# Tokenization Repair in the Presence of Spelling Errors

**Hannah Bast, Matthias Hertel, Mostafa M. Mohamed**
Department of Computer Science
University of Freiburg
79110 Freiburg, Germany
{bast,hertelm,amin}@informatik.uni-freiburg.de

## Abstract

We consider the following *tokenization repair problem*: Given a natural language text with any combination of missing or spurious spaces, correct these. Spelling errors can be present, but it's not part of the problem to correct them. For example, given: "*Tispa per isabout token izaionrep air*", compute "*Tis paper is about tokenizaion repair*".

It is tempting to think of this problem as a special case of spelling correction or to treat the two problems together. We make a case that tokenization repair and spelling correction *should* and *can* be treated as separate problems. We investigate a variety of neural models as well as a number of strong baselines. We identify three main ingredients to high-quality tokenization repair: deep language models with a bidirectional component, training the models on text with spelling errors, and making use of the space information already present.

Our best methods can repair all tokenization errors on 97.5% of the correctly spelled test sentences and on 96.0% of the misspelled test sentences. With all spaces removed from the given text (the scenario from previous work), the accuracy falls to 94.5% and 90.1%, respectively. We conduct a detailed error analysis.

## 1 Introduction

Tokenizing a given text into words is the first step in many natural language processing applications, including: search engines, translation services, spell checkers and all kinds of learning tasks performed on text. This tokenization is typically performed by the following simple method or a variant of it: define a set of word characters and take each maximal sequence of word characters as one token.[1] For example, for

This algoritm runs in linear time

a simple such tokenization yields the six words

This, algoritm, runs, in, linear, time. (1)

Note the spelling error in the second word. Spelling correction is not part of tokenization. We come back to this important aspect in Section 1.2.

Missing and spurious spaces are common errors in digital text documents. We refer to the union of both types of errors as *tokenization errors*. Here is a variant of the sentence above with one missing space and one spurious space:

This algor itm runsin linear time (2)

In this paper, we consider the following *tokenization repair* problem: Given a sequence of characters representing a natural language text, with an arbitrary amount of missing and spurious spaces and possibly also with spelling errors, compute the variant of the text with correct spacing. For example, given (2) above, compute (1).

Tokenization repair can be considered as a generalization of the *word segmentation* problem, where the text is given without any space information. Indeed, we also evaluate our methods on this special case in Section 4.

### 1.1 Sources of tokenization errors

The widely used PDF format stores no information about spaces. Text is represented as characters with bounding boxes. When extracting text from PDF documents, the space positions must be inferred from the distance between the characters' bounding boxes. This is a non-trivial and error-prone task (Bast and Korzen, 2017).

Tokenization errors are also typical in texts that are digitized by optical character recognition (OCR) techniques. For example, tokenization errors are known to be frequent in the ACL anthology corpus (Nastase and Hitschler, 2018) and in digitized newspapers (Soni et al., 2019; Adesam et al., 2019). There is large body of research on OCR error correction (Kumar, 2016). However, not all methods can deal with tokenization errors, and it is stated in Hämäläinen and Hengchen (2019) that:

"A limitation of our approach is that it cannot do word segmentation in case multiple words have

[1] Some languages, like Chinese, do not use word delimiters like spaces; they are out of scope for this paper.

been merged together as a result of the OCR process. However, this problem is complex enough on its own right to deserve an entire publication of its own and is thus not in the scope of our paper."

Tokenization errors can also be found in human-typed texts. The fraction of these errors among misspellings was found to be 15% in Kukich (1992).

Tokenization errors degrade the performance of any natural language processing (NLP) system, if it does not account for them. A search engine will not find "*algorithm*" in a document containing "*algo rithm*". Syntax parsers and word labelers will not give the correct results if a word is split into multiple words, or multiple words merged into one. A text classifier based on word statistics or word vector representations will fail to retrieve statistics or vector representations for wrongly tokenized words, which can result in wrong classifications.

### 1.2 Tokenization and Spelling Correction

It is tempting to regard tokenization repair as a special case of spelling correction or to try to solve both problems simultaneously. We next argue that it makes sense to consider the two problems separately and the rest of the paper provides evidence that they *can* be considered separately.

All the approaches we consider in this paper can be adapted to correct not only tokenization errors but also spelling errors. For example, consider the best approach from previous work: a left-to-right character-based language model (that predicts the next character from the previous characters) combined with a standard beam search (which at each point maintains the $b$ best corrections of the sequence thus far); see Section 3.1. This method can achieve good (not great) results for tokenization *and* spelling correction, but only with a very large $b$ and a correspondingly impractically large running time. The reason is that when we cannot be sure about the spaces, sub-sequences can be misspellings of many words. For example, even when allowing only one spelling error per token, "*o the*" might be a misspelling of "*of the*", "*other*", "*oath*", etc. We come back to this issue in Section 4.

It is therefore not surprising that existing spelling correction programs first tokenize the text (using the simplistic approach described earlier) and then correct the individual words. Some spell checkers also repair tokenization errors, but only to a limited extent. In Section 4, we evaluate the tokenization repair capabilities of a popular word segmentation tool and a popular spell checker. For spell checkers that disregard tokenization errors, tokenization repair will improve the quality of the correction.

However, our evaluation shows that when separating the two problems, tokenization repair *must* consider that the text might have spelling errors. The reader may wonder how it is possible to consider spelling errors during tokenization without fixing them in the first place. It is indeed one of the insights from our paper that this is possible. In a nutshell, models that are not aware of spelling errors have a strong tendency to end a word after a spelling error (because no known continuation exists) or to wrongly merge misspelled words into a word from the dictionary. We will discuss this in more detail in Section 4.7.

### 1.3 Contributions

We consider these as our main contributions:

- We investigate the tokenization repair problem in depth, providing several new insights, in particular on: the complex interaction with spelling errors, taking advantage of already existing spaces and the importance of bidirectional models.
- We show that it is crucial that the language models are trained on text with spelling errors and that the quality of the predictions does not deteriorate much on test sequences without spelling errors. This aspect was either disregarded in previous work or it played only a secondary role.
- We point out the difficulties of using a bidirectional model for tasks that involve changing the sequence, and we show how to overcome them. In previous work, forward models combined with a beam search gave the best results.
- A crucial component of our best methods are so-called penalties for inserting or deleting a space. They are learned from the data and no hyperparameter tuning is required. Previous work removes all existing spaces from the text in advance and thus cannot make use of any correct tokenization information that is already there.
- We provide an extensive evaluation on text with and without spelling errors, various amounts of spacing errors, and with various baselines, including a commercial and an open-source product. We also conduct a detailed error analysis.
- We make our code, data, benchmarks and trained models publicly available under `https://github.com/ad-freiburg/tokenization-repair`. It includes a Docker setup that allows an easy replica-

tion of our results, and a web application to try out our methods.

## 2 Related Work

A beam search with neural and character n-gram language models is used for word segmentation in Doval and Gómez-Rodríguez (2019). In Section 4.7, we evaluate our own implementation of this approach on our benchmark and we also compare against their results on their benchmark. We improve on their approach in several respects: integrating a bidirectional model (which is not trivial), considering the given spaces in the input (they remove all spaces), and explicitly considering typos (they test their approach on tweets, but do not explicitly handle typos).

A beam search with a word bigram language model, instead of a character-based language model, is used in Mikša et al. (2010) to correct missing spaces in Croatian texts that were digitized by OCR.

Tokenization repair on the ACL anthology corpus is done in Nastase and Hitschler (2018) as a neural machine translation model translating from the sequence without spaces to the sequence with spaces. They also remove all spaces from the input text, thus discarding valuable information. Unfortunately, the materials and information in their paper were not sufficient to evaluate their approach in our setting or our approach in their setting. Nevertheless, in Section 4.7, we compare our results against theirs as good as we can.

In Soni et al. (2019), $n$-gram statistics are used to determine when to split an out-of-vocabulary token. It was shown that the context provided by the $n$-grams improves the results. By using neural language models, we extend the scope of this context beyond the boundaries of $n$-gram models.

## 3 Approach

Our approaches are based on deep character-based models, unidirectional and bidirectional. The unidirectional models are combined with a beam search. The bidirectional models can either repair a sequence directly or be combined with the unidirectional models.

### 3.1 Character-based models

We model the strings as sequences of one-hot encoded characters, where we use the 200 most frequent characters, while replacing the others by a special character UNK for unknown characters. Sentences are appended with start and end of sentence special characters (SOS and EOS).

#### 3.1.1 Unidirectional language models

Character-based language models estimate the probability of a string to occur in some language based on the probabilities of the individual characters in the string, using one of the following:

- $p_f(s|c_b)$ is the probability that a character $s$ occurs after a context string $c_b$.
- $p_b(s|c_a)$ is the probability that a character $s$ occurs before a context string $c_a$.

For example, given the string "*The algorithm runs in linear time.*", if the character $s$ is the underlined '*r*', then $c_b$ = "*The algorithm* " and $c_a$ = "*uns in linear time.*".

Following Graves (2013), we implement these models as recurrent neural networks, using LSTM cells. We adapt an architecture consisting of an LSTM cell of 1024 units, followed by a dense layer (with 1024 units and $ReLU$ activation function), then a softmax output layer for character classification. This architecture consists of 6,287,563 trainable parameters, which are trained using categorical cross entropy as a loss function. Two separate models with this architecture are implemented to process a sequence forwards to predict $p_f(s|c_b)$ or backwards to predict $p_b(s|c_a)$, using the same principle.

#### 3.1.2 Bidirectional sequence labeling model

We utilize a bidirectional model that predicts the probability of having a space before a given character when the whole sequence of non-space characters is given as input. We adapt an architecture consisting of a bidirectional LSTM cell of 1024 units, followed by a dense layer (with 1024 units and $ReLU$ activation function), then a sigmoid output for space classification. This architecture consists of 12,158,980 trainable parameters, which are trained using binary cross entropy as a loss function.

The model can repair a string by estimating the space probability $p^{bi}_{␣,i}$ at every position $i$ in the sequence without spaces. It uses two thresholds $T_{ins}$ and $T_{del}$ and inserts a space when $p^{bi}_{␣,i} > T_{ins}$ and deletes a space when $p^{bi}_{␣,i} < T_{del}$. The values of $T_{ins}$ and $T_{del}$ are chosen such that the F-score (defined in Section 4.5) is optimized on a small training set of pairs of input sequences with corresponding ground truth sequences.

### 3.2 Beam search

Beam search is a search algorithm similar to breadth-first search, but instead of maintaining all search states at a given level, it maintains only the best $b$ states, which correspond to an estimation of the best $b$ partial solutions (Medress et al., 1977).

**Correction procedure:** Given a mistokenized string $Q$, with its corresponding sequence of $m$ non-space characters $T$ (we refer to $T_i$ to be aligned with $Q_j$), the procedure executes beam search for $m$ levels. At level $i$, given a partial solution's search state $(S_{i-1}, R_{i-1})$ of accumulated score and solution string respectively, we extend it based on two candidates:

1. Adding $T_i$ without space, which results in:
$$S_i = S_{i-1} - \log p_f(T_i|R_{i-1}) + P_{del}$$
$$R_i = R_{i-1}T_i$$
2. Adding a space before $T_i$, which results in:
$$S'_i = S_{i-1} - \log p_{␣}(T_i|R_{i-1}) + P_{ins}$$
$$R'_i = R_{i-1}{␣}T_i$$

Where $p_{␣}(T_i|R_{i-1})$ is the probability of adding a space before $T_i$, given by:
$$p_{␣}(T_i|R_{i-1}) = p_f({␣}|R_{i-1}) \cdot p_f(T_i|R_{i-1}{␣})$$
$P_{del}$ and $P_{ins}$ are non-negative penalties that are used only when the introduced extension is not originally in $Q$, otherwise they are assumed to be 0. In other words, $P_{ins}$ is used when $Q_{j-1} \neq {␣}$ and $P_{del}$ is used when $Q_{j-1} = {␣}$. They aim to regularize the effect of making too many edits.

The final solution $R = (R_1, ..., R_{|R|})$ is the estimated corrected sequence of lowest penalized negative log-likelihood score (highest probability):
$$-\log p(R) + n_{ins}P_{ins} + n_{del}P_{del}$$
where $n_{ins}$ is the number of space insertions and $n_{del}$ is the number of space deletions, and:
$$p(R) = \prod_{i=1}^{|R|} p_f(R_i|R_1, \cdots, R_{i-1})$$
During execution, we additionally keep the internal states of the LSTM cells with the beam search's states $(S_i, R_i)$, in order not to recompute the candidate probabilities using the whole previous context, but rather predicting them using a constant number of operations. Consequently, the time complexity is $\mathcal{O}(|Q| \cdot b)$, because we process $2b$ candidates at $m$ levels ($m \leq |Q|$) using a constant number of LSTM predictions. We use a beam size $b = 5$ in our implementation. As a result, the algorithm runs in linear time.

**Variants:** We utilize the following variants of the beam search (BS) procedure:

1. BS fw: Left-to-right via the forward model.
2. BS bw: Right-to-left via the backward model.
3. 2-pass BS: Double pass using 1. then 2.
4. BS bidir: Left-to-right combining the forward model with the bidirectional model introduced in section 3.1.2. For this variant, the update formulas of the scores are the following:

$$S_i = S_{i-1} - \log(p_f(T_i|R_{i-1}) \cdot (1 - p^{bi}_{␣,i})) + P_{del}$$
$$S'_i = S_{i-1} - \log(p_{␣}(T_i|R_{i-1}) \cdot p^{bi}_{␣,i}) + P_{ins}$$

**Penalty optimization:** The penalties $P_{ins}$ and $P_{del}$ are set using a small training set of pairs of input sequences with corresponding ground truth sequences. We simulate a beam search under the assumption that the left context is always predicted correctly, and that the procedure takes a decision after processing the next two characters. Given the ground truth string $Q$, for every non-space character $Q_i$ and its previous non-space character $Q_j$, the space probability $p_s$ and non-space probability $p_n$ are computed:
$$p_s = p_{␣}(Q_i|Q_{1:j}) \cdot p_f(Q_{i+1}|Q_{1:j}{␣}Q_i)$$
$$p_n = p_f(Q_i|Q_{1:j}) \cdot p_f(Q_{i+1}|Q_{1:j}Q_i)$$
If the space is present in the input sequence, the scores $S$ and $S'$ of the candidate sequences without and with space are:
$$S = -\log p_n + P_{del}$$
$$S' = -\log p_s$$
The space gets deleted if $P_{del} < \log p_n - \log p_s$. Furthermore, If the space is not present in the input sequence, the scores are:
$$S = -\log p_n$$
$$S' = -\log p_s + P_{ins}$$
The space gets inserted if $P_{ins} < \log p_s - \log p_n$. Depending on whether the space is present in the ground truth sequence, the corresponding edit is a true positive or false positive. Finally, the F-score is evaluated for every penalty value that makes an example flip, and the optimal penalties are chosen.

The penalty optimization for the backward pass is analogous, but using the predictions of the forward pass on the penalty training set as inputs, and minimizing the sum of false positives and false negatives as a proxy of maximizing the overall F-score.

### 3.3 Baseline approaches

We utilize baselines from three different classes: greedy, dynamic programming and commercial.

### 3.3.1 Greedy bigram model

We tokenize the training data with the NLTK tokenizer (Bird et al., 2009) and count unigram and bigram frequencies. The greedy corrector processes a sequence from left to right. Two tokens get merged if the merged unigram is more frequent than the bigram. A token gets split into two, if the bigram frequency of the split is greater than the token's unigram frequency. A rule-based postprocessing deals with spaces before and after punctuation.

### 3.3.2 Dynamic programming bigram model

This baseline is a Viterbi algorithm (Viterbi, 1967) with a word bigram model (Jurafsky and Martin, 2009). First, all possible words (substrings of length $\leq$ 20 with non-zero unigram frequency) in the sequence without spaces are located. The states of the Viterbi algorithm are equivalent to the words. A transition between two states is possible if the next word starts at the end of the first word. State transition probabilities are determined by a combination of a unigram and a bigram model:

$$p(w_{i+1}|w_i) = \frac{1}{2}(p_{bi}(w_{i+1}|w_i) + p_{uni}(w_{i+1}))$$

The output is the most likely segmentation of the sequence without spaces into words.

### 3.3.3 Wordsegment

Wordsegment is an open-source library, based on Halpern (2015), that uses precomputed frequencies of unigrams and bigrams to segment words.

### 3.3.4 Google

To compete with a commercial spell checker, we copy the erroneous sentences into a Google document[2] and manually apply all suggested edits that comprise splitting or merging words. When suggestions were ambiguous with respect to the space edits, or close to the correct solution (for example, the suggestion "*gave up*" for the input "*gavemeup*"), we decided in favour of the spell checker.

## 4 Evaluation

### 4.1 Dataset

We use Wikipedia as a text corpus to evaluate our approaches. We extracted the articles from the Wikipedia dump of June 20, 2019[3] using WikiExtractor (Attardi, 2017). The articles were divided into development and test sets containing 10,000 randomly selected articles each, and the remaining as a training set. The articles were split into paragraphs, and development and test paragraphs further split into sentences with the NLTK sentence segmenter (Bird et al., 2009). All sequences were stripped from leading and trailing spaces, and empty sequences removed. All types of spaces, like non-breaking or thin spaces, were replaced by regular spaces. We excluded sentences matching the regular expression:
“ `[.,;]( |$)|<|>|\"\"|\(\)| ′ |\([,;]`“
These are incomplete sentences or sentences containing Markup, which occured due to incomplete extraction by WikiExtractor or wrong sentence splits. One sequence was chosen randomly per development article and test article. We thus obtain 43,103,197 sequences for training, 10,000 for development and 10,000 for test. We also select 10,000 sequences from the training set for optimizing the $T_{ins}$, $T_{del}$, $P_{ins}$ and $P_{del}$ described in Sections 3.1.2 and 3.2.

### 4.2 Typo noise induction

To test whether our approaches work in the presence of misspellings, we induce randomized noise into the ground truth sequences. Each token gets misspelled with probability $p_{\text{spell}}$. A misspelling is one of the following operations: insertion of a lowercase character, deletion of an alphabetic character, replacement of an alphabetic character by a lowercase character, swap of two neighboring alphabetic characters.

### 4.3 Tokenization error induction

To create test benchmarks, we introduce random tokenization errors into the sequences. A parameter $p$ controls the tokenization error rate. Let $N_t$ be the number of tokens in the dataset, $N_{ms}$ the number of spaces and $N_{es}$ the number of neighboring non-space character pairs. We remove each space in a sequence with probability $\frac{p \cdot N_t}{2 \cdot N_{ms}}$ and insert a space between every neighboring non-space character pair with probability $\frac{p \cdot N_t}{2 \cdot N_{es}}$. This results in a balanced number of inserted and deleted spaces. The expected number of errors is equal to $p \cdot N_t$.

### 4.4 Benchmarks

We prepare ground truth sequences with varying typo noise levels $p_{\text{spell}} \in \{0, 0.1\}$ for the penalty training, development and test sets. For each of the aforementioned sets and typo noise level, we create three benchmarks with varying amounts of

[2] https://docs.google.com, accessed April 10, 2020

[3] https://dumps.wikimedia.org/enwiki

tokenization errors, by applying the tokenization error induction with $p \in \{0.1, 1\}$ , in addition to a benchmark with no spaces at all.

### 4.5 Metrics for tokenization repair

Given a corrupt text $C$, a ground truth text $T$ and a predicted text $P$, we can frame a repair algorithm as a classifier which predicts a set of edit operations that ideally would transform $C$ into $T$. We use two metrics for the evaluation: F-score and sequence accuracy.

**F-score:** We define $\text{edits}(A, B)$ as the edit operations that transform $A$ into $B$. An edit operation is either the insertion or deletion of a space. If we let $\mathcal{C} = \text{edits}(C, T)$ be the ground truth edit operations and $\mathcal{P} = \text{edits}(C, P)$ the predicted edit operations, the number of true positives is $\text{TP} = |\mathcal{C} \cap \mathcal{P}|$, the number of false positives is $\text{FP} = |\mathcal{P} \setminus \mathcal{C}|$ and the number of false negatives is $\text{FN} = |\mathcal{C} \setminus \mathcal{P}|$. The binary classification metric F-score is the harmonic mean of precision and recall, and is computed with:

$$F(T, C, P) = \frac{2 \cdot \text{TP}}{2 \cdot \text{TP} + \text{FP} + \text{FN}}$$

**Sequence accuracy:** The sequence accuracy is the fraction of sequences that are completely corrected ($P = T$). A sequence is correctly predicted if and only if the set of predicted operations is exactly equal to the set of ground truth operations for that sequence.

Before evaluating on the benchmarks with spelling errors, we remove from the predicted sequences, ground truth sequences and wrongly tokenized sequences the non-space characters that were introduced as a spelling error, and resulting multi-spaces. This allows for ambiguous cases like "*hellox world*", "*hello xworld*" or "*hello x world*" to be accepted as multiple correct solutions for the same sentence (after removing the "*x*" and the multi-space, all three sequences become "*hello world*").

### 4.6 Models training

We train forward models, backward models and bidirectional models. An evaluation of the language models is given in Table 1.

Additionally, typo-robust versions of these models are trained, where the training happens on misspelled sequences. In this setting, the misspelled sequences are constructed by the mechanism explained in section 4.2, with $p_{\text{spell}} = 0.2$.

| Model | Acc | Top-5 | CCE | ␣ F-score |
|---|---|---|---|---|
| forward | 71.30 | 90.47 | 96.3 | 89.20 |
| backward | 70.81 | 90.56 | 97.4 | 95.84 |

Table 1: Results in percentages of accuracy, top-5 accuracy, categorical cross entropy and spaces F-score for the language models on the development set.

All six models are trained for one epoch on the training data, using a GeForce Titan X GPU. Training a unidirectional model takes 27 hours, while the bidirectional model takes 64 hours. The training is performed using the Adam optimization algorithm (Kingma and Ba, 2015), with learning rate $\alpha = 0.001$, and mini-batch size 128. The sequences were cut after 256 characters, while shorter sequences were padded with EOS symbols that got masked in the loss function. The models are implemented using TensorFlow (Abadi et al., 2015).

For the benchmarks with $p \in \{0.1, 1\}$ we optimize the penalties $P_{ins}$ and $P_{del}$ separately for all approaches. For the benchmarks with all spaces removed, we set $P_{ins} = P_{del} = 0$.

### 4.7 Results and discussion

Our main results are shown in Table 2, which provides F-scores and sequence accuracies for all the approaches we implemented, on all benchmarks. We use a beam size of $b = 5$ for all beam search approaches; increasing this to $b = 10$ has shown only minimal improvements while doubling the running time. We first discuss the main takeaways from Table 2. Along with that and afterwards, we discuss the other tables (comparison to previous work, error analysis, running time).

The first takeaway from Table 2 is that the forward beam search combined with a bidirectional model (*BS bidir*) is the clear winner in *all* scenarios (with or without typos, with any amount of tokenization errors). In particular, it beats all the baseline methods by a wide margin, as well as the unidirectional beam search methods. The best method from previous work (Doval and Gómez-Rodríguez, 2019) is an instance of unidirectional beam search. In Table 5, we explicitly compare our implementation of their method to their implementation on their benchmark. Our implementation of their method has better accuracy; we suspect because we use more training data. The results of their method on their benchmark is better than on our benchmark; we suspect because their benchmark (news articles) uses cleaner language and is

| | F-score | | | | | | sequence accuracy | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | no typos | | | 10 % typos | | | no typos | | | 10 % typos | | |
| approach | 10 % | 100 % | no ␣ | 10 % | 100 % | no ␣ | 10 % | 100 % | no ␣ | 10 % | 100 % | no ␣ |
| do nothing | - | - | - | - | - | - | 34.86 | 3.03 | 4.10 | 34.89 | 3.22 | 4.10 |
| greedy | 85.47 | 56.21 | 20.46 | 68.73 | 53.68 | 19.93 | 76.69 | 9.96 | 12.96 | 46.43 | 8.53 | 11.42 |
| bigram DP | 92.48 | 99.20 | 99.16 | 68.24 | 95.56 | 95.39 | 86.16 | 86.16 | 86.16 | 45.61 | 45.61 | 45.61 |
| wordsegment | 58.90 | 93.48 | 92.97 | 45.94 | 89.48 | 88.51 | 41.12 | 41.12 | 41.12 | 22.37 | 22.37 | 22.37 |
| Google [1] | 89.78 | 65.43 | 18.40 | 82.95 | 53.14 | 15.14 | 82.00 | 15.00 | 15.00 | 73.00 | 10.00 | 13.00 |
| bidirectional | 98.86 | 99.72 | 99.67 | 93.38 | 98.27 | 98.01 | 97.05 | 93.80 | 93.15 | 84.33 | 69.27 | 67.62 |
| bidir. robust | 98.67 | 99.65 | 99.59 | 97.42 | 99.30 | 99.20 | 96.52 | 92.23 | 91.84 | 93.33 | 84.90 | 84.17 |
| BS fw | 98.68 | 99.66 | 99.61 | 93.86 | 97.96 | 97.53 | 96.59 | 92.86 | 92.22 | 85.29 | 67.71 | 65.13 |
| BS bw | 98.61 | 99.68 | 99.59 | 93.33 | 97.87 | 97.49 | 96.41 | 93.05 | 92.25 | 84.11 | 67.00 | 64.99 |
| BS fw robust | 98.56 | 99.61 | 99.54 | 97.98 | 99.36 | 99.23 | 96.39 | 91.65 | 90.88 | 94.98 | 87.67 | 86.16 |
| BS bw robust | 98.54 | 99.59 | 99.51 | 98.14 | 99.35 | 99.21 | 96.31 | 91.62 | 90.72 | 95.32 | 87.34 | 85.85 |
| 2-pass BS | 98.70 | 99.71 | 99.66 | 94.01 | 98.30 | 98.01 | 96.70 | 93.68 | 93.17 | 85.77 | 71.73 | 69.99 |
| 2-pass BS robust | 98.58 | 99.65 | 99.59 | 98.07 | 99.46 | 99.37 | 96.41 | 92.48 | 91.88 | 95.27 | 89.36 | 88.49 |
| BS bidir | **99.01** | **99.77** | **99.74** | 95.66 | 98.69 | 98.44 | **97.49** | **94.79** | **94.47** | 89.23 | 76.18 | 74.07 |
| BS bidir robust | 98.85 | 99.73 | 99.68 | **98.39** | **99.57** | **99.49** | 97.11 | 93.92 | 93.54 | **96.04** | **91.05** | **90.11** |

Table 2: F-scores and sequence accuracy percentages for all models and all benchmarks. The benchmarks either contain no typos or 10% chance of introducing a typo. The tokenization errors either have $p$ as $0.1$, $1.0$ or all spaces removed. The F-scores are micro-averaged across the test set. The difference of the sequence accuracy of the best approach compared with all other approaches is statistically strongly significant ($p < 0.01$ with a paired two-sided randomization test). [1] Google's tokenization repair was manually evaluated on 100 development sequences.

| Error type | w/o | with | Prediction | Ground T. |
|---|---|---|---|---|
| Compound | 29% | 24% | box set | boxset |
| Foreign | 21% | 18% | De Clercq | DeClercq |
| Entities | 19% | 18% | First Energy | FirstEnergy |
| Typo ambig. | - | 14% | be a | bea~~r~~ |
| Punctuation | 19% | 12% | (March) | ( March ) |
| Measures | 8% | 6% | 10 m | 10m |
| Abbrev. | 2% | 4% | K. K. | K.K. |
| Unk.alphabet | 2% | 2% | Rev-erb $\alpha$ | Rev-erb$\alpha$ |

Table 3: Error analysis with relative frequencies and examples for our best approaches on sentences with 0.1 tokenization error rate, without and with 10% typos.

thus simpler. In Section 2, we also mention the recent work from Nastase and Hitschler (2018). Unfortunately, their dataset has no ground truth, so we cannot evaluate our methods on it. By inspecting some of their processed files, we find that their method introduces many new tokenization errors.

It might look obvious that the bidirectional methods are the best, but as we explain next it is not obvious. This might also be the reason why previous work used unidirectional methods. A unidirectional model has the advantage that the tokenization errors are incrementally fixed from left to right (or from right to left in a backward model), so that the language model predictions can be based on text that is (almost) free from such errors. However, the text after the current position has not yet been repaired, so that predictions from the other direction are based on text with tokenization errors. Using these predictions actually *deteriorates* the quality of the unidirectional methods. Our trick was to combine a unidirectional model that makes use of the space information with a bidirectional model that disregards all space information and thus does not have the aforementioned problem. Our second-best approach (*2-pass BS*) overcame this by employing two separate unidirectional passes.

The second takeaway from Table 2 is that training the models on text with spelling errors (the methods named "robust") is crucial for a good tokenization quality. Conversely, when using these models on text without spelling errors, the quality is very close to that of the models trained on text without spelling errors. A detailed error analysis of the non-robust methods shows that these methods have a strong tendency to split words with a typo because there is no meaningful continuation (e.g., "*unwnted pregnancies*" is wrongly repaired to "*unw nted pregnancies*") and to wrongly merge misspelled words when they happen to form a correct word (e.g., if "*as well*" is mistyped as "*s well*", it is wrongly repaired to "*swell*").

The third takeaway from Table 2 is that tokenization repair is much harder when there are many tokenization errors. This is not surprising, yet previous work chose to remove all spaces from the text. This simplifies the approaches somewhat (one only has to predict space insertions, no space deletions), but the price is much worse results when there were actually only few tokenization errors. Our approach regulates this via the two penalties $P_{ins}$ and $P_{del}$ described in Section 3.2, which are optimized on a separate small penalty training set. We also conducted a sensitivity analysis, which shows that our results are very robust against small changes in these penalties. When $P_{ins} = P_{del} = 0$, spaces in the given text are ignored (which is equivalent to removing all spaces from the text). This can also be used as a default setting when there are many tokenization errors.

Also not surprisingly, text with spelling errors is more difficult to repair. On text with spelling errors and no spaces, the sequence accuracy of our best method drops to 90.1% (which is still very good compared to the 65.1% of the *BS fw* method though). The reason is that the combination of spelling and tokenization errors can create ambiguous situations, which are very hard to fix. For example, deleting the 'r' in "*The stems bear single flower heads*" is fixed as "*The stems be a single flower heads*".

Table 3 shows the result of a detailed error analysis of our best method. In particular, we found that the beam search with a bidirectional model does a much better job on correctly tokenizing compound words and foreign words. In contrast, a method such as *2-pass BS* merges "*off season*", and splits "*northeastern*" and "*Tornakápolna*". However, some compound words are difficult for all models. For example, "*Waikiki BeachBoys*" is a baseball team and "*The Beach Boys*" is a rock band. Fixing these correctly would require explicit knowledge of these entity names. There are also several inconsistencies in the ground truth, for example regarding spacing near commas and full stops.

Table 4 shows the average running time for our main methods on a Nvidia GTX 1060 GPU. We also implemented a variant of *BS fw* that uses its predictions to fix both tokenization and spelling errors. The results are comparable to those of the method trained on text with spelling errors. However, the price is a *much* larger beam size ($b = 100$ instead of $b = 5$) because of the exploding number of possible interpretations. Consequently, the running time of this approach increases by a factor of more than 20, which makes it impractical for actual applications.

| | bidir. | BS | 2-pass BS | BS bidir |
|---|---|---|---|---|
| runtime ($s$/KB) | 0.45 | 2.03 | 3.95 | 2.61 |

Table 4: Runtimes averaged on all benchmarks.

| Approach | F-score | Seq.acc. |
|---|---|---|
| Doval and Gómez-Rodríguez (2019) | 99.63 | 92.2 |
| BS fw | 99.81 | 95.4 |
| BS bidir | **99.84** | **96.4** |

Table 5: Evaluation (percentages) on the English benchmark by Doval and Gómez-Rodríguez (2019).

## 5 Conclusion

We introduced *tokenization repair* as an important pre-processing step for all kinds of NLP applications. Our best method uses a deep character-based language model, combining a unidirectional beam search with a bidirectional model. It is crucial that the unidirectional model takes the existing space information into account, while the bidirectional disregards it. It is also crucial that the models are trained on text with spelling errors.

Our best approaches improve significantly over previous work and several strong baselines. There is still room for improvement, especially in a scenario with spelling errors and many tokenization errors. However, we show that the remaining errors are hard, with many ambiguous situations.

We paid attention to practical running times: all our methods have linear complexity, using a few seconds per 1,000 characters. However, there is also room for improvement here and it would be interesting to investigate whether a similar quality can be achieved with much faster algorithms.

We made a case that the problem should and can be separated from that of spelling correction. In particular, our study makes it clear that tokenization repair is an interesting, challenging and practically relevant problem on its own. That being said, it would be interesting to follow up on this study and investigate how to best attack spelling correction *after* tokenization repair, using methods similar to or inspired by the ones developed in this paper. One of the main challenges will be the huge search space. Reducing it sufficiently will be key for methods with a practical running time.

## References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Yvonne Adesam, Dana Dannélls, and Nina Tahmasebi. 2019. Exploring the quality of the digital historical newspaper archive KubHist. In *Proceedings of the Digital Humanities in the Nordic Countries 4th Conference, Copenhagen, Denmark, March 5-8, 2019*, volume 2364 of *CEUR Workshop Proceedings*, pages 9–17.

Giuseppe Attardi. 2017. WikiExtractor: A tool for extracting plain text from Wikipedia dumps. `https://github.com/attardi/wikiextractor`. Accessed: 2019-06-20.

Hannah Bast and Claudius Korzen. 2017. A benchmark and evaluation for text extraction from PDF. In *2017 ACM/IEEE Joint Conference on Digital Libraries, JCDL 2017, Toronto, ON, Canada, June 19-23, 2017*, pages 99–108. IEEE Computer Society.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python*. O'Reilly Media.

Yerai Doval and Carlos Gómez-Rodríguez. 2019. Comparing neural- and n-gram-based language models for word segmentation. *Journal of the Association for Information Science and Technology*, 70(2):187–197.

Alex Graves. 2013. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

Orit Halpern. 2015. *Beautiful data: A history of vision and reason since 1945*. Duke University Press.

Mika Hämäläinen and Simon Hengchen. 2019. From the paft to the fiiture: a fully automatic NMT and word embeddings method for OCR post-correction. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing, RANLP 2019, Varna, Bulgaria, September 2-4, 2019*, pages 431–436. INCOMA Ltd.

Dan Jurafsky and James H. Martin. 2009. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Karen Kukich. 1992. Spelling correction for telecommunications network for the deaf. *Communications of the ACM*, 35(5):80–90.

Atul Kumar. 2016. A survey on various OCR errors. *International Journal of Computer Applications*, 143(4):8–10.

Mark F. Medress, Franklin S Cooper, Jim W. Forgie, CC Green, Dennis H. Klatt, Michael H. O'Malley, Edward P Neuburg, Allen Newell, DR Reddy, B Ritea, et al. 1977. Speech understanding systems: Report of a steering committee. *Artificial Intelligence*, 9(3):307–316.

Mladen Mikša, Jan Šnajder, and Bojana Dalbelo Bašic. 2010. Correcting word merge errors in Croatian texts. *The Seventh International Conference on Formal Approaches to South Slavic and Balkan Languages*, pages 67–76.

Vivi Nastase and Julian Hitschler. 2018. Correction of OCR word segmentation errors in articles from the ACL collection through neural machine translation methods. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation, LREC 2018, Miyazaki, Japan, May 7-12, 2018*. European Language Resources Association (ELRA).

Sandeep Soni, Lauren F. Klein, and Jacob Eisenstein. 2019. Correcting whitespace errors in digitized historical texts. In *Proceedings of the 3rd Joint SIGHUM Workshop on Computational Linguistics for Cultural Heritage, Social Sciences, Humanities and Literature, LaTeCH@NAACL-HLT 2019, Minneapolis, MN, USA, June 7, 2019*, pages 98–103. Association for Computational Linguistics.

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