stringmetric

String metrics and phonetic algorithms for Scala.

API (Scaladoc)

Enhancements

View On GitHub

This project is maintained by Rocky Madden

Overview

The stringmetric library provides facilities to perform approximate string matching, measurement of string similarity/distance, indexing by word pronunciation, and sounds-like comparisons. In addition to the core library, each metric and algorithm has a command line interface. Heavy emphasis is placed on unit testing and performance (verified via microbenchmark suites).

Metrics and algorithms

- Dice / Sorensen (Similarity metric)
- Double Metaphone (Queued phonetic metric and algorithm)
- Hamming (Similarity metric)
- Jaccard (Similarity metric)
- Jaro (Similarity metric)
- Jaro-Winkler (Similarity metric)
- Levenshtein (Similarity metric)
- Metaphone (Phonetic metric and algorithm)
- Monge-Elkan (Queued similarity metric)
- Match Rating Approach (Queued phonetic metric and algorithm)

- Needleman-Wunch (Queued similarity metric)
- N-Gram (Similarity metric)
- NYSIIS (Phonetic metric and algorithm)
- Overlap (Similarity metric)
- Ratcliff-Obershelp (Similarity metric)
- Refined NYSIIS (Phonetic metric and algorithm)
- Refined Soundex (Phonetic metric and algorithm)
- Tanimoto (Queued similarity metric)
- Tversky (Queued similarity metric)
- Smith-Waterman (Queued similarity metric)
- Soundex (Phonetic metric and algorithm)
- **Weighted Levenshtein** (Similarity metric)

Depending upon

The project is available on the Maven Central Repository. Here is how you would add a dependency to the core sub-project in various build systems (add other sub-projects as needed):

Simple Build Tool:

```
libraryDependencies += "com.rockymadden.stringmetric" % "stringmetric-core" % "0.25.3"
```

Gradle:

```
compile 'com.rockymadden.stringmetric:stringmetric-core:0.25.3'
```

Maven:

```
<dependency>
    <groupId>com.rockymadden.stringmetric</groupId>
```

```
    <artifactId>stringmetric-core</artifactId>
    <version>0.25.3</version>
</dependency>
```

Similarity package

Useful for approximate string matching and measurement of string distance. Most metrics calculate the similarity of two strings as a double with a value between 0 and 1. A value of 0 being completely different and a value of 1 being completely similar.

Dice / Sorensen Metric: *(Note you must specify the size of the n-gram you wish to use. This can be done implicitly.)*

```
println(DiceSorensenMetric.compare("night", "nacht")(1))
println(DiceSorensenMetric.compare("context", "contact")(1))
```

Output:

```
0.6
0.7142857142857143
```

Hamming Metric:

```
println(HammingMetric.compare("toned", "roses"))
println(HammingMetric.compare("1011101", "1001001"))
```

Output: *(Note the exception of integers, rather than doubles, being returned.)*

```
3
2
```

Jaccard Metric: *(Note you must specify the size of the n-gram you wish to use.*

This can be done implicitly.)

```
println(JaccardMetric.compare("night", "nacht")(1))
println(JaccardMetric.compare("context", "contact")(1))
```

Output:

```
0.3
0.35714285714285715
```

Jaro Metric:

```
println(JaroMetric.compare("dwayne", "duane"))
println(JaroMetric.compare("jones", "johnson"))
println(JaroMetric.compare("fvie", "ten"))
```

Output:

```
0.8222222222222223
0.7904761904761904
0
```

Jaro-Winkler Metric:

```
println(JaroWinklerMetric.compare("dwayne", "duane"))
println(JaroWinklerMetric.compare("jones", "johnson"))
println(JaroWinklerMetric.compare("fvie", "ten"))
```

Output:

```
0.8400000000000001
```

```
0.8323809523809523
0
```

Levenshtein Metric:

```
println(LevenshteinMetric.compare("sitting", "kitten"))
println(LevenshteinMetric.compare("cake", "drake"))
```

Output: *(Note the exception of integers, rather than doubles, being returned.)*

```
3
2
```

N-Gram Metric: *(Note you must specify the size of the n-gram you wish to use. This can be done implicitly.)*

```
println(NGramMetric.compare("night", "nacht")(1))
println(NGramMetric.compare("night", "nacht")(2))
println(NGramMetric.compare("context", "contact")(2))
```

Output:

```
0.6
0.25
0.5
```

Overlap Metric: *(Note you must specify the size of the n-gram you wish to use. This can be done implicitly.)*

```
println(OverlapMetric.compare("night", "nacht")(1))
```

```
println(OverlapMetric.compare("context", "contact")(1))
```

Output:

```
0.6
0.7142857142857143
```

Ratcliff/Obershelp Metric:

```
println(RatcliffObershelpMetric.compare("aleksander", "alexandre"))
println(RatcliffObershelpMetric.compare("pennsylvania", "pencilvaneya"))
```

Output:

```
0.7368421052631579
0.6666666666666666
```

Weighted Levenshtein Metric: *(Note you must specify the weight of each operation. Delete, insert, and then substitute. This can be done implicitly.)*

```
println(WeightedLevenshteinMetric.compare("book", "back")(10, 0.1, 1))
println(WeightedLevenshteinMetric.compare("hosp", "hospital")(10, 0.1, 1)
)
println(WeightedLevenshteinMetric.compare("hospital", "hosp")(10, 0.1, 1)
)
```

Output: *(Note that while a double is returned, it can be outside the range of 0 to 1, based upon the weights used.)*

```
2
0.4
```

```
40
```

Phonetic package

Useful for indexing by word pronunciation and performing sounds-like comparisons. All metrics return a boolean value indicating if the two strings sound the same, per the algorithm used. All metrics have an algorithm counterpart which provide the means to perform indexing by word pronunciation.

Metaphone Metric:

```
println(MetaphoneMetric.compare("merci", "mercy"))
println(MetaphoneMetric.compare("dumb", "gum"))
```

Output:

```
true
false
```

Metaphone Algorithm:

```
println(MetaphoneAlgorithm.compute("dumb"))
println(MetaphoneAlgorithm.compute("knuth"))
```

Output:

```
tm
n0
```

NYSIIS Metric:

```
println(NysiisMetric.compare("ham", "hum"))
```

```
println(NysiisMetric.compare("dumb", "gum"))
```

Output:

```
true
false
```

NYSIIS Algorithm:

```
println(NysiisAlgorithm.compute("macintosh"))
println(NysiisAlgorithm.compute("knuth"))
```

Output:

```
mcant
nnat
```

Refined NYSIIS Metric:

```
println(RefinedNysiisMetric.compare("ham", "hum"))
println(RefinedNysiisMetric.compare("dumb", "gum"))
```

Output:

```
true
false
```

Refined NYSIIS Algorithm:

```
println(RefinedNysiisAlgorithm.compute("macintosh"))
```

```
println(RefinedNysiisAlgorithm.compute("westerlund"))
```

Output:

```
mcantas
wastarlad
```

Refined Soundex Metric:

```
println(RefinedSoundexMetric.compare("robert", "rupert"))
println(RefinedSoundexMetric.compare("robert", "rubin"))
```

Output:

```
true
false
```

Refined Soundex Algorithm:

```
println(RefinedSoundexAlgorithm.compute("hairs"))
println(RefinedSoundexAlgorithm.compute("lambert"))
```

Output:

```
h093
l7081096
```

Soundex Metric:

```
println(SoundexMetric.compare("robert", "rupert"))
```

```
println(SoundexMetric.compare("robert", "rubin"))
```

Output:

```
true
false
```

Soundex Algorithm:

```
println(SoundexAlgorithm.compute("rupert"))
println(SoundexAlgorithm.compute("lukasiewicz"))
```

Output:

```
r163
l222
```

Decorating

It is possible to decorate algorithms and metrics with additional functionality. The most common decorations are filters, which are useful for filtering strings prior to evaluation (e.g. ignore case, ignore non-alpha, ignore spaces). **NOTE:** Memoization decorator queued.

Basic examples with no filtering:

```
JaroWinklerMetric.compare("string1", "string2")
JaroWinklerMetric().compare("string1", "string2")
(new JaroWinklerMetric).compare("string1", "string2")
```

Basic example with single filter:

```
(new JaroWinklerMetric with IgnoreAsciiLetterCaseFilter).compare("stri
ng1", "string2")
```

Basic example with stacked filters. Filters are applied in reverse order:

```
(new JaroWinklerMetric with IgnoreAsciiLetterCaseFilter with AsciiLet
terOnlyFilter).compare("string1", "string2")
```

Convenience objects

Convenience objects are available to make interactions with the library easier.

StringAlgorithm:

```
// Easy access to compute methods.
StringAlgorithm.computeWithMetaphone("string")

// Easy access to types and companion objects.
val metaphone: StringAlgorithm.Metaphone = StringAlgorithm.Metaphone()
```

StringMetric:

```
// Easy access to compare methods.
StringMetric.compareWithJaroWinkler("string1", "string2")

// Easy access to types and companion objects.
val jaro: StringMetric.Jaro = StringMetric.Jaro()
```

StringFilter:

```
// Easy access to types and instances.
```

```
val metaphone: StringAlgorithm.Metaphone = StringAlgorithm.Metaphone()
 with StringFilter.asciiSpace
val asciiSpace: StringFilter.AsciiSpace = StringFilter.asciiSpace
```

Building the CLIs

When built, the CLI sub-project creates an individual shell command for each algorithm and metric. The sub-project is built via a Gradle task:

```
git clone https://github.com/rockymadden/stringmetric.git
cd stringmetric
gradle :stringmetric-cli:tar
```

Running the `tar` task will create a compressed archive **and** an unarchived copy of the built files. The files can be found under the `build` directory that Gradle creates. The archive is named `stringmetric-cli.tar.gz` and the unarchived files can be found in the directory named `stringmetric-cli`.

To run a command from the current directory that you would be in from doing the above:

```
./cli/build/stringmetric-cli/jaroMetric abc xyz
```

Lastly, you may need to chmod the files because of the inability for Gradle to do so reliably.

Using the CLIs

Every metric and algorithm has a command line interface. Said code is housed in a separate sub-project called `stringmetric-cli`.

The help option prints command syntax and usage:

```
$ metaphonemetric --help
```

```
Compares two strings to determine if they are phonetically similarly, per the
Metaphone algorithm.

Syntax:
  metaphonemetric [Options] string1 string2...

Options:
  -h, --help
    Outputs description, syntax, and options.
```

```
$ jarowinklermetric --help
Compares two strings to calculate the Jaro-Winkler distance.

Syntax:
  jarowinklermetric [Options] string1 string2...

Options:
  -h, --help
    Outputs description, syntax, and options.
```

Compare "dog" to "dawg":

```
$ metaphonemetric dog dawg
true
```

```
$ jarowinklermetric dog dawg
0.75
```

Get the phonetic representation of "dog" using the Metaphone phonetic algorithm:

```
$ metaphonealgorithm dog
```

```
tk
```

API

Scaladoc is available on the project website.

Requirements

- Scala 2.10.x
- Gradle 1.x

Versioning

Semantic Versioning v2.0

License

Apache License v2.0