# A Phonetic Approach to Identify Linguistic Information in Strings

- Somdev Sangwan

## Abstract

This paper proposes an algorithm to calculate the amount of valid linguistic data in a given text. The approach is based on phonetics and how words are formed around the comfort of pronunciation. This paper discusses the approach in the context of the English language but it is applicable to any verbally expressible language.

## Contents



## Introduction

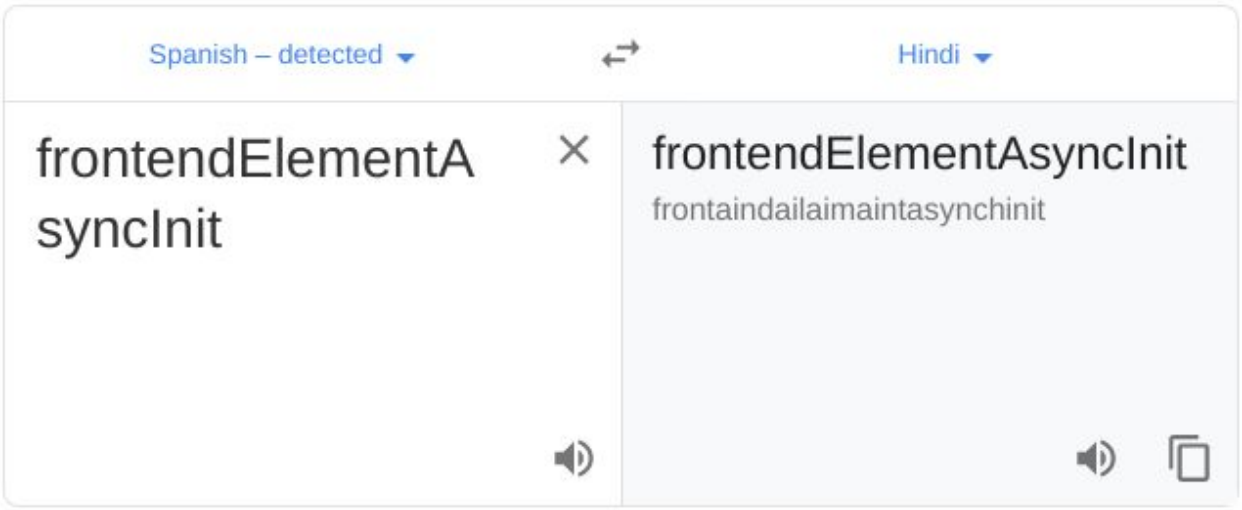
The recent evolution of computer science has made the decades old problem of differentiating between meaningful and gibberish information prevalent again. Computer generated pseudo-random strings are being widely used in various fields such obfuscation of data, cryptography, generating information for bots, authentication mechanisms etc. An optimal method to detect such strings is yet to be discovered.
This paper introduces an algorithm to identify linguistic information in linear time.

## Understanding the Problem

The aim is to write a computer program that can tell informational and gibberish text apart. Two obvious solutions for this problem are detecting valid words in the text or calculating its entropy. Both of these methods have their own advantages and disadvantages.
The common problems with approaches based on detection of valid words are

1. Inability to detect made up words e.g. `reddit`
2. Resource heavy computations to detect misspellings
3. Inability to process poorly structured text e.g. `frontendElementAsyncInit`

For example, Google Translate[1] is one of the most advanced language detection and translation programs but it fails to even detect the correct language of the string `frontendElementInit`.




Such errors can be reduced to an extent by using misspelling tolerant methods such as Levenshtein Distance[2] but such methods require significantly more resources which might not be favorable for a machine processing a large amount of text.

The other widely adopted solution is using Shannon Entropy to determine if a string is random enough to be considered as meaningless data.
Let us discuss the concept of entropy and determine if it is a good measure of *randomness*.

# Entropy

Information entropy is the average rate at which information is produced by a stochastic source of data.[3]

*In absolute layman's terms, "Every time something happens, its probability of it happening again increases. The less it happens, the more surprising it is i.e. the more information it generates. Entropy is the average of the amount of generated information by a source.*
*The more unique things happen, the more is the entropy and the more random the source is. The more same things happen, the less is the entropy and the more predictable the source is."*

The entropy of a text string is inversely proportional to the average of the frequencies of individual characters present in the string. Hence, the more unique characters are present in a string, the more entropy it has.

In my personal opinion, the common application of entropy as a measure of randomness is flawed because it is the average of information generated by events and has nothing to do with predictability of the events.

For example, let us consider two computer programs

1. **Program A:** Iterates over an array of printable Unicode characters and outputs them one at a time.
2. **Program B:** Monitors a sample of Polonium[4], a radioactive element. Radioactive elements emit particles randomly and it is considered to be the only truly practically unpredictable event.[5] Program B outputs either 0 or 1 respective to the emission of a particle from the atoms in a given time period.

**Program A** generates a new character every time. It yields more information, hence has more entropy. **Program B** only generates two characters either 0 or 1. It yields less information, hence has less entropy.
**Program A** is not random but has much higher entropy than **Program B** which is truly random. This establishes the fact that the number of possible outcomes impacts the value of entropy.

Discussing it within the scope of this paper, current solutions define a threshold value (usually 3.5) required for a string to be random. To test the implementation, let us calculate the entropy of the string `9033e0e305f247c0c3c80d0c7848c8b3`
Let us start by counting the occurrences of each character and dividing it by the total number of characters in the string

```
0 (occurs 6 times) -> 6/32 = 0.188
2 (occurs 1 time)  -> 1/32 = 0.031
3 (occurs 5 times) -> 5/32 = 0.156
4 (occurs 2 times) -> 2/32 = 0.063
5 (occurs 1 time)  -> 1/32 = 0.031
7 (occurs 2 times) -> 2/32 = 0.063
8 (occurs 4 times) -> 3/32 = 0.125
9 (occurs 1 time)  -> 1/32 = 0.031
b (occurs 1 time)  -> 1/32 = 0.031
c (occurs 5 times) -> 5/32 = 0.156
d (occurs 1 time)  -> 1/32 = 0.031
e (occurs 2 times) -> 2/32 = 0.063
f (occurs 1 time)  -> 1/32 = 0.031
```

The formula for Shannon Entropy is

$$\mathrm{H}(X) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log_b \mathrm{P}(x_i)$$

Putting the values into the formula, we get

```
H(X) = -[(0.188log20.188)+(0.031log20.031)+(0.156log20.156)+
(0.063log20.063)+ (0.031log20.031)+(0.063log20.063)+ (0.125log20.125)+
(0.031log20.031)+(0.031log20.031)+(0.156log20.156)+(0.031log20.031)+
(0.063log20.063)+(0.031log20.031)]
```

Solving further, we get

```
H(X) = -[(-0.453)+(-0.156)+(-0.418)+(-0.25)+(-0.156)+(-0.25)+(-0.375)+
(-0.156)+(-0.156)+(-0.418)+(-0.156)+(-0.25)+(-0.156)]

H(X) = -[-3.35222]

H(X) = 3.35222
```

The entropy of `9033e0e305f247c0c3c80d0c7848c8b3` is 3.35222. Similarly, calculating the entropy of `frontendElementAsyncInit` gives us 3.70499.

Ironically, as per the predefined threshold of 3.5 in most programs, `frontendElementAsyncInit` is random enough to be a chunk of random data but `9033e0e305f247c0c3c80d0c7848c8b3` is not.

This contradiction arises as `9033e0e305f247c0c3c80d0c7848c8b3` is a hexadecimal string and hence is limited to the character set `abcdef0123456789`. No matter how random the source of a hexadecimal string is, its entropy is limited by its character set.

Contrary to popular belief, Shannon Entropy is the average amount of information but it does not indicate whether the information has a meaning to it or how random its source is.

# Linguistic Approach to the Problem

The most noticeable difference between meaningful text and a sequence of random alphabets is the language. A meaningful text can only be written in a well-defined language and every

language has a grammar and its own set of rules. However, these rules are susceptible to errors when applied directly to a text in the wild because of the presence of imperfections.

The algorithm demonstrated in this paper solves the problem by assessing the *pronounceability* of a given text to determine whether it belongs to a verbally expressible language or not. It does so by breaking the text down to bigrams[6] and validating it using an array of valid bigrams from the language of interest.

Before we discuss the algorithm and its implementation, let us look at a sample output of a program using this algorithm

| | **Output** | | |
|---|---|---|---|
| **Input** | Total bigrams | Pronounceable bigrams | Unpronounceable bigrams |
| Pulp | 3 | 3 | 0 |
| Pelp | 3 | 3 | 0 |
| Pqlp | 3 | 0 | 3 |

Pulp is a valid word and it can be pronounced without any difficulties. Pelp is a made-up word but it is pronounceable, it is eligible to be incorporated into the English language. Pqlp is another made-up word but it's not pronounceable and hence has no place in English and can be considered as gibberish.

To elaborate on the working of the proposed algorithm, let us go through the complete implementation of it.

## Programmatic Implementation

Initially, an array containing two-character combinations of all English alphabet is created i.e.

```
[‘aa’, ‘ab’, ‘ac’, ‘ad’, . . . 'mo', 'mp', 'mq', 'mr', . . . 'zw', 'zx', 'zy', 'zz']
```

Then, a database containing all valid English words is loaded into the memory and each combination is searched in every word. The total frequencies of each combination in the database are then assigned to the respective variables in the array.

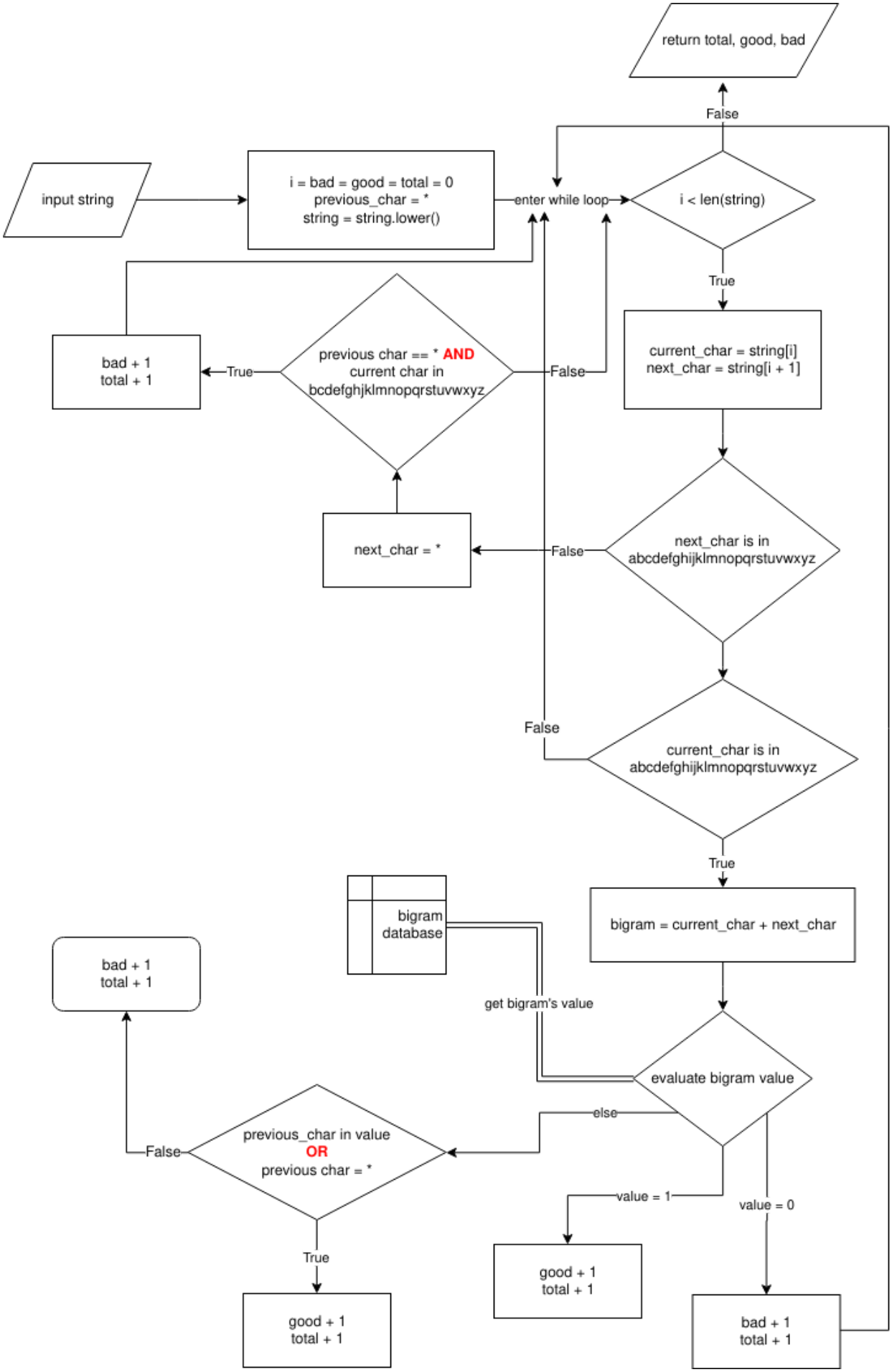
Combinations with 0 frequencies are assigned a value of 0. Combinations with frequencies larger than 1000 are assigned a value of 1. Finally, combinations with frequencies smaller than 1000 are compared against the database of valid words again to extract the characters preceding the combinations. This time, an array containing all the unique prefixes is assigned to each of the respective variables.

The following is an excerpt from the resultant array

```
...
"gk": 0,
"gl": 1,
"gm": [
"a”,
"y",
"o",
"d",
"e",
"i",
"n",
"u"
],
...
```

This array is ready to be stored locally and can be used by a computer program.

The algorithm is explained in detail below followed by source code of it’s implementation in Python programming language.


return total, good, bad
False
input string
i = bad = good = total = 0
previous_char = *
string = string.lower()
enter while loop
i < len(string)
True
bad + 1
total + 1
True
previous char == * AND
current char in
bcdefghjklmnopqrstuvwxyz
False
current_char = string[i]
next_char = string[i + 1]
next_char = *
False
next_char is in
abcdefghijklmnopqrstuvwxyz
False
current_char is in
abcdefghijklmnopqrstuvwxyz
True
bigram
database
bigram = current_char + next_char
bad + 1
total + 1
get bigram's value
evaluate bigram value
else
previous_char in value
OR
previous char = *
False
value = 1
value = 0
True
good + 1
total + 1
good + 1
total + 1
bad + 1
total + 1

# Python Implementation

```
def somdev(string, bigrams):
    i = bad = good = total = 0
    string = string.lower()
    previous_char = '*'
    string_length = len(string)
    alphas = 'abcdefghijklmnopqrstuvwxyz'
    while i < string_length - 1:
        current_char = string[i]
        next_char = string[i + 1]
        if next_char not in alphas:
            next_char = '*'
            if previous_char == '*' and current_char in 'bcdefghjklmnopqrstuvwxyz':
                bad += 1
            previous_char = current_char
            i += 1
            continue
        if current_char in alphas:
            bigram = current_char + next_char
            value = bigrams[bigram]
            if value == 0:
                bad += 1
            elif value == 1:
                good += 1
            else:
                if previous_char in value or previous_char == '*':
                    good += 1
                else:
                    bad += 1
            total += 1
        previous_char = current_char
        i += 1
    return total, good, bad
```

# References.


1. Google Translate https://translate.google.com
2. Levenshtein Distance https://en.wikipedia.org/wiki/Levenshtein_distance

3. Information Theory https://en.wikipedia.org/wiki/Entropy_(information_theory)
4. Polonium https://en.wikipedia.org/wiki/Polonium
5. Radioactive Decay https://en.wikipedia.org/wiki/Radioactive_decay
6. Bigram https://en.wikipedia.org/wiki/Bigram

## Explanation

This algorithm is based on the deduction that some character combinations are never used in a language because they are inconvenient to pronounce. The array discussed earlier is used as a reference to check if two adjacent characters can be pronounced. Two adjacent elements (alphabets in this case) make up a bigram.[6]

For instance, bigrams for the string `resin` are `re`, `es`, `si`, `in`. Let us now walk through the various steps of the algorithm

Once an input string is received