White Paper

Approximate string-matching Algorithm (Fuzzy Matching Algorithm)

July 24, 2018

Author: *Ravi Tiwari*
Email: *rtiwariops@gmail.com*
GitHub: *https://github.com/rtiwariops*

Purpose

The purpose of this paper is to dive deep into fuzzy matching algorithms. The fuzzy matching algorithms helps the credit unions and banks in scanning large datasets for money laundering and fraud detection. It also helps the healthcare industry for records matching to identify patients, match records and build and maintain their internal master patient indexes. In the data cleansing step of data curation process we quantify the data so you can build models.

Algorithms

- Soundex
- Levenshtein
- n-gram
- Jaro Winkler distance

Soundex

- Phonetic Algorithm
- Indexing words by sound as pronounced in the English language.
- Assign a Soundex coding value which you can use to perform calculations.
- Ideal for spelling inconsistencies for searching semantic searches, ideal for messy data in general.

American Soundex Coding

Every Soundex code consists of a letter and three numbers, such as W-252. The letter is always the first letter of the word. The numbers are assigned to the remaining letters of the word according to the Soundex guide shown below. Zeroes are added at the end if necessary to produce a four-character code. Additional letters are disregarded. Examples:

NUMBER	LETTER
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

Ignore: A, E, I, O, U, H, W, Y

A python library named **Jellyfish** for doing approximate and phonetic matching of strings. **Jellyfish** version >=0.7 supports Python 3.

```
import jellyfish as js
val1 = js.soundex('WASHINGTON')
print(val1)
val2 = js.soundex('WUSHINGTON')
print(val2)
val3 = js.soundex('SUM')
print(val3)
val4 = js.soundex('SOME')
print(val4)
val5 = js.soundex('STARE')
print(val5)
val6 = js.soundex('STAIR')
print(val6)
val7 = js.soundex('STEAL')
print(val7)
val8 = js.soundex('STEEL')
print(val8)
```

- **Washington** is coded W-252 (W, 2 for the S, 5 for the N, 2 for the G, remaining letters disregarded)
- **Sum** is coded S-500 (S, 5 for the M, remaining letters disregarded)
- **Stare** is coded S-360 (S, 3 for the T, 6 for the R, remaining letters disregarded)
- **Steal** is coded S-340 (S, 3 for the T, 4 for the L, remaining letters disregarded)

```
W252
W252
S500
S500
S360
S360
S340
S340
```

Soundex algorithm does not account for numbers and non-ascii characters as it is tailored towards the English language. Therefore, the comparison below returns an identical result.

```
val3 = js.soundex('SUM#123')
print(val3)
val4 = js.soundex('SUM')
print(val4)
```

```
S500
S500
```

Soundex algorithm returns different values in case when the two words have identical numeric values but starts with different letters but sounds alike.

```
val3 = js.soundex('accept')
print(val3)
val4 = js.soundex('except')
print(val4)
```

```
A213
E213
```

Pros:

- Very easy to implement.
- Fast computation.
- Great for realtime analysis.

Cons:

- Only works on ASCII characters (No foreign languages, symbols or numbers)
- Accuracy lower for most likely words.

Levenshtein distance

- Known as “edit distance” as well.
- Accounts for how many characters you have to change to have the same string.
- Fast computation.
- Great for realtime analysis.
- Pairwise comparison.

A python library named **Python-Levenshtein** for computing string distances and similarities. **Python-Levenshtein** version >=0.12 supports Python 3.

```
import Levenshtein as lv
val = lv.distance('SMYTHE', 'SMITH')
val1= lv.distance('WASHINGTON', 'WUSHINGTON')
val2 = lv.distance('STEAL', 'STEEL')
print(val)
print(val1)
print(val2)
```

Levenshtein algorithm returns the following three distance values for above three matches. The first word differs by two characters and the other two words differ by only one.

```
2
1
1
```

Levenshtein algorithm returns the value of one for the following words match even though the word looks similar but they are two different words and requires a single word change to match.

```
val = lv.distance('Marrón', 'Marron')
print(val)
```

```
1
```

Levenshtein algorithm returns the value of three and one when comparing the first distance which in real life is further than the second set of distances.

```
import Levenshtein as lv
val1 = lv.distance('99 ivoryridge', '100 ivoryridge')
val2 = lv.distance('99 ivoryridge', '999 ivoryridge')
print(val1)
print(val2)
```

```
3
1
```

Damerau Levenshtein

Damerau Levenshtein is similar to Levenshtein algorithm but it accounts for transposition of adjacent characters.

A python library named **pyxDamerauLevenshtein** for computing string distances and similarities. **pyxDamerauLevenshtein** version >=1.5 supports Python 3.

```
import pyxdameraulevenshtein as dlv
val1 = dlv.damerau_levenshtein_distance('recieve', 'receive')
val2 = lv.distance('recieve', 'receive')
print(val1)
print(val2)
```

Damerau Levenshtein algorithm will return the value of one whereas the regular Levenshtein algorithm will return the value of two. Damerau Levenshtein performs transposition swap of one character whereas Levenshtein algorithm is switching two characters.

```
1
2
```

Pros:

- Fast computation.
- Great for realtime analysis.
- Fast way to flag possible typos.

Cons:

- Only works when the two names are written in same script.

n-gram

- Takes into context on what is around the string.
- Grouping of letters.
- Proper unit of analysis (1-gram, 2-gram, 3-gram)

A python library named **ngram** for providing fuzzy search based on n-grams. **ngram** version >=3.3 supports Python 3.

```
import ngram as ng
val1 = ng.NGram.compare('Ham','Spam',N=1)
val2 = ng.NGram.compare('receved data','received date',N=3)
print(val1)
print(val2)
```

n-gram will return the value of 0.4 and 0.5 for both comparisons.

Pros:

- Easy to implement
- Great for auto-suggestions

Cons:

- Slow calculation (need to calculate n-gram for each string).

Jaro Winkler Distance

Jaro Winkler Distance which indicates the similarity score between two Strings. The Jaro measure is the weighted sum of percentage of matched characters from each file and transposed characters. Winkler increased this measure for matching initial characters.

```
from pyjarowinkler import distance as pyj
val = pyj.get_jaro_distance("WASHINGTON", "WUSHINGTON", winkler=True, scaling=0.1)
print(val)
```

Jaro Winkler Distance algorithm will return the value of 0.94 as there is change of a single character when comparing both words.

Pros:

- Works better than Levenshtein algorithm for shorter strings of text.
- Fast computation.
-

Cons:

- Harder to implement.

Libraries and modules:
The following libraries and modules are used during the demo for this presentation:

1. Jellyfish - Python
2. fuzzystrmatch – PostgreSQL
3. Python-Levenshtein – Python
4. pyxDamerauLevenshtein – Python
5. pg_trgm – PostgreSQL.
6. Pyjarowinkler - Python

References:
https://www.archives.gov/research/census/soundex.html
https://pypi.org/project/jellyfish/
https://www.postgresql.org/docs/9.1/fuzzystrmatch.html
https://pypi.org/project/python-Levenshtein/
https://pypi.org/project/ngram/
https://pypi.org/project/pyjarowinkler/

Summary

I would think that the choice of the distance is very much domain-dependent (i.e., it' s empirical) and you should likely measure effectiveness of several metrics for your specific data sets. Last, but not least, an **ensemble** would likely do better than any specific metric alone.