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 or here). 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):

```
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:

$$\text{argmax}_{c \in candidates}\ P(c|w)$$

By Bayes' Theorem this is equivalent to:

$$\text{argmax}_{c \in candidates}\ P(c)\ P(w|c)\ /\ P(w)$$

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

$$\text{argmax}_{c \in candidates}\ P(c)\ P(w|c)$$

The four parts of this expression are:

1. **Selection Mechanism**: argmax
   We choose the candidate with the highest combined probability.

2. **Candidate Model**: $c \in candidates$

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`.

3. **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`. It is a concatenation of public domain book excerpts from Project Gutenberg and lists of most frequent words from Wiktionary and the British National Corpus. 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
```

```
 ('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
```

4. **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(*wlc*) 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 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."
```

```
    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'))) # Development set
spelltest(Testset(open('spell-testset2.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 for a book and in a Jupyter notebook.)

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.

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 `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 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

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 `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 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)
```

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

Acknowledgments

Ivan Peev, Jay Liang, Dmitriy Ryaboy and Darius Bacon pointed out problems in earlier versions 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
Awk	28	Gregory Grefenstette
C	184	Marcelo Toledo
C++	98	Felipe Farinon
C#	43	Lorenzo Stoakes
C#	69	Frederic Torres
C#	160	Chris Small
C#	---	João Nuno Carvalho
Clojure	18	Rich Hickey
Coffeescript	21	Daniel Ribeiro
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 8	23	Peter Kuhar
Java	35	Rael Cunha
Java	372	Dominik Schulz
Javascript	92	Shine Xavier
Javascript	53	Panagiotis Astithas
Lisp	26	Mikael Jansson
OCaml	148	Stefano Pacifico
Perl	63	riffraff
PHP	68	Felipe Ribeiro
PHP	103	Joe Sanders
R	2	Rasmus Bååth
Rebol	133	Cyphre
Ruby	34	Brian Adkins
Scala	20	Pathikrit Bhowmick
Scala	23	Thomas Jung
Scheme	45	Shiro
Scheme	89	Jens Axel
Swift	108	Airspeed Velocity

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