Yang Li · Otmar Hilliges
Editors

Artificial Intelligence for Human Computer Interaction: A Modern Approach

Springer

Editors
Yang Li
Google Research (United States)
Mountain View, CA, USA

Otmar Hilliges
Advanced Interactive Technologies Lab
ETH Zurich
Zurich, Switzerland

ISSN 1571-5035 ISSN 2524-4477 (electronic)
Human–Computer Interaction Series
ISBN 978-3-030-82680-2 ISBN 978-3-030-82681-9 (eBook)
https://doi.org/10.1007/978-3-030-82681-9

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

AI-Driven Intelligent Text Correction Techniques for Mobile Text Entry

Mingrui "Ray" Zhang, He Wen, Wenzhe Cui, Suwen Zhu, H. Andrew Schwartz, Xiaojun Bi, and Jacob O. Wobbrock

1 Introduction

Text entry techniques on touch-based mobile devices today are generally well developed. Ranging from tap-based keyboard typing to swipe-based gesture typing [64], today's mobile text entry methods employ a range of sophisticated algorithms designed to maximize speed and accuracy. Although the results reported from various papers [46, 55] show that mobile text entry can reach reasonably high speeds, some even as fast as desktop keyboards [55], the daily experience of mobile text composition is still often lacking. One bottleneck lies in the text correction process. On mobile touch-based devices, text correction often involves repetitive backspacing and moving the text cursor with repeated taps and drags over very small targets

Portions of this chapter are reproduced with permission of the ACM from the following previously published papers [10, 66].

M. "Ray" Zhang (✉) · J. O. Wobbrock
The Information School, University of Washington, Seattle, WA 98195, USA
e-mail: mingrui@uw.edu

J. O. Wobbrock
e-mail: wobbrock@uw.edu

H. Wen
The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA
e-mail: tigermored@gmail.com

W. Cui · H. Andrew Schwartz · X. Bi
Department of Computer Science, Stony Brook University, Stony Brook, NY 11794, USA
e-mail: wecui@cs.stonybrook.edu

H. Andrew Schwartz
e-mail: has@cs.stonybrook.edu

X. Bi
e-mail: xiaojun@cs.stonybrook.edu

S. Zhu
Grammarly, Inc., San Francisco, CA 94104, USA
e-mail: suwzhu@gmail.com

Y. Li and O. Hilliges (eds.), *Artificial Intelligence for Human Computer Interaction: A Modern Approach*, Human–Computer Interaction Series,
https://doi.org/10.1007/978-3-030-82681-9_5

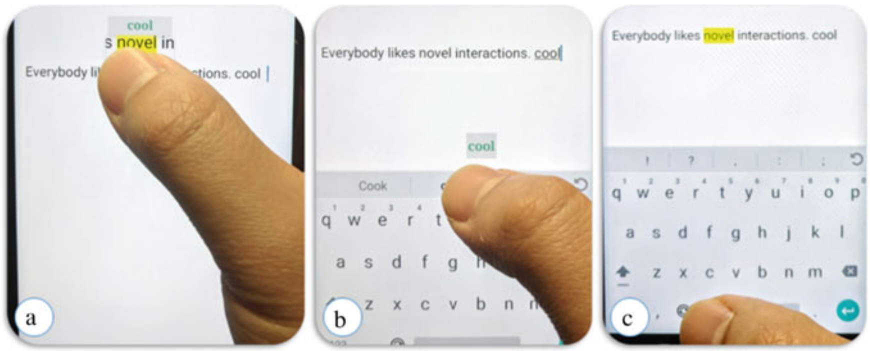

cool
s novel in
Everybody li
tions. cool
a
Everybody likes novel interactions. cool
cool
Cook
b
Everybody likes novel interactions. cool
c

Fig. 1 The three correction interactions of Type, Then Correct: **a** *Drag-n-Drop* lets the user drag the last word typed and drop it on an erroneous word or gap between words; **b** *Drag-n-Throw* lets the user drag a word from the suggestion list and flick it into the general area of the erroneous word; **c** *Magic Key* highlights each possible error word after the user types a correction. Directional dragging from atop the magic key navigates among error words, and tapping the magic key applies the correction

(i.e., the characters and spaces between them). Owing to the fat finger problem [56], this process can be slow and tedious indeed. In this chapter, we will introduce two projects that apply techniques in Natural Language Processing (NLP) to improve the text correction interaction for touch screen text entry.

Correcting text is a consistent and vital activity during text entry. A study by MacKenzie and Soukoreff showed that backspace was the second most common keystroke during text entry (pp. 164–165) [36]. Dhakal et al. [12] found that during typing, people made 2.29 error corrections per sentence, and that slow typists actually made and corrected more mistakes than the fast typists.

For immediate error corrections, i.e., when an error is noticed right after it is made, the user can press backspace to delete the error [53]. However, for overlooked error corrections, the current cursor movement-based text correction process on smartphones is laborious: one must navigate the cursor to the error position, delete the error text, re-enter the correct text, and finally navigate the cursor back. There are three ways to position the cursor: (1) by repeatedly pressing the backspace key [53]; (2) by pressing arrow keys on some keyboards or making gestures such as swipe-left; and (3) by using direct touch to move the cursor. The first two solutions are more precise than the last one, which suffers from the fat finger problem [56], but they require repetitive actions. The third option is error-prone when positioning the cursor amid small characters, which increases the possibility of cascading errors [3]; it also increases the cognitive load of the task and takes on average 4.5 s to perform the tedious position-edit-reposition sequence [18].

The two projects in this chapter are based on the same premise: What if we can skip positioning the cursor and deleting errors? Given that the *de facto* method of correcting errors relies heavily on these actions, such a question is subtly quite radical. *What if we just type the correction text, and apply it to the error?* The first project, “Type, Then Correct” (TTC) contains three interactions (Fig. 1): (1) *Drag-n-Drop* is a simple baseline technique that allows users to drag the last-typed word as a correction and drop it on the erroneous text to correct substitution and omission errors [59]. (2) *Drag-n-Throw* is the “intelligent” version of Drag-n-Drop: it allows the user to flick a word from the keyboard’s suggestion list toward the approximate area of the erroneous text. The deep learning algorithm finds the most likely error within the general target area and automatically corrects it. (3) *Magic Key* does not require direct interaction with the text input area at all. After typing a correction, the user simply presses a dedicated key on the keyboard, and the deep learning algorithm highlights possible errors according to the typed correction. The user could then dragging atop the key to navigate through the error candidates and tap the key again to apply the correction. All three of our interaction techniques require no movement of the text cursor and no use of backspace.

The second project, JustCorrect, is the evolution of the TTC project. It simplifies the concept even further by reducing the need to specify the error position. To substitute an incorrect word or insert a missing word in the sentence, the user simply types the correction at the end, and JustCorrect will automatically commit the correction without the 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.

We evaluated the two text correction projects with multiple text entry experiments and compared their performances. The results revealed that both TTC and JustCorrect resulted in faster correction times, and were preferred over the *de facto* technique.

2 Related Work

In the following subsections, we first review research related to text entry correction behaviors on touch screens. We then present current text correction techniques for mobile text entry and multi-modal text input techniques. Finally, we provide a short introduction to natural language processing (NLP) algorithms for text correction.

2.1 Text Correction Behaviors on Touch Screens

Many researchers have found that typing errors are common using touch-based keyboards and that current correction techniques are left wanting in many ways. For example, sending error-ridden messages, such as typos and errors arising from autocorrection [27], is of greatest concern when it comes to older adults. Moreover, Komninos et al. [28] observed and recorded in-the-wild text entry behaviors on Android phones, and found that users made around two word-level errors per typing session, which slowed text entry considerably. Also, participants “predominantly employed backspacing as an error correction strategy.” Based on their observations, Komninos et al. recommended that future research needed to "develop better ways for managing correction," which is the very focus of this chapter.

In most character-level text entry schemes, there are three types of text entry errors [35, 59]: substitutions, where the user enters different characters than intended; omissions, where the user fails to enter characters; and insertions, where the user injects erroneous characters. Substitutions were found to be the most frequent error among these types. In a smart watch-based text entry study [30], out of 888 phrases, participants made 179 substitution errors, 31 omission errors, and 15 insertion errors. In a big data study of keyboarding [12], substitution errors (1.65%) were observed more frequently than omission (0.80%) and insertion (0.67%) errors. Our correction techniques address substitution and omission errors; we do not address insertion errors because users can just delete insertions without typing any corrections. Moreover, insertion errors are relatively rare.

2.2 Mobile Text Correction Techniques

While much previous work focused on user behaviors during mobile text entry, there have been a few projects that improved upon the text correction process. Previous work often adopted a cursor-based editing approach. For example, previous research proposed controlling the cursor by using magnifying lens [2], pressing hard on the keyboard to turn it into a touchpad [2], or adding arrow keys [58]. Gestural operations have also been proposed to facilitate positioning the cursor. Examples included using left and right gestures [18], sliding left or right from the space key [22] to move the cursor, or using a "scroll ring" gesture along with swipes in four directions [65].

The smart-restorable backspace [4] project had the most similar goal to that of this chapter: to improve text correction without extensive backspacing and cursor positioning. The technique allowed users to perform a swipe gesture on the backspace key to delete the text back to the position of an error, and restore that text by swiping again on the backspace key after correcting the error. To determine error positions, the technique compares the edit distance of the text and the word in a dictionary. The error detection algorithm is the main limitation of that work: it only detects misspellings. It cannot detect grammar errors or word misuse. By contrast, the two projects in this chapter could detect a wide range of errors based on deep learning techniques.

Commercial products exhibit a variety of text correction techniques. Gboard [34] allows a user to touch on a word and replace it by tapping on another word in a suggestion list. However, this technique is only limited to misspellings. Some keyboards, such as the Apple iOS 9 keyboard, support indirect cursor control by treating the keyboard as a trackpad. Unfortunately, prior research [48] showed that this design brought no time or accuracy benefits compared to direct pointing. The Grammarly keyboard [23] will keep track of the input text, and provide corrections in the suggestion list. Grammarly uses NLP algorithms to provide correction suggestions, and it is able to detect both spelling and grammar errors. The user simply taps the suggestion to commit a correction. However, because Grammarly provides correction suggestions without guidance (e.g., it provides all possible error correction options without knowing which one the user wants to correct), the suggestion bar can become cluttered in the presence of many suggestions.

Different from the above techniques, the correction techniques presented in this chapter have the user enter a correction first, typed at the end of the current text input stream. Informed by the correction, the techniques can better understand what text the user wants to correct. Thus, they can not only correct "real errors" such as misspellings or grammar errors but also address other issues, such as offering to substitute synonyms.

2.3 Multi-Modal Text Input

Many soft keyboards (e.g., Gboard [34]) 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 [41, 51], using finger touch to specify the word boundaries in speech recognition [50], or using unistrokes together with key landings [25] to improve input efficiency.

JustCorrect also investigated how different input modalities affected the performance. It was particularly inspired by ReType [52], 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.

2.4 NLP Algorithms for Error Correction

The projects in this chapter use deep learning algorithms from natural language processing (NLP) to find possible errors based on typed corrections. We therefore provide a brief introduction to related techniques.

Traditional error correction algorithms utilize N-grams and edit distances to provide correction suggestions. For example, Islam and Inkpen [24] presented an algorithm that uses the Google 1T 3-gram dataset and a string-matching algorithm to detect and correct spelling errors. For each word in the original string, they first search for candidates in the dictionary, and assign each possible candidate a score derived from their frequency in the N-gram dataset and the string-matching algorithm. The candidate with the highest score above a threshold is suggested as a correction.

Recently, deep learning has gained popularity in NLP research because of its generalizability and significantly better performance than traditional algorithms. For NLP tasks, convolutional neural networks (CNN) and recurrent neural networks

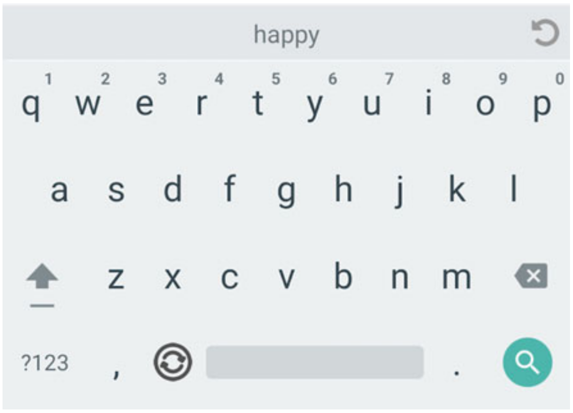


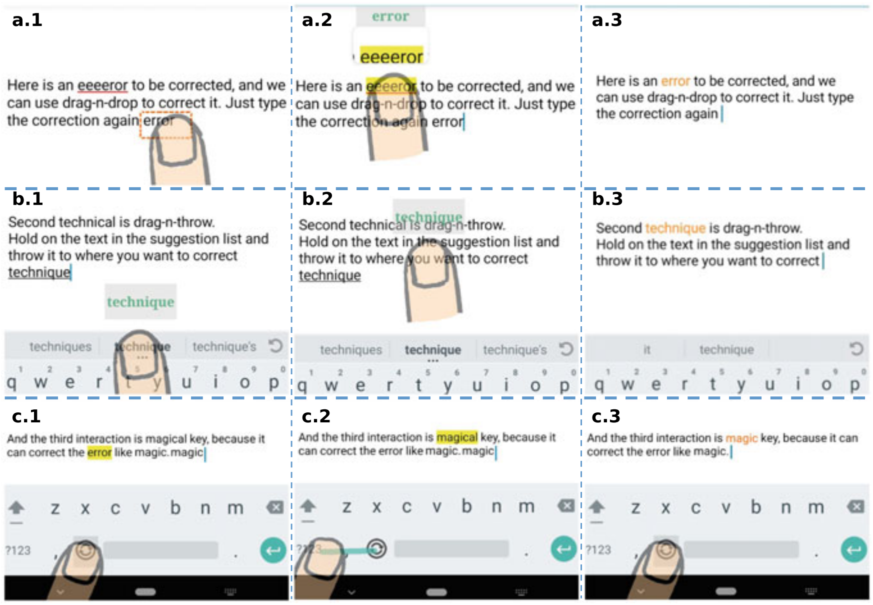
Fig. 2 Our customized keyboard interface. The undo key is located in the top-right corner. The *Magic Key* is the circular key immediately to the left of the space bar

(RNN) are extensively used. They often follow a structure called encoder–decoder, where part of the model encodes the input text into a feature vector, then decodes the vector into the result. In TTC, we utilize an RNN in this encoder–decoder pattern. A thorough explanation of these methods is beyond the current scope. Interested readers are directed to prior work [6, 54, 61].

Most researchers treat the error correction task as a language translation task in deep learning because their input and output are both sentences—for error correction, the input is a sentence with errors and the output is an error-free sentence; for language translation, the input is a sentence in one language and the output is a sentence in another language. For example, Xie et al. [60] presented an encoder–decoder RNN correction model that operates input and output at the character level. Their model was built upon a sequence-to-sequence model for translation [6], which was also used in the algorithm of the TTC project for error detection.

3 Type, Then Correct: The Three Interactions

We present the design and implementation of the three interaction techniques of Type, Then Correct (TTC). The common features of these interactions are: (1) the first step is always to type the correction text at the current cursor position, usually the end of the current input stream; (2) all correction interactions can be undone by tapping the undo key on the keyboard (Fig. 2, top right); (3) after a correction is applied, the text cursor remains at the last character of the text input stream, allowing the user to continue typing without having to move the cursor. A current, but not theoretical, limitation is that we only allow the correction text to be contiguous alphanumeric text without special characters or spaces.



a.1 Here is an eeeeror to be corrected, and we can use drag-n-drop to correct it. Just type the correction again error

a.2 Here is an eeeeror to be corrected, and we can use drag-n-drop to correct it. Just type the correction again error

a.3 Here is an error to be corrected, and we can use drag-n-drop to correct it. Just type the correction again

b.1 Second technical is drag-n-throw. Hold on the text in the suggestion list and throw it to where you want to correct technique

b.2 Second technical is drag-n-throw. Hold on the text in the suggestion list and throw it to where you want to correct technique

b.3 Second technique is drag-n-throw. Hold on the text in the suggestion list and throw it to where you want to correct

c.1 And the third interaction is magical key, because it can correct the error like magic. magic

c.2 And the third interaction is magical key, because it can correct the error like magic. magic

c.3 And the third interaction is magic key, because it can correct the error like magic.

Fig. 3 The three interaction techniques. *Drag-n-Drop*: **a.1** Type a word and then touch it to initiate correction; **a.2** Drag the correction to the error position. Touched words are highlighted and magnified, and the correction shows above the magnifier; **a.3** Drop the correction on the error to finish. *Drag-n-Throw*: **b.1** Dwell on a word from the suggestion list to initiate correction. The word will display above the finger; **b.2** Flick the finger toward the area of the error: here, the flick ended on "the," not the error text "technical"; **b.3** The algorithm determines the error successfully, and confirming animation appears. *Magic Key*: **c.1** Tap the magic key (the circular button) to trigger correction. Here, "error" is shown as the nearest potential error. **c.2** Drag left from atop the magic key to highlight the next possible error in that direction. Now, "magical" is highlighted. **c.3** Tap the magic key again to commit the correction "magic"

3.1 *Drag-n-Drop*

Drag-n-Drop is the simplest interaction technique. With *Drag-n-Drop*, after typing the correction, the user then drags the correction text and drops it on the error location. As shown in Fig. 3a.1, if the finger's touchdown point is within the area of the last word, the correction procedure will be initiated. The user can then move the correction and drop it either on another word to substitute it, or on a space to insert the correction.

While moving the correction, a magnifier appears above the finger to provide an enlarged image of the touched text; text to be corrected will be highlighted in a yellow background (Fig. 3a.2). When the finger drags over an alphanumeric character, we highlight its surrounding text bounded by any special character or space. When the finger drags over a space character, we highlight the single space character. The correction text also displays above the magnifier during the drag to remind the user what the correction is.

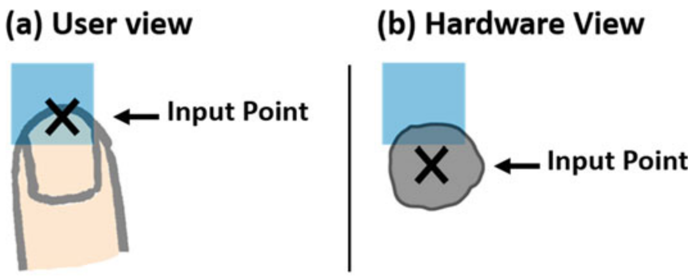


Fig. 4 Perceived input point: **a** the user views the top of the fingernail as the input point [21]; **b** but today’s hardware regards the center of the contact area as the touch input point, which is not the same. Figure adapted from [56]

Similar to Shift by Vogel and Baudisch [56], we adjusted the input point to 30 pixels above the actual contact point, to reflect the user’s perceived input point [21]. Vogel and Baudisch suggested that “users perceived the selection point of the finger as being located near the top of the finger tip” [7, 56], while the actual touch point was roughly at the center of the finger contact area [49], as shown in Fig. 4. After the correction is dropped on a space (for insertion) or on a word (for substitution), there is an animated color change from orange to black, confirming the successful application of the correction text.

3.2 *Drag-n-Throw*

Similar to *Drag-n-Drop*, *Drag-n-Throw* also requires the user to drag the correction. But unlike *Drag-n-Drop*, with *Drag-n-Throw*, the user flicks the correction from the word suggestion list atop the keyboard, not from the text area, allowing the user’s fingers to stay near the keyboard area. As before, the correction text shows above the touch point as a reminder (Fig. 3b.1). Instead of dropping the correction on the error position, the user throws (i.e., flicks) the correction to the general area of the text to be corrected. Once the correction is thrown, our deep learning algorithm determines the error position, and corrects the error either by substituting the correction for a word, or by inserting the correction. Color animation is displayed to confirm the correction. The procedure is shown in Fig. 3b.1–3.

We enable the user to drag the correction from the suggestion list because it is quicker and more accurate than directly interacting with the text, which has smaller targets. Moreover, our approach provides more options and saves time because of the word-completion function. For example, if the user wants to type “dictionary,” she can just type “dic” and “dictionary” appears in the list. Or, if the user misspells “dictonary,” omitting an “i,” the correct word still appears in the list because of the keyboard’s decoding algorithm.

Throwing the text speeds up the interaction beyond precise drag-and-dropping. The user does not have to carefully move the finger to drop the correction. Our deep learning algorithm will output candidate positions for substitution or insertion within the general finger-up area. (More implementation details are explained below.) In our implementation, if the candidate is within 250 pixels of the finger-lift point in any direction, the error will be corrected and confirmed by color animation. Otherwise, there will be no effect. The 250-pixel threshold was derived empirically from iterative trial-and-error. Larger thresholds allow corrections too far away from the finger-lift point, which can cause unexpected results and user frustration. Smaller thresholds reduce the benefits of "throwing" and eventually start to feel like "dropping."

3.3 Magic Key

Drag-n-Drop required interaction within the text input area; Drag-n-Throw kept the fingers closer to the keyboard but still required some interaction in the text input area. With Magic Key, the progression "inward" toward the keyboard is fulfilled, as the fingers do not interact with the text input area at all, never leaving the keyboard. Thus, round trips [15] between the keyboard and text input area are eliminated.

With Magic Key, after typing the correction, the user taps the magic key on the keyboard (Fig. 3c.1), and the possible error text is highlighted. If a space is highlighted, an insertion is suggested; if a word is highlighted, a substitution is suggested. The nearest possible error to the just-typed correction will be highlighted first; if it is not the desired correction, the user can drag from atop the magic key to left to show the next possible error. The user can drag left or right from atop the magic key to rapidly navigate among different error candidates. Finally, the user can tap the magic key to commit the correction. The procedure is shown in Fig. 3c.1–3. To cancel the operation, the user can simply tap any key (other than undo or the magic key itself).

4 Type, Then Correct: The Correction Algorithm

In this section, we present the deep learning algorithm for text correction and its natural language processing (NLP) model, the data collection and processing procedures, and the training process and validation results.

4.1 Expected Correction Categories

We first list error types that our model should correct:

Typos A typographical error ("typo") happens when a few characters of a word are mistyped. For example, *fliwer (flower)* or *feetball (football)*. Among typos, misspellings can usually be auto-corrected by current keyboards; however, auto-correction might yield another wrong word. For example, *best (bear)* or *right (tight)*. Our model should be able to handle different types of typo errors.

Grammar Errors Grammar errors caused by one mistaken word should be corrected, such as misuse of verb tense, lack of articles or pronouns, subject–verb disagreement, etc.

Semantic Substitution Our model should also be able to substitute words that are semantically related to the correction, such as synonyms and antonyms. For example, "what a nice day" can be corrected to "what a beautiful day." Semantic substitution is not necessarily correcting an error, but is useful when the user wants to change the expression.

4.2 The Deep Neural Network Structure

Inspired by Xie et al. [60], we applied a recurrent neural network (RNN) encoder–decoder model similar to the translation task for text corrections. The encoder contains a character-level convolutional neural network (CNN) [26] and two bi-directional gated recurrent unit (GRU) layers [9]. The decoder contains a word-embedding layer and two GRU layers. The overall flow of the model is shown in Fig. 5, and the encoder–decoder structure is shown in Fig. 6.[1]

Traditional recurrent neural networks (RNN) cannot output positional information. Our key insight is that instead of outputting the whole error-free sentence, we make the decoder only output five words around the proposed correction position, e.g., the correction word and its four neighboring words (two before, two after). If there are not enough words, the decoder will output the flags <*bos*> or <*eos*> instead for beginning-of-sentence and end-of-sentence, respectively. To locate the correction position, we compare the output with the input sentence word-by-word, and choose the position that aligns with most words. For the example in Fig. 5, we first tokenize the input and add two <*bos*> and two <*eos*> to the start and end of the tokens. Then we compare the output with the input:

```
Input: <bos> <bos> thanks the reply <eos> <eos>
CS:          <bos> thanks for the   reply
CI:          <bos> thanks the reply
```

Above, "CS" means compare for substitution, which finds the best alignment for substitution (it uses all five words of the output trying to align with the input); "CI" means compare for insertion, which finds the best alignment for insertion (it only uses the first and last two words of the output for alignment, as the center word is the insertion correction). In the example, CI has best alignment of four tokens (<*bos*>,

[1] The model and data processing codes are available at https://github.com/DrustZ/CorrectionRNN.

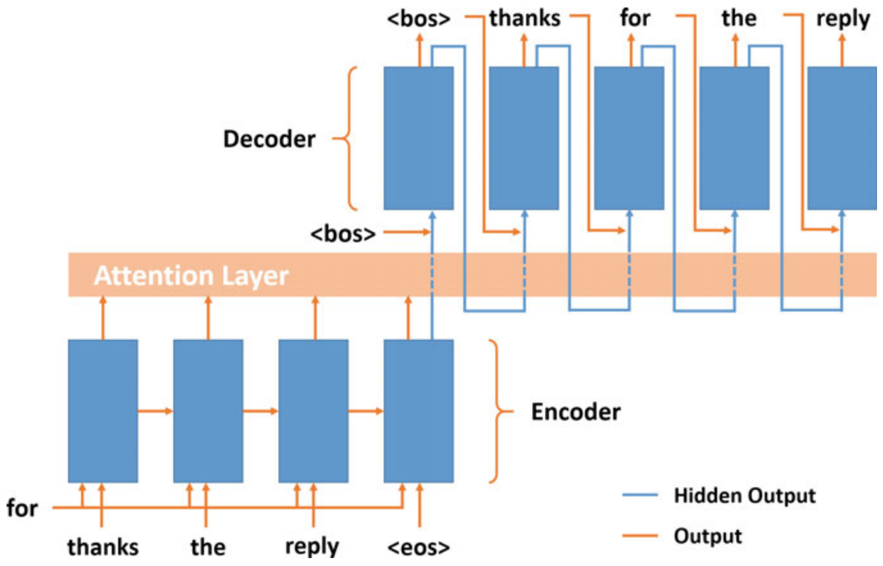


Fig. 5 The encoder–decoder model for text correction. The model outputs five words in which the middle word is the correction. In this way, we get the correction's location

thanks, the, reply), thus "for" will be inserted between "thanks" and "the." If the number of aligned tokens is the same in both comparisons, we would use insertion in our implementation.

We now explain the details of the encoder and the decoder (Fig. 6). For the encoder, because there might be typos and rare words in the input, operating on the character level is more robust and generalizable than operating on the word level. We first apply the character-level CNN [26] composed of *Character Embedding, Multiple Conv. Layers* and *Max-over-time Pool layers* (Fig. 6, left). Our character-level CNN generates an embedding for each word at the character level. The character embedding layer converts the characters of a word into a vector of $L \times E_c$ dimensions. We set E_c to 15, and fixed L to 18 in our implementation, which means the longest word can contain 18 characters (longer words are discarded). Words with fewer than 18 characters are appended with zeroes in the input vector. We then apply multiple convolution layers on the vector. After convolution, we apply max-pooling to obtain a fixed-dimensional (E_w) representation of the word. In our implementation, we used convolution filters with width [1, 2, 3, 4, 5] of size [15, 30, 50, 50, 55], yielding a fixed vector with the size of 200. E_c was set to 200 in the decoder.

We also needed to provide the correction information for the encoder. We achieved this by feeding the correction into the same character-level CNN, and concatenated the correction embedding with the embedding of the current word. This yielded a vector of size $2E_w$, which was then fed into two bi-directional GRU layers. The hidden size H of GRU was set to 300 in encoder and decoder.

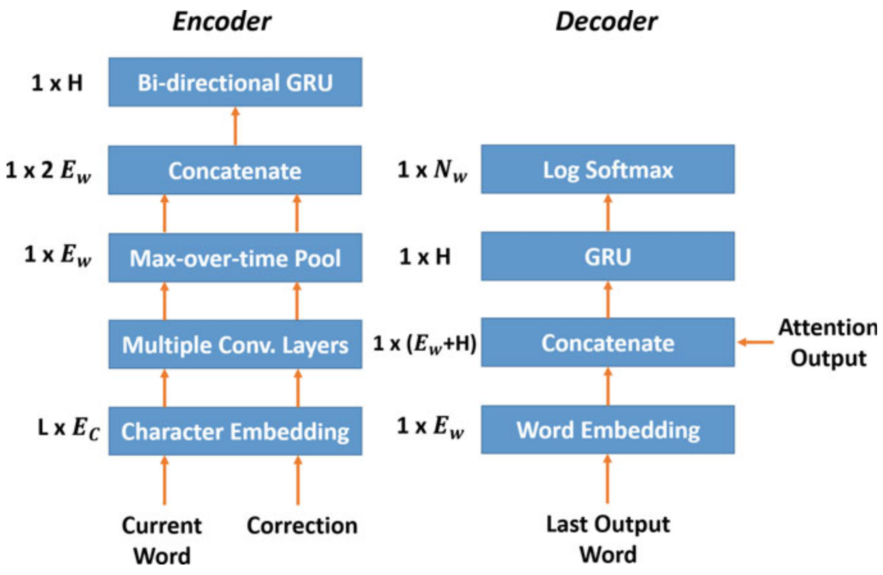


Fig. 6 Illustration of the encoder and decoder, which is every vertical blue box in Fig. 5. L is the length of characters in a word; E_c is the character embedding size; H is the hidden size; E_w is the word embedding size; N_w is the word dictionary size

The decoder first embedded the word in a vector of size E_w, which was set to 200. Then it was concatenated with the attention output. We used the same attention mechanism as Bahdanau et al. [6]. Two GRU layers and a log-softmax layer then followed to output the predicted word.

4.3 Data Collection and Processing

We used the *CoNLL 2014 Shared Task* [40] and its extension dataset [6] as a part of the training data. The data contained sentences from essays written by English learners with correction and error annotations. We extracted the errors that were either insertion or substitution errors. In all, we gathered over 33,000 sentences for training.

To gather even more training data, we perturbed several large datasets containing normal text. We used the Yelp reviews (containing two million samples) and part of the Amazon reviews dataset (containing 130,000 samples) generated by Zhang et al. [67]. We treated these review data as if they were error-free texts, and applied artificial perturbation to them. Specifically, we applied four perturbation methods:

1. **Typo simulation**. In order to simulate a real typo, we applied the simulation method similar to Fowler et al. [16]. The simulation treated the touch point dis-

tribution on a QWERTY layout as a 2-D Gaussian spatial model, and selected a character based on the sampling coordinates. We used the empirical parameters of the spatial model from Zhu et al. [68]. For each sentence in the review dataset, we randomly chose a word from the sentence, and simulated the typing procedure for each character of the word until the simulated word was different from the original word. We then applied a spellchecker to "recover" the typo. This maneuver was to simulate the error of auto-correction functions, where the "corrected" word actually becomes a different word. We then used the "recovered" word as a typo if it was different from the original word, or used the typing simulation result if the spellchecker successfully recovered the typo.

2. **Dropping words** To enable the model to learn about insertion corrections, we randomly dropped a word from a sentence, and labeled the dropped word as the correction. We prioritized dropping common stop words first if any of them appeared in the sentence, because people were most likely to omit words like a, the, my, in, and very.
3. **Word deformation** We randomly changed or removed a few characters from a word. If the word was a verb, we would replace it with a word sharing the same lexeme. For example, we would pick one of broken, breaking, breaks, or broke to replace break. If the word was a noun, we would use a different singular or plural word. For example, we would replace star with stars. Otherwise, we would just remove a few characters from the word.
4. **Semantic word substitution** This perturbation enabled the model to learn semantic information. For a given word in the sentence, we looked for words that were semantically similar to it and made a substitution. We used the GloVe [44] Twitter-100 model from Gensim [45] to represent similarity. Synonyms and antonyms were generated using this method.

For each sentence in the review dataset, we randomly applied a perturbation method. We then combined the perturbed data with the CoNLL data, and filtered out sentences containing less than 3 words or more than 20 words. In all, the final training set contained 5.6 million phrases.

For testing, we used two datasets: *CoNLL 2013 Shared Task* [39], which was also a grammatical error correction dataset, and the Wikipedia revision dataset [62], which contains real-word spelling errors mined from Wikipedia's revision history. We generated 1665 phrases from the CoNLL 2013 dataset and 1595 phrases from the Wikipedia dataset.

4.4 Training Process

We implemented our model in PyTorch [43]. We only included lowercase alphabetical letters (*a*–*z*) and ten numerals (0–9) in the character vocabulary of the encoder. We used the Adam optimizer with a learning rate of 0.0001 (1e-4) for the encoder and 0.0005 (5e-4) for the decoder, and a batch size of 128. We applied weight clipping of [−10, +10], and a teacher forcing ratio of 0.5. We also used dropout with probability 0.2 in all GRU layers. For the word embedding layer in the decoder, we labeled words with frequencies less than 2 in the training set as <*unk*> (unknown).

4.5 Results

Table 1 shows the evaluation results on the two testing datasets. The recall is 1 because all our testing data contained errors. We regarded a prediction as correct if the error position predicted was correct using the comparison algorithm described above.

4.6 Other Implementation Details

We developed a custom Android keyboard and a notebook application to implement our three text correction interaction techniques. Our keyboard was based on the Android Open-Source Project (AOSP)[2] from Google. In building on top of this keyboard, we added the long-press interaction on suggested words for *Drag-n-Throw*.

The notebook application was built on an open-source project Notepad,[3] and most of the interactions were implemented as part of the notebook application. For *Drag-n-Drop*, when a user touched within the last word area (within 100 pixels of the *(x, y)*-coordinate of the last character), the interaction was initiated. We used the default magnifier on the Android system and added a transparent view showing the correction above the finger as it moves.

For *Drag-n-Drop* and *Magic Key*, the keyboard needed to communicate with the notebook application. The keyboard used the Android Broadcast mechanism to send the correction and endpoint of the throw gesture of *Drag-n-Throw*. When the information was received, the notebook would search within the three lines near the release point. For each line, the notebook extracted up to 60 surrounding characters near the release point, and sent them to a server running the correction model. The server then replied with possible correction options and the corresponding probabilities. The notebook then selected the most likely option to update the correction. To avoid

Table 1 The performance of our correction model on the two testing datasets

Dataset	Accuracy (%)
CoNLL 2013	75.68
Wikipedia revisions	81.88

[2] https://android.googlesource.com/platform/packages/inputmethods/LatinIME/.

[3] https://github.com/farmerbb/Notepad.

the correction happening too far away from the throwing endpoint, we constrained the x-coordinate of the correction to be within 250 pixels of the finger-lift endpoint.

For the *Magic Key* technique, the keyboard notified the notebook when the magic key was pressed or dragged. The notebook would treat the last word typed as the correction, and sent the last 1000 characters to the server. The server then split the text into groups of 60 characters with overlaps of 30 characters, and predicted a correction for each group. When the notebook received the prediction results, it first highlighted the nearest option, and then switched to further error options when the key was dragged left. For substitution corrections, it would highlight the whole word to be substituted; for insertion corrections, it would highlight the space where the correction was to be inserted.

The server running the correction model handled responses via HTTP requests. To increase the accuracy of the model for typos, we first calculated the matching score between each token of the input text and the correction using the Levenshtein algorithm [31]. The score equaled the number of matches divided by the total character number of the two words. If the score of a word in the sentence was above 0.75, we treated the word as the error to be corrected. Otherwise, we fed the text and correction into the aforementioned neural network model.

5 Type, Then Correct: Experiment

We evaluated three aspects of our correction techniques: (1) timing and efficiency; (2) success rate of *Drag-n-Throw* and *Magic Key*; and (3) users' subjective preferences. We conducted an experiment containing two tasks: a correction task and a composition task. The correction task purely evaluated the efficiency of the interactions, and the composition task evaluated the usability and success rate of the intelligent techniques in more realistic scenarios.

5.1 Participants

We recruited 20 participants (8 male, 12 female, aged 23–52) for the study. We used emails, social media, and word-of-mouth for recruitment. All participants were familiar with entering and correcting text on mobile devices. The experiment lasted 1 hour, and participants were compensated $20 USD for their time.

5.2 Apparatus

A Google Pixel 2 XL was used for the study. The phone had a 6.0" screen with a 1440–2880 resolution. We added logging functions from the notebook application to record correction time. The server running the correction model had a single GTX 1080.

5.3 Phrases Used in the Correction Task

Both tasks utilized a within-subjects study design. For the correction task, we chose 30 phrases from the test dataset on which the correction model had been 100% correct because we wanted purely to evaluate the performance of the interaction technique, not of the predictive model. We split the phrases evenly into three categories: *typos, word changes,* and *insertions*. *Typos* required replacement of a few characters in a word; *word changes* required replacing a whole word in a phrase; and *insertions* required inserting a correction. For each category, we had five *near-error* phrases where the error positions were within the last three words; and five *far-error* phrases where the error positions were farther away. The reason was to see whether error positions would affect correction efficiency. Examples of phrases in each category are provided in the appendix.

5.4 Procedure

Participants were first introduced to our different interaction techniques, including the categories of errors that *Drag-n-Throw* and *Magic Key* were able to correct. Then participants practiced the three techniques with three practice phrases each. After practicing, the 30 phrases as well as their corresponding corrections were presented to the participants. Then they began to correct the phrases using four techniques: (1) today's *de facto* cursor-positioning and backspace-based method, (2) *Drag-n-Drop*, (3) *Drag-n-Throw*, and (4) *Magic Key*. The order of the four techniques was counterbalanced using a balanced Latin Square.

When the correction task started, the participant would be shown the next phrase to be corrected on a desktop computer screen, as well as how to correct it. The notebook application would display the phrase with an error. After the participant corrected the error, a dialog box appeared asking the participants to enter the next phrase. The experimenter then showed the next phrase on the computer. When the participant was ready, she entered the next phrase by tapping the OK button. The interface is shown in Fig. 7. The rationale for showing how to correct the phrase on a computer screen was to filter out the learning effect and visual search time caused by the unfamiliarity of the phrases, and to isolate the interaction time.

After the correction task, the participants started composing messages freely. They were told to type for 3 minutes as they would in normal messaging situations. However, they were told not to correct any errors during typing. After finishing their compositions, they then corrected all errors with the four interaction techniques. This composition task endeavored to evaluate usability in a more realistic scenario.

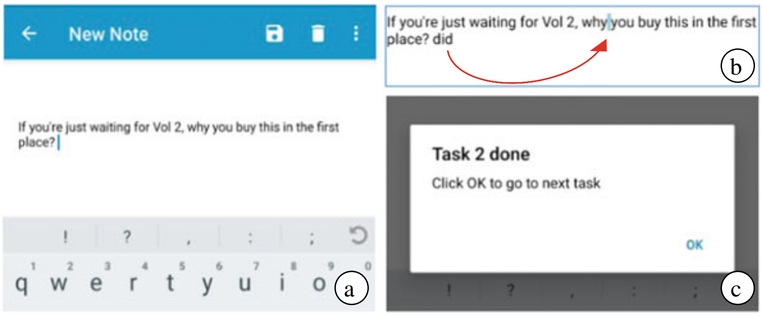

New Note
If you're just waiting for Vol 2, why you buy this in the first place?
If you're just waiting for Vol 2, why you buy this in the first place? did
Task 2 done
Click OK to go to next task
OK
a
b
c

Fig. 7 **a** The notebook application showing the test phrase. **b** The intended correction displayed on the computer screen. **c** After each correction, a dialog box appeared

When participants were correcting errors with *Drag-n-Throw* and *Magic Key*, the experimenter recorded whether any failure happened in order to calculate the error rate.

When the two tasks ended, participants filled out a NASA-TLX survey [47] and a usability survey adapted from the SUS questionnaire [8] for each interaction.

6 Type, Then Correct: Results

For the correction task, 2400 phrases were collected in total. For the correction task, we focus on task completion times; for the composition task, we focus on the success rate of the two intelligent interaction techniques and users' preferences.

6.1 Correction Time

Figure 8 shows correction times for the four techniques. In addition to overall times, the correction times for *near-error* and *far-error* phrases are also shown. We log-transformed correction times to comply with the assumption of conditional normality, as is often done with time measures [32]. We used linear mixed model analyses of variance [17, 33], finding that there was no order effect on correction time ($F(3, 57) = 1.48, n.s.$), confirming that our counter-balancing worked. Furthermore, technique had a significant effect on correction time for all phrases ($F(3, 57) = 26.49$, $p < 0.01$), *near-error* phrases ($F(3, 57) = 29.02$, $p < 0.01$), and *far-error* phrases ($F(3, 57) = 17.04$, $p < 0.01$), permitting us to investigate *post hoc* pairwise comparisons.

We performed six *post hoc* paired-sample t-tests with Holm's sequential Bonferroni procedure [20] to correct for Type I error rates, finding that for all phrases,

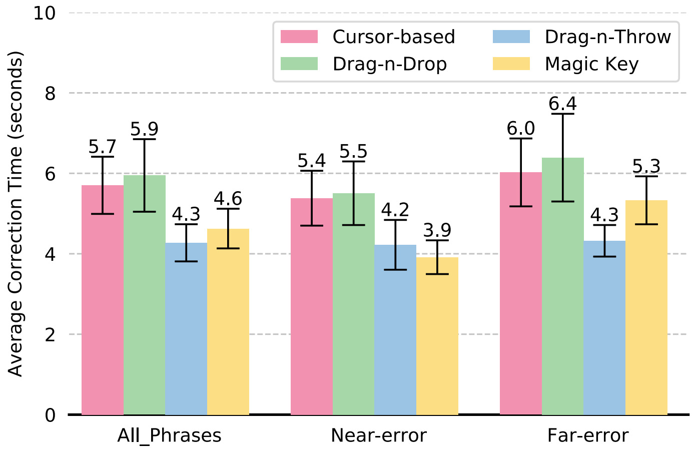


Fig. 8 Average correction times in seconds for different interaction techniques (lower is better). *Drag-n-Throw* was the fastest for all phrases and far-error phrases, while *Magic Key* was the fastest for near-error phrases. Error bars are +1 SD

the *de facto* cursor-based method was significantly slower than *Drag-n-Throw* ($t(19) = 6.66$, $p < 0.01$) and *Magic Key* ($t(19) = 4.79$, $p < 0.01$); *Drag-n-Drop* was also significantly slower than *Drag-n-Throw* ($t(19) = 7.49$, $p < 0.01$) and *Magic Key* ($t(19) = 5.62$, $p < 0.01$). For near-error phrases, the *de facto* method was significantly slower than *Drag-n-Throw* ($t(19) = 5.58$, $p < 0.01$) and *Magic Key* ($t(19) = 7.02$, $p < 0.01$); *Drag-n-Drop* was also significantly slower than *Drag-n-Throw* ($t(19) = 5.00$, $p < 0.01$) and *Magic Key* ($t(19) = 7.44$, $p < 0.01$). For far-error phrases, *Drag-n-Throw* was significantly faster than all other interactions: the *de facto* method ($t(19) = -5.64$, $p < 0.01$), *Drag-n-Drop* ($t(19) = -6.60$, $p < 0.01$), and *Magic Key* ($t(19) = -3.68$, $p < 0.01$).

We then looked at different correction types. Figure 9 shows the average correction times for *typos*, *word changes*, and *insertions*. Again, we used linear mixed model analyses of variance [17, 33] on log correction time [32]. Technique had a statistically significant effect for all correction types: *typo* ($F(3, 57) = 5.11$, $p < 0.01$), *word change* ($F(3, 57) = 10.87$, $p < 0.01$), and *insertion* ($F(3, 57) = 55.55$, $p < 0.01$).

We then performed *post hoc* paired-sample t-tests with Holm’s sequential Bonferroni procedure, finding that for *typos*, the *de facto* cursor-based method was significantly slower than *Drag-n-Throw* ($t(19) = 3.80$, $p < 0.01$); *Drag-n-Drop* was also significantly slower than *Drag-n-Throw* ($t(19) = 2.70$, $p < 0.05$). For *word change*, the *de facto* cursor-based method was significantly slower than all other techniques: *Drag-n-Drop* ($t(19) = 3.54$, $p < 0.01$), *Drag-n-Throw* ($t(19) = 5.58$, $p < 0.01$), and *Magic Key* ($t(19) = 3.74$, $p < 0.01$). For *insertion*, *Drag-n-Drop* was significantly slower than all other interactions: the *de facto* method ($t(19) = 5.72$, $p < 0.01$), *Drag-n-Throw* ($t(19) = 11.17$, $p < 0.01$), and *Magic Key* ($t(19) = 10.92$, $p <$

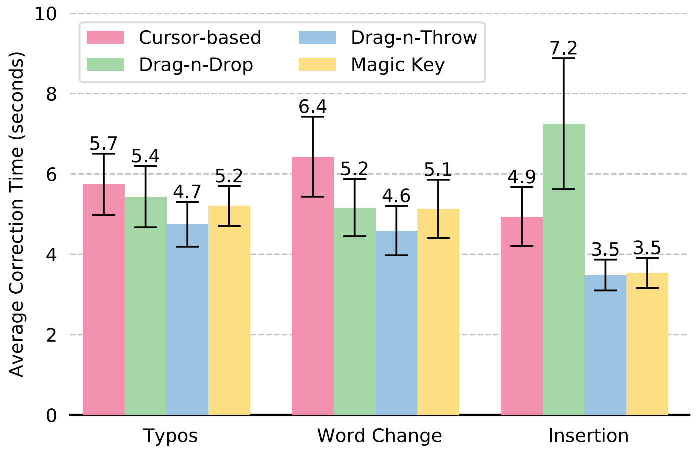


Fig. 9 Average correction times in seconds for different correction types (lower is better). *Drag-n-Throw* was the fastest for all three types. Error bars are +1 SD

0.01); also, the *de facto* cursor-based method was significantly slower than *Drag-n-Throw* ($t(19) = 5.45$, $p < 0.01$) and *Magic Key* ($t(19) = 5.20$, $p < 0.01$).

6.2 Success Rate

In the text composition task, we recorded errors when participants were using *Drag-n-Throw* and *Magic Key*. With *Drag-n-Throw*, participants made 108 errors in all, and 95 of them were successfully corrected, a success rate of 87.9%. Among the successfully corrected errors, nine were attempted more than once because the corrections were not applied to expected error positions. With *Magic Key*, participants made 101 errors in all, and 98 of them were successfully corrected, a success rate of 97.0%.

6.3 Subjective Preference

The composite scores of the SUS usability [8] and TLX [47] surveys for different interaction techniques are shown in Fig. 10. Participants generally enjoyed using *Magic Key* and *Drag-n-Throw* more than the *de facto* cursor-based method and *Drag-n-Drop*. Also, the two deep learning techniques were perceived to have lower workload than the other two.

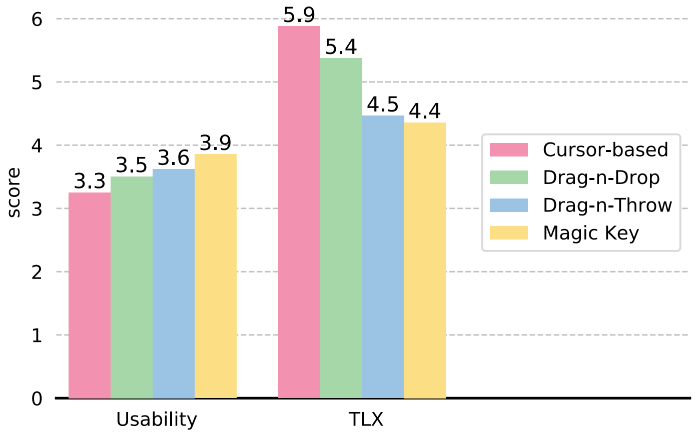


Fig. 10 Composite usability (higher is better) and NASA-TLX (lower is better) scores for different techniques. *Magic Key* was rated as the most usable and having the lowest workload

7 JustCorrect: Simplifying the Text Correction Based on TTC

In this section, we present the second project, JustCorrect, which is an extension of the Type, Then Correct (TTC) project, and simplifies the correction interaction one step further. The interaction flow is demonstrated in Fig. 11. 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 mis-spelling: *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.

7.1 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. The post hoc correction algorithm takes

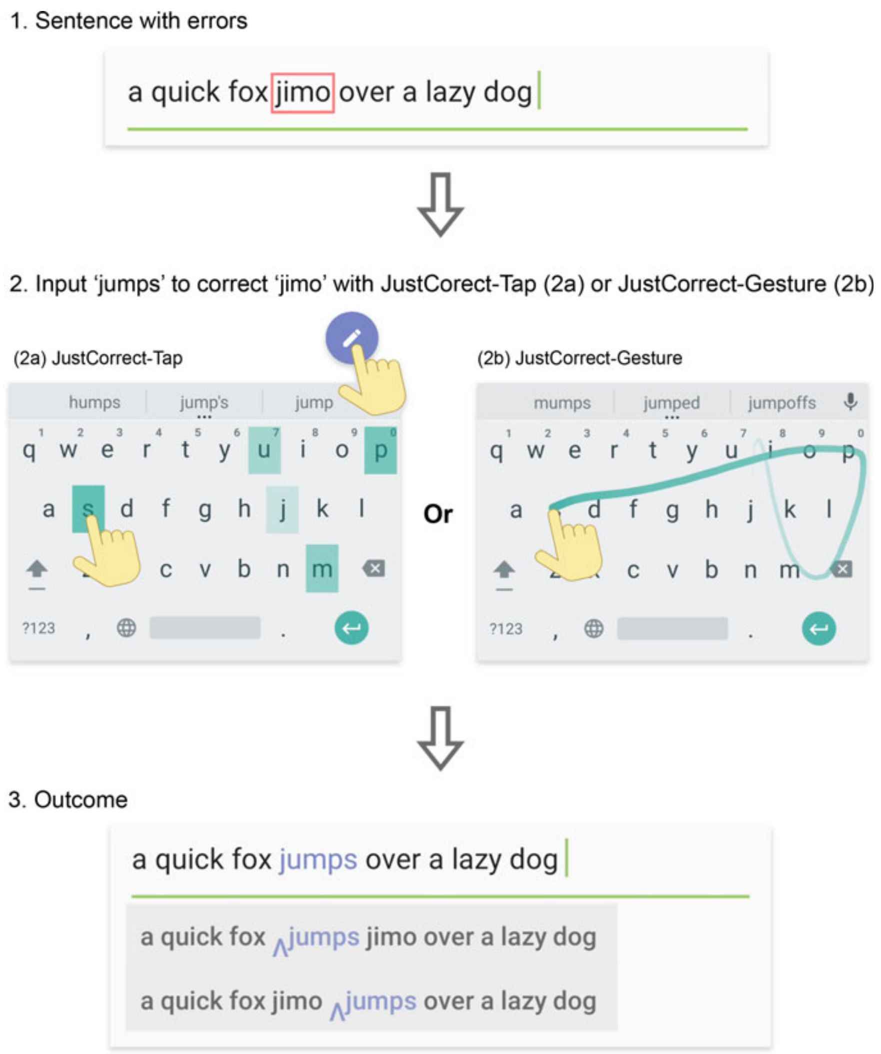

1. Sentence with errors
a quick fox jimo over a lazy dog
2. Input 'jumps' to correct 'jimo' with JustCorect-Tap (2a) or JustCorrect-Gesture (2b)
(2a) JustCorrect-Tap
humps
jump's
jump
Or
(2b) JustCorrect-Gesture
mumps
jumped
jumpoffs
?123
3. Outcome
a quick fox jumps over a lazy dog
a quick fox ^jumps jimo over a lazy dog
a quick fox jimo ^jumps over a lazy dog

Fig. 11 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

Table 2 An example of eight 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 ith 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 Sentence channels, Edit Distance, and Word Embedding, respectively

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

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 eight words, there are eight substitution and nine insertion possibilities: *_a_quick_fox_jimo_over_a_lazy_dog_*. The nine possible insertion positions are indicated by the underscores. The post hoc correction algorithm then generates eight substitution candidates ($S_1 - S_8$), as shown in Table 2, and nine insertion candidates ($I_1 - I_9$) as shown in Table 3.

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.

Table 3 An example of nine 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 ith insertion location. $InserScore_i$ is insertion score for ranking insertion candidates

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

7.2 Substitution Score

The substitution score reflects how likely a substitution candidate represents the user's actual editing intention. 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 ith substitution candidate S_i, its substitution score $SubScore_i$ is defined as

$$SubScore_i = ES_i + WS_i + SS_i, \tag{1}$$

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 [57].

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 the overall coherence of the word choice or replacement within its context.

7.2.1 Edit Distance Channel

The edit distance channel calculates the editing similarity for each substitution candidate. The Levenshtein edit distance [31] between two strings is the minimum number of single-character edits including deletions, insertions, or substitutions needed to transform one string into another string. The editing similarity ES_i is defined as

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

where w^* is the correction and w_i is the ith word in the previous text. $max(|w^*|, |w_i|)$ denotes the max length of w^* and w_i.

7.2.2 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 this channel, words from the vocabulary are mapped to vectors derived from statistics on the co-occurrence of words within documents [13]. The distance between two vectors can then be used as a measure of syntactic and semantic difference [1].

We trained our word embedding model over the "Text8" dataset [37] using the Word2Vec skip-gram approach [38]. The cosine similarity $WS_C(w^*, w_i)$ is then calculated as the WS_i [1]. WS_i was normalized in the range [0, 1].

7.2.3 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 certain 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 [19]. The language model takes each substitution candidate sentence 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:

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

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) [11] (2012 to 2017), which contains over 500 million words.

7.3 *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 candidate I_i is calculated as

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

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 $(I_1, I_2, \ldots, I_{N+1})$. $InserScore_i$ is also normalized in [0,1].

7.4 Combining Substitution and Insertion Candidates

The post hoc correction algorithm combines the substitution and insertion candidates to generate correction suggestions by calculating each candidate's scores. We compare substitution and insertion candidates by their sentence channel scores because it is the common component in both score calculations. The candidates with the highest top-3 scores would be shown on the interface. If all three candidates are of the same error kind (substitution/insertion), we change the last candidate with the top one of the other kind to ensure the diversity of the suggestions. The top suggestion will be automatically committed to the text (see Sect. 7, Part 3).

8 JustCorrect: Experiment

We evaluated the usability of three forms of JustCorrect: JustCorrect-Gesture, JustCorrect-Tap, and JustCorrect-Voice. These variations are different JustCorrect techniques with different input modalities, as explained below.

8.1 Participants

We recruited 16 participants (4 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.

8.2 Apparatus

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

8.3 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.
- *JustCorrect-Tap*. After entering a word with tap typing, the user presses the editing button to invoke the post-hoc correction algorithm (see Fig. 11).
  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 Fig. 11). 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 [29, 63, 64]. 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.

8.4 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 mobile typing dataset of Palin et al. [42]. This dataset included actual input errors from 37, 370 users when

Table 4 Examples of phrases in the experiment. The first three sentences contained substitution errors. The last sentence contained an omission error

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

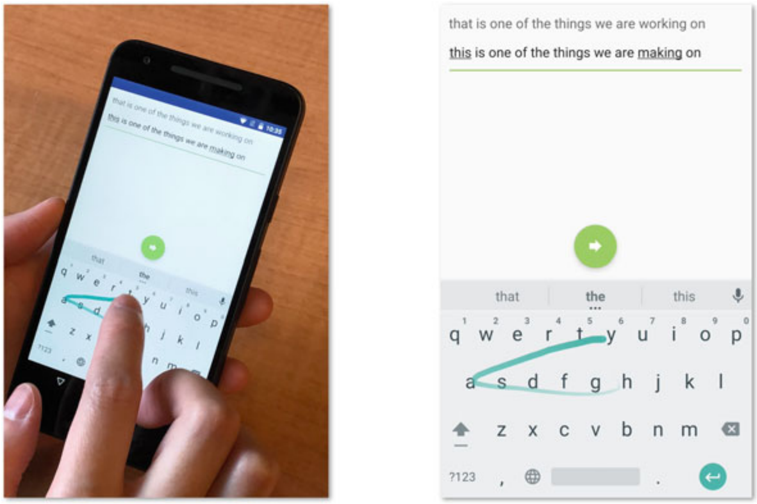
typing with smartphones and their target sentences. We focused on omission and substitution errors since the post-hoc correction algorithm was designed to handle these two types of errors. We also filtered out sentences with punctuation or number errors because our focus was on word correction. Among 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 4 shows four examples of phrases in experiment.

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 Fig. 12. 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 a 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 3-minute break. The order of the four conditions was counterbalanced using a balanced Latin Square.

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


that is one of the things we are working on

this is one of the things we are making on

Fig. 12 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

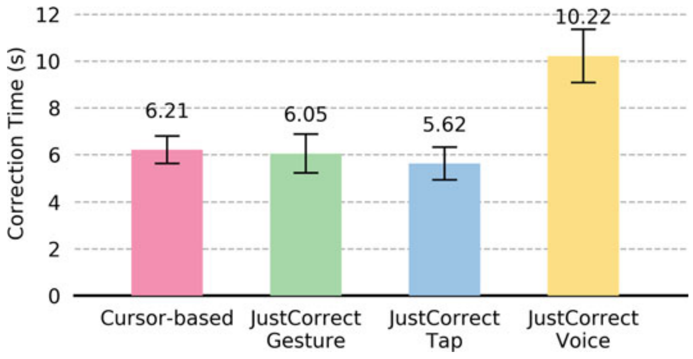


Fig. 13 Mean (95% CI) text correction times for each method for successful trials

8.5 *Results*

8.5.1 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 13 shows text correction time for trials that were successfully corrected using the designated editing method in each condition (unsuccessful trials are

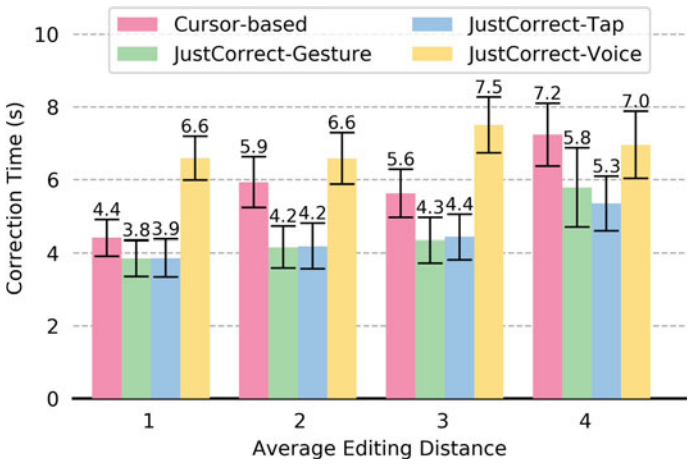


Fig. 14 Mean (95% CI) text correction times for the tasks successfully completed on the first attempt

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

To understand the effectiveness of the algorithm under different conditions, we analyzed cases that 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 Fig. 14. 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.

8.5.2 Success Rate

We define the success rate as the percentage of correct trials out of all trials for a given correction technique. Figure 15 shows success rates across conditions. The mean ± 95% CI for success rate for each input technique was: 100.0 ± 0% for the *de facto* cursor-based technique, 96.2 ± 2.2% for JustCorrect-Gesture, 97.1 ± 0.03% for JustCorrect-Tap, and 95.1 ± 0.03% for JustCorrect-Voice. A repeated measures

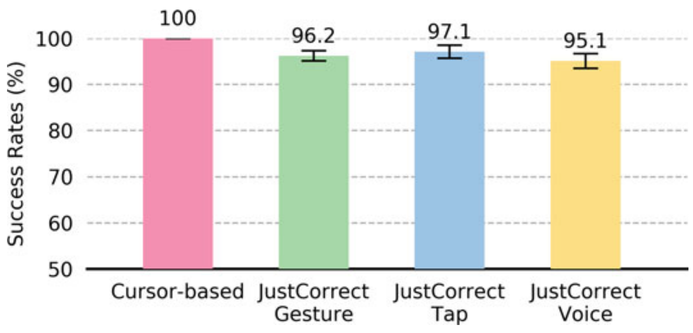


Fig. 15 Success rate by input technique. While not 100%, the three interactions of JustCorrect achieved pretty close success rate to the cursor-based interaction

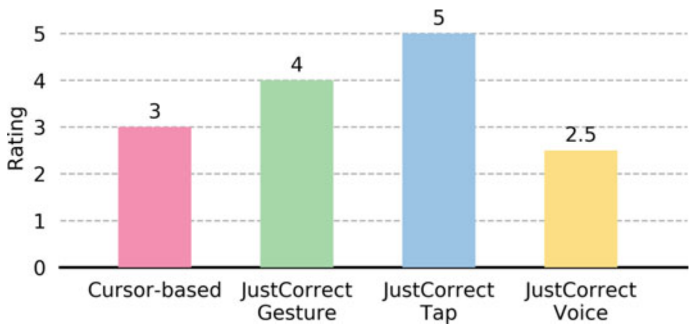


Fig. 16 The median preference rating for cursor-based correction, JustCorrect-Gesture, JustCorrect-Tap, and JustCorrect-Voice. JustCorrect Tap received the highest rating

ANOVA showed that text editing technique had a significant effect on the overall success rate ($F_{3,45} = 14.31$, $p < 0.001$). Pairwise comparisons with Bonferroni correction showed the difference was significant between JustCorrect-Tap versus cursor-based, JustCorrect-Gesture versus cursor-based, JustCorrect-Voice versus cursor-based ($p < 0.01$). All other pairwise comparisons were not statistically significant.

8.5.3 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 shown in Fig. 16, the median rating for cursor-based editing, JustCorrect-Gesture, JustCorrect-Tap, and JustCorrect-Voice were 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.

8.6 Discussion

We introduced two projects in this chapter: Type, Then Correct (TTC), and JustCorrect. We first discuss the user study results of TTC, followed by the discussion of JustCorrect.

In TTC, *Drag-n-Throw* performed fastest among different correction types. Moreover, its performance was not affected by whether the error was far away or not (Fig. 8). *Magic Key* also achieved reasonable speeds across different correction types. For *near-errors* within the last three words, it even surpassed *Drag-n-Throw*, because the errors would be highlighted and corrected with just two taps. For *far-errors*, participants had to drag atop the *Magic Key* a few times to highlight the desired error, leading to longer correction times. *Drag-n-Drop* performed the slowest over all phrases, which was mainly caused by the insertion corrections. As shown in Fig. 9, it was faster than the de facto cursor-based method for typos and word changes, but significantly slower than other interactions for insertions. To insert a correction between two words, a user had to highlight the narrow space between those words. Many participants spent a lot of time adjusting their fingers in order to highlight the desired space. They also had to redo the correction if they accidentally made a substitution instead of an insertion. Our undo key proved to be vital in such cases. To evaluate the performance of our algorithm in more realistic scenarios, we analyzed the results from text composition tasks. *Drag-n-Throw* achieved a success rate of 87.9%. A failure was when two possible error candidates were too close to each other. For example, if the user wanted to insert "the" in the phrase "I left keys in room," there were two possible positions (before keys and before room), but only one of them would be corrected. *Magic Key* achieved a higher success rate of 97.0%, as it searched every possible error in the text.

As for participants' subjective preferences, 12 of 20 participants liked *Magic Key* the most. The major reason was convenience: all the actions were done on the keyboard. P1 commented, "Just one button handles everything. I don't need to touch the text anymore. It was also super intelligent. I am lazy, and that's why I enjoyed it so much." Another reason was that *Magic Key* provided feedback (highlights) before committing the correction, making the user confident about the target of their actions. As P4 pointed out, "It provides multiple choices, and the uncertain feeling is gone." The main critique of *Magic Key* was about the dragging interaction required to move among error candidates. P5 commented: "If the text is too long and the error is far away, I have to drag a lot to highlight the error. Also, the button is kinda small, and hard to drag."

Interestingly, we found that all three participants above age 40 had positive feedback about the two intelligent correction techniques, and negative feedback about the *de facto* cursor-based method. P14, aged 52, commented, "I dislike the cursor-based method most. I have a big finger, and it is hard to tap the text precisely. Throw is easy and works great. I also like *Magic Key*, because I don't need to interact with the text." Older adults are known to perform touch screen interactions more slowly and with less precision than younger adults [14], and the intelligent correction techniques might benefit them by removing the requirement of precise touch. Moreover, people walking on the street or holding the phone with one hand might also benefit from the interactions, because touching precisely is difficult in such situations.

In JustCorrect, 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 with large editing distances relative to the target sentences. As shown in Fig. 14, 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 hand, in JustCorrect-Gesture, gesture typing is used to correct text only, limiting its scope of usage.

Third, contrary to the promising performance of JustCorrect-Gesture and JustCorrect-Tap, JustCorrect-Voice under-performed. The reason was that JustCorrect required a user to first enter the editing word. However, 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.

There is an exciting point in both projects: employing the power of machine learning to automate the text correction and realize interactions that were not possible before. The advantage of deep learning is that longer context can be incorporated in the language models than the traditional n-gram-based methods, which enables the models to "understand" the intention of the user on a deeper level.

8.7 Future Work

On the basis of our work here, we propose four possible future directions: (1) Punctuation handling: currently both TTC and JustCorrect do not handle punctuation, so errors like “lets” (let’s) currently cannot be corrected. (2) Adding better feedback mechanics to reduce the uncertainty of the outcome: although the interactions were intelligent and did the work right most of the time, they were not transparent to the user, and the outcome of the interactions was not obvious. For example, participants felt unconfident when using the *Drag-n-Throw*, as there was a lack of feedback as to where the corrections would be applied. Adding proper feedback, such as highlighting the surrounding text of the throwing position to provide cues about where the correction will occur. (3) Multilingual correction support: the two interaction techniques could be applied to other languages as well, such as the Chinese language. (5) Interactions beyond keyboard correction: the concept can also be applied to other correction scenarios, such as voice input and handwriting.

9 Conclusion

In this chapter, we demonstrated how artificial intelligence could be applied to the text correction interaction on touch screens. The first project, Type, Then Correct (TTC), includes three novel interaction techniques with one common concept: to type the correction and apply it to the error, without needing to reposition the text cursor or use backspace, which break the typing flow and slow touch-based text entry. The second project, JustCorrect, brought the concept further by removing the need to manually specify the error position. Both projects utilized machine learning algorithms in NLP fields to help identify the possible error text. The user studies showed that both TTC and JustCorrect were significantly faster than *de facto* cursor-based correction methods and garnered more positive user feedback. They provide examples of how, by breaking from the desktop paradigm of arrow keys, backspacing, and mouse-based cursor positioning, we can rethink text entry on mobile touch devices and develop novel methods better suited to this paradigm.

References

1. Agirre E, Cer D, Diab M, Gonzalez-Agirre A, Guo W (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, pp 32–43. https://www.aclweb.org/anthology/S13-1004
2. Apple (2018) About the keyboards settings on your iphone, ipad, and ipod touch. https://support.apple.com/en-us/HT202178. Accessed 22 Aug 2019

3. Arif AS, Sturzlinger W (2013) Pseudo-pressure detection and its use in predictive text entry on touchscreens. In: Proceedings of the 25th australian computer-human interaction conference: augmentation, application, innovation, collaboration, Association for Computing Machinery, New York, NY, USA, OzCHI '13, p 383–392. https://doi.org/10.1145/2541016.2541024
4. Arif AS, Kim S, Stuerzlinger W, Lee G, Mazalek A (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, Association for Computing Machinery, New York, NY, USA, CHI '16, pp 5151–5162. https://doi.org/10.1145/2858036.2858407
5. Azure M (2019) Text to speech api. https://azure.microsoft.com/en-us/services/cognitive-services/text-to-speech/. Accessed 25 Aug 2019
6. Bahdanau D, Cho K, Bengio Y (2016) Neural machine translation by jointly learning to align and translate. 1409.0473
7. Benko H, Wilson AD, Baudisch P (2006) Precise selection techniques for multi-touch screens. In: Proceedings of the SIGCHI conference on human factors in computing systems, Association for Computing Machinery, New York, NY, USA, CHI '06, pp 1263–1272. https://doi.org/10.1145/1124772.1124963
8. Brooke J (2013) Sus: a retrospective. J Usability Studies 8(2):29–40
9. Cho K, van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using RNN encoder–decoder for statistical machine translation. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), Association for Computational Linguistics, Doha, Qatar, pp 1724–1734. https://doi.org/10.3115/v1/D14-1179. https://www.aclweb.org/anthology/D14-1179
10. Cui W, Zhu S, Zhang MR, Schwartz A, Wobbrock JO, Bi X (2020) Justcorrect: Intelligent post hoc text correction techniques on smartphones. In: Proceedings of the 33rd annual ACM symposium on user interface software and technology, Association for Computing Machinery, New York, NY, USA, UIST '20, pp 487–499. https://doi.org/10.1145/3379337.3415857
11. Davies M (2018) The corpus of contemporary American english: 1990-present
12. Dhakal V, Feit AM, Kristensson PO, Oulasvirta A (2018) Observations on Typing from 136 Million Keystrokes, Association for Computing Machinery, New York, NY, USA, pp 1–12. https://doi.org/10.1145/3173574.3174220
13. Erk K (2012) Vector space models of word meaning and phrase meaning: a survey. Lang Ling Compass 6(10):635–653
14. Findlater L, Froehlich JE, Fattal K, Wobbrock JO, Dastyar T (2013) Age-related differences in performance with touchscreens compared to traditional mouse input. In: Proceedings of the SIGCHI conference on human factors in computing systems, Association for Computing Machinery, New York, NY, USA, CHI '13, pp 343–346. https://doi.org/10.1145/2470654.2470703
15. Fitzmaurice G, Khan A, Pieké R, Buxton B, Kurtenbach G (2003) Tracking menus. In: Proceedings of the 16th Annual ACM symposium on user interface software and technology, Association for Computing Machinery, New York, NY, USA, UIST '03, pp. 71–79. https://doi.org/10.1145/964696.964704
16. Fowler A, Partridge K, Chelba C, Bi X, Ouyang T, Zhai S (2015) 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, Association for Computing Machinery, New York, NY, USA, CHI '15, pp. 649–658. https://doi.org/10.1145/2702123.2702503
17. Frederick BN (1999) Fixed-, random-, and mixed-effects anova models: a user-friendly guide for increasing the generalizability of anova results. Advances in social science methodology, Stamford. JAI Press, CT, pp 111–122
18. Fuccella V, Isokoski P, Martin B (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, ACM, New York, NY, USA, CHI '13, pp 2785–2794. https://doi.org/10.1145/2470654.2481385, http://doi.acm.org/10.1145/2470654.2481385

19. Heafield K (2011) KenLM: faster and smaller language model queries. In: Proceedings of the EMNLP 2011 sixth workshop on statistical machine translation, Edinburgh, Scotland, United Kingdom, pp 187–197. https://kheafield.com/papers/avenue/kenlm.pdf
20. Holm S (1979) A simple sequentially rejective multiple test procedure. Scand J Stat 6(2):65–70. http://www.jstor.org/stable/4615733
21. Holz C, Baudisch P (2011) Understanding touch. In: Proceedings of the SIGCHI conference on human factors in computing systems, Association for Computing Machinery, New York, NY, USA, CHI '11, pp 2501–2510. https://doi.org/10.1145/1978942.1979308
22. Inc E (2018) Messagease - the smartest touch screen keyboard. https://www.exideas.com/ME/index.php. Accessed 22 Aug 2019
23. Inc G (2020) Grammarly keyboard. https://en.wikipedia.org/wiki/Grammarly. Accessed May 2020
24. Islam A, Inkpen D (2009) Real-word spelling correction using google web it 3-grams. In: Proceedings of the 2009 conference on empirical methods in natural language processing: Volume 3 - Volume 3, Association for Computational Linguistics, USA, EMNLP '09, pp 1241–1249
25. Isokoski P, Martin B, Gandouly P, Stephanov T (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, ACM, New York, NY, USA, NordiCHI '10, pp 683–686. https://doi.org/10.1145/1868914.1869004. http://doi.acm.org/10.1145/1868914.1869004
26. Kim Y, Jernite Y, Sontag D, Rush AM (2016) Character-aware neural language models. In: Proceedings of the Thirtieth AAAI conference on artificial intelligence, AAAI Press, AAAI'16, pp 2741–2749
27. Komninos A, Nicol E, Dunlop MD (2015) Designed with older adults to supportbetter error correction in smartphone text entry: the maxiekeyboard. In: Proceedings of the 17th international conference on human-computer interaction with mobile devices and services adjunct, Association for Computing Machinery, New York, NY, USA, MobileHCI '15, pp 797–802. https://doi.org/10.1145/2786567.2793703
28. Komninos A, Dunlop M, Katsaris K, Garofalakis J (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, Association for Computing Machinery, New York, NY, USA, MobileHCI '18, pp 221–228. https://doi.org/10.1145/3236112.3236143
29. Kristensson PO, Zhai S (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, ACM, New York, NY, USA, UIST '04, pp 43–52. https://doi.org/10.1145/1029632.1029640. http://doi.acm.org/10.1145/1029632.1029640
30. Leiva LA, Sahami A, Catala A, Henze N, Schmidt A (2015) Text entry on tiny qwerty soft keyboards. In: Proceedings of the 33rd annual ACM conference on human factors in computing systems, Association for Computing Machinery, New York, NY, USA, CHI '15, pp 669–678. https://doi.org/10.1145/2702123.2702388
31. Levenshtein VI (1965) Binary codes capable of correcting deletions, insertions, and reversals. Soviet Phys Doklady 10:707–710
32. Limpert E, Stahel WA, Abbt M (2001) Log-normal distributions across the sciences: keys and clues: on the charms of statistics, and how mechanical models resembling gambling machines offer a link to a handy way to characterize log-normal distributions, which can provide deeper insight into variability and probability–normal or log-normal: that is the question. BioScience 51(5):341–352. https://doi.org/10.1641/0006-3568(2001)051[0341:LNDATS]2.0.CO;2. https://academic.oup.com/bioscience/article-pdf/51/5/341/26891292/51-5-341.pdf
33. Littell R, Henry P, Ammerman C (1998) Statistical analysis of repeated measures data using sas procedures. J Animal Sci 76(4):1216–1231. https://doi.org/10.2527/1998.7641216x
34. LLC G (2020) Gboard. URLhttps://en.wikipedia.org/wiki/Gboard. Accessed May 2020

35. MacKenzie IS, Soukoreff RW (2002) A character-level error analysis technique for evaluating text entry methods. In: Proceedings of the second nordic conference on human-computer interaction, Association for Computing Machinery, New York, NY, USA, NordiCHI ’02, pp 243–246. https://doi.org/10.1145/572020.572056
36. MacKenzie IS, Soukoreff RW (2002) Text entry for mobile computing: Models and methods,theory and practice. Hum-Comput Int 17(2-3):147–198. https://doi.org/10.1080/07370024.2002.9667313. https://www.tandfonline.com/doi/abs/10.1080/07370024.2002.9667313
37. Mahoney M (2011) About text8 file. http://mattmahoney.net/dc/textdata.html. Accessed May 2020
38. Mikolov T, Chen K, Corrado GS, Dean J (2013) Efficient estimation of word representations in vector space. arXiv:1301.3781
39. Ng HT, Wu SM, Wu Y, Hadiwinoto C, Tetreault J (2013) The CoNLL-2013 shared task on grammatical error correction. In: Proceedings of the seventeenth conference on computational natural language learning: shared task, Association for Computational Linguistics, Sofia, Bulgaria, pp 1–12. https://www.aclweb.org/anthology/W13-3601
40. Ng HT, Wu SM, Briscoe T, Hadiwinoto C, Susanto RH, Bryant C (2014) The CoNLL-2014 shared task on grammatical error correction. In: Proceedings of the eighteenth conference on computational natural language learning: shared task, Association for Computational Linguistics, Baltimore, Maryland, pp 1–14. https://doi.org/10.3115/v1/W14-1701. https://www.aclweb.org/anthology/W14-1701
41. Ola Kristensson P, Vertanen K (2011) Asynchronous multimodal text entry using speech and gesture keyboards. In: Proceedings of the international conference on spoken language processing, pp 581–584
42. Palin K, Feit A, Kim S, Kristensson PO, Oulasvirta A (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
43. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) Pytorch: an imperative style, high-performance deep learning library. In: Wallach H, Larochelle H, Beygelzimer A, d’ Alché-Buc F, Fox E, Garnett R (eds) Advances in neural information processing systems 32, Curran Associates, Inc., pp 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
44. Pennington J, Socher R, Manning C (2014) GloVe: global vectors for word representation. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), Association for Computational Linguistics, Doha, Qatar, pp 1532–1543. https://doi.org/10.3115/v1/D14-1162. https://www.aclweb.org/anthology/D14-1162
45. Řehůřek R, Sojka P (2010) Software framework for topic modelling with large corpora. In: Proceedings of the LREC 2010 workshop on new challenges for NLP frameworks, ELRA, Valletta, Malta, pp 45–50. http://is.muni.cz/publication/884893/en
46. Ruan S, Wobbrock JO, Liou K, Ng A, Landay JA (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). https://doi.org/10.1145/3161187
47. Rubio S, Díaz EM, Martín J, Puente J (2004) Evaluation of subjective mental workload: a comparison of swat, nasa-tlx, and workload profile methods. Appl Psychol 53:61–86
48. Schmidt D, Block F, Gellersen H (2009) A comparison of direct and indirect multi-touch input for large surfaces. In: Gross T, Gulliksen J, Kotzé P, Oestreicher L, Palanque P, Prates RO, Winckler M (eds) Human-computer interaction - INTERACT 2009. Springer, Berlin, pp 582–594
49. Sears A, Shneiderman B (1991) High precision touchscreens: design strategies and comparisons with a mouse. Int J Man Mach Stud 34:593–613

50. Sim KC (2010) Haptic voice recognition: Augmenting speech modality with touch events for efficient speech recognition. In: 2010 IEEE spoken language technology workshop, pp 73–78. https://doi.org/10.1109/SLT.2010.5700825
51. Sim KC (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, ACM, New York, NY, USA, ICMI '12, pp 555–560. https://doi.org/10.1145/2388676.2388793. http://doi.acm.org/10.1145/2388676.2388793
52. Sindhwani S, Lutteroth C, Weber G (2019) Retype: Quick text editing with keyboard and gaze. In: Proceedings of the 2019 CHI conference on human factors in computing systems, ACM, New York, NY, USA, CHI '19, pp 203:1–203:13. https://doi.org/10.1145/3290605.3300433. http://doi.acm.org/10.1145/3290605.3300433
53. Soukoreff RW, MacKenzie IS (2004) Recent developments in text-entry error rate measurement. In: CHI '04 extended abstracts on human factors in computing systems, Association for Computing Machinery, New York, NY, USA, CHI EA '04, pp 1425–1428. https://doi.org/10.1145/985921.986081
54. Sutskever I, Vinyals O, Le QV (2014) Sequence to sequence learning with neural networks. In: Proceedings of the 27th international conference on neural information processing systems - Volume 2, MIT Press, Cambridge, MA, USA, NIPS'14, pp 3104–3112
55. Vertanen K, Memmi H, Emge J, Reyal S, Kristensson PO (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, Association for Computing Machinery, New York, NY, USA, CHI '15, pp 659–668. https://doi.org/10.1145/2702123.2702135
56. Vogel D, Baudisch P (2007) Shift: a technique for operating pen-based interfaces using touch. In: Proceedings of the SIGCHI conference on human factors in computing systems, Association for Computing Machinery, New York, NY, USA, CHI '07, pp 657–666. https://doi.org/10.1145/1240624.1240727
57. Wagner RA, Fischer MJ (1974) The string-to-string correction problem. J ACM (JACM) 21(1):168–173
58. Weidner K (2018) Hackers keyboard. http://code.google.com/p/hackerskeyboard/. Accessed 22 Aug 2019
59. Wobbrock JO, Myers BA (2006) Analyzing the input stream for character- level errors in unconstrained text entry evaluations. ACM Trans Comput-Hum Interact 13(4):458–489. https://doi.org/10.1145/1188816.1188819
60. Xie Z, Avati A, Arivazhagan N, Jurafsky D, Ng A (2016) Neural language correction with character-based attention. ArXiv:1603.09727
61. Young T, Hazarika D, Poria S, Cambria E (2018) Recent trends in deep learning based natural language processing [review article]. IEEE Comput Intell Mag 13:55–75
62. Zesch T (2012) Measuring contextual fitness using error contexts extracted from the Wikipedia revision history. In: Proceedings of the 13th conference of the European chapter of the association for computational linguistics, Association for Computational Linguistics, Avignon, France, pp 529–538. https://www.aclweb.org/anthology/E12-1054
63. Zhai S, Kristensson PO (2003) Shorthand writing on stylus keyboard. In: Proceedings of the SIGCHI conference on human factors in computing systems, Association for Computing Machinery, New York, NY, USA, CHI '03, pp 97–104. https://doi.org/10.1145/642611.642630
64. Zhai S, Kristensson PO (2012) The word-gesture keyboard: reimagining keyboard interaction. Commun ACM 55(9):91–101. https://doi.org/10.1145/2330667.2330689
65. Zhang MR, Wobbrock OJ (2020) Gedit: keyboard gestures for mobile text editing. In: Proceedings of graphics interface (GI '20), Canadian information processing society, Toronto, Ontario, GI '20, pp 97–104
66. Zhang MR, Wen H, Wobbrock JO (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, Association for Computing Machinery, New York, NY, USA, UIST '19, pp 843–855. https://doi.org/10.1145/3332165.3347924

67. Zhang X, Zhao J, LeCun Y (2015) Character-level convolutional networks for text classification. In: Proceedings of the 28th international conference on neural information processing systems - Volume 1, MIT Press, Cambridge, MA, USA, NIPS'15, pp 649–657
68. Zhu S, Luo T, Bi X, Zhai S (2018) Typing on an invisible keyboard. In: Proceedings of the 2018 CHI Conference on human factors in computing systems, Association for Computing Machinery, New York, NY, USA, CHI '18, pp 1–13. https://doi.org/10.1145/3173574.3174013