COPADS, I: Distance Coefficients between Two Lists or Sets

Maurice HT Ling
School of Chemical and Life Sciences, Singapore Polytechnic
Department of Zoology, The University of Melbourne, Australia
mauriceling@acm.org

Abstract
The similarity between two objects can be denoted as a scalar distance between them, calculated by comparing the common regions with respect to all possible regions. However, there are many methods in computing the distance as the weights given to each region may differ. This article presents a collection of 35 distance coefficient measures with worked examples. These codes are licensed under Python Software Foundation License version 2.

1. Description

Given two vectors describing comparable features between two objects, the collective differences or dissimilarity can be denoted as a distance measure between the objects which can be seen as a scalar measure between the 2 objects in question (Basilevsky, 1983). Dissimilarity measures have been used in a number of applications where discrimination between objects is crucial, such as image processing (Birchfield and Tomasi, 1998), data mining (Rossi et al., 2006), ecological analysis (Werle et al., 2007), DNA fingerprinting (Lee at al., 2010), chemical analysis (Snarey et al., 1998) and handwriting analysis (Zhang and Srihani, 2003).

The distance can correspond to the size of overlapping region (the intersecting area) between the objects as shown in Figure 1, denoted as “A” (number of elements in both ‘test’ and ‘original’), “B” (number of elements in ‘original’ only), “C” (number of elements in ‘test’ only), “D” (number of elements in neither ‘test’ nor ‘original’). The ‘test’ and ‘original’ data are translated to “A”, “B”, “C”, “D” using *compare* function which can perform both set and list comparison. Set comparison uses set memberships between ‘test’ and ‘original’ data. This implies that set comparison may be used for unordered list. List comparison performs equality test for each element and only when the same datum exist for both ‘test’ and ‘original’ data at the same subscript will it be considered to be present for both. However, a presence test can be used in place of equality test by passing ‘test’ and ‘original’ data through *binarize* function independently.

However, there are numerous ways in computing the distance coefficients as the weights given to each region may differ. A common feature among different distance coefficients is the presence of both upper and lower boundaries signifying the highest and lowest possible value. The lower boundary of a coefficient signifies complete difference (no similarity) between the two objects while the upper boundary of a coefficient signifies complete similarity (no difference). It is also important to note that the presence of upper and lower boundaries is not universal. For example, the lower boundary of zero is defined for Hamming coefficient (Hamming, 1950) but its upper boundary is infinite.

This article presents a collection of 35 distance coefficient measures with worked examples:

1. Jaccard (Jaccard, 1901, Jaccard 1908)
2. Dice (Dice, 1945, Sorensen, 1948)
3. Sokal and Michener (Sokal and Michener, 1958)
4. Matching (Dunn and Everitt, 1982, Sokal and Michener, 1958)
5. Anderberg (Anderberg, 1973)
6. Ochiai (Ochiai, 1957)
7. Ochiai 2 (Ochiai, 1957)
8. First Kulcsynski (Holliday et al., 2002)
9. Second Kulcsynski (Holliday et al., 2002)
10. Forbes (Forbes, 1907)
11. Hamann (Hamann, 1961)
12. Simpson (Fallaw, 1979)
13. Russel and Rao (Russel and Rao, 1940)
14. Roger and Tanimoto (Roger and Tanimoto, 1960)
15. Sokal and Sneath (Sokal and Sneath, 1963)
16. Sokal and Sneath 2 (Sokal and Sneath, 1963)
17. Sokal and Sneath 3 (Sokal and Sneath, 1963)
18. Buser (Holliday et al., 2002)
19. Fossum (Fossum, 1966)
20. Yule Q (Yule, 1912)
21. Yule Y (Yule, 1912)
22. McConnaughey (McConnaughey, 1964)
23. Stiles (Stiles, 1923)
24. Pearson (Ellis et al., 1993)
25. Dennis (Dennis, 1965)
26. Gower and Legendre (Gower and Legendre, 1986)
27. Tulloss (Tulloss, 1997)
28. Hamming (Hamming, 1950)
29. Euclidean
30. Minkowski (Basilevski, 1983)
31. Manhattan (Krause, 1987)
32. Canberra (Lance and William, 1966)
33. Complement Bray and Curtis (Bray and Curtis, 1957)
34. Cosine (Basilevski, 1983)
35. Tanimoto (Fligner et al., 2002, Tanimoto, 1958)

These codes are implemented as part of COPADS library (http://copads.sf.net) and licensed under Python Software Foundation License version 2.

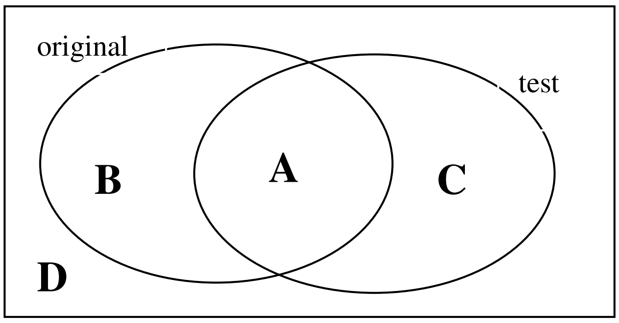

Figure 1: Venn diagram illustrating the overlapping regions between two objects. The objects are listed as 'original' and 'test'. 'A' is the region of intersection and region 'D' signifies elements not present in both 'original' and 'test' objects.

With reference to Figure 1, the above coefficient measures can be defined as follows:

$$Jaccard\ coefficient = \frac{A}{A+B+C}$$

$$Dice\ coefficient = \frac{2A}{(A+B)+\ (A+C)}$$

$$Sokal\ and\ Michener\ coefficient = \frac{A+D}{A+B+C+D}$$

$$Matching\ coefficient = \frac{A+D}{(A+B)+\ (A+C)}$$

$$Ochiai\ coefficient = \frac{A}{\sqrt{(A+B)(A+C)}}$$

$$Ochiai\ 2\ coefficient = \frac{A\times D}{\sqrt{(A+B)(A+C)(B+D)(C+D)}}$$

$$Anderberg\ coefficient = \frac{A}{A+2\times(B+C)}$$

$$First\ Kulczynski\ coefficient = \frac{A}{B+C}$$

$$Second\ Kulczynski\ coefficient = \frac{\frac{A}{A+B}+\frac{A}{A+C}}{2}$$

$$Forbes\ coefficient = \frac{A\ \times(A+B+C+D)}{(A+B)\times(A+C)}$$

$$Hamann\ coefficient = \frac{(A+D)-(B+C)}{A+B+C+D}$$

$$Simpson\ coefficient = \frac{A}{\min((A+B),(A+C))}$$

$$Russell\ and\ Rao\ coefficient = \frac{A}{A+B+C+D}$$

$$Roger\ and\ Tanimoto\ coefficient = \frac{A+D}{A+2B+2C+D}$$

$$Sokal\ and\ Sneath\ coefficient = \frac{A}{A+2B+2C}$$

$$Sokal\ and\ Sneath\ 2\ coefficient = \frac{2A+2D}{2A+B+C+2D}$$

$$Sokal\ and\ Sneath\ 3\ coefficient = \frac{A+D}{B+C}$$

$$Buser\ coefficient = \frac{\sqrt{(A+D)}+A}{\sqrt{(A+D)}+A+B+C}$$

$$Fossum\ coefficient = \frac{(A+B+C+D)(A-0.5)^2}{(A+B)(A+C)}$$

$$Yule\ Q\ coefficient = \frac{(A\times D)-(B\times C)}{(A\times D)+(B\times C)}$$

$$Yule\ Y\ coefficient = \frac{\sqrt{(A\times D)}-\sqrt{(B\times C)}}{\sqrt{(A\times D)}+\sqrt{(B\times C)}}$$

$$McConnaughey\ coefficient = \frac{A^2-(B\times C)}{(A+B)(A+C)}$$

$$Stiles\ coefficient = \log_{10}\frac{(A+B+C+D)\left[|(A\times D)-(B\times C)|-\frac{(A+B+C+D)}{2}\right]^2}{(A+B)(A+C)(B+D)(C+D)}$$

$$Pearson\ coefficient = \frac{(A\times D)-(B\times C)}{\sqrt{(A+B)(A+C)(B+D)(C+D)}}$$

$$Dennis\ coefficient = \frac{(A\times D)-(B\times C)}{\sqrt{(A+B+C+D)(A+B)(A+C)}}$$

$$Gower\ and\ Legendre\ coefficient = \frac{A+D}{A+\big(0.5\times(B+C)\big)+D}$$

$$Tulloss\ coefficient = \sqrt{\left[\frac{\log_{10}\left(\frac{\min(B,C)+A}{\max(B,C)+A}\right)}{\log_{10}2}\right]\times\left[\frac{1}{\sqrt{\frac{\log_{10}\left(2+\frac{\min(B,C)}{A+1}\right)}{\log_{10}2}}}\right]\times\left[\frac{\log_{10}\left(1+\frac{\mathrm{A}}{A+B}\right)\log_{10}\left(1+\frac{\mathrm{A}}{A+C}\right)}{(\log_{10}2)^2}\right]}$$

$$Hamming\ coefficient = \sum_{i=1} m_i \begin{cases} m=1, (A+B)_i \neq (A+C)_i \\ m=0, (A+B)_i = (A+C)_i \end{cases}$$

$$Euclidean\ coefficient = \sqrt{\sum_{i=1}((A+B)_i - (A+C)_i)^2}$$

$$Minkowski\ coefficient\ (p) = \sqrt[p]{\sum_{i=1}((A+B)_i - (A+C)_i)^p}$$

$$Manhattan\ coefficient = \sum_{i=1}|(A+B)_i - (A+C)_i|$$

$$Canberra\ coefficient = \sum_{i=1}\frac{|(A+B)_i - (A+C)_i|}{|(A+B)_i + (A+C)_i|}$$

$$Complement\ Bray\ and\ Curtis\ coefficient = 1 - \frac{\sum_{i=1}|(A+B)_i - (A+C)_i|}{\sum_{i=1}(A+B)_i + \sum_{i=1}(A+C)_i}$$

$$Cosine\ coefficient = \frac{\sum_{i=1}(A+B)_i \times (A+C)_i}{\sqrt{\sum_{i=1}(A+B)_i^2} \times \sqrt{\sum_{i=1}(A+C)_i^2}}$$

$$Tanimoto\ coefficient = \frac{\sum_{i=1}(A+B)_i \times (A+C)_i}{\sum_{i=1}(A+B)_i^2 + \sum_{i=1}(A+C)_i^2 - (\sum_{i=1}(A+B)_i \times (A+C)_i)}$$

2. Worked Examples

Given 3 sets of data

```
O1 = ['C', 'D', 'E', 'F', 'G', 'H']
T1 = ['B', 'E', 'D', 'F', 'G', 'H']

O2 = [1, 1, 1, 0, 1, 1]
T2 = [1, 1, 0, 1, 1, 1]

O3 = [1.0, 2.0, 3.0, 6.0, 8.0, 9.0]
T3 = [1.0, 2.0, 4.0, 4.0, 8.0, 9.0]
```

Measure	Set distance between O1 and T1	List distance between O2 and T2
Jaccard	$\frac{5}{5+1+1} = 0.7143$	$\frac{4}{4+1+1} = 0.6667$
Sokal and Michener	$\frac{5+0}{5+1+1+0} = 0.7143$	$\frac{4+0}{4+1+1+0} = 0.6667$
Matching	$\frac{5+0}{(5+1)+(5+1)} = 0.4167$	$\frac{4+0}{(4+1)+(4+1)} = 0.4000$

Dice	$\frac{2(5)}{(5+1)+(5+1)}=0.8333$	$\frac{2(4)}{(5+1)+(5+1)}=0.6667$
Ochiai	$\frac{5}{\sqrt{(5+1)(5+1)}}=0.8333$	$\frac{4}{\sqrt{(4+1)(4+1)}}=0.8000$
Ochiai 2	$\frac{5\times 0}{\sqrt{6\times 6\times 1\times 1}}=0.000$	$\frac{4\times 0}{\sqrt{6\times 6\times 1\times 1}}=0.000$
Anderberg	$\frac{5}{5+2\times(1+1)}=0.5555$	$\frac{4}{4+2\times(1+1)}=0.5000$
First Kulczynski	$\frac{5}{1+1}=2.5000$	$\frac{4}{1+1}=2.0000$
Second Kulczynski	$\left(\frac{5}{5+1}+\frac{5}{5+1}\right)/_{2}=0.8333$	$\left(\frac{4}{4+1}+\frac{4}{4+1}\right)/_{2}=0.8000$
Forbes	$\frac{5\times(5+1+1+0)}{(5+1)\times(5+1)}=0.9722$	$\frac{4\times(4+1+1+0)}{(4+1)\times(4+1)}=0.9600$
Hamann	$1-\frac{(5+0)-(1+1)}{5+1+1+0}=0.4286$	$\frac{(4+0)-(1+1)}{4+1+1+0}=0.3333$
Simpson	$\frac{5}{\min((5+1),(5+1))}=0.8333$	$\frac{4}{\min((4+1),(4+1))}=0.8000$
Russell and Rao	$\frac{5}{5+1+1+0}=0.7143$	$\frac{4}{4+1+1+0}=0.6667$
Roger and Tanimoto	$\frac{5+0}{5+2+2+0}=0.5555$	$\frac{4+0}{4+2+2+0}=0.5000$
Sokal and Sneath	$\frac{5}{5+2+2}=0.5555$	$\frac{4}{4+2+2}=0.5000$
Sokal and Sneath 2	$\frac{2(5)+2(0)}{2(5)+1+1+2(0)}=0.8333$	$\frac{2(4)+2(0)}{2(4)+1+1+2(0)}=0.8000$
Sokal and Sneath 3	$\frac{5+0}{1+1}=2.5000$	$\frac{4+0}{1+1}=2.000$
Buser	$\frac{\sqrt{(5+0)}+5}{\sqrt{(5+0)}+5+1+1}=0.7835$	$\frac{\sqrt{(4+0)}+4}{\sqrt{(4+0)}+4+1+1}=0.7500$
Fossum	$\frac{(5+1+1+0)(5-0.5)^2}{(5+1)(5+1)}=3.9375$	$\frac{(4+1+1+0)(4-0.5)^2}{(4+1)(4+1)}=2.9400$
Yule Q	$\frac{(5\times 0)-(1\times 1)}{(5\times 0)+(1\times 1)}=-1.000$	$\frac{(4\times 0)-(1\times 1)}{(4\times 0)+(1\times 1)}=-1.000$
Yule Y	$\frac{\sqrt{(5\times 0)}-\sqrt{(1\times 1)}}{\sqrt{(5\times 0)}+\sqrt{(1\times 1)}}=-1.000$	$\frac{\sqrt{(4\times 0)}-\sqrt{(1\times 1)}}{\sqrt{(4\times 0)}+\sqrt{(1\times 1)}}=-1.000$

McConnaughey	$\frac{5^2-(1\times 1)}{(5+1)(5+1)}=0.6667$	$\frac{4^2-(1\times 1)}{(4+1)(4+1)}=0.6000$
Stiles	$\log_{10}\frac{7\left[\lvert(0)-(1)\rvert-\frac{7}{2}\right]^2}{(6)(6)(1)(1)}=0.847$	$\log_{10}\frac{6\left[\lvert(0)-(1)\rvert-\frac{6}{2}\right]^2}{(5)(5)(1)(1)}=-0.0177$
Pearson	$\frac{0-1}{\sqrt{(6)(6)(1)(1)}}=-0.1667$	$\frac{0-1}{\sqrt{(5)(5)(1)(1)}}=-0.2000$
Dennis	$\frac{0-1}{\sqrt{(7)(6)(6)}}=-0.630$	$\frac{0-1}{\sqrt{(6)(5)(5)}}=-0.0816$
Gower and Legendre	$\frac{5+0}{5+\left(0.5\times(1+1)\right)+0}=0.8333$	$\frac{4+0}{4+\left(0.5\times(1+1)\right)+0}=0.8000$
Tulloss	$\sqrt{\left[\frac{0}{0.301}\right]\times\left[\frac{1}{\sqrt{\frac{0.336}{0.301}}}\right]\times\left[\frac{0.526}{0.0906}\right]}=0.8509$	$\sqrt{\left[\frac{0}{0.301}\right]\times\left[\frac{1}{\sqrt{\frac{0.342}{0.301}}}\right]\times\left[\frac{0.511}{0.0906}\right]}=0.8211$

Table 1: List of Coefficients for Binary Data

Measure	**List distance between O2 and T3**	**List distance between O3 and T3**
Hamming	$0+0+1+1+0+0=2$	$0+0+1+1+0+0=2$
Euclidean	$\sqrt{0+0+1+1+0+0}=1.4142$	$\sqrt{0+0+1+4+0+0}=2.2361$
Minkowski(3)	$\sqrt[3]{0+0+1+1+0+0}=1.2599$	$\sqrt[3]{0+0+1+8+0+0}=2.0801$
Manhattan	$0+0+1+1+0+0=2$	$0+0+1+2+0+0=3$
Canberra	$\frac{0}{2}+\frac{0}{2}+\frac{1}{1}+\frac{1}{1}+\frac{0}{2}+\frac{0}{2}=2$	$\frac{0}{2}+\frac{0}{4}+\frac{1}{7}+\frac{2}{10}+\frac{0}{16}+\frac{0}{18}=0.3429$
Complement Bray and Curtis	$1-\frac{0+0+1+1+0+0}{5+5}=0.8000$	$1-\frac{0+0+1+2+0+0}{29+28}=0.9474$
Cosine	$\frac{1+1+0+0+1+1}{\sqrt{5}\times\sqrt{5}}=0.8000$	$\frac{1+4+12+24+64+81}{\sqrt{195}\times\sqrt{182}}=0.9873$
Tanimoto	$\frac{1+1+0+0+1+1}{5+5-4}=0.6667$	$\frac{1+4+12+24+64+81}{195+182-186}=0.9738$

Table 2: List of Coefficients for Binary and Interval Data

3. Code Files

File: objectdistance.py

```
"""
Functions for Calculating Similarity Coefficients between 2 Objects.

Generally, the lower boundary of similar coefficient
signifies complete difference (no similarity) while the upper boundary
(if any) signifies complete similarity (no difference).

In the following formulae, the following notations will be used
    - A = found in both 'original' and 'test'
    - B = found in 'original' only
    - C = found in 'test' only
    - D = not found in either 'original' or 'test'

Copyright (c) Maurice H.T. Ling <mauriceling@acm.org>

Date created: 17th August 2005
"""

import math
from copadsexceptions import DistanceInputSizeError


def binarize(data, absent=0):
    """
    Converts input data in a list of presence or absence of values.
    For example,
    binarize([1, 2, 0, 3, 4, 0], 0) --> [1, 1, 0, 1, 1, 0]
    binarize([1, 2, 0, 3, 4, 0], 2) --> [1, 0, 1, 1, 1, 1]
    """
    return [{absent: 0}.get(x, 1) for x in data]

def compare(original, test, absent, cmp_type='Set'):
    """
    Used for processing list-based (positional) or set-based
    (non-positional) distance of categorical data.

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: indicator to define absent data
    @param cmp_type: {List | Set}
    """
    if cmp_type == 'Set':
        original_only = float(len([x for x in original if x not in test]))
        test_only = float(len([x for x in test if x not in original]))
        both = float(len([x for x in original if x in test]))
        return (original_only, test_only, both, 0.0)
    if cmp_type == 'List':
        original, test = list(original), list(test)
        original_only, test_only, both, none = 0.0, 0.0, 0.0, 0.0
        for i in range(len(original)):
            if original[i] == absent and test[i] == absent:
                none = none + 1
            elif original[i] == test[i]:
                both = both + 1
            elif original[i] <> absent and test[i] == absent:
                original_only = original_only + 1
            elif original[i] == absent and test[i] <> absent:
                test_only = test_only + 1
            else: pass
        return (original_only, test_only, both, none)

def Jaccard(original, test, absent=0, cmp_type='Set'):
    """
    Jaccard coefficient for nominal or ordinal data.

    Coefficient: M{A / (A + B + C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    return both / (both + original + test)

def Sokal_Michener(original, test, absent=0, cmp_type='Set'):
    """
    Sokal and Michener coefficient for nominal or ordinal data.

    Coefficient: M{(A + D) / (A + B + C + D)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    if len(original) <> len(test):
        raise DistanceInputSizeError("Size (length) of inputs must be \
                equal for Sokal & Michener's distance")
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    return (both + none) / (original + test + both + none)

def Matching(original, test, absent=0, cmp_type='Set'):
    """
    Matching coefficient for nominal or ordinal data

    Coefficient: M{(A + D) / (2A + B + C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    return (both + none) / (original + both + test + both)

def Dice(original, test, absent=0, cmp_type='Set'):
    """
    Dice coefficient for nominal or ordinal data.

    Coefficient: M{2A / (2A + B + C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    all_region = float(len(original)) + float(len(test))
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    return (2 * both) / all_region

def Ochiai(original, test, absent=0, cmp_type='Set'):
    """
    Ochiai coefficient for nominal or ordinal data.

    Coefficient: M{2A / sqrt((A + B)(A + C)))}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    return both / ((both + original) * (both + test)) ** 0.5

def Ochiai2(original, test, absent=0, cmp_type='Set'):
    """
    Ochiai 2 coefficient for nominal or ordinal data, and requires
    the presence of regions whereby both original and test are not
    present.

    Coefficient: M{(A * D) / sqrt((A + B)(A + C)(D + B)(D + C))}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    denominator = ((both + original) * (both + test) * \
                  (none + original) * (none + test)) ** 0.5
    return (both * none) / denominator

def Anderberg(original, test, absent=0, cmp_type='Set'):
    """
    Anderberg coefficient for nominal or ordinal data.

    Coefficient: M{A / (A + 2(B + C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    return both / (both + 2 * (original + test))

def Kulczynski2(original, test, absent=0, cmp_type='Set'):
    """
    Second Kulczynski coefficient for nominal or ordinal data.

    Coefficient: M{((A / (A + B)) + (A / (A + C))) / 2}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    x1 = both / (original + both)
    x2 = both / (test + both)
    return (x1 + x2) / 2

def Kulczynski(original, test, absent=0, cmp_type='Set'):
    """
    First Kulczyski coefficient for nominal or ordinal data.

    Coefficient: M{A / (B + C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    return both / (original + test)

def Forbes(original, test, absent=0, cmp_type='Set'):
    """
    Forbes coefficient for nominal or ordinal data.

    Coefficient: M{A(A + B + C + D) / ((A + B)(A + C))}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    numerator = both * (both + original + test + none)
    denominator = (original + both) * (test + both)
    return numerator / denominator

def Hamann(original, test, absent=0, cmp_type='Set'):
```

```
    """
    Hamann coefficient for nominal or ordinal data.

    Coefficient: M{((A + D) - (B + C)) / (A + B + C + D)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    numerator = (both + none) - (test + original)
    return numerator / (original + test + both + none)

def Simpson(original, test, absent=0, cmp_type='Set'):
    """
    Simpson coefficient for nominal or ordinal data.

    Coefficient: M{A / min(A + B, A + C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    return both / min(both + test, both + original)

def Russel_Rao(original, test, absent=0, cmp_type='Set'):
    """
    Russel and Rao coefficient for nominal or ordinal data.

    Coefficient: M{A / (A + B + C + D)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    return both / (original + test + both + none)

def Roger_Tanimoto(original, test, absent=0, cmp_type='Set'):
    """
    Roger and Tanimoto coefficient for nominal or ordinal data.

    Coefficient: M{(A + D) / (A + 2B + 2C + D)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    denominator = (2 * original) + (2 * test) + both + none
    return (both + none) / denominator

def Sokal_Sneath(original, test, absent=0, cmp_type='Set'):
    """
    Sokal and Sneath coefficient for nominal or ordinal data.

    Coefficient: M{A / (A + 2B + 2C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    denominator = (2 * original) + (2 * test) + both
    return both / denominator

def Sokal_Sneath2(original, test, absent=0, cmp_type='Set'):
    """
    Sokal and Sneath 2 coefficient for nominal or ordinal data.

    Coefficient: M{(2A + 2D) / (2A + B + C + 2D)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    numerator = (2 * both) + (2 * none)
    denominator = (2 * both) + original + test + (2 * none)
    return numerator / denominator

def Sokal_Sneath3(original, test, absent=0, cmp_type='Set'):
    """
    Sokal and Sneath 3 coefficient for nominal or ordinal data.

    Coefficient: M{(A + D) / (B + C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    return (both + none) / (test + original)

def Buser(original, test, absent=0, cmp_type='Set'):
    """
    Buser coefficient (also known as Baroni-Urbani coefficient)
    for nominal or ordinal data.
    Coefficient: M{(sqrt(A * D) + A) / (sqrt(A * D) + A + B + C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    t = (both + none) ** 0.5
    return (t + both) / (t + both + test + original)

def Fossum(original, test, absent=0, cmp_type='Set'):
    """
    Fossum coefficient for nominal or ordinal data.

    Coefficient:
    M{((A + B + C + D)(A - 0.5)^2) / ((A + B)(A + C))}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    numerator = original + test + both + none
    numerator = numerator * (both - 0.5) ** 2
    return numerator / ((both + test) * (both + original))

def YuleQ(original, test, absent=0, cmp_type='Set'):
    """
    Yule Q coefficient (also known as First Yule coefficient) for
    nominal or ordinal data.

    Coefficient: M{((A * D) - (B * C)) / ((A * D) + (B * C))}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    numerator = (both * none) - (test * original)
    denominator = (both * none) + (test * original)
    return numerator / denominator

def YuleY(original, test, absent=0, cmp_type='Set'):
    """
    Yule Y coefficient (also known as Second Yule coefficient) for
    nominal or ordinal data.

    Coefficient:
    M{(sqrt(A * D) - sqrt(B * C)) / (sqrt(A * D) + sqrt(B * C))}
    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    numerator = ((both * none) ** 0.5) - ((test * original) ** 0.5)
    denominator = ((both * none) ** 0.5) + ((test * original) ** 0.5)
    return numerator / denominator

def Mcconnaughey(original, test, absent=0, cmp_type='Set'):
    """
    McConnaughey coefficient for nominal or ordinal data.

    Coefficient: M{(A^2 - (B * C)) / (A + B)(A + C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    numerator = (both * both) - (original * test)
    denominator = (both + original) * (both + test)
    return numerator / denominator

def Stiles(original, test, absent=0, cmp_type='Set'):
    """
    Stiles coefficient for nominal or ordinal data.

    Coefficient:
    M{log10(((A + B + C + D)(|(A * D) - (B * C)| - ((A + B + C + D) / 2)) ^ 2)
    / (A + B)(A + C)(B + D)(C + D))}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    n = original + test + both + none
    t = abs((both * none) - (test * original))
    numerator = n * ((t - (0.5 * n)) ** 2)
    denominator = (both + original) * (both + test) * \
                  (none + original) * (none + test)
    return math.log10(numerator / denominator)

def Pearson(original, test, absent=0, cmp_type='Set'):
    """
    Pearson coefficient for nominal or ordinal data.

    Coefficient:
    M{((A * D) - (B * C)) / (A + B)(A + C)(B + D)(C + D)}
    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    numerator = (both * none) - (test * original)
    denominator = ((both + original) * (both + test) * \
                   (none + original) * (none + test)) ** 0.5
    return numerator / denominator

def Dennis(original, test, absent=0, cmp_type='Set'):
    """
    Dennis coefficient for nominal or ordinal data.

    Coefficient:
    M{((A * D) - (B * C)) / (A + B + C + D)(A + B)(A + C)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    n = original + test + both + none
    numerator = (both * none) - (test * original)
    denominator = (n * (both + original) * (both + test)) ** 0.5
    return numerator / denominator

def Gower_Legendre(original, test, absent=0, cmp_type='Set'):
    """
    Gower and Legendre coefficient for nominal or ordinal data.

    Coefficient: M{(A + D) / ((0.5 * (B + C)) + A + D)}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    n = original + test + both + none
    numerator = both + none
    denominator = (0.5 * (test + original)) + numerator
    return numerator / denominator

def Tulloss(original, test, absent=0, cmp_type='Set'):
    """
    Tulloss coefficient for nominal or ordinal data.

    Coefficient:
    M{sqrt(U * S * R)}
        - M{U = log(1 + ((min(B, C) + A) / (max(B, C) + A))) / log2}
        - M{S = 1 / sqrt(log(2 + (min(B, C) / (A + 1))) / log2)}
        - M{R = log(1 + A / (A + B))log(1 + A / (A + C)) / log2log2}

    @param original: list of original data
    @param test: list of data to test against original
    @param absent: user-defined identifier for absent of region,
        default = 0
    @param cmp_type: {Set | List}, define whether use Set comparison
        (non-positional) or list comparison (positional), default = Set
    """
    (original, test, both, none) = compare(original, test, absent, cmp_type)
    U = (min(test, original) + both) / (max(test, original) + both)
    U = math.log10(1 + U) / math.log10(2)
    S = math.log10(2 + (min(test, original) / (both + 1)))
    S = 1 / ((S / math.log10(2)) ** 0.5)
    R = math.log10(1 + (both / (both + original)))
    R = R * math.log10(1 + (both / (both + test)))
    R = R / (math.log10(2) * math.log10(2))
    return (U * S * R) ** 0.5

def Hamming(original, test):
    """
    Hamming coefficient for ordinal data - only for positional data.

    Coefficient: number of mismatches with respect to position

    @param original: list of original data
    @param test: list of data to test against original
    """
    if len(original) <> len(test):
        raise DistanceInputSizeError("Size (length) of inputs must be \
            equal for Hamming's distance")
    mismatch = 0
    for index in range(len(original)):
        if original[index] <> test[index]: mismatch = mismatch + 1
    return mismatch

def Euclidean(original, test):
    """
    Euclidean coefficient for interval or ratio data.

    Coefficient: M{sqrt(S{sum}(((A + B)(i) - (A + C)(i)) ^ 2))}

    euclidean(original, test) -> euclidean distance between original
    and test. Adapted from BioPython

    @param original: list of original data
    @param test: list of data to test against original"""
    # lightly modified from implementation by Thomas Sicheritz-Ponten.
    # This works faster than the Numeric implementation on shorter
    # vectors.
    if len(original) != len(test):
        raise DistanceInputSizeError("Size (length) of inputs must be \
            equal for Euclidean distance")
    sum = 0
    for i in range(len(original)):
        sum = sum + (original[i] - test[i]) ** 2
    return math.sqrt(sum)

def Minkowski(original, test, power=3):
    """
    Minkowski coefficient for interval or ratio data.

    Coefficient: M{power-th root(S{sum}(((A + B)(i) - (A + C)(i)) ^ power))}

    Minkowski Distance is a generalized absolute form of Euclidean
    Distance. Minkowski Distance = Euclidean Distance when power = 2

    @param original: list of original data
    @param test: list of data to test against original
    @param power: expontential variable
    @cmp_type power: integer"""
    if len(original) != len(test):
        raise DistanceInputSizeError("Size (length) of inputs must be \
            equal for Minkowski distance")
    sum = 0
    for i in range(len(original)):
        sum = sum + abs(original[i] - test[i]) ** power
    return sum ** (1 / float(power))

def Manhattan(original, test):
    """
    Manhattan coefficient for interval or ratio data.

    Coefficient: M{S{sum}(abs((A + B)(i) - (A + C)(i)))}

    Manhattan Distance is also known as City Block Distance. It is
    essentially summation of the absolute difference between each
    element.

    @see: Krause, Eugene F. 1987. Taxicab Geometry. Dover. ISBN 0-486-
    25202-7.

    @param original: list of original data
    @param test: list of data to test against original"""
    if len(original) != len(test):
        raise DistanceInputSizeError("Size (length) of inputs must be \
            equal for Manhattan distance")
    sum = 0
    for i in range(len(original)):
        sum = sum + abs(original[i] - test[i])
    return float(sum)

def Canberra(original, test):
    """
    Canberra coefficient for interval or ratio data.

    Coefficient:
    M{S{sum}(abs((A + B)(i) - (A + C)(i)) / abs((A + B)(i) + (A + C)(i)))}

    @see: Lance GN and Williams WT. 1966. Computer programs for
    hierarchical polythetic classification. Computer Journal 9: 60-64.

    @param original: list of original data
    @param test: list of data to test against original"""
    if len(original) != len(test):
        raise DistanceInputSizeError("Size (length) of inputs must be \
            equal for Canberra distance")
    sum = 0
    for i in range(len(original)):
        sum = sum + (abs(original[i] - test[i]) / abs(original[i] + \
            test[i]))
    return sum

def Bray_Curtis(original, test):
    """
    Complement Bray and Curtis coefficient for interval or ratio data.
    Lower boundary of Bray and Curtis coefficient represents complete
    similarity (no difference).

    Coefficient:
    M{1 - S{sum}(abs((A + B)(i) - (A + C)(i))) /
    (S{sum}((A + B)(i)) + S{sum}((A + C)(i)))}

    @see: Bray JR and Curtis JT. 1957. An ordination of the upland
    forest communities of S. Winconsin. Ecological Monographs 27:
    325-349.

    @param original: list of original data
    @param test: list of data to test against original"""
    if len(original) != len(test):
        raise DistanceInputSizeError("Size (length) of inputs must be \
            equal for Bray-Curtis distance")
    return 1 - (Manhattan(original, test) / \
            float(sum(original) + sum(test)))

def Cosine(original, test):
    """
    Cosine coefficient for interval or ratio data.

    Coefficient:
    M{S{sum}(abs((A + B)(i) * (A + C)(i))) /
    (S{sum}((A + B) ^ 2) * S{sum}((A + C) ^ 2))}

    @param original: list of original data
    @param test: list of data to test against original"""
    if len(original) != len(test):
        raise DistanceInputSizeError("Size (length) of inputs must be \
            equal for Cosine distance")
    original = [float(x) for x in original]
    test = [float(x) for x in test]
    numerator = sum([original[x] * test[x] for x in range(len(original))])
    denominator = sum([x * x for x in original]) ** 0.5
    denominator = denominator * (sum([x * x for x in test]) ** 0.5)
    return numerator / denominator

def Tanimoto(original, test):
    """
    Tanimoto coefficient for interval or ratio data.

    Coefficient:
    M{S{sum}(abs((A + B)(i) * (A + C)(i))) /
    (S{sum}((A + B) ^ 2) + S{sum}((A + C) ^ 2) -
    S{sum}(abs((A + B)(i) * (A + C)(i))))}

    @param original: list of original data
    @param test: list of data to test against original"""
    if len(original) != len(test):
        raise DistanceInputSizeError("Size (length) of inputs must be \
            equal for Cosine distance")
    original = [float(x) for x in original]
    test = [float(x) for x in test]
    numerator = sum([original[x] * test[x] for x in range(len(original))])
    denominator = sum([x * x for x in original])
    denominator = denominator + (sum([x * x for x in test])) - numerator
    return numerator / denominator
```

File: testfile.py

```
import sys
import os
import unittest

sys.path.append(os.path.join(os.path.dirname(os.getcwd()), 'copads'))
import objectdistance as D

# for set comparison
# 5 elements in both
# 1 element in test
# 1 element in original
o1 = ['C', 'D', 'E', 'F', 'G', 'H']
t1 = ['B', 'E', 'D', 'F', 'G', 'H']

# for list comparison
# 4 elements in both
# 1 element in test
# 1 element in original
o2 = [1, 1, 1, 0, 1, 1]
t2 = [1, 1, 0, 1, 1, 1]

# for list comparison in interval/ratio measures
o3 = [1.0, 2.0, 3.0, 6.0, 8.0, 9.0]
t3 = [1.0, 2.0, 4.0, 4.0, 8.0, 9.0]


class testSet(unittest.TestCase):
    def testJaccard(self):
        'Jaccard for set'
        distance = D.Jaccard(o1, t1)
        actual = 0.7143
        self.assertAlmostEqual(distance, actual, places=4)

    def testJaccard1(self):
        'Jaccard for set (self)'
        distance = D.Jaccard(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Michener(self):
        'Sokal and Michener for set'
        distance = D.Sokal_Michener(o1, t1)
        actual = 0.7143
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Michener1(self):
        'Sokal and Michener for set (self)'
        distance = D.Sokal_Michener(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)
    def testMatching(self):
        'Matching for set'
        distance = D.Matching(o1, t1)
        actual = 0.4167
        self.assertAlmostEqual(distance, actual, places=4)

    def testMatching1(self):
        'Matching for set (self)'
        distance = D.Matching(o1, o1)
        actual = 0.5
        self.assertAlmostEqual(distance, actual, places=4)

    def testDice(self):
        'Dice for set'
        distance = D.Dice(o1, t1)
        actual = 0.8333
        self.assertAlmostEqual(distance, actual, places=4)

    def testDice1(self):
        'Dice for set (self)'
        distance = D.Dice(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testOchiai(self):
        'Ochiai for set'
        distance = D.Ochiai(o1, t1)
        actual = 0.8333
        self.assertAlmostEqual(distance, actual, places=4)

    def testOchiai_1(self):
        'Ochiai for set (self)'
        distance = D.Ochiai(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testOchiai2(self):
        'Ochiai 2 for set'
        distance = D.Ochiai2(o1, t1)
        actual = 0.0000
        self.assertAlmostEqual(distance, actual, places=4)

    def testAnderberg(self):
        'Anderberg for set'
        distance = D.Anderberg(o1, t1)
        actual = 0.55555
        self.assertAlmostEqual(distance, actual, places=4)

    def testAnderberg1(self):
        'Anderberg for set (self)'
        distance = D.Anderberg(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testKulczynski2(self):
        'Kulczynski 2 for set'
        distance = D.Kulczynski2(o1, t1)
        actual = 0.8333
        self.assertAlmostEqual(distance, actual, places=4)
    def testKulczynski2_1(self):
        'Kulczynski 2 for set (self)'
        distance = D.Kulczynski2(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testKulczynski(self):
        'Kulczynski for set'
        distance = D.Kulczynski(o1, t1)
        actual = 2.5000
        self.assertAlmostEqual(distance, actual, places=4)

    def testForbes(self):
        'Forbes for set'
        distance = D.Forbes(o1, t1)
        actual = 0.9722
        self.assertAlmostEqual(distance, actual, places=4)

    def testForbes1(self):
        'Forbes for set (self)'
        distance = D.Forbes(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testHamann(self):
        'Hamann for set'
        distance = D.Hamann(o1, t1)
        actual = 0.4286
        self.assertAlmostEqual(distance, actual, places=4)

    def testHamann1(self):
        'Hamann for set (self)'
        distance = D.Hamann(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testSimpson(self):
        'Simpson for set'
        distance = D.Simpson(o1, t1)
        actual = 0.8333
        self.assertAlmostEqual(distance, actual, places=4)

    def testSimpson1(self):
        'Simpson for set (self)'
        distance = D.Simpson(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testRussel_Rao(self):
        'Russel and Rao for set'
        distance = D.Russel_Rao(o1, t1)
        actual = 0.7143
        self.assertAlmostEqual(distance, actual, places=4)

    def testRussel_Rao1(self):
        'Russel and Rao for set (self)'
        distance = D.Russel_Rao(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)
    def testRoger_Tanimoto(self):
        'Roger and Tanimoto for set'
        distance = D.Roger_Tanimoto(o1, t1)
        actual = 0.55555
        self.assertAlmostEqual(distance, actual, places=4)

    def testRoger_Tanimoto1(self):
        'Roger and Tanimoto for set (self)'
        distance = D.Roger_Tanimoto(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Sneath(self):
        'Sokal and Sneath for set'
        distance = D.Sokal_Sneath(o1, t1)
        actual = 0.55555
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Sneath1(self):
        'Sokal and Sneath for set (self)'
        distance = D.Sokal_Sneath(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Sneath2(self):
        'Sokal and Snealth 2 for set'
        distance = D.Sokal_Sneath2(o1, t1)
        actual = 0.8333
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Sneath2_1(self):
        'Sokal and Snealth 2 for set (self)'
        distance = D.Sokal_Sneath2(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Sneath3(self):
        'Sokal and Snealth 3 for set'
        distance = D.Sokal_Sneath3(o1, t1)
        actual = 2.5000
        self.assertAlmostEqual(distance, actual, places=4)

    def testBuser(self):
        'Buser for set'
        distance = D.Buser(o1, t1)
        actual = 0.7835
        self.assertAlmostEqual(distance, actual, places=4)

    def testBuser1(self):
        'Buser for set (self)'
        distance = D.Buser(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testFossum(self):
        'Fossum for set'
        distance = D.Fossum(o1, t1)
        actual = 3.9375
        self.assertAlmostEqual(distance, actual, places=4)
```

```
def testFossum1(self):
    'Fossum for set (self)'
    distance = D.Fossum(o1, o1)
    actual = 5.04167
    self.assertAlmostEqual(distance, actual, places=4)

def testYuleQ(self):
    'Yule Q for set'
    distance = D.YuleQ(o1, t1)
    actual = -1.0000
    self.assertAlmostEqual(distance, actual, places=4)

def testYuleY(self):
    'Yule Y for set'
    distance = D.YuleY(o1, t1)
    actual = -1.0000
    self.assertAlmostEqual(distance, actual, places=4)

def testMcconnaughey(self):
    'McConnaughey for set'
    distance = D.Mcconnaughey(o1, t1)
    actual = 0.6667
    self.assertAlmostEqual(distance, actual, places=4)

def testMcconnaughey1(self):
    'McConnaughey for set (self)'
    distance = D.Mcconnaughey(o1, o1)
    actual = 1.0
    self.assertAlmostEqual(distance, actual, places=4)

def testStiles(self):
    'Stiles for set'
    distance = D.Stiles(o1, t1)
    actual = 0.0847
    self.assertAlmostEqual(distance, actual, places=4)

def testPearson(self):
    'Pearson for set'
    distance = D.Pearson(o1, t1)
    actual = -0.1667
    self.assertAlmostEqual(distance, actual, places=4)

def testDennis(self):
    'Dennis for set'
    distance = D.Dennis(o1, t1)
    actual = -0.0630
    self.assertAlmostEqual(distance, actual, places=4)

def testGower_Legendre(self):
    'Gower and Legendre for set'
    distance = D.Gower_Legendre(o1, t1)
    actual = 0.8333
    self.assertAlmostEqual(distance, actual, places=4)

def testGower_Legendre1(self):
    'Gower and Legendre for set (self)'
    distance = D.Gower_Legendre(o1, o1)
    actual = 1.0
    self.assertAlmostEqual(distance, actual, places=4)
    def testTulloss(self):
        'Tulloss for set'
        distance = D.Tulloss(o1, t1)
        actual = 0.8509
        self.assertAlmostEqual(distance, actual, places=4)

    def testTulloss1(self):
        'Tulloss for set (self)'
        distance = D.Tulloss(o1, o1)
        actual = 1.0
        self.assertAlmostEqual(distance, actual, places=4)


class testList(unittest.TestCase):

    def testJaccard(self):
        'Jaccard for list'
        distance = D.Jaccard(o2, t2, 0, 'List')
        actual = 0.6667
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Michener(self):
        'Sokal and Michener for list'
        distance = D.Sokal_Michener(o2, t2, 0, 'List')
        actual = 0.6667
        self.assertAlmostEqual(distance, actual, places=4)

    def testMatching(self):
        'Matching for list'
        distance = D.Matching(o2, t2, 0, 'List')
        actual = 0.4000
        self.assertAlmostEqual(distance, actual, places=4)

    def testDice(self):
        'Dice for list'
        distance = D.Dice(o2, t2, 0, 'List')
        actual = 0.6667
        self.assertAlmostEqual(distance, actual, places=4)

    def testOchiai(self):
        'Ochiai for list'
        distance = D.Ochiai(o2, t2, 0, 'List')
        actual = 0.8000
        self.assertAlmostEqual(distance, actual, places=4)

    def testOchiai2(self):
        'Ochiai 2 for list'
        distance = D.Ochiai2(o2, t2, 0, 'List')
        actual = 0.0000
        self.assertAlmostEqual(distance, actual, places=4)

    def testAnderberg(self):
        'Anderberg for list'
        distance = D.Anderberg(o2, t2, 0, 'List')
        actual = 0.5000
        self.assertAlmostEqual(distance, actual, places=4)

    def testKulczynski2(self):
        'Kulczynski 2 for list'
        distance = D.Kulczynski2(o2, t2, 0, 'List')
        actual = 0.8000
        self.assertAlmostEqual(distance, actual, places=4)

    def testKulczynski(self):
        'Kulczynski for list'
        distance = D.Kulczynski(o2, t2, 0, 'List')
        actual = 2.0000
        self.assertAlmostEqual(distance, actual, places=4)

    def testForbes(self):
        'Forbes for list'
        distance = D.Forbes(o2, t2, 0, 'List')
        actual = 0.9600
        self.assertAlmostEqual(distance, actual, places=4)

    def testHamann(self):
        'Hamann for list'
        distance = D.Hamann(o2, t2, 0, 'List')
        actual = 0.3333
        self.assertAlmostEqual(distance, actual, places=4)

    def testSimpson(self):
        'Simpson for list'
        distance = D.Simpson(o2, t2, 0, 'List')
        actual = 0.8000
        self.assertAlmostEqual(distance, actual, places=4)

    def testRussel_Rao(self):
        'Russel and Rao for list'
        distance = D.Russel_Rao(o2, t2, 0, 'List')
        actual = 0.6667
        self.assertAlmostEqual(distance, actual, places=4)

    def testRoger_Tanimoto(self):
        'Roger and Tanimoto for list'
        distance = D.Roger_Tanimoto(o2, t2, 0, 'List')
        actual = 0.5000
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Sneath(self):
        'Sokal and Sneath for list'
        distance = D.Sokal_Sneath(o2, t2, 0, 'List')
        actual = 0.5000
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Sneath2(self):
        'Sokal and Sneath 2 for list'
        distance = D.Sokal_Sneath2(o2, t2, 0, 'List')
        actual = 0.8000
        self.assertAlmostEqual(distance, actual, places=4)

    def testSokal_Sneath3(self):
        'Sokal and Sneath 3 for list'
        distance = D.Sokal_Sneath3(o2, t2, 0, 'List')
        actual = 2.0000
        self.assertAlmostEqual(distance, actual, places=4)

    def testBuser(self):
        'Buser for list'
        distance = D.Buser(o2, t2, 0, 'List')
        actual = 0.7500
        self.assertAlmostEqual(distance, actual, places=4)

    def testFossum(self):
        'Fossum for list'
        distance = D.Fossum(o2, t2, 0, 'List')
        actual = 2.9400
        self.assertAlmostEqual(distance, actual, places=4)

    def testYuleQ(self):
        'Yule Q for list'
        distance = D.YuleQ(o2, t2, 0, 'List')
        actual = -1.000
        self.assertAlmostEqual(distance, actual, places=4)

    def testYuleY(self):
        'Yule Y for list'
        distance = D.YuleY(o2, t2, 0, 'List')
        actual = -1.000
        self.assertAlmostEqual(distance, actual, places=4)

    def testMcconnaughey(self):
        'McConnaughey for list'
        distance = D.Mcconnaughey(o2, t2, 0, 'List')
        actual = 0.6000
        self.assertAlmostEqual(distance, actual, places=4)

    def testStiles(self):
        'Stiles for list'
        distance = D.Stiles(o2, t2, 0, 'List')
        actual = -0.0177
        self.assertAlmostEqual(distance, actual, places=4)

    def testPearson(self):
        'Pearson for list'
        distance = D.Pearson(o2, t2, 0, 'List')
        actual = -0.2000
        self.assertAlmostEqual(distance, actual, places=4)

    def testDennis(self):
        'Dennis for list'
        distance = D.Dennis(o2, t2, 0, 'List')
        actual = -0.0816
        self.assertAlmostEqual(distance, actual, places=4)

    def testGower_Legendre(self):
        'Gower and Legendre for list'
        distance = D.Gower_Legendre(o2, t2, 0, 'List')
        actual = 0.8000
        self.assertAlmostEqual(distance, actual, places=4)

    def testTulloss(self):
        'Tulloss for list'
        distance = D.Tulloss(o2, t2, 0, 'List')
        actual = 0.8211
        self.assertAlmostEqual(distance, actual, places=4)

    def testHamming2(self):
        'Hamming for list (Test 1)'
        distance = D.Hamming(o2, t2)
        actual = 2
        self.assertEqual(distance, actual)

    def testHamming3(self):
        'Hamming for list (Test 2)'
        distance = D.Hamming(o3, t3)
        actual = 2
        self.assertEqual(distance, actual)

    def testEuclidean2(self):
        'Euclidean for list (Test 1)'
        distance = D.Euclidean(o2, t2)
        actual = 1.4142
        self.assertAlmostEqual(distance, actual, places=4)

    def testEuclidean3(self):
        'Euclidean for list (Test 2)'
        distance = D.Euclidean(o3, t3)
        actual = 2.2361
        self.assertAlmostEqual(distance, actual, places=4)

    def testMinkowski2(self):
        'Minkowski for list (Test 1)'
        distance = D.Minkowski(o2, t2)
        actual = 1.2599
        self.assertAlmostEqual(distance, actual, places=4)

    def testMinkowski3(self):
        'Minkowski for list (Test 2)'
        distance = D.Minkowski(o3, t3)
        actual = 2.0801
        self.assertAlmostEqual(distance, actual, places=4)

    def testManhattan2(self):
        'Manhattan for list (Test 1)'
        distance = D.Manhattan(o2, t2)
        actual = 2.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testManhattan3(self):
        'Manhattan for list (Test 2)'
        distance = D.Manhattan(o3, t3)
        actual = 3.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testCanberra2(self):
        'Canberra for list (Test 1)'
        distance = D.Canberra(o2, t2)
        actual = 2.0
        self.assertAlmostEqual(distance, actual, places=4)

    def testCanberra3(self):
        'Canberra for list (Test 2)'
        distance = D.Canberra(o3, t3)
        actual = 0.3429
        self.assertAlmostEqual(distance, actual, places=4)

    def testBray_Curtis2(self):
        'Bray Curtis for list (Test 1)'
        distance = D.Bray_Curtis(o2, t2)
        actual = 0.8000
        self.assertAlmostEqual(distance, actual, places=4)

    def testBray_Curtis3(self):
        'Bray Curtis for list (Test 2)'
        distance = D.Bray_Curtis(o3, t3)
        actual = 0.9474
        self.assertAlmostEqual(distance, actual, places=4)

    def testCosine2(self):
        'Cosine for list (Test 1)'
        distance = D.Cosine(o2, t2)
        actual = 0.8000
        self.assertAlmostEqual(distance, actual, places=4)

    def testCosine3(self):
        'Cosine for list (Test 2)'
        distance = D.Cosine(o3, t3)
        actual = 0.9873
        self.assertAlmostEqual(distance, actual, places=4)

    def testTanimoto2(self):
        'Tanimoto for list (Test 1)'
        distance = D.Tanimoto(o2, t2)
        actual = 0.6667
        self.assertAlmostEqual(distance, actual, places=4)

    def testTanimoto(self):
        'Tanimoto for list (Test 2)'
        distance = D.Tanimoto(o3, t3)
        actual = 0.9738
        self.assertAlmostEqual(distance, actual, places=4)


if __name__ == '__main__':
    unittest.main()
```

4. References

Basilevsky, A. 1983. Applied matrix algebra in the statistical sciences. Dover Publications.

Birchfield, S, Tomasi, C. 1998. A pixel dissimilarity measure that is insensitive to image sampling. IEEE Transactions on Pattern Analysis and Machine Intelligence 20: 401-406.

Bray JR, Curtis JT. 1957. An ordination of the upland forest communities of S. Winconsin. Ecological Monographs 27: 325-349.

Dennis, SF. 1965. The construction of a thesaurus automatically from a sample of text. In: Stevens, ME, Guiliano, VE and Heilprin, LB (eds). Statistical association techniques for mechanized documentation: Symposium proceedings. National Bureau of Standards. Miscellaneous publication 269.

Dice, LR. 1945. Measures of the amount of ecologic association between species. Ecology 26: 297-302.

Dunn, G, Everitt, BS. 1982. An introduction to mathematical taxonomy. Dover Publications.

Ellis, D, Furner-Hines, J, Willett, P. 1993. Measuring the degree of similarity between objects in text retrieval systems. Perspectives in Information Management 3(2): 128-149.

Fallaw, WC. 1979. A test of the Simpson coefficient and other binary coefficients of faunal similarity. Journal of Paleontology 53(4):1029.

Fligner, MA, Verducci, JS, Blower, PE. 2002. A modification of the Jaccard-Tanimoto similarity index for diverse selection of chemical compounds using binary strings. Technometrics, May 2002.

Forbes, SA. 1907. On the local distribution of certain Illinois fishes. Bulletin of the Illinois State Laboratory of Natural History 7: 8.

Fossum, EG. 1966. Optimization and standardization of information retrieval language and systems. Springfield VA: Clearinghouse for Federal Scientific and Technical Information.

Gower, JC, Legendre, P. 1986. Metric and Euclidean properties of dissimilarity coefficients. Journal of Classification 5: 5-48.

Hamann, U. 1961. Merkmalbestand und Verwandtschaftsbeziehungen den Farinosae: Ein Betrag zum System der Monokotyledonen. Willdernowia 2: 639-768.

Hamming, RW. 1950. Error detecting and error correcting codes. Bell System Technical Journal 29(2): 147-160.

Holliday, JD, Hu, CY, Willett, P. 2002. Grouping of coefficients for the calculation of inter-molecular similarity and dissimilarity using 2D fragment bit-strings. Combinatorial Chemistry & High Throughput Screening 5(2):155-166

Jaccard, P. 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. Bull. Soc. Vaudoise Sci. Nat. 37: 547-579.

Jaccard, P. 1908. Nouvelles recherches sur la distribution florale. Bull Soc Vaud Sci Nat 44:223-270

Krause, EF. 1987. Taxicab Geometry. Dover. ISBN 0-486-25202-7.

Lance, GN, Williams, WT. 1966. Computer programs for hierarchical polythetic classification. Computer Journal 9: 60-64.

Lee, CH, Oon, JSH, Lee, KC, Ling, MHT. 2010. Bactome, I: Python in DNA Fingerprinting. Proceedings of PyCon Asia-Pacific 2010. The Python Papers Monograph 2.

McConnaughey, BH. 1964. The determination and analysis of plankton communities. Penelitian laut de Indonesia (Marine research in Indonesia), Special number, 1-40.

Ochiai, A. 1957. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. Bull. Jpn. Soc. Sci. Fish. 22: 526-530.

Rogers, DJ. and Tanimoto, TT. 1960. A computer program for classifying plants. Science 132: 1115-1118.

Rossi, F, De Carvalho, F, Lechevallier, Y, Da Silva, A. 2006. Dissimilarities for web usage mining. Data Science and Classification. Springer, Berlin Heidelberg.

Russel, PF, Rao, TR. 1940. On habitat and association of species of anopheline larvae in south-eastern Madras. J. Malaria Inst. India 3: 153-178.

Snarey, M, Terrett, NK, Willett, P, Wilton, DJ. 1998. Comparison of algorithms for dissimilarity-based compound selection. Journal of Molecular Graphics and Modelling 15: 372-385.

Sokal, RR, Michener, CD 1958. A statistical method for evaluating systematic relationships. Univ Kansas Sci Bull 38:1409-1438.

Sokal, RR, Sneath, PHA. 1963. Principles of Numeric Taxonomy. W.H. Freeman, San Francisco.

Sorensen, T. 1948. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons. K. Dan. Vidensk. Selsk. Biol. Skr. 5: 1-34

Stiles, W. 1923. The indicator method for the determination of coefficients of diffusion in gels, with special reference to the diffusion of chlorides. Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character 103(721): 260-275.

Tanimoto, TT. 1958. An elementary mathematical theory of classification and prediction. IBM Internal Report, November 17.

Tulloss, RE. 1997. Assessment of similarity indices for undesirable properties and a new tripartite similarity index based on cost functions. In: Palm, M. E. and I. H. Chapela, eds. Mycology in Sustainable Development: Expanding Concepts, Vanishing Borders. Parkway Publishers, Boone, North Carolina.

Werle, SF, Johnson, NA, Elizabeth R. Dumont, ER, Parasiewicz, P. 2006. Ecological Dissimilarity Analysis: A Simple Method of Demonstrating Community-Habitat Correlations for Frequency Data. Northeastern Naturalist 14: 439-446.

Yule, GU. 1912. On the methods of measuring association between two attributes. Journal of the Royal Statistical Society LXXV: 579-652.

Zhang, B, Srihani, SN. 2003. Binary vector dissimilarity measures for handwriting identification. Proceedings of SPIE-IS&T Electronic Imaging. SPIE Volume 5010.