AUTOMATED TYPO CORRECTOR

Prepared for

Dr Vishal Gupta

Instructor In-charge, Artificial Intelligence (CS F407)
Department of Computer Science & Information Systems
Birla Institute of Technology & Science
PILANI, 333031 Rajasthan, INDIA

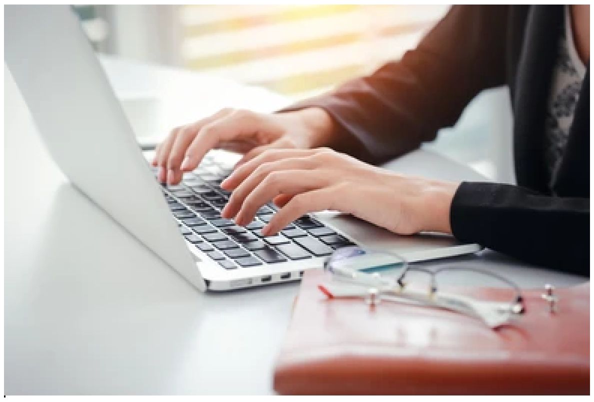
PREPARED BY

SHASHANK AGRAWAL (2020A7PS0073P)

SHADAN HUSSAIN (2020A7PS0134P)

ARYAN MADAAN (2020A1PS0222P)

Introduction



The ability to communicate effectively is crucial in today's world. However, even the most skilled writers and typists can make mistakes, such as typos, misspellings, and grammatical errors. These errors can be embarrassing and lead to misunderstandings. In some cases, they can even be costly, such as when an email containing a typo is sent to an important client.

To address this problem, we have developed a typo error correction model from scratch using the classic spelling-checking model. The Model is a modified version of Peter Norvigs' Spell-checker accommodating several heuristics for typos.

In this study, we will also examine the existing literature, identify current issues and propose potential future study areas. Since the algorithm for checking errors depends on the particular language, here we provide analysis solely for the English language.

In this project, we will describe the methodology and results of our typo error corrector, as well as its potential applications and future developments.

Different Types of Spell-Checking Algorithms

Here we will discuss different types of Algorithms, a brief history, how they work, and a comparison between them.

Naive Approach:-

The problem of checking errors in words can be trivially boiled down to checking for that word in a dictionary. Now, this way we can say whether or not a spelling/typo error is present or not, but to suggest a spelling suggestion we need to find the "closest" word in the dictionary to the query word.
How can we define this "closeness"?
There are several ways to compute the distance between two words, one of which is the "edit distance" or the Levenshtein distance between two words. The edit distance between word1 and word2 is trivially the number of steps that are required for word1 for becoming word2, by performing any of the 4 operations(insert, delete, replace or transpose).

We can compute the edit distance from the query term to each dictionary term, before selecting the string(s) of minimum edit distance as a spelling suggestion. Let the average length of every dictionary word be ld and the length of input word be len. Converting our input word to the given word will require in the worst case a time complexity of O(26*ld*len). Now if there are N words in the dictionary, our time complexity will be O(26*N*ld*len). The program will store all the dictionary words in an array at run time by reading a text file. Then it will compute all the edit distances and select the word with minimum edit distance. Thus the run time space complexity will be the number of words in the dictionary, i.e. O(N). One way to improve the space complexity is to store the dictionary in a trie data structure. However, the time complexity will still remain the same.

Limitations:-

- Requires a high amount of computing power.
- Very slow for providing instantaneous suggestions.

Peter Norvig's Approach :- http://norvig.com/spell-correct.html
Peter Norvig (born December 14, 1956) is an American computer scientist and Distinguished Education Fellow at the Stanford Institute for Human-Centered AI. He has discussed a basic spell checking model approach that achieves 80 or 90% accuracy at a processing speed of at least 10 words per second in about half a page of code.

His approach is based on the following heuristics:

1.) Majority of the misspelled words are utmost at edit distance 2 from any word of the dictionary.
2.) Words at edit distance 1 are always having higher probability than the words at edit distance of 2.

He generates all possible terms within an edit distance of 2 by deleting, inserting, replacing and transposing the letters (the standard 4 operations we discussed above) and then searches each of them in a dictionary.

For a word of length n, an alphabet size a, an edit distance d=1, there will be

- n deletions
- n-1 transpositions
- a*n alterations
- a*(n+1) insertions

for a total of 2n+2an+a-1 terms at search time. This ways the overall complexity is O(a*n) order, just for edit distance 1. If we go till distance of 2 complexity is of O(a*n*n).

Analysis:-

- Much better than the previous approach since it doesn't require iterating for every word of the dictionary. We generate the possible edits and then just check for their presence in the dictionary, which can be done in constant time by hash data structures.
- Highly language dependent. Some languages may have a lot of characters, due to which Time complexity may increase a lot.
- Due to 4 operations, it's still expensive to edit distance 2 checking as there are a lot of words to check at the second stage in runtime.

Symmetric Delete Spelling Correction (SymSpell) :

Here we do a little more pre calculations on our dictionary. We generate all terms with edit distances(or deletes only) and add it together with the dictionary terms. While checking for any query word, we again generates edits(again comprising only of deletes) and search them in the dictionary

There will be only n deletions for a total of n terms at search time for a word of length n, alphabet size a, and edit distance 1.

Now, we are required to account for insertions,deletions,transpose and replacement edits that were being used in Norvig's approach by just delete operations. We can do a total of n deletions on the input word and compare it with the original words as well as the words with deleted letters that had been stored in the dictionary previously.

1. Insert : Inserting a letter into the input word is the same as deleting a letter from a word in the dictionary and comparing it with the input word.
2. Delete: Deleting a letter from input word and comparing it words with deletions in the dictionary.
3. Replace: If a particular letter on replacement in a word generates an actual word, then on deletion of that letter from the input there will exist a corresponding deletion on the actual word that will match our deletion.
4. Transpose: If the input word is a transpose of the actual word then deleting different alternate positions on the input and the actual word will generate a match.
   Eg. "artcile" -> deleting c gives "artile"
   "article" -> deleting c gives "artile"

It is an excellent example of meet in the middle technique used in algorithms to reduce complexity, and serves here as well.
A major time complexity improvement in this code comes from the fact that we do not need to iterate on alphabets to generate replacements or inserts. For example, this technique is approximately six orders of magnitude faster on edit distance 3 in the English language. This is especially advantageous when the alphabet is large. Thus, it also makes the technique's time complexity independent of the alphabet size of the language.

Limitations:-
It's very powerful and fast but uses a lot of space.
E.g. for a maximum edit distance of 2 and an average word length of 5 and 100,000 dictionary entries we need to additionally store 1,500,000 deletes.

Improvements over Norvig’s approach for typo checking

We identified two main improvements over Norvig's approach that are highly suitable for typo corrections. The 4 operations that were considered are still valid for typos as for spellings, however the probability of each of them should not be the same. We name the two improvement heuristics as H1 and H2 respectively.

1.) H1 – While typing it’s very less likely for a transpose error to occur than a insert or delete kind of error. All 4 operations are not equiprobable.

2.) H2 – For Replacements, the probability of having a close letter to the original word in the keyboard, is much higher than a by far letter. For eg: if the typed word is “lpck”, it's much more probable that the intended word was “lock”, rather than “lack”. If one checks the dictionary for probabilities, one would find “lack” to be more common than “lock”, hence generating a wrong outcome. This is a major issue that needs to be addressed in Norvig's approach.

- ☐ Accounting for the first problem, we went through many research papers having analysis of common typo errors. According to et. al Peterson** of all typing errors
  - Transposition errors are : 13.1%
  - One extra letter errors are : 20.3%
  - One missing letter errors are : 34.4%
  - One wrong letter errors are : 26.9%

This data has been gathered after analysis of plenty of web documents.

We have designed our corrector taking this data as a base.

** See this for an actual research paper.

☐ For the second problem, we linked the probabilities to the "key distances" of a qwerty keyboard. Specifically, the more far a key from the to be replaced key, the less its chances. So the probability shares an inversely proportional relationship with the key distances. For replacement [a -> b] the probability can be computed as

$$P(a,\ b) \ = \ \frac{\frac{1}{dist(a,b)}}{\frac{1}{dist(a,b)}+\frac{1}{dist(a,c)}+\frac{1}{dist(a,d)}\ \ldots\ldots+.\frac{1}{dist(a,z)}}$$

Now the Replacement probability will become P(R)*P(a, b) – instead of ¼ originally taken in Norvig's Approach.

Implementation –

Let the typed word be w, and the intended correct word are c's

We want to compute argmax(P(ci|w)), that is the correct word is ci, given the typed word is w.

P(ci|w) = $\frac{P(w|ci)*P(ci)}{P(w)}$

Since P(w) is the same for every possible candidate c, we will factor it out, giving:

argmaxc ∈ candidates P(c) * P(w|c)

P(c) is computed from the corpus text, a big text file containing thousands of words(generated from Project gutenberg and wikidictionary).

Till now we are moving in accordance with Norvig.

P(w|c) is dependent solely on edit distance in the earlier, but we saw it is dependent on the type of the error a lot.

However the hypothesis that for a w having edit distance 2, P(w|c) should be very less than a w having edit distance 1 still holds.

So P(w_d| c) = P(op) for, op = insert, delete or transpose.

= P(op) * P(a,b) for op = replace char ‘a’ with char ‘b’

Computing these probabilities we ran the model on 2 types on datasets -

1. Spelling Errors – dataset taken from Norvig’s website
2. Typing Errors - dataset generated using the latest chatGPT AI model 🙂, having around 45% real typing errors.

The accuracies and results for the same are shown in the following table

Model Used	Accuracy on dataset with Spelling Errors.	Accuracy on dataset with Typo Errors.
Norvig Original	75%	68%
Norvig + H1	75%	84%
Norvig + H1 + H2	76%	**92.46%**

As the stats clearly reflects, for spelling errors the new heuristics(H1 and H2) added do not improve the accuracy significantly, however for documents typos there is a marvelous improvement. The average accuracy was around 92.5% and The max accuracy achieved for typos was around 99.5% for a document which is surreal.

Conclusions

By using simple heuristics and modeling we were able to use a spell checker for typo checking with great accuracy, checked on real documents having typo errors.

Future Works

Many other minor error heuristics can be thought of in typos.

1.) Certain transpositions are much more common than the others, for example writing “friend” as “freind" or “field” as “feild” or “being” as “bieng”. Some study can be done on common transposition error patterns and models can be more improved.

2.) Slips while typing – Another kind of error which is different from what we discussed. For eg. party for partyyy.

3.) Accommodate special characters, Numbers, Uppercase errors as well for the alphabet.

References

1.) Peter Norvig – How to write a spelling corrector https://norvig.com/spell-correct.html

2.) Wolf Garbe – 1000x Faster Spelling Correction algorithm
https://wolfgarbe.medium.com/1000x-faster-spelling-correction-algorithm-2012-8701fcd87a5f

3.) Data on KEYBOARD DISTANCES
https://gist.github.com/pxeger/ae20549834c04f1e40a4fa4ba91a0079

4.) Note on Undetected Typing Errors – James L Peterson
https://dl.acm.org/doi/pdf/10.1145/6138.6146#:~:text=In%20general%2C%20the%20mistake%20will,than%20or%20equal%20to%20577.

CODE REPOSITORIES

The complete source code can be found here –
https://github.com/Shashank22082002/spell-checker.git