JustCorrect: Intelligent Post Hoc Text Correction Techniques on Smartphones

Wenzhe Cui[1], **Suwen Zhu**[1], **Mingrui Ray Zhang**[2], **Andrew Schwartz** [1],
Jacob O. Wobbrock[2], **Xiaojun Bi**[1]
[1]Department of Computer Science, Stony Brook University, Stony Brook, NY, United States
[2]The Information School, University of Washington, Seattle, Washington, United States
{wecui, suwzhu, has, xiaojun}@cs.stonybrook.edu, {mingrui, wobbrock}@uw.edu

ABSTRACT
Correcting errors in entered text is a common task but usually difficult to perform on mobile devices due to tedious cursor navigation steps. In this paper, we present JustCorrect, an intelligent post hoc text correction technique for smartphones. To make a correction, the user simply types the correct text at the end of their current input, and JustCorrect will automatically detect the error and apply the correction in the form of an insertion or a substitution. In this way, manual navigation steps are bypassed, and the correction can be committed with a single tap. We solved two critical problems to support JustCorrect: (1) Correction Algorithm: we propose an algorithm that infers the user's correction intention from the last typed word. (2) Input Modalities: our study revealed that both tap and gesture were suitable input modalities for performing JustCorrect. Based on our findings, we integrated JustCorrect into a soft keyboard. Our user studies show that using JustCorrect reduces the text correction time by 12.8% over the stock Android keyboard and by 9.7% over the "Type, then Correct" text correction technique by Zhang et al. (2019). Overall, JustCorrect complements existing post hoc text correction techniques, making error correction more automatic and intelligent.

Author Keywords
Text entry; error correction; smartphones.

CCS Concepts
•**Human-centered computing** → **Human computer interaction (HCI);** *Interaction techniques;*

INTRODUCTION
Correcting errors in entered text is an inseparable yet difficult part of mobile text entry. The bottleneck lies in the need for precise and repetitive *manual* control. The *de facto* cursor-based text correction technique requires accurately positioning the cursor at the error text, repeatedly pressing backspace to delete errors, and re-positioning the cursor back at its original location. The recent "Type, then Correct" technique (TTC) [42] eliminated these cursor control operations by (1) letting the user "throw" the correction at the error text, or (2) pressing a key to locate error candidates and eventually commit the correction. However, TTC still requires users to specify the correction location. Our question is whether we can further simplify this correction process by reducing the necessary user actions even further. In particular, can we make *text correction* as efficient as *text entry*?

Figure 1. This figure shows how JustCorrect works. 1. The user enters a sentence with an error *jimo* using tap typing; 2. To correct *jimo* to *jumps*, they can either tap-type *jumps* and press the editing button (2a), or switch to gesture type *jumps*(2b). 3. JustCorrect then substitutes *jimo* with *jumps*. Two alternative correction options are also presented. The editing procedure involves no manual operations except entering the correct text.

UIST '20, October 20–23, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7514-6/20/10 ...$15.00.
http://dx.doi.org/10.1145/3379337.3415857

In this paper, we present JustCorrect, an intelligent post hoc text correction technique. To substitute an incorrect word or insert a missing word in the current entered sentence, the user simply types the correction at the end of their entered text, and JustCorrect will automatically commit the correction without user's intervention. Additional options are also provided for better correction coverage. In this way, JustCorrect makes post hoc text correction on the recently entered sentence as straightforward as text entry.

In creating and evaluating JustCorrect, we make the following contributions. (1) We offer a post hoc correction algorithm that infers a user's correction intention in the current entered sentence based on the newly entered word. (2) In our user study, we found both tap typing and gesture typing are appropriate input modalities for JustCorrect. Based on these findings, we propose two forms of JustCorrect: JustCorrect-Tap and JustCorrect-Gesture. The former uses tap typing, and the latter uses gesture typing for JustCorrect. Finally, (3) we integrated both JustCorrect-Tap and JustCorrect-Gesture into a soft keyboard. Our evaluation shows that using JustCorrect reduces post hoc text correction time by 12.8% over the stock Android keyboard, which uses *de facto* cursor-based text correction, and by 9.7% over TTC [42]. Overall, our research shows JustCorrect complements existing post hoc text correction techniques and makes text correction more automatic and intelligent than prior techniques. It well serves the users who type a full sentence ahead before checking mistakes or rephrasing the wording.

RELATED WORK

We review previous research on text correction and multimodal text input.

Error Correction Techniques on Smartphones

Correcting errors is an inseparable and costly part of the mobile text entry process [19, 29]. Previous work often adopted a cursor-based editing approach. For example, previous research proposed controlling cursor by using magnifying lens [3], pressing hard on the keyboard to turn it into a touchpad [3], or adding arrow keys [39]. Gestural operations have also been proposed to facilitate positioning cursor. Examples included using left and right gestures [11], sliding left or right from the space-key [16] to move the cursor, or using a "scroll ring" gesture along with swipes in four directions [43].

In addition to controlling cursor, a number of techniques have also been proposed to facilitate text selection. The default operations on a text field on Apple iOS devices [3] and Android include pressing and sliding the finger to select a word, holding the finger down on a word to select it, and double-tapping to select a word. Gestural operations have also been explored, such as using two-finger gestures [11], and clockwise gestures [16]. To edit the text after selecting it, modern keyboards [3] often adopt a widget- or menu-based approach: displaying a widget or pop-up menu filled with possible actions. Gesture-based command input methods [2, 8, 21] have been proposed to operate on the selected text.

These cursor-based correction methods often involve repetitive manual operations challenging for users due to small screen sizes and underlying ambiguities in finger touch locations [6, 15, 37]. To address these challenges, intelligent interaction techniques such as auto-correction was introduced. Auto-correction – automatically correcting the word being composed – has been a signature feature of modern keyboards on smartphones [12, 7, 36]. However, autocorrection is largely limited to the immediate word being composed, and not suitable for correcting errors in entered text. The smart-restorable backspace technique [4] adds intelligence to the backspace usage. It determines the number of deleted characters for each backspace press based on the predicted correction positions and can restore the previously deleted text. It can reduce the number of needed backspace presses in error correction. Differently, our work aims to completely bypass the backspace usage. Relatedly, grammar check has been widely adopted for correcting typos in entered text. For example, Gboard [24] allows a user to tap on a word and shows alternatives on suggestion bar. But correction is only limited to misspellings. Grammarly keyboard [17] continuously tracks entered text and provides suggestions on the suggestion bar. However, it offers all possible correction suggestions without knowing user's correction intention, which could clutter the suggestion bar. In contrast, JustCorrect adopts a more user-guided approach. The user decides when to trigger the correction and indicates which word she will use for correction.

The "Type, then Correct" technique [42] is a recent effort at reducing cursor operations by injecting intelligence into the post hoc text correction process. To correcct an error, a user types a correction word and either: (1) "throws" the word from the suggestion bar towards the error, or (2) drags their finger atop a designated "Magic Key" to navigate among highlighted error candidates, pressing the key to commit the correction. This technique saves cursor manipulation, but still requires manual operations to specify the correction location. JustCorrect removes these manual operations. As we demonstrate, our study shows that saving these additional manual operations significantly improves post hoc text correction speed over the "Type, then Correct" technique [42].

Multi-Modal Text Input

Many soft keyboards (e.g., Gboard [24]) support entering text via different modalities, such as tap typing, gesture typing, and voice input. Previous research has explored fusing information from multiple modalities to reduce text entry ambiguity, such as combining speech and gesture typing [28, 32], using finger touch to specify the word boundaries in speech recognition [31], or using unistrokes together with key landings [18] to improve input efficiency. In desktop computing, combining eye gaze with keyboard typing is an effective approach to improve text editing [33]. Our research investigated how different input modalities performed for JustCorrect. It involved multiple input modalities, but for correcting post hoc errors, not for text entry decoding which was the main focus of previous research. JustCorrect was particularly inspired by ReType [33], which used eye-gaze input to estimate the text editing location. We advanced it by inferring the editing intention based on the entered word only, making the technique suitable for mobile devices, which typically are not equipped with eye-tracking capabilities.

A USAGE SCENARIO FOR JUSTCORRECT

JustCorrect significantly improves text correction efficiency by allowing the user to enter a correction at the end of their text and simply applying it to a previous error. Before explaining the technical details, we first show a usage scenario.

Sarah was texting a message to her friend Tom when she typed: *We worked on the project lsst week*. She discovered a misspelling: *lsst*. Instead of moving the cursor five characters back, deleting the wrong characters, and typing the correct characters, Sarah simply typed the word *last* and pressed the edit button. JustCorrect automatically replaced *lsst* with *last*. Sarah also noticed that it might be better to replace *worked* with *focused*, so she typed *focused* at the end and pressed the edit button again to correct the word. Lastly, she wanted to insert the modifier *mainly* before *focused*. She gesture typed *mainly* and JustCorrect automatically completed the task for her. In this case, JustCorrect was triggered by switching from tap typing to gesture typing. The final sentence then became *We mainly focused on the project last week*. In this example, Sarah successfully corrected a typo, substituted a word, and inserted a new word without ever adjusting the cursor position.

THE POST HOC CORRECTION ALGORITHM

The key to JustCorrect lies in successfully inferring a user's editing intention based on the entered word and the prior context. To enable this, we developed a post hoc correction algorithm, as described below.

The post hoc correction algorithm takes the current entered sentence S and an editing word w^* as input, and revises S by either substituting a word w_i in S with w^*, or inserting w^* at an appropriate position. The post hoc correction algorithm offers three post hoc correction suggestions, with the top suggestion automatically adopted by default and the others easily selected with only one additional tap.

Take the sentence S = *a quick fox jimo over a lazy dog*. The user inputs *jumps* as the editing word w^*. Because the sentence has 8 words, there are 8 substitution and 9 insertion possibilities: *_a_quick_fox_jimo_over_a_lazy_dog_*. The 9 possible insertion positions are indicated by the underscores. The post hoc correction algorithm then generates 8 substitution candidates ($S_1 - S_8$), as shown in Table 1, and 9 insertion candidates ($I_1 - I_9$) as shown in Table 2.

Substitution candidates	$SubScore_i$	SS_i	ES_i	WS_i
S_1: **jumps** quick fox jimo over a lazy dog	0.56	0	0	0.56
S_2: a **jumps** fox jimo over a lazy dog	0.89	0.2	0.2	0.48
S_3: a quick **jumps** jimo over a lazy dog	0.42	0.42	0	0
S_4: a quick fox **jumps** over a lazy dog	1.71	1	0.6	0.11
S_5: a quick fox jimo **jumps** a lazy dog	0.75	0.18	0	0.57
S_6: a quick fox jimo over **jumps** lazy dog	0.56	0	0	0.56
S_7: a quick fox jimo over a **jumps** dog	1.11	0.11	0	1
S_8: a quick fox jimo over a lazy **jumps**	0.48	0.18	0	0.31

Table 1. An example of 8 substitution candidates. They are generated by replacing a word in the sentence "a quick fox jimo over a lazy dog" with "jumps". S_i means that i^{th} word in the sentence w_i is replaced by w^*. $SubScore_i$ is substitution score for ranking substitution candidates. SS_i, ES_i, and WS_i are scores from Edit Distance, Word Embedding, and Sentence channels, respectively.

Insertion candidates	$InserScore_i$
I_1: **jumps** a quick fox jimo over a lazy dog	0.06
I_2: a **jumps** quick fox jimo over a lazy dog	0.04
I_3: a quick **jumps** fox jimo over a lazy dog	0.52
I_4: a quick fox **jumps** jimo over a lazy dog	1
I_5: a quick fox jimo **jumps** over a lazy dog	0.91
I_6: a quick fox jimo over **jumps** a lazy dog	0.24
I_7: a quick fox jimo over a **jumps** lazy dog	0
I_8: a quick fox jimo over a lazy **jumps** dog	0
I_9: a quick fox jimo over a lazy dog **jumps**	0.5

Table 2. An example of 9 insertion candidates. They are generated by inserting "jumps" before or after every word in the sentence "a quick fox jimo over a lazy dog". I_i means w^* is inserted at the i^{th} insertion location. $InserScore_i$ is insertion score for ranking insertion candidates.

The algorithm then ranks the substitution candidates according to the substitution scores, and ranks the insertion candidates according to the insertion scores. These scores are later compared to generate ultimate correction suggestions.

Substitution Score

The substitution score reflects how likely it is that a substitution candidate represents the user's actual editing intention. It is calculated based on the assumption that there are two main intentions behind word substitutions: (1) correcting typos, or (2) replacing valid words with new words. We look for robust evidence of the substituted word along three dimensions: orthographic (i.e. character) distance, syntactosemantic (i.e. meaning) distance, and sequential coherence (i.e. making sense in context). More specifically, for the i^{th} substitution candidate S_i, its substitution score $SubScore_i$ is defined as:

$$SubScore_i = ES_i + WS_i + SS_i, \quad (1)$$

where ES_i is editing similarity, WS_i is word embedding similarity, and SS_i is the sentence score for substitution candidates (explained below). The edit distance channel ES_i is intended to handle spelling corrections. The edit distance between a typo and a correct word is usually small [38]. On the other hand, when replacing a word with a more preferred choice, e.g., replacing 'great' with 'fantastic', or replacing 'road' with 'path', the two words are both valid spellings and usually close in meaning. The word embedding channel WS_i captures similar meanings. Finally, the sentence channel SS_i ensures overall coherence of the word choice or replacement within its context. For example, in "the cost of that dresser is too great," replacing 'great' with 'fantastic' would change the meaning of the sentence, whereas inserting 'fantastic' before dresser would not.

Edit Distance Channel

The edit distance channel calculates the editing similarity for each substitution candidate. The Levenshtein edit distance [22] between two strings is the minimum number of single-character edits including deletions, insertions, or substitutions needed to transform one string into another string. For example, the Levenshtein edit distance between "heel" and "health" is 3: 1 edit for replacing *e* with *a* and 2 edits for inserting *t* and *h*. In this channel, we first calculate the Levenshtein edit distance $L(w_i, w^*)$ between the editing word w^* and the substituted word w_i in the i^{th} substitution candidate S_i. The

editing similarity ES_i is defined as:

$$ES_i = \frac{L(w^*, w_i)}{max(|w^*|, |w_i|)}, \quad (2)$$

where $max(|w^*|, |w_i|)$ denotes the max length of w^* and w_i. Equation 2 normalizes the edit distance score, similar to a previously introduced text entry error metric [34].

Word Embedding Channel

The word embedding channel estimates the semantic and syntactic similarity WS_i between the editing word w^* and the substituted word w_i in S_i. In an "embedding model", words from the vocabulary are mapped to vector of real numbers derived from statistics on the co-occurence of words within documents [10]. The distance between two vectors (i.e. word embeddings) can then be used as a measure of syntactic and semantic difference [1].

We learned our word embedding model over the "Text8" dataset [25] using the Word2Vec skip-gram approach [26]. Then, we calculate the cosine similarity $WS_C(w^*, w_i)$ between w^* and each w_i using word vectors [1]. For example, in the second row of Table 1, the substituted word $w_i =$`"quick"` is replaced by the editing word $w^* =$`"jumps"`. We then obtain the word embedding similarity WS_i by normalizing $WS_C(w^*, w_i)$ in the range [0,1].

Sentence Channel

The sentence channel estimates the normalized sentence score of S_i using a language model – a model that estimates the probability of a sequence of words.

To compute the language model probability for a given sentence, we trained a 3-gram language model using the KenLM Language Model Toolkit [14], which is a memory- and time-efficient implementation of a Kneser-Ney smoothed *language model* [27]. Based on word frequencies, word pairs, and word triples, a 3-gram *language model* takes each substitution candidate S_i as the input, and outputs its estimated log probability $P(S_i)$. By normalizing $P(S_i)$ in the range of 0 to 1, we get the normalized sentence score SS_i:

$$SS_i = \frac{P(S_i) - min(P(S_j))}{max(P(S_j)) - min(P(S_j))}, (j = 1, 2, ..., N) \quad (3)$$

where $min(P(S_j))$ and $max(P(S_j))$ are the minimum and maximum sentence channel scores among all the N substitution possibilities, assuming the sentence S has N words. The language model itself was trained over the Corpus of Contemporary American English (COCA) [9] (2012 to 2017), which contains over 500 million words. The fitted language model file was compiled into a binary file to accelerate processing.

Insertion Score

For insertion candidates, we only use the *sentence channel* for insertion scores, as there are no word-to-word comparisons for insertion candidates. Assuming S has N words and therefore $N+1$ candidates for insertion, the insertion score $InserScore_i$ for the candidaite I_i is calculated as:

$$InserScore_i = \frac{P(I_i) - min(P(I_j))}{max(P(I_j)) - min(P(I_j))}, (j = 1, 2, ..., N+1) \quad (4)$$

where $min(P(I_j))$ and $max(P(I_j))$ are the minimum and maximum sentence channel scores among all the $N+1$ insertion possibilities $(I_1, I_2, ..., I_{N+1})$. As shown, $InserScore_i$ is normalized in [0,1].

Combining Substitution and Insertion Candidates

The post hoc correction algorithm combines the substitution and insertion candidates to generate correction suggestions according to the pseudocode in Algorithm 1. It outputs three correction suggestions, and automatically commits the top suggestion to the text (see Figure 1, part 3). The algorithm first compares top suggestions from the substitution candidate list and the insertion candidate list, respectively. The one with a higher log probability($P(I_i)$ or $P(S_i)$) in its sentence channel is the top correction suggestion of combined candidates, while the other is the second suggestion. This operation ensures that at least one substitution and one insertion will be provided to the user. We compare substitution and insertion candidates by their log probabilities in sentence channel because sentence channel is the common component between these two types of suggestions: insertion score is calculated by sentence channel only; one of the three channels for substitution scores is sentence channel. Using sentence log probability could avoid potential bias toward substitution candidates.

EXPERIMENT 1: EVALUATING THE POST HOC CORRECTION ALGORITHM WITH DIFFERENT INPUT MODALITIES

To understand whether the post hoc correction algorithm is effective, especially when combined with different input modalities, we evaluated three forms of JustCorrect: JustCorrect-Gesture, JustCorrect-Tap, and JustCorrect-Voice. These variations are different JustCorrect techniques with different input modalities, as explained below.

Participants

We recruited 16 participants (four females) from 19 to 40 years old ($Mean = 26.4, Std. = 4.4$). All were right-handed. The self-reported median familiarity with tap typing, gesture typing, and voice input (1: not familiar, 5: very familiar) were 5.0, 3.5, and 2.5 respectively. Seven participants had gesture typing experience. The participants were instructed to use their preferred hand posture throughout the study.

Apparatus

A Google Nexus 5X device (Qualcomm Snapdragon 808 Processor, 1.8GHz hexa-core 64-bit Adreno 418 GPU, RAM: 2GB LPDDR3, Internal storage: 16GB) with a 5.2 inch screen (1920×1080 LCD at 423 ppi) was used for the experiment.

Design

The study was a within-subjects design. The sole independent variable was the text correction method with four levels:

- *Cursor-based Correction*. This was identical to the existing *de facto* cursor-based text correction method on the stock Android keyboard.

Algorithm 1 Post Hoc Correction Algorithm

procedure GET CORRECTION RESULT
input:
  w^* ← editing word
  S ← editing sentence
process:
  SC : list of substitution candidates
  IC : list of insertion candidates
  SC ← creating substitution candidate list (e.g., table 1)
  IC ← create insertion candidate list (e.g., table 2)
  calculate substitution scores for each S_i **in** $SC(eq.\ (1))$
  calculate insertion scores for each I_i **in** $IC(eq.\ (4))$
  $SortedSC$ ← sort SC by descending substitution scores
  $SortedIC$ ← sort IC by descending insertion scores
  $P(SC0)$ ← Sentence log probability of $SortSC(0)$
  $P(IC0)$ ← Sentence log probability of $SortIC(0)$
  if $P(SC0) > P(IC0)$ **then**
    $firstSuggestion \leftarrow SortedSC(0)$
    $secondSuggestion \leftarrow SortedIC(0)$
  else
    $firstSuggestion \leftarrow SortedIC(0)$
    $secondSuggestion \leftarrow SortedSC(0)$
  $P(SC1)$ ← Sentence log probability of $SortedSC(1)$
  $P(IC1)$ ← Sentence log probability of $SortedIC(1)$
  if $P(SC1) > P(IC1)$ **then**
    $thirdSuggestion \leftarrow SortSC(1)$
  else
    $thirdSuggestion \leftarrow SortIC(1)$
output:
  $firstSuggestion, secondSuggestion, thirdSuggestion$

- *JustCorrect-Tap*. After entering a word with tap typing, the user presses the editing button to invoke the post hoc correction algorithm (see Section 1).

  Taking the sentence "`a quick fox jimo over a lazy dog`", for example, if the user wants to replace "`jimo`" with "`jumps`", she tap types the editing word "`jumps`" and then presses the editing button (see Section 1). The post hoc correction algorithm takes "`jumps`" as the editing word and outputs "`a quick fox jumps over a lazy dog`".

- *JustCorrect-Gesture*. A user performed JustCorrect with gesture typing [20, 41, 40]. After entering the correction word w^* with a gesture and the finger lifts off, the system applied the post hoc correction algorithm to correct the existing phrase with the word. The other interactions were the same as those in JustCorrect-Tap. The only difference is that in JustCorrect-Tap a button was used to trigger JustCorrect because tap typing required a signal to indicate the end of inputting a word, while this step is omitted in JustCorrect-Gesture because gesture typing naturally indicates the end of entering a word when the finger lifts.

- *JustCorrect-Voice*. A user performed JustCorrect with voice input. The user first pressed the voice input button on the keyboard, and spoke the editing word. The post hoc correction algorithm took the recognized word from a speech-to-text recognition engine as the editing word w^* to edit the phrase. We used the Microsoft Azure speech-to-text engine [5] for speech recognition. The remaining interactions were identical to the previous two conditions.

Procedure

Each participant was instructed to correct errors in the same set of 60 phrases in each condition, and the orders of the sentences were randomized. We randomly chose 60 phrases with omission and substitution errors from Palin et al.'s mobile typing dataset [29]. This dataset included actual input errors from 37,370 users when typing with smartphones, and their target sentences. We focused on omission and substitution errors since the post hoc correction algorithm was designed to handle these two types of errors. We also filtered out sentences with punctuation or number errors because our focus was on word correction. Among 60 phrases, 8 contained omission errors, and the rest contained substitution errors. The average(SD) edit distance between the sentence with errors and target sentences was 1.9(1.2). Each phrase contained an average(SD) of 1.1(0.3) errors. The average length of a target phrase in this experiment was 37 ± 14 characters. The largest phrase length was 68 characters, and the shortest was 16 characters. Table 3 shows 4 examples of phrases in experiment.

Sentences with errors	Target sentences
1. Tjank for sending this	Thanks for sending this
2. Should systematic manage the migration	Should systems manage the migration
3. Try ir again and let me know	Try it again and let me know
4. Kind like silent fireworks	Kind of like silent fireworks

Table 3. Examples of phrases in the experiment. The first three sentences contained substitution errors. The last sentence contained an omission error.

In each trial, participants were instructed to correct errors in the "input phrase" so that it matched the "target phrase" using the designated editing method. Both the input phrase and the target phrase were displayed on the screen. The errors in the input phrase were underlined to minimize the cognitive effort required to identify errors across conditions, as shown in Figure 2. The participants were required to correct errors in their current trial before advancing to their next trial.

Should a participant fail to correct the errors in the current trial, they could use the *undo* button to revert the correction and redo it, or use the *de facto* cursor-based editing method. We kept the cursor-based method as fallback in each editing condition because our JustCorrect techniques were proposed to *augment* rather than *replace* it. We recorded the number of trials corrected by this fallback mechanism in order to measure the effectiveness of each JustCorrect technique.

Prior to each condition, each participant completed a warm-up session to familiarize themselves with each method. The sentences in the warm-up session were different from those in the formal test. After the completion of each condition, participants took a three minutes break. The order of the four conditions was counterbalanced using a balanced Latin Square.

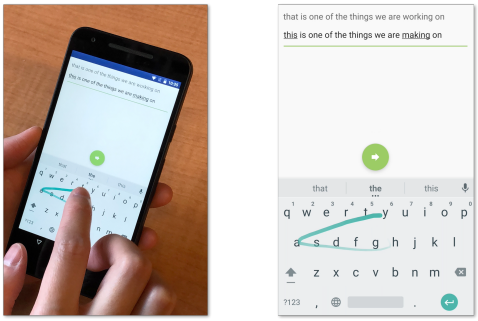

Figure 2. A user editing a sentence using *JustCorrect-Gesture*. The target sentence is displayed at the top of the screen, and the sentence with errors is displayed below. The underlines show two errors in the phrase: this –> that, making –> working. The user is shown gesture typing the word *that* to correct the first error.

In total, the experiment included: 16 participants × 4 conditions × 60 trials = 3,840 trials.

Results

Text Correction Time

We defined the "text correction time" as the duration from when a sentence was displayed on the screen to when it was submitted and completely revised. Thus, this metric conveys the efficiency of each JustCorrect text correction technique.

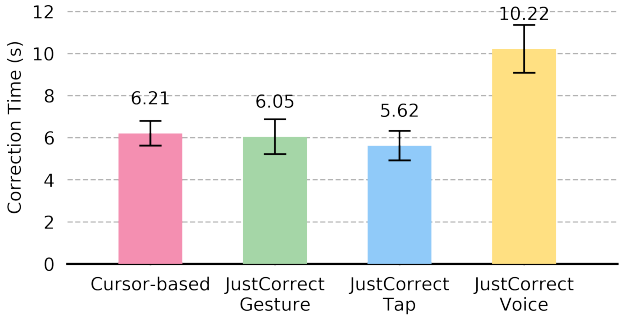

Figure 3. Mean (95% CI) text correction times for each method for successful trials.

Figure 3 shows text correction time for trials that were successfully corrected using the designated editing method in each condition (unsuccessful trials are described below in the next subsection). The mean ± 95% CI of text correction time was 6.21 ± 0.59 seconds for the *de facto* cursor-based technique, 6.05 ± 0.83 seconds for JustCorrect-Gesture, 5.62 ± 0.70 seconds for JustCorrect-Tap, and 10.22 ± 1.14 seconds for JustCorrect-Voice. A repeated measures ANOVA showed that the text correction technique had a significant main effect of on overall trial time ($F_{3,45} = 71.96, p < .001$). Pairwise comparisons with Bonferroni correction showed that differences were statistically significant between all pairs ($p < 0.001$) except for JustCorrect-Tap vs. JustCorrect-Gesture ($p = 0.17$) and JustCorrect-Gesture vs. the cursor-based technique ($p = 0.67$).

To understand the effectiveness of the algorithm under different conditions, we analyzed cases which were successfully edited in the first editing attempt. In total, there were 3328 such trials, among 3840 total trials. We grouped these trials by edit distance between the target sentence and the incorrect sentence. The average text correction times on different methods are shown in Figure 4. When the edit distance was 1, the correction times in *de facto* cursor-based technique were close to those in the gesture-based and tap-based techniques. When the edit distance was 2, 3 or 4, the gesture- and tap-based techniques were faster than the *de facto* baseline.

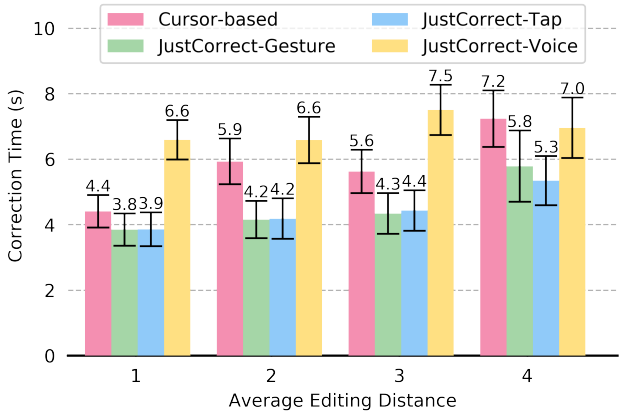

Figure 4. Mean (95% CI) text correction times for the tasks successfully completed on the first attempt.

Success Rate

We define the success rate as the percentage of correct trials out of all trials for a given correction technique. Figure 5 shows success rates across conditions. The mean ± 95% CI for success rate for each input technique was: $100.0 \pm 0\%$ for the *de facto* cursor-based technique, $96.2 \pm 2.2\%$ for JustCorrect-Gesture, $97.1 \pm 0.03\%$ for JustCorrect-Tap, and $95.1 \pm 0.03\%$ for JustCorrect-Voice. A repeated measures ANOVA showed that text editing technique had a significant effect of on the overall success rate ($F_{3,45} = 14.31, p < .001$). Pairwise comparisons with Bonferroni correction showed the difference was significant between JustCorrect-Tap vs. Cursor-based, JustCorrect-Gesture vs. Cursor-based, JustCorrect-Voice vs. Cursor-based ($p < 0.01$). All other pairwise comparisons were not statistically significant.

We discovered that there were some cases that were challenging to correct with our JustCorrect techniques (i.e., JustCorrect-Tap, JustCorrect-Gesture, and JustCorrect-Voice). In the first example of Table 4, some users found it was difficult to input "John" correctly using JustCorrect-Voice. In the second example, users sometimes failed to input *contract* correctly because they often mistyped it as *contact*.

Sentences with errors	Target Sentences
phone this message concern me	John this message concerns me
Has Brian had his concert yet	Has Brian had his contract yet

Table 4. Two examples of sentences that were hard to correct with JustCorrect (JustCorrect-Tap, JustCorrect-Gesture, and JustCorrect-Voice).

Subjective Feedback

At the end of the study, we asked the participants to rate each method on a scale of 1 to 5 (1: dislike, 5: like). As

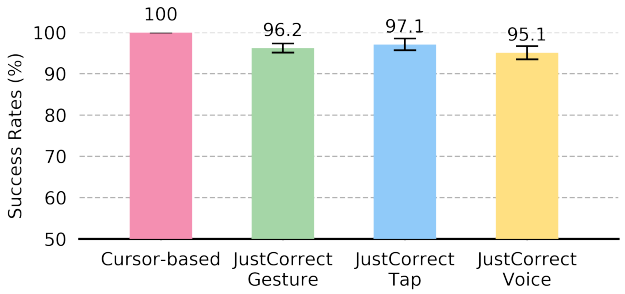

Figure 5. Success rate by input technique.

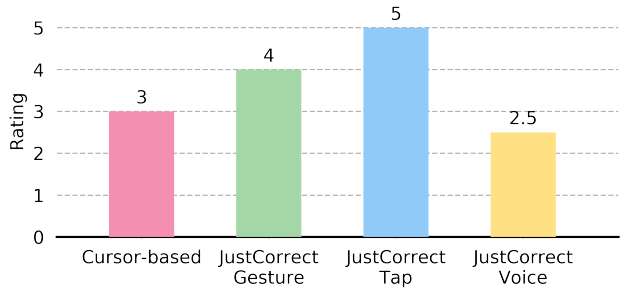

Figure 6. The median rating for cursor-based correction, JustCorrect-Gesture, JustCorrect-Tap and JustCorrect-Voice.

shown by Figure 6, the median rating for cursor-based editing, JustCorrect-Gesture, JustCorrect-Tap, and JustCorrect-Voice was 3.0, 4.0, 5.0, and 2.5, respectively. A non-parametric Friedman test of differences among repeated measure was carried out to compare the ratings for the four conditions. There was a significant difference between the methods ($X_r^2(3) = 17.29, p < 0.001$).

Participants were also asked which method(s) they would like to use during text entry on their phones. Twelve participants mentioned they would use JustCorrect-Tap, and eight would also like to use JustCorrect-Gesture. Six participants also considered the *de facto* cursor-based method useful, especially for revising short words or character-level errors. Only two participants liked to use JustCorrect-Voice for text editing, while most participants had privacy concerns about using it in a public environment.

Discussion

Our investigation led to the following findings.

First, both JustCorrect-Gesture and JustCorrect-Tap showed good potential as correction methods. Both JustCorrect-Gesture and JustCorrect-Tap successfully corrected more than 95% of the input phrases. They both saved average correction time over the *de facto* cursor-based correction method. These two methods were especially beneficial for correcting sentences that had large editing distances relative to the target sentences. As shown in Figure 4, for sentences with an editing distance of 4, JustCorrect-Gesture and JustCorrect-Tap reduced correction time by nearly 30% over the cursor-based method.

Second, JustCorrect-Gesture and JustCorrect-Tap exhibited their own pros and cons. Participants had differing preferences: users who were familiar with gesture typing liked JustCorrect-Gesture because it did not require pressing the editing button, while other participants preferred JustCorrect-Tap because they mostly used tap-typing for text entry. JustCorrect-Gesture saved the editing button-tap compared to JustCorrect-Tap because gesture typing naturally signals the end of entering a word by the lifting of the finger. On the other had, in JustCorrect-Gesture, gesture typing is used to correct text only, limiting its scope of usage. We implemented both of the methods on our keyboard prototype (see below) and investigated how users would choose between them.

Third, the cursor-based text editing method serves as a reliable fallback technique. JustCorrect-Gesture and JustCorrect-Tap failed to edit 3 - 4 % of input phrases, while the cursor-based editing method was successful for all phrases. We suggest using JustCorrect-Gesture and JustCorrect-Tap to complement the cursor-based editing method, rather than replacing it.

Fourth, contrary to the promising performance of JustCorrect-Gesture and JustCorrect-Tap, JustCorrect-Voice underperformed. The reason was that JustCorrect required a user to first enter the editing word, but the existing speech-to-text recognition engine often performed poorly when recognizing a single word in isolation, especially for short words. We discovered that entering common words such as *for*, *to*, and *are* are challenging when using voice, which caused difficulty in correcting phrases with errors on these words.

Overall, our study suggested that JustCorrect nicely augments a soft keyboard; both tap typing and gesture typing are promising modalities for JustCorrect. As a result, we developed a fully functional keyboard prototype implementing both JustCorrect-Tap and JustCorrect-Gesture, called JustCorrect Keyboard, and systematically evaluated it against existing text editing methods.

AUGMENTING A SOFT KEYBOARD WITH JUSTCORRECT

Based on the findings and lessons learned from Experiment 1, we augmented a soft keyboard with JustCorrect, called JustCorrect Keyboard.

Supporting JustCorrect-Tap and JustCorrect-Gesture

First, we integrated the two most promising JustCorrect techniques: JustCorrect-Gesture and JustCorrect-Tap, into the keyboard. We expected both of them would *complement* the *de facto* cursor-based correctly method. In other words, JustCorrect-Gesture, JustCorrect-Tap, and the cursor-based correction method all co-existed together in JustCorrect Keyboard.

As in Experiment 1, the default setting is that a user triggers JustCorrect-Tap by pressing the editing button, and triggers JustCorrect-Gesture by switching from tap typing to gesture typing. In the case where a user wants to have gesture typing available for text entry and not just text editing, the keyboard has an option for using the editing button to trigger JustCorrect-Gesture. If this option is selected, a user needs to press the editing button after gesture typing to trigger JustCorrect-Gesture,

just as with JustCorrect-Tap. This option keeps the gesture typing available for regular text entry.

Double-Tapping as a Fallback

Second, we introduced double-tapping as a fallback technique to address the problem that some errors were hard to correct with JustCorrect, as revealed by Experiment 1. It works as follows: In the case where JustCorrect cannot provide accurate correction suggestions, a user can refine the correction location by double-tapping the area. Then the keyboard automatically applies the correction to the specified area. More specifically, if the user double-taps a word, the post hoc text correction algorithm will substitute the tapped word w_i with the editing word w^*, and insert w^* before or after the tapped word as the two additional suggestions. If the user double-taps on a space between two words, the algorithm will insert w^* into the space, and substitute the words before and after the space with w^* for the two additional suggestions.

JustCorrect Keyboard, was implemented based on the Android AOSP keyboard. It used a commonly known statistical decoder [12] for tap typing input, and a commonly known gesture typing algorithm [20, 41] to decode gestures. It used a trigram language model with a lexicon size of 60K words.

After the integration, we evaluated JustCorrect Keyboard in a controlled experiment.

EXPERIMENT 2: EVALUATING JUSTCORRECT KEYBOARD

We conducted a controlled experiment to formally evaluate JustCorrect Keyboard in a post hoc text correction task. Our goal was to understand whether having multiple text correction techniques available together would benefit users. In other words, we aimed to understand whether JustCorrect complements existing text correction methods. On JustCorrect Keyboard, users could choose whatever correction method they preferred, including JustCorrect-Tap, JustCorrect-Gesture, the cursor-based method, or the newly added fallback method of double-tapping to indicate the correction position.

Our experiment included two studies. The first study compared JustCorrect Keyboard with the stock Android Keyboard with cursor-based method, while the second study compared JustCorrect Keyboard with the recently published "Type, then Correct" (TTC) keyboard [42]. The two studies were almost identical except for the levels of independent variable and participants. We ran these two separate studies to minimize the potential carryover effects from learning across conditions.

Participants and Apparatus

In the first study, we recruited 16 participants (4 females) between 21 and 30 years old ($Mean = 25.4, Std. = 2.3$). The self-reported median familiarity with tap typing and gesture typing (1: not familiar, 5: very familiar) were 4.5 and 4.0, respectively.

In the second study, we recruited an entirely different group of 16 participants (4 females) between 19 and 24 years old ($Mean = 21.1, Std. = 1.3$). The self-reported median familiarity with tap typing and gesture typing (1: not familiar, 5: very familiar) were 5.0 and 4.0, respectively. A Google Nexus 5X smartphone was used in both studies. A 2017 Macbook Pro (Processor: 2.9 GHz Quad-Core Intel Core i7, Memory: 16 GB 2133 MHz LPDDR3) with a 15-inch screen was used as the server for "Type, then Correct" (TTC) condition [42].

Design

Both studies 1 and 2 adopted within-subject designs. The independent variable was the keyboard with different text correction methods in both studies. The only difference was that this independent variable had different levels in two studies.

In study 1 the independent variable had two levels : (1) Android stock keyboard with the *de facto* cursor-based method for correction and (2) JustCorrect Keyboard, the keyboard augmented by JustCorrect.

In study 2, the first level of the independent variable changed. Its two levels were: (1) "Type, then Correct" (TTC) keyboard [42], and (2) JustCorrect Keyboard.

Procedure

We designed a text editing task similar to Experiment 1. We first randomly chose 120 sentences with errors from Palin's dataset [29]. This dataset included both the target sentences and the input sentences with errors from users. Palin's dataset [29] was collected on mobile devices, which suited our study well. We then evenly divided 120 sentences into "Set 1" and "Set 2", and balanced the number of errors in both sets. Among the 60 sentences in each set, 8 of them had word omission errors, and 52 had word substitution errors. The average length of sentences was 41 character long for "Set 1" and 42 character long for "Set 2". The average(SD) editing distance between the input sentences and target sentences were 2.0(1.4) for "Set 1" and 2.2(1.6) for "Set 2".

In the first study, each participant was instructed to correct phrases containing errors to transform them into target phrases. They would not advance to the next trial until all the errors were corrected. The two levels of the independent variable were counterbalanced across participants. Half of the participants first edited "Set 1" with JustCorrect Keyboard and then edited "Set 2" with the cursor-based method of the stock Android keyboard. The other half did so in the opposite order. A similar experiment design was used in prior work [13, 23, 30, 35, 36] to avoid potential carryover effects from learning. Each participant completed a warm-up session for 3-5 minutes using both methods before the study.

The second study followed exactly the same design as the first. The only difference was that the two keyboards in this study were JustCorrect Keyboard and TTC [42]. In the TTC condition, users could use either the "Magic Key" or "Throw" methods for correction. These were the two best performing designs according to the authors' published report [42].

Results

Error Rate

Because participants were required to successfully correct errors in a trial to advance to the next one, no error was left. In other words, error rate was 0% for all trials. Text correction

time became the main quantitative metric for measuring the performance.

Text Correction Time

As in Experiment 1, text correction time was the duration from when a sentence was displayed on the screen to when it was submitted and completely revised. Figure 7 shows mean±95% CI text correction time for all trials for JustCorrect Keyboard and the cursor-based methods in the first study, and for JustCorrect Keyboard and TTC methods in the second study.

In the first study, the mean±95% CI text correction time across trials was 6.90±0.67 seconds for the cursor-based method, and 6.02±0.58 seconds for JustCorrect Keyboard, as shown in Figure 7. A paired-samples t-test indicates that the difference was statistically significant ($t_{15} = 2.52, p = 0.0237$).

In the second study, the average text correction time for each task was 7.30±0.56 seconds for TTC, and 6.59±0.74 seconds for JustCorrect Keyboard, as shown in Figure 7. A paired-samples t-test indicates that this difference was statistically significant ($t_{15} = 3.37, p = 0.0042$).

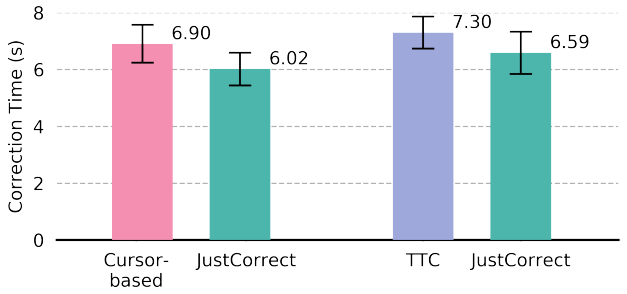

Figure 7. Mean (95% CI) text correction time for all trials per condition in the first study (left) and the second study (right). Lower is better.

To understand how participants performed as they progressed in the studies, we grouped the 60 trials evenly into 6 blocks. The first 10 trials formed Block 1 while the last 10 trails formed Block 6. Figure 8 shows mean (95% CI) text correction time across blocks. As shown, the mean text correction time in the JustCorrect Keyboard condition was lower than that in cursor-based condition and in the TTC condition for the majority of the trial blocks.

Correction Behavior

To further understand users' behaviors, we analyzed the percentage of each method's feature usage in both studies (Figure 9). Both JustCorrect Keyboard and TTC have multiple text editing methods available, while the cursor-based method has only one editing method available. With JustCorrect Keyboard, participants used either JustCorrect-Tap or JustCorrect-Gesture to edit more than 85% of trials. They occasionally used the cursor-based method for trials that were hard to edit for JustCorrect-Tap or JustCorrect-Gesture. The mixed usage of different editing methods showed that these editing methods complemented each other. Participants took advantage of the automatic editing method most of the time, but also for 12.3% of trials in study 1 and 5.7% of trials in study 2 they reverted to the cursor-based method.

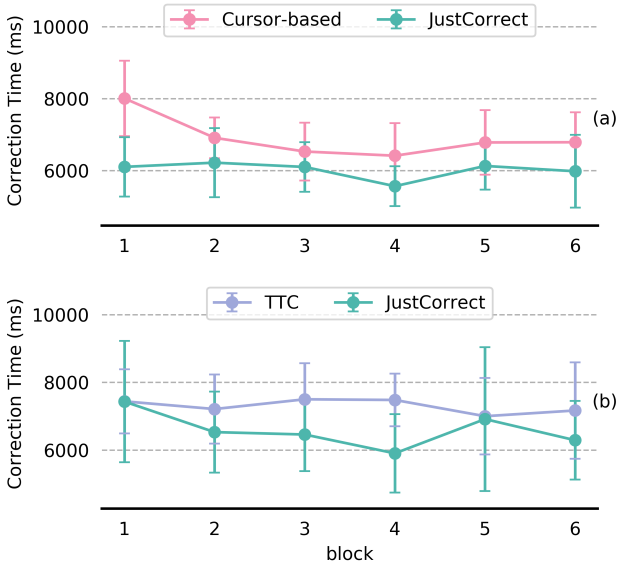

Figure 8. Average (95% CI) text correction time (seconds) by block in (a) the cursor-based condition and in the JustCorrect Keyboard condition, and in (b) the TTC and JustCorrect Keyboard conditions. Blocks were formed according to the testing order of trials. Each block had 10 trials. Lower is better.

The second study comparing TTC to JustCorrect Keyboard showed that JustCorrect Keyboard reduced the text correction time compared to TTC. We particularly broke the text correction time for both JustCorrect Keyboard and TTC into two parts: (1) target word input time, which represented the amount of time for entering the target word, and (2) editing time, which represented the time for using the entered word to edit a sentence. Figure 10 showed JustCorrect Keyboard saves overall text correction time over TTC, probably because JustCorrect Keyboard required only minimal user operation to edit the sentence after the target word was entered.

In the second study, we also discovered that on average users reverted to the cursor-based method for 11.1 (SD = 3.2) trials in TTC, and for 4.6 (SD = 2.2) trials in JustCorrect. The average number of TTC (or JustCorrect) usage before reverting to cursor-based method among these trials were 1.4 (SD = 0.3) for TTC and 1.6 (SD = 0.5) for JustCorrect, indicating that users quickly switched to the cursor-based method after discovering TTC (or JustCorrect) failed to correct the errors.

Subjective Feedback

At the end of the study, subjects were asked to provide a numerical rating (1: least demanding, 10: most demanding) on mental and physical demand. Mental demand describes how much mental effort is required. Physical demand describes how much physical effort is required. Figure 11 showed the mean subjective ratings. Subjects' subjective ratings were in favor of JustCorrect Keyboard for physical demand. For mental demand, for the cursor-based method and for JustCorrect Keyboard, the scores were approximately the same. In the TTC and JustCorrect Keyboard conditions, subjective ratings were in favor of TTC for mental demand. We also

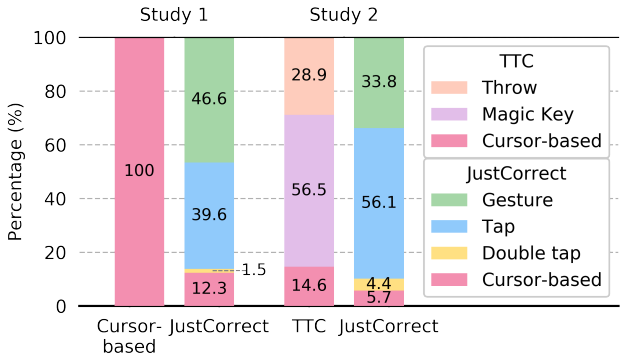

Figure 9. The percentage of different text editing features used per condition. Note that JustCorrect Keyboard, JustCorrect-Tap, JustCorrect-Gesture, and cursor-based editing method were all available together; for TTC, the throw, Magic Key, and cursor-based methods were all available together.

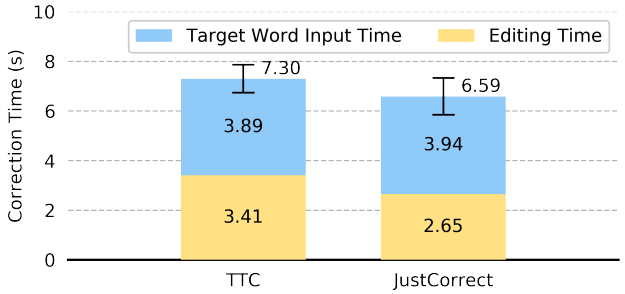

Figure 10. Mean (95% CI) target word input time and editing time for TTC and JustCorrect Keyboard in the second study.

asked the participants to rate each method on a scale of 1 to 5 (1: dislike, 5: like). The median rating for cursor-based method and JustCorrect Keyboard was 3 and 5 respectively. A Wilcoxon Signed-Ranks Test indicated that the subjective ratings of JustCorrect Keyboard was significantly higher than that of Cursor-based method ($Z = 3.07, p = 0.002$). Indeed, median Pain Score rating was 5.0 both pre- and post-treatment. For TTC and JustCorrect Keyboard, the median rating was 3 and 5, respectively. A Wilcoxon Signed-Ranks Test indicated that the subjective ratings of JustCorrect Keyboard was significantly higher than that of TTC ($Z = 2.97, p = 0.003$).

Participants were also asked which methods on JustCorrect Keyboard they would like to use for real-world text entry on their phones. Twelve participants chose JustCorrect-Gesture, and eight chose JustCorrect-Tap.

Discussion

First, the study results showed that adopting JustCorrect substantially improved text correction efficiency. JustCorrect Keyboard shortened the average text correction time by 12.8% over the stock Android keyboard, and by 9.7% over the TTC keyboard. The subjective ratings were also overwhelmingly in favor of JustCorrect Keyboard. The improved efficiency was largely attributed to the reduction of user intervention. For example, compared with the TTC keyboard, as shown in Figure 10, the time saved with JustCorrect Keyboard occurred mainly in the manual operation stage when editing.

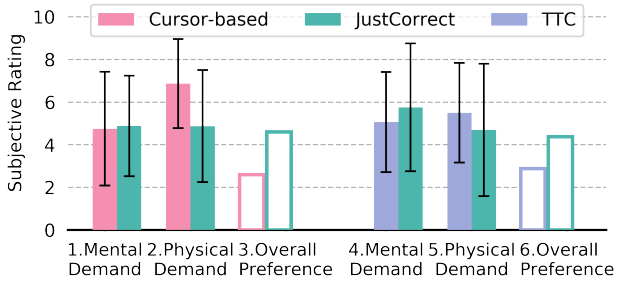

Figure 11. Mean (SD) of subjective ratings and median of overall preference. For measure 1, 2, 4 and 5, a lower rating means lower mental and physical demand. For measure 3 and 6 (1: least, 5: most preferred), a higher score means the method is more preferred. JustCorrect Keyboard received favorable ratings in categories 2, 5, and 6.

Second, in the JustCorrect Keyboard conditions, participants corrected a majority (87.7% in the first study and 94.3% in the second study) of the sentences using JustCorrect, and corrected the rest using the cursor-based method. This result showed that participants were able to take advantage of the high efficiency of JustCorrect, and reverted to cursor-based correction when necessary. Our results therefore showed that different text correction methods can complement each other.

Third, participants had split preferences on JustCorrect-Tap and JustCorrect-Gesture (Figure 9). In the first study, 39.6% of all trials were corrected by JustCorrect-Tap, and 46.6% were corrected by JustCorrect-Gesture; in the second study, 56.1% were by JustCorrect-Tap, and 33.8% were by JustCorrect-Gesture. Among the 32 participants in both studies, 15 used only JustCorrect-Tap on JustCorrect Keyboard, and the others used a mix of JustCorrect-Tap and JustCorrect-Gesture on JustCorrect Keyboard. Some users commented that they used JustCorrect-Gesture because it saved the button-pressing action compared to JustCorrect-Tap. Some users preferred JustCorrect-Tap over JustCorrect-Gesture because they were not familiar with gesture typing to begin with.

Overall, the results show incorporating JustCorrect significantly improves text correction efficiency. Having an automatic and intelligent post-hoc text correction benefits users.

LIMITATION AND FUTURE WORK

JustCorrect is designed to facilitate error correction, which is only one of the text editing actions. Other editing actions such as changing text formats and copying/pasting text are beyond the scope of the present work.

JustCorrect assumes that users type a full sentence and then check typing mistakes or rephrase their wording, so it will mostly be useful for users who type a sentence ahead. If a user corrects mistakes during the middle of entering a sentence, the edit distance and word embedding channels will still be able to make appropriate corrections, but the sentence channel may be affected. The reason is that the edit distance and word embedding channels use only word-level information for correction which will not be affected by the incomplete sentence. In contrast, the sentence channel uses the trigram language model to estimate the sentence score. It may be

affected by the incomplete sentence because the unseen words and missing sentence ending may affect the prediction made by the trigrams in the language model. For correcting errors in the middle of entering a sentence, one option is to increase the weights of edit distance and word embedding channels and decrease the weight of sentence channel in the post hoc correction algorithm. Future research is needed to investigate whether this option is effective.

JustCorrect is designed based on the assumption that there are two main types of word substitutions: 1. replacing a string with a word which shares similar characters, or 2. replacing a word with a new word which has a similar meaning. The type 1 substitution is often observed in correcting typos, simple grammatical mistakes, or false autocorrection, because in these cases the intended word often shares similar characters with the incorrect text. The type 2 substitution is often observed in rephrasing the wording. If a user wants to replace one word with multiple words such as replacing *making* with *working on*, JustCorrect should be triggered in two steps: first substituting *making* with *working*, and then inserting *on*.

The scope of JustCorrect is limited to the most recently entered sentence. Because the post hoc correction algorithm adopts an exhaustive search algorithm to determine the intended editing location, scaling it up to cover more text would be challenging. If the search scope is beyond one sentence, the algorithm would benefit from additional location information input such as using the finger touch to approximately specify the search area (e.g., double-tapping).

Additionally, because JustCorrect relies on the keyboard to decode any input word w_i from input signals, the keyboard decoding algorithm may affect the entering of the editing word (denoted by w_t because it is the last entered word). For example, the word entered right before entering the editing word (denoted by w_{t-1}) may negatively influence the keyboard decoding of the editing word w_t, because the editing word w_t will be eventually placed in the middle of the sentence but the word w_{t-1} is unlikely to be the final preceding word for the editing word w_t. In other words, w_{t-1} is the wrong preceding word for w_t. To mitigate this issue, one option is to instruct the keyboard to switch to a unigram language model for decoding the editing word w_t. Because a unigram language model does not involve previously entered word w_{t-1} for decoding, the potential negative effect from the previously entered word w_{t-1} on the keyboard decoding of editing word w_t would be eliminated. Implementing this feature requires the keyboard to detect when a user starts entering the editing word. In JustCorrect-Gesture or JustCorrect-Voice, the keyboard can detect the start of entering the editing word by observing the input modality switching. In JustCorrect-Tap, a small requirement such as requiring the user to press the editing button before entering the editing word would signal the start of entering the editing word.

CONCLUSION

The key takeaway from this work is that JustCorrect nicely complements existing text correction methods and significantly enhances text editing performance for users. By reducing manual navigation operations through machine intelligence, it makes the post hoc text correction process easier for users. We solved two critical problems for enabling JustCorrect. (1) We devised the post hoc correction algorithm, which infers a user's correction intention based on the entered word. (2) We investigated which input modality was suitable for JustCorrect and found that both tap and gesture typing are appropriate for performing JustCorrect. Based on these findings, we augmented a soft keyboard with JustCorrect. Our second experiment, consisting of two studies, showed that JustCorrect Keyboard outperformed the *de facto* cursor-based editing method on the stock Android keyboard, and it also outperformed the "Type, then Correct" TTC keyboard [42] in post hoc text correction tasks. JustCorrect Keyboard reduced the correction time by 12.8% over the stock Android keyboard, and 9.7% over TTC, and was favored most. Participants were able to use JustCorrect to successfully correct errors in more than 95% of testing phrases. Overall, the results showed that JustCorrect nicely complements existing text correction methods, making text correction more automatic and intelligent than prior techniques.

ACKNOWLEDGEMENT

We thank anonymous reviewers for their insightful comments, and our user studies participants. This work was supported by NSF CHS-1815514, and NIH R01EY030085. This work was done as part of the Ph.D. dissertation of Wenzhe Cui, a Stony Brook Ph.D. student supervised by Dr. Xiaojun Bi.

REFERENCES

[1] Eneko Agirre, Daniel Cer, Mona Diab, Aitor Gonzalez-Agirre, and Weiwei Guo. 2013. *SEM 2013 shared task: Semantic Textual Similarity. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 1: Proceedings of the Main Conference and the Shared Task: Semantic Textual Similarity*. Association for Computational Linguistics, Atlanta, Georgia, USA, 32–43. https://www.aclweb.org/anthology/S13-1004

[2] Jessalyn Alvina, Carla F. Griggio, Xiaojun Bi, and Wendy E. Mackay. 2017. CommandBoard: Creating a General-Purpose Command Gesture Input Space for Soft Keyboard. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 17–28. DOI: http://dx.doi.org/10.1145/3126594.3126639

[3] Apple. 2018. About the keyboards settings on your iPhone, iPad, and iPod touch. https://support.apple.com/en-us/HT202178. (2018). [Online; accessed 22-August-2019].

[4] Ahmed Sabbir Arif, Sunjun Kim, Wolfgang Stuerzlinger, Geehyuk Lee, and Ali Mazalek. 2016. Evaluation of a Smart-Restorable Backspace Technique to Facilitate Text Entry Error Correction. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. Association for Computing Machinery, New York, NY, USA, 5151–5162. DOI: http://dx.doi.org/10.1145/2858036.2858407

[5] Microsoft Azure. 2019. Text to Speech API. (2019). https://azure.microsoft.com/en-us/services/cognitive-services/text-to-speech/ [Online; accessed 25-August-2019].

[6] Xiaojun Bi, Yang Li, and Shumin Zhai. 2013. FFitts Law: Modeling Finger Touch with Fitts' Law. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. Association for Computing Machinery, New York, NY, USA, 1363–1372. DOI: http://dx.doi.org/10.1145/2470654.2466180

[7] Xiaojun Bi, Tom Ouyang, and Shumin Zhai. 2014. Both Complete and Correct?: Multi-objective Optimization of Touchscreen Keyboard. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2297–2306. DOI: http://dx.doi.org/10.1145/2556288.2557414

[8] Wenzhe Cui, Jingjie Zheng, Blaine Lewis, Daniel Vogel, and Xiaojun Bi. 2019. HotStrokes: Word-Gesture Shortcuts on a Trackpad. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 165, 13 pages. DOI: http://dx.doi.org/10.1145/3290605.3300395

[9] Mark Davies. 2018. The corpus of contemporary American English: 1990-present. (2018).

[10] Katrin Erk. 2012. Vector space models of word meaning and phrase meaning: A survey. *Language and Linguistics Compass* 6, 10 (2012), 635–653.

[11] Vittorio Fuccella, Poika Isokoski, and Benoit Martin. 2013. Gestures and Widgets: Performance in Text Editing on Multi-touch Capable Mobile Devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 2785–2794. DOI: http://dx.doi.org/10.1145/2470654.2481385

[12] Joshua Goodman, Gina Venolia, Keith Steury, and Chauncey Parker. 2002. Language Modeling for Soft Keyboards. In *Proceedings of the 7th International Conference on Intelligent User Interfaces (IUI '02)*. ACM, New York, NY, USA, 194–195. DOI: http://dx.doi.org/10.1145/502716.502753

[13] Tovi Grossman, Pierre Dragicevic, and Ravin Balakrishnan. 2007. Strategies for Accelerating On-line Learning of Hotkeys. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 1591–1600. DOI:http://dx.doi.org/10.1145/1240624.1240865

[14] Kenneth Heafield. 2011. KenLM: Faster and Smaller Language Model Queries. In *Proceedings of the EMNLP 2011 Sixth Workshop on Statistical Machine Translation*. Edinburgh, Scotland, United Kingdom, 187–197. https://kheafield.com/papers/avenue/kenlm.pdf

[15] Christian Holz and Patrick Baudisch. 2011. Understanding Touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. Association for Computing Machinery, New York, NY, USA, 2501–2510. DOI: http://dx.doi.org/10.1145/1978942.1979308

[16] ExIdeas Inc. 2018. MessagEase - The Smartest Touch Screen keyboard. https://www.exideas.com/ME/index.php. (2018). [Online; accessed 22-August-2019].

[17] Grammarly Inc. 2020. Grammarly Keyboard. (2020). https://en.wikipedia.org/wiki/Grammarly [Online; accessed May-2020].

[18] Poika Isokoski, Benoît Martin, Paul Gandouly, and Thomas Stephanov. 2010. Motor Efficiency of Text Entry in a Combination of a Soft Keyboard and Unistrokes. In *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries (NordiCHI '10)*. ACM, New York, NY, USA, 683–686. DOI:http://dx.doi.org/10.1145/1868914.1869004

[19] Andreas Komninos, Mark Dunlop, Kyriakos Katsaris, and John Garofalakis. 2018. A Glimpse of Mobile Text Entry Errors and Corrective Behaviour in the Wild. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct (MobileHCI '18)*. ACM, New York, NY, USA, 221–228. DOI: http://dx.doi.org/10.1145/3236112.3236143

[20] Per-Ola Kristensson and Shumin Zhai. 2004. SHARK2: A Large Vocabulary Shorthand Writing System for Pen-based Computers. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*. ACM, New York, NY, USA, 43–52. DOI:http://dx.doi.org/10.1145/1029632.1029640

[21] Per Ola Kristensson and Shumin Zhai. 2007. Command Strokes with and Without Preview: Using Pen Gestures on Keyboard for Command Selection. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 1137–1146. DOI: http://dx.doi.org/10.1145/1240624.1240797

[22] Vladimir Iosifovich Levenshtein. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 10, 8 (feb 1966), 707–710. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[23] Frank Chun Yat Li, Richard T. Guy, Koji Yatani, and Khai N. Truong. 2011. The 1Line Keyboard: A QWERTY Layout in a Single Line. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 461–470. DOI: http://dx.doi.org/10.1145/2047196.2047257

[24] Google LLC. 2020. Gboard. (2020). https://en.wikipedia.org/wiki/Gboard [Online; accessed May-2020].

[25] Matt Mahoney. 2011. About Text8 file. http://mattmahoney.net/dc/textdata.html. (2011). [Online; accessed May-2020].

[26] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. (2013). http://arxiv.org/abs/1301.3781

[27] Hermann Ney, Ute Essen, and Reinhard Kneser. 1994. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language* 8, 1 (1994), 1 – 38. DOI:http://dx.doi.org/https://doi.org/10.1006/csla.1994.1001

[28] Per Ola Kristensson and Keith Vertanen. 2011. Asynchronous Multimodal Text Entry Using Speech and Gesture Keyboards.. In *Proceedings of the International Conference on Spoken Language Processing*. 581–584.

[29] Kseniia Palin, Anna Feit, Sunjun Kim, Per Ola Kristensson, and Antti Oulasvirta. 2019. How do People Type on Mobile Devices? Observations from a Study with 37,000 Volunteers.. In *Proceedings of 21st International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI'19)*. ACM.

[30] Sherry Ruan, Jacob O. Wobbrock, Kenny Liou, Andrew Ng, and James A. Landay. 2018. Comparing Speech and Keyboard Text Entry for Short Messages in Two Languages on Touchscreen Phones. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 4, Article 159 (Jan. 2018), 23 pages. DOI: http://dx.doi.org/10.1145/3161187

[31] Khe Chai Sim. 2010. Haptic Voice Recognition: Augmenting speech modality with touch events for efficient speech recognition. In *2010 IEEE Spoken Language Technology Workshop*. 73–78. DOI: http://dx.doi.org/10.1109/SLT.2010.5700825

[32] Khe Chai Sim. 2012. Speak-as-you-swipe (SAYS): A Multimodal Interface Combining Speech and Gesture Keyboard Synchronously for Continuous Mobile Text Entry. In *Proceedings of the 14th ACM International Conference on Multimodal Interaction (ICMI '12)*. ACM, New York, NY, USA, 555–560. DOI: http://dx.doi.org/10.1145/2388676.2388793

[33] Shyamli Sindhwani, Christof Lutteroth, and Gerald Weber. 2019. ReType: Quick Text Editing with Keyboard and Gaze. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 203, 13 pages. DOI:http://dx.doi.org/10.1145/3290605.3300433

[34] R. William Soukoreff and I. Scott MacKenzie. 2001. Measuring Errors in Text Entry Tasks: An Application of the Levenshtein String Distance Statistic. In *CHI '01 Extended Abstracts on Human Factors in Computing Systems (CHI EA '01)*. Association for Computing Machinery, New York, NY, USA, 319–320. DOI: http://dx.doi.org/10.1145/634067.634256

[35] Desney S. Tan, Darren Gergle, Peter Scupelli, and Randy Pausch. 2006. Physically Large Displays Improve Performance on Spatial Tasks. *ACM Trans. Comput.-Hum. Interact.* 13, 1 (March 2006), 71–99. DOI:http://dx.doi.org/10.1145/1143518.1143521

[36] Keith Vertanen, Haythem Memmi, Justin Emge, Shyam Reyal, and Per Ola Kristensson. 2015. VelociTap: Investigating Fast Mobile Text Entry Using Sentence-Based Decoding of Touchscreen Keyboard Input. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 659–668. DOI: http://dx.doi.org/10.1145/2702123.2702135

[37] Daniel Vogel and Patrick Baudisch. 2007. Shift: A Technique for Operating Pen-Based Interfaces Using Touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. Association for Computing Machinery, New York, NY, USA, 657–666. DOI: http://dx.doi.org/10.1145/1240624.1240727

[38] Robert A Wagner and Michael J Fischer. 1974. The string-to-string correction problem. *Journal of the ACM (JACM)* 21, 1 (1974), 168–173.

[39] Klaus Weidner. 2018. Hackers Keyboard. (2018). http://code.google.com/p/hackerskeyboard/ [Online; accessed 22-August-2019].

[40] Shumin Zhai and Per-Ola Kristensson. 2003. Shorthand Writing on Stylus Keyboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '03)*. Association for Computing Machinery, New York, NY, USA, 97–104. DOI: http://dx.doi.org/10.1145/642611.642630

[41] Shumin Zhai and Per Ola Kristensson. 2012. The Word-Gesture Keyboard: Reimagining Keyboard Interaction. *Commun. ACM* 55, 9 (Sept. 2012), 91–101. DOI:http://dx.doi.org/10.1145/2330667.2330689

[42] Mingrui Ray Zhang, He Wen, and Jacob O. Wobbrock. 2019. Type, Then Correct: Intelligent Text Correction Techniques for Mobile Text Entry Using Neural Networks. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 843–855. DOI: http://dx.doi.org/10.1145/3332165.3347924

[43] Mingrui Ray Zhang and O. Jacob Wobbrock. 2020. Gedit: Keyboard gestures for mobile text editing. In *Proceedings of Graphics Interface (GI '20) (GI '20)*. Canadian Information Processing Society, Toronto, Ontario, 97–104.