Developer Manual
for py-stringmatching 0.2.x

The Magellan Development Team

July 10, 2016

Package's homepage: https://sites.google.com/site/anhaidgroup/projects/py_stringmatching.

1 Introduction

py_stringmatching is a Python package that consists of a variety of string tokenizers (e.g., whitespace tokenizer, qgram tokenizer) and string similarity measures (e.g., edit distance, Jaccard measure, TF/IDF) [1]. The goal is to build a comprehensive and scalable set of string tokenizers and similarity measures for the Python data management eco-system.

This document discusses the motivation for the package, the implemented tokenizers and similarity measures, the design decisions, current known limitations, and future work. The document should be useful to any developer who plans to extend the package. People who just want to use the package should consult the User Manual (available from the package's homepage).

2 What is New?

Compared to Version 0.1.0, the following are new:

- Qgram tokenizers have been modified to take a flag called "padding". If this flag is True (the default), then a prefix and a suffix will be added to the input string before tokenizing (see Section 7 for a reason for this).
- Version 0.1.0 does not handle strings in unicode correctly. Specifically, if an input string contains non-ascii characters, a string similarity measure may interpret the string incorrectly and thus compute an incorrect similarity score. In this version we have fixed the string similarity measures. Specifically, we convert the input strings into unicode before computing similarity measures. NOTE: the tokenizers are still not yet unicode-aware.
- In Version 0.1.0, the flag "dampen" for TF/IDF similarity measure has the default value of False. In this version we have modified it to have the default value of True, which is the more common value for this flag in practice.
- We have clarified the missing value policy used by this package (see Section 9).

Compatibility: As described above, this version is not compatible with Version 0.1.0. Specifically, if you have used qgram tokenizers or TF/IDF measure in your code, using Version 0.1.0, then the code will not run with Version 0.2.0.

3 Motivation and Goals

In recent years a lot more data has been generated in many more types of applications, and increasingly more people want to work with such data. This has led to the Big Data and data science phenomena, and has motivated the development of many data management eco-systems.

One popular eco-system is the Python data management eco-system, also often referred to as the Python data science and Big Data stacks. These stacks consist of numerous popular packages such as scipy, numpy, pandas, scikit-learn, etc. together with packages to run Python programs on Hadoop or Spark. It is very likely that in the coming years, more and more data management tools, captured in packages, will be contributed to this Python eco-system, and more and more data scientists/analysts will use tools in this eco-system to manage their data applications.

Many such applications require string similarity capabilities, and yet today there are few packages in the the Python data eco-system that provide such capabilities. The package py_stringmatching (together with the sister package py_stringsimjoin) aims to fill this need.

Current Work: Tokenizing and string similarity measures have been studied extensively. For new developers, the string matching chapter of the book "Principles of Data Integration" (available from the py_stringmatching package homepage) provides perhaps the easiest introduction to string similarity measures. See also [2] for a brief but informative discussion of tokenizing text.

In contrast to the wealth of research work, there has been relatively few implemented tokenizing and string similarity measure packages. As of June 2016, we counted 2 non-Python packages and 7 Python packages for string matching (see Appendix A for a detailed description). The main limitations of these packages are:

- *Language:* The two non-Python packages are in Java, and thus are hard to use in Python. To use them, users would need to install and run JVM (Java Virtual Machine), a cumbersome process. (If a package is in C, then it is a bit easier to incorporate into Python, due to the fact that the Python compiler was written in C. For example, the package python-Levenshtein was originally written in C. Later a wrap-around was added so that it can behave as a Python package.)

- *Coverage:* The packages do not provide a comprehensive coverage of the most common string similarity measures, and it is not clear if they have plans to provide comprehensive coverage. For example, 4 out of 7 Python packages cover only edit distance variations, while the remaining 3 packages do not implement certain common similarity measures (e.g., TF/IDF, Jaccard).

- *Runtime Performance:* The performance of certain measures is unsatisfactory. For example, our experiments show that the runtime of edit distance can vary by as much as 180 times across the packages. In particular, it appears that it is difficult to obtain satisfactory performance for edit distance (say) using just Python (C and Cython appear to provide a far better performance).

- *Installation:* Some of the packages are cumbersome to use. For example, to use Abydos, the user needs to install the complete package, of which the module on similarity measure constitutes just a small part.

- *Licensing:* Some of the packages use restrictive copyright licenses. For example, python-Levenshtein (which has the best runtime performance for edit distance in our experiments) uses GPL. Roughly speaking, any code using this code would also become "GPL" open-source code. In contrast, we want to allow any code to use py_stringmatching with acknowledgment. For these reasons, we plan to use BSD 3-Clause license (which is also used by well-known packages such as pandas and sklearn).

- *Extensibility:* It is not clear if the existing packages are designed for extension, and if so, how to extend them. In contrast, we envision that py_stringmatching will have to be extended to handle more tokenizers and similarity measures, and to be adapted to various domains. So we plan to design py_stringmatching to be extensible from the scratch. For example, we design tokenizers and string similarity measures as classes as opposed to functions (see more below).

Goals: As a result, we want to develop a new Python package for string tokenizers and similarity measures. Our goals are as follows:

- Develop a comprehensive package. It should at least cover all common tokenizers and similarity measures, and then expand over time. Ideally it should subsume current packages.
- The package should be in Python with minimal dependencies, so that many other Python packages can use it easily. For example, if some of our code is in Java, then we may have to require the user to install and start the Java Virtual Machine (JVM) before using the package, a cumbersome process that may not work on certain machines.
- The tokenizers and similarity measures should be as fast as possible.
- The licensing should allow liberal usage (with acknowledgment) yet shield us from legal responsibilities.
- The package should be designed such that it can be easily extended.

4 Currently Implemented Tokenizers and Similarity Measures

Currently we have implemented five tokenizer classes (organized into a class taxonomy): alphabetic-, alphanumeric-, delimiter-, qgram-, and whitespace tokenizers. We have also implemented 14 similarity measures, organized into four groups:

- *Sequence-based Measures:* affine gap, Hamming distance, Jaro, Jaro Winkler, Levenshtein, Needleman Wunsch, Smith Waterman.
- *Set-based Measures:* Cosine, Dice, Jaccard, Overlap.
- *Bag-based Measures:* TF/IDF.
- *Hybrid Measures:* Monge Elkan, Soft TF/IDF.

We plan to add more tokenizers and similarity measures to this package over time. A list of far more similarity measures that can be added is enclosed in Appendix B.

5 Requirements and Dependencies

py_stringmatching requires

- Python 2.7 or Python 3.3+. We plan to continue to support both (as both appear likely to still be used in the near future).
- numpy 1.7.0 or higher. We use numpy to perform matrix/array computation for certain similarity measures (such as edit distance related ones, which have to fill in a dynamic programming matrix).

- six. We use six to write compatible code between Python 2 and Python 3.

In addition, when developing py_stringmatching, developers will need access to Cython (used to scale up certain similarity measures). Users of py_stringmatching do not need access to Cython. But they need access to a C or C++ compiler to compile the part of py_stringmatching that is in C code.

Platform Testing: As of June 7 2016, py_stringmatching Version 0.1.0 has been tested on Linux (Ubuntu with Kernel Version 3.13.0-40-generic), OS X (Darwin with Kernel Version 13.4.0), and Windows 8.1. Subsequent versions will be tested on similar platforms (see the package's homepage for details).

We used continuous integration tools in GitHub to test the package on different operating systems. Specifically, we used Travis-CI to test on Linux and OS X, and Appveyor to test on Windows.

6 Usage Scenarios

There are two main usage scenarios: (1) tokenizing and then computing similarity scores for just one or several string pairs, and (2) doing the same for a very large number of string pairs, such as those in the Cartesian product of two large tables A and B of strings.

Scenario 1 is relatively straightforward. Scenario 2, in contrast, may raise significant runtime challenges. For example, suppose we want to match two tables A and B, each having 10K strings. This produces 100M string pairs. Suppose for each string pair (x, y) we start by tokenizing x and y into t_x and t_y, respectively, then compute the similarity score between them, using a similarity function $foo(t_x, t_y)$. Then there are at least two major runtime challenges:

1. For each string x in table A, we will tokenize it 10K times, once for each string y in table B. This is clearly very inefficient. We should have to tokenize each string in each table only once. This tokenization output should be "cached" somewhere and re-used later.

2. The similarity function foo may need to do some pre-processing. For example, if foo is TF/IDF, then it may have to process a corpus and compute TF/IDF statistics for the terms in the corpus. If foo needs to do this pre-processing once for each pair (x, y), then it would be very inefficient.

An important question is: where should we address these challenges? In the package py_stringmatching or in packages that use py_stringmatching. For now, there is no obvious place to cache the tokenization output in py_stringmatching, so we delay addressing Challenge 1 to whichever package using py_stringmatching. (For example, the package py-stringsimjoin that implements string matching between two tables would need to address this challenge.)

We address Challenge 2 by designing similarity functions to be Python objects. We then call pre-processing to "initialize" these objects, before using them repeatedly to compute similarity scores for multiple string pairs (see more below.) This way, pre-processing is done only once, when we initialize the similarity measure object.

7 Challenges in Building Tokenizers and Design Decisions

Building good string tokenizers is highly non-trivial, as we discuss below (see also [2] for a discussion of tokenizing).

Types of Tokenizers: There are two types and we have implemented both. The first type defines what a token is (e.g., alphabetic, alphanumeric, qgram). For instance, an alphabetic tokenizer defines a token to be a maximal sequence s of consecutive alphabetic characters inside the input string t (here "maximal" means there is no other token of t that subsumes s).

The second type of tokenizers does not define the tokens. Instead, it defines the delimiters. For example, a whitespace tokenizer uses a set of delimiters such as whitespace, tab, newline character, etc. to chop an input string into tokens.

Types of Tokenizer Output: We want the output to be a bag or a set. The default is to return a bag of tokens, but in some cases we may want to return sets. For example, a Jaccard measure takes as input two sets of tokens. There should be a flag that the user can use to indicate which type of output he/she wants.

One can argue that a tokenizer can just output a bag of tokens. Then applications calling a tokenizer can process it into a set if they want. This is suboptimal because an application tends to call a tokenizer many times. Consider for example matching a string x in Table A with 10K strings in Table B. Suppose we tokenize x into a bag of tokens, tokenize each string in B into a bag of tokens, and then call a similarity measure (e.g., Jaccard) for each of the 10K string pairs. Then the conversion of bags to sets would happen 10K times for string x. This is clearly very inefficient.

Thus, we argue that the conversion from bags to sets should be pushed inside the tokenizer. The developer can then focus on making this conversion efficient.

Should Tokenizers be Functions or Class Objects? The simplest solution is to design tokenizers as functions. For example, we can design the alphabetic tokenizer as a function f that takes a string s and returns a bag/set of alphabetic tokens. There are several problems with this approach, which lead us to adopt an object-oriented approach in which each tokenizer is a class object.

- A tokenizer may need to do some pre-processing work. If we design it to be a function, then it would need to do this work every time we apply it to a string. This is highly inefficient.

  For example, consider a tokenizer that has a list L of strings that should not be tokens (such as "a", "the", "and", etc.). Such a list is commonly called a token blacklist. To tokenize a string s, we can tokenize s in the usual way, then drop all tokens that appear in the list L.

  Here a pre-processing work may be converting list L into a hash (to facilitate fast checking). If the tokenizer is designed as a function, then hash building will need to be done every time we apply the tokenizer to a string, a clear waste of effort.

  Instead, if each tokenizer is a class object, then we can do pre-processing just once, when initializing the object and before applying to many strings.

- During operation, an application may need to query a tokenizer to obtain its parameter (e.g., if the tokenizer is a qgram tokenizer, then the application may want to know the value q). If the tokenizer is a function, it would be difficult to store and query such values. If the tokenizer is a class object, we can store these values in data fields.

- There are usually a set of operations that are all related to a single tokenizer, such as tokenizing, querying for tokenizer's parameters, etc. We would like to group them somehow, which we cannot do if they are all just disparate functions. Using class objects, we can group these operations as member functions.

- In the future we will often need to extend and customize the tokenizers. If we design them as classes, it will facilitate this need.

Current Design for Tokenizers: Currently we have the following tokenizer class hierarchy:

```
Tokenizer -- Definition Tokenizer -- Alphabetic Tokenizer
                                  -- Alphanumeric Tokenizer
                                  -- Qgram Tokenizer
          -- Delimiter Tokenizer  -- Whitespace Tokenizer
```

Here, all tokenizers that define what it means to be a token fall under "Definition Tokenizer". Those that define the delimiters instead fall under "Delimiter Tokenizer". At construction time

- We can set the parameters of a tokenizer. All tokenizers have a parameter called "return_set". If this is True, the tokenizer returns a set of tokens, otherwise it returns a bag (the default).
- Note that all parameters supplied to the tokenizer at the construction time are stored in the data fields of the tokenizer. There are methods to allow users to set and get the values of these fields.
- The tokenizer also does all pre-processing work, if any (e.g., building a hash out of the list of stop words).

After the construction, we can call the "tokenize" method to tokenize an input string.

Adding Prefix and Suffix to a String for Qgram Tokenizers: Consider computing a similarity score between two strings "mo" and "moo", using 3gram tokenizing and Jaccard similarity measure. Tokenizing "mo" produces an empty set, since this string contains no 3gram. Tokenizing "moo" produces the set {"moo"}. Thus, the Jaccard score between these two strings will be 0.

Intuitively this is unsatisfying. Strings "mo" and "moo" are fairly similar, and so one would expect that similarity scores between them exceed zero.

To address such cases, we provide a flag to qgram tokenizers. If this flag is set to True (which is the default), then any input string will be padded with a prefix of $(q-1)$ characters '#' and a suffix of $(q-1)$ characters '$', before tokenizing.

For example, given the input string "mo", a 3gram tokenizer will first add a prefix and a suffix to obtain "##mo$$", then tokenize this new string into the set {"##m", "#mo", "mo$", "o$$"}. Note that here the token "##m" can be viewed as capturing the fact that the input string starts with character 'm'. Similarly, the token "o$$" captures the fact that the input string ends with character 'o'.

Note also that given an empty input string, the result of tokenizing even with padding is still an empty set, which corresponds to our expectation.

8 Challenges in Building Similarity Measures and Design Decisions

Types of Similarity Measures: We distinguish six types of similarity measures: sequence-based, set-based, bag-based, hybrid, phonetic, and others. The first three types treat input strings as sequences of characters, sets of tokens, and bags of tokens, respectively. The fourth type, hybrid, typically uses a primary similarity measure that in turn uses a secondary similarity measure. The fifth type, phonetic, computes a similarity score between two strings based on how similar they sound. See [1] for more information.

Types of Similarity Measure Output: There are two main types of similarity measure output:

- Given two input strings, compute a true similarity score, which is a number in the range [0,1] such that the higher this number, the more similar the two input strings are.
- Given two input strings, compute a distance score, which is a number such that the higher this number, the more *dissimilar* the two input strings are (this number is often not in the range [0,1]). Clearly, Type-2 measures (also known as distance measures), are the reverse of Type-1 measures.

For example, Jaccard similarity measure will compute a true similarity score in [0,1] for two input strings. Levenshtein similarity measure, on the other hand, is really a distance measure, which computes the edit distance between the two input strings. It is easy to convert a distance score into a true similarity score.

Given the above, each similarity measure object in the package py_stringmatching is supplied with two methods: get_raw_score and get_sim_score. The first method will compute the raw score as defined by that type of similarity measures, be it similarity score or distance score. For example, for Jaccard this method will return a true similarity score, whereas for Levenshtein it will return an edit distance score.

The method get_sim_score normalizes the raw score to obtain a true similarity score (a number in [0,1], such that the higher this number the more similar the two strings are). For Jaccard, get_sim_score will simply call get_raw_score. For Levenshtein, however, get_sim_score will normalize the edit distance to return a true similarity score in [0,1].

Should Similarity Measures Be Functions or Class Objects? The reasoning here would be very similar to that in the case of tokenizers. So we will design similarity measures as class objects.

Current Design for Similarity Measures: Currently we have designed the following class hierarchy for similarity measures:

```
SimilarityMeasure
     -- SequenceSimilarityMeasure -- Affine, HammingDistance,
                                     Jaro, JaroWinkler, Levenshtein,
                                     NeedlemanWunsch, SmithWaterman
     -- TokenSimilarityMeasure -- Cosine, Dice, Jaccard,
                                  OverlapCoefficient, TfIdf
     -- HybridSimilarityMeasure -- MongeElkan, SoftTfIdf
```

Here "TokenSimilarityMeasure" refers to set- and bag-based similarity measures, as defined earlier.

Again, similar to the case of tokenizing, when creating a similarity measure object, we will pre-process: all the work that must be done once will be done during the object creation and initialization, not repeating when we apply the measure to each string pair. Put differently, when we apply a similarity object to a string pair, we should do only work that is absolutely required to be done for that pair.

Once created (and finished with pre-processing), we can call get_raw_score to obtain a raw numeric score between two input strings (e.g., the edit distance, or the Jaccard score). For token-based and hybrid measures, the input for get_raw_score will be two sets or two bags (henceforth will be referred to as two lists) which are the output of applying a tokenizer to two strings. For sequence-based measures, the input for get_raw_score will be two strings.

For similarity measures whose scores can be normalized, the user can call get_sim_score instead of calling get_raw_score to obtain a normalized sim score between 0 and 1.

Currently, we do not support get_sim_score method for the following five measures : Affine, Monge Elkan, Needleman Wunsch, Smith Waterman and Soft TF/IDF. The first four measures take a secondary similarity measure as input. Thus, to normalize these measures, we need to know the range of the secondary similarity measures. Since we accept any arbitrary function as a secondary similarity measure, we do not have an easy way to determine this range. A possible future solution is to ask the user to provide the range of the secondary similarity measure.

Regarding the last measure, Soft TF/IDF, given two input lists A and B, the similarity score can go above 1, because a particular token from B can be matched with multiple tokens from A. A possible solution is to restrict that a token from B match with only one token from A, thereby yielding only similarity scores in $[0, 1]$.

Scaling: In Usage Scenario 2 (see Section 6), a similarity measure will be repeatedly called for numerous string pairs. Thus, it is critical that we implement these measures very efficiently.

In general, we have found that plain-vanilla Python implementations can be quite slow. To scale them up, a good solution is to use Cython. Briefly, Cython is a superset of Python. The developer writes the code (to be scaled up) in Cython, then it will be compiled into C code, then binary code. The Cython code still needs to be written in a way that makes the resulting C code efficient.

We have used the above strategy to scale up Levenshtein similarity measure. Our experiments show that the new code is about 200 times faster than the original Python code, runs in roughly the same time as that of the editdistance package (a Python package also using Cython, see Appendix A), and only about 3 times slower than that of the python-Levenshtein package, which was originally written in C.

Handling Corner Cases of Dividing by Zero: For certain string similarity measures (e.g., Jaccard), if we apply the scoring formula (e.g., $|X \cap Y|/|X \cup Y|$) in the case of two empty sets (or one empty set), we may have 0/0, which is undefined.

The typical way to handle such cases is to define what similarity scores we want in such cases. For all other cases then use the scoring formula. For example, for Jaccard measure, we will define the similarity score to be 1 given two empty sets, and to be computed using the formula $|X \cap Y|/|X \cup Y|$ in all other cases.

We follow the above approach for several similarity measures like this, such as Dice, cosine, overlap coefficient, Jaccard, etc. (see the User Manual).

9 Further Discussion

Interplay between the Tokenizer and the Similarity Measure: We find that it is important to think about the interplay between the tokenizers and the string similarity measures, especially from a runtime point of view. For example, consider tokenizing two strings x and y into two sets, then apply a Jaccard similarity measure object to these two sets to compute a similarity score.

At the moment, the tokenizer will initially create two bags of tokens, then convert them into sets (as discussed earlier). The Jaccard object then checks the inputs to see if they are indeed sets, and converts them to sets if they are not. Finally, the Jaccard object uses the sets to compute a similarity score.

As described, there is some inefficiency in the interaction between the tokenizer and the Jaccard object, e.g., checking if the inputs are indeed sets is somewhat redundant. If we are to match millions of string pairs this way, then this small inefficiency can accumulate into a big one.

Thus, it is important to consider in the future what we can do to solve this problem. A possible solution is that in general, a similarity measure object such as the Jaccard object should still check if the inputs are indeed sets. But there can be some flags that the user can set to turn off such checking, in the cases of having to do so to minimize runtime.

Handling a Large Number of String Pairs: In general, we believe that ample care should be devoted to the scenario of matching millions of string pairs, and efforts be devoted to making the package very efficient for this scenario. We have discussed such efforts at various places in this document.

Missing Values: By “missing values” we mean cases where the values of one or more strings are missing (e.g., represented as None or NaN in Python). For example, consider the row “David,,38” in a CSV file. The value for the second cell of this row is missing. So when reading this file into a data frame, the corresponding cell will have the value NaN. Note that missing values are different from empty strings, which are represented as ““.

To handle missing values, we can first define multiple policies. One policy would be to always throw an error if encountering a missing value. Another policy is to quietly return a missing value. Two more common policies are “optimistic” and “pessimistic”. Consider computing the similarity score between a missing value u and a string v. The optimistic policy would return 1, on the ground that the missing string u can be the same as the string v in the optimistic case. Similarly, the pessimistic policy would return 0.

Ideally, we will provide flags that the user can set to select an appropriate policy to handle missing values, depending on the application.

For py_stringmatching, it turns out that it is difficult to enforce optimistic and pessimistic policies. This is because for certain similarity measures, it is not clear what should be returned. For example, for edit distance between a string and a missing value, we simply do not know what to return for the pessimistic case (or rather, we would need to return plus infinity in this case).

Allowing the policy of quietly returning a missing value is in general not a good idea, because the user would most likely be unaware of missing value cases, which can then propagate and cause serious problems, all quietly.

As a result, for py_stringmatching, we use the policy of raising an error. Specifically, tokenizers and sim measures expect the inputs to be strings. If the inputs are not (e.g., missing values), they will stop, raising an error.

Handling Unicode: Version 0.1.0 does not handle strings in unicode correctly. Specifically, if an input string contains non-ascii characters, a string similarity measure may interpret the string incorrectly and thus compute an incorrect similarity score. In this version we have fixed the string similarity measures. Specifically, we convert the input strings into unicode before computing similarity measures. NOTE: the tokenizers are still not yet unicode-aware.

10 Current Limitations and Possible Future Work

The main current limitations are:

- The coverage is still not as comprehensive as we would like. More similarity measures should be added. In particular, this package should be "backward compatible" with existing popular packages in that this package should contain all tokenizers/measures in those packages. We still have a few similarity measure for which we have developed the code but haven't added them to the package. We should consider adding them to the package next.
- So far we have only focused on benchmarking and scaling up Levenshtein. We need to scale up more measures.
- For each measure we may want to provide as much support as possible. These can include: provenance, debugging, saving/loading, logging, etc. For example, for TF/IDF we may consider capabilities to save the statistics (e.g., for debugging purposes). For hybrid measures, it may be important to provide debugging/provenance capabilities, and so on.

Some desirable capabilities/future work for tokenizing:

- It would be nice to have some basic text processing capabilities such as lowercasing everything, removing certain phrases before tokenizing, etc. Stemming may also be considered. Perhaps they can be built as flags into the tokenizer. Right now we have not implemented these.
- We will handle English text for now. Other languages can have issues that our current tokenizers do not take into account.
- For whitespace tokenizer, we have treated whitespace, tab, and new-line characters as delimiters. Even though this tokenizer inherits from Delimiter Tokenizer, its behavior is somewhat unusual. For example, it does not really use the delim_set field because it uses the split procedure in Python.

- Periods can be difficult. Sometimes they are considered part of the word (e.g., with initials, titles, abbreviations, and numbers), and sometimes they should not (e.g., end of the sentence). Same for hyphens and apostrophes. We have not handled periods in any way.
- For qgram tokenizers, it may be a good option to allow min and max. For example, if we set min = 2 and max = 4, then the tokenizer returns all grams that have 2, 3, and 4 characters.
- A tokenizer may be given a list of stopwords. They should not appear as tokens. We have not considered this.
- Study what it means for a tokenizer to be unicode-aware and then do that.

Some desirable capabilities/future work for similarity measures:

- Cosine coefficient as implemented right now is not the familiar vector-based cosine measure. That measure would need to be implemented.
- Soft TF/IDF is currently implemented without dampening. This needs to be fixed.
- We should make the code more efficient by minimizing redundant work between the tokenizer and the similarity measure object, as discussed in the discussion section.
- Currently the method get_sim_score often just calls get_raw_score. We need to fix this inefficiency issue.

11 Additional Pointers

Some interesting links:

- http://jmgomezhidalgo.blogspot.com.es/2013/07/performance-analysis-of-n-gram.html

A Current String Similarity Packages

As of June 2016, we counted 2 major non-Python packages and 7 major Python packages for string similarity. In what follows we describe these packages.

Non-Python Packages:

- **SimMetrics:** This is a Java library developed at Sheffield University, United Kingdom. It implements common sequence-based and set-based similarity measures (e.g., Levenshtein, Jaro, Jaro-Winkler, Jaccard, Dice, etc.).

- **SecondString:** Similar to SimMetrics, but developed at CMU.

Python Packages that Implement Only Edit Distance and Related Measures:

- **editdistance:** Python library for the fast computation of edit distance (also known as Levenshtein distance), using C++ and Cython.

- **pyxDamerauLevenshtein:** This library implements Damerau-Levenshtein (DL), an edit distance variation, using Cython. This distance differs from the classical Levenshtein distance by including transpositions among its allowable operations (insertion, deletion, or substitution of a single character, or a transposition of two adjacent characters). In contrast, the classical Levenshtein distance only allows insertion, deletion, and substitution operations.

- **pylev:** This library implements the Levenshtein distance and its variants (such as Damerau-Levenshtein distance), using Python. In our experiments, this package provides the slowest implementation of edit distance.

- **FuzzyWuzzy:** Python library that implements the Levenshtein distance and uses it to implement similarity measures such as simple ratio, partial ratio, token sort ratio, and token set ratio. It is implemented in Python.

Python Packages that Implement a Variety of Similarity Measures:

- **python-Levenshtein:** Despite the name, this package implements a variety of similarity measures, including edit distance, Jaro, Jaro-Winkler, Hamming, set ratio, etc. It does not implement common measures such as Jaccard, TF/IDF. The original code is in C. The package provides a wrap-around for the code so that it can be imported and run inside Python. In our experiments, this package provides the fastest implementation of edit distance.

- **jellyfish:** This package (in Python, but C implementation appears to be available) implements a variety of measures, including edit distance, Damerau-Levenshtein distance, Jaro distance, Jaro-Winkler distance, Hamming distance, and phonetic measures such as American soundex, metaphone, NYSIIS (New York State Identification and Intelligence System), and match rating codex. It does not implement any set-based similarity measures.

- **Abydos:** Python library for NLP/IR. It contains a module (distance.py) for string similarity measures. Implemented measures include edit distance, Hamming distance, Jaro distance, Jaro-Winkler distance, Smith-Waterman, soundex, Jaccard, cosine, etc. It does not implement common set-based similarity measures such as TF/IDF. Further, to use these measures, the user has to install the complete Abydos library.

Tokenizing: The above packages implement a limited set of tokenizers. Some packages do not implement tokenizing (e.g., editdistance). In some other packages (e.g., Abydos) the tokenizers are not exposed to the end users.

Notes: Recently we also found a string similarity package for Scala: https://rockymadden.com/stringmetric. It is likely that there are several such packages for each major programming language.

B Similarity Measures Implemented in Various Packages

In what follows we list similarity measures that have been implemented in various packages. These are the candidate measures that can be added to py_stringmatching in the future.

The following measures are either discussed in the book chapter [1], or implemented in SimMetrics, SecondString, Abydos, and python-Levenshtein.

- Sequence-based measures:
  - Edit distance
  - Jaro distance
  - Jaro-Winkler distance
  - Needleman-Wunsch distance
  - Smith-Waterman distance
  - Hamming distance
  - Affine gap
  - Optimal string alignment
  - Ratcliff-Obershelp metric
  - Matrix similarity
  - Gotoh score
  - Length similarity
  - Prefix, suffix, identity similarity
  - Modified language-independent product name search
  - Simon-White similarity
  - Approximate memo matrix
  - Approximate Needleman-Wunsch
  - Winkler rescorer
- Token-based measures:
  - Jaccard
  - Overlap coefficient
  - TF/IDF
  - Tanimoto index
  - Cosine
  - Tversky index

  - Sorensen-Dice index
  - Block distance
  - Dice
  - Euclidean distance
  - Bag distance
  - Jelinek-Mercer similarity
  - Jenson-Shannon distance
  - Softtoken Fellegi-Sunter

- Hybrid measures:
  - Monge-Elkan
  - Soft TF/IDF
  - Generalized Jaccard
  - Jaro TF/IDF
  - Jaro-Winkler TF/IDF
  - Tag Link

- Phonetic-based measures:
  - Soundex
  - Match rating
  - Editex distance

- Others:
  - Longest common substring
  - Normalized compression distance

References

[1] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012. Chapter 4 ”String Matching”, available from the package’s homepage.

[2] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2011. Pages 328-329.