IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 6 June 2023, accepted 5 July 2023, date of publication 13 July 2023, date of current version 20 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3295004

RESEARCH ARTICLE

How to Progressively Build Thai Spelling Correction Systems?

ANURUTH LERTPIYA[1], TAWUNRAT CHALOTHORN[1], AND PAKPOOM BUABTHONG[2]
[1]Innovation Research and Development, Kasikorn Labs, Kasikorn Business-Technology Group, Pak Kret 11120, Thailand
[2]Faculty of Science and Technology, Nakhon Ratchasima Rajabhat University, Nakhon Ratchasima 30000, Thailand

Corresponding author: Tawunrat Chalothorn (tawunrat.c@kbtg.tech)

ABSTRACT Neural-based sequence-to-sequence methods (Seq2Seq) have proven to be highly effective for Context-sensitive Thai spelling correction. However, they also inherit the drawbacks of Seq2Seq, such as a fixed vocabulary and large data requirements. However, dictionary-based methods and their typical applications are insufficiently robust to produce corrections with reduced error rates. These drawbacks inhibit the application of these methods in a broader range of use cases. In this paper, we provide a practical guide on how to build correction systems progressively and efficiently with three main contributions. First, we present a process for efficiently and progressively producing training data for both neural-based and dictionary-based methods. Our annotation process enables existing methods to be trained with only two percent of the data hand annotated. Second, we propose the Extendable Neural Contextual Corrector (XNCC), a novel text correction approach that decouples the dictionary from the neural model. This enables the dictionary to be extended post-training. Finally, we compare text correction systems with various configurations to demonstrate how these systems can be effectively used to produce corrections. Our experiments show that 1) minor changes to dictionary-based methods can significantly improve correction performance, 2) neural-based correction systems can be trained using a fraction of the data, and 3) XNCC can have the dictionary extended to generalize to new data without re-training. Lastly, we provide recommendations for progressively building text correction systems at multiple levels of implementation effort based on our findings.

INDEX TERMS Natural language processing, machine learning, artificial neural networks, text generation, spelling correction, text normalization, Thai language.

I. INTRODUCTION

With the ubiquitousness of the Internet and smart devices, text-based communication has increased to an unprecedented scale. Accurate spelling correction systems have become essential for businesses when conducting critical written communications (e.g., Emails, customer support chats, and social media presences). In addition, proper application of spelling correction on user-generated social text has been shown to improve accuracy in downstream Natural Language Processing tasks [1]. Nevertheless, the development of such systems remains difficult. The complexity of the Thai language, with its ambiguous word boundaries and multiple valid alternative words for minor character-level modifications, poses challenges for accurate spelling correction. Existing dictionary-based methods lack the necessary robustness without human assistance [2]. On the other hand, state-of-the-art approaches like Seq2Seq models rely on expensive human-annotated corpora of erroneous and corrected text.

A significant hurdle in spelling correction is handling out-of-vocabulary (OOV) tokens [2], [3], [4]. While dictionary-based methods can be easily extended by the user, Seq2Seq models are limited by the initial vocabulary. As a result, special structures are employed to enable correctors to produce OOV tokens. However, these techniques only allow the model to leave out OOV tokens and not produce corrections outside the initial vocabulary.

This brings us to the more general issue that different use-cases for correctors will have different target

The associate editor coordinating the review of this manuscript and approving it for publication was Long Xu.

vocabularies. Despite the existence of official Thai dictionaries [5], [6], they are rarely used in isolation by dict-based correctors since new words, borrowed words, slangs, and domain-specific words are a significant part of written communication. Off-the-self correctors often use dictionaries built from the target text corpus [7] or add external dictionaries[1] to alleviate this issue. Although these general-purpose dictionaries are far from perfect, they can be extended to fit the needs of the users.

In this study, we demonstrate how to build text correction systems progressively and effectively. First, we introduce our data annotation pipeline that efficiently produces data that is applicable to a variety of existing text correction methods. Our approach to annotation allows us to produce both data for both dictionary-based and neural-based text correction systems. Second, we introduce Extendable Neural Contextual Corrector (XNCC), a neural-based text correction method that can be extended with new vocabulary post-training. Our corrector decouples the internal dictionary from the neural text embeddings, allowing extensions to the dictionary during inference time like traditional dictionary-based methods while producing correction based on context provided by the surrounding text. Finally, we outline steps implementors can take to progressively build effective text corrector at different total effort and resources.

This paper is structured as follows. Section II outlines works relating the Thai text correction. Section III details our recommended annotation pipeline for efficient data annotation. Section IV layout our proposed text correction method, the Extendable Neural Contextual Corrector (XNCC). Section V describes our experimental setup to evaluate various text correction methods at various stages of development. Section VI discusses the results of the experiments. Section VII concludes the trade-offs of various text correction systems and provides recommendation for progressively building effective text corrector.

II. RELATED WORKS

The aim of spell correction is to correct erroneous words into the words originally intended. Spell correctors can be viewed as noisy channel models which aim to produce the most probable correction. In noisy channel modeling [8], the corrector models the signal (language modeling) and the channel (i.e., errors being introduced into the text) to produce corrections. In addition to the corrector, correction systems often feature a detector to improve both speed and accuracy. In this section, we will examine the various approaches to spell correction by formulating how each method models the noisy channel problem.

Dictionary-based correctors have the most simplistic modeling. Error detection is quite simple, a token is considered erroneous if it does not exist within the dictionary [9]. For languages without explicit token boundaries (e.g., Thai), a word tokenizer is required for preprocessing. These correctors are publicly available as Free open-sourced software (e.g., Peter Norvig [10], SymSpell [9], Hunspell[2] and often come with ready-to-use dictionaries. Dictionary-based correctors (e.g., Peter Norvig, SymSpell) use lexical methods to model the channel. For example, Peter Norvig and SymSpell use Damerau-Levenshtein distance, that is correction candidates with higher distance are less prefered. Simplistic forms of language modeling such as word frequency and word grams are used for tie breaking.

To better model the corrections, methods have been proposed to augment both the channel model and the language model. Character confusion [11], multi-character edits [12] have been proposed for error correction in Thai optical-character-recognition systems. References [13] and [14] have proposed Soundex with multi-model reranking for error correction in Thai search engine queries.

Since tokenizers are not accurate on noisy text, propagation errors from tokenization are problematic for token-based correctors. Specialized correctors such as [3] and [11] for Thai and [15] for Chinese operate on the character-level and utilize a pruned search algorithm.

Modern Thai correction adopted neural-based sequence-to-sequence (Seq2Seq) methods from the English grammatical error correction literature. Seq2Seq methods model both the channel and the language in an end-to-end manner. This exploits neural-based ability to learn features automatically from erroneous-correct text pairs. In addition, modern correctors meant for automatic correction have structures to handle OOV tokens (e.g., names, new words). Copy-Augmented Transformer [4] can copy tokens from the source text, allowing the model to produce corrections with unaltered OOV tokens. The two-stage corrector [2] uses a detection-stage to mask out non-erroneous OOV tokens prior to correction. However, these only allow leaving OOV tokens uncorrected. On the other hand, dictionary-based methods can produce correction with new tokens by adding them into the dictionary. Since internal vocabulary of Seq2Seq models is tied to the learnt neural embeddings, model re-training and additional text with tokens are required to expand the dictionary.

Modern English grammatical error correction (GEC) research has focused on expanding the model training stage (e.g., training on noisy annotated data [4], [16], training on synthetic data [4], [17], [18]). In addition, researchers have reformulated GEC into iterative editing task to utilize large language models pre-trained on unannotated data [19], [20].

III. ANNOTATION

Our purposed annotation routine aims to address two issues: expanding compatibility with text correctors and reducing human effort required to annotate data.

Producing annotation data that is compatible with a variety of text correctors allows implementors to pick text correction

[1] LibreOffice dictionaries GitHub and SyafiqHadzir, Hunspell-TH GitHub.

[2] Hunspell website.

methods that best suit their needs without locking them into a specific class of models in the future. We annotate at the character-level instead of word-level, corrections are marked with the goal of one annotation covering one word instead of a contiguous segment of errors. For example, the erroneous text "I sea u" would be annotated as "I see you" instead of "I see you" or "I see you". This lets us derive a dictionary of individual correct and erroneous tokens required for dictionary-based methods. Unlike previous methods [2], [21] where the absence of corrective annotation denotes correct tokens, annotators explicitly annotate both correct and erroneous tokens. We found this method to be highly effective at preventing erroneous tokens from being introduced to the correct-tokens vocabulary, which is important for dictionary-based text correctors.

To reduce the effort required to build enough data to develop text correction systems, we utilize a dictionary-based maximum-matching tokenizer to dynamically produce automatic annotations. This exploits the fact that most errors are non-word errors (erroneous tokens outside of the dictionary), as shown in Table 7, and thus can be detected by dictionary-based methods. Erroneous tokens detected are automatically annotated with the most common correction. As a result, texts that feature tokens from previous annotations are automatically annotated and only require verification from the human operators. Unannotated data are queued up for annotation by the number of meaningful characters not covered by the automatic annotations. To prevent train-test leakage, annotations from the test-sets must not be used by the automatic annotator when developing the training-sets and the development-sets. All development-sets and test-sets are fully annotated to ensure accurate evaluation. As a result, for a each data source we recommend annotating the development-set first, followed by the training-set, and then test-set. This ordering maximizes vocabulary coverage since the annotators do not need to label tokens already present in the development-set. Fig. 1 shows annotation coverage of our Channel A data (see Section V-B) as we annotate the data. 89.4% character coverage 89.8% token coverage on the training set is achieved from automatic annotation purely with data from the development-set.

For our experiments in Section V, we explicitly instruct our annotators to verify every automatic annotation for every line they annotated. However, our preliminary experiment on similar data only required our annotators to fill in the gaps left by the automatic annotations (see Appendix B). Both methods have their own pros and cons. Although, only annotating the gaps left by the automatic annotations reduced the effort required for each data entry. This approach can lead to more unique non-word errors being discovered given the same annotation effort but at the expense of real-word errors left unannotated.

At training time, unannotated data are annotated using the same method for producing automatic annotations. Although some of the automatic annotations are mislabeled, they are still helpful when either in-domain data is abundant, or annotation capacity is limited. Nevertheless, our results (in Section VI) demonstrate that performant neural correctors can be built using this practice.

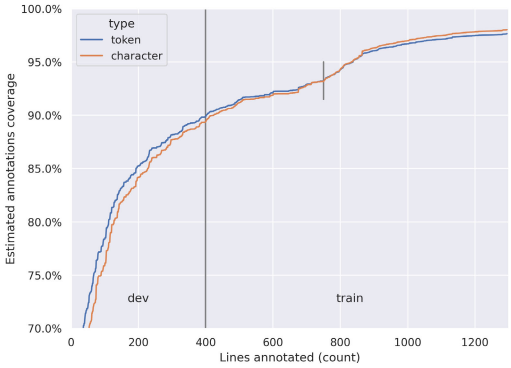


FIGURE 1. Estimated coverage of Channel A training-set as data is being progressively annotated. The vertical line at the 750th annotation marks the switch to queuing data based on number of noncovered characters. Only 859 out of 60,405 lines of the training set were annotated by humans, coverage primarily comes from automatic annotations. Coverage is only an estimatation since some automatic annotations are mislabeled.

IV. EXTENDABLE NEURAL CONTEXTUAL CORRECTOR (XNCC)

Our proposed method, XNCC, consists of four modules: text normalization, tokenization & masking (TokM), error detection, and correction. The overall structure is shown in Fig. 2. The primary contribution of XNCC is the novel correction module which utilizes neural-based generation techniques to separately model the error channel and the language. The input text is passed to the text normalization module (detailed in Appendix A). The normalized text is tokenized with tokens outside our scope masked out by predefined rules (see Section IV-B). Concurrently, the normalized text is detected for errors using a neural-based error detector (see Section IV-C). The error ranges produced from the detection module are projected into the tokenized text with masking. The unmasked tokens that overlap with error ranges are merged into erroneous segments (requiring correction). The tokens, along with erroneous segments, are passed to the correction module (see Section IV-D), which produces the final corrected sequence.

A. EXTENDABLE DICTIONARIES

In addition to the static vocabularies used by neural models, XNCC also features two extendable dictionaries: the error dictionary (error-dict) and the correct dictionary (correct-dict). This allows recognition and correction of new words without model retraining.

The error-dict is a many-to-many mapping from misspellings (error-tokens) to their possible corrections (correct-tokens). For example, the erroneous token "คราฟ" is mapped to two correct tokens: "ครับ" and "คาร์ป". The error-dict can be produced from the individual annotations that contain a correction. Moreover, frequently misspelled words can also be directly added to the error-dict.

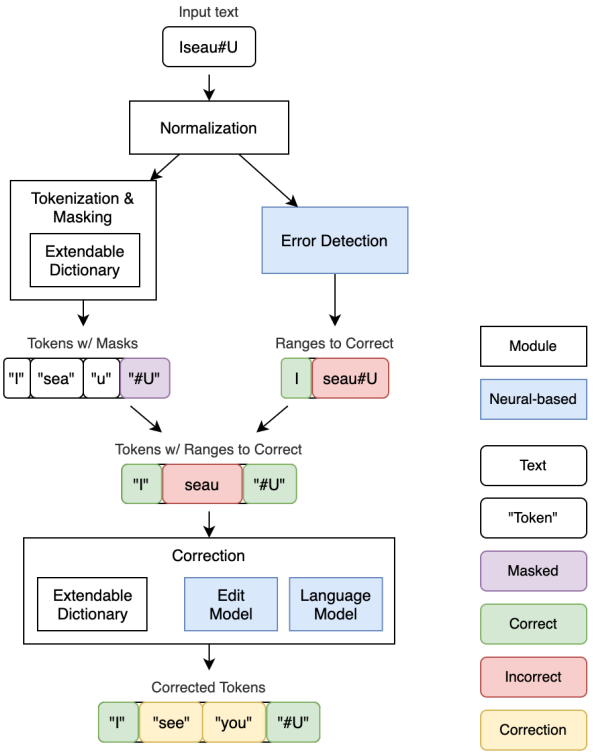


FIGURE 2. Structural overview of the Extendable Neural Contextual Corrector (XNCC). The figure shows the model operating on an example input "I sea u #U". The English example is chosen for clarity, XNCC operates on Thai.

The correct-dict is a many-to-one mapping from correct-tokens to a token in the neural-based Language Model vocabulary (LM-vocab). Since both the correct-dict and LM-vocab are built from the correct-tokens present in the training data, the tokens in the correct-dict are typically mapped to LM-token of the same spelling. On the other hand, rare correct-tokens that are cut off from LM-vocab are mapped to the `UNK` token. When a correct-token is added to the correct-dict, it should be mapped to LM-token that is a synonym. If a synonym does not exist, it is mapped to `UNK` token.

Manual additions to the extendable dictionaries are highly task specific. Our preliminary experiment of directly adding the whole official Thai dictionary [5] has resulted in unsatisfactory performance since archaic words are not present in our data.

B. TOKENIZATION & MASKING (TokM)

Our TokM module enables the model to recognize tokens present in the correct-dict, while masking out some portion of the text to prevent correction. The TokM module consists of four stages: multi-mask-tokens, dictionary-tokens, single-mask-tokens, and special-tokens. Each stage extracts tokens in positions not previously marked by the prior stages. Since each stage also masks the stages after it, the stages are ordered according to their size. For our task, we have categorized six types of mask-tokens: URLs, Hashtags, numbers, alphanumeric codes, English text, and text from other languages. First, the multi-mask-tokens stage is responsible for extracting URLs and Hashtags, which can contain other regular tokens. Second, the dictionary-tokens stage uses the maximum-matching tokenizer with the correct-tokens dictionary to extract regular tokens. Third, the single-mask-tokens stage extract and mask-out numbers, alphanumeric codes, English text, and text from other languages. Lastly, the special-tokens stage collects the remains text which are symbols (e.g., space, '–', ';', ':', '!', '?', '.', '$', '$, '฿') and single character regular tokens (i.e., "ๆ", "ฯ", "ๆ").

C. ERROR DETECTION

Our error detection module features the same error detector as [2], which is a multi-layer Bidirectional-LSTM sequence tagger with Bi-LSTM character encoding [22], [23].

The input text is tokenized into a sequence of tokens $\vec{w} = \{w_1, w_2, \ldots, w_N\}$ by a maximum-matching tokenizer. This specific tokenizer uses the detector vocabulary as its dictionary. The error detector produces a label of either *Error* or *Correct* for each input token w_i.

The vocab is derived from source tokens in the training data. Therefore, it includes both correct and erroneous tokens. The normalized text from our annotation routine (see Section III) is tokenized with the vocab. The word-level tokens are labeled as *Error* if they overlap with any corrective annotation. The model is trained with gradient descent. The hyperparameters are listed in Appendix C.

D. ERROR CORRECTION

The corrector searches for a sequence of *corrective operations* which produce the lowest *correction cost*. Corrective operations are split into character-operations (char-ops) and token-operations (token-ops). There are three types of char-ops (i.e., char-acceptance CA_*, char-deletion CD_*, char-insertion CI_*) and three types of token-ops (i.e., sequence-begin T_{BEGIN}, token-acceptance TA_*, sequence-terminate T_{END}). Intuitively, the char-ops dictate how to input characters are accepted or modified, while the token-ops dictate how the resulting characters are decoded into tokens. Fig. 3 shows a correction of an example input "I sea u #U", which produces the target sequence "I see you #U", alongside the corrective operations. Calculation of the correction cost is detailed in Section IV-D1. Section IV-D2 outline the rules for generating corrective operations and implementation details on how to keep track of the search state. Section IV-D3 explains how the corrector optimizes for the corrective operations with the lowest cost.

1) CORRECTION COST

The corrector uses the Edit Model and the Language Model to compute the correction cost of possible output sequences. Given a possible output sequence *Tokens*, the correction cost is defined as the sum of fluency cost and edit cost for each token t_i (see Eq 1). Intuitively, fluency cost penalize

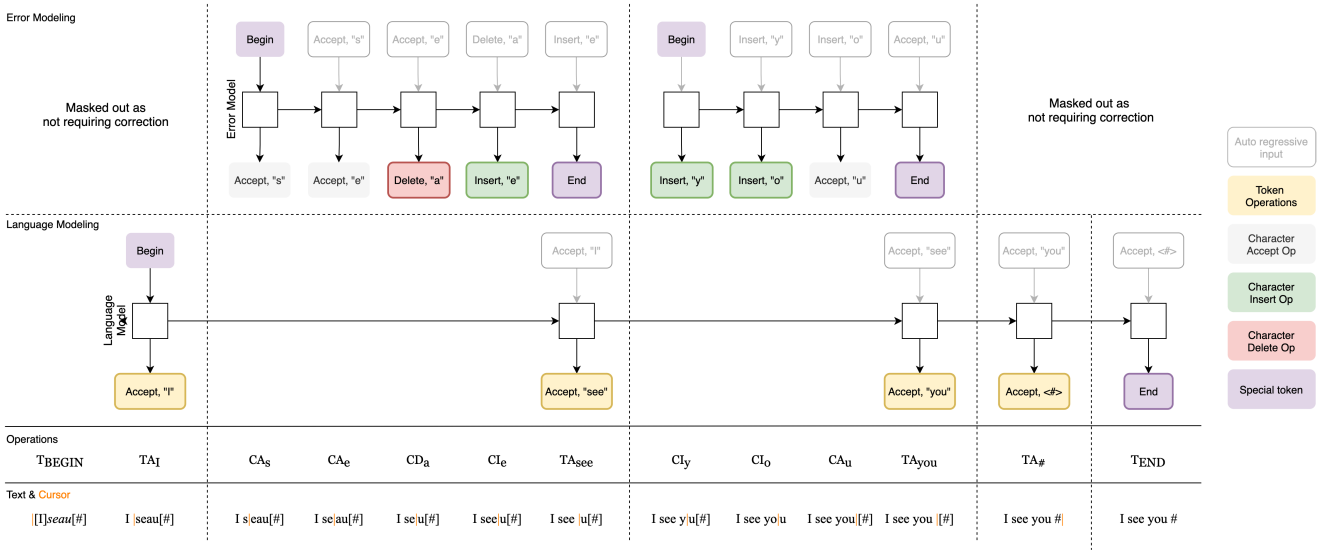


FIGURE 3. XNCC correcting the input "I sea u #U" which is preprocessed by prior stages as shown in Fig. 2. The English example is chosen for clarity, XNCC operates on Thai. Outputs with a highlighted border contribute to the correction cost.

improbable output sequences while the edit cost penalize improbable edits.

The fluency costs Flu_{t_i} are computed from negative log-probabilities for each token t_i in *Tokens* estimated by the Language Model minus the token-reward constant α clipped to a minimum value of zero (see Eq 2). The token-reward constant α is used to alleviate preference for short sequences when decoding [24], [25].

If the token t_i was generated using modifying char-ops, such as *CD* or *CI*, the edit cost $Edit_{t_i}$ is determined by the scaled sum (β) of the edit-cost constant γ and the negative log-probabilities associated with each modifying char-op ($e_j \in Edits$) used to produce the token t_i (see Eq 3). However, if the token t_i was solely produced through char-acceptance operations (CA_*), the edit cost is influenced by the dictionary from which the token was decoded. In the case of the token t_i, the edit cost can take either a value of zero ($Edit_{t_i} = 0$) or the map-cost constant ($Edit_{t_i} = \delta$), depending on whether the correct dictionary or the error dictionary was utilized, respectively.

$$Cor = \sum_{t_i}^{Tokens} (Flu_{t_i} + Edit_{t_i}) \tag{1}$$

$$Flu_{t_i} = \max(0, -\log(p(t_i \mid t_1, \ldots t_{i-1})) - \alpha) \tag{2}$$

$$Edit_{t_i} = \beta \sum_{e_j}^{Edits} (\gamma - \log(p(e_j \mid e_1, \ldots e_{j-1}))) \tag{3}$$

2) SEQUENCE GENERATION

The corrector produces a target sequence by performing corrective operations. The specific rules used for generating operations are outlined below.

Tokens from the correct-dict, the error-dict, and special-tokens (from Section IV-A) are populated into a trie. The trie is used to constrain the search space by preventing the exploration of char-ops that would not lead to a valid token. Tokens from the error-dict and special-tokens in the trie are unreachable if CD_* or CI_* was performed since the last TA_*.

For the production of every possible sequence, the corrector keeps track of the following as the search state: a pointer to the trie to represent the current token being produced, a cursor on the input text, a set of characters that have been deleted since the last CA_*; and the correction cost of the sequence.

The root search state starts with the initial operation T_{BEGIN}. The pointer is pointed to the root trie node, The cursor is set to the beginning of the input. And the cost is set to zero.

Character-acceptance (CA_*) can be performed if the cursor is not at the end, if there exist an edge of the same character as the cursor on the current trie node, and the character is not in the delete-set. When performed, the pointer advances to the node of the same character, the cursor is moved by one character, and the delete-set is cleared.

Character-deletion (CD_*) can be performed if the cursor is not at the end, the previous op is either CA_*, CD_*, or T_{BEGIN}. When performed, the cursor is moved by one character, add edit cost for deleting the cursor character, and the character is added to the delete-set.

Character-insertion (CI_*) can be performed for any character that is an edge from the trie node, and is not a character in the delete-set. When performed, the pointer advances to the node of the same character and adds edit cost for inserting the character.

Token-acceptance (TA_*) can be performed if the trie node is a token, and either the cursor is at the end or the cursor character is not in the delete-set. When performed, the pointer resets to the root trie node, adds fluency cost for producing the corresponding correct token. If the token is from the error-

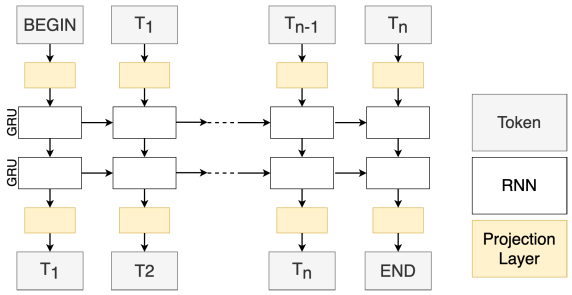


FIGURE 4. The sequence-modeling architecture used in both the language model and the edit model.

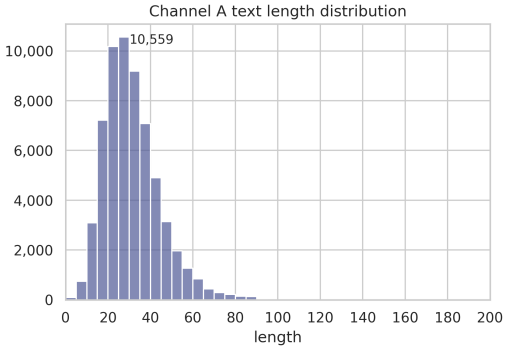


FIGURE 5. Distribution of normalized text length in characters for channel A.

dict, the edit cost is the map-cost δ. If all character ops since last TA_* are CA_*, edit cost is zero. Otherwise, edit cost is the cost of terminating the edit sequence.

Terminate (TA_{END}) can be performed if the cursor is at the end, and the pointer is at the root trie node. When performed, add fluency cost for producing an END token. This operation marks the end of the sequence generation.

When the cursor is operating on portions of text that are not marked for correction, only TA_* ops are allowed. The generation of TA_* are based on token boundaries produced from TokM (see Section IV-B).

3) OPTIMIZATION

Our approach for optimizing the target sequence involves using Dijkstra's algorithm [26] to find the lowest-cost path from the initial state TA_{BEGIN} to the terminal state TA_{END}. This method was inspired by [11] and [15] use of search for token decoding. We have also incorporated a modified beam search to reduce the search space. Beam search is a greedy algorithm that restricts the number of paths to explore at each level [27]. Our preliminary experiments have revealed that defining the beam depth as the output length leads to the incorrect insertion of new characters for the sake of increasing number of tokens. Therefore, we propose a modification to the traditional implementation of beam search where we define the depth of the input characters consumed (i.e., the cursor position, see Section IV-D2) to prioritize input consumption instead.

4) LANGUAGE MODEL (LM)

The Language Model (LM) is a simple autoregressive recurrent neural network with a two-layer Gated recurrent unit [28] and shared weights between the embeddings and the output layer. The architecture is shown in Fig. 4. Tokens are embedded by the projection layer. The embeddings are encoded by the two-layer bi-directinal GRU. The encodings are projected back to the token space by the projection layer. Effectively the model performs cosine similarity between the each encoding and the token embeddings.

The vocab is derived from corrected tokens in the training data plus three special tokens: *BEGIN*, *END*, and *UNK*. LM is trained with gradient descent to model a sequence of correct token. The hyperparameters are listed in Appendix C.

5) EDIT MODEL (EM)

The Edit Model shares the same architecture as the Language Model, as shown in Fig. 4. However, EM models a sequence of edit operations instead of a sequence of word-tokens. The vocabulary consists of three variants of edits for each character in character-set and two special tokens: *BEGIN* and *END*. The three variants correspond to the char-ops (detailed in Section IV-D2). Vocabulary coverage of input is ensured by text normalization (see Appendix A) and token masking (see Section IV-B).

EM is trained on sequences of edit operations produced from error-correct word pairs derived from the annotations. The distribution frequency of each error-correct pair in data is maintained during training. The hyperparameters are listed in Appendix C.

V. EXPERIMENT SETUP

Our experiment setup aims to evaluate three scenarios: 1) building text correctors from scratch with data annotation, 2) domain transfer text correctors to another with additional data annotation, and 3) domain transfer text correctors to another without data annotation. Our experiment consists of three data channels, which corresponds to the three scenarios. Details about the data from each channel are detailed in Section V-A. Data entries from each channel are shuffled and split into three sets: training-set, development-set, test-set. Models are built or trained using the training-set and the development-set unless explicitly stated otherwise, whereas the test-set is used for evaluation. Of the three scenarios, the models are evaluated in various configurations detailed in Section V-C.

A. DATA SOURCES

Our data is collected from 3 automated text-based chatbot channels, which is named in the paper as A, B, and C. Data is only collected for the client side (does not include chatbot automated response). Channel A and B are used by separate groups of our customers whereas channel C is used by our internal staff. Due to the short nature of chat messages (as shown in Fig. 5, 6, and 7), many duplicate entries exist across multiple users and channels. The text data is normalized

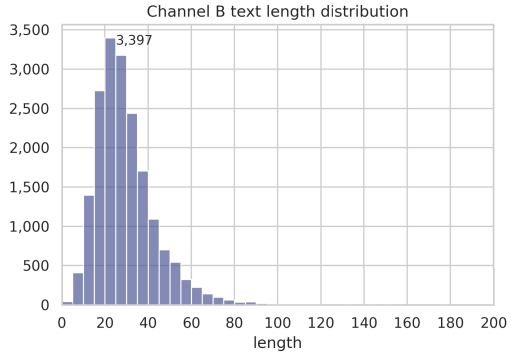


FIGURE 6. Distribution of normalized text length in characters for channel B.

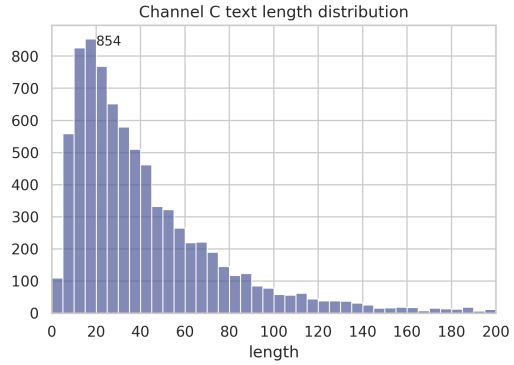


FIGURE 7. Distribution of normalized text length in characters for channel C.

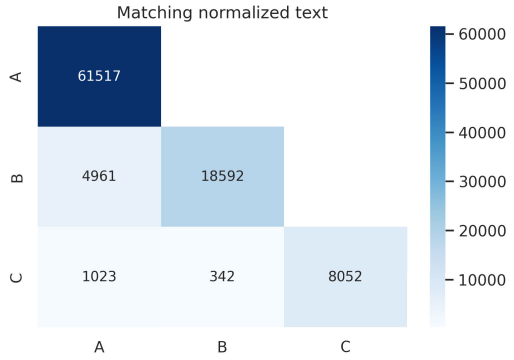


FIGURE 8. Overlap of normalized unique channel data.

(detailed in Appendix A) with the duplicates removed on a per channel basis. A data entry is a unique normalized text from some channel. The overlap of data entry between different channels is shown in Fig. 8 and 9.

Overall, texts from channel A and B are shorter and contain more errors than channel C. This is apparent when observing the length distribution (shown in Fig. 5) and base error-rate of the data (shown in the "Do nothing" row of Tables 3, 4, and 5)

B. ANNOTATION INFORMATION

Our data is annotated in the following order, A development-set, A training-set, A test-set, B development-set, B test-set, and C test-set. Our annotation routine is detailed in

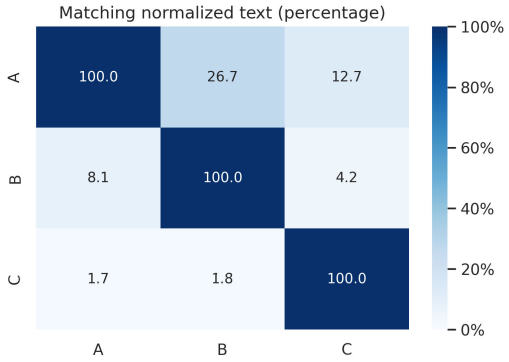


FIGURE 9. Overlap of normalized unique channel data normalized along the vertical axis. "26.7" denotes that 26.7% of data in channel B is also present in channel A.

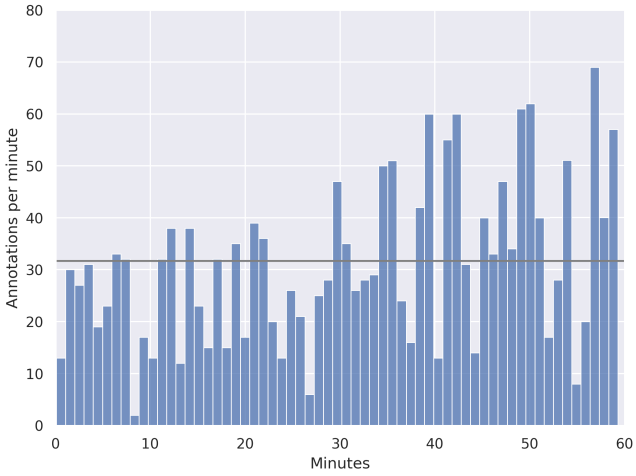


FIGURE 10. Annotation speed from 60 minutes of annotation. The horizontal line shows the average speed of 31.7 tokens per minutes.

Section III. The training-set of channel A is partially annotated until the automatic annotations cover ∼98% of the data. The training-set of channel B is annotated purely with automatic annotations. Details of each data split are shown in Table 1. Fig. 10 shows the annotation speed of an hour of continuous annotation. A single annotator is able to annotate 1,899 tokens in an hour, averaging 31.7 tokens per minute. Annotations include both confirming tokens from automatic annotations and annotating new tokens.

C. CONFIGURATIONS

Given one of the three scenarios, an implementor has multiple options to utilize the available data.

First scenario: the implementor is developing a new text corrector without pre-existing annotated data of their target domain (channel A). When starting from scratch, the implementor would start experimenting with off-the-shelf solutions. We start off with evaluating off-the-shelf methods on channel A, requiring only annotating the test-set. Then the implementor might start annotating their data for model training. We experimented with utilizing different amounts of the annotated training-set (i.e., none, half, and all) alongside a fully annotated development-set.

TABLE 1. Details of each data split.

Split	Ch. A - Train	Ch. A - Dev	Ch. B - Train	Ch. B - Dev
Lines				
Total	60,405	400	17,695	400
Annotated by human	859 (1.4%)	400 (100.0%)	-	400 (100.0%)
Fully annotated by human	845 (1.4%)	400 (100.0%)	-	400 (100.0%)
Fully annotated automatically	51,718 (85.6%)	400 (100.0%)	14,554 (82.2%)	400 (100.0%)
Characters				
Total	1,870,448	12,041	514,352	11,628
Non-symbols	1,749,121 (93.5%)	11,127 (92.4%)	495,100 (96.3%)	11,206 (96.4%)
Covered by human-annotations	35,282 (2.0%)	11,127 (100.0%)	-	11,206 (100.0%)
Covered by auto-annotations	1,718,969 (98.3%)	-	483,817 (97.7%)	-
Tokens				
Human-annotations	8,433	2,699	-	2,922
Human-corrections	1,217 (14.4%)	346 (12.8%)	-	363 (12.4%)
Auto-annotations	427,162	-	127,212	-
Auto-corrections	41,443 (9.7%)	-	11,954 (9.4%)	-

Split	Ch. A - Test	Ch. B - Test	Ch. C - Test
Lines			
Total	400	400	400
Fully annotated by human	400 (100.0%)	400 (100.0%)	400 (100.0%)
Characters			
Total	12,733	11,845	17,412
Non-symbols	11,920 (93.6%)	11,533 (97.4%)	14,959 (85.9%)
Covered by human-annotations	11,920 (100.0%)	11,533 (100.0%)	14,909 (99.7%)
Tokens			
Human-annotations	2,909	2,942	3,689
Human-corrections	332 (11.4%)	349 (11.9%)	118 (3.2%)

Second scenario: the implementor is developing a text corrector for a new domain (channel B) that is similar to an existing domain (channel A). We experimented with directly utilizing the existing text correctors to see how they generalize to the new domain. Then, we evaluate how each model performs when given additional in-domain data for training (also known as transfer learning for neural-based methods).

Third scenario: the implementor is developing a text corrector for a new domain (channel C) that is significantly different from the annotated data they already have. Like the second scenario, existing text correctors were evaluated on the new domain. However, instead of continuing experimenting by annotating more data, we focus on methods that can easily (retraining not required) have their vocabularies extended to evaluate how each text corrector performs when new words are added by the users.

D. EVALUATION CRITERIA

Word-error-rates (WER) and General Language Evaluation Understanding (GLEU) were chosen for their use in prior Thai text correction research [2], [3]. WER is the relation of errors in some given sequence to the length reference sequence. Errors are defined as the minimum number of insertions, deletions, and substitutions needed to correct the given sequence to match the reference. Whereas GLEU, at a high level, compares the given sequence against the reference at the grams-level instead of word-tokens. Given two sequences of the same WER, GLEU will penalize sequences with sparse distribution of errors, which is found to have a higher correlation with human judgment [29].

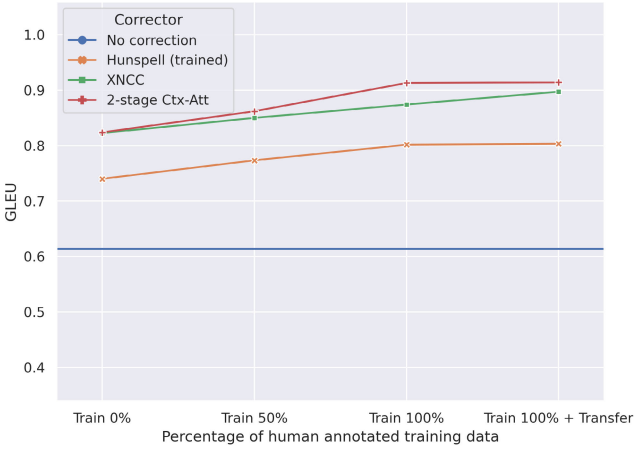


FIGURE 11. GLEU scores on channel A with varying amount of training data.

VI. RESULTS & DISCUSSION

This section shows and discusses the results of the experiments outlined in Section V. Ablation study of XNCC is detailed in Appendix B-A.

For the first scenario, we experiment with developing text correction systems from scratch for channel A. Evaluation of all models and configuration are shown in Tables 2 and 3. Publicly available dictionary-based methods along with the built-in dictionary were unable to reduce the overall errors. However, the combination of Hunspell with a clean dictionary produced with our annotation routine and an error aware tokenizer was able to reduce the overall number of errors. Fig. 11 and 12 shows the results at varying amount of training data. The two-stage contextual attention (2-stage Ctx-Attn)

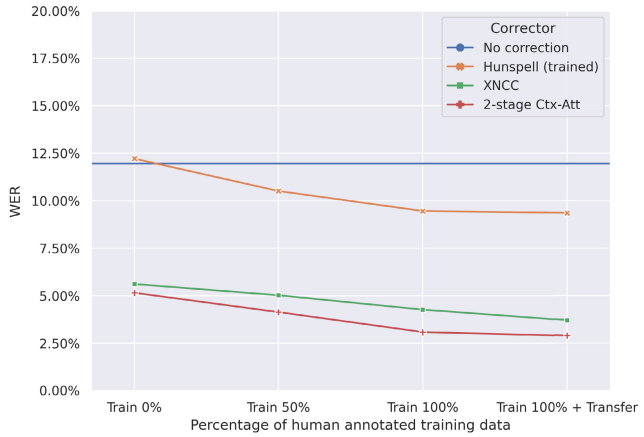


FIGURE 12. Word-error-rate on channel A with varying amount of training data.

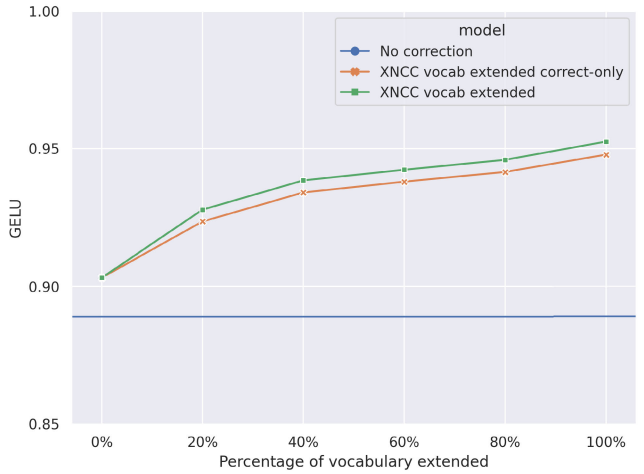


FIGURE 13. XNCC GLEU scores on channel C with varying amount of test-set tokens added to the dictionary. The 0% and 100% results are the same as the ones reported in Table 5.

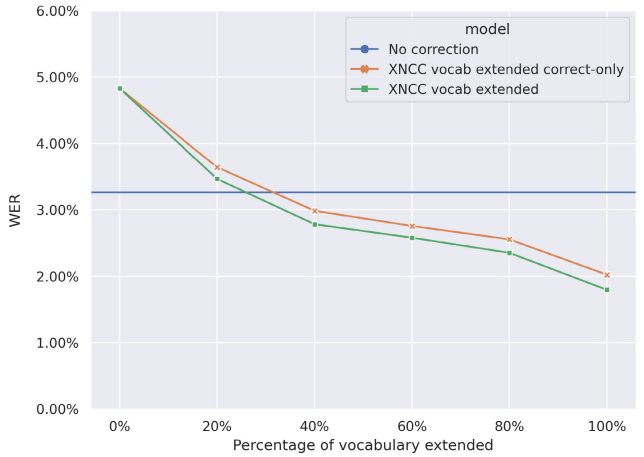


FIGURE 14. XNCC WER on channel C with varying amount of test-set tokens added to the dictionary. The 0% and 100% results are the same as the ones reported in Table 5.

performed the best in reducing the overall number of errors at all amounts of annotated data. While XNCC, although

TABLE 2. Evaluation of dictionary-based correction methods on channel A.

Tokenizer-dict	Corrector	Cor-dict	WER	ΔWER	GLEU
Do nothing			0.1196	0.00%	0.6138
Ideal			0.0000	-100.00%	1.0000
Off-the-shelf (OTS)					
OTS	PeterNorvig	OTS	0.3118	+160.71%	0.4201
OTS	Symspell	OTS	0.3118	+160.71%	0.4201
OTS	Hunspell	OTS	0.2953	+146.98%	0.4640
Trained					
Correct	PeterNorvig	Correct	0.2257	+88.74%	0.5782
Correct	Symspell	Correct	0.1879	+57.14%	0.6400
Correct	Hunspell	Correct	0.1301	+8.79%	0.7604
All	PeterNorvig	Correct	0.1902	+59.07%	0.6152
All	Symspell	Correct	0.1741	+45.60%	0.6537
All	Hunspell	Correct	**0.0946**	**-20.88%**	**0.8016**

TABLE 3. Evaluation of correction methods on channel A.

Corrector	WER	ΔWER	GLEU
Do nothing	0.1196	0.00%	0.6138
Ideal	0.0000	-100.00%	1.0000
Off-the-shelf			
Hunspell	0.2953	+146.98%	0.4640
Trained on A			
Hunspell	0.0946	-20.88%	0.8016
XNCC	0.0427	-64.29%	0.8741
2-stage Ctx-Att	0.0329	-72.53%	0.8996
2-stage Ctx-Att*	**0.0309**	**-74.18%**	**0.9130**
Pre-trained on A & B, finetuned on A			
Hunspell	0.0936	-21.70%	0.8032
XNCC	0.0371	-68.96%	0.8972
2-stage Ctx-Att*	**0.0289**	**-75.82%**	**0.9140**

* Pre-training data is noise-injected according to 2-stage Ctx-Att paper [2].

significantly better than Hunspell, is not as accurate as the 2-stage Ctx-Attn.

For the second scenario, we experiment with developing text correction system for another domain (i.e., channel B). Results show that XNCC and 2-stage Ctx-Attn perform similarly with around 60% error rate reduction (Table 4). Transferring the annotated data from channel B back to channel A results in higher correction performance for all models across the board (Table 3, Fig. 11 and 12).

For the third scenario, we experiment with repurposing existing text correction systems in a significantly different and challenging domain (i.e., channel C and significantly lower base error rate). Without modifications, all methods were unable to produce corrections that further reduce the error rate. However, XNCC was able to further reduce the error rate by 38% and 45% when its dictionary was extended with correct tokens and error-correct token pairs respectively (Table 5). Fig. 13 and 14 show the GLEU scores and WER of XNCC at varing amounts of extensions to the dictionary. The correct tokens and the corresponding error-correct token pairs are added in descending order of their frequency.

VII. CONCLUSION

This paper evaluated how a multitude of text correction approaches perform at various stages of development.

When starting from scratch in some domain (channel A), off-the-shelf systems alone are unsuitable for performing automatic text correction since they introduce more errors than they correct. While prior work has not found success

TABLE 4. Evaluation of correction methods with various configurations on channel B.

Corrector	WER	ΔWER	GLEU
Do nothing	0.1235	0.00%	0.6161
Ideal	0.0000	-100.00%	1.0000
Trained on A			
Hunspell	0.1135	-8.31%	0.7780
XNCC	0.0674	-45.43%	0.8340
2-stage Ctx-Att*	**0.0495**	**-59.95%**	**0.8860**
Pre-trained on A & B, finetuned on B			
Hunspell	0.0960	-22.31%	0.8095
XNCC	**0.0488**	**-60.48%**	0.8917
2-stage Ctx-Att*	0.0495	-59.95%	**0.8923**

* Pre-training data is noise-injected according to 2-stage Ctx-Att paper [2].

TABLE 5. Evaluation of correction methods with various configurations on channel C.

Corrector	WER	ΔWER	GLEU
Do nothing	0.0326	0.00%	0.8890
Ideal	0.0000	-100.00%	1.0000
Trained on A & B			
Hunspell	0.1395	+327.91%	0.7623
XNCC	0.0483	+48.06%	0.9032
2-stage Ctx-Att*	0.0543	+66.67%	0.8805
Extended Vocabulary			
Hunspell†	0.0614	+88.37%	0.9073
Hunspell‡	0.0531	+62.79%	0.9222
XNCC†	0.0202	-37.98%	0.9478
XNCC‡	**0.0179**	**-44.96%**	**0.9525**

* Pre-training data is noise-injected according to 2-stage Ctx-Att paper [2].
† Correct vocabulary extended with vocabulary derived from the test-set.
‡ Correct and Error vocabularies extended with vocabulary derived from the test-set.

in utilizing dictionary-based methods [2], we have found that given a clean dictionary and a tokenizer that is aware of erroneous tokens, dictionary-based systems can produce correction that reduce the overall error rate. Since Hunspell and maximal matching tokenization implementations are publicly available, they serve as a good starting point for implementors. Given access to computational resources and engineering effort, the two-stage contextual attention (2-stage Ctx-Attn) [2] performed the best on all amount of annotated data for correcting in-domain text. Our proposed correction method significantly out performed Hunspell but is not as accurate as 2-stage Ctx-Attn.

When adapting existing resources to produce text correction systems to a new domain of similar nature (channel B), directly utilizing effective systems developed for the existing domain (Hunspell, XNCC, 2-stage Ctx-Attn trained on channel A) proved robust at reducing the total amount of errors. However, more accurate corrections can be achieved with little annotation (only annotating the development-set). When jointly using annotated data from both domains XNCC and 2-stage Ctx-Attn performed comparably on the new domain. Given the additional annotations from the new domain, adapting the data back to the original domain also provides uplift in correction performance for all three effective methods.

When utilizing existing correctors on a significantly different and more challenging (having a lower base error rate) domain, XNCC is recommended. XNCC can have its dictionary easily extended with tokens for the new domain and produce corrections that further reduce the error rate. Since XNCC was specifically developed for Thai correction, it can be adapted to other languages without explicit token boundaries (e.g., Chinese, Japanese) or provide correction on inputs with incorrect boundary markers.

APPENDIX A
TEXT NORMALIZATION

Our text normalization routine has three primary objectives: produce stable normalization (i.e., consistent output across multiple passes), conform to industry-standard NKFC-based normalization,[3] and provide obvious non-destructive corrections.

The Thai character-set consists of consonants (**C**), vowel characters, and tone marks (**T** i.e., ่, ้, ๊, ๋). One or more vowel characters are use to write actual vowels in the Thai Language. There are four types of vowel characters used in modern text: leading vowel (**L** i.e., เ, แ, ไ, ใ, โ), hanging vowel (**H** i.e., ิ, ี, ึ, ื, ุ, ู, ั, ็), following vowel (**F** i.e., ะ, า, ๅ). This normalization attempts to produce text with the following pattern: "LCHTF". Thus, specific patterns of characters can be reordered non-destructively.

The text normalization routine is as follows:

1) Performing standard NKFC Unicode normalization
2) Replacing two consecutive "เ " with an "แ"
3) Undoing decomposition of "อำ" from NKFC normalization
4) Merging consecutive instances of the same vowel character or tone mark
5) Reordering "LTC" as "LCT"
6) Reordering "CTH" as "CHT"
7) Reordering "CFT" as "CTF"
8) Merging consecutive instances of the same vowel or tone mark, again
9) Redoing decomposition of "อำ"

APPENDIX B
PRELIMINARY EXPERIMENTATION

Prior to the experiments in Section V, we carried out a preliminary experimentation on an older version of channel A data. There are three differences between the preliminary data and the final data. First, development-set is not considered a separate data split that is fully annotated like the test-set. Instead, the development-set is part of the larger training-set and as a result mostly comprised of automatic annotations. Second, the annotators are not instructed to confirm every automatic annotation. As a result, the annotators effectively perform partial annotations to fill in the gaps between the automatic annotations. Third, the data from other channels

[3] Unicode Normalization Forms.

TABLE 6. Annotation information of our preliminary data.

Split	Total	Out-of-domain	Train	Dev	Test
Lines					
Total	828,010	757,805	69,605	300	300
Annotated by human	1,647	462	883	2	300
Fully annotated by human	542	7	235	0	300
Fully annotated automatically	643,198	581,039	61,589	270	300
Characters					
Total	38M	35.8M	2.2M	9,192	9,651
Non-symbols	26.4M	24.4M	2M	8,682	8,990
Covered by human-annotations	37,797 (0.1%)	16,014 (0.1%)	11,781 (0.6%)	12 (0.1%)	8,990 (100.0%)
Covered by auto-annotations	25.4M (96.2%)	23.4M (95.9%)	2M (98.7%)	8,581 (98.8%)	8,990 (100.0%)
Human-annotations	9,573	4,260	3,005	2	2,306
Human-corrections	1,301	60	951	1	289

TABLE 7. Type of tokens in source text of our preliminary test-split (channel A), which is comprised of fully annotated lines.

Token types	Count	Percentage
Correct tokens	2,017	87.47%
Non-word errors	244	10.58%
Real-word errors	45	1.95%

TABLE 8. Results comparing various text correctors from our preliminary experiments.

Corrector	WER	ΔWER	GLEU
Do nothing.	0.1696	0.00%	0.5911
Ideal	0.0000	-100.00%	1.0000
Extendable-dict			
Hunspell*	0.2803	+65.32%	0.4847
Hunspell	0.1216	-28.27%	0.7542
XNCC	0.0580	-65.80%	0.8684
XNCC**	**0.0528**	**-68.88%**	**0.8782**
Fixed-dict			
2-stage Ctx-Att	0.0511	-69.83%	0.8840
2-stage Ctx-Att**	**0.0447**	**-73.63%**	**0.8969**

*: Off-the-shelf, using the built-in dictionary,
**: Model is jointly trained on out-of-domain-split and training-split before being fine-tuned on the training-split.

TABLE 9. Comparison of XNCC with various components in the corrector removed on our preliminary data (older version of channel A).

#	XNCC	WER	ΔWER	GLEU
	Do nothing	0.1696	0.00%	0.5911
	Ideal	0.0000	-100.00%	1.0000
1	Dict	0.1418	-16.39%	0.7314
2	Dict + E2C	0.0834	-50.83%	0.8144
3	Dict + E2C + LM*	0.0693	-59.14%	0.8341
4	Dict + E2C + EM*	0.0548	-67.70%	0.8751
5	Dict + E2C + LM + EM	0.0580	-65.80%	0.8684
6	Dict + E2C + LM* + EM*	**0.0528**	**-68.88%**	**0.8782**
7	Dict** + E2C + LM* + EM*	0.0511	-69.83%	0.8817
8	Dict** + E2C** + LM* + EM*	0.0451	-73.40%	0.8946

*: Model is jointly trained on out-of-domain and training split before fine-tuned on training-split,
**: Dictionary is extended with vocabulary from the test-split.

TABLE 10. XNCC inference time on 300 lines of text (2,483 tokens).

Inference time	seconds	uplift	lines/s	tokens/s
Without caching				
XNCC	9.14	-	32.8	271.6
With caching				
XNCC cold cache	7.98	12.7%	37.6	311.2
XNCC hot cache	4.34	52.5%	69.1	572.1

were combined into an out-of-domain set. Information about our preliminary data is shown in Tables 6 and 7.

TABLE 11. Hyperparameters of XNCC.

Hyperparameter	Value
Error Detection	
Vocabulary	12,840 tokens
Word embeddings layer	128 nodes
Character embeddings layer	256 nodes
Character bi-LSTM Encoder	128 nodes (in each direction)
Bi-LSTM Encoder	3 layers × 128 nodes (in each direction)
Bi-LSTM Encoder dropout	0.2
Batch size	32 entries
Window size	128 tokens
Optimizer	Adam
Exponential LR decay	
- Step size	10
- Gamma	0.1
Training learning-rate	0.0002
Finetuning learning-rate	0.00002
Early stopping patience	30 epochs
Error Correction	
α	5.0
β	1.0
γ	5.0
δ	0.0
Language Model	
Vocabulary	6,228 + 3 tokens
Word embeddings layer	256 nodes
GRU Encoder	512 nodes, 256 nodes
Batch size	20
Window size	35
Clips gradient norm	5.0
Optimizer	Adam
Learning-rate	0.0001
Finetuning learning-rate	0.00003
Early stopping patience	20 epochs
Edit Model	
Vocabulary	3 × 179 + 2 operations
Word embeddings layer	512 nodes
GRU Encoder	2 layers × 512 nodes
Batch size	32
Window size	128
Clips gradient norm	5.0
Optimizer	Adam
Learning-rate	0.0001
Finetuning learning-rate	0.00001
Early stopping patience	30 epochs

Results for the preliminary experiments are shown in Table 8 and are in-line with results in Table 3 from Section VI.

A. ABLATION STUDY

During our preliminary experimentation with XNCC, we conducted an ablation study to analyze the effect of each XNCC Corrector sub-module on end-to-end performance. We start with a bare corrector with only the correct-dict. In the absence of the LM and EM, the fluency cost Flu_{t_i} is zero, and the edit-cost of character-edits is γ. The results of various configurations are shown in Table 9.

TABLE 12. Error analysis of XNCC and 2-stage Ctx-Att on 30 lines sampled from the test-split.

	XNCC	2-stage Ctx-Att
Correctly corrected		
Non-word	5.1, 6.2, 7, 11, 15, 17, 18.1, 18.2, 19.1, 21, 23, 25, 29, 30.1 (14)	1, 5.1, 6.2, 7, 8, 11, 15, 19.1, 16, 17, 18.1, 18.2, 21, 23, 25, 26.1, 26.2, 27.2, 29, 30.1 (20)
Real-word	30.2 (1)	30.2 (1)
Uncorrected		
Non-word	2, 8, 16, 22, 26.1, 26.2, 27.1, 27.2 (8)	22, 27.1 (2)
Real-word	4.1, 4.2, 5.2, 6.1, 10, 12, 14, 18.3, 19.2 (9)	4.1, 4.2, 5.2, 6.1, 10, 12, 14, 18.3, 19.2 (9)
Mis-corrected		
Non-word	1 (1)	2 (1)
Real-word	- (0)	- (0)
Errors introduced	- (0)	- (0)
Already Correct	3, 9, 13, 20, 24, 28 (6)	3, 9, 13, 20, 24, 28 (6)

A.B: The B^{th} error of the A^{th} line
(n): parenthesized numbers denote the count of each correction type

The mapping from error-dict to correct-dict was first added since it has the most significant impact on performance. This order aims to underscore that the mapping does not supersede other modules.

Overall, the results show that all three sub-modules in the XNCC Corrector module and the fine-tuning routine play a part in improving the final correction performance. In addition, the strong correction performance of entry #2 also suggests that a dictionary of misspelled tokens might be the missing trick to improve dictionary-based text correctors.

Lastly, we also experiment with extending the dictionary post-training. Entries #7 and #8 demonstrate the ideal case of extending the dictionary with relevant tokens.

APPENDIX C
HYPERPARAMETERS

Hyperparameters for all modules in XNCC are shown in Table 11. Reference to hyperparameters of the Error Detection module follows the same naming convention as [2]. The rest of the hyperparameters are named as per Section IV.

APPENDIX D
ERROR ANALYSIS

We sampled and analyzed 30 corrected lines on the test-split of channel A made by XNCC and 2-stage Ctx-Att. The results are shown in Table 12. Of the 30 lines sampled, 6 lines do not require any correction. Of the 24 lines that required correction, 33 errors required correction. Of the 33 errors, 23 were non-word errors and 10 were real-words errors. Of all 6 lines that do not require correction, both methods operate correctly and left the lines alone. Both XNCC and 2-stage Ctx-Att share the same set of corrected and uncorrected real-word errors. For the non-word errors, errors that 2-stage Ctx-Att is able to correctly corrected is a superset of the ones by XNCC.

Overall, both correctors are very conservative with their corrections. Of the 30 lines analyzed, both methods only made one mis-correction, and no errors introduced to any existing correct tokens.

APPENDIX E
INFERENCE TIME

This section discuss about XNCC inference time and optimization opportunities. The contextless nature of XNCC error modeling enables caching of token-level edit-cost. Table 10 shows inference time of XNCC on a separate dataset consisting of 300 lines (2,483 tokens) with a base word-error-rate of 16.96% executed on a single CPU core. XNCC is experimented on three configurations: without caching, with cold cache, and with hot cache. Special thanks to Atthakorn Petchsod for optimizing XNCC and running the experiments.

REFERENCES

[1] P. Nakwijit and M. Purver, "Misspelling semantics in Thai," in *Proc. 13th Lang. Resour. Eval. Conf.*, Marseille, France, Jun. 2022, pp. 227–236. [Online]. Available: https://aclanthology.org/2022.lrec-1.24

[2] A. Lertpiya, T. Chalothorn, and E. Chuangsuwanich, "Thai spelling correction and word normalization on social text using a two-stage pipeline with neural contextual attention," *IEEE Access*, vol. 8, pp. 133403–133419, 2020.

[3] S. Meknavin, B. Kijsirikul, A. Chotimongkol, and C. Nuttee, "Combining trigram and winnow in Thai OCR error correction," in *Proc. 17th Int. Conf. Comput. Linguistics*, 1998, pp. 836–842. [Online]. Available: https://aclanthology.org/C98-2133

[4] W. Zhao, L. Wang, K. Shen, R. Jia, and J. Liu, "Improving grammatical error correction via pre-training a copy-augmented architecture with unlabeled data," in *Proc. Conf. North*, Minneapolis, MN, USA. 2019, pp. 156–165. [Online]. Available: https://aclanthology.org/N19-1014

[5] *Office of the Royal Society*, The Royal Institute Dictionary 2542, BE Nanmeebooks, Bangkok, Thailand, 1999.

[6] *Office of the Royal Society*, The Royal Institute Dictionary 2554, BE Nanmeebooks, Bangkok, Thailand, 2011.

[7] W. Phatthiyaphaibun, K. Chaovavanich, C. Polpanumas, A. Suriyawongkul, L. Lowphansirikul, and P. Chormai, "PyThaiNLP: Thai natural language processing in Python," Tech. Rep., Jun. 2016, doi: 10.5281/zenodo.3519354.

[8] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2009.

[9] W. Garbe. (Jun. 2012). *SymSpell*. [Online]. Available: https://github.com/wolfgarbe/SymSpell

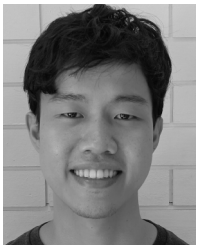
[10] P. Norvig. (Feb. 2007). *How to Write a Spelling Corrector*. [Online]. Available: http://norvig.com/spell-correct.html

[11] M. Rodphon, K. Siriboon, and B. Kruatrachue, "Thai OCR error correction using token passing algorithm," in *Proc. IEEE Pacific Rim Conf. Commun., Comput. Signal Process.*, Feb. 2001, pp. 599–602.

[12] E. Brill and R. C. Moore, "An improved error model for noisy channel spelling correction," in *Proc. 38th Annu. Meeting Assoc. Comput. Linguistics*, Hong Kong, 2000, pp. 286–293. [Online]. Available:https://aclanthology.org/P00-1037
[13] N. Angkawattanawit, C. Haruechaiyasak, and S. Marukatat, "Thai Q-Cor: Integrating word approximation and Soundex for Thai query correction," in *Proc. 5th Int. Conf. Electr. Eng./Electron., Comput., Telecommun. Inf. Technol.*, May 2008, pp. 121–124.
[14] P. Kittiworapanya, N. Saelek, A. Lertpiya, and T. Chalothorn, "Survey of query correction for Thai business-oriented information retrieval," in *Proc. 15th Int. Joint Symp. Artif. Intell. Natural Lang. Process. (iSAI-NLP)*, Nov. 2020, pp. 1–6.
[15] H. Zhao, D. Cai, Y. Xin, Y. Wang, and Z. Jia, "A hybrid model for Chinese spelling check," *ACM Trans. Asian Low-Resource Lang. Inf. Process.*, vol. 16, no. 3, pp. 1–22, Mar. 2017, doi: 10.1145/3047405.
[16] R. Grundkiewicz and M. Junczys-Dowmunt, "Near human-level performance in grammatical error correction with hybrid machine translation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, New Orleans, LA, USA, 2018, pp. 284–290. [Online]. Available: https://aclanthology.org/N18-2046
[17] F. Stahlberg and S. Kumar, "Synthetic data generation for grammatical error correction with tagged corruption models," in *Proc. 16th Workshop Innov. Use NLP Building Educ. Appl.*, Apr. 2021, pp. 37–47. [Online]. Available: https://aclanthology.org/2021.bea-1.4
[18] S. Rothe, J. Mallinson, E. Malmi, S. Krause, and A. Severyn, "A simple recipe for multilingual grammatical error correction," in *Proc. 59th Annu. Meeting Assoc. Comput. Linguistics, 11th Int. Joint Conf. Natural Lang. Process.*, 2021, pp. 702–707. [Online]. Available: https://aclanthology.org/2021.acl-short.89
[19] E. Malmi, S. Krause, S. Rothe, D. Mirylenka, and A. Severyn, "Encode, tag, realize: High-precision text editing," in *Proc. Conf. Empirical Methods Natural Lang. Process., 9th Int. Joint Conf. Natural Lang. Process. (EMNLP-IJCNLP)*, Hong Kong, 2019, pp. 5054–5065. [Online]. Available: https://aclanthology.org/D19-1510
[20] K. Omelianchuk, V. Atrasevych, A. Chernodub, and O. Skurzhanskyi, "GECToR—Grammatical error correction: Tag, not rewrite," in *Proc. 15th Workshop Innov. Use NLP Building Educ. Appl.*, Seattle, WA, USA, 2020, pp. 163–170. [Online]. Available: https://aclanthology.org/2020.bea-1.16
[21] H. T. Ng, S. M. Wu, T. Briscoe, C. Hadiwinoto, R. H. Susanto, and C. Bryant, "The CoNLL-2014 shared task on grammatical error correction," in *Proc. 18th Conf. Comput. Natural Lang. Learn., Shared Task*, Baltimore, MD, USA, 2014, pp. 1–14. [Online]. Available: https://aclanthology.org/W14-1701
[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.
[23] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, 1997.
[24] K. Murray and D. Chiang, "Correcting length bias in neural machine translation," in *Proc. 3rd Conf. Mach. Transl., Res. Papers*, Brussels, Belgium, 2018, pp. 212–223. [Online]. Available: https://aclanthology.org/W18-6322
[25] W. He, Z. He, H. Wu, and H. Wang, "Improved neural machine translation with SMT features," in *Proc. AAAI Conf. Artif. Intell.*, Feb. 2016, vol. 30, no. 1, pp. 1–7. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/9983
[26] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959, doi: 10.1007/bf01386390.
[27] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," in *Proc. 1st Workshop Neural Mach. Transl.*, Vancouver, BC, USA, 2017, pp. 56–60. [Online]. Available: https://aclanthology.org/W17-3207
[28] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proc. SSST-8, 8th Workshop Syntax, Semantics Struct. Stat. Transl.*, Doha, Qatar, 2014, pp. 103–111. [Online]. Available: https://aclanthology.org/W14-4012
[29] C. Napoles, K. Sakaguchi, M. Post, and J. Tetreault, "GLEU without tuning," 2016, *arXiv:1605.02592*.

ANURUTH LERTPIYA received the B.Eng. and M.Eng. degrees in computer engineering from Chulalongkorn University, Bangkok, Thailand, in 2018 and 2020, respectively. He joined the Natural Language Processing Team, Kasikorn Labs (KLabs) Company Ltd., Kasikorn Business-Technology Group (KBTG), which mostly research based on Thai language. His research interests include natural language generation, information extraction, and named-entity recognition.

TAWUNRAT CHALOTHORN received the B.S., M.S., and Ph.D. degrees from the University of Northumbria, Newcastle, U.K., in 2010, 2011, and 2016, respectively. She joined the Natural Language Processing Team, Kasikorn Labs (KLabs) Company Ltd., Kasikorn Business-Technology Group (KBTG), which mostly research based on Thai language. Her research interests include chatbots, social listening, and text analytics.



PAKPOOM BUABTHONG received the B.S. degree from the University of Illinois at Urbana–Champaign, in 2015, and the Ph.D. degree from the California Institute of Technology, in 2021. He is currently a Lecturer of physics with the Department of Science and Technology, Nakhon Ratchasima Rajabhat University. He is also a member of the Natural Language Processing Team, Kasikorn Labs (KLabs) Company Ltd., Kasikorn Business-Technology Group (KBTG). His research interests include graph neural networks for information retrieval and natural language processing for knowledge graph construction.