How to Write a Spelling Corrector

In the past week, two friends (Dean and Bill) independently told me they were amazed at how Google does spelling correction so well and quickly. Type in a search like [speling] and Google comes back in 0.1 seconds or so with Did you mean: ***spelling***. (Yahoo and Microsoft are similar.) What surprised me is that I thought Dean and Bill, being highly accomplished engineers and mathematicians, would have good intuitions about statistical language processing problems such as spelling correction. But they didn't, and come to think of it, there's no reason they should: it was my expectations that were faulty, not their knowledge.

I figured they and many 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 or here). What I wanted to do here is to develop, in less than a page of code, a toy spelling corrector that achieves 80 or 90% accuracy at a processing speed of at least 10 words per second.

So here, in 21 lines of Python 2.5 code, is the complete spelling corrector:

```
import re, collections

def words(text): return re.findall('[a-z]+', text.lower())

def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model

NWORDS = train(words(file('big.txt').read()))

alphabet = 'abcdefghijklmnopqrstuvwxyz'

def edits1(word):
   splits     = [(word[:i], word[i:]) for i in range(len(word) + 1)]
   deletes    = [a + b[1:] for a, b in splits if b]
   transposes = [a + b[1] + b[0] + b[2:] for a, b in splits if len(b)>1]
   replaces   = [a + c + b[1:] for a, b in splits for c in alphabet if b]
   inserts    = [a + c + b     for a, b in splits for c in alphabet]
   return set(deletes + transposes + replaces + inserts)

def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in NWORDS)

def known(words): return set(w for w in words if w in NWORDS)

def correct(word):
    candidates = known([word]) or known(edits1(word)) or known_edits2(word) or [word]
    return max(candidates, key=NWORDS.get)
```

The code defines the function `correct`, which takes a word as input and returns a likely correction of that word. For example:

```
>>> correct('speling')
'spelling'
>>> correct('korrecter')
'corrector'
```

The version of `edits1` shown here is a variation on one proposed by Darius Bacon; I think this is clearer than the version I originally had. Darius also fixed a bug in the function `correct`.

How It Works: Some Probability Theory

How does it work? First, a little theory. Given a word, we are trying to choose the most likely spelling correction for that word (the "correction" may be the original word itself). There is no way to know for sure (for example, should "lates" be corrected to "late" or "latest"?), which suggests we use probabilities. We will say that we are trying to find the correction *c*, out of all possible corrections, that maximizes the probability of *c* given the original word *w*:

$$\text{argmax}_c \ P(c|w)$$

By Bayes' Theorem this is equivalent to:

$$\text{argmax}_c \ P(w|c) \ P(c) \ / \ P(w)$$

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

$$\text{argmax}_c \ P(w|c) \ P(c)$$

There are three parts of this expression. From right to left, we have:

1. P(*c*), the probability that a proposed correction *c* stands on its own. This is called the **language model**: think of it as answering the question "how likely is *c* to appear in an English text?" So P("the") would have a relatively high probability, while P("zxzxzxzyyy") would be near zero.

2. P(*w*|*c*), the probability that *w* would be typed in a text when the author meant *c*. This is the **error model**: think of it as answering "how likely is it that the author would type *w* by mistake when *c* was intended?"

3. argmax_c, the control mechanism, which says to enumerate all feasible values of *c*, and then choose the one that gives the best combined probability score.

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 perhaps the typist's finger slipped off the "e" onto the "w". 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.

Now we are ready to show how the program works. First P(*c*). We will read a big text file, big.txt, which consists of about a million words. The file is a concatenation of several public domain books from Project Gutenberg and lists of most frequent words from Wiktionary and the British National Corpus. (On the plane home when I was writing the first version of the code all I had was a collection of Sherlock Holmes stories that happened to be on my laptop; I added the other sources later and stopped adding texts when they stopped helping, as we shall see in the Evaluation section.)

We then extract the individual words from the file (using the function `words`, which converts everything to lowercase, so that "the" and "The" will be the same and then defines a word as a sequence of alphabetic characters, so "don't" will be seen as the two words "don" and "t"). Next we train a probability model, which is a fancy way of saying we count how many times each word occurs, using the function `train`. It looks like this:

```
def words(text): return re.findall('[a-z]+', text.lower())

def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model

NWORDS = train(words(file('big.txt').read()))
```

At this point, `NWORDS[w]` holds a count of how many times the word `w` has been seen. There is one complication: novel words. What happens with a perfectly good word of English that wasn't seen in our training data? It would be bad form to say the probability of a word is zero just because we haven't seen it yet. There are several standard approaches to this problem; we take the easiest one, which is to treat novel words as if we had seen them once. This general process is called **smoothing**, because we are smoothing over the parts of the probability distribution that would have been zero, bumping them up to the smallest possible count. This is achieved through the class `collections.defaultdict`, which is like a regular Python dict (what other languages call hash tables) except that we can specify the default value of any key; here we use 1.

Now let's look at the problem of enumerating the possible corrections *c* of a given word *w*. It is common to talk of the **edit distance** between two words: the number of edits it would take to turn one into the other. An edit can be a deletion (remove one letter), a transposition (swap adjacent letters), an alteration (change one letter to another) or an insertion (add a letter). Here's a function that returns a set of all words *c* that are one edit away from *w*:

```
def edits1(word):
   splits     = [(word[:i], word[i:]) for i in range(len(word) + 1)]
   deletes    = [a + b[1:] for a, b in splits if b]
   transposes = [a + b[1] + b[0] + b[2:] for a, b in splits if len(b)>1]
   replaces   = [a + c + b[1:] for a, b in splits for c in alphabet if b]
   inserts    = [a + c + b     for a, b in splits for c in alphabet]
   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('something')) -- that is, the number of elements in the result of edits1('something') -- is 494.

The literature on spelling correction claims that 80 to 95% of spelling errors are an edit distance of 1 from the target. As we shall see shortly, I put together a development corpus of 270 spelling errors, and found that only 76% of them have edit distance 1. Perhaps the examples I found are harder than typical errors. Anyway, I thought this was not good enough, so we'll need to consider edit distance 2. That's easy: just apply `edits1` to all the results of `edits1`:

```
def edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1))
```

This is easy to write, but we're starting to get into some serious computation: len(edits2('something')) is 114,324. However, we do get good coverage: of the 270 test cases, only 3 have an edit distance greater than 2. That is, edits2 will cover 98.9% of the cases; that's good enough for me. Since we aren't going beyond edit distance 2, we can do a small optimization: only keep the candidates that are actually known words. We still have to consider all the possibilities, but we don't have to build up a big set of them. The function `known_edits2` does this:

```
def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in NWORDS)
```

Now, for example, known_edits2('something') is a set of just 4 words: {'smoothing', 'seething', 'something', 'soothing'}, rather than the set of 114,324 words generated by edits2. That speeds things up by about 10%.

Now the only part left is the error model, P(*w*|*c*). Here's where I ran into difficulty. Sitting on the plane, with no internet connection, I was stymied: I had no training data to build a model of spelling errors. I had some intuitions: mistaking one vowel for another is more probable than mistaking two consonants; making an error on the first letter of a word is less probable, etc. But I had no numbers to back that up. So I took a shortcut: I defined a trivial 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. By "known word" I mean a word that we have seen in the language model training data -- a word in the dictionary. We can implement this strategy as follows:

```
def known(words): return set(w for w in words if w in NWORDS)

def correct(word):
    candidates = known([word]) or known(edits1(word)) or known_edits2(word) or [word]
    return max(candidates, key=NWORDS.get)
```

The function `correct` chooses as the set of candidate words the set with the shortest edit distance to the original word, as long as the set has some known words. Once it identifies the candidate set to consider, it chooses the element with the highest P(*c*) value, as estimated by the `NWORDS` model.

Evaluation

Now it is time to evaluate how well this program does. On the plane I tried a few examples, and it seemed okay. After my plane landed, I downloaded Roger Mitton's Birkbeck spelling error corpus 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. Here I show an excerpt of the two tests and the function to run them; to see the complete set of tests (along with the rest of the program), see the file spell.py.

```
tests1 = { 'access': 'acess', 'accessing': 'accesing', 'accommodation':
    'accomodation acommodation acomodation', 'account': 'acount', ...}

tests2 = {'forbidden': 'forbiden', 'decisions': 'deciscions descisions',
    'supposedly': 'supposidly', 'embellishing': 'embelishing', ...}

def spelltest(tests, bias=None, verbose=False):
    import time
    n, bad, unknown, start = 0, 0, 0, time.clock()
    if bias:
        for target in tests: NWORDS[target] += bias
    for target,wrongs in tests.items():
        for wrong in wrongs.split():
            n += 1
            w = correct(wrong)
            if w!=target:
                bad += 1
                unknown += (target not in NWORDS)
                if verbose:
                    print '%r => %r (%d); expected %r (%d)' % (
                        wrong, w, NWORDS[w], target, NWORDS[target])
    return dict(bad=bad, n=n, bias=bias, pct=int(100. - 100.*bad/n),
                unknown=unknown, secs=int(time.clock()-start) )

print spelltest(tests1)
print spelltest(tests2) ## only do this after everything is debugged
```

This gives the following output:

```
{'bad': 68, 'bias': None, 'unknown': 15, 'secs': 16, 'pct': 74, 'n': 270}
{'bad': 130, 'bias': None, 'unknown': 43, 'secs': 26, 'pct': 67, 'n': 400}
```

So on the development set of 270 cases, we get 74% correct in 13 seconds (a rate of 17 Hz), and on the final test set we get 67% correct (at 15 Hz).

> **Update:** *In the original version of this essay I incorrectly reported a higher score on both test sets, due to a bug. The bug was subtle, but I should have caught it, and I apologize for misleading those who read the earlier version. In the original version of `spelltest`, I left out the `if bias:` in the fourth line of the function (and the default value was bias=0, not bias=None). I figured that when bias = 0, the statement `NWORDS[target] += bias` would have no effect. In fact it does not change the value of `NWORDS[target]`, but it does have an effect: it makes `(target in NWORDS)` true. So in effect the `spelltest` routine was cheating by making all the unknown words known. This was a humbling error, and I have to admit that much as I like `defaultdict` for the brevity it adds to programs, I think I would not have had this bug if I had used regular dicts.*
>
> **Update 2:** *defaultdict strikes again. Darius Bacon pointed out that in the function `correct`, I had accessed `NWORDS[w]`. This has the unfortunate side-effect of adding `w` to the defaultdict, if `w` was not already there (i.e., if it was an unknown word). Then the next time, it would be present, and we would get the wrong answer. Darius correctly suggested changing to `NWORDS.get`. (This works because `max(None, i)` is `i` for any integer `i`.)*

In conclusion, I met my goals for brevity, development time, and runtime speed, but not for accuracy.

Future Work

Let's think about how we could do better. (I've done some more in a separate chapter for a book.) We'll again look at all three factors of the probability model: (1) P(*c*); (2) P(*w*|*c*); and (3) argmaxc. We'll look at examples of what we got wrong. Then we'll look at some factors beyond the three...

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`:

   ```
   correct('economtric') => 'economic' (121); expected 'econometric' (1)
   ```

```
correct('embaras') => 'embargo' (8); expected 'embarrass' (1)
correct('colate') => 'coat' (173); expected 'collate' (1)
correct('orentated') => 'orentated' (1); expected 'orientated' (1)
correct('unequivocaly') => 'unequivocal' (2); expected 'unequivocally' (1)
correct('generataed') => 'generate' (2); expected 'generated' (1)
correct('guidlines') => 'guideline' (2); expected 'guidelines' (1)
```

In this output we show the call to `correct` and the result (with the `NWORDS` count for the result in parentheses), and then the word expected by the test set (again with the count in parentheses). What this shows is that if you don't know that 'econometric' is a word, you're not going to be able to correct 'economtric'. We could mitigate by adding more text to the training corpus, but then we also add words that might turn out to be the wrong answer. Note the last four lines above are inflections of words that do appear in the dictionary in other forms. So we might want a model that says it is okay to add '-ed' to a verb or '-s' to a noun.

The second potential source of error in the language model is bad probabilities: two words appear in the dictionary, but the wrong one appears more frequently. I must say that I couldn't find cases where this is the only fault; other problems seem much more serious.

We can simulate how much better we might do with a better language model by cheating on the tests: pretending that we have seen the correctly spelled word 1, 10, or more times. This simulates having more text (and just the right text) in the language model. The function `spelltest` has a parameter, `bias`, that does this. Here's what happens on the development and final test sets when we add more bias to the correctly-spelled words:

Bias	Dev.	Test
0	74%	67%
1	74%	70%
10	76%	73%
100	82%	77%
1000	89%	80%

On both test sets we get significant gains, approaching 80-90%. This suggests that it is possible that if we had a good enough language model we might get to our accuracy goal. On the other hand, this is probably optimistic, because as we build a bigger language model we would also introduce words that are the wrong answer, which this method does not do.

Another way to deal with unknown words is to allow the result of `correct` 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 (such as "-ally"), but it is far easier to base it on sequences of characters: 2-, 3- and 4-letter sequences.

2. 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 `correct` returns a word at edit distance 1 when it should return one at edit distance 2:

```
correct('reciet') => 'recite' (5); expected 'receipt' (14)
correct('adres') => 'acres' (37); expected 'address' (77)
correct('rember') => 'member' (51); expected 'remember' (162)
correct('juse') => 'just' (768); expected 'juice' (6)
correct('accesing') => 'acceding' (2); expected 'assessing' (1)
```

Here, for example, the alteration of 'd' to 'c' to get from 'adres' to 'acres' should count more than the sum of the two changes 'd' to 'dd' and 's' to 'ss'.

Also, some cases where we choose the wrong word at the same edit distance:

```
correct('thay') => 'that' (12513); expected 'they' (4939)
correct('cleark') => 'clear' (234); expected 'clerk' (26)
correct('wer') => 'her' (5285); expected 'were' (4290)
correct('bonas') => 'bones' (263); expected 'bonus' (3)
correct('plesent') => 'present' (330); expected 'pleasant' (97)
```

The same type of lesson holds: In 'thay', changing an 'a' to an 'e' should count as a smaller change than changing a 'y' to a 't'. How much smaller? It must be a least a factor of 2.5 to overcome the prior probability advantage of 'that' over 'they'.

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 26^6, 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:

```
correct('wonted') => 'wonted' (2); expected 'wanted' (214)
correct('planed') => 'planed' (2); expected 'planned' (16)
correct('forth') => 'forth' (83); expected 'fourth' (79)
correct('et') => 'et' (20); expected 'set' (325)
```

3. 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.

4. There's actually a fourth (and best) way to improve: change the interface to `correct` to look at more context. So far, `correct` 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:

```
correct('where') => 'where' (123); expected 'were' (452)
correct('latter') => 'latter' (11); expected 'later' (116)
correct('advice') => 'advice' (64); expected 'advise' (20)
```

We can't possibly know that `correct('where')` should be 'were' in at least one case, but should remain 'where' in other cases. But if the query had been `correct('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:

```
correct('hown') => 'how' (1316); expected 'shown' (114)
correct('ther') => 'the' (81031); expected 'their' (3956)
correct('quies') => 'quiet' (119); expected 'queries' (1)
correct('natior') => 'nation' (170); expected 'nature' (171)
correct('thear') => 'their' (3956); expected 'there' (4973)
correct('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 `correct('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 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...

5. We could improve our accuracy scores by improving the training data and the test data. We grabbed a million words of text and assumed they were all spelled correctly; but it is very likely that the training data

contains several errors. We could try to identify and fix those. Less daunting a task is to fix the test sets. I noticed at least three cases where the test set says our program got the wrong answer, but I believe the program's answer is better than the expected answer:

```
correct('aranging') => 'arranging' (20); expected 'arrangeing' (1)
correct('sumarys') => 'summary' (17); expected 'summarys' (1)
correct('aurgument') => 'argument' (33); expected 'auguments' (1)
```

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):

```
correct('humor') => 'humor' (17); expected 'humour' (5)
correct('oranisation') => 'organisation' (8); expected 'organization' (43)
correct('oranised') => 'organised' (11); expected 'organized' (70)
```

6. 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 have a lookup table that is specialized to strings rather than Python's general-purpose dict. 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 on spell checking.
- Jurafsky and Martin cover spelling correction well in their text *Speech and Language Processing*.
- Manning and Schutze cover statistical language models very well in their text *Foundations of Statistical Natural Language Processing*, but they don't seem to cover spelling (at least it is not in the index).
- The aspell project has a lot of interesting material, including some test data that seems better than what I used.
- The LingPipe project has a spelling tutorial.

Errata

Originally my program was 20 lines, but Ivan Peev pointed out that I had used `string.lowercase`, which in some locales in some versions of Python, has more characters than just the `a-z` I intended. So I added the variable `alphabet` to make sure. I could have used `string.ascii_lowercase`.

Thanks to Jay Liang for pointing out there are only 54n+25 distance 1 edits, not 55n+25 as I originally wrote.

Thanks to Dmitriy Ryaboy for pointing out there was a problem with unknown words; this allowed me to find the `NWORDS[target] += bias` bug.

Other Computer Languages

After I posted this article, various people wrote versions in different programming languages. While the purpose of this article was to show the algorithms, not to highlight Python (and certainly not to play "code golf" in an attempt to find the shortest program), the other examples may be interesting for those who like comparing languages, or for those who want to borrow an implementation in their desired language:

Language	Lines Code	Author (and link to implementation)
Awk	15	Tiago "PacMan" Peczenyj
Awk	28	Gregory Grefenstette
C	184	Marcelo Toledo
C++	98	Felipe Farinon
C#	43	Lorenzo Stoakes
C#	69	Frederic Torres
C#	160	Chris Small
Clojure	18	Rich Hickey
D	23	Leonardo M
Erlang	87	Federico Feroldi
F#	16	Dejan Jelovic
F#	34	Sebastian G
Go	57	Yi Wang
Groovy	22	Rael Cunha
Haskell	24	Grzegorz
Java	35	Rael Cunha
Java	372	Dominik Schulz
Javascript	92	Shine Xavier
Javascript	53	Panagiotis Astithas
Lisp	26	Mikael Jansson
Perl	63	riffraff
PHP	68	Felipe Ribeiro
PHP	103	Joe Sanders
Python	21	Peter Norvig
Rebol	133	Cyphre
Ruby	34	Brian Adkins
Scala	23	Thomas Jung
Scheme	45	Shiro
Scheme	89	Jens Axel

Other Natural Languages

This essay has been translated into:

- Simplified Chinese by Eric You XU
- Japanese by Yasushi Aoki
- Korean by JongMan Koo
- Russian by Petrov Alexander

- 60 languages by Google Translate:

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

Peter Norvig