Project Report

Noisy Channel And N-gram Model Based Spelling Correction

Mariam Bangura

Department of Language Science & Technology, Saarland University

Computational Linguistics

Prof. Dr. Alexander Koller

March 19, 2022

Noisy Channel And N-gram Model Based Spelling Correction

Automatic spelling correction is widely used in the modern world. With the huge amount of the text information, we deal with daily, it is impossible to strictly follow grammar rules or simply not to do errors. Thus, spelling correctors always come at handy by suggesting corrections for typos while using search engines, messengers or text editors.

The current project is devoted to the implementation of a spelling checker with noisy channel and n-gram models using Python. Although, it is not the most accurate and efficient algorithm, its implementation and analysis provide an insight on what affects spelling checker performance and how it can be improved. In particular, the algorithm aims to correct spelling errors by finding the most probable word existing in a language corpus as a candidate for correction. In other words, the correct word is the one that maximizes the probability of a word ***w*** given the error ***e*** in a vocabulary V:

$$\hat{\mathrm{w}} = \underset{w \in V}{\operatorname{argmax}} P(w|e) = \underset{w \in V}{\operatorname{argmax}} P(e|w)P(w)$$

where P(e|w) is a likelihood or the noisy channel and P(w) is prior or a language model that can be modelled with n-grams.

Spelling Correction: Python Implementation[1]

Noisy Channel Modelling

Creating Confusion Matrices with Error Counts

It is implied that the majority of the misspellings are 1 edit distance (ED) from the correct word, and the rest is usually at 2-ED from it. Following Jurafsky & Martin (2021), the current implementation considers candidate words, which are at most 2-ED from error according to the Damerau-Levenshtein notion of ED. Thus, the deletion of a letter, insertion of an extra letter, substitution of one letter with another or transposition (replacement) of two adjacent letters is considered 1-ED.

[1] If the script where the function is defined is not mentioned, then it is in the *SP_func.py* script

The P(e|w) is the probability of an incorrect letter given a correct letter (or pairs of letters). In general, what is needed in order to calculate P(e|w) are error counts for each letter that can be derived from an error corpus: e.g. how often is “a” substituted with “o” or how often is “te” written as “t” (in other words, how often is “e” deleted if written after “t”).

The goal of obtaining these counts is achieved by using a Birkbeck error corpus (assembled by Roger Milton[2]), which contains correct spellings (6136 words) and misspellings (36133) associated with them[3] (“missp.dat” file). The error corpus was read as a dictionary with correct spellings as keys and lists of misspellings as values. Additionally, all letters in strings appearing in the dictionary were presented in lower case, and “`” sign symbolizing the word boundary was added to the beginning and to the end of each word string: this allows to represent deletions or insertions that occur at the word boundary (e.g. the deletion of “s” from the word “`cats`” can be represented as “s” -> “`”).

The next step was to create four confusion matrices: one for each of the error types. Each matrix has the same number of rows and columns which corresponds to the alphabet (in our case it consists of “a-z” lower case letters and “`’-_” signs).

Firstly, a dictionary with empty matrices of a needed size, where the key is the error type and the value is a matrix, is created. Secondly, the algorithm identifies the type of an error in each of misspellings obtained from error corpus (“edit1_errors” function). For each of these types of errors, the corresponding cell in the matrix incremented by one (“fill_conf_mtrx” function). The logic of an algorithm defining the error type is the following:

“deletion”: a correct word is one character longer than a misspelling and there is a character that can be deleted from a word, so that word equals misspelling (if “ab” written as “a”, (a, b) cell is incremented).

“insert”: a correct word is one character shorter than a misspelling and there is a character that can be deleted from a misspelling, so that word string equals misspelling (“a” -> “ab”: (a,b) cell).

[2] https://www.dcs.bbk.ac.uk/~ROGER/corpora.html

[3] Correct and incorrect spellings containing “.” or “_” were excluded for simplification: “.” is likely to impair the sentence tokenization; “_” marks the space, but the created algorithm aims to find only one word correction.

"substitution": word and misspelling lengths are equal and there is only one pair of different characters, if a pairwise comparison between characters in a word and misspelling is performed ("a" -> "b": (a,b) cell).

"transposition": word and misspelling lengths are equal, the collection of characters they contain is the same and there are two pairs of adjacent characters that differ, if a pairwise comparison between characters in a word and misspelling is performed ("ab" -> "ba": (a,b) cell).

First, the "fill_conf_mtrx" identifies errors at 1-ED by using "edit1_errors" on a word. Second, "edit1_errors" is run again, but this time the identified 1-ED-misspellings are treated as correct words and all other misspellings of the word are treated as misspellings of these 1-ED-misspellings. If there is a 1-ED between misspellings, then the misspelling of a 2-ED distance from a word is found: the corresponding cell in a confusion matrix is incremented by one as well as the cell incremented at the first iteration incremented once again (in order to account for two errors). Note that one 2-ED error can result from multiple 1-ED errors, which could lead to inflated error counts in the table. The algorithm does not consider the potential 2-ED error, if it was already derived from some other 1-ED error.

Corpus Reading

Following P. Norvig (2007), the corpus "big.txt" consisting of Project Gutenberg's book excerpts and most used words lists from Wiktionary and British National Corpus (about 1 million words in total) is used as a language model. The first step is to tokenize the corpus into sentences and then into words, which was performed with the "sentence_reader" function. The sentence tokenization is initially performed with "sent_tokenize" function from the Natural Language Toolkit (NLTK) library. However, some sentences in the text do not end with periods (for example, titles) and the "sent_tokenize" function does not capture it. Therefore, the tokenization is improved further. The algorithm iterates over each sentence, and if the sentence contains two **\n** symbols in a row or one **\n** succeeded by a capital letter or **\t** sign, then this sentence is split into two different sentences at this spot.

After that the sentence (or two sentences, if a sentence was split) is tokenized into words by using “RegExpTokenizer” from NLTK, which allows to use a regular expression (regexp) for tokenization. For creating a new instance of “RegExpTokenizer”, a regexp pattern that defines what strings are considered to be a word is passed. In the current project, the word is a string of characters that can contain all English letters in any order (including capital ones), no more than one apostrophe in any position (so that words like “don’t”, “mothers’” “’cause” are considered one word) and/or at most one dash “-“ or underscore “_” signs in middle position. The pattern was chosen based on brief visual inspection of the words in the corpus. Note that for the sake of the project the assumption is made that the words in the corpus are correct words, although they still can be incorrect according to spelling dictionaries.

At the end of the word tokenization the <#> sign is added at the end of each sentence. The result of the tokenization is the list of words in the order of their appearance in the corpus.

Channel Model (P(e|w)) Calculation

For P(e|w) calculation, “p_typo_char” function is used. It takes the misspelling, confusion matrix, corpus and dictionaries of counts of all characters and all character pairs found in the corpus. If the model is allowed to propose no corrections for the misspellings existing in the corpus, then function checks the misspelling’s presence in the corpus, and if it is there, adds it as a possible candidate correction. However, as it is 0-ED correction, P(e|w) is not calculated.

Next step is to generate all possible 1-ED correction candidates for the misspelling, which is performed by the “generate_cors” function (it is run inside the “p_typo_char”). The result is a list of all possible 1-ED correction candidates. Lists with candidates also include the error type, indices of altered letters in the alphabet and the misspelling the candidate was generated from. With regards to error types, in case the candidate was generated per deletion, the error type that is assumed in the misspelling is insertion (consequently, the generation with insertion entails the error type “deletion”). The pair of letter indices (or row-column coordinates of the error count in the confusion matrix) is the same for these two types of errors (e.g. the insertion “a” -> “ab” results in an increment in (a,b) cell, as well as the deletion “ab” -> “a”), and for that reason are written exactly the same as in “edit1_errors” function; for the substitution and transposition the error type corresponds to the generation method, however the coordinates of the cell in the confusion matrix are stored in the inversed order in order to reflect the error and not the correction (e.g. if a candidate is generated by the “a” -> “o” substitution, (o,a) coordinates are stored, because the candidate corrects the “o” -> “a” error).

Some of the 1-ED candidates can be non-words, but they are needed in order to generate all possible 2-ED candidates from them with the “generate_cors2” function: this function actually runs “generate_cors” on already generated 1-ED candidates, thus, yielding candidates that are at the 2-ED distance from the initial misspelling. Additionally, the “generate_cors2” filters out non-words (candidates that are not observed in the corpus) and candidates that happened to be the same as the initial misspelling (e.g. the 1-ED distance candidate for “abc” could be “abcd”, but the 2-ED could be again “abc” resulting in actual 0-ED distance from the initial misspelling).

The P(e|w) is calculated as follows: the count of the error is divided by the count of the correct spellings. The denominator for deletions and transpositions is a count of character pairs in the corpus (for “ab” -> “a” deletion or “ab” - > “ba” transposition, the denominator is the “ab” count) and for insertions and substitutions it is a single character count (for “a” -> “ab” insertion or “a” -> “b” substitution, the denominator is the “a” count). In order to account for characters or character pairs that occur in misspellings, but do not occur in the corpus, the Laplace smoothing is performed.

In case the calculation is performed for the 1-ED candidate the negative natural logarithm (logprob) of the obtained P(e|w) value is stored. For 2-ED candidates one additional step is needed: the logprob of the 2-ED candidate is summed with the logprob of the 1-ED candidate it was generated from, because of the underlying assumption that two errors are done independently from one another (for that reason P(e|w) for non-word 1-ED candidates are calculated as well, as they could potentially yield 2-ED candidates that are in fact in corpus).

Language Model

For the language n-gram modelling, “ngram_counter” and “ngram_model” functions are used. The former takes language corpus and the n-gram size as parameters and returns two dictionaries: one with n-grams as keys and their counts in a corpus as values and another one with (n-1)-grams as keys and their counts as values. For example, if n=3, the “ngram_counter” returns count of all word triples and all word pairs existing in the corpus.

The ”ngram_model” function takes the word of interest as a parameter, dictionaries produced by “ngram_counter”, and optionally preceding (pre-text) and succeeding context (post-text) (e.g. for the unigram both context parameters should remain empty) and calculates the word probability using Laplace smoothing. First, the function checks if our model is unigram, and if it is the case, calculates the logprob of the word in the corpus. If n>1, the function returns the sum of logprobs for the word in a pre-text and a post-text (e.g. if only pre-text is present, the logprob for a post-text equals 0).

Choosing The Best Correction

The “cand_posterior” function is used for the posterior probability calculation of all correction candidates proposed for the particular misspelling. The function creates a list of tuples containing proposed corrections and their posteriors. The “best_correction” function chooses the candidate that exists in the corpus (remember, the 1-ED non-word corrections were not filtered out yet) and has the lowest posterior logprob in comparison to other candidates. If there are no candidates that fit both criteria, the function returns the misspelling itself.

Spelling Correction Evaluation

For the evaluation of the algorithm, the Peter Norvig’s test script[4] was used. Additionally, the performance of the current noisy channel model (NC) was compared to a simpler Peter Norvig’s model.

[4] https://nlp.fi.muni.cz/trac/research/wiki/en/AdvancedNlpCourse/AutomaticCorrection - this version of P. Norvig’s script is a bit different from the one on his website in terms of the structure, but has the same functionality and produces same results

Peter Norvig's Spelling Correction Model

The model is to some extent similar to the NC model implemented in the current project: it also makes the assumption that the most probable typos occur at up to 2-ED. However, the probabilistic model is rather simple: it is assumed that the lower the ED of correction from the misspelling, the higher is the probability of correction. Thus, the misspelling itself, if it happens to exist in the corpus (the same corpus used in the project) would be the first-choice correction (0-ED); the next choice would be the 1-ED correction with the highest frequency in the corpus in comparison to other 1-ED corrections; if there are no appropriate 1-ED corrections, then similarly the 2-ED correction is chosen, and if no corrections are found in the corpus, the misspelling is returned.

Test Script

Two datasets of correct target words and their misspellings (*testsets.py*), as well as testing algorithm provided in P. Norvig's script are used for testing. The "spelltest" function (*spell.py*) passes misspellings for every target word through "correct" function, and if the resulting correction does not match the target word, the variable "bad" is incremented by one. The "spelltest" function was corrected to make it possible to test the NC model in case of a unigram language model ("spelltest_unigram" function in *SP_main.py*). Additionally, the "spelltest_ngram" function (*SP_main.py)* was created for the evaluation of n-gram models with n>1. Initially, there is no provided context in P. Norvig's test datasets. In order to add some context, the following method was used ("test_ngram" and "ctx_dict" functions[5]): for every target word, the most frequent cooccurring context of an n-gram size was found in the corpus, and thus, the "SP_correct" function (*SP_main.py*) tried to find the best correction for misspelling, if it is surrounded by this context. The "spelltest_ngram" also allows to test the performance with the provided post-context.

In case, if the target word from a test set does not exist in the corpus, its misspellings are not passed to the algorithm: although initially in Peter Norvig's script the unknown targets count as

[5] Functions are appropriate for models with pre-text (post-text is optional)

"bad", as the algorithm cannot propose valid corrections that are not in the corpus. However, it was changed for the current project, as at least in case of testing the n-gram models, it is impossible to find any context to words that do not exist in the corpus. Thus, unknown targets are simply not used in a testing procedure not only for the project's script, but also for the Norvig's algorithm, in order to make results comparable.

After all words are processed, the script calculates the success rate as the proportion of the correct corrections to the number of all processed misspellings.

Model Performance

Table 1

Test Performance of Spelling Correction Models

	Peter Norvig's Spelling Corrector	NC Models			
		Unigram	Bigram (pre-text only)	Bigram (pre-text and post-text)	Trigram (pre-text only)
Test 1 n = 256	79%, 3s	80%, 127s (81%)	85%, 82s (87%)	81%, 82s (82%)	86%, 82s (87%)
Test 2 n = 357	75%, 4s	78%, 194s (80%)	81%, 126s (83%)	75%, 126s (76%)	81%, 127s (83%)

Note. The table shows the percentage of proposed corrections that matched targets and the duration of the test in seconds. In the brackets, the percentage of matched targets is indicated if the algorithm was not allowed to suggest the misspelling existing in a corpus as a correction. The 'n' indicates the total number of processed misspellings.

As can be seen from the Table 1, P. Norvig's algorithm demonstrates lower percentage of successful corrections in comparison to NC models but tends to be significantly faster. The size of n-grams generally leads to the performance increase: bigram model performs much better than the unigram one. However, the trigram model only slightly outperforms the bigram with pre-text and only in the test 1. Notably, the bigram model with pre-text and post-text, performs only 1% better than the unigram model in the test 1, but does worse in the test 2. Perhaps, the post-text is not especially crucial for the correction decision as pre-text (in particular, for this corpus).

Interestingly, if the misspelling existing in a corpus is not used in a model as a potential correction candidate, the accuracy of the model slightly increases. However, that might be due to the specifics of the used test set or/and the language corpus. Nonetheless, in general, the observed trend still holds with this configuration.

Thus, the performance of the bigram and trigram models with the pre-text only is the highest in terms of the success rate, but that advantage comes at cost of a much longer processing time. Moreover, the testing time does not include the runtime of the algorithm initialization for NC models (e.g. creating counters for corpus, filling in confusion matrix, etc.), which takes about 40 seconds[6] in comparison to P. Norvig's model whose initialization duration takes less than a second.

Improvements Suggestions

Language Model

The model used in the project does not allow to propose valid correction candidates that are not in the language corpus. P. Norvig (2007) suggests that the solution could be to use morphology-based language models, so that model can correct morphemes based on their probability distribution, even if the whole resulting word is not in the corpus.

The introduction of some parameters defining language regional properties would improve model as well. For instance, the probabilistic model usually suggested the word "organizational" for various misspellings of the word "organisational", which was counted by the test as fail. However, both spellings are correct depending on whether it is British or American English.

Noisy Channel

There are several ways to improve the channel model. As Jurafsky & Martin (2021) mention, citing Bill & Moore (2000), the probability of an occurring error can be conditioned on its place in the word. Additionally, following Toutanova & Moore (2002) the probability can be conditioned not only on the spelling, but also on pronunciation, by considering how the letter string is transformed into

[6] The search for most common context for targets is time consuming as well, however, its goal is to adapt the testing script to n-gram models, and for that reason I would not consider it as a measure of the time efficiency.

phonemes. The other possible improvement suggestion is the usage of the expectation maximization algorithm for confusion matrices computation by iteratively running the spelling correction algorithm itself and updating error probabilities each time (Kernighan, 1990).

Runtime

The runtime could be improved by using more time efficient functions or algorithms, especially for correction candidates generation (e.g. somehow narrow down the number of generated corrections).

Evaluation

It is possible that the model performance is underestimated by the evaluation procedure to some extent. For example, the issue with differences in British and American English spelling could be also solved by including both possible spellings into the test set as target words. Additionally, the test set is biased to always demand the correction of the misspelling, which does not allow to test the ability of the algorithm to determine, whether the misspelling is actually a misspelling, and as the test results demonstrate, when the model is allowed to not correct misspellings, its performance worsens. The other problem is specifically tied to n-gram models: the context of the misspelling was chosen based on the most frequent context for the target word in the corpus, but that could be not the best decision, as some contexts were relatively frequent for the specific targets, but others could occur only once. In the future, one could possibly come up with the more balanced test set for n-gram based spelling correction.

Conclusions

To sum up, the use of noisy channel and n-grams is beneficial for the accuracy of the spelling correction. Nonetheless, there is still room for improvement of the current implementation, mostly with regards to the runtime efficiency and accuracy.

Literature

Jurafsky, D., & Martin, J. H. (2021). Appendix B: Spelling Correction and the Noisy Channel. In *Speech and Language Processing*. https://web.stanford.edu/~jurafsky/slp3/

Norvig, P. (2007). *How to Write a Spelling Corrector*. http://norvig.com/spell-correct.html