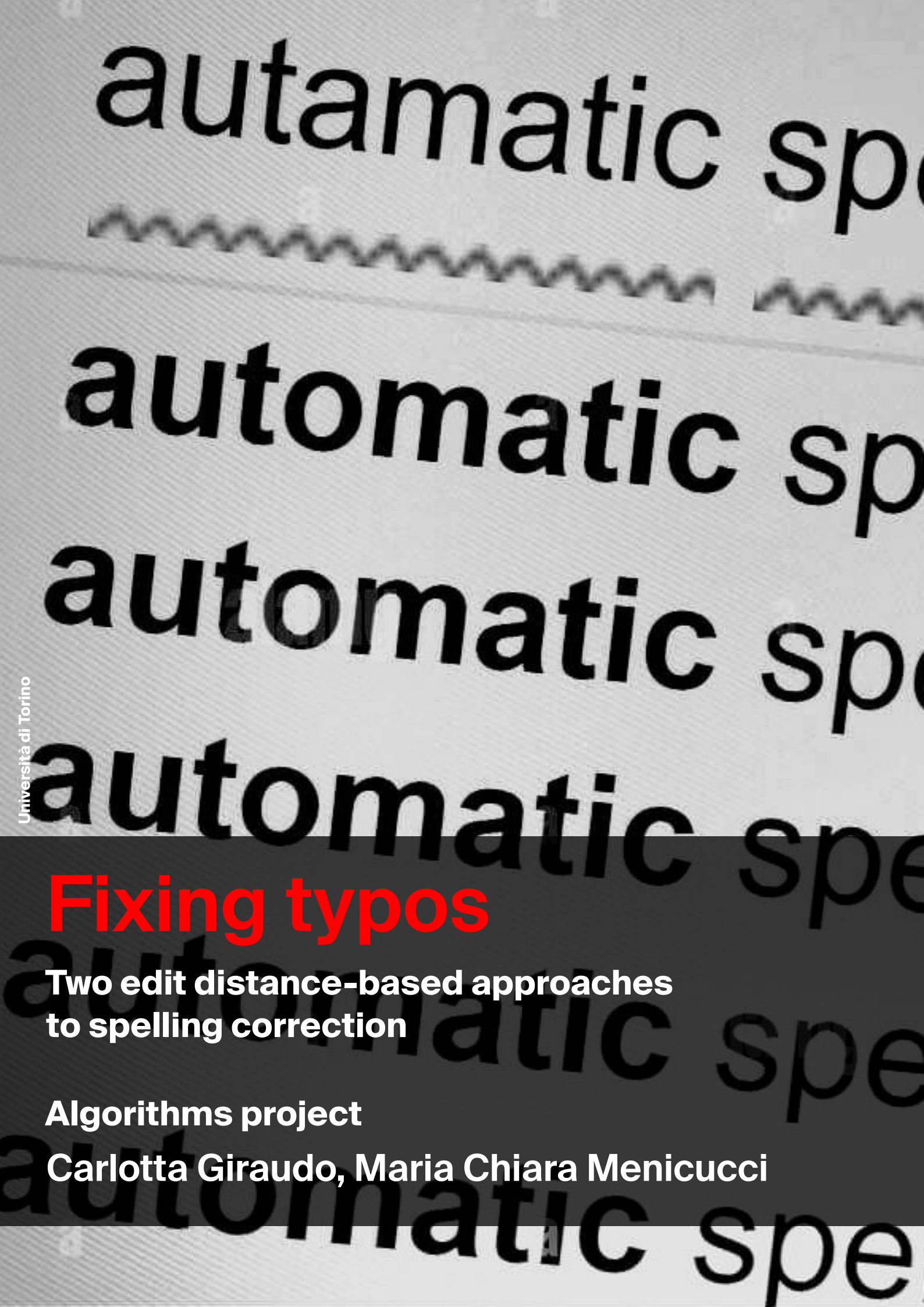


Università di Torino

Fixing typos

Two edit distance-based approaches to spelling correction

Algorithms project

Carlotta Giraudo, Maria Chiara Menicucci

Fixing typos

Two edit distance-based approaches to spelling correction

by

Carlotta Giraudo, Maria Chiara Menicucci

Student Name	Student Number
Carlotta Giraudo	975787
Maria Chiara Menicucci	1165302

Contents

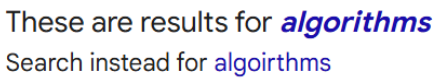
1

Introduction

In this chapter we introduce the problem of spelling correction and review some techniques that have been developed to tackle it.

1.1. Problem presentation

In everyday life it is very common to make mistakes while typing on a keyboard. A misspelled word can change the meaning of a sentence, leading to confusion and miscommunication. Consequently, spelling correction has become a common task in the field of Natural Language Processing and several algorithms have been implemented to fix this kind of mistakes: for instance, if in a Google search we type *algoirthms* instead of *algorithms*, Google gives results for the correct word, preceded by a banner stating



The idea underpinning some of the techniques that have been developed to tackle the problem - and the one underpinning this project - is to consider possible changes in the misspelled term and eventually find, through some rules, a connection with a word in a dictionary, which is proposed as correction. In general, there could be more than one possible solution. For instance, imagine we wrote

”peath”

The right word would be ”peach”? Or ”path”? This is why is important to establish clearly how the algorithm should choose the suggestion to propose to the user.

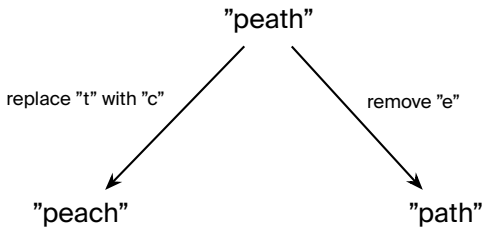
In this work, we stick to spelling correction in the English language, for words without any capital letter, spaces between them or symbols.

1.2. Literature review

In this section we will give a brief overview of the numerous algorithms which were developed in order to find the best solution. These can be roughly divided in two classes: algorithms who rely on deep learning and algorithms who do not.

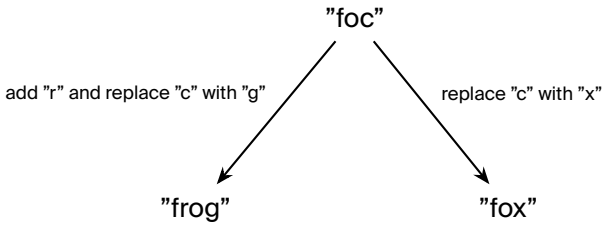
Peter Norvig's algorithm [5], the BK-tree algorithm [1] and SymSpell algorithm [2] belong to the latter class. The theoretical foundation of all these approaches is the edit distance: this measures how "far" two words are by counting the minimum number of single-character edits (such as deletions or insertions) we have to perform to change one word into the other. The next chapter provides a more detailed explanation about this concept.

To grasp quickly how edit distance works, let us consider the same example as above:



In this case both the new options require one operation, so they both have edit distance 1 from "peath". Therefore, the algorithms may propose both of them, if no other selection criteria are involved.

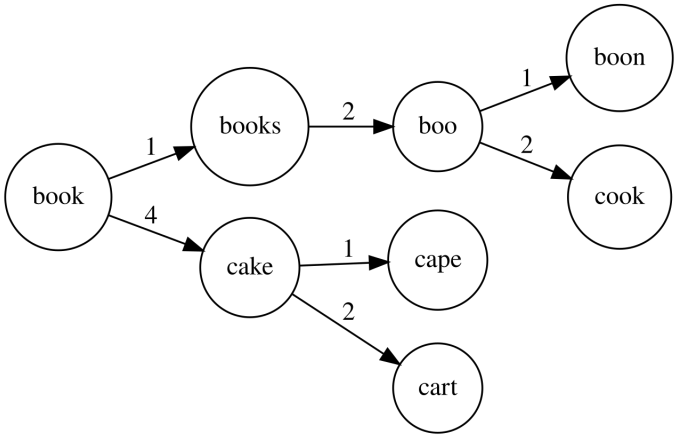
We now consider another possible case:



Here the first suggestion needs two different single-character edits, while the second one just a substitution; therefore the algorithms will suggest "fox", since it will give the priority to less distant options. The idea behind these algorithms, indeed, is that the correct version of a misspelled word is likely to have a small edit distance from it, so we are interested in finding words having a small edit distance from the input term.

Norvig's algorithm [5] and SymSpell [2] use this edit distance in order to fix the maximum number of edits to do on a term. If this is set equal to 2, for example, I can change *"cat"* into *"gas", "bat", "at"*, but not to *"cattle"*, since this transformation would require more than two changes. Apart from this similarity, the two algorithms will differ in the way they create this edited words: Norvig will make a set of all possible changes, while SymSpell focuses on deletions.

The BK-tree algorithm [1] instead uses the edit distance to build a metric tree, i. e. a tree structure where each node contains a word and the weight on each edge is the edit distance between one word and the other. The tree is built based on a dictionary of correct words, and to find the best correction for a misspelled term, the tree structure is searched (exploiting the triangle inequality holding for the edit distance).



Now, it is clear that all the approaches above, based on examining the misspelled word alone to correct the typo, have several limitations: the most serious is that they don't exploit the context of the word in the text, when present. This does not allow to detect spelling errors which produce existing words. For instance, in the sentence

Coding is there favorite activity

the context makes it clear that *there* is a misspelled word: the right word would be *their*. But if we inspect the sentence word by word, *there* will not raise any suspicion in itself, as it is an existing word.
More in general, the context provides precious information to understand which corrections make sense for a misspelled word: for instance, it allows to understand the syntactical role of the misspelled word, so that many candidate corrections may be quickly ruled out. More advanced and recently-developed models rely on neural networks in order to account for the context, as well, and thus have largely outperformed the accuracy of previous models, becoming quickly the new state of the art [4].

1.3. Data preparation

In this project we use two main datasets:

- A **frequency dictionary**, whose keys are about 80.000 distinct English words and the value associated to each of them is an indicator of the frequency with which the word is used in the everyday language. The higher the value, the higher the frequency. This frequency dictionary stands in as a language model, and the reason why we need it will be explained thoroughly in the next chapter. We import the frequency dictionary as a dictionary data structure in Python[1].
  The dictionary is the one used in the official implementation of SymSpell [3], and arises from the combination of Google Books Ngram data (which provides representative word frequencies) and SCOWL — Spell Checker Oriented Word Lists (which ensures genuine English vocabulary, but contained no word frequencies). To have an idea of the structure of the frequency dictionary, we report the first five most frequent words and their frequencies.

[1]The choice to have the word as the key is due to the fact that two words may have the same frequency, so this value cannot be used as key.

Word	Frequency
the	23135851162
of	13151942776
and	12997637966
to	12136980858
a	9081174698

- A **corrected-misspellings dictionary**: each key is a misspelled term, associated to its correct form. We have a total of 1000 misspellings. This dictionary will serve as a test dataset to check the accuracy of the algorithms: when testing the algorithms, we compare the correct version of the misspelled word with the correction proposed by the algorithm. The accuracy will be the ratio of correct proposed corrections.
  This dataset, as well, is imported as a dictionary to maintain this connection between the two terms.
  This dictionary was created by using the file proposed by the creator of SymSpell itself [3].
  We report the first five entries.

Misspelled	Correct
te	the
aojecm	project
gutenberg	gutenberg
eboo	ebook
yof	of

In order to test our algorithms on a single text, we will also use an artificial text with a misspelled version.

2

Theoretical basis

In this section we present the theoretical foundations of our algorithms.

2.1. The edit distance

Obviously, a good candidate correction for a misspelled word w is a word c which is *similar* to w. The concept of similarity is formalized through the mathematical notion of distance. In the following we discuss the edit distance, which essentially defines distance between words as the minimum number of edits required to turn a word into the other. Such concept is the theoretical foundation of the whole project.

Generally speaking, edits are single-character operations we can perform on a word w. We are interested in the following ones:

1. Deletion: remove a letter from w
2. Transposition: exchange two adjacent letters in w
3. Replacement: replace a letter in w with another letter
4. Insertion: insert a letter in the midst of w

Example. *algoithms* contains a spelling error arising from deletion (the letter r has been deleted); *algoirthms* contains a spelling error arising from transposition (r and i have been exchanged); *algoritgms* contains a spelling error arising from replacement (the letter h has been replaced by g); *algoruithms* contains a spelling error arising from insertion (u has been inserted).

We can now define the edit distance.

Definition. *Given two words w_1, w_2, we say that the edit distance*[1] *d between w_1 and w_2 is equal to k if k is the minimum number of operations above required to turn w_1 into w_2. In such a case, we write $d(w_1, w_2) = k$.*

The edit distance d takes positive integer values and it is symmetric (the latter property will be relevant in the following).

[1] also called Damerau-Levenshtein distance

Example. The edit distance between *algorithms* and *algoithms* is 1. The edit distance between *algorithms* and *agloitms* is 3.

Actually, we will also be interested in a slightly different version of the edit distance, which considers only deletion as admissible operation.

Definition. *Given two words w_1, w_2, we say that the edit distance based on deletions d_1 between w_1 and w_2 is equal to k if k is the minimum number of deletes, defined as above, required to turn w_1 into w_2. In such a case, we write $d_1(w_1, w_2) = k$.*

We still call it edit distance, since the context will make clear which definition we are referring to. So, this edit distance is the minimum number of deletes required to turn a word into another.

2.2. Correcting spelling

Now, given a misspelled word w, we call a candidate correction of w an existing word c, i.e. a word present in the dictionary, which minimizes the edit distance from w.
In general, a candidate correction as defined above exists, but it is not unique: rather, we will have a set of candidate corrections $C = \{c_1, ..., c_k\}$.

Example. Both *on* and *of* are candidate corrections of the misspelling *og*, as they both achieve minimum edit distance (1).

This definition means that, if there exist words having edit distance 1 from the input word w, then - for misspelling correction purposes - we ignore all words with higher edit distance from w: we are considering an error model where known words at distance 1 are infinitely more "probable" than known words at distance 2.
In practice, the algorithms we present consider the possible candidates in order of increasing distance: first we will check if the word was correctly spelled (so has distance 0 from the original); if not, we will look at terms at distance 1; if none is found, we will move on to distance 2 and so on. This is reasonable: we "trust" that the user has made little to no typos while writing. Directly considering the possibility of many errors in each word would make the problem much more difficult and could lead to some nonsense corrections.

Example. Given the misspelling *og*, we know that *of* and *on* are candidate corrections. If we assumed both letters are wrong, we would end up considering other, completely different, possible solutions, such as *at*.

Now we should understand how to choose the proposed correction, given that we have a *set* of candidate corrections. Such indeterminacy can be solved in the following way. We assume we are given a probability distribution $\mathbb{P}$ defined on the set of existing words, which represents the frequency of words in the language. Then, given a set of candidate corrections C, the chosen correction should be

$$c^* = \arg\max_{c \in C} \mathbb{P}(c)$$

i. e. the candidate correction which is most common in the language.

At this point, many details should be discussed.
First of all, how can we recover $\mathbb{P}$? Norvig's approach relies on considering an enormous corpus

of texts, containing, let us say, N different words: for each word w_i in such corpus, we count the number of time it appears in the corpus, n_i, and its probability is estimated as

$$\hat{\mathbb{P}}(w_i) = \frac{n_i}{N}$$

This explains why we use a dictionary with the frequency of each word: instead of creating a function to evaluate it, starting by a big corpus, we just recover it by the dictionary itself. The set of words in the corpus, paired with their frequency, is called frequency dictionary.

An important observation, which will be clearer later, is that a word not appearing in our dictionary would never be proposed as a correction. Therefore, if the user wanted to write that word and made an error, our algorithm will not propose that term as a solution, but another one present in its dictionary (or none). So it is important that the number of words in the dictionary is as near as possible to the real number of unique words in the language. This, and the fact that the context is neglected, are the main limits of these approaches.

2.3. Appendix: further details on the selection mechanism

We have said that, given a set of candidate corrections C, the chosen correction should be

$$c^* = \arg\max_{c \in C} \mathbb{P}(c)$$

Why is this reasonable? We provide a theoretical answer.

We establish that our is aim to propose the correction $c \in C$ that maximizes the probability that c is the intended correction, given that the word w is typed. So we look for

$$\arg\max_{c \in C} \mathbb{P}(c|w)$$

which, by Bayes' theorem, is the same as

$$\arg\max_{c \in C} \frac{\mathbb{P}(w|c)\mathbb{P}(c)}{\mathbb{P}(w)}$$

Since the maximization runs over c, the term $\mathbb{P}(w)$ can be neglected.

Now, the model assumes that all words at a given edit distance k from w are infinitely more probable than words at distance $k' > k$ from w. This implies that $\mathbb{P}(w|c)$ is the same for all candidates, and thus the maximization boils down to finding the candidate with highest probability[2].

[2] In the whole project we actually find the candidate with highest frequency. Of course the only difference is a renormalization coefficient, which does not affect maximization.

3

A first approach: Norvig's algorithm

In this chapter we go through the algorithm proposed by Peter Norvig [5] to tackle spelling correction, based on the model we have presented in the previous chapter. After presenting the algorithm and a pseudocode, we illustrate its execution with a toy example and then provide the full implementation in Python, together with an analysis of its complexity and the employed data structures. Finally, we run the algorithm and comment results.

3.1. Algorithm

Throughout the whole chapter we assume we are given a dictionary which pairs each word with its frequency (estimated as we have explained in the previous chapter).

Norvig's algorithm is a naive way to build a spelling corrector.

Recall that we have defined candidate corrections of w as existing words minimizing the edit distance from w. Intuition suggests that this minimum edit distance should not be very high, since it is unlikely that someone makes that many errors while typing. Notice that in the previous chapter we *did not* discuss about the possible values of the minimum edit distance. The naive, brute force approach would lead to compute directly the edit distance between w and any existing word, and then consider the words minimizing the distance. This is computationally expensive, given the high number of all possible words.

Norvig solves such issue as follows: he simplifies the model by deciding to neglect the case when the minimum edit distance is greater than $k = 2$, so that candidate corrections will be words having distance at most 2 from w.[1]. So, in this case, we assume that the person writing will pay enough attention to make no more than two typos.

Initially, we check if the word was correctly spelled, in which case we return it as it is. Otherwise we compute existing words with edit distance 1 from w in the following way: we generate all strings of characters resulting from a single edit of w (by deleting, transposing, inserting or replacing one character) and we search each string in the dictionary in order to retain only those

[1]This threshold is arbitrary: k may be any positive integer

which are existing words:

- if the resulting set is non-empty, then it is the set C of candidate corrections for w.
- if it is empty, we repeat the same procedure on all strings resulting from two edits, not only one, of w:
  - if the resulting set is non-empty, then it is the set C of candidate corrections for w;
  - if it is empty, no correction is proposed and the algorithm fails. This could happen if the meant word does not exist in the vocabulary we are working with or if there are more than two misspelling errors.

Finally, if C is non-empty, we find the candidate correction with maximum frequency in the dictionary and return it.
Notice that the algorithm searches possible candidates by proceeding gradually, following the increasing edit distance.

We may now write the following pseudoalgorithm, where the input is the word w to be corrected.

We assume `known` to be a function that, given a list of words, returns only those words that appear in the frequency dictionary (i. e. the "known" words).

Algorithm 1 Norvig's Spelling Corrector

Require: frequency dictionary, misspelled word `word`, max edit distance $k = 2$
Ensure: proposed correction c^*

```
if known(word) then
    return word
end if

perform all possible 1-deletes on word
perform all possible 1-transposes on word
perform all possible 1-replacements on word
perform all possible 1-insertions on word
edits1(word) = union of deletes, transposes, replacements, insertions

candidates={}
if known(edits1(word)) is non-empty then
    candidates=known(edits1(word))
else
    edits2={}
    for all word1 in edits1(word) do
        for all word2 in edits1(word1) do
            add word2 to edits2
        end for
    end for

    if known(edits2(word)) is non-empty then
        candidates=known(edits2(word))
    end if
end if


if candidates is non-empty then
    correction = arg max_candidate frequency(candidate)
    return correction
end if

return none
```

Of course, the pseudocode should be understood as a basic version of Norvig's algorithm, as it arises from the conceptual explanation we have given. In the practical implementation some optimizations are applied and the algorithm is structured in a modular way, using multiple functions.

3.2. Toy example

In the following we simulate the execution of the algorithm on a toy example: we will consider the word

thw

which is a misspelling for the, and we compute all the possible edits up to distance 1 that can be retrieved by it, in order to show step by step how this algorithm works. It is not restrictive to consider $k = 1$, since it already gives us a candidate set not empty.

Deletions:

hw tw th

Insertions:

athw	tahw	thaw	thwa	bthw	tbhw	thbw	thwb
cthw	tchw	thcw	thwc	dthw	tdhw	thdw	thwd
ethw	tehw	thew	thwe	fthw	tfhw	thfw	thwf
gthw	tghw	thgw	thwg	hthw	thhw	thhw	thwh
ithw	tihw	thiw	thwi	jthw	tjhw	thjw	thwj
kthw	tkhw	thkw	thwk	lthw	tlhw	thlw	thwl
mthw	tmhw	thmw	thwm	nthw	tnhw	thnw	thwn
othw	tohw	thow	thwo	pthw	tphw	thpw	thwp
qthw	tqhw	thqw	thwq	rthw	trhw	thrw	thwr
sthw	tshw	thsw	thws	tthw	tthw	thtw	thwt
uthw	tuhw	thuw	thwu	vthw	tvhw	thvw	thwv
wthw	twhw	thww	thww	xthw	txhw	thxw	thwx
ythw	tyhw	thyw	thwy	zthw	tzhw	thzw	thwz

Transpositions:

htw hwt twh wth wht

Replacements:

ahw	bhw	chw	dhw	ehw	fhw
ghw	hhw	ihw	jhw	khw	lhw
mhw	nhw	ohw	phw	qhw	rhw
shw	thw	uhw	vhw	whw	xhw
yhw	zhw	taw	tbw	tcw	tdw
tew	tfw	tgw	thw	tiw	tjw
tkw	tlw	tmw	tnw	tow	tpw
tqw	trw	tsw	ttw	tuw	tvw
tww	txw	tyw	tzw	tha	thb
thc	thd	the	thf	thg	thh
thi	thj	thk	thl	thm	thn
tho	thp	thq	thr	ths	tht
thu	thv	thw	thx	thy	thz

Now, assuming we work with the same frequency dictionary used in the data preparation, we ask ourself how many of these words are in the dictionary, i.e., truly have meaning. Through a detailed research, we find the meaningful candidates, here colored in yellow. The non-existing words are colored in red.

hw	tw	th	athw	tahw	thaw	thwa	bthw
tbhw	thbw	thwb	cthw	tchw	thcw	thwc	dthw
tdhw	thdw	thwd	ethw	tehw	thew	thwe	fthw
tfhw	thfw	thwf	gthw	tghw	thgw	thwg	hthw
thhw	thhw	thwh	ithw	tihw	thiw	thwi	jthw
tjhw	thjw	thwj	kthw	tkhw	thkw	thwk	lthw
tlhw	thlw	thwl	mthw	tmhw	thmw	thwm	nthw
tnhw	thnw	thwn	othw	thow	thwo	pthw	tphw
thpw	thwp	qthw	tqhw	thqw	thwq	rthw	trhw
thrw	thwr	sthw	tshw	thsw	thws	tthw	tthw
thtw	thwt	uthw	tuhw	thuw	thwu	vthw	tvhw
thvw	thwv	wthw	twhw	thww	thww	xthw	txhw
thxw	thwx	ythw	tyhw	thyw	thwy	zthw	tzhw
thzw	thwz	htw	hwt	twh	wth	wht	ahw
bhw	chw	dhw	ehw	fhw	ghw	hhw	ihw
jhw	khw	lhw	mhw	nhw	ohw	phw	qhw
rhw	shw	thw	uhw	vhw	whw	xhw	yhw
zhw	taw	tbw	tcw	tdw	tew	tfw	tgw
thw	tiw	tjw	tkw	tlw	tmw	tnw	tow
tpw	tqw	trw	tsw	ttw	tuw	tvw	tww
txw	tyw	tzw	tha	thb	thc	thd	the
thf	thg	thh	thi	thj	thk	thl	thm
thn	tho	thp	thq	thr	ths	tht	thu
thv	thw	thx	thy	thz			

Thus the eight candidate corrections are *thaw, thew, taw, tow, the, tho, thu, thy*. We need to consider their frequencies in the frequency dictionary in order to choose the best correction: searching the dictionary yields

Candidate word	Frequency
thaw	735810
thew	96759
taw	166123
tow	2869301
the	23135851162
tho	2468927
thu	61622542
thy	10017433

The most frequent among them (i.e. the most probable candidate), thus the proposed correction, is *the*.

3.3. Implementation

We now present the Python implementation of this algorithm. First we implement a class to create all the possible edits, up to distance 2, of a single word.

```
class WordEdits:
    def __init__(self, word):
        self.word = word
        self.letters = string.ascii_lowercase

    def delete(self):
    # generate 1-character deletes of self.word
        return {self.word[:i] + self.word[i+1:] for i in range(len(self.word))}

    def transpose(self):
    # generate 1-character transposes of self.word
        return {
            self.word[:i] + self.word[i+1] + self.word[i] + self.word[i+2:]
            for i in range(len(self.word) - 1)
        }

    def replace(self):
    # generate 1-character replacements of self.word
        return {
            self.word[:i] + c + self.word[i+1:]
            for i in range(len(self.word))
            for c in self.letters if self.word[i] != c
        }

    def insert(self):
    # generate 1-character inserts of self.word
        return {
            self.word[:i] + c + self.word[i:]
            for i in range(len(self.word) + 1)
            for c in self.letters
        }

    def edits1(self):
    # return all words that are 1 edit away from the input word
        return (
            self.delete() |
            self.transpose() |
            self.replace() |
            self.insert()
        )

    def edits2(self,list_of_edits=None):
    # generate all words that are 2 edits away from the input word

      # compute 1-edits only if they are not given as input
      if list_of_edits is None:
          one_edit_candidates = WordEdits(self.word).edits1()
      else:
          one_edit_candidates = list_of_edits

      # compute 2-edits
      results = set()
```

```
    for e1 in one_edit_candidates:
        two_edits_candidates=WordEdits(e1).edits1()
        for e2 in two_edits_candidates:
            results.add(e2)

    return results
```

The class `WordEdits` contains the methods `delete`, `transpose`, `replace and insert`, which generate respectively all the deletions, transpositions, replacements and insertions of a single character at a time.
`edits1` returns all the possible edits, at distance 1, of the word passed to the class, while `edits2` computes first those at distance 1, if they are not passed as an input, and then those at distance 2. The possibility of getting 1-edits as an input avoids to call the function `edits1` uselessly, since we know - by the structure of Norvig's algorithm - that we compute 2-edits only if, after computing 1-edits, we do not find existing words among them.

Next we create another class, which implements the body of Norvig's algorithm. We called it generically `SpellingCorrection` because we will want to reuse some of its methods in the `SymSpell` implementation.

```
class SpellingCorrection:
    def __init__(self, vocabulary, word):
        self.vocabulary = vocabulary
        self.word = word
        self.candidateCorrections = {}
        self.proposedCorrection = None

    def known(self, words):
    # select words that appear in the dictionary from list of words (and save also their frequency
     )
        known_words = {}
        for w in words:
          if w in self.vocabulary:
            known_words[w] = self.vocabulary[w]
        return known_words

    def candidates(self):
    # generate candidate spelling corrections for a word
        edits0 = {self.word}

        if self.known(edits0):
            self.candidateCorrections = self.known(edits0)
            return
        else:
            edits1 = WordEdits(self.word).edits1()
            e1 = self.known(edits1)
            if e1:
                self.candidateCorrections = e1
                return
            else:
                edits2 = WordEdits(self.word).edits2(edits1)
                e2 = self.known(edits2)
                if e2:
                    self.candidateCorrections = e2
                    return
```

```
        return

    def get_candidates(self):
        self.candidates()
        return self.candidateCorrections

    def correction(self):
    # return most likely correction for word, chosen among candidates
        self.candidates()
        possible_corrections = self.candidateCorrections

        if not possible_corrections:
            return None

        max_frequency = 0
        for candidate, freq in possible_corrections.items():
            if freq > max_frequency:
                found_word = candidate
                max_frequency = freq

        self.proposedCorrection = found_word

    def get_correction(self):
        self.correction()
        return self.proposedCorrection
```

The method `known` is given a set of words (in our case all the edits at distance 1 or 2) and searches them in the original dictionary, in order to find words that have meaning.
This is used by the method `candidates` to create the set of possible corrections: at first it checks if the word is correctly typed (distance=0), otherwise searches in the edits at distance 1 from it and only if this set is empty moves onto the words at distance 2. Thus the algorithm proceeds gradually, according to an order of increasing distance, as we explained before.
Finally, the method `correction` proposes, as correction, the candidate with the highest frequency. We have already discussed why the algorithm prefers most used words when correcting spelling.
To get the set of candidates and the proposed correction, two getter methods (`get_candidates`, `get_correction`) are implemented.

3.4. Algorithm analysis

3.4.1. Data structures

In this section we discuss the operations we need to perform in Norvig's algorithm and, consequently, the Python data structures that we have chosen to use in our implementation.
We need to perform mostly the following operations:

- **Search**: since we need to check multiple times if words are known (i. e. if they appear in the frequency dictionary), we perform repeated searches through the dictionary. This suggests that the frequency dictionary should be implemented in a structure based on hashing, since hash tables have a $\mathcal{O}(1)$ search complexity. Moreover, we need to store frequencies associated to each word, so choosing a Python dictionary data structure to

implement the frequency dictionary is natural. This choice speeds up the search phase of possible candidates, which is the core of the algorithm.

- **Generate edited strings**: generating 1 and 2-edits of the input word results in performing multiple concatenations of parts of words. Storing the words as strings makes the code pretty readable, and optimized methods are available to perform splits, concatenations etc.
- **Add elements to sets which should not contain duplicates**: we continuously need to build sets of words adding the elements one by one. This happens when we create the set of 1-edits or 2-edits, and also when we create the set of candidates. Moreover, these are sets in the mathematical sense of the term, so that there is no order among their elements and they should not contain duplicates: thus it makes sense to use a data structure based on hashing (since the `add` method has constant complexity, and there are no duplicates among keys), i. e. a set or a dictionary data structure.
  To store 1 and 2-edits we do not need values, thus we choose a Python set data structure. To store candidates, instead, since we need the associated frequencies, we use a dictionary.

3.4.2. Complexity

In this subsection we analyze the computational complexity of Norvig's algorithm.

We consider the following parameters:

$$N = \text{length of the input word } w$$
$$l = \text{number of letters in the alphabet}$$

If the words to be corrected are written in the Latin alphabet, $l = 26$.

Before analyzing the time complexity of Norvig's algorithm, we do a brief detour on space complexity: we will need, indeed, to know how many 1-edits has a word w with length N.
We can notice that there exist:

- N words resulting from 1-deletes;
- $N - 1$ words resulting from 1-transpositions[2];
- $(l - 1)N$ words resulting from 1-replacements, since each character of the N characters in the word may be replaced with any of the other letters in the alphabet;
- $l(N + 1)$ words resulting from 1-insertions, since any of the l letters can be inserted in any position.

The total is $(2l + 1)N + l - 1$ words: they are not necessarily distinct, as different edits may yield the same word, so the set of 1-edits has maximum cardinality $(2l + 1)N + l - 1 = \mathcal{O}(Nl)$.

Let us now turn to time complexity.
We do a worst-case analysis, i. e. we assume that we are in the following situation: w does not appear in the frequency dictionary, thus we generate all 1-edits of w, but none of them appears in the frequency dictionary, so we need to generate all 2-edits.

The first step of the algorithm is generating the set of candidate corrections. In order, we have to:

[2] Recall that we consider only transpositions of two adjacent characters!

- **Search w in the frequency dictionary**: it is known that searching in a hash table has complexity $\mathcal{O}(1)$.
- **Generate 1-edits of w**: for each of the four possible types of edit, for each character i in w, we split w depending on i and concatenate some substrings of w (and some other characters, depending on the type of edit).
  The complexity of string concatenation is $\mathcal{O}(N)$, and this is performed for all N characters in w. So for deletes and transposes the resulting complexity is $\mathcal{O}(N^2)$, for replaces and inserts (since we also run a `for` loop on the l letters in the alphabet) it is $\mathcal{O}(N^2 l)$.
  Overall, the complexity of this step is $\mathcal{O}(N^2 l)$.
- **Look for known words**: for each word in the set of words generated above, we need to check if it appears in the frequency dictionary and, if so, add it to the dictionary to be returned. The cardinality of the set of 1-edits is $\mathcal{O}(Nl)$, as proved above, and the search and adding require $\mathcal{O}(1)$, so the complexity of this part is $\mathcal{O}(Nl)$.
- **Generate 2-edits of w**: assuming the previous search has produced an empty set of known words (since we are in the worst-case scenario), we need to generate 2-edits. Since we have already computed 1-edits, we may simply compute 1-edits of each word in such set[3]. Since the cardinality of the set is $\mathcal{O}(Nl)$, the resulting complexity is $\mathcal{O}(N^3 l^2)$.
- **Look for known words**: again, for each word in the set generated above, we do a search in the dictionary. The maximum cardinality of the set generated above, without considering possible duplicates, is $((2l+1)N + l - 1)^2$, thus the complexity of this part is $\mathcal{O}(N^2 l^2)$.
- **Find the candidate correction with maximum frequency**: assuming that at least one word has been found, so that the set of candidate corrections is non-empty, we need to find the word with maximum frequency. Since the cardinality of the set to be searched is $\mathcal{O}(N^2 l^2)$, and the operation of search requires $\mathcal{O}(1)$, the search of the maximum has complexity $\mathcal{O}(N^2 l^2)$. In fact we cannot avoid scanning all values.

As a result, the general complexity is

$$\mathcal{O}(1) + \mathcal{O}(N^2 l) + \mathcal{O}(Nl) + \mathcal{O}(N^3 l^2) + \mathcal{O}(N^2 l^2) + \mathcal{O}(N^2 l^2) = \mathcal{O}(N^3 l^2)$$

We need to make a few remarks about this result.

The first is that the estimate is quite pessimistic. Indeed, the term driving the complexity is the one due to the generation of 2-edits of w: this is performed only if the set of existing words, among 1-edits, is empty. Most of the times this is not the case, unless the correct word is unknown or tremendous spelling errors have been done[4].

As a second remark, notice that our parameters are the length N of the input term and the number of letters l in the alphabet. It makes perfect sense to consider l in such asymptotic analysis, because - depending on the language - l may be enormous: if we were to do spelling correction in Chinese, we would have to deal with over 70000 Unicode Han characters.
Now, one may argue that it is pointless to conduct the analysis also on N, since length of words

[3] Notice that computing 2-edits in this way actually yields a set of words which is slightly larger than that of words that are 2 edits away from w. Indeed, such a word is always the result of two subsequent edits, but performing two subsequent edits may produce the original word w, e. g. if we perform an insertion and then a deletion of the same character. This - apart from the useless computations - does not raise any issue, since the original word still will not be found in the dictionary

[4] For instance, in our test dataset 2-edits are computed for less than 300 words out of 1000.

does not grow to plus infinity: actually, N is bounded. The actual execution time of the algorithm is not really influenced by the complexity class in N. In fact, as we will show later on, also SymSpell has the same time complexity in N but is significantly faster (as there are constants involved, etc.).

3.5. Simulations: accuracy and conclusions

The testing proceeds as follows: for each misspelled word (i. e., for each key in the dictionary), we run the algorithm and compute a proposed correction: if it coincides with the correct word, then the algorithms has succeeded, otherwise it has failed.
We compute accuracy of the algorithm $\mathcal{A}$ on the dataset as

$$acc(\mathcal{A}) = \frac{s}{n}$$

where s is the number of successes, recovered by a counter which keeps track of the number of cases in which our proposed solution coincides with the expected one given by the dataset. Of course, this setting is much more trivial than what happens in reality, as in real-life spelling corrections we never have the expected correct word.

We now show the results of different tests we made: first with the dataset imported at the beginning, then on the text we mentioned in the first chapter.
We do not expect an exceptionally high accuracy for Norvig's algorithm, since it is likely that some of the correct words in the test dataset do not appear in our dictionary, or the misspellings arise from more than two typos.

Test on dataset

```
    Evaluation Results:
Total misspelled words tested: 1000
Number of correct predictions: 569
Accuracy: 56.90%
```

So the above procedure yields an accuracy

$$acc(\mathcal{N}) = 56.9\%$$

As we expected, the accuracy is not particularly high. We notice that there are 3 main sources of mistakes:

- **Non-existing words**, i. e., the correct word does not appear in our frequency dictionary. For instance, in our test dataset there is the correct word *ebook*, which does not exist in our frequency dictionary.
  This is a serious source of mistakes, as it arises from an inner weakness of the frequency-dictionary that has been chosen. Notice, however, that in real life this problem can be handled in several ways: for instance, depending on the context where spelling correction is needed, generating a frequency dictionary tailored to the specific use-case may be useful. E. g., it is expectable that a "normal" frequency dictionary is not excellent in correcting spellings of bureaucratic university files; in such a case, it is reasonable to use a dictionary specialized in the bureaucratic vocabulary.

- **Words that are more than two edits away**, i. e., the correct word has an edit distance higher than 2 from the misspelling. For instance, in the dataset there is the couple (*aojecm,project*), which are three changes distant.
  In such a case, the correct word does not even appear in the set of candidates. The rate of occurrence of this mistake can be decreased by allowing the algorithm to compute candidates up to an edit distance of 3, as well; however, this increases a lot the computational burden (and as we discussed at the beginning of this chapter this kind of mistake is, in general, not so common to justify it).
- **Words that are two edits away, when there exist words one edit away**, i. e., the correct word is two edits away from the misspelling, but there exist words in the frequency dictionary that are zero or one edit away from the misspelling, so only the latter words are included in the set of candidates, as for *re*, which is seen as an existing word in our dictionary but was meant to be corrected with *sure*. This is a consequence of the priority that has been established in the model: even extremely likely two-edits-away words are obscured by the presence of one-edit-away words. This error model, as Norvig himself suggests, may be improved by gathering a lot of data about misspellings and computing a "cost" associated to the kind of misspelling, which allows to include also two-edits-away words.

However this kind of fixes produces small improvements if compared to the effect of accounting for context, by using methods of deep learning.

To understand better where and why the algorithm fails, and to see how neglecting context harms performance, let us test the algorithm on a text.

Text

The text is the following (the misspellings are coloured in red):

I was definately looking forward to the meeting today, but unfortunatly it looks like we'll have to pospone it once again. Something unexpected came up this morning, and I won't be able to make it at the originally schedualed time. My calender is alredy quite full this week, and trying to reschedule might be a bit of a challange. I'll realy need to take a closer look at my availibility over the next few days and see what might work. Thursday afternoon could be an option, though I'm still waiting on confirmation for another commitment at that time. Friday is looking very tight too, and I'd rather avoid pushing it to next week if we can help it. I hope this doesn't effect your workflow or plans too much. I know how important this discussion is, and I truely appreciate your flexibility and understanding. Please let me know if there are any windows that work better for you, and I'll try to allign accordingly. The sooner we can nail down a new time, the better. In the meantime, I'd ask that you continue preparing the necessary documentation. We still need to recieve everything by Friday at the latest. There's very little room for delay at this stage, and having the materials ready in advanse will really help things move smoothly when we do meet. Also, if you have any questiond or if anything remains unclear from our previous correspondance, feel free to reach out. I'll try to respond promptly, although I may be slow to reply during certain hours due to other meetings and tasks that have pilled up. Again, thanks so much for your patience and support. I'm confident we'll get everything sorted out shortly, even if there are a few hiccups along the way. Talk to you soon.

We get the following output:

```
    Misspelled: calender | Expected: calendar | Predicted: calender
Misspelled: realy | Expected: really | Predicted: real
Misspelled: effect | Expected: affect | Predicted: effect
Misspelled: pilled | Expected: piled | Predicted: filled

 Evaluation Results:
Number of misspelled words: 18
Number of correct predictions: 14
Accuracy: 77.78%
```

These results are consistent with what we said so far. Notice, for example, that the word *calender* is not recognized as wrong because it appears in our dictionary, therefore is not corrected in *calendar*. Analogously, *effect* is a correct word and so is kept as it it written.
However, the context makes it obvious that they are both wrong: *calender* is a technical term, referring to a type of machine which obviously has nothing to do with a discussion about scheduling a meeting; *effect* is a noun, but it should clearly be corrected to a verb, by the structure of the sentence.

4

A smarter approach: SymSpell

All the remarks we have done about Norvig's algorithm point out that it is not even remotely the most efficient way to tackle the problem of spelling correction: performing all possible edits on the input word, using all the possible letters in the alphabet, creates a gigantic set of possible corrections[1], which will need to be resized by keeping only existing words.

SymSpell[2] is an algorithm relying on the same theoretical principles as Norvig's algorithm, but it is considerably faster, as it exploits properties of the edit distance to move some of the computation to a pre-calculation step, which is done once and for all.
In the following we explain in detail the intuition behind this improvement of Norvig's algorithm, provide the Python implementation and test it on our dataset.

4.1. Algorithm

The theoretical setting is exactly the same we have used in the previous chapter:

- we still assume we are given a dictionary which pairs each word with its frequency;
- we aim to build a spelling corrector which, given a possibly misspelled word w, proposes a correction;
- we still look for candidate corrections C, words minimizing the edit distance from w, up to 2 (but in principle can be any positive integer).
- we choose as proposed correction the word in the candidate corrections with the highest frequency.

Recall that Norvig's method to find candidates is performing all possible edits (deletions,..., insertions) up to order k on w and checking if they are existing words (by searching through the dictionary).
SymSpell, instead, relies on an intuition which allows to speed up the process, removing the computation of insertions, transpositions and replacements and restricting to deletions, therefore by using the edit distance d_1 we introduced in chapter 2.

[1] which is also language dependent, so that running the algorithm may even be unfeasible, depending on the language
[2] SymSpell stands for *Symmetric spelling correction*

We first explain the algorithm and then provide a theoretical justification for it.
The steps of the algorithm are the following:

- **Precalculation** This step should be executed only once. For each term in the frequency dictionary, we compute all its deletes up to order k and add them to a dictionary as keys; for each of them, we store a set containing all the original terms (and their frequencies) from which the word has been obtained by means of deletions.
  Notice that any existing word will be a key itself, corresponding to a deletion of order 0 from itself. This allows to recognize whether a word was correctly spelled.
  This step produces a new frequency dictionary which we will call edited dictionary or deletions dictionary. See the toy example (4.2) for an example of edited dictionary.
- **Candidates generation** First, we check if the misspelled word w is in the *original* frequency dictionary, i. e. if it was spelled correctly. In such a case case, w itself is proposed as a correction.
  Otherwise, we check if w is a key of the edited dictionary: if this is the case, we retrieve the associated original terms and add them to the set of candidates.
  Then, we compute all deletes D of w up to order $k = 2$ and search each of them in the edited dictionary. If a delete d is found, we look for the original words associated, which will be added to the set of candidates.
  This step produces a set of candidate corrections[3].
- **Correction choice** We inspect candidates to find words at distance 1 from w, computing directly the edit distance. If they exist, the one with maximum frequency is proposed as correction; otherwise we look for words at distance 2 and do the same, exactly as in Norvig's algorithm.

As we discuss in the theoretical justification, the proposed correction is the same as Norvig's algorithm, but it is produced with a considerably faster procedure.
Intuitively, SymSpell's idea is that we do not need to generate all possible replacements, insertions, transposes and deletions of w and then look for them in the dictionary: we can find words at distance k from w by computing *only* k*-deletes* of w and searching them in the edited dictionary. Indeed, the precalculation step has added to the dictionary all possible k-deletions of existing words, preserving the link with the original word. So, if one of the deletions of w matches one of the entries of the dictionary, we can trace back to a correctly spelled word.

Example. Given *thw*, misspelling of *the*, we can trace back to *the*: if we generate all 1-deletes of *thw* we get also *th*, which will be found in the edited dictionary associated to the original word *the*, since it arises from a 1-deletion from it.

The pseudoalgorithm is the following:

[3]Here we are committing a slight abuse of language. The set of candidates we mention is *not* the set of candidate corrections in the sense of the definition we gave in chapter 2, i. e. words minimizing the edit distance from w. Indeed, here the set of candidates correction includes words at different edit distance from w. Still, it is a set of candidates in that we inspect it to look for the proposed correction, and so we stick to such terminology.

Algorithm 2 Symspell Spelling Corrector

Require: frequency dictionary, misspelled word `word`, max edit distance $k = 2$
Ensure: proposed correction c^*

```
function dictionary_edits(dictionary)
    for all word in dictionary do
        compute all deletions of word up to order 2
        add them to the dictionary, storing the original words (with their frequency) as values
    end for
    return edited dictionary
end function

function SymSpell(word,dictionary,edited dictionary)
    if word is in original frequency dictionary then
        return word
    end if

    candidates={}
    if word is in edited dictionary then
        add to candidates the set associated to word
    end if
    generate 1-deletes of word
    for all edit in 1-deletes of word do
        if edit is in edited dictionary then
            add to candidates the set associated to edit
        end if
    end for
    generate 2-deletes of word
    for all edit in 2-deletes of word do
        if edit is in edited dictionary then
            add to candidates the set associated to edit
        end if
    end for

    for all word in candidates do
        if word has distance 1 from input word then
            add word to 1-edits
        end if
    end for
    if the set of 1-edits is non-empty then
        return 1-edit with maximum frequency
    end if

    for all word in candidates do
        if word has distance 2 from input word then
            add word to 2-edits
        end if
    end for
    if the set of 2-edits is non-empty then
        return 2-edit with maximum frequency
    end if

    return none
end function
```

Notice that we assume we are given a function that can compute the edit distance, or at least check if the edit distance is 1 or 2.

Theoretical justification

To grasp why it makes sense to restrict to deletions, we first discuss the case of maximum edit distance equal to 1 (i. e. we store 1-deletes in the edited dictionary, we generate 1-deletes of w).
In such a case, the set of candidate corrections generated by SymSpell coincides with Norvig's. Formally, the reason is the following fact:

$d(w', w) = 1$ if and only if there exists a word w_0 such that both w' and w can be transformed into w_0 by performing *at most one deletion*.

Indeed, the following situations occur:

- if w can be obtained from w' by an insertion, then w can be transformed into $w_0 := w'$ by deleting the letter that has been inserted.
- if w can be obtained from w' by a deletion, then w' can be transformed into $w_0 := w$ by deleting the letter.
- if w can be obtained from w' by a transposition, then we set w_0 to be w' deprived of one of the characters that have been transposed; then, w' and w can be transformed into w_0 by deleting such character.
- if w can be obtained from w' by a replacement, then we set w_0 to be w' deprived of the replaced character; then, w' and w can be transformed into w_0 by deleting the same character.

Notice that these arguments work thanks to the symmetry of the edit distance and the fact that deletion is the inverse of insertion.

In the case with maximum edit distance equal to 2 (our case), things are not that easy: the sets of candidates of SymSpell and Norvig do not coincide; the former is bigger.
Indeed, if $d(w', w) = 2$ then there exists a word w_0 such that both w and w' can be transformed into w_0 by performing at most two deletions (this can be proved with similar steps as above). But the reverse does not hold: SymSpell could find candidates that have an edit distance of more than 2 from our misspelled term. Let us show it with an example:

Example. Suppose *comport* is a word in our dictionary. A 2-edits term, in the d_1 sense, in the edited dictionary could be *coprt*.
If the misspelled term was *copyrgt*, generating 2-deletions would also yield *coprt*. Therefore, we would have *comport* as a candidate correction for *copyrgt*, with an edit distance d greater than 2.

So, as an intermediate step, some candidates need to be filtered out (so that the proposed correction is not at more than two edits of distance). SymSpell does this looking for candidates at distance 1 (the edit distance is evaluated directly, unlike in Norvig), and - if there are none - for those at distance 2.

So, at the end of the day, the proposed correction will be the same as Norvig's.

4.2. Toy example

We simulate the execution of the algorithm with input word *thw*, for which the correct spelling is known to be *the* (so the edit distance is equal to 1).

To make the example basic and easily understandable, we do not assume we have a real frequency dictionary (as simulating the generation of all possible deletions for all words in the dictionary would take too long) and we rely on a toy dictionary, containing only four words:

frequency dictionary={”cat”:5, ”bat”:8, ”the”:10, ”tho”:9}

We also stick to maximum edit distance equal to 1, ignoring 2-edits.
So, we first need to perform the pre-calculation, i. e. create a dictionary where the keys are all the possible terms obtained by deleting at most one character from the words in our dictionary, and we associate to them a set of tuples, given by the words from which we obtain that term and their frequency.
From *cat* we obtain

cat *at* *ct* *ca*

From *bat*,

bat *at* *bt* *ba*

From *the*,

the *he* *te* *th*

From *tho*,

tho *th* *to* *ho*

Thus the dictionary of deletions (edited dictionary) is

”cat” : {(”cat”:5)}
”at” : {(”cat”:5),(”bat”:8)}
”ct” : {(”cat”:5)}
”ca” : {(”cat”:5)}
”bat” : {(”bat”:8)}
”bt” : {(”bat”:8)}
”ba” : {(”bat”:8)}
”the” : {(”the”:10)}
”he” : {(”the”:10)}
”te” : {(”the”:10)}
”th” : {(”the”:10), (”tho”:9)}
”tho” : {(”tho”:9)}
”to” : {(”tho”:9)}
”ho” : {(”tho”:9)}

We now execute the algorithm to correct *thw*. It does not belong to the original dictionary, so we generate the set of terms obtained by removing a letter:

{”th”, ”tw”, ”hw”}

The only term that appears in our dictionary of deletions is *th*, associated to {("the":10), ("tho":9)}. This is the set of candidates. It is straightforward to compute that *"thw"* has edit distance 1 both from *"the"* and *"tho"*. So we compare the frequencies and choose the one with the highest value. The output is *the*.

4.3. Implementation

The first thing to do is the pre-calculation step: we define a class which will create the dictionary of strings obtained by deleting 0, 1 or 2 characters from terms in the frequency dictionary. Such strings will be the keys, to which we associate a set containing the tuples (*word*, *frequency*) from which we got the edited string.

```
class VocabularyEdits:
    def __init__(self, vocabulary, max_dist=2):
        if max_dist < 0 or max_dist > 2:
            raise ValueError("Maximum distance should be 0, 1 or 2")
        self.max_dist = max_dist
        self.vocabulary = vocabulary
        self.deletions_dict = {}

    def deletion(self, word, dist, deletions): # notice that deletions is a set in which we will
     store all the edited words
    # generate all deletions of word up to distance 2
      queue = deque()
      queue.append((word, dist))

      while queue:
        current_word, current_dist = queue.popleft()

        # add the current word (wheter it is the original or already an edit one)
        if current_word not in deletions:
            deletions.add(current_word)

        if current_dist < self.max_dist:
            for i in range(len(current_word)):
                edit_word = current_word[:i] + current_word[i + 1:]
                if edit_word not in deletions:
                    queue.append((edit_word, current_dist + 1))

    def create_deletions_dict(self):
    # create edited dictionary adding all deletions

        for word,freq in self.vocabulary.items():
            # add each correct word
            if word not in self.deletions_dict:
                self.deletions_dict[word] = set()
                self.deletions_dict[word].add((word, freq))

            # generate deletions
            deletions = set()
            self.deletion(word, 0, deletions)

            # add each deletion as key, with original value
            for edit_word in deletions:
                if edit_word not in self.deletions_dict:
```

```
                    self.deletions_dict[edit_word] = set()
                self.deletions_dict[edit_word].add((word,freq))

    def get_deletions_dict(self):
        return self.deletions_dict
```

The method `deletion` generates all the possible deletions of a term, from distance 0 up to distance 2. It uses a queue to make a BFS of the edited words; in this way we are sure to create all deletions at the same distance, before going deeper. Is important to consider also the original word itself, to always cover the case where the term is correctly spelled.
The method `create_deletions_dict` stores the original words with their frequencies in a dictionary `deletions_dict` (again, to cover the case of a correctly spelled term), then calls `deletion` to consider all the deletions of all terms and store them as keys, associating to them the words from which we got it (and their frequencies). So the output `deletions_dict` is a dictionary where the value associated to the key is a set.
Finally, we report the code for the body of SymSpell.

```
class SymSpell(SpellingCorrection):
    # constructor of the inherited class + additional attribute
    def __init__(self,vocabulary,word,vocabulary_edits):
        super().__init__(vocabulary,word)
        self.deletions_dict = vocabulary_edits.get_deletions_dict()

    def generate_deletions_distance_2(self):
        word = self.word
        deletions = set()
        for i, j in combinations(range(len(word)), 2):
            edited = word[:i] + word[i+1:j] + word[j+1:]
            deletions.add(edited)
        return deletions

    def candidates(self):
      # generate candidates by inspecting the deletions dictionary
      word = self.word
      candidates = {}

      # search word
      if word in self.deletions_dict:
        candidates.update(self.deletions_dict[word])

      # search 1-deletes
      for deletion in WordEdits(word).delete():
        if deletion in self.deletions_dict:
            candidates.update(self.deletions_dict[deletion])

      # search 2-deletes
      for deletion in self.generate_deletions_distance_2():
        if deletion in self.deletions_dict:
            candidates.update(self.deletions_dict[deletion])

      self.candidateCorrections = candidates


    def correction(self):
      # generate proposed correction
```

```
    word = self.word

    if word in self.vocabulary:
      self.proposedCorrection=word
      return

    self.candidates()
    if not self.candidateCorrections:
      self.proposedCorrection=None
      return

    # look for one edits
    one_edits = {
            c: freq
            for c, freq in self.candidateCorrections.items()
            if textdistance.damerau_levenshtein(word, c) == 1
        }
    if one_edits:
      max_frequency = 0
      for candidate, freq in one_edits.items():
          if freq > max_frequency:
              found_word = candidate
              max_frequency = freq

      self.proposedCorrection = found_word
      return

    # look for two edits
    two_edits = {
          c: freq
          for c, freq in self.candidateCorrections.items()
          if textdistance.damerau_levenshtein(word, c) == 2
      }
    if two_edits:
      max_frequency = 0
      for candidate, freq in two_edits.items():
          if freq > max_frequency:
              found_word = candidate
              max_frequency = freq

      self.proposedCorrection = found_word
      return

    self.proposedCorrection=None
```

The class `SymSpell` is defined as an inherited class from`SpellingCorrection` and relies on the following attributes: the frequency dictionary, the misspelled term and the edited dictionary, generated by `VocabularyEdits`.
We have two methods to generate deletes of the input word up to order 2: `WordEdits(word).delete()` for 1-edits, `generate_deletions_distance_2` for 2-edits.
The method `candidates` searches the misspelled word as it is written among the keys of the dictionary of deletions: if it is found, all the associated words associated are stored in a set of candidates. The same is repeated for all the edits at distance 1 and 2 from the misspelled word.

The method `correction` first checks if the misspelled word is in the vocabulary (therefore is correctly spelled, in which case is returned as it is), otherwise searches the most frequent word among the candidates following an increasing distance order, by computing directly the edit distance among the misspelled word and the set of candidates with a built-in Python function. If the set of candidates is empty, no correction is proposed.
The correction can be recovered with the method `get_correction`, inherited by *SpellingCorrection*.

4.4. Algorithm analysis

4.4.1. Data structures

There are no major changes with respect to the data structures we used in the implementation of Norvig's algorithm: we still use mainly dictionary and set data structures, for the same purposes we highlighted in the previous chapter.
Notice, however, that in this case we have an edited frequency dictionary, so that for each key (a word) the associated value is no more a single frequency, but rather the set of tuples (word,frequency) from which the key can be obtained by means of deletions up to order 2. The process of search is equally efficient.
Moreover, it is worth noticing that in `VocabularyEdits` the function `deletion`, used to generate all deletions up to order 2 of a given word, relies on a queue data structure, whose methods `append` and `pop` are very fast, and allows to implement the generation of deletes in a Breadth-First-Search fashion (computes first all the deletions at the same distance and then moves deeper).

4.4.2. Complexity

In this subsection we analyze the computational complexity of SymSpell.

We consider the following parameters:

$$N = \text{length of the input word } w$$
$$W = \text{number of words in the frequency dictionary}$$

Space complexity

Unlike in Norvig's algorithm, the precalculation step in SymSpell leads to storing a large amount of data, resulting in a non-negligible space complexity.

For each word in the frequency dictionary, we generate its deletes up to order 2: if the word has length k, we obtain (in the worst case) $k + \binom{k}{2}$ words (generating 1-deletes is equivalent to choosing one letter in w to be deleted, generating 2-deletes is equivalent to choosing two letters in w). Calling $\bar{k}$ the supremum of length of words in the dictionary, we need to add

$$W(\bar{k} + \binom{\bar{k}}{2})$$

words to the dictionary.

This implies that the total number of keys in the edited dictionary is

$$\mathcal{O}(W\bar{k}^2)$$

which is quite high: for a dictionary of 80000 words, assuming $\bar{k} = 25$, we need to add 50 millions of words.

Of course, these results are quite pessimistic, as the number of unique words we get performing deletions is often lower than $\bar{k} + \binom{\bar{k}}{2}$.

Time complexity

First, we analyze the time complexity of the pre-calculation step (which is executed only once). For each of the W words in the frequency dictionary, we do

- perform $\mathcal{O}(1)$ operations such as search and adding to a dictionary or a set
- **Generate deletions of w up to order 2**: the `while` loop in the `deletion` is executed 3 times (for `current_dist`$= 0, 1, 2$). In each iteration, $\mathcal{O}(1)$ operations are performed, and then the `for` loop is executed $\mathcal{O}(\bar{k})$ times (since `current_word` is w or w with at most two deleted characters): the complexity of the string concatenation performed in the loop is $\mathcal{O}(\bar{k})$, hence the resulting complexity of this step is $\mathcal{O}(\bar{k}^2)$.
- **Add deletions to the dictionary**: for each of the $\mathcal{O}(\bar{k}^2)$ deletions, $\mathcal{O}(1)$ operations are performed.

As a result, the time complexity of the pre-calculation step is

$$\mathcal{O}(W\bar{k}^2)$$

We now turn to the time complexity of the body of the algorithm. Recall that, again, we do a worst-case analysis: we assume the input word w is not in the original frequency dictionary and that we find no words having distance 1 to w.

In order, we have to

- **Generate candidates**: we need to generate 1 and 2-deletes of w. We have already seen that this has complexity $\mathcal{O}(N^2)$, since we generate a set with cardinality $N + \binom{N}{2}$. This is the complexity of this step.

  To compute the total complexity we will need to know the cardinality of the set of candidates. Computing this is not as straightforward as in Norvig's case: we know how many 1 and 2-deletes w has, but if we look for them in the dictionary, how long will be the list of the original words (i. e. the value) they are key to and that therefore need to be added to the set of candidates? That is: how many different words, at most, will produce the same string by means of a 1 or 2-deletion? This value, which we call v, is actually a property intrinsic to the language and thus to the dictionary; we restrict to considering the value of such parameter for our dictionary and analyze our dictionary to find its maximum, i. e. the maximum number of words that can be associated to an edited string. It turns out that in our case $v = 357$.

  Since the set of 1 and 2-deletes has cardinality $\mathcal{O}(N^2)$, the set of candidates has cardinality $\mathcal{O}(vN^2)$.

- **Look for 1-edits**: for each candidate c we check if $d(c, w) = 1$. It is known that it is possible to implement the evaluation of the edit distance with an algorithm having a complexity $\mathcal{O}(N)$ (see, for instance, [6]).

  Therefore generating `one_edits` has complexity $\mathcal{O}(vN^3)$.
- **Look for 2-edits and find the maximum-frequency term**: since we assume we do not find candidates that are one edit away, we need to to look for 2-edits among candidates. This requires repeating the same procedure as above, resulting again in a complexity $\mathcal{O}(vN^3)$.

As a result, the general complexity is[4]

$$\mathcal{O}(N^2) + \mathcal{O}(vN^3) + \mathcal{O}(vN^3) = \mathcal{O}(vN^3)$$

We need to make a few remarks about this result.

First of all, the complexity does not depend on the number of letters l in the alphabet: SymSpell is *language independent*. This is an extremely relevant result, and one of the main reasons why performing deletes only (and not replaces and inserts) is so convenient: as we have proved in the previous chapter, complexity grows quadratically in the number of letters in the alphabet, so that more complex alphabets than the Latin one make Norvig's algorithm a bad choice.

Second, notice that - unlike Norvig's algorithm - SymSpell *always* generates 2-deletions (unless the input word is in the original vocabulary). This means that in general there are not the "lucky situations" when the quadratic term vanishes and the algorithm is faster (which is a concrete possibility in Norvig's algorithm, in the case when 1-edits which are existing words are found).

Third, we will see that SymSpell is still significantly faster than Norvig's algorithm: this may seem strange, as both algorithms belong to the same complexity class in N. The issue is that, as we have already remarked, N is not really a parameter growing indefinitely: in this situation, N is in fact bounded, and thus the constants make a difference. This result about time complexity is, in fact, of little relevance to the actual time the algorithm takes to run.

Indeed, evaluating the running time of each algorithm on the same dataset, with the same vocabulary, yields these results:

```
Norvig algorithm took: 0 minutes and 47.83 seconds
VocabularyEdits took: 0 minutes and 12.33 seconds
SymSpell took: 0 minutes and 0.56 seconds
```

Notice that this result could vary, to some extent, based on the hardware.
SymSpell is clearly faster than Norvig, even if we also need to run once `VocabularyEdits`.
As we said, Norvig's algorithm generates a very high number of edits from each single words, because it uses transpositions, replacements, insertions and deletions, while SymSpell focuses only on the latter ones. For instance, that given a word of length 9, with an alphabet of 26 letters, the number of edits at distance 2 with Norvig's method is 502; SymSpell would generate just 36 terms.
One of the strengths of SymSpell is its speed, but is pretty expensive in terms of space. Depending on the main aim, other algorithms could be better.

[4] This time we avoided mentioning all the constant costs due to adding, searching etc.

4.5. Simulations: accuracy and conclusions

We now test these two classes.

We first generate, once and for all, the dictionary with the deletions.

```
voc_edit=VocabularyEdits(vocabulary, max_dist=2)
voc_edit.create_deletions_dict()
```

Test on dataset

Unsurprisingly, testing SymSpell on the test dataset yields the same results as Norvig's algorithm.

```
    --- Evaluation Results ---
Total misspelled words tested: 1000
Number of correct predictions: 569
Accuracy: 56.90%
```

We already knew, indeed, that they propose the same corrections.

Text

On the same note, turning to the texts yields

```
    Misspelled: calender | Expected: calendar | Predicted: calender
Misspelled: realy | Expected: really | Predicted: real
Misspelled: effect | Expected: affect | Predicted: effect
Misspelled: pilled | Expected: piled | Predicted: filled


 Evaluation Results:
Number of misspelled words: 18
Number of correct predictions: 14
Accuracy: 77.78%
```

Thus te same observations made for Norvig are valid here.

5

Conclusion

We have presented two different algorithms used to solve the problem of spelling correction: both look for existing words minimizing the edit distance from the input misspelled word. Norvig's solution achieves this generating all possible 1-edits of the input word, selecting known words among them and moving to 2-edits only if necessary (so checking gradually, depending on the distance). From the resulting set of candidate corrections, the proposed correction is the one with maximum frequency.
SymSpell, after a pre-calculation step, generates only deletions of a term (up to order 2) and looks for connections with strings in the edited dictionary, to then suggest (again, based on an increasing edit distance hierarchy) the word which is most used as correction.

Norvig's algorithm is easier to understand and implement; however, it is not so efficient in that it generates a very large set of edits, and it is language dependent. It represents, all the same, a good starting point. SymSpell, in fact, builds on Norvig's ideas to reduce computations and remove language dependency. This is done at the cost of a bigger portion of memory to be devoted to storing the edited dictionary, and a less transparent functioning of the algorithm.

Both the algorithms have a moderate, far from excellent accuracy, as one could expect from the overall reasonably simple structure of the algorithms, if compared to the complexity of processing human language.
We should anyway remember that some issues in the algorithms can be fixed with little effort: for instance, since they cannot correct misspellings to words that do not appear in the frequency dictionary, choosing it suitably is essential to increase the accuracy. Increasing the maximum edit distance allowed from 2 to 3, as well, allows to correct more mistakes. Deeper issues, such as mistakes in correction due to the neglect of the context, cannot be solved by simply tuning the models in a different way: a dramatically different, data-based approach is required, which is today implemented in deep learning models.

Bibliography

[1] W. Burkhard and R. Keller. Some approaches to best-match file searching. 1973.

[2] Wolf Garbe. 1000x faster spelling correction algorithm.

[3] Wolf Garbe. Symspell.

[4] A. Cumhur Kinaci. Spelling correction using recurrent neural networks and character level n-gram.

[5] Peter Norvig. How to write a spelling corrector. 2016.

[6] Esko Ukkonen. Algorithms for approximate string matching. 1985.