OCR Correction of Le Temps

Master Thesis Report

Submitted by:
Laurent Valette (laurent.valette@epfl.ch)

Under the supervision of:
Prof. Frédéric Kaplan
Maud Ehrmann

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Digital Humanities Laboratory
École Polytechnique Fédérale de Lausanne
January 2017

Abstract
Optical Character Recognition (OCR) is a technique to convert textual materials into texts encoded in machine-readable formats. It is an important step when digitizing documents, as it enables to execute other processes, like data-mining or information retrieval. However, OCR algorithms are not always accurate and the resulting texts can contain errors, due to old fonts or degraded materials. In this paper, we analyze how automatic error detection and correction techniques can improve the quality of a corpus of two Swiss French newspapers. Our dictionary based error detection model is capable of detecting 40 % of the errors at a precision of 88 %. Our error correction algorithm reaches a precision of 21 %.

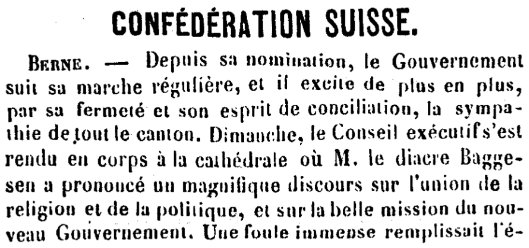
1 Introduction

The *Le Temps* corpus[1] is composed of articles published by the Swiss newspapers *La Gazette de Lausanne* and *Le Journal de Genève.*

The first issue for the *Journal de Genève* dates from January 1826 and the last issue dates from February 1998. The collection includes 550 000 printed pages representing 2 000 000 articles. The articles from the *Gazette de Lausanne* cover a period from February 1798 to February 1998. There are 450 000 printed pages, equivalent to 1 700 000 articles.

The text of the articles has been numerically acquired through digitization of the printed issues and Optical Character Recognition (OCR). However, due to old fonts or degraded materials, the OCR algorithms introduced transcription errors in the text, as illustrated in Figures 1 and 2.

In this context, the objectives of this project were to design, implement and evaluate automated OCR correction techniques, as well as to quantify the noise contained in the corpus. The first part, correcting OCR, can be split into two main tasks: error *detection* and error *correction.* The goal of error detection is to determine if words were correctly recognized or not. Error correction consists in generating and ranking candidates to replace misrecognized words.


CONFÉDÉRATION SUISSE.

Berne. — Depuis sa nomination, le Gouvernement suit sa marche régulière, et il excite de plus en plus, par sa fermeté et son esprit de conciliation, la sympathie de tout le canton. Dimanche, le Conseil exécutif s'est rendu en corps à la cathédrale où M. le diacre Baggesen a prononcé un magnifique discours sur l'union de la religion et de la politique, et sur la belle mission du nouveau Gouvernement. Une foule immense remplissait l'é-

Figure 1: Extract of a scanned article from the *Journal de Genève* (1850).

[1] http://www.letempsarchives.ch/

```
CONFÉDÉRATION SUISSE. BERNE. - Depuis sa
nomination, le Gouvernement suii sa marche
régulière, et il excite de plus en plus,
par sa fermeté et son esprit de conciliation,
la sympathie de. lout le canton. Dimanche,
le Conseil exécutifs'est rendu eu corps a la
cathédrale où M. le diacre Baggesen a prononcé
un magnifique discours sur l'union île la
religion et de lu politique, et sur la belle
mission du nouveau Gouvernement. [...]
```

Figure 2: Text of the article shown in Figure 1 as recognized by the OCR algorithm. Errors have been manually highlighted.

Automated error detection and correction are quite challenging tasks, as a high precision needs to be reached for both tasks. Indeed, since no humans are involved in the process, if the results from the error detection are not accurate enough, then correct words will be replaced during the error correction, as if they were misrecognized words. Similarly, the error correction process has to generate good candidates, but it also has to be extremely accurate at ranking the candidates. Indeed, it is not enough that the correct candidate appear with a high score in the list of suggestions; the correct candidate should have *the highest* score, due to the automated nature of this process. In contrast, *interactive spell checkers* can tolerate a lower accuracy, since a person will ultimately decide if a word is actually an error and will pick a correction from a list, if needed.

For the task of detecting errors, we tested three different approaches. The first approach relies on word lists, *lexicons*, to determine if a word was correctly recognized. The second approach is based on *n-grams*. N-grams are a type of language models: they estimate probabilities of elements — characters or words — given a window of previous elements. They do so by using the statistics for a language, computed over a corpus of texts. The third approach works in a similar way as n-grams, but is based on *neural networks*. That is, the neural networks receive a window of preceding elements and have to predict the probability of the next element. The difference is that, instead of using statistics, the neural networks are trained to reconstruct this probability using supervised learning. In other words, during the learning process, examples of sentences are used to adjust internal parameters so that the output of the neural networks converges to the desired behavior: fitting the probability of the next character or word.

Concerning the error correction, we tackled the generation of candidates using character replacement heuristics, as well as some context-dependent methods. Specifically, we evaluated three models: *word bigrams*, *word embeddings* and a *neural network*. To rank the results generated by those methods, we used a simple weighting scheme and filtered the candidates using the Levenshtein distance, which is a measure of string similarity.

To estimate the amount of noise in the corpus, we manually annotated a number of randomly chosen articles. We found that the average word error rate is 11 %, but there are disparities depending on the newspapers: the *Journal de Genève* is the most noisy. This annotated corpus also provides a *gold standard*, which we used to evaluate the performances of the error detection and correction techniques.

For the task of detecting errors, the system we propose is based on *lexicons*. We excluded the use of *n-grams* because they have a significantly lower detection rate compared to *lexicons*. Some *neural networks* reach good error detection performances, but they suffer from being computationally heavy. Our final error detection model detects 40 % of the true errors and reaches a precision of 88 %.

Concerning the error correction, we found that the quality of the suggestions was extremely sensitive to the choice of parameters. Our error detection algorithm reaches a precision of 21 %. Unfortunately, it also replaces the same fraction of tokens detected as errors with wrongly chosen suggestions. We found that the main cause of the poor performances of the error correction was the generation of candidate corrections.

The report is structured as follows: In Section 2, we review related work in the domain of automated error correction. In Section 3, we detail the three approaches used to detect errors. The error correction, generating and ranking candidates, is detailed in Section 4. In Section 5, we estimate the amount of noise, based on a number of manually annotated articles, and we evaluate the performances of the error detection and correction models. Finally, we propose directions for future work in Section 6.

2 Related Work

There is a lot of literature detailing the many approaches used to tackle the problems of automatic error correction and OCR correction. Kukich [15] provides a survey of a lot of the methods used to automatically detect and correct errors. In particular, the paper reviews some of the models we have used, like *dictionary lookup* techniques and *n-grams*, as well as others like *similarity-key* and *probabilistic* techniques, based on transition probabilities and confusion probabilities. The author also emphasizes some of the difficulties we have encountered, like the necessity to reach high precision levels or the problems related to word boundaries.

A traditional approach to automated error correction is to use statistical models. Afli et al. [1] implemented this approach. The idea is to estimate statistics of the language, as well as statistics on how the OCR process misrecognized characters. That is, which characters were replaced by which ones and at which frequency — the table with these informations is called the *confusion matrix*. Those statistics are usually collected on a large corpus for which the corrections are known. Then, Bayesian probabilities are used to determine which correction maximizes the likelihood of a word or a sentence.

An issue with statistical approaches is that computing the confusion matrix requires to have the noisy version as well as the corrected version of a large corpus. To cope with this, Tong and Evans [24] propose an interesting approach where the confusion table is learned iteratively. The idea is that corrections found after a pass of the algorithm are used as an approximation of the original text. So, confusion probabilities can be learned by comparing the corrections to the OCR text. Then, the probabilities are fed back into the system to be used during the next pass. This way, the probabilities are refined at each iteration of the algorithm.

Another approach is to use neural networks models for language modeling. For example, Zaremba, Sutskever, and Vinyals [25] train a neural network to perform automatic translation from English to French. We used a similar approach, except our system attempts to translate noisy texts into cleaned versions of the texts.

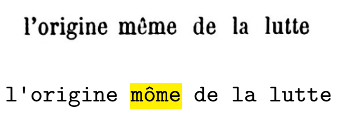
3 Error detection techniques

This section presents the models evaluated for the *error detection* phase. The objective is to classify tokens as correct or misrecognized. It is important that our classifiers reach a high precision for this task, otherwise, we might wrongly detect words as incorrect and end up adding more errors in the text, during the error correction.

To implement error detection, we experimented with three different approaches. The first one is based on *lexicons*. It verifies tokens from the corpus against a pre-compiled list of words. That is, it signals a token as an error if it is not contained in a lexicon (Section 3.1). The second approach relies on *n-grams*, a specific instance of *language models*. Their goal is to estimate the likelihood of sequences of characters or words. They do so by reading texts and computing statistical properties (Section 3.2). Finally, the third approach is based on *neural networks*. They work on similar principles as n-grams models, in that they are fed with sequences of characters or words and are trained to give an estimation of the likelihood of the next character or word. The difference is that, neural networks learn to reconstruct the output during an optimization phase by looking at examples of sentences (Section 3.3).

3.1 Lexicon lookup

Lexicon lookup is one of the simplest technique to detect errors, yet it is quite efficient. This spell checking method works by looking up words in a dictionary. We consider as correct any word that appears in the dictionary. The drawback of this approach is that, any correct word that is not in the dictionary (like *proper nouns* or *foreign words*) will be classified as incorrect (a *false positive* detection). Similarly, it is possible that a word contained in the dictionary is incorrect in a sentence (a *false negative* detection). An example of such a case is called the *word error* (Figure 3). That is, the words that were misrecognized, but resulted in valid French words. Any method that works only at the lexical level will not be able to detect this category of errors. In order to address this particular issue, a method needs to have some level of syntactic or semantic comprehension of a sentence, which dictionaries do not have.


l'origine même de la lutte

l'origine môme de la lutte

Figure 3: Extract of an article from the *Gazette de Lausanne* (1886). The OCR process recognized the word "*même*" as the valid French word "*môme*", creating a *word error*.

We have investigated the use of three different implementations for this project. The first one is the open-source spell-checker *Hunspell*[2]. It is currently the default spell-checker for a lot of softwares and is considered as a state of the art. Hunspell uses dictionaries made of two lists: one enumerates words and the second one lists affix rules for the words. The advantage of this approach is that it does not require to list all the morphological derivatives and inflected forms of a word. For example, we can write a set of rules describing the endings for a group of verbs and associate the base form of all the verbs in this group to this set of rules. The French dictionary we used[3] contains 81 359 words and about 11 000 rules.

The second dictionary we used is based on the *Lexique des Formes Fléchies du Français* (Le*fff*) [21], augmented with lists of cities and locations, proper nouns, currencies and organizations. This implementation does not use affix rules, but the Le*fff* already contains inflected forms that were automatically compiled using affix rules. Overall, this lexicon contains 613 004 words.

The third dictionary is really simple and is used to provide a baseline in order to compare performances. The lexicon is composed of the set of words that appear in the *Chambers-Rostand Corpus of Journalistic French* [6]. This corpus is composed of 1 723 articles published in 2002 and 2003 in the French newspapers *Le Monde*, *L'Humanité* and *La Dépêche du Midi*. It contains about 1 000 000 words, corresponding to 66 579 unique tokens. Articles were collected from the websites of the newspapers, thus this corpus does not have OCR errors. Because of this, we also used it to train n-grams models and neural networks. So, it is interesting to have a point of comparison on how this lexicon performs.

[2]http://hunspell.github.io/

[3]http://www.dicollecte.org/download.php?prj=fr

3.2 N-grams

N-grams are sequences of n elements, where an element can be a character, a word, a sentence or any unit that is relevant to the domain. In our case, we studied the use of character n-grams and word bigrams.

Character n-grams To use this model for error detection, we pre-compute a table of frequency counts of each grams. More precisely, given a sequence of characters $S = c_1, c_2, ..., c_n$, we want to estimate the conditional probability:

$$p = P(c_n \mid c_1, c_2, ..., c_{n-1}) = \frac{P(c_1, c_2, ..., c_{n-1}, c_n)}{P(c_1, c_2, ..., c_{n-1})}$$

The underlying idea is that, this probability will allow us to attribute a score to each character in a word and will help us to detect parts of words that look uncommon.

We compute an estimate of this probability by first counting the number of occurrences of all the sequences of size n in a French text. We will denote the number of occurrences of the sequence S by $|S|$. We have $p \approx \frac{|c_1, c_2, ..., c_{n-1}, c_n|}{|c_1, c_2, ..., c_{n-1}|}$. We compute the denominator of this fraction by using marginal probabilities:

$$p \approx \frac{|c_1, c_2, ..., c_{n-1}, c_n|}{\sum_{c \in \mathcal{A}} |c_1, c_2, ..., c_{n-1}, c|} \tag{1}$$

where $\mathcal{A}$ is the alphabet.

Character n-grams with likelihood In order to attribute a score to the tokens based on their character n-grams, we propose to average the scores given by the n-grams. Given a token $t = c_1, c_2, ..., c_m$ of size m, we first pad it with $n-1$ spaces at the start and at the end.

For each of the $n+m-1$ n-grams in the padded token, we compute its probability p_i, given by Equation 1. We define the likelihood $\mathcal{L}(t)$ of a token as the average of those scores:

$$\mathcal{L}(t) = \frac{1}{n+m-1} \sum_{i=1}^{n+m-1} p_i$$

Here is an example for the token "cat". Let's assume $n = 5$. We first pad this token with four spaces at the start and at the end: "␣␣␣␣cat␣␣␣␣".

Then, we use a sliding window to compute the probability of each *5-gram*. That is:

$$\begin{aligned} p_1 &= P(\text{c} \mid \text{␣␣␣␣}) \\ p_2 &= P(\text{a} \mid \text{␣␣␣c}) \\ &\vdots \\ p_7 &= P(\text{␣} \mid \text{t␣␣␣}) \end{aligned}$$

Finally, we compute the likelihood of the entire character sequence as the average of the probabilities:

$$\mathcal{L}(\text{cat}) = \frac{1}{7}\sum_{i=1}^{7} p_i$$

Initially, the padding was used specifically to solve the issue of tokens smaller than n, but an evaluation of the performances showed that this method performs slightly better when padding every token. A possible explanation is that we have more information to attribute a score, since we know the relative position of a gram: the start, the middle or the end of a token.

Word bigrams In a similar way, we can also count occurrences of sequences of tokens. We limited ourselves to the use of two-words n-grams, also known as *bigrams*.

This technique works in the following way, given two successive tokens t_1, t_2, we approximate $p = P(t_2 \mid t_1)$ as $p \approx \frac{|t_1,t_2|}{|t_1|}$. As for character n-grams, the idea is that we attribute a likelihood to t_2, knowing it comes in the context of t_1.

A special case to handle is when t_1 is unknown, i.e. $|t_1| = 0$. There are multiple techniques to deal with this issue. We can say that $p = 0$ for this particular case or apply Laplace smoothing. That is: $p \approx \frac{|t_1,t_2|+1}{|t_1|+|\mathcal{V}|}$, where $|\mathcal{V}|$ is the size of the vocabulary of words seen when counting frequencies.

We also investigated the use of other smoothing techniques, namely the *Simple Good-Turing* method [12][11] and the *Kneser-Ney smoothing* [19]. Both methods are trying to account for the fact that we have not seen all the valid words during the training phase, i.e. when counting frequencies. They realize that by assigning a non-zero probability to words never seen before. This probability is based on the number of words observed once.

Eventually, we did not use those techniques as they have a significant computational overhead and they would not help to distinguish whether an out-of-vocabulary token is an actual error — a true positive detection — or if it is a valid word, i.e. a false positive detection. Instead, we chose to return $p = 0$ when one of the token is unknown.

The first two models, i.e. the character n-grams with and without token likelihood, were trained using the *Chambers-Rostand Corpus of Journalistic French* [6]. We also trained the *word bigrams* model using this corpus, but found it had really poor error detection capabilities. Thus, we trained this model using a French Wikipedia dump of 600 000 000 words.

The reason we did not used trigrams of words, or higher orders, is that the memory complexity grows proportionally. More specifically, there are $|\mathcal{V}|^2$ possible bigrams, but there are $|\mathcal{V}|^3$ possible trigrams. It is not unreasonable to assume a vocabulary size $|\mathcal{V}|$ of a few hundreds of thousands of words. Thus, computing trigrams might still be tractable, because not all trigrams are valid combination of words, but the problem quickly becomes intractable for higher orders. For character n-grams, we can use higher values for n, since the alphabet has a size of approximately 100 characters (in French).

3.3 Recurrent Neural networks

In this subsection, we describe the architecture of neural network based models. We used a particular kind of neural networks known as *recurrent neural networks* (RNN), which are designed to learn sequences. The approach we used for neural networks is similar to the one used for n-grams. The RNNs are fed with sequences of characters or words, and trained to predict the probability of the next character or word. The reasoning is that the networks should be able to use the context given by the previous characters or words to evaluate the likelihood.

We tried many different RNN models during the course of this project.

Two layers RNN Our first neural network model is a network with two-layers of *long short-term memory* cells (LSTM) [13]. This system is fed with one-hot encoded vectors corresponding to characters from the text and trained to output a probability distribution for the next character. Figure 4 illustrates the architecture of this model.

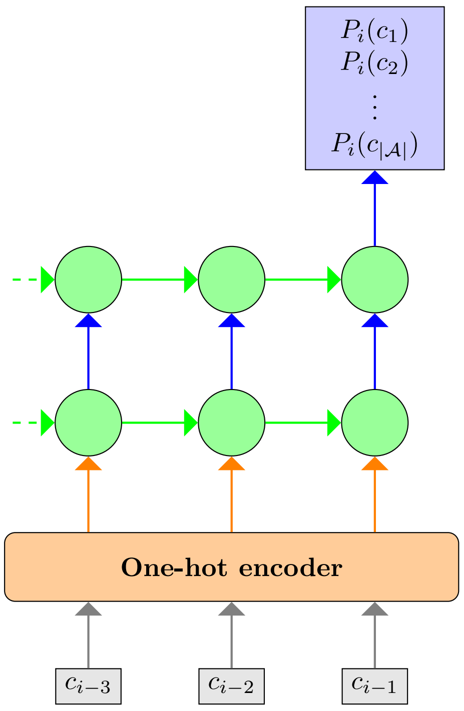


Figure 4: High-level representation of the first RNN model. When we feed a window of characters c_{i-n}, ..., c_{i-2}, c_{i-1}, the network cells (in green) combine the inputs (orange arrow) with the previous states (green arrows) to produce a probability distribution P_i predicting the likelihood for the character c_i.

One-hot encoding is a way to represent data with categories separated into different features. In our case, this means that a character with index i is encoded as a boolean vector with a 1 at the index i and zeros elsewhere. For example, assuming we only use the 26 lowercase letters from a to z, we would encode the word "french" as the vectors shown in Table 1.

	a	b	c	d	e	f	g	...	r	...	z
f	0	0	0	0	0	1	0		0		0
r	0	0	0	0	0	0	0		1		0
e	0	0	0	0	1	0	0		0		0
⋮											

Table 1: Example of one-hot encoded row-vectors for the letters in the word "*french*".

This encoding is useful because it separates each character into its own feature. We make use of this for the output of the neural network, which is a vector of probabilities over every character from the alphabet. This way, we train the network to predict the probability distribution over the next character, similarly to what has been done with character n-grams. The alphabet we used is composed of all the letters seen during the training phase, including lowercase and uppercase letters, characters with diacritics and punctuation signs. We experimented with using only lowercase letters, but found that it slightly decreased the performances at predicting characters.

Multiple parameters can be adjusted:

- The type of RNN cells: We tried to use LSTM and Gated Recurrent Units (GRU) [7]. No significant difference was seen in terms of error detection performance, but the convergence was slower using GRU.
- The number of hidden units: This is a compromise between memory usage and performances at detecting errors. We found that 512 units per layer was a reasonable compromise, as increasing this number would consume too much memory and decreasing it would hurt the detection capabilities.
- The use of Dropout [23]: Using dropout leads to a slightly better detection of errors.

Two layers RNN with auxiliary features Based on the same architecture, we adapted the model so that it predicts not only the next character, but also if this character will be a vowel, if it will be in uppercase, if it will be a letter with a diacritical sign and if it will be a punctuation sign. The intuition is that, forcing the network to learn some of these auxiliary features might help it to predict the main feature: the probability distribution of the next character.

Bidirectional RNN We evaluated a third architecture, based on *bidirectional recurrent neural network* (BRNN) [22]. BRNN relies on the fact that it could be useful to predict based not only on data on the left of the current character (prior), but also on data placed on the right (posterior). To achieve this, BRNN combines two layers: one layer reads the sequence of characters from left to right and the other layer reads characters from right to left. The outputs of the forward and backward layers are then combined together to produce one output for the bidirectional layer.

The rest of the architecture of this model is similar to the architecture of the first RNN model. We use one-hot encoded vectors as input, but this time, we also feed characters on the right of the current character. The first layer is a bidirectional layer with 256 units. We reduced the number of hidden units. Indeed, the bidirectional layer contains two hidden layers and thus, uses twice as much memory. As before, we have a second regular RNN layer and the output is a probability distribution over the letters. The architecture is shown in Figure 5.

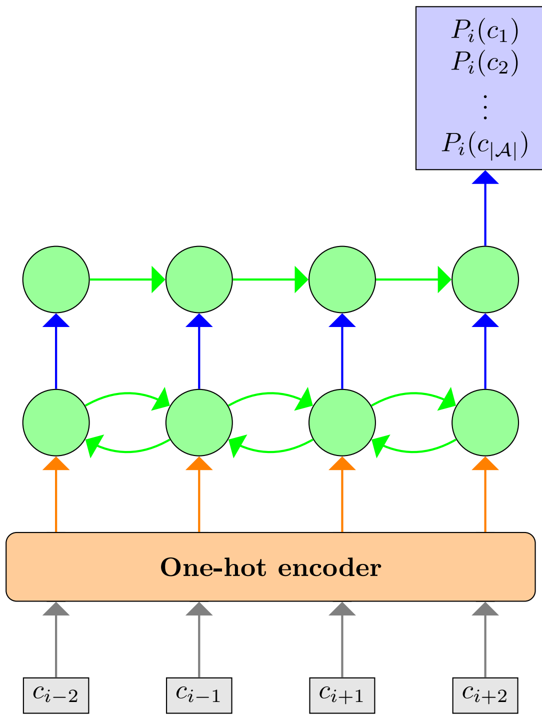


Figure 5: High-level representation of the bidirectional RNN model. We feed a window of prior and posterior characters c_{i-n}, ..., c_{i-1}, c_{i+1}, ..., c_{i+n}. The bidirectional cells of the first layer (in green) read this input in the two directions and produce an intermediate output. A unidirectional layer collects those outputs to produce a probability distribution P_i predicting the likelihood for the character c_i. The figure illustrates this for $n = 2$.

Bidirectional RNN with auxiliary features We explored a fourth model, which is a combination of the second and third models we presented. That is, a bidirectional recurrent neural network with auxiliary outputs. It uses the same architecture as the one described in Figure 5. But we train the model to also predict if the next character is a vowel, has a diacritical mark, is uppercase or is a punctuation sign. As explained before, the intuition is that these auxiliary features might help the network for the main task of predicting the next character.

RNN with word embeddings The last neural network model we tried works at the level of words, instead of characters. It uses the *fastText* word embeddings model [4] to transform words into numerical vectors. Those vectors are then fed into a RNN, which outputs a probability distribution over a vocabulary of known words (Figure 6).

Word embeddings techniques are useful in our case, because they group together the word vectors for the words that appear in a similar context. By training a word embedding on the *Le Temps* corpus, we should be able to have word vectors of errors positioned closely to their corrections, since they share a similar context. Another useful property of *fastText* is that it also uses the character n-grams of a word to generate its embedding. So, words that are close morphologically should also be positioned together.

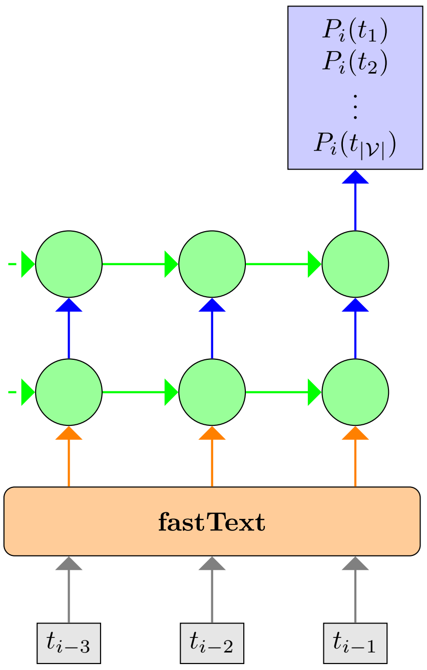


Figure 6: High-level representation of the *fastText with RNN* model. We feed the network with a window of embedded tokens t_{i-n}, ..., t_{i-2}, t_{i-1}. The network uses those inputs and the states to output a probability distribution P_i predicting the likelihood for the token t_i.

The five neural networks were all trained using the *Chambers-Rostand Corpus of Journalistic French* [6]. The *fastText* word embedding was trained separately on the *Le Temps* corpus.

4 Error correction techniques

This section reports the methods we used to generate suggestions of corrections for words that are detected as errors. There are two main parts in the task of automatically correcting errors: generating possible corrections and ranking those candidates to choose the most likely. The techniques used to generate candidates include replacement rules (Sections 4.1 and 4.2), words bigrams (Section 4.3), word embeddings (Section 4.4) and neural networks (Section 4.5). The second part — i.e. the candidates ranking system — is detailed in Section 4.6.

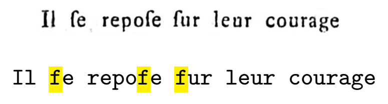
4.1 Hunspell

Hunspell provides a module that generates suggestions using replacement rules. In particular, it checks for the insertion, omission, substitution or move of a letter in a word. It also checks for concatenation of words and tries to replace characters by similar ones — like 'ç' for 'c'.

4.2 Replacement rules

In order to address some systematic errors, we added a few replacement rules. They all work the same way: If a word is detected as an error, we try to apply a rule and check if the result is detected as a valid word.

Long s character A first issue is related to the *long s* character 'ſ', as shown in Figure 7. It was used in printed materials until around 1820. Due to its similarity with the letter 'f', it is generally misrecognized by OCR algorithms. Thus, for tokens detected as incorrect, we replace all the occurrences of the letter 'f' by the letter 's' and check if the resulting tokens correspond to valid words.


Il ſe repoſe ſur leur courage

```
Il fe repofe fur leur courage
```

Figure 7: Extract of an article from the *Gazette de Lausanne* (1798). The OCR algorithm recognized the *long s* characters as the letter 'f'.

Hyphenation A second systematic issue is caused by texts being placed in fully justified columns. Due to this, words that are normally not hyphenated may appear with a hyphen in order to wrap the line. Unfortunately, we do not know where newline characters were placed, since they were removed by the OCR. Thus we check if an incorrect token containing a hyphen becomes valid when removing the hyphen.

Heuristics Bornand et al. [5] proposed a few heuristics in order to clean the text of the *Le Temps* Corpus. In particular, they propose the use of the following replacement rules: $iii \to m$, $ii \to n$ and $q[a-z] \to qu$. The results for the first two rules were not really precise enough to be useful in our case, but the last rule proved to be quite accurate at correcting tokens.

Partitioned tokens Another replacement rule attempts at correcting the insertion of whitespace characters specifically. We use a heuristic similar to what is described by Elmi and Evens [10] for word boundary errors. That is, given two successive tokens t_i, t_{i+1}, if one or both tokens are detected as errors, then we try to concatenate $t_i + t_{i+1}$ together. If it results in a valid word, we keep the concatenation as a candidate correction.

Concatenated tokens Finally, this last rule does the opposite of the previous one, in that it attempts to correct concatenated tokens by finding which partitions of a token are valid. More precisely, if a token is long enough, then this heuristic cuts the token and checks if the two parts are valid, using an error detection algorithm.

4.3 Bigrams

We can reuse some of the models used for detecting errors in order to generate suggestions based on neighboring words. The simplest model for this is the *word bigram* model detailed in Section 3.2. Given two successive tokens t_{i-1}, t_i, when t_i is detected as an error, we search for the most likely words w that follow t_{i-1}:

$$\arg\max_{w \in \mathcal{V}} P(w \mid t_{i-1})$$

This gives us a set of words that are possible candidate replacements for t_i. Since those candidate words could have a totally different spelling from

t_i, we filter out the words w that are too far from t_i.

There exist many possible string similarity algorithms to compute a distance between two tokens. In this project, we used the *normalized Levenshtein distance*. It is defined as:

$$d(t_j, t_k) = \frac{lev(t_j, t_k)}{\max(|t_j|, |t_k|)}$$

Where $lev(t_j, t_k)$ is the Levenshtein distance between the tokens t_j and t_k. That is, the smallest number of insertion, deletion or substitution of letters needed to transform t_j into t_k.

The underlying idea is that, we first generate a set of candidates from the context given by t_{i-1}. So, if t_{i-1} is discriminative enough, those candidates should belong to the same domain of discourse. Since it could happen that t_{i-1} gives little information (if it is a *stop word* like "the" or "of"), we make sure the suggestions have a spelling similar to the target token t_i using the Levenshtein distance.

4.4 Word embeddings

In Section 3.3, we have described how we use *fastText* [4] with a recurrent neural network to detect errors. We can reuse *fastText* to generate suggestions in the following way.

We have a *fastText* model trained on the entire *Le Temps* corpus, which we use to precompute the embeddings of all the words from a dictionary. During the error correction, for each token detected as incorrect in the corpus, we compute its embedding and we compare it to the pre-computed embeddings using the cosine-similarity. That gives us a set of words from the dictionary which are semantically close to the target token according to the *fastText* model. Then, we filter those suggestions using the normalized Levenshtein distance, to keep only the candidates that are spelled similarly to the target.

4.5 Neural networks

We can also reuse the recurrent neural network which uses fastText as a first layer, detailed in Section 3.3. We input the context of the previous words, given by an array of embeddings, and the system outputs the most likely candidates. Those candidates belong to the vocabulary of words observed during the training phase. In a more formal way, given a window of previous tokens t_{i-n}, ..., t_{i-2}, t_{i-1}, we search for the most likely words w such that:

$$\arg\max_{w \in \mathcal{V}} P(w \mid t_{i-n}, ..., t_{i-2}, t_{i-1})$$

As for the other models, we filter out the suggestions for which the normalized Levenshtein distance is too big. We have tested models with $n = 2$ and with $n = 5$. We found that $n = 5$ provides slightly better performances at generating suggestions for almost no increase in computational cost. Thus, in the following sections, we report the results for $n = 5$.

4.6 Candidates ranking

Once we have generated suggestions, we have to rank them. This is a difficult task since replacement rules provides no similarity measures between the replacement and the token. Furthermore, even if the bigram model, the word embeddings and the neural network model all output some form of similarity measures, these measures have different orders of magnitude. For example, likelihoods from the bigram model will typically take values from 0.1 down to 10^{-6}, whereas the cosine-similarity values for the suggestions from the word embeddings will generally be in the range from 0.5 to 1.

To address this issue, we propose to assign weights to the different suggestion generators, in order to rescale the scores. More precisely, we weighted generators by categories: replacement rules, Hunspell, word embeddings and bigrams. We ended up not using the neural network based model to generate candidates because of its high computation cost. We use the weights in the following way: Suggestions generated by Hunspell and by replacement rules take the value of the corresponding weights. For the other generators, we multiply the similarity scores that are returned by the methods with the weights corresponding to the generators. Finally, since it is possible that multiple generators suggest the same candidate word, we sum the scores when this is the case. This is a way to boost the confidence attributed to a suggestion, when multiple methods find that it is a likely candidate.

The values of the weights were determined using *grid-search*. That is, we generated pre-defined values and computed how our error correction algorithm performs on a test dataset.

As a post-processing step to this ranking system, we filter out the suggestions which are too different from the target token by using a threshold on the normalized Levenshtein distance. At the end of this process, we have a list of suggestions, where the highest scores correspond to the most likely candidates to replace a token detected as incorrect.

5 Evaluation

In Section 5.1, we detail how we annotated a number of randomly chosen articles in order to create a *gold standard*. Based on this annotated corpus, we perform an analysis of the OCR errors found in the two newspapers in order to estimate the amount of noise. We report the performances of the *lexicons*, *n-grams* and *neural network* models at detecting errors in Section 5.2. We detail how we adjusted our error detection model to cope with a number of systematic errors in Section 5.3. The Section 5.4 details the results of the error correction process.

5.1 Gold standard

In order to measure the performances of the models, we need to have a *gold standard*. That is, a manually annotated set of articles from the corpus. We created it by picking 80 articles at random in the corpus and cutting them to have a similar size. Errors were then annotated using the *brat* annotation software[4], by comparing the scanned issues to the OCRed text.

We have used the following categories to annotate errors:

- split: `recommanda tions` `recommandations`
  When one word is split into two or more parts.

- concatenation: `leprojet` `le projet`
  When two or more words are concatenated together.

- substitution: `KOMAINMÔI'tER` `ROMAIN MÔTIER`
  For substitutions of characters.

- insertion: `signallé` `signalé`
  For insertions of characters.

- deletion: `rpellations` `interpellations`
  For deletions of characters.

- nonsense: `•••«••» >`
  When the OCR process tried to recognize elements that are not characters, like lines separating the columns or ink splatters.

- uncorrectable: `JJmTpE GEIVË` `JOURNAL DE GENÈVE`
  For words that are so noisy that even a human needs to check the scanned pages in order to retrieve the original words. Generally, this kind of error comes in bursts, which makes it impossible to guess from the context.

- uncorrectable number: `72 Ô 4` `7224`
  This sub-category is specifically for numbers, since deleting digits, adding spaces or substituting characters in a number generally makes it impossible to guess the original number — or even to detect that there was a modification without looking at the scan.

- foreign: `TanghUuser` `Tannhäuser`
  For foreign words, generally proper nouns, that have been misrecognized.

Here are the statistics concerning this test dataset. It is composed of 80 newspaper articles, with 39 articles from the *Journal de Genève* and 41 from the *Gazette de Lausanne*. This corresponds to a total of 6 285 tokens.

All articles have a normalized size. That is, we manually cut them at the first end of sentence coming after 400 characters.

There are 691 tokens with an annotation, for an average word error rate of 11%. However, Figure 8 suggests that the *Gazette de Lausanne* has less noise, compared to the *Journal de Genève*. Indeed, we have found that the error rate in the test dataset for the *Journal de Genève* is 14% and 8% for the *Gazette de Lausanne*.

In Figure 8, we can also observe that some samples diverge from the average, with a sample with an error rate of 67 %. This is usually caused by degraded materials, where the OCR process was unable to properly recognize a lot of words. Thus, it causes error bursts. This kind of errors is especially difficult to correct, as the context in which a noisy token appears is itself noisy.

[4] http://brat.nlplab.org/

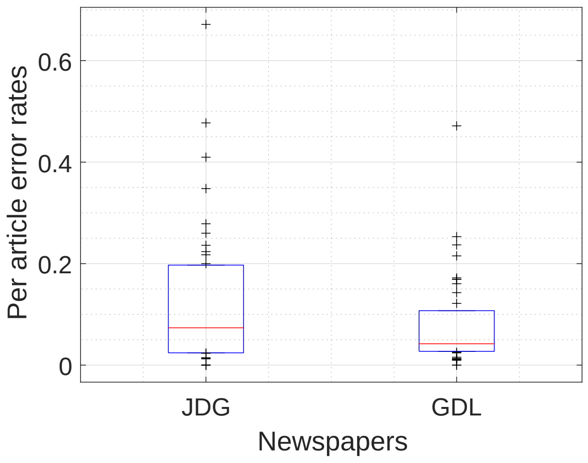


Figure 8: Boxplot of the error rates per article in the test dataset.

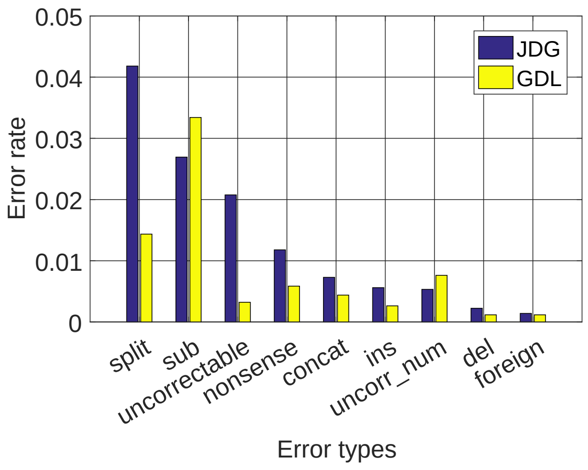


Figure 9: Error rate per category of error for each of the two newspapers.

Figure 9 shows that the distribution of errors per category is quite different between the two newspapers. The *Journal de Genève* has a majority of *split* errors, while the *Gazette de Lausanne* mostly contains *substitution* errors. Also, we observe that the amount of *uncorrectable* errors is significantly higher for the *Journal de Genève*. This suggests that error bursts, which generally are the cause of this type of error, could be more frequent in the *Journal de Genève* than in the *Gazette de Lausanne*. Although, the population size is quite small (80 articles in the test dataset), so this could also be a local deviation.

5.2 Error detection evaluation

The models were evaluated using the annotated corpus described in the previous subsection. Since we know where the errors are located, we can compare this *gold standard* with the predictions of each classifier.

Error detection ROC curves To measure the performances of each classifier, we used *Receiver Operating Characteristic* curves (ROC curves). They are created by plotting the *true positive rate* against the *false positive rate* for a set of thresholds. In our case, the *true positive rate* corresponds to the probability of correctly detecting errors and the *false positive rate* to the probability of false alarm, i.e. signaling as an error a token that is actually correct. This way, we have a visual representation of how each classifier behaves for different thresholds. Perfect classifiers would be on the upper left corner, while random classifiers would be placed on the diagonal.

Figures 10 and 11 show the ROC curves for the *n-grams* models and the *recurrent neural network* models respectively. On each figure, the three data points corresponding to the *lexicons* provide reference points.

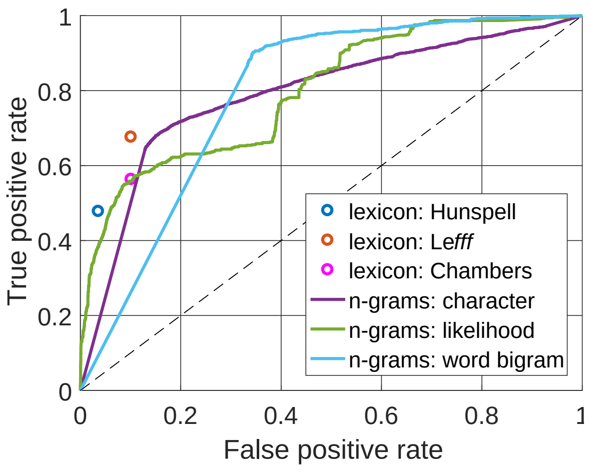


Figure 10: ROC curves comparing the *lexicons* and the *n-grams* models at the task of detecting errors.

In Figure 10, we can see that none of the *n-grams* models can really compete with good dictionaries. In particular, we can remark that Hunspell has a really low rate of false positives, with a decent true positive rate. This is a desirable property for our project, since we really want to avoid having false positives. Indeed, a false positive detection means that, during the error correction phase, we will actually introduce errors in the corpus. So, in this project, we chose to favor classifiers that have a low probability of false alarm. Although, it is worth noting that the Le*fff* also has good performances. It trades off having more false positives against a better detection rate.

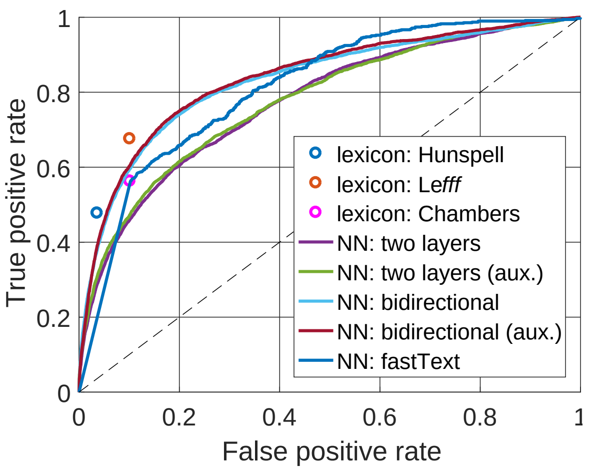


Figure 11: ROC curves comparing the *lexicons* and the *neural network* models at the task of detecting errors. The models marked with "(aux.)" correspond to the models trained using auxiliary features.

In Figure 11, we notice that the dictionaries based on *Hunspell* and the *Lefff* dominates the other models. It is interesting to note that the neural network models trained with auxiliary features do not bring any performance gain in terms of error detection. There could be multiple causes to this.

Either, those auxiliary tasks are too different from the main task, thus not useful at predicting characters. Or, it could be that the network already learned the main task well enough. So, learning to categorize characters does not bring additional benefit. Indeed, if the system can reliably predict characters, then it becomes easy to determine the categories associated to each character (vowel, uppercase, diacritic or punctuation).

Execution time We measured the execution time of the classifiers. The values we report are the time in seconds needed to classify 10 000 tokens (or an equivalent amount of characters, about 60 000, for character-level classifiers). They are measured as the minimum execution time obtained over 10 runs of the same classifier. The machine we used is a server with 2 Intel Xeon E5-2680 2.5 GHz CPUs with 256 GB of RAM and 2 Nvidia Titan X GPUs.

Classifier	Time (CPU)	Time (GPU)
lexicon: Hunspell	0.198	—
lexicon: Le*fff*	0.0113	—
lexicon: Chambers	0.00293	—
n-grams: character	0.199	—
n-grams: likelihood	0.581	—
n-grams: word bigram	0.0513	—
NN: two layers	68.6	4.55
NN: two layers (aux.)	69.5	4.22
NN: bidirectional	111	6.50
NN: bidirectional (aux.)	56.9	3.37
NN: fastText	55.1	46.5

Table 2: Execution times in seconds of the error detection classifiers, rounded to 3 significant digits.

Table 2 provides a few interesting points. First, there is a clear separation between the neural network models and the others. We were expecting neural networks to be slower, but such a difference means that we cannot use neural networks when applying this project to the *Le Temps* Corpus. Indeed, it would take an unreasonable amount of time to apply the neural network models to 200 years of data. With approximately 2.3×10^9 tokens in the corpus, we can compute that it would take roughly 9 days to apply the fastest RNN model, i.e. the "NN: bidirectional (aux.)" model. This might not seem that much, but it should be noted that these models compute likelihoods at the level of characters. Because we are more interested to have the likelihood of an entire token, we can expect the processing rate to go down due to the additional work required to compute the overall likelihood — similarly to what happened with the *n-grams: character* and *n-grams: likelihood* models. Furthermore, it is worth noting that during the error correction, the replacement rules make a heavy use of the error detection module, in order to verify if a replacement produces a valid token.

When having the GPUs enabled, the computation times are much faster. But, the fastest model is still several times slower than the dictionaries or the n-grams based models.

Secondly, we found that the two multi-outputs RNN models sometimes execute faster than their simpler counterparts. This was a surprise, but it can be partially explained by the fact that the auxiliary features of the multi-outputs models do not have to be computed during the prediction phase, after the training is over. Another explanation is that, for the two simpler versions, we used the *TFLearn* [9] library, which is a high-level wrapper of *Tensorflow* [16]. Due to some limitations in *TFLearn*, we used *Tensorflow* alone for the two multi-outputs models. So, this suggests that the *predict* function of *TFLearn* has an important overhead or performance issues. We also used *TFLearn* for the "NN: fastText" model.

Finally, the *dictionaries* and *n-grams* models all have acceptable computation times. From those, the "n-grams: likelihood" model is the slowest, but it can be explained by the fact that it has to loop over all the n-grams of a word, lookup their probabilities and take the average. Similarly, "n-grams: character" is slow because it has to look up for the probabilities of every character in a text. It is comparable to "lexicon: Hunspell", which is slower than the other lexicons, but this probably comes from the fact that Hunspell has to search which affix rules can be applied to a token before looking up that token in the dictionary.

To conclude this part about error detection, we chose to use Hunspell. As stated before, it has a low false alarm rate, which is especially important for our project, while it conserves a good detection rate. It also runs at a decent speed, which will be needed since some replacement rules make a heavy use of the error detector.

5.3 Error correction with Hunspell

Having chosen Hunspell as the main component for error detection, we realized that some issues had to be addressed.

Diacritics The first problem comes from the non-systematic use of diacritical marks for capital letters in the corpus — as in the word "l'Etat" instead of "l'État" or in "THEATRE" instead of "THÉÂTRE". Since all the French dictionaries follow the usage of having diacritical signs for uppercase letters, as recommended by the *Académie française*[5], those types of words were always misclassified as errors.

We fixed this by transforming every accentuated word in the Hunspell dictionary file into non-accentuated uppercase variants. Next, we added those derived forms into Hunspell, associating them with the same affix rules as the original words.

Typrographic ligatures A similar issue are the words containing the typographical ligatures 'æ' or 'œ'. Hunspell accepts only the words with the ligatures, but those words were often printed with 'ae' or 'oe' instead. In a similar way as before, we created derived forms for words with ligatures and added them to Hunspell.

Language evolutions Another difficulty is that, the *Le Temps* corpus covers 200 years and the language has evolved. There have been multiple reforms of the French orthography. In particular, the reform of 1835 changed the orthography of about a quarter of the words that were in the dictionary of the *Académie française*. As illustrated in Figure 12, this reform changed the spelling of the plural forms of the words ending with *-nt*. For example, the word "gouvernement" was spelled "gouvernemens" in plural form, which became "gouvernements". It also systematized the use of *ai*, instead of *oi*, in words where it was already pronounced as *ai*.

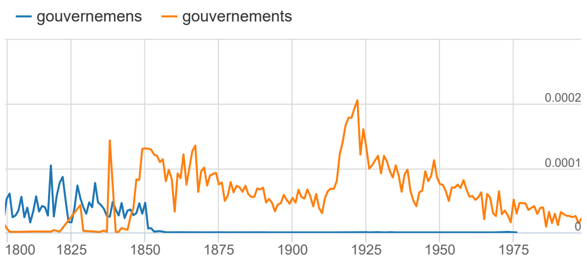


Figure 12: Frequencies of the words "gouvernemens" and "gouvernements" in the corpus. The use of the old form "gouvernemens" completely dropped after 1850.

These evolutions are difficult to handle in this project, because any modern dictionary will indicate those old variants as incorrect words, even if they were correctly recognized. We mitigated this problem by adding the 1798, 1835 and 1932 editions of the dictionary of the *Académie française*[6] to our dictionaries.

When classifying a token, we first check if it exists

[5]http://www.academie-francaise.fr/questions-de-langue#5_strong-em-accentuation-des-majuscules-em-strong

[6]Several editions of the dictionary of the *Académie française* can be found on the website of the ARTFL project: http://artfl.atilf.fr/dictionnaires/ACADEMIE/index.htm

in Hunspell and, if not, we check if it exists in the edition of the dictionary of the *Académie française* that was valid for that year. This does not entirely solve the issue of old orthographic forms, as those dictionaries do not include all the inflected forms or all the morphological derivatives. Additionally, these dictionaries were digitalized using OCR, so they may also contain OCR errors. Empirically, it seems that the error rate is pretty low.

Proper nouns Finally, we remarked that proper nouns account for most of the false detections. Due to the lack of time at the end of the project, we did not integrated advanced techniques like *named-entity recognition* to address this issue. As a workaround, we propose that the error detector accepts any token beginning with a capital letter and followed by lowercase letters. This is not ideal, as the error detection module will detect as correct all the tokens that were misrecognized and that start with an uppercase letter followed by lowercase letters. The ROC curves in Figure 13 illustrates the consequences of this trade-off.

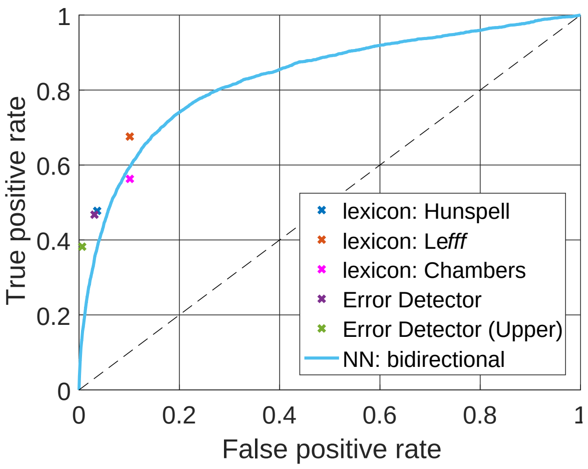


Figure 13: ROC curves for the dictionary based models and for the bidirectional RNN model, shown for reference. "Error Detector" corresponds to Hunspell augmented with the dictionaries of the *Académie française* and with the rules listed above for handling uppercase words with diacritical signs and ligatures. "Error Detector (Upper)" is like the previous one, but also accepts words in titlecase.

Final model performances Figure 13 illustrates the performances at detecting errors for these variants of Hunspell. Accepting title case words decreases the false positive rate from 3.07 % down to 0.67 %. This obviously comes at the cost of a decreasing true detection rate: from 46.7 % to 38.2 %. Table 3 resumes the performances of our final error detection model.

Class	Precision	Recall	F1-score
Errors	0.88	0.40	0.55
Corrects	0.93	0.99	0.96

Table 3: Performance report of our error detection model for the two classes of tokens: *misrecognized* or *correctly recognized*.

Error analysis of the corpus We applied this classifier to the *Le Temps* corpus. Figures 14 and 15 report the error rates for the two newspapers.

With this automated error detecting algorithm, the average error rate is 4.3 % for the *Journal de Genève* and 3.0 % for the *Gazette de Lausanne*. This is much lower than the true error rates we had previously measured on the manually annotated corpus, which were of 14.0 % and 8.0 %. But, given that we used a filter with a low sensitivity, it is coherent that we detect a lower number of errors than there really are.

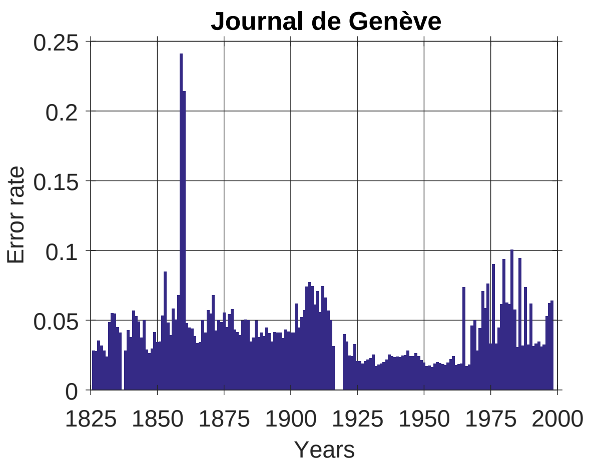


Figure 14: Error rate per year as reported by our automated error detector.

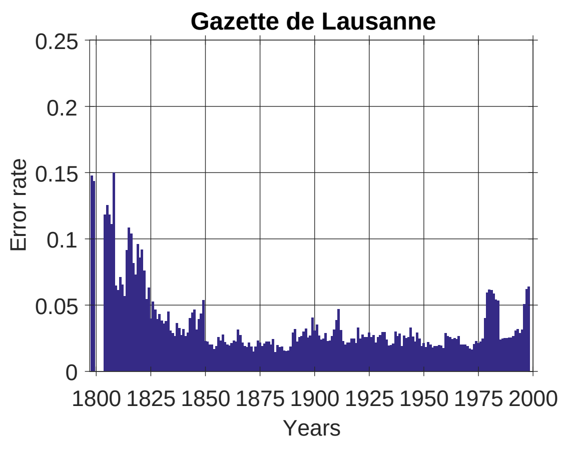


Figure 15: Error rate per year as reported by our automated error detector.

A few outliers can be observed in Figure 14. In 1859 and 1860, the word error rate for the *Journal de Genève* is way higher than it is for any other years. Similarly, from 1965 to 1990, an error pattern appears, where the error rate for some years is significantly higher. This seems to be caused by some settings used during the OCR process, as can be seen in Figure 16. The size of the directories containing the digitized issues is generally significantly lower for the years with a high error rate. Surprisingly, there appears to be no correlation between the resolution of the scans and the error rates, so maybe it is a problem caused by an uneven image compression.

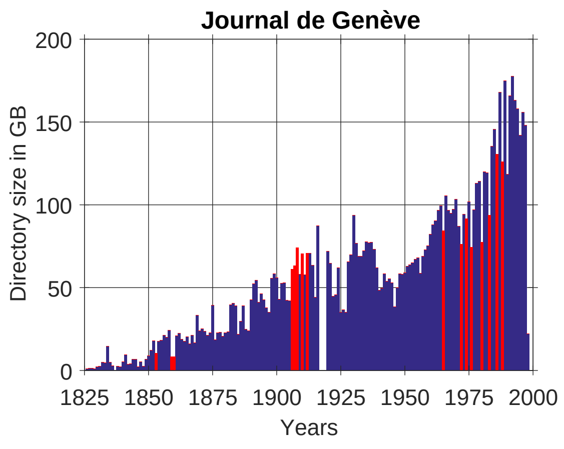
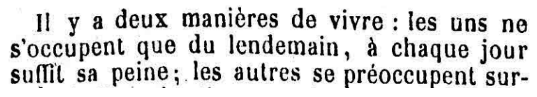


Figure 16: Directory size of the scanned issues of the *Journal de Genève* per year. The years with an error rate bigger than 7 % have been outlined. Except for the group of years around 1910, the directory size is significantly lower for the years with a high error rate than for the adjacent years, despite having a similar number of tokens.

Another outlier for the *Journal de Genève* is the year 1853. The OCR algorithm failed to recognize a lot of whitespaces for the months of July and August of that year (Figure 17).


Il y a deux manières de vivre : les uns ne s'occupent que du lendemain, à chaque jour suffit sa peine ; les autres se préoccupent sur-

```
 Ilyadeuxmanièresde. vivre : lesun ~ ne
s'occupentquedulendetnain, àchaqueJour
suflltsapeine ; lesautressepréoccupentsur-
```

Figure 17: Extract of an article from the *Journal de Genève* (July 1853). The OCR process has concatenated most of the words.

No cause was found to explain the increases of the error rates seen around 1910 and after 1995.

Concerning the *Gazette de Lausanne*, we can observe there are more errors at the start. This is likely caused by the *long s* characters, recognized as the letter ‘f’.

We found no cause that would explain the increase of the word error rate for the years around 1980 and after 1995.

5.4 Error correction evaluation

Experiment settings Our final error correction system is based on the models detailed in Section 4. More precisely, we first check for partitioned words: Given two successive tokens t_i, t_{i+1}, if one or both are detected as errors, we check if the concatenation $t_i + t_{i+1}$ is valid. If it is, we keep it as the correction and we continue to the next token t_{i+2}. Otherwise, if t_i is predicted as incorrect, we generate suggestions. To do so, we use replacement rules from Hunspell, as well as other replacement rules to handle special cases like *long s* characters and hyphens. We also use *word bigrams* and the *word embeddings* based on *fastText* to generate contextual suggestions. We did not used the neural network model described in Section 4.5. Indeed, Table 2 suggests it would cost too much computation time.

The next step is to rank the suggestions. We apply the system described in Section 4.6. It rescales the similarity scores returned by the generators using weights and aggregates the results when multiple generators suggest the same corrections. We used *grid-search* to attribute values to the weights, so that the values maximize the number of times the algorithm find the correct suggestions.

Performance analysis In order to evaluate the accuracy of the suggestions, we use the corpus of annotated articles. Since it contains the correction for each true error, we can compare those expected corrections to the output of our error correction algorithm. We found that our method ranked the correct suggestion as the top 1 for 61 tokens. Given that there are 293 tokens detected as errors, this represents a precision of 21 %. This is lower than some recent work in the same domain — for example, Mei et al. [17] report a correction rate of 61.5 % for the top 1 candidates at a similar task of OCR correction, although the test dataset they used contains only 3.22 % of errors.

To find why our method performs poorly, we looked at some more detailed statistics (Table 4). We discriminated between the case where there is exactly one suggestion with the highest score and the case where multiple suggestions have the highest score. The idea was that in the first case, our algorithm can reliably pick the suggestion in the top 1 as the correction, because the ranking system deem that suggestion as the best. Whereas, in the second case, we cannot distinguish between the suggestions, thus the algorithm does not attempt to decide.

Real errors	Top 1: $size = 1$	110
	— Correct suggestion	**55**
	— Wrong suggestion	55
	Top 1: $size > 1$	25
	— Correct suggestion	**6**
	— Wrong suggestions	19
	No suggestions generated	122
	Total	257
Not errors	Top 1: $size = 1$	18
	Top 1: $size > 1$	13
	No suggestions generated	5
	Total	36

Table 4: Detailed token counts for all possible outcomes of the error correction over the 293 tokens detected as errors. The good outcomes are highlighted in bold characters.

From Table 4, we can say that the error correction is not really precise enough to be applied to the corpus. Indeed, we were able to correct 55 of the detected errors, but we also replaced the same amount of tokens with incorrect suggestions. Worst than that is the fact that the error correction algorithm found suggestions for 31 tokens that were actually correct.

This suggests that, either the generation or the ranking of suggestions is not precise enough. To check the quality of the candidates generation, we verified if the expected corrections match suggestions among the whole lists of suggestions. We discovered that the expected corrections were found 69 times using the entire suggestion lists. We recall that there are 257 tokens correctly detected as errors, which corresponds to the upper bound for this measure. So, for most of the tokens, our generators did not even generated the true corrections, or those corrections were rejected as being too different from the tokens. Moreover, there are 122 tokens for which no suggestion was generated at all.

We also remark that when the true corrections appear in the lists of candidates, the ranking system generally succeeds to attribute high scores to those corrections. Indeed, in 80 % of those 69 cases, the correct suggestion was ranked as the top 1 and was the unique suggestion with the highest score. Overall, this means that, while the ranking system seems to perform well, we have a low correction rate that is caused by the candidates generation process, which usually fails to generate the expected corrections.

Analysis per error types In order to better understand which errors were corrected, we collected statistics per error type for the 257 tokens correctly detected as errors. Figure 18 shows that while we corrected about 28 % of the *substitutions*, our algorithm also chose wrong suggestions for the same amount of tokens. No suggestions were generated for 45 % of the *substitutions*, which is a big issue considering it is the category with the most errors.

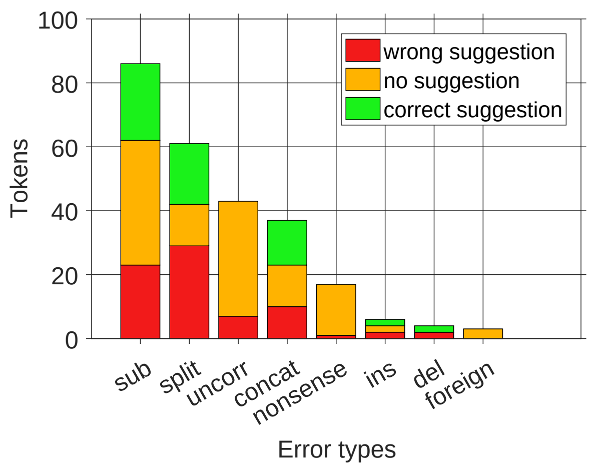


Figure 18: Number of tokens for which a correct suggestion, a wrong suggestion or no suggestions were found per type of error.

The *split* error type is also a major concern, as our algorithm generally decides wrong suggestions. We found that this is caused by split tokens that also contain another error (usually a substitution). For example, the word “entre” was once recognized as “enr (re”. We have a replacement rule that attempts to fix split tokens, but it will fail to correct such cases. One solution could be to run the suggestion generators on the concatenation of two tokens $t_i + t_{i+1}$ when one of them is detected as incorrect. An issue is that it would have an important computational cost, as it doubles the number of calls to the candidate generators — and most of the runtime is already spent executing the generators. It might also make it more difficult to rank suggestions, as we would have suggestions corresponding to the token t_i alone and suggestions for the concatenation $t_i + t_{i+1}$.

Uncorrectable errors were mostly untouched, as our algorithm did not generated suggestions for the majority of the tokens in this category. This is pretty good, as tokens were deemed uncorrectable when there was so much noise in a sentence that it seemed impossible to correct the sentence without looking at the scanned issues.

About *concatenated* tokens, the conclusions are similar to the ones concerning *split* tokens. That is, our algorithm is capable of correcting simple concatenation, but fails when there are additional errors.

For the tokens classified as *nonsense*, our algorithm did not generated suggestions for the majority of the cases. This is good, as we recall that the nonsense category represent tokens for which the OCR algorithm tried to recognize elements that are not characters — for example, separating columns or ink splatters. It would be nice to have a post-processing step to our error correction algorithm in order to try detecting nonsense tokens. Indeed, the correction for this kind of tokens is to delete them.

Parameters tuning We show that the results from the error correction can be tweaked as needed by tuning the parameters of the ranking system. The results detailed previously were collected with a set of parameters that maximizes the number of correct suggestions found. Here, we show the error correction results when using parameters that minimize the number of wrong suggestions. We recall that the values for the parameters were chosen with *grid-search*.

With this second set of parameters, the rate of correct suggestions found by our algorithm drops to 9 %. This is even lower than before, but the advantage is that our algorithm picked the wrong suggestion for only 11 tokens (Table 5).

Real errors	Top 1: *size* $= 1$	27
	— Correct suggestion	**23**
	— Wrong suggestion	4
	Top 1: *size* > 1	3
	— Correct suggestion	**3**
	— Wrong suggestions	0
	No suggestions generated	227
	Total	257
Not errors	Top 1: *size* $= 1$	3
	Top 1: *size* > 1	1
	No suggestions generated	32
	Total	36

Table 5: Results of the error correction over the 293 tokens detected as errors when using parameters that minimizes the number of wrong suggestions. The good outcomes are highlighted in bold characters.

The expected corrections were generated 27 times. Given that the ranking system succeeded 23 times to rank those suggestions as the unique Top 1, this gives a success rate of 85 % at ranking suggestions.

6 Future work

About the quantification of noise, our estimations, based on the annotated articles, have shown that the *Journal de Genève* has an average word error rate of 14 %, while the rate is 8 % for the *Gazette de Lausanne*. This difference in the error rates is coherent with the results found when applying our error detection algorithm to the corpus. But we did not investigated how the corpus might have changed over time. It would be interesting to have a diachronic analysis of the corpus, for example by annotating a number of articles for every fixed period of time.

Concerning the error detection, our system is capable of detecting about 40 % of the errors. *Named-Entity Recognition* (NER) has already been applied to the *Le Temps* corpus, but we lacked the time to integrate those results to this project. As we have discussed in Section 5.3, detecting as correct all the title-case tokens significantly lowers the amount of false positives. This suggests that proper nouns are an issue, to which NER could be a solution — provided the results remain reliable despite the noisy input.

We based our error detection model on the Hunspell software, which we favored due to its low false positive rate. However, we have seen in Section 5.2 that the Le*fff* also performs well at detecting errors. It could be interesting to see how the whole OCR correction pipeline would behave if we replace Hunspell by the Le*fff*.

We investigated the use of more advanced probabilistic models, like *statistical machine translation* or the *Viterbi algorithm*, but ended up not using those models. This was because they require to compute the confusion matrix, which is a table describing how often a character was mistakenly recognized as another. Usually, this table is computed by comparing a large amount of manually corrected texts to the texts including the errors. The issue was that we did not had such a large annotated corpus available. However, Tong and Evans [24] describe an iterative method to learn the characters confusion table. This could be an interesting way to compute an estimation of the confusion table for the OCR algorithm used to digitize the *Le Temps* corpus.

Concerning the error correction, other directions could include the use of *Natural Language Processing* tools. For example, using *Part of Speech* tags, we could restrict the search of suggestions to some categories, depending on the tags of the adjacent tokens. As for named-entity recognition, this technique might not work in all situations, as when there are error bursts because the whole context of a sentence is noisy.

We have seen that the candidates generation process of our algorithm was performing poorly. Indeed, the expected corrections were rarely generated. Instead of trying to find replacements based on the tokens detected as incorrect, Mei et al. [17] propose a different approach. In their correction technique, they first build a set with all the words from a lexicon that are within a limited edit distance of a token to correct. Then, scores are computed for each candidate based on the edit distance, the longest common subsequence, the frequency of the candidate and the context popularity, that is if this candidate is coherent within the n-gram contexts of the token to correct. The final choice of the correction is done using a regression model. Because a lot more words will appear in the candidate sets, this approach is much more likely to include the expected corrections among the candidates. However, it is unclear how fast this method would run, as computing the Levenshtein distance for every word in a lexicon and for every token detected as an error would be a computationally heavy operation.

7 Acknowledgments

I would like to thank Maud Ehrmann for the support during the whole project and for the really good suggestions of improvements of this report. Pierre Runavot provided numerous advices about the different models. I would like to express my gratitude for the time they spent helping me completing this challenging, but interesting project.

References

[1] Haithem Afli et al. “Using SMT for OCR Error Correction of Historical Texts”. In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation LREC 2016, Portorož, Slovenia, May 23-28, 2016.* 2016. URL: `http://www.lrec-conf.org/proceedings/lrec2016/summaries/280.html`.

[2] Mathias Berglund et al. “Bidirectional Recurrent Neural Networks as Generative Models”. In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Mon-*

treal, Quebec, Canada. 2015, pp. 856–864. URL: http://papers.nips.cc/paper/5651-bidirectional-recurrent-neural-networks-as-generative-models.

[3] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python.* 1st. O'Reilly Media, Inc., 2009.

[4] Piotr Bojanowski et al. "Enriching Word Vectors with Subword Information". In: *CoRR* abs/1607.04606 (2016). URL: http://arxiv.org/abs/1607.04606.

[5] Nicolas Bornand et al. *A Study of Linguistic Drift On Le Temps Newspaper Corpus.* 2015. URL: https://github.com/AblionGE/A-study-of-linguistic-drift-on-Le-Temps-Newspaper-Corpus/blob/master/report/LinguisticDriftReport.pdf.

[6] Angela Chambers and Séverine Rostand. "The Chambers-Rostand Corpus of Journalistic French". In: *Oxford Text Archive* (2005). URL: http://ota.ox.ac.uk/desc/2491.

[7] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL.* 2014, pp. 1724–1734. URL: http://aclweb.org/anthology/D/D14/D14-1179.pdf.

[8] François Chollet. *Keras.* https://github.com/fchollet/keras. 2015.

[9] Aymeric Damien et al. *TFLearn.* https://github.com/tflearn/tflearn. 2016.

[10] Mohammad Ali Elmi and Martha W. Evens. "Spelling Correction Using Context". In: *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, COLING-ACL '98, August 10-14, 1998, Université de Montréal, Montréal, Quebec, Canada. Proceedings of the Conference.* 1998, pp. 360–364. URL: http://aclweb.org/anthology/P/P98/P98-1059.pdf.

[11] William Gale and Geoffrey Sampson. "Good-Turing smoothing without tears". In: *Journal of Quantitative Linguistics* 2.3 (1995), pp. 217–237.

[12] Irving J Good. "The population frequencies of species and the estimation of population parameters". In: *Biometrika* 40.3-4 (1953), pp. 237–264.

[13] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[14] Yoon Kim et al. "Character-Aware Neural Language Models". In: *CoRR* abs/1508.06615 (2015). URL: http://arxiv.org/abs/1508.06615.

[15] Karen Kukich. "Techniques for Automatically Correcting Words in Text". In: *ACM Comput. Surv.* 24.4 (1992), pp. 377–439. DOI: 10.1145/146370.146380. URL: http://doi.acm.org/10.1145/146370.146380.

[16] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[17] Jie Mei et al. "Statistical Learning for OCR Text Correction". In: *CoRR* abs/1611.06950 (2016). URL: http://arxiv.org/abs/1611.06950.

[18] Arnaud Miribel. *Correcting the Optical Character Recognition of Swiss daily newspaper Le Temps.* 2016.

[19] Hermann Ney, Ute Essen, and Reinhard Kneser. "On the Estimation of 'Small' Probabilities by Leaving-One-Out". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 17.12 (1995), pp. 1202–1212. DOI: 10.1109/34.476512. URL: http://dx.doi.org/10.1109/34.476512.

[20] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[21] Benoît Sagot. "The Lefff, a Freely Available and Large-coverage Morphological and Syntactic Lexicon for French". In: *Proceedings of the International Conference on Language Resources and Evaluation, LREC 2010, 17-23 May 2010, Valletta, Malta.* 2010. URL: http://www.lrec-conf.org/proceedings/lrec2010/summaries/701.html.

[22] Mike Schuster and Kuldip K. Paliwal. "Bidirectional recurrent neural networks". In: *IEEE Trans. Signal Processing* 45.11 (1997), pp. 2673–2681. DOI: 10.1109/78.650093. URL: http://dx.doi.org/10.1109/78.650093.

[23] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958. URL: http://dl.acm.org/citation.cfm?id=2670313.

[24] Xian Tong and David A. Evans. "A Statistical Approach to Automatic OCR Error Correction In Context". In: *Proceedings of the Fourth Workshop on Very Large Corpora (WVLC-4*. 1996, pp. 88–100.

[25] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. "Recurrent Neural Network Regularization". In: *CoRR* abs/1409.2329 (2014). URL: http://arxiv.org/abs/1409.2329.