williams. Stability and accuracy in incremental speech recognition. In *Proceedings of the SIGDIAL 2011 Conference*, pages 110–119. Association for Computational Linguistics, 2011.

P. Y. Simard, D. Steinkraus, and J. C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition - Volume 2*, ICDAR '03, pages 958–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1960-1. URL `http://dl.acm.org/citation.cfm?id=938980.939477`.

G. Skantze and A. Hjalmarsson. Towards incremental speech generation in dialogue systems. In *Proceedings of the 11th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 1–8. Association for Computational Linguistics, 2010.

S. L. Smith and N. C. Goodwin. Alphabetic data entry via the Touch-Tone pad: A comment. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 13(2):189–190, 1971a.

S. L. Smith and N. C. Goodwin. Alphabetic data entry via the touch-tone pad: A comment. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 13(2):189–190, 1971b.

T. Stocky, A. Faaborg, and H. Lieberman. A commonsense approach to predictive text entry. In *CHI'04 Extended Abstracts on Human Factors in Computing Systems*, pages 1163–1166. ACM, 2004.

A. Stolcke. Entropy-based pruning of back-off language models. In *Proceedings of News Transcription and Understanding Workshop*, pages 270–274, Lansdowne, VA, 1998. DARPA.

P. Sturt and V. Lombardo. Processing coordinated structures: Incrementality and connectedness. *Cognitive Science*, 29(2):291–305, 2005.

K. Tanaka-Ishii. Word-based predictive text entry using adaptive language models. *Natural Language Engineering*, 13(01):51–74, 2007.

L. Tesnière. *Eléments de Syntaxe Structurale*. Klincksieck, Paris, 1959.

K. Trnka and K. F. McCoy. Evaluating word prediction: framing keystroke savings. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, pages 261–264. Association for Computational Linguistics, 2008.

K. Trnka, D. Yarrington, J. McCaw, K. F. McCoy, and C. Pennington. The effects of word prediction on communication rate for aac. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Companion Volume, Short Papers*, pages 173–176, 2007.

J. J. Van Berkum, C. M. Brown, P. Zwitserlood, V. Kooijman, and P. Hagoort. Anticipating upcoming words in discourse: evidence from erps and reading times. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 31(3):443, 2005.

A. van den Bosch and T. Bogers. Efficient context-sensitive word completion for mobile devices. In *Proceedings of the 10th international conference on Human computer interaction with mobile devices and services*, pages 465–470. ACM, 2008.

K. Vertanen and P. O. Kristensson. A versatile dataset for text entry evaluations based on genuine mobile emails. In *Proceedings of the International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '11, August 2011a.

K. Vertanen and P. O. Kristensson. The imagination of crowds: Conversational aac language modeling using crowdsourcing and large data sources. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 700–711. ACL, 2011b.

K. Vertanen and P. O. Kristensson. Complementing text entry evaluations with a composition task. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 21(2):8, 2014.

A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.

T. Wandmacher and J.-Y. Antoine. Training language models without appropriate language resources: Experiments with an aac system for disabled people. In *Proceedings of LREC*, 2006.

D. J. Ward, A. F. Blackwell, and D. J. MacKay. Dasher – a data entry interface using continuous gestures and language models. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 129–137. ACM, 2000.

D. Weir, H. Pohl, S. Rogers, K. Vertanen, and P. O. Kristensson. Uncertain text entry on mobile devices. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2307–2316. ACM, 2014.

G. Weiss, Y. Goldberg, and E. Yahav. On the practical computational power of finite precision RNNs for language recognition. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 740–745, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-2117. URL `https://www.aclweb.org/anthology/P18-2117`.

P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

I. H. Witten and T. C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *Information Theory, IEEE Transactions on*, 37(4):1085–1094, 1991.

J. R. Wolpaw, N. Birbaumer, D. J. McFarland, G. Pfurtscheller, and T. M. Vaughan. Brain–computer interfaces for communication and control. *Clinical neurophysiology*, 113(6):767–791, 2002.

Z. Xie, S. I. Wang, J. Li, D. Lévy, A. Nie, D. Jurafsky, and A. Y. Ng. Data noising as smoothing in neural network language models. *arXiv preprint arXiv:1703.02573*, 2017.

H. Yamada and Y. Matsumoto. Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, volume 3, 2003.

M. Yarmohammadi, V. K. R. Sridhar, S. Bangalore, and B. Sankaran. Incremental segmentation and decoding strategies for simultaneous translation. In *Proceedings of the International Joint Conference on Natural Language Processing*, pages 1032–1036. Asian Federation of Natural Language Processing, 2013.

Y. Yin, T. Y. Ouyang, K. Partridge, and S. Zhai. Making touchscreen keyboards adaptive to keys, hand postures, and individuals: a hierarchical spatial backoff model approach. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2775–2784. ACM, 2013.

D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and control*, 10(2):189–208, 1967.

D. M. Zajic, B. J. Dorr, and J. Lin. Single-document and multi-document summarization techniques for email threads using sentence compression. *Information Processing & Management*, 44(4):1600–1610, 2008.

S. Zhai and P. O. Kristensson. Shorthand writing on stylus keyboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 97–104, New York, NY, USA, 2003. ACM. ISBN 1-58113-630-7. doi: 10.1145/642611.642630. URL `http://doi.acm.org/10.1145/642611.642630`.

S. Zhai and P. O. Kristensson. The word-gesture keyboard: reimagining keyboard interaction. *Commun. ACM*, 55(9):91–101, Sept. 2012. ISSN 0001-0782. doi: 10.1145/2330667.2330689. URL `http://doi.acm.org/10.1145/2330667.2330689`.