boggle

January 18, 2016

1 Common Boggle Words

The primary purpose of this notebook is to find the most common valid words appearing in Boggle boards. There are $16! \cdot 6^{16} \approx 5.9 \times 10^{25}$ different boards, so that we must recourse to sampling to estimate word probabilities.

Boggle is a game with a 4x4 board with slots for cubes, each of which has a letters on each of its 6 faces. Each game, the board is shuffled so that each cube is in a random slot, and each cube has a random face showing.

The goal of the game is to find as many chains of showing letters as possible on the resultant board. A player's score is based on the number of words that no other player found (larger unique words get higher scores). To be precise, a word chain must be a non-repeating sequence of horizontally, vertically, or diagonally contiguous cubes. Now, the same letter may appear twice in a path, but not the same cube. The words must belong to an agreed upon dictionary, and have some minimum length.

1.1 Simulating Boggle

Here are the 16 cubes one will find in a standard Boggle board:

```
In [1]: cubes = [('m', 'u', 'qu', 'h', 'n', 'i'),
                 ('n', 'g', 'e', 'e', 'a', 'a'),
                 ('d', 'r', 'y', 'v', 'l', 'e'),
                 ('t', 't', 'r', 'e', 'y', 'l'),
                 ('o', 'a', 'b', 'b', 'o', 'j'),
                 ('h', 'w', 'v', 'e', 'r', 't'),
                 ('s', 's', 'o', 'i', 't', 'e'),
                 ('s', 'h', 'a', 'p', 'c', 'o'),
                 ('g', 'e', 'w', 'n', 'e', 'h'),
                 ('u', 'm', 'c', 'o', 't', 'i'),
                 ('n', 'h', 'n', 'l', 'z', 'r'),
                 ('t', 'w', 'a', 'o', 't', 'o'),
                 ('p', 's', 'a', 'f', 'f', 'k'),
                 ('t', 's', 't', 'i', 'd', 'y'),
                 ('e', 'r', 'l', 'i', 'x', 'd'),
                 ('s', 'u', 'e', 'e', 'n', 'i')]
```

To simulate the process of shuffling a board, both the slot and face of each cube will be chosen uniformly.

```
In [2]: from random import choice, shuffle

        def rand_board():
            board = []
            shuffle(cubes)
            cube_order = iter(cubes)
            for i in range(4):
```

```
            board.append([])
            for j in range(4):
                cube = next(cube_order)
                board[-1].append(choice(cube))
        return board
```

```
In [3]: rand_board()
```

```
Out[3]: [['a', 'o', 's', 'g'],
         ['v', 'o', 'x', 'a'],
         ['e', 't', 'f', 'a'],
         ['e', 's', 'm', 'z']]
```

To find viable words on a board, we'll recursively traverse adjoining letters from each of the 16 cube slots. Once a sequence goes out of bounds or repeats a cube, traversal in that direction will stop.

Now, at each step in this process we could look up the sequence in a hash table (dictionary). But, it may be the case that the current sequence is no prefix of any dictionary word. So, it will be much more efficient to store the dictionary in a trie. Here is an inspiring elegant implementation.

```
In [4]: def make_trie(words):
            root = {}
            for word in words:
                node = root
                for letter in word:
                    node = node.setdefault(letter, {})
                node[None] = None    # signifies that node is a word
            return root
```

```
In [5]: make_trie(['broo', 'brooch'])
```

```
Out[5]: {'b': {'r': {'o': {'o': {None: None, 'c': {'h': {None: None}}}}}}}
```

Now we implement a recursive function that generates unique words on a given board, of at least some length.

```
In [6]: from itertools import product

        def get_words(board, trie, min_len=3):
            offsets = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]

            def recur(i, j, path, seen, node):
                if (i, j) in seen:    # path can't cross itself
                    return
                if not (0 <= i < 4) or not (0 <= j < 4):    # path can't wrap around board
                    return
                letter = board[i][j]
                path += letter
                seen.add((i, j))
                if letter in node:    # path is a prefix
                    if None in node[letter]:    # path is a word
                        yield path
                    for i_off, j_off in offsets:    # traverse neighbors even if path is word
                        yield from recur(i + i_off, j + j_off, path, seen.copy(), node[letter])

            seen = set()
            for i, j in product(range(4), range(4)):
```

```
            for word in recur(i, j, '', set(), trie):
                if len(word) >= min_len and word not in seen:
                    yield word
                    seen.add(word)
```

Finding words necessitates a dictionary, and a Scrabble-esque dictionary seems fitting. Conveniently, the analagous Words With Friends game's dictionary is publicly available: the Enhanced North American Benchmark Lexicon. So I'm going to use that.

```
In [7]: with open('enable1.txt') as f:
            trie = make_trie(word.strip() for word in f)
```

1.2 Estimating Word Probabilities

To estimate word frequencies, we'll sample relative frequencies from many random boards. *But how many boards to sample from?* We can view the appearance of a particular word in a random Boggle board as a Bernoulli random variable with unknown probability. We want to sample enough boards so that the ordering of the emperical k most common words is probably correct, or at the very least they're the same set of words.

Hoeffdings inequality tells us how many boards to construct a ϵ-sized confidence interval around a Benroulli variable: Let $X_1, \ldots, X_n \sim \text{Bernoulli}(p)$. Then, for any $\epsilon > 0$,

$$\mathbb{P}(\left|\overline{X_n} - p\right| > \epsilon) \leq 2e^{-2n\epsilon^2}$$

So that to achieve error ϵ with confidence α,

$$n \geq \frac{1}{2\epsilon^2} \log \frac{2}{\alpha}$$

boards will be sufficient.

Now, this doesn't guarantee that the top k words are in fact the top k, and in order, but it does ossify those top probabilities with a diligently chosen ϵ. Experimented with 1000 boards, showed that top 10 words' probabilities overlap, suggesting that we need $\epsilon < 0.001$ to be, say, $\alpha = 0.1$ confident that the ordering is correct. Unfortunately, this would lead to $n \geq 1.5 \times 10^6$ boards, which is too many hours of computation. Instead we'll shoot for just having the same set of top words, rather than correct ranking. The difference in probability between the empirical most common word and the 20th (we choose k arbitrarily) was about 0.01, yielding $n \geq 14979$ boards, which is more manageable.

```
In [8]: from collections import Counter
        from tabulate import tabulate

        NBOARDS = 14979
        counts = Counter()

        for _ in range(NBOARDS):
            board = rand_board()
            counts += Counter(get_words(board, trie, min_len=4))

        table = [(i, word, count/NBOARDS) for i, (word, count) in enumerate(counts.most_common(20), sta
        print(tabulate(table, headers=['Rank', 'Word', 'Frequency'], floatfmt='.4f'))
```

```
Rank  Word      Frequency
------  ------  -----------
     1  teen         0.0656
     2  note         0.0626
     3  tees         0.0617
     4  tent         0.0580
     5  toes         0.0575
```

```
 6  tone         0.0573
 7  teat         0.0561
 8  toea         0.0557
 9  rete         0.0531
10  tote         0.0525
11  tens         0.0522
12  test         0.0521
13  nets         0.0521
14  ates         0.0519
15  seta         0.0519
16  nett         0.0506
17  sent         0.0501
18  nest         0.0499
19  teas         0.0491
20  rote         0.0490
```

1.3 Common and Ordinary Word Ranking

The most common list is interesting, but some of the words are so ordinary that other players will probably know them and also find them. To improve one's chances of winning Boggle, it would be nice to have a ranking that takes into account both a word's probability of occurence, *and ordinariness*.

This is reminiscent tf-idf weighting! Our current list gives us the *term frequency*, so all that remains is to find each term's *inverse document frequency*. Here, we assume that usage corresponds to how likely it is for a word to be known.

We'll use word usage counts from American National Corpus. However, some of the words from Boggle might have various endings (e.g. "apple", "apples"), and those should add in frequency for "ordinariness." So we'll stem the words for lookup.

In [9]:
```
from nltk.stem import SnowballStemmer
from collections import defaultdict

snow = SnowballStemmer('english')
freqs = defaultdict(int)
min_freq = 1    # smallest frequency found, for use with Boggle words not in list

with open('ANC-token-count.txt', encoding='ISO-8859-1') as f:    # there's weird characters
    for word, *_, freq in map(str.split, f):
        if word == 'Total':    # there was a total count of words at the end
            continue
        word, freq = snow.stem(word), float(freq)
        freqs[word] += freq
        min_freq = freqs[word]
```

Now, we just need to calculate the scores of the words output a new sorted list.

In [10]:
```
from math import log

tfs, idfs = [], []
for word, count in counts.items():
    word = snow.stem(word)
    tfs.append(1 + log(count))
    idfs.append(log(1 + 1/freqs.get(word, min_freq)))

scores = sorted([(tf*idf, word) for tf, idf, word in zip(tfs, idfs, counts.keys())], reverse=T
table = [(i, word, score) for i, (score, word) in enumerate(scores[:20], start=1)]
print(tabulate(table, headers=['Rank', 'Word', 'Score']))
```

```
Rank  Word      Score
------  ------  -------
     1  toea    130.702
     2  seta    129.484
     3  nett    129.066
     4  teel    127.767
     5  sett    127.767
     6  stet    127.549
     7  hest    127.229
     8  haet    127.079
     9  tret    126.414
    10  thae    125.829
    11  hent    125.639
    12  eath    125.502
    13  tela    123.267
    14  erne    122.23
    15  tost    122.129
    16  teth    122.015
    17  seel    121.926
    18  sere    121.755
    19  rete    121.436
    20  olea    120.581
```

1.4 Epilogue

There are several interesting questions that still remain to answered:

- Someone gave a comment that the "best" Boggle boards are comprised of 40% vowels. Best, here, was intended to mean highest number of word solutions. What vowel/consonant proportions give the highest mean of number of words?
- What are the min, max, mean, and var number of words for Boggle boards?
- How many unique words can be created in the game Boggle, given a standard Scrabble-esque dictionary?
- Given the true probabilities of each word occurence, there exists a natural ranking of word frequencies. Precisely how many samples does it take to make sure the set of empirical top k words is the same? What about in the exact same order?

1.5 Best Proportion of Vowels

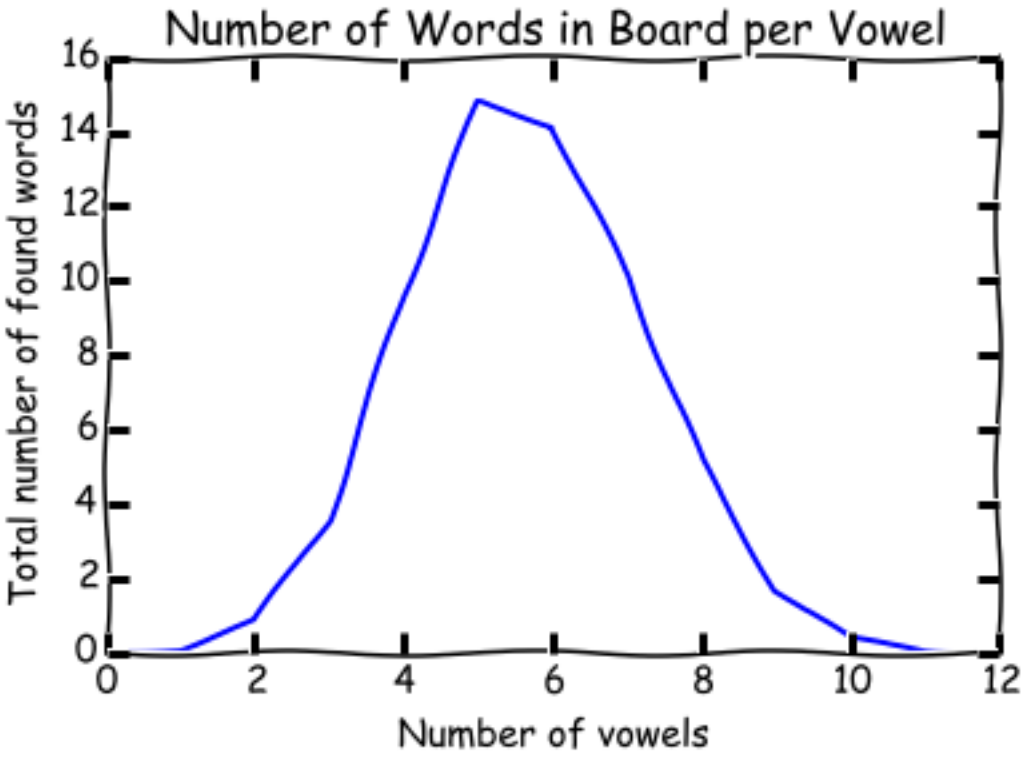
A friend gave the comment that the "best" Boggle boards are comprised of 40% vowels. Best, here, was intended to mean highest number of word solutions. Now to experimentally prove this claim. Unlike in previous sections, we will be lax proving the correct number of boards.

```
In [11]: from itertools import chain
         from collections import defaultdict

         NBOARDS = 5000
         data = defaultdict(int)
         vowels = set('aeiou')    # for simplicity we assume that these are the only vowels

         for _ in range(NBOARDS):
             num_vowels = 0
             board = rand_board()
             for letter in chain(*board):
                 if letter in vowels:
```

```
            num_vowels += 1
        num_words = len(list(get_words(board, trie, min_len=4)))
        data[num_vowels] += num_words
```

Now to plot the mean number of found words per number of vowels:

In [12]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.xkcd()

x, y = zip(*data.items())
plt.plot(x, [t/NBOARDS for t in y])
plt.title('Number of Words in Board per Vowel')
plt.xlabel('Number of vowels')
plt.ylabel('Total number of found words')
plt.show()
```



In [13]:
```
6/16
```

Out[13]:
```
0.375
```

Would you look at that, 40% was pretty spot on.