**How to Write a Spelling Corrector**

*Feb 2007 to August 2016*

One week in 2007, two friends (Dean and Bill) independently told me they were amazed at Google's spelling correction. Type in a search like [[speling]](http://www.google.com/search?q=speling) and Google instantly comes back with **Showing results for:** [***spelling***](http://www.google.com/search?q=spelling). I thought Dean and Bill, being highly accomplished engineers and mathematicians, would have good intuitions about how this process works. But they didn't, and come to think of it, why should they know about something so far outisde their specialty?

I figured they, and others, could benefit from an explanation. The full details of an industrial-strength spell corrector are quite complex (you can read a little about it [here](http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/pubs/archive/36180.pdf) or [here](http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=52A3B869596656C9DA285DCE83A0339F?doi=10.1.1.146.4390&rep=rep1&type=pdf)). But I figured that in the course of a transcontinental plane ride I could write and explain a toy spelling corrector that achieves 80 or 90% accuracy at a processing speed of at least 10 words per second in about half a page of code.

And here it is (or see [spell.py](https://norvig.com/spell.py)):

|  |
| --- |
| import re from collections import Counter  def words(text): return re.findall(r'\w+', text.lower())  WORDS = Counter(words(open('big.txt').read()))  def P(word, N=sum(WORDS.values())):      "Probability of `word`."     return WORDS[word] / N  def correction(word):      "Most probable spelling correction for word."     return max(candidates(word), key=P)  def candidates(word):      "Generate possible spelling corrections for word."     return (known([word]) or known(edits1(word)) or known(edits2(word)) or [word])  def known(words):      "The subset of `words` that appear in the dictionary of WORDS."     return set(w for w in words if w in WORDS)  def edits1(word):     "All edits that are one edit away from `word`."     letters    = 'abcdefghijklmnopqrstuvwxyz'     splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]     deletes    = [L + R[1:]               for L, R in splits if R]     transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]     replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]     inserts    = [L + c + R               for L, R in splits for c in letters]     return set(deletes + transposes + replaces + inserts)  def edits2(word):      "All edits that are two edits away from `word`."     return (e2 for e1 in edits1(word) for e2 in edits1(e1)) |

The function correction(word) returns a likely spelling correction:

|  |
| --- |
| >>> correction('speling') 'spelling'  >>> correction('korrectud') 'corrected' |

**How It Works: Some Probability Theory**

The call correction(w) tries to choose the most likely spelling correction for w. There is no way to know for sure (for example, should "lates" be corrected to "late" or "latest" or "lattes" or ...?), which suggests we use probabilities. We are trying to find the correction *c*, out of all possible candidate corrections, that maximizes the probability that *c* is the intended correction, given the original word *w*:

argmax*c ∈ candidates* P(*c*|*w*)

By [Bayes' Theorem](http://en.wikipedia.org/wiki/Bayes'_theorem) this is equivalent to:

argmax*c ∈ candidates* P(*c*) P(*w*|*c*) / P(*w*)

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

argmax*c ∈ candidates* P(*c*) P(*w*|*c*)

The four parts of this expression are:

1. **Selection Mechanism**: argmax   
   We choose the candidate with the highest combined probability.
2. **Candidate Model**: *c ∈ candidates*  
   This tells us which candidate corrections, *c*, to consider.
3. **Language Model**: P(*c*)   
   The probability that *c* appears as a word of English text. For example, occurrences of "the" make up about 7% of English text, so we should have P(*the*) = 0.07.
4. **Error Model**: P(*w*|*c*)  
   The probability that *w* would be typed in a text when the author meant *c*. For example, P(*teh*|*the*) is relatively high, but P(*theeexyz*|*the*) would be very low.

One obvious question is: why take a simple expression like P(*c*|*w*) and replace it with a more complex expression involving two models rather than one? The answer is that P(*c*|*w*) is *already* conflating two factors, and it is easier to separate the two out and deal with them explicitly. Consider the misspelled word *w*="thew" and the two candidate corrections *c*="the" and *c*="thaw". Which has a higher P(*c*|*w*)? Well, "thaw" seems good because the only change is "a" to "e", which is a small change. On the other hand, "the" seems good because "the" is a very common word, and while adding a "w" seems like a larger, less probable change, perhaps the typist's finger slipped off the "e". The point is that to estimate P(*c*|*w*) we have to consider both the probability of *c* and the probability of the change from *c* to *w* anyway, so it is cleaner to formally separate the two factors.

**How It Works: Some Python**

The four parts of the program are:

1. **Selection Mechanism**: In Python, max with a key argument does 'argmax'.
2. **Candidate Model**: First a new concept: a **simple edit** to a word is a deletion (remove one letter), a transposition (swap two adjacent letters), a replacement (change one letter to another) or an insertion (add a letter). The function edits1 returns a set of all the edited strings (whether words or not) that can be made with one simple edit:

|  |
| --- |
| def edits1(word):     "All edits that are one edit away from `word`."     letters    = 'abcdefghijklmnopqrstuvwxyz'     splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]     deletes    = [L + R[1:]               for L, R in splits if R]     transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]     replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]     inserts    = [L + c + R               for L, R in splits for c in letters]     return set(deletes + transposes + replaces + inserts) |

This can be a big set. For a word of length *n*, there will be *n* deletions, *n*-1 transpositions, 26*n* alterations, and 26(*n*+1) insertions, for a total of 54*n*+25 (of which a few are typically duplicates). For example,

|  |
| --- |
| >>> len(edits1('somthing')) 442 |

However, if we restrict ourselves to words that are *known*—that is, in the dictionary— then the set is much smaller:

|  |
| --- |
| def known(words): return set(w for w in words if w in WORDS)  >>> known(edits1('somthing')) {'something', 'soothing'} |

We'll also consider corrections that require *two* simple edits. This generates a much bigger set of possibilities, but usually only a few of them are known words:

|  |
| --- |
| def edits2(word): return (e2 for e1 in edits1(word) for e2 in edits1(e1))  >>> len(set(edits2('something')) 90902  >>> known(edits2('something')) {'seething', 'smoothing', 'something', 'soothing'}  >>> known(edits2('somthing')) {'loathing', 'nothing', 'scathing', 'seething', 'smoothing', 'something', 'soothing', 'sorting'} |

We say that the results of edits2(w) have an **edit distance** of 2 from w.

1. **Language Model**: We can estimate the probability of a word, P(word), by counting the number of times each word appears in a text file of about a million words, [big.txt](https://norvig.com/big.txt). It is a concatenation of public domain book excerpts from [Project Gutenberg](http://www.gutenberg.org/wiki/Main_Page) and lists of most frequent words from [Wiktionary](http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists) and the [British National Corpus](http://www.kilgarriff.co.uk/bnc-readme.html). The function words breaks text into words, then the variable WORDS holds a Counter of how often each word appears, and P estimates the probability of each word, based on this Counter:

|  |
| --- |
| def words(text): return re.findall(r'\w+', text.lower())  WORDS = Counter(words(open('big.txt').read()))  def P(word, N=sum(WORDS.values())): return WORDS[word] / N |

We can see that there are 32,192 distinct words, which together appear 1,115,504 times, with 'the' being the most common word, appearing 79,808 times (or a probability of about 7%) and other words being less probable:

|  |
| --- |
| >>> len(WORDS) 32192  >>> sum(WORDS.values()) 1115504  >>> WORDS.most\_common(10) [('the', 79808),  ('of', 40024),  ('and', 38311),  ('to', 28765),  ('in', 22020),  ('a', 21124),  ('that', 12512),  ('he', 12401),  ('was', 11410),  ('it', 10681),  ('his', 10034),  ('is', 9773),  ('with', 9739),  ('as', 8064),  ('i', 7679),  ('had', 7383),  ('for', 6938),  ('at', 6789),  ('by', 6735),  ('on', 6639)]  >>> max(WORDS, key=P) 'the'  >>> P('the') 0.07154434228832886  >>> P('outrivaled') 8.9645577245801e-07  >>> P('unmentioned') 0.0 |

1. **Error Model**: When I started to write this program, sitting on a plane in 2007, I had no data on spelling errors, and no internet connection (I know that may be hard to imagine today). Without data I couldn't build a good spelling error model, so I took a shortcut: I defined a trivial, flawed error model that says all known words of edit distance 1 are infinitely more probable than known words of edit distance 2, and infinitely less probable than a known word of edit distance 0. So we can make candidates(word) produce the first non-empty list of candidates in order of priority:
   1. The original word, if it is known; otherwise
   2. The list of known words at edit distance one away, if there are any; otherwise
   3. The list of known words at edit distance two away, if there are any; otherwise
   4. The original word, even though it is not known.

Then we don't need to multiply by a P(*w*|*c*) factor, because every candidate at the chosen priority will have the same probability (according to our flawed model). That gives us:

|  |
| --- |
| def correction(word): return max(candidates(word), key=P)  def candidates(word):      return known([word]) or known(edits1(word)) or known(edits2(word)) or [word] |

**Evaluation**

Now it is time to evaluate how well this program does. After my plane landed, I downloaded Roger Mitton's [Birkbeck spelling error corpus](http://ota.ahds.ac.uk/texts/0643.html) from the Oxford Text Archive. From that I extracted two test sets of corrections. The first is for development, meaning I get to look at it while I'm developing the program. The second is a final test set, meaning I'm not allowed to look at it, nor change my program after evaluating on it. This practice of having two sets is good hygiene; it keeps me from fooling myself into thinking I'm doing better than I am by tuning the program to one specific set of tests. I also wrote some unit tests:

|  |
| --- |
| def unit\_tests():     assert correction('speling') == 'spelling'              # insert     assert correction('korrectud') == 'corrected'           # replace 2     assert correction('bycycle') == 'bicycle'               # replace     assert correction('inconvient') == 'inconvenient'       # insert 2     assert correction('arrainged') == 'arranged'            # delete     assert correction('peotry') =='poetry'                  # transpose     assert correction('peotryy') =='poetry'                 # transpose + delete     assert correction('word') == 'word'                     # known     assert correction('quintessential') == 'quintessential' # unknown     assert words('This is a TEST.') == ['this', 'is', 'a', 'test']     assert Counter(words('This is a test. 123; A TEST this is.')) == (            Counter({'123': 1, 'a': 2, 'is': 2, 'test': 2, 'this': 2}))     assert len(WORDS) == 32192     assert sum(WORDS.values()) == 1115504     assert WORDS.most\_common(10) == [      ('the', 79808),      ('of', 40024),      ('and', 38311),      ('to', 28765),      ('in', 22020),      ('a', 21124),      ('that', 12512),      ('he', 12401),      ('was', 11410),      ('it', 10681)]     assert WORDS['the'] == 79808     assert P('quintessential') == 0     assert 0.07 < P('the') < 0.08     return 'unit\_tests pass'  def spelltest(tests, verbose=False):     "Run correction(wrong) on all (right, wrong) pairs; report results."     import time     start = time.clock()     good, unknown = 0, 0     n = len(tests)     for right, wrong in tests:         w = correction(wrong)         good += (w == right)         if w != right:             unknown += (right not in WORDS)             if verbose:                 print('correction({}) => {} ({}); expected {} ({})'                       .format(wrong, w, WORDS[w], right, WORDS[right]))     dt = time.clock() - start     print('{:.0%} of {} correct ({:.0%} unknown) at {:.0f} words per second '           .format(good / n, n, unknown / n, n / dt))      def Testset(lines):     "Parse 'right: wrong1 wrong2' lines into [('right', 'wrong1'), ('right', 'wrong2')] pairs."     return [(right, wrong)             for (right, wrongs) in (line.split(':') for line in lines)             for wrong in wrongs.split()]  print(unit\_tests()) spelltest(Testset(open('[spell-testset1.txt](https://norvig.com/spell-testset1.txt)'))) # Development set spelltest(Testset(open('[spell-testset2.txt](https://norvig.com/spell-testset1.txt)'))) # Final test set |

This gives the output:

|  |
| --- |
| unit\_tests pass 75% of 270 correct at 41 words per second 68% of 400 correct at 35 words per second None |

So on the development set we get 75% correct (processing words at a rate of 41 words/second), and on the final test set we get 68% correct (at 35 words/second). In conclusion, I met my goals for brevity, development time, and runtime speed, but not for accuracy. Perhaps my test set was extra tough, or perhaps my simple model is just not good enough to get to 80% or 90% accuracy.

**Future Work**

Let's think about how we could do better. (I've developed the ideas some more in a [separate chapter](http://norvig.com/ngrams/) for a book and in a [Jupyter notebook](http://nbviewer.jupyter.org/url/norvig.com/ipython/How%20to%20Do%20Things%20with%20Words.ipynb).)

1. P(*c*), the language model. We can distinguish two sources of error in the language model. The more serious is unknown words. In the development set, there are 15 unknown words, or 5%, and in the final test set, 43 unknown words or 11%. Here are some examples of the output of spelltest with verbose=True):

**correction('transportibility') => 'transportibility' (0);**

**expected 'transportability' (0)**

**correction('addresable') => 'addresable' (0);**

**expected 'addressable' (0)**

**correction('auxillary') => 'axillary' (31);**

**expected 'auxiliary' (0)**

In this output we show the call to correction and the actual and expected results (with the WORDS counts in parentheses). Counts of (0) mean the target word was not in the dictionary, so we have no chance of getting it right. We could create a better language model by collecting more data, and perhaps by using a little English morphology (such as adding "ility" or "able" to the end of a word).

Another way to deal with unknown words is to allow the result of correction to be a word we have not seen. For example, if the input is "electroencephalographicallz", a good correction would be to change the final "z" to an "y", even though "electroencephalographically" is not in our dictionary. We could achieve this with a language model based on components of words: perhaps on syllables or suffixes, but it is easier to base it on sequences of characters: common 2-, 3- and 4-letter sequences.

1. P(*w*|*c*), the error model. So far, the error model has been trivial: the smaller the edit distance, the smaller the error. This causes some problems, as the examples below show. First, some cases where correction returns a word at edit distance 1 when it should return one at edit distance 2:

**correction('reciet') => 'recite' (5);**

**expected 'receipt' (14)**

**correction('adres') => 'acres' (37);**

**expected 'address' (77)**

**correction('rember') => 'member' (51);**

**expected 'remember' (162)**

**correction('juse') => 'just' (768);**

**expected 'juice' (6)**

**correction('accesing') => 'acceding' (2);**

**expected 'assessing' (1)**

Why should "adres" be corrected to "address" rather than "acres"? The intuition is that the two edits from "d" to "dd" and "s" to "ss" should both be fairly common, and have high probability, while the single edit from "d" to "c" should have low probability.

Clearly we could use a better model of the cost of edits. We could use our intuition to assign lower costs for doubling letters and changing a vowel to another vowel (as compared to an arbitrary letter change), but it seems better to gather data: to get a corpus of spelling errors, and count how likely it is to make each insertion, deletion, or alteration, given the surrounding characters. We need a lot of data to do this well. If we want to look at the change of one character for another, given a window of two characters on each side, that's 266, which is over 300 million characters. You'd want several examples of each, on average, so we need at least a billion characters of correction data; probably safer with at least 10 billion.

Note there is a connection between the language model and the error model. The current program has such a simple error model (all edit distance 1 words before any edit distance 2 words) that it handicaps the language model: we are afraid to add obscure words to the model, because if one of those obscure words happens to be edit distance 1 from an input word, then it will be chosen, even if there is a very common word at edit distance 2. With a better error model we can be more aggressive about adding obscure words to the dictionary. Here are some examples where the presence of obscure words in the dictionary hurts us:

**correction('wonted') => 'wonted' (2);**

**expected 'wanted' (214)**

**correction('planed') => 'planed' (2);**

**expected 'planned' (16)**

**correction('forth') => 'forth' (83);**

**expected 'fourth' (79)**

**correction('et') => 'et' (20);**

**expected 'set' (325)**

1. The enumeration of possible corrections, argmax*c*. Our program enumerates all corrections within edit distance 2. In the development set, only 3 words out of 270 are beyond edit distance 2, but in the final test set, there were 23 out of 400. Here they are:

**purple perpul**

**curtains courtens**

**minutes muinets**

**successful sucssuful**

**hierarchy heiarky**

**profession preffeson**

**weighted wagted**

**inefficient ineffiect**

**availability avaiblity**

**thermawear thermawhere**

**nature natior**

**dissension desention**

**unnecessarily unessasarily**

**disappointing dissapoiting**

**acquaintances aquantences**

**thoughts thorts**

**criticism citisum**

**immediately imidatly**

**necessary necasery**

**necessary nessasary**

**necessary nessisary**

**unnecessary unessessay**

**night nite**

**minutes muiuets**

**assessing accesing**

**necessitates nessisitates**

We could consider extending the model by allowing a limited set of edits at edit distance 3. For example, allowing only the insertion of a vowel next to another vowel, or the replacement of a vowel for another vowel, or replacing close consonants like "c" to "s" would handle almost all these cases.

1. There's actually a fourth (and best) way to improve: change the interface to correction to look at more context. So far, correction only looks at one word at a time. It turns out that in many cases it is difficult to make a decision based only on a single word. This is most obvious when there is a word that appears in the dictionary, but the test set says it should be corrected to another word anyway:

**correction('where') => 'where' (123);**

**expected 'were' (452)**

**correction('latter') => 'latter' (11);**

**expected 'later' (116)**

**correction('advice') => 'advice' (64);**

**expected 'advise' (20)**

We can't possibly know that correction('where') should be 'were' in at least one case, but should remain 'where' in other cases. But if the query had been correction('They where going') then it seems likely that "where" should be corrected to "were".

The context of the surrounding words can help when there are obvious errors, but two or more good candidate corrections. Consider:

**correction('hown') => 'how' (1316);**

**expected 'shown' (114)**

**correction('ther') => 'the' (81031);**

**expected 'their' (3956)**

**correction('quies') => 'quiet' (119);**

**expected 'queries' (1)**

**correction('natior') => 'nation' (170);**

**expected 'nature' (171)**

**correction('thear') => 'their' (3956);**

**expected 'there' (4973)**

**correction('carrers') => 'carriers' (7);**

**expected 'careers' (2)**

Why should 'thear' be corrected as 'there' rather than 'their'? It is difficult to tell by the single word alone, but if the query were correction('There's no there thear') it would be clear.

To build a model that looks at multiple words at a time, we will need a lot of data. Fortunately, Google has released a [database of word counts](http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html) for all sequences up to five words long, gathered from a corpus of a *trillion* words.

I believe that a spelling corrector that scores 90% accuracy will *need* to use the context of the surrounding words to make a choice. But we'll leave that for another day...

We could also decide what dialect we are trying to train for. The following three errors are due to confusion about American versus British spelling (our training data contains both):

**correction('humor') => 'humor' (17);**

**expected 'humour' (5)**

**correction('oranisation') => 'organisation' (8);**

**expected 'organization' (43)**

**correction('oranised') => 'organised' (11);**

**expected 'organized' (70)**

1. – **5** - Finally, we could improve the implementation by making it much faster, without changing the results. We could re-implement in a compiled language rather than an interpreted one. We could cache the results of computations so that we don't have to repeat them multiple times. One word of advice: before attempting any speed optimizations, profile carefully to see where the time is actually going.

**Further Reading**

* Roger Mitton has a [survey article](http://www.dcs.bbk.ac.uk/~roger/spellchecking.html) on spell checking.
* Jurafsky and Martin cover spelling correction well in their text [*Speech and Language Processing*](http://www.cs.colorado.edu/~martin/slp.html).
* Manning and Schutze cover statistical language models very well in their text [*Foundations of Statistical Natural Language Processing*](http://nlp.stanford.edu/fsnlp/), but they don't seem to cover spelling (at least it is not in the index).
* The [aspell](http://aspell.net) project has a lot of interesting material, including some [test data](http://aspell.net/test/) that seems better than what I used.
* The [LingPipe](http://alias-i.com/lingpipe) project has a [spelling tutorial](http://alias-i.com/lingpipe/demos/tutorial/querySpellChecker/read-me.html).

**Acknowledgments**

Ivan Peev, Jay Liang, Dmitriy Ryaboy and Darius Bacon pointed out problems in [earlier versions](https://web.archive.org/web/*/http:/norvig.com/spell-correct.html) of this document.

**Other Computer Languages**

After I posted this article, various people wrote versions in different programming languages. These may be interesting for those who like comparing languages, or for those who want to borrow an implementation in their desired target language:

|  |  |  |
| --- | --- | --- |
| **Language** | **Lines Code** | **Author (and link to implementation)** |
| Awk | 15 | [Tiago "PacMan" Peczenyj](http://pacman.blog.br/wiki/index.php?title=Um_Corretor_Ortogr%C3%A1fico_em_GAWK) |
| Awk | 28 | [Gregory Grefenstette](http://feedback.exalead.com/feedbacks/191466-spell-checking) |
| C | 184 | [Marcelo Toledo](http://blog.marcelotoledo.org/2007/08/10/how-to-write-a-spelling-corrector/) |
| C++ | 98 | [Felipe Farinon](http://scarvenger.wordpress.com/2007/12/11/how-to-write-a-spelling-corrector/) |
| C# | 43 | [Lorenzo Stoakes](https://github.com/lorenzo-stoakes/spell-correct) |
| C# | 69 | [Frederic Torres](http://frederictorres.blogspot.com/2011/04/how-to-write-spelling-corrector-from.html) |
| C# | 160 | [Chris Small](http://www.anotherchris.net/csharp/how-to-write-a-spelling-corrector-in-csharp/) |
| C# | --- | [João Nuno Carvalho](https://github.com/joaocarvalhoopen/USB_SpellChecker_GUI_in_C_Sharp/blob/master/SpellChecker_GUI/SpellChecker_GUI/TweakedPeterNorvigSpellChecker.cs) |
| Clojure | 18 | [Rich Hickey](http://en.wikibooks.org/wiki/Clojure_Programming/Examples#Norvig.27s_Spelling_Corrector) |
| Coffeescript | 21 | [Daniel Ribeiro](https://metaphysicaldeveloper.wordpress.com/2011/03/31/354/) |
| D | 23 | [Leonardo M](http://leonardo-m.livejournal.com/59589.html) |
| Erlang | 87 | [Federico Feroldi](http://www.pixzone.com/blog/223/spell-corrector-aka-google-suggest-in-erlang-first-part/) |
| F# | 16 | [Dejan Jelovic](http://www.jelovic.com/weblog/?p=201) |
| F# | 34 | [Sebastian G](http://cs.hubfs.net/forums/thread/3085.aspx) |
| Go | 57 | [Yi Wang](http://cxwangyi.wordpress.com/2012/02/15/peter-norvigs-spelling-corrector-in-go/) |
| Groovy | 22 | [Rael Cunha](http://raelcunha.com/spell-correct.php#groovy) |
| Haskell | 24 | [Grzegorz](http://pithekos.net/brainwave/) |
| Java 8 | 23 | [Peter Kuhar](https://github.com/unrelatedlabs/SpellingCorrector-Java8) |
| Java | 35 | [Rael Cunha](http://raelcunha.com/spell-correct.php) |
| Java | 372 | [Dominik Schulz](http://developer.gauner.org/jspellcorrect/) |
| Javascript | 92 | [Shine Xavier](http://stoi.wordpress.com/2012/12/31/jspell/) |
| Javascript | 53 | [Panagiotis Astithas](http://astithas.blogspot.com/2009/08/spell-checking-in-javascript.html) |
| Lisp | 26 | [Mikael Jansson](https://github.com/mikaelj/snippets/blob/master/lisp/spellcheck/spellcheck.lisp) |
| OCaml | 148 | [Stefano Pacifico](http://spacifico.org/programming/norvig-spell-corrector-ocaml) |
| Perl | 63 | [riffraff](http://www.riffraff.info/2007/5/20/a-spell-corrector-in-perl6-part-3) |
| PHP | 68 | [Felipe Ribeiro](http://www.phpclasses.org/browse/package/4859.html) |
| PHP | 103 | [Joe Sanders](http://soundofemotion.com/spellcorrect.txt) |
| R | 2 | [Rasmus Bååth](http://www.sumsar.net/blog/2014/12/peter-norvigs-spell-checker-in-two-lines-of-r/) |
| Rebol | 133 | [Cyphre](http://www.rebol.cz/~cyphre/spell.r) |
| Ruby | 34 | [Brian Adkins](http://lojic.com/blog/2008/09/04/how-to-write-a-spelling-corrector-in-ruby/) |
| Scala | 20 | [Pathikrit Bhowmick](https://gist.github.com/pathikrit/d5b26fe1c166a97e2162) |
| Scala | 23 | [Thomas Jung](http://theyougen.blogspot.com/2009/12/peter-norvigs-spelling-corrector-in.html) |
| Scheme | 45 | [Shiro](http://practical-scheme.net/wiliki/wiliki.cgi?Gauche%3aSpellingCorrection&l=en) |
| Scheme | 89 | [Jens Axel](http://scheme.dk/blog/2007/04/writing-spelling-corrector-in-plt.html) |
| Swift | 108 | [Airspeed Velocity](http://airspeedvelocity.net/2015/05/02/spelling/) |

**Other Natural Languages**

This essay has been translated into:

* [Simplified Chinese](http://blog.youxu.info/spell-correct.html) by Eric You XU
* [Japanese](http://www.aoky.net/articles/peter_norvig/spell-correct.htm) by Yasushi Aoki
* [Korean](http://theyearlyprophet.com/spell-correct.html) by JongMan Koo
* [Russian](http://gmdidro.googlepages.com/Ru_HowtoWriteaSpellingCorrector.html) by Petrov Alexander

Thanks to all the authors for creating these implementations and translations.

[*Peter Norvig*](http://norvig.com)