Info 2950, Lecture 20

18 Apr 2017

Prob Set 6: due Mon night 24 Apr

Prob Set 7: due Tue night 2 May (?)

Prob Set 8: due Wed night 10 May (?)

Consider networks with power law exponent dependent on parameter p (Easley/Kleinberg 18.3)

Model has directed links (so more in the spirit of web pages than social network)

Add new page (node) j, give link (edge) to an earlier page, according to probabilistic rule:

(a) With probability p, page j links to page i chosen at random from all earlier pages;

(b) With probability $1 - p$, page j instead links to a page i chosen with probability proportional to i's current number of in-links.

(a) permits discovery of pages that start with zero in-links, (b) is "preferential attachment"

Let $x_j(t)$ be the number of in-links to node j at time t (the in-degree).

Condition that node has zero in-coming links when created: $x_j(j) = 0$.

Now determine the expected number of nodes with k in-links at time t.

Probability that a new node created at time $t+1$ links to node j:

$$p/t + (1-p)x_j(t)/t$$

(at time t, by rule (a) j is chosen from t nodes with uniform probability $1/t$,

and by rule (b) the choice is instead according to node j's fraction of in-links, $x_j(t)/t$).

Approximating with continuous time t (captures the essential behavior):

$$\frac{\mathrm{d}x_j(t)}{\mathrm{d}t} = p\frac{1}{t} + (1-p)\frac{x_j(t)}{t} = p\frac{1}{t} + q\frac{x_j(t)}{t}$$

$$\frac{\mathrm{d}x_j(t)}{\mathrm{d}t} = p\frac{1}{t} + (1-p)\frac{x_j(t)}{t} = p\frac{1}{t} + q\frac{x_j(t)}{t}$$

(where $q = 1 - p$), with the boundary condition $x_j(t = j) = 0$.

Rewrite as

$$\frac{\mathrm{d}x_j}{p + qx_j(t)} = \frac{\mathrm{d}t}{t},$$

hence integrates to $\ln(p + qx_j(t)) = q \ln t + c$, or equivalently

$$p + qx_j(t) = At^q \ ,$$

where A is a constant determined by the boundary condition.

Solving for $x_j(t)$,

$$x_j(t) = \frac{1}{q}(At^q - p) \ ,$$

the condition $0 = x_j(j) = \frac{1}{q}(Aj^q - p)$ implies that $A = p/j^q$, and the solution can be written

$$x_j(t) = \frac{p}{q}\Big((t/j)^q - 1\Big) \ .$$

To determine the number of nodes with degree k at (large) time t, can ignore second term:

$$x_j(t) \approx a(t/j)^q \ .$$

$$x_j(t) \approx a(t/j)^q \ .$$

The fraction of nodes $F(k)$ with $x_j(t) \geq k$ are those with j satisfying: $a(t/j)^q \geq k$, i.e., the early ones, with small j:

$$j/t \leq a^{1/q} k^{-1/q} \quad \Longrightarrow \quad F(k) = a^{1/q} k^{-1/q}$$

In the discrete time version, the fraction of nodes with degree *equal* to k, $Pr(x_i(t) = k)$, would be given by

$$F(k) - F(k+1) = -\big(F(k+\Delta k) - F(k)\big)/\Delta k$$

(subtract those with degree at least $k+1$ from those with degree at least k).

In the continuous version ($\Delta k \to 0$), and with the fraction with *at least* in-degree k behaving as $F(k) \sim k^{-1/q}$, then the fraction with exactly k is given by

$$f(k) = -\frac{\mathrm{d}F}{\mathrm{d}k} \sim -\frac{\mathrm{d}}{\mathrm{d}k} k^{-1/q} \sim k^{-1-1/q} \ ,$$

$$f(k) = -\frac{\mathrm{d}F}{\mathrm{d}k} \sim -\frac{\mathrm{d}}{\mathrm{d}k} k^{-1/q} \sim k^{-1-1/q} \ ,$$

This is a power law with exponent $\alpha = 1 + 1/q = 1 + 1/(1-p)$

The limit $p \to 1$ gives back the random network, where $\alpha \to \infty$ signals loss of the power law behavior (the tail is extinguished).

In the $p \to 0$ limit, the exponent $\alpha \to 2$, and the tail of the distribution is that much more pronounced.

Smaller p permits nodes with even larger in-degree, giving a longer tail.

(https://en.wikipedia.org/wiki/Barabási–Albert_model)

Notes on derivation of power law for preferential attachment

A slightly different preferential attachment process:

Let $d_i(t)$ be the degree $\deg(v_i)$ of node v_i at time t.

At time $t+1$ add new vertex v_{t+1} with m new edges to the earlier nodes v_i, with probability proportional to their degree $d_i(t)$:

$$Pr(\text{attaching to } v_i) = \frac{d_i(t)}{\sum_{j=1}^{n} d_j(t)} \ . \tag{1}$$

(You implement this in the current programming assignment for $m = 2$.)

Approximate probabilistic discrete time dynamics with continuous time deterministic process:

Suppose $d_i(t)$ depends on a continuous time t, with boundary condition that node v_i is created at time t_i with degree m, so that

$$d_i(t_i) = m \ , \tag{2}$$

sum of degrees $d_j(t)$ satisfies $\sum_{j=1}^{n} d_j(t) = 2mt$

(by time t have added mt edges, each contributes 1 to the degree of two nodes).

As new nodes and edges added according to (1), degree of v_i increases as

$$\frac{\partial}{\partial t} d_i(t) = \frac{m d_i(t)}{\sum_{j=1}^{n} d_j(t)} = \frac{d_i(t)}{2t} \tag{3}$$

(each has m independent chances).

Rewritten as $\partial d_i(t)/d_i(t) = \partial t/2t$, integrates to

$$\ln d_i(t) = \frac{1}{2} \ln t + \text{const} \ . \tag{4}$$

Exponentiating, we have $d_i(t) = Ct^{1/2}$, where the new constant C can be determined by the condition (2) and therefore (for $t \geq t_i$):

$$d_i(t) = m\sqrt{\frac{t}{t_i}} \ . \tag{5}$$

Again consider $F(k)$, the fraction of nodes that have degree $d_i(t)$ at least equal to k (where $k \geq m$ since nodes are created with degree m).

Substituting (5) into the condition $d_i(t) \geq k$ gives $m\sqrt{t/t_i} \geq k$, or equivalently

$$t_i \leq (m^2/k^2)t$$

So at any time t, the fraction of nodes with degree at least equal to k consists of those created in the first m^2/k^2 fraction of the time, and

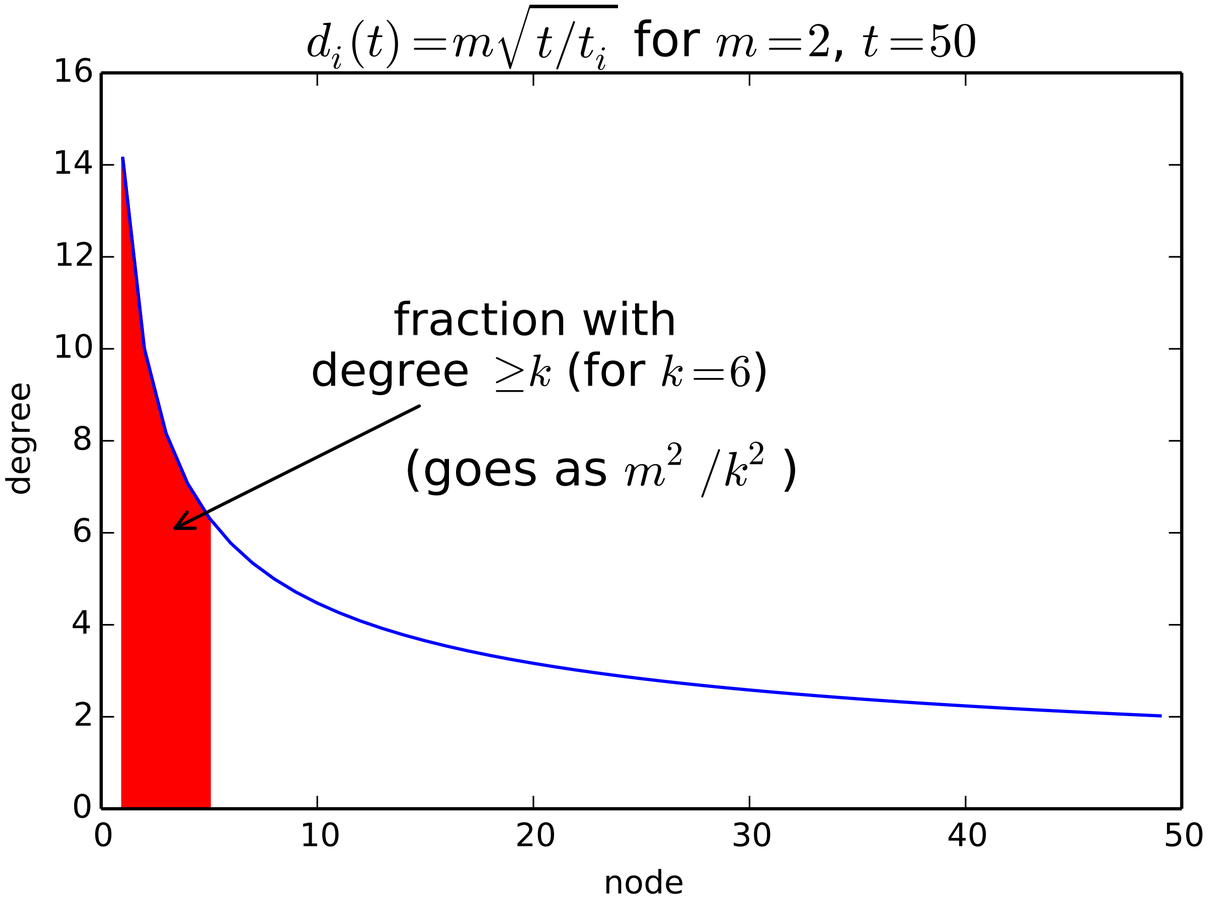
$$F(k) = m^2/k^2$$

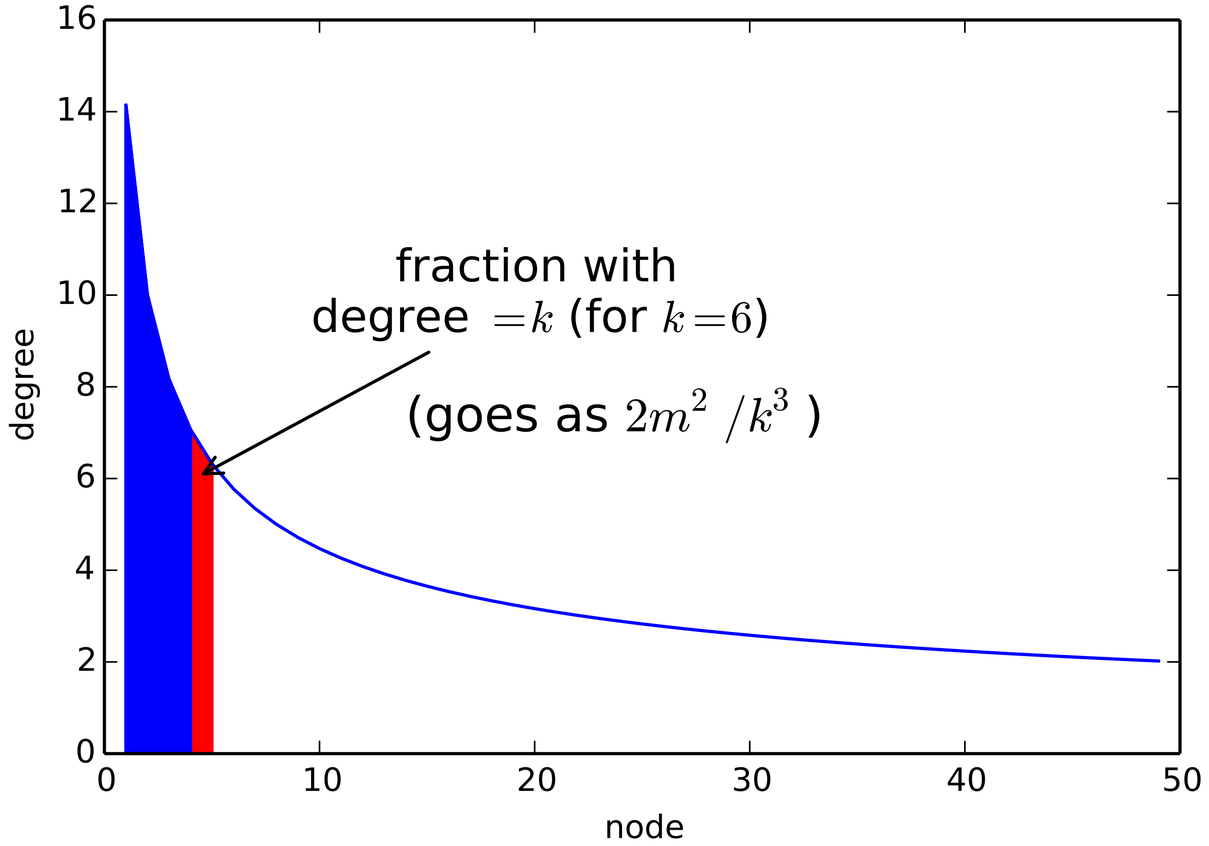
In the discrete time version, the fraction of nodes with degree equal to k, $Pr(d_i(t) = k)$, would be given by $F(k) - F(k+1) = -\big(F(k+\Delta k) - F(k)\big)/\Delta k$ (subtract those with degree at least $k+1$ from those with degree at least k). In the continuous version ($\Delta k \to 0$), that probability becomes (see figures following):

$$Pr(d_i(t) = k) = -\frac{\partial F}{\partial k} = \frac{2m^2}{k^3} \ ,$$

and we see the emergence of the power law behavior $k^{-\alpha}$ with exponent $\alpha = 3$.

The exponent itself does not depend on m, as will be seen in the programming assignment.


$d_i(t) = m\sqrt{t/t_i}$ for $m = 2$, $t = 50$
fraction with
degree $\geq k$ (for $k = 6$)
(goes as m^2 / k^2)
degree
node
0
2
4
6
8
10
12
14
16
0
10
20
30
40
50


fraction with
degree $= k$ (for $k = 6$)
(goes as $2m^2 / k^3$)
degree
node
0
2
4
6
8
10
12
14
16
0
10
20
30
40
50

More Statistical Methods

Peter Norvig, "How to Write a Spelling Corrector"
`http://norvig.com/spell-correct.html`

(See video:
`http://www.youtube.com/watch?v=yvDCzhbjYWs`
"The Unreasonable Effectiveness of Data", given 23 Sep 2010.)

Additional related references:
`http://doi.ieeecomputersociety.org/10.1109/MIS.2009.36`
A. Halevy, P. Norvig, F. Pereira,
The Unreasonable Effectiveness of Data,
Intelligent Systems Mar/Apr 2009 (copy at resources/unrealdata.pdf)

`http://norvig.com/ngrams/ch14.pdf`
P. Norvig, "Natural Language Corpus Data"

naive bayes

Bayes: $p(C|w) = p(w|C)p(C)/p(w)$

Naive: $p(\{w_i\}|C) = \prod_i p(w_i|C)$

- **spam filter (**$p(S|\{w_i\})/p(\overline{S}|\{w_i\})$**)**
- **text classification (on arXiv** $> 95\%$ **now)**
- **spell correction**
- **voice recognition**
- $\ldots$

simplest algorithm works better with more data.

for arXiv use multigram vocab: genetic_algorithm, black_hole

"The Unreasonable Effectiveness of Naive Bayes in the Data Sciences"

Error model

Language model

$p(c \mid w) \propto p(w \mid c)\, p(c)$

(spell correction)

Translation model

Language model

$p(e \mid f) \propto p(f \mid e)\, p(e)$

(machine translation)

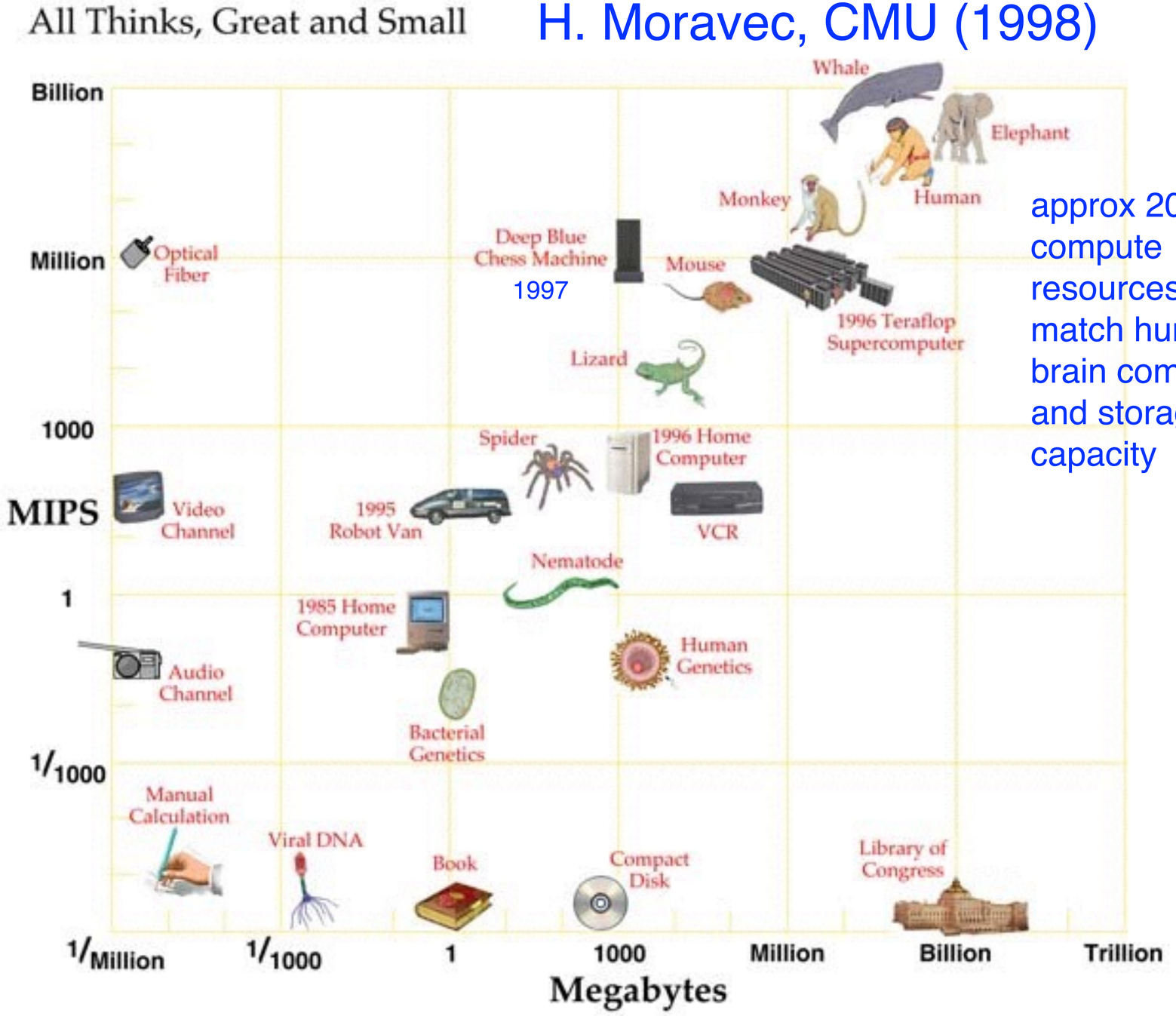
Acoustic model

Language model

$p(c \mid s) \propto p(s \mid c)\, p(c)$

(speech recognition)

H. Moravec, CMU (1998)



approx 2030 $1k compute resources will match human brain compute and storage capacity

General "Big Data" Procedure

- Define a probabilistic model
  (i.e., use data to create language model, a probability distribution over all strings in the language, learned from corpus, and use model to determine probability of candidates)
- Enumerate candidates
  (e.g., segmentations, corrected spellings)
- Choose the most probable candidate:

$$\text{best} = \text{argmax}_{c \in \text{candidates}} P(c)$$

Python: best = max(candidates, key=P)

Big Data = Simple Algorithm

http://norvig.com/spell-correct.html

How to Write a Spelling Corrector

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] and Google instantly comes back with **Showing results for: *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 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.

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

```
>>> correction('speling')
'spelling'

>>> correction('korrectud')
'corrected'
```

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))
```

A little theory

Find the correction c that maximizes the probability of c given the original word w:

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

By Bayes' Theorem, equivalent to $\operatorname{argmax}_c \; P(w|c)P(c)/P(w)$. $P(w)$ the same for every possible c, so ignore, and consider:

$$\operatorname{argmax}_c \; P(w|c)P(c) \; .$$

Three parts :

- $P(c)$, the probability that a proposed correction c stands on its own. The language model: "how likely is c to appear in an English text?" ($P($"the"$)$ high, $P($"zxzxzxzyyy"$)$ near zero)
- $P(w|c)$, the probability that w would be typed when author meant c. The error model: "how likely is author to type w by mistake instead of c?"
- argmax_c, the control mechanism: choose c that gives the best combined probability score.

Example

$w =$ "thew"

- two candidate corrections $c =$ "the" and $c =$ "thaw".
- which has higher $P(c|w)$?
- "thaw" has only small change "a" to "e"
- "the" is a very common word, and perhaps the typist's finger slipped off the "e" onto the "w".

To estimate $P(c|w)$, have to consider both the probability of c and the probability of the change from c to w

[Recall the joint probability "p of A given B", written $P(A|B)$, for events A and B, can be estimated by counting the number of times that A and B both occur, and dividing by the total number of times B occurs. Intuitively it is the fraction of times A occurs out of the total times that B occurs.]

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'
```

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:

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

(For word of length n: n deletions, n-1 transpositions, 26n alterations, and 26(n+1) insertions, for a total of 54n+25 at edit distance 1)

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():
    …
```

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.

Improvements

language model $P(c)$: need more words. add -ed to verb or -s to noun, -ly for adverbs

bad probabilities: wrong word appears more frequently? (didn't happen)

error model $P(w|c)$: sometimes edit distance 2 is better
('adres' to 'address', not 'acres')
or wrong word of many at edit distance 1
(in addition better error model permits adding more obscure words)
allow edit distance 3?

best improvement:
look for context ('they where going', 'There's no there thear')
$\Rightarrow$ Use n-grams

(See Whitelaw et al. (2009), "Using the Web for Language Independent Spellchecking and Autocorrection": Precision, recall, F1, classification accuracy)