Approximate Event Occurance Counting Using Probabilistic Counters – Use Case of Text Files

Tomás Oliveira Costa <89016>, tomascosta@ua.pt

***Abstract* – This article's main objective is to give an approximate count of words in a book, using different probabilistic counters and an exact counter for measure comparison. On this work we focus on the half probability counter and the decreasing probability logarithmic counter.**

I. Introduction

The problem of the approximate counters is very relevant, since it provides a way to give close estimates to exact counters, but in a much more efficient way. There's several types of probabilistic counters, but for this work we focus on:

- ½ probability counter
- Logarithmic with descending probability

II. Description of the Problem

The goal is to read a book (text file) and count the occurrences for each distinct word. This can be done in one of three ways: the exact counter, the "half" counter and the "log" counter. These provide different values, but should all be approximate to the exact counter.

A. Counter with half probability

The "half" counter should have at each increment the probability of ½ to increment (a balanced coin throw), and at the end of the counting we will have about ½ of the counter increments of the exact counter. To estimate the number of words with this counter, we simply multiply counter increments by 2.

B. Counter with decreasing logarithmic probability

The "log" counter is cumulative, in the way that the probability of incrementing the counter keeps decreasing as the counter increments, therefore, several increments of the counter make it less likely to be incremented.

Afterwards we use the next formula to estimate the number of words:

```
(base^k - base + 1 ) / ( base-1 )
```

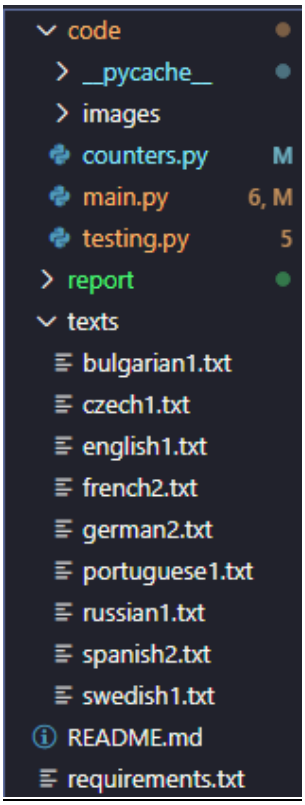
Given that the k is the counter value and our base for this work is the square root of 2 (√2)

III. Project Structure

For this project to have a standardized structure, the following folder were created:

- Code
- Report
- Texts

These folders represent the folders for the python scripts, the written reports and the example texts, respectively. Which leads to this following figure:



IV. Description of the Solution

Given the short length of the solution in lines of code (LOC), the project was created to be very modular and readable, therefore, the code is divided in a main script and a counters class.

Three counter classes were inherited from one Counter.class, which was contained the methods that were common between all of them.

This class is very simple in processing, and it has 2 main stages: Tokenizing and Indexing.

Tokenizing is the step of reading the text and generating tokens, which are the "words", already filtered that we will later add to our dictionary.

```
def tokenize(self):
    final_tokens = []
    with open(self.file_path) as file:
        for lines in file:
            tokens = re.sub("[^0-9a-zA-Z]+"," ",lines).lower().split(" ")
            final_tokens.extend(token for token in tokens if len(token)>3)
    self.tokens.extend(final_tokens)
```

After tokenizing, we can now index, and this is where each counter will vary.

A. *Indexing the exact counter*

The indexing for the exact counter is very trivial, for each token, we add it to the dictionary and increment the counter value by 1. These are the exact measures and are only used for comparison.

```
class ExactCounter(Counter):
    def index(self):
        # treatment just for counter
        for token in self.tokens:
            self.addToken(token)
            self.counter_value += 1

        # calculate word_value
        self.word_counter += len(self.tokens)
```

B. *Indexing the half counter*

As for the half counter, we do an interesting trick, which is to create an array of 0s or 1s with a 1/2 probability each and then we iterate the array and only had the token if the value was 1. To get the estimated word values, we only need to get the sum of all elements of the array and then multiply by 2 (our probability inversed).

```
class HalfCounter(Counter):
    def index(self):
        # treatment just for counter
        # other way
        random_array = [random.randint(0,1) for _ in range(len(self.tokens))]
        for i in range(len(random_array)):
            if random_array[i] == 1:
                self.addToken(self.tokens[i])
                self.counter_value += 1
```

C. *Indexing the log counter*

This counter is the most efficient, or the one that was providing the best results, since it doesn't really increment the counter by as much as the other two did, but still provides really good results when it comes to estimating words.

For this counter we take into account the probability of past increments and then decide based on a new calculated probability if we should increment of not.

```
class LogCounter(Counter):
    def index(self):
        # logarithmic base
        base = math.sqrt(2)
        # treatment just for counter
        for token in self.tokens:
            prob = 1/base**self.counter_value
            if random.random() < prob:
                self.counter_value += 1
                self.addToken(token)

        # this formula is provided in the slides as ( a^k-a + 1 ) / ( a -1 )
        k = self.counter_value
        self.word_counter = int((base**k - base + 1 ) / ( base-1 ))
```

V. TESTING TOOLS AND ANALYSIS

For a better understanding of the results, the program outputs the top ten most frequent words by the used counter and then shows some metrics such as the mean absolute error, the mean relative error as well as showing the estimated count of that word, compared to the exact couting. Below we show examples for 100 runs, which is a good measure, but for more accurate results we should have about 1000 to 10000 runs.

Besides the most frequent words, the program also outputs metrics relative to the global counter for the whole book.

Below is the example for the exact counter, and as it is expected, the errors are 0, since they are compared to the exact values.

```
Executing 100 trials on exact counter:

Ten most frequent words: (average)
Word            Exact Count     Avg. Est. Count         Mean absolute error     Mean relative error
------------------------------------------------------------------------------------------------------
that            665             665.0                   0.0                     0.00%
said            442             442.0                   0.0                     0.00%
with            385             385.0                   0.0                     0.00%
they            381             381.0                   0.0                     0.00%
little          241             241.0                   0.0                     0.00%
have            234             234.0                   0.0                     0.00%
mowgli          228             228.0                   0.0                     0.00%
when            213             213.0                   0.0                     0.00%
there           196             196.0                   0.0                     0.00%
down            193             193.0                   0.0                     0.00%

Global counter measures:
Maximum relative error: 0.000%
Minimum relative error: 0.000%
Mean relative error: 0.000%

Calculated in 6.05 seconds
```

Below is the counter with half probability and as we can see, after 100 runs, the program gives off really good estimates on the number of events, since the relative errors are very low (on average).

```
Executing 100 trials on half counter:

Ten most frequent words: (average)
Word            Exact Count     Avg. Est. Count         Mean absolute error     Mean relative error
------------------------------------------------------------------------------------------------------
that            665             665.52                  0.5                     0.08%
said            442             442.88                  0.9                     0.20%
with            385             381.2                   3.8                     0.99%
they            381             377.96                  3.0                     0.80%
little          241             241.94                  0.9                     0.39%
have            234             234.5                   0.5                     0.21%
mowgli          228             228.74                  0.7                     0.32%
when            213             214.34                  1.3                     0.63%
there           196             196.12                  0.1                     0.06%
down            193             194.02                  1.0                     0.53%

Global counter measures:
Maximum relative error: 1.952%
Minimum relative error: 0.014%
Mean relative error: 0.568%

Calculated in 9.24 seconds
```

As for the logarithmic counter, we notice that is more efficient, since it doesn't increment as much as the other ones. But, since the probability is constantly decreasing, its hard for the program to find the most common words, and it will instead find the most common among the first ones, since they are more probable to be incremented.

But, the program is still efficient and gives results that are not too bad for the global count of words in the book.

```
Executing 100 trials on log counter:

Ten most frequent words: (average)
Word            Exact Count     Avg. Est. Count         Mean absolute error     Mean relative error
------------------------------------------------------------------------------------------------------
jungle          145             3                       142.0                   97.93%
illustration            45              3                       42.0                    93.33%
book            3               3                       0.0                     0.00%
rudyard                 3               2                       1.0                     33.33%
little          241             2                       239.0                   99.17%
kipling                 3               2                       1.0                     33.33%
toomai          96              2                       94.0                    97.92%
down            193             2                       191.0                   98.96%
with            385             2                       383.0                   99.48%
page            3               2                       1.0                     33.33%

Global counter measures:
Maximum relative error: 177.705%
Minimum relative error: 1.818%
Mean relative error: 36.606%

Calculated in 5.56 seconds
```

For the evaluation of the increments of each counter, the user can also use the testing.py program and compare the averages of time and increments for each counter on a given number of trials, as follows:

```
Executing 1000 trials on all counters:

Exact:   0.02 seconds, with 7001.0 counter increments (average)
Half:    0.02 seconds, with 3500.76 counter increments (average)
Log:     0.02 seconds, with 22.688 counter increments (average)
```

(the following testing is done on the german2.txt book)

VI. Results and Discussion

After analyzing the results for a set of 1000 trials, what we could easily visualize is that the half counter performs very well and gives a really close estimate to the global number of words in the book as well as the individual counters for each word. But, for very large instances, we still increment the counter by a lot, and the dictionary growth is followed by it.

As for the logarithmic counter with decreasing probability, it gives a good estimate to the global number of words in the book, but is rather ineffective at individual counters, since the words that appear first on the book might not be the most common on the whole book, and since the probability keeps decreasing, words that appear further down the book are not likely to appear on the most frequent words.

References

[1] http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf

[2] http://www.gutenberg.org/