Record Linkage Toolkit Documentation

Release 0.14

Jonathan de Bruin

Dec 04, 2019

First steps

All you need to start linking records.

CHAPTER 1

About

1.1 Introduction

The **Python Record Linkage Toolkit** is a library to link records in or between data sources. The toolkit provides most of the tools needed for record linkage and deduplication. The package contains indexing methods, functions to compare records and classifiers. The package is developed for research and the linking of small or medium sized files.

The project is inspired by the Freely Extensible Biomedical Record Linkage (FEBRL) project, which is a great project. In contrast with FEBRL, the recordlinkage project makes extensive use of data manipulation tools like pandas and numpy. The use of *pandas*, a flexible and powerful data analysis and manipulation library for Python, makes the record linkage process much easier and faster. The extensive *pandas* library can be used to integrate your record linkage directly into existing data manipulation projects.

One of the aims of this project is to make an extensible record linkage framework. It is easy to include your own indexing algorithms, comparison/similarity measures and classifiers. The main features of the Python Record Linkage Toolkit are:

- Clean and standardise data with easy to use tools
- Make pairs of records with smart indexing methods such as **blocking** and **sorted neighbourhood indexing**
- Compare records with a large number of comparison and similarity measures for different types of variables such as strings, numbers and dates.
- Several classifications algorithms, both supervised and unsupervised algorithms.
- Common record linkage evaluation tools
- Several built-in datasets.

1.2 What is record linkage?

The term record linkage is used to indicate the procedure of bringing together information from two or more records that are believed to belong to the same entity. Record linkage is used to link data from multiple data sources or to find

duplicates in a single data source. In computer science, record linkage is also known as data matching or deduplication (in case of search duplicate records within a single file).

In record linkage, the attributes of the entity (stored in a record) are used to link two or more records. Attributes can be unique entity identifiers (SSN, license plate number), but also attributes like (sur)name, date of birth and car model/colour. The record linkage procedure can be represented as a workflow [Christen, 2012]. The steps are: cleaning, indexing, comparing, classifying and evaluation. If needed, the classified record pairs flow back to improve the previous step. The Python Record Linkage Toolkit follows this workflow.

See also:

Christen, Peter. 2012. Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer Science & Business Media.

Fellegi, Ivan P and Alan B Sunter. 1969. "A theory for record linkage." Journal of the American Statistical Association 64(328):1183–1210.

Dunn, Halbert L. 1946. "Record linkage." American Journal of Public Health and the Nations Health 36(12):1412–1416.

Herzog, Thomas N, Fritz J Scheuren and William E Winkler. 2007. Data quality and record linkage techniques. Vol. 1 Springer.

1.3 How to link records?

Import the `recordlinkage` module with all important tools for record linkage and import the data manipulation framework **pandas**.

```
import recordlinkage
import pandas
```

Consider that you try to link two datasets with personal information like name, sex and date of birth. Load these datasets into a pandas `DataFrame`.

```
df_a = pandas.DataFrame(YOUR_FIRST_DATASET)
df_b = pandas.DataFrame(YOUR_SECOND_DATASET)
```

Comparing all record can be computationally intensive. Therefore, we make smart set of candidate links with one of the built-in indexing techniques like **blocking**. Only records pairs agreeing on the surname are included.

```
indexer = recordlinkage.Index()
indexer.block('surname')
candidate_links = indexer.index(df_a, df_b)
```

Each `candidate_link` needs to be compared on the comparable attributes. This can be done easily with the Compare class and the available comparison and similarity measures.

```
compare = recordlinkage.Compare()

compare.string('name', 'name', method='jarowinkler', threshold=0.85)
compare.exact('sex', 'gender')
compare.exact('dob', 'date_of_birth')
compare.string('streetname', 'streetname', method='damerau_levenshtein', threshold=0.
↪7)
compare.exact('place', 'placename')
compare.exact('haircolor', 'haircolor', missing_value=9)
```

(continues on next page)

(continued from previous page)

```
# The comparison vectors
compare_vectors = compare.compute(candidate_links, df_a, df_b)
```

This record linkage package contains several classification algorithms. Plenty of the algorithms need trainings data (supervised learning) while some others are unsupervised. An example of supervised learning:

```
true_linkage = pandas.Series(YOUR_GOLDEN_DATA, index=pandas.MultiIndex(YOUR_MULTI_
↪INDEX))

logrg = recordlinkage.LogisticRegressionClassifier()
logrg.fit(compare_vectors[true_linkage.index], true_linkage)

logrg.predict(compare_vectors)
```

and an example of unsupervised learning (the well known ECM-algorithm):

```
ecm = recordlinkage.BernoulliEMClassifier()
ecm.fit_predict(compare_vectors)
```

CHAPTER 2

Installation guide

2.1 Python version support

The Python Record Linkage Toolkit supports the versions of Python that Pandas supports as well. You can find the supported Python versions in the Pandas documentation.

2.2 Installation

The Python Record linkage Toolkit requires Python 3.5 or higher (since version >= 0.14). Install the package easily with pip

```
pip install recordlinkage
```

Python 2.7 users can use version <= 0.13, but it is advised to use Python >= 3.5.

You can also clone the project on Github. The license of this record linkage package is BSD-3-Clause.

2.3 Dependencies

The following packages are required. You probably have most of them already ;)

- numpy
- pandas (>=0.18.0)
- scipy
- sklearn
- jellyfish: Needed for approximate string comparison and string encoding.
- numexpr (optional): Used to speed up numeric comparisons.

CHAPTER 3

Link two datasets

3.1 Introduction

This example shows how two datasets with data about persons can be linked. We will try to link the data based on attributes like first name, surname, sex, date of birth, place and address. The data used in this example is part of Febrl and is fictitious.

First, start with importing the `recordlinkage` module. The submodule `recordlinkage.datasets` contains several datasets that can be used for testing. For this example, we use the Febrl datasets 4A and 4B. These datasets can be loaded with the function `load_febrl4`.

[2]:
```
import recordlinkage
from recordlinkage.datasets import load_febrl4
```

The datasets are loaded with the following code. The returned datasets are of type `pandas.DataFrame`. This makes it easy to manipulate the data if desired. For details about data manipulation with `pandas`, see their comprehensive documentation http://pandas.pydata.org/.

[3]:
```
dfA, dfB = load_febrl4()

dfA
```

[3]:

rec_id	given_name	surname	street_number	address_1	\
rec-1070-org	michaela	neumann	8	stanley street	
rec-1016-org	courtney	painter	12	pinkerton circuit	
rec-4405-org	charles	green	38	salkauskas crescent	
rec-1288-org	vanessa	parr	905	macquoid place	
rec-3585-org	mikayla	malloney	37	randwick road	
...	...	...	...	...	
rec-2153-org	annabel	grierson	97	mclachlan crescent	
rec-1604-org	sienna	musolino	22	smeaton circuit	
rec-1003-org	bradley	matthews	2	jondol place	
rec-4883-org	brodee	egan	88	axon street	

(continues on next page)

(continued from previous page)

```
rec-66-org        koula  houweling             3       mileham street

                        address_2            suburb postcode state  \
rec_id
rec-1070-org                miami     winston hills     4223   nsw
rec-1016-org           bega flats         richlands     4560   vic
rec-4405-org                kela             dapto     4566   nsw
rec-1288-org    broadbridge manor     south grafton     2135    sa
rec-3585-org              avalind  hoppers crossing     4552   vic
...                           ...               ...      ...   ...
rec-2153-org        lantana lodge            broome     2480   nsw
rec-1604-org              pangani          mckinnon     2700   nsw
rec-1003-org         horseshoe ck       jacobs well     7018    sa
rec-4883-org          greenslopes          wamberal     2067   qld
rec-66-org    old airdmillan road      williamstown     2350   nsw

             date_of_birth soc_sec_id
rec_id
rec-1070-org      19151111    5304218
rec-1016-org      19161214    4066625
rec-4405-org      19480930    4365168
rec-1288-org      19951119    9239102
rec-3585-org      19860208    7207688
...                    ...        ...
rec-2153-org      19840224    7676186
rec-1604-org      19890525    4971506
rec-1003-org      19481122    8927667
rec-4883-org      19121113    6039042
rec-66-org        19440718    6375537

[5000 rows x 10 columns]
```

3.2 Make record pairs

It is very intuitive to compare each record in DataFrame `dfA` with all records of DataFrame `dfB`. In fact, we want to make record pairs. Each record pair should contain one record of `dfA` and one record of `dfB`. This process of making record pairs is also called 'indexing'. With the `recordlinkage` module, indexing is easy. First, load the `index.Index` class and call the `.full` method. This object generates a full index on a `.index(...)` call. In case of deduplication of a single dataframe, one dataframe is sufficient as argument.

[4]:
```
indexer = recordlinkage.Index()
indexer.full()
pairs = indexer.index(dfA, dfB)
```

```
WARNING:recordlinkage:indexing - performance warning - A full index can result in␣
↪large number of record pairs.
```

With the method `index`, all possible (and unique) record pairs are made. The method returns a `pandas.MultiIndex`. The number of pairs is equal to the number of records in `dfA` times the number of records in `dfB`.

[5]:
```
print (len(dfA), len(dfB), len(pairs))
```

```
5000 5000 25000000
```

Many of these record pairs do not belong to the same person. In case of one-to-one matching, the number of matches

should be no more than the number of records in the smallest dataframe. In case of full indexing, `min(len(dfA), len(N_dfB))` is much smaller than `len(pairs)`. The `recordlinkage` module has some more advanced indexing methods to reduce the number of record pairs. Obvious non-matches are left out of the index. Note that if a matching record pair is not included in the index, it can not be matched anymore.

One of the most well known indexing methods is named *blocking*. This method includes only record pairs that are identical on one or more stored attributes of the person (or entity in general). The blocking method can be used in the `recordlinkage` module.

[6]:
```
indexer = recordlinkage.Index()
indexer.block('given_name')
candidate_links = indexer.index(dfA, dfB)

print (len(candidate_links))
```

```
77249
```

The argument 'given_name' is the blocking variable. This variable has to be the name of a column in `dfA` and `dfB`. It is possible to parse a list of columns names to block on multiple variables. Blocking on multiple variables will reduce the number of record pairs even further.

Another implemented indexing method is *Sorted Neighbourhood Indexing* (`recordlinkage.index.SortedNeighbourhood`). This method is very useful when there are many misspellings in the string were used for indexing. In fact, sorted neighbourhood indexing is a generalisation of blocking. See the documentation for details about sorted neighbourd indexing.

3.3 Compare records

Each record pair is a candidate match. To classify the candidate record pairs into matches and non-matches, compare the records on all attributes both records have in common. The `recordlinkage` module has a class named `Compare`. This class is used to compare the records. The following code shows how to compare attributes.

[7]:
```
# This cell can take some time to compute.
compare_cl = recordlinkage.Compare()

compare_cl.exact('given_name', 'given_name', label='given_name')
compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, label=
↪'surname')
compare_cl.exact('date_of_birth', 'date_of_birth', label='date_of_birth')
compare_cl.exact('suburb', 'suburb', label='suburb')
compare_cl.exact('state', 'state', label='state')
compare_cl.string('address_1', 'address_1', threshold=0.85, label='address_1')

features = compare_cl.compute(candidate_links, dfA, dfB)
```

The comparing of record pairs starts when the `compute` method is called. All attribute comparisons are stored in a DataFrame with horizontally the features and vertically the record pairs.

[8]:
```
features
```

[8]:

		given_name	surname	date_of_birth	suburb	\
rec_id_1	rec_id_2					
rec-1070-org	rec-3024-dup-0	1	0.0	0	0	
	rec-2371-dup-0	1	0.0	0	0	
	rec-4652-dup-0	1	0.0	0	0	
	rec-4795-dup-0	1	0.0	0	0	

(continues on next page)

(continued from previous page)

```
             rec-1314-dup-0           1      0.0              0       0
...                                   ...      ...            ...     ...
rec-4528-org rec-4528-dup-0           1      1.0              1       1
rec-4887-org rec-4887-dup-0           1      1.0              1       0
rec-4350-org rec-4350-dup-0           1      1.0              1       1
rec-4569-org rec-4569-dup-0           1      1.0              1       1
rec-3125-org rec-3125-dup-0           1      1.0              1       0

                              state  address_1
rec_id_1     rec_id_2
rec-1070-org rec-3024-dup-0       1        0.0
             rec-2371-dup-0       0        0.0
             rec-4652-dup-0       0        0.0
             rec-4795-dup-0       1        0.0
             rec-1314-dup-0       1        0.0
...                             ...        ...
rec-4528-org rec-4528-dup-0       1        1.0
rec-4887-org rec-4887-dup-0       1        1.0
rec-4350-org rec-4350-dup-0       1        1.0
rec-4569-org rec-4569-dup-0       1        0.0
rec-3125-org rec-3125-dup-0       1        1.0

[77249 rows x 6 columns]
```

[9]:
```
features.describe()
```

[9]:
```
       given_name      surname  date_of_birth       suburb        state  \
count     77249.0  77249.00000    77249.00000  77249.00000  77249.00000
mean          1.0      0.04443        0.03793      0.03226      0.24877
std           0.0      0.20604        0.19103      0.17669      0.43230
min           1.0      0.00000        0.00000      0.00000      0.00000
25%           1.0      0.00000        0.00000      0.00000      0.00000
50%           1.0      0.00000        0.00000      0.00000      0.00000
75%           1.0      0.00000        0.00000      0.00000      0.00000
max           1.0      1.00000        1.00000      1.00000      1.00000

         address_1
count  77249.00000
mean       0.03670
std        0.18802
min        0.00000
25%        0.00000
50%        0.00000
75%        0.00000
max        1.00000
```

The last step is to decide which records belong to the same person. In this example, we keep it simple:

[10]:
```
# Sum the comparison results.
features.sum(axis=1).value_counts().sort_index(ascending=False)
```

[10]:
```
6.0     1566
5.0     1332
4.0      343
3.0      146
2.0    16427
1.0    57435
dtype: int64
```

[11]:
```
features[features.sum(axis=1) > 3]
```

[11]:
```
                              given_name  surname  date_of_birth  suburb  \
rec_id_1     rec_id_2
rec-2371-org rec-2371-dup-0           1      1.0              1       1
rec-3024-org rec-3024-dup-0           1      1.0              1       0
rec-4652-org rec-4652-dup-0           1      1.0              1       0
rec-4795-org rec-4795-dup-0           1      1.0              1       1
rec-1016-org rec-1016-dup-0           1      1.0              1       1
...                                 ...      ...            ...     ...
rec-4528-org rec-4528-dup-0           1      1.0              1       1
rec-4887-org rec-4887-dup-0           1      1.0              1       0
rec-4350-org rec-4350-dup-0           1      1.0              1       1
rec-4569-org rec-4569-dup-0           1      1.0              1       1
rec-3125-org rec-3125-dup-0           1      1.0              1       0

                              state  address_1
rec_id_1     rec_id_2
rec-2371-org rec-2371-dup-0       1        1.0
rec-3024-org rec-3024-dup-0       1        0.0
rec-4652-org rec-4652-dup-0       1        1.0
rec-4795-org rec-4795-dup-0       1        1.0
rec-1016-org rec-1016-dup-0       0        1.0
...                             ...        ...
rec-4528-org rec-4528-dup-0       1        1.0
rec-4887-org rec-4887-dup-0       1        1.0
rec-4350-org rec-4350-dup-0       1        1.0
rec-4569-org rec-4569-dup-0       1        0.0
rec-3125-org rec-3125-dup-0       1        1.0

[3241 rows x 6 columns]
```

3.4 Full code

[12]:
```
import recordlinkage
from recordlinkage.datasets import load_febrl4

dfA, dfB = load_febrl4()

# Indexation step
indexer = recordlinkage.Index()
indexer.block('given_name')
candidate_links = indexer.index(dfA, dfB)

# Comparison step
compare_cl = recordlinkage.Compare()

compare_cl.exact('given_name', 'given_name', label='given_name')
compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, label=
↪'surname')
compare_cl.exact('date_of_birth', 'date_of_birth', label='date_of_birth')
compare_cl.exact('suburb', 'suburb', label='suburb')
compare_cl.exact('state', 'state', label='state')
compare_cl.string('address_1', 'address_1', threshold=0.85, label='address_1')
```

(continues on next page)

(continued from previous page)

```
features = compare_cl.compute(candidate_links, dfA, dfB)

# Classification step
matches = features[features.sum(axis=1) > 3]
print(len(matches))
```

```
3241
```

CHAPTER 4

Data deduplication

4.1 Introduction

This example shows how to find records in datasets belonging to the same entity. In our case,we try to deduplicate a dataset with records of persons. We will try to link within the dataset based on attributes like first name, surname, sex, date of birth, place and address. The data used in this example is part of Febrl and is fictitious.

First, start with importing the `recordlinkage` module. The submodule `recordlinkage.datasets` contains several datasets that can be used for testing. For this example, we use the Febrl dataset 1. This dataset contains 1000 records of which 500 original and 500 duplicates, with exactly one duplicate per original record. This dataset can be loaded with the function `load_febrl1`.

[1]:
```
import recordlinkage
from recordlinkage.datasets import load_febrl1
```

The dataset is loaded with the following code. The returned datasets are of type `pandas.DataFrame`. This makes it easy to manipulate the data if desired. For details about data manipulation with `pandas`, see their comprehensive documentation http://pandas.pydata.org/.

[2]:
```
dfA = load_febrl1()

dfA.head()
```

[2]:

rec_id	given_name	surname	street_number	address_1	address_2	suburb	postcode	state	date_of_birth	soc_sec_id
rec-223-org	NaN	waller	6	tullaroop street	willaroo	st james	4011	wa	19081209	6988048
rec-122-org	lachlan	berry	69	giblin street						
rec-373-org	deakin	sondergeld	48	goldfinch circuit						
rec-10-dup-0	kayla	harrington	NaN	maltby circuit						
rec-227-org	luke	purdon	23	ramsay place						

(continues on next page)

(continued from previous page)

```
rec-122-org   killarney     bittern     4814   qld      19990219    7364009
rec-373-org     kooltuo  canterbury     2776   vic      19600210    2635962
rec-10-dup-0    coaling    coolaroo     3465   nsw      19150612    9004242
rec-227-org      mirani     garbutt     2260   vic      19831024    8099933
```

4.2 Make record pairs

It is very intuitive to start with comparing each record in DataFrame `dfA` with all other records in DataFrame `dfA`. In fact, we want to make record pairs. Each record pair should contain two different records of DataFrame `dfA`. This process of making record pairs is also called 'indexing'. With the `recordlinkage` module, indexing is easy. First, load the `recordlinkage.Index` class and call the `.full` method. This object generates a full index on a `.index(...)` call. In case of deduplication of a single dataframe, one dataframe is sufficient as input argument.

[3]:
```
indexer = recordlinkage.Index()
indexer.full()
candidate_links = indexer.index(dfA)
```

```
WARNING:recordlinkage:indexing - performance warning - A full index can result in␣
↪large number of record pairs.
```

With the method `index`, all possible (and unique) record pairs are made. The method returns a `pandas.MultiIndex`. The number of pairs is equal to the number of records in `dfA` times the number of records in `dfB`.

[4]:
```
print (len(dfA), len(candidate_links))
# (1000*1000-1000)/2 = 499500
```

```
1000 499500
```

Many of these record pairs do not belong to the same person. The `recordlinkage` toolkit has some more advanced indexing methods to reduce the number of record pairs. Obvious non-matches are left out of the index. Note that if a matching record pair is not included in the index, it can not be matched anymore.

One of the most well known indexing methods is named *blocking*. This method includes only record pairs that are identical on one or more stored attributes of the person (or entity in general). The blocking method can be used in the `recordlinkage` module.

[6]:
```
indexer = recordlinkage.Index()
indexer.block('given_name')
candidate_links = indexer.index(dfA)

print (len(candidate_links))
```

```
2082
```

The argument 'given_name' is the blocking variable. This variable has to be the name of a column in `dfA` and `dfB`. It is possible to parse a list of columns names to block on multiple variables. Blocking on multiple variables will reduce the number of record pairs even further.

Another implemented indexing method is *Sorted Neighbourhood Indexing* (`recordlinkage.index.sortedneighbourhood`). This method is very useful when there are many misspellings in the string were used for indexing. In fact, sorted neighbourhood indexing is a generalisation of blocking. See the documentation for details about sorted neighbourd indexing.

4.3 Compare records

Each record pair is a candidate match. To classify the candidate record pairs into matches and non-matches, compare the records on all attributes both records have in common. The `recordlinkage` module has a class named `Compare`. This class is used to compare the records. The following code shows how to compare attributes.

[6]:
```
# This cell can take some time to compute.
compare_cl = recordlinkage.Compare()

compare_cl.exact('given_name', 'given_name', label='given_name')
compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, label=
↪'surname')
compare_cl.exact('date_of_birth', 'date_of_birth', label='date_of_birth')
compare_cl.exact('suburb', 'suburb', label='suburb')
compare_cl.exact('state', 'state', label='state')
compare_cl.string('address_1', 'address_1', threshold=0.85, label='address_1')

features = compare_cl.compute(pairs, dfA)
```

The comparing of record pairs starts when the `compute` method is called. All attribute comparisons are stored in a DataFrame with horizontally the features and vertically the record pairs. The first 10 comparison vectors are:

[7]:
```
features.head(10)
```

[7]:

		given_name	surname	date_of_birth	suburb	\
rec_id	rec_id					
rec-122-org	rec-183-dup-0	1	0.0	0	0	
	rec-248-org	1	0.0	0	0	
	rec-469-org	1	0.0	0	0	
	rec-74-org	1	0.0	0	0	
	rec-183-org	1	0.0	0	0	
	rec-360-dup-0	1	0.0	0	0	
	rec-248-dup-0	1	0.0	0	0	
	rec-469-dup-0	1	0.0	0	0	
rec-183-dup-0	rec-248-org	1	0.0	0	0	
	rec-469-org	1	0.0	0	0	

		state	address_1
rec_id	rec_id		
rec-122-org	rec-183-dup-0	0	0.0
	rec-248-org	1	0.0
	rec-469-org	0	0.0
	rec-74-org	0	0.0
	rec-183-org	0	0.0
	rec-360-dup-0	0	0.0
	rec-248-dup-0	1	0.0
	rec-469-dup-0	0	0.0
rec-183-dup-0	rec-248-org	0	0.0
	rec-469-org	1	0.0

[8]:
```
features.describe()
```

[8]:

	given_name	surname	date_of_birth	suburb	state	\
count	2082.0	2082.000000	2082.000000	2082.000000	2082.000000	
mean	1.0	0.144092	0.139289	0.108549	0.327089	
std	0.0	0.351268	0.346331	0.311148	0.469263	
min	1.0	0.000000	0.000000	0.000000	0.000000	

(continues on next page)

(continued from previous page)

```
25%           1.0     0.000000       0.000000     0.000000     0.000000
50%           1.0     0.000000       0.000000     0.000000     0.000000
75%           1.0     0.000000       0.000000     0.000000     1.000000
max           1.0     1.000000       1.000000     1.000000     1.000000

         address_1
count  2082.000000
mean      0.133045
std       0.339705
min       0.000000
25%       0.000000
50%       0.000000
75%       0.000000
max       1.000000
```

The last step is to decide which records belong to the same person. In this example, we keep it simple:

[9]:
```
# Sum the comparison results.
features.sum(axis=1).value_counts().sort_index(ascending=False)
```

[9]:
```
6.0     142
5.0     145
4.0      30
3.0       9
2.0     376
1.0    1380
dtype: int64
```

[10]:
```
matches = features[features.sum(axis=1) > 3]

print(len(matches))
matches.head(10)
```

```
317
```

[10]:
```
                          given_name  surname  date_of_birth  suburb  state  \
rec_id        rec_id
rec-183-dup-0 rec-183-org           1      1.0              1       1      1
rec-122-dup-0 rec-122-org           1      1.0              1       1      1
rec-248-dup-0 rec-248-org           1      1.0              1       1      1
rec-373-dup-0 rec-373-org           1      1.0              1       1      1
rec-10-dup-0  rec-10-org            1      1.0              1       1      1
rec-342-dup-0 rec-342-org           1      1.0              0       1      1
rec-330-dup-0 rec-330-org           1      0.0              1       1      1
rec-397-dup-0 rec-397-org           1      1.0              1       1      1
rec-472-dup-0 rec-472-org           1      1.0              1       1      1
rec-190-dup-0 rec-190-org           1      1.0              0       1      1

                          address_1
rec_id        rec_id
rec-183-dup-0 rec-183-org        1.0
rec-122-dup-0 rec-122-org        1.0
rec-248-dup-0 rec-248-org        1.0
rec-373-dup-0 rec-373-org        1.0
rec-10-dup-0  rec-10-org         1.0
rec-342-dup-0 rec-342-org        1.0
rec-330-dup-0 rec-330-org        0.0
rec-397-dup-0 rec-397-org        0.0
```

(continues on next page)

(continued from previous page)

```
rec-472-dup-0 rec-472-org        0.0
rec-190-dup-0 rec-190-org        1.0
```

4.4 Full code

[7]:
```
import recordlinkage
from recordlinkage.datasets import load_febrl1

dfA = load_febrl1()

# Indexation step
indexer = recordlinkage.Index()
indexer.block(left_on='given_name')
candidate_links = indexer.index(dfA)

# Comparison step
compare_cl = recordlinkage.Compare()

compare_cl.exact('given_name', 'given_name', label='given_name')
compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, label=
↪'surname')
compare_cl.exact('date_of_birth', 'date_of_birth', label='date_of_birth')
compare_cl.exact('suburb', 'suburb', label='suburb')
compare_cl.exact('state', 'state', label='state')
compare_cl.string('address_1', 'address_1', threshold=0.85, label='address_1')

features = compare_cl.compute(candidate_links, dfA)

# Classification step
matches = features[features.sum(axis=1) > 3]
print(len(matches))
```

```
317
```

CHAPTER 5

0. Preprocessing

Preprocessing data, like cleaning and standardising, may increase your record linkage accuracy. The Python Record Linkage Toolkit contains several tools for data preprocessing. The preprocessing and standardising functions are available in the submodule *recordlinkage.preprocessing*. Import the algorithms in the following way:

```
from recordlinkage.preprocessing import clean, phonetic
```

5.1 Cleaning

The Python Record Linkage Toolkit has some cleaning function from which *recordlinkage.preprocessing.clean()* is the most generic function. Pandas itself is also very usefull for (string) data cleaning. See the pandas documentation on this topic: Working with Text Data.

recordlinkage.preprocessing.**clean**(*s*, *lowercase=True*, *replace_by_none='[^ \\-_A-Za-z0-9]+'*, *replace_by_whitespace='[\\-_]'*, *strip_accents=None*, *remove_brackets=True*, *encoding='utf-8'*, *decode_error='strict'*)

Clean string variables.

Clean strings in the Series by removing unwanted tokens, whitespace and brackets.

Parameters

- **s** (*pandas.Series*) – A Series to clean.
- **lower** (*bool, optional*) – Convert strings in the Series to lowercase. Default True.
- **replace_by_none** (*str, optional*) – The matches of this regular expression are replaced by ‘’.
- **replace_by_whitespace** (*str, optional*) – The matches of this regular expression are replaced by a whitespace.
- **remove_brackets** (*bool, optional*) – Remove all content between brackets and the bracket themselves. Default True.

- **strip_accents** (*{'ascii', 'unicode', None}, optional*) – Remove accents during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.
- **encoding** (*str, optional*) – If bytes are given, this encoding is used to decode. Default is 'utf-8'.
- **decode_error** (*{'strict', 'ignore', 'replace'}, optional*) – Instruction on what to do if a byte Series is given that contains characters not of the given *encoding*. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

Example

```
>>> import pandas
>>> from recordlinkage.preprocessing import clean
>>>
>>> names = ['Mary-ann',
             'Bob :)',
             'Angel',
             'Bob (alias Billy)',
             None]
>>> s = pandas.Series(names)
>>> print(clean(s))
0    mary ann
1         bob
2       angel
3         bob
4         NaN
dtype: object
```

Returns *pandas.Series* – A cleaned Series of strings.

recordlinkage.preprocessing.**phonenumbers**(*s*)

Clean phonenumbers by removing all non-numbers (except +).

Parameters **s** (*pandas.Series*) – A Series to clean.

Returns *pandas.Series* – A Series with cleaned phonenumbers.

recordlinkage.preprocessing.**value_occurence**(*s*)

Count the number of times each value occurs.

This function returns the counts for each row, in contrast with pandas.value_counts.

Returns *pandas.Series* – A Series with value counts.

5.2 Phonetic encoding

Phonetic algorithms are algorithms for indexing of words by their pronunciation. The most well-known algorithm is the Soundex algorithm. The Python Record Linkage Toolkit supports multiple algorithms through the *recordlinkage.preprocessing.phonetic()* function.

Note: Use phonetic algorithms in advance of the indexing and comparing step. This results in most siutations in better performance.

recordlinkage.preprocessing.**phonetic**(*s*, *method*, *concat=True*, *encoding='utf-8'*, *decode_error='strict'*)

Convert names or strings into phonetic codes.

The implemented algorithms are soundex, nysiis, metaphone or match_rating.

Parameters

- **s** (*pandas.Series*) – A pandas.Series with string values (often names) to encode.
- **method** (*str*) – The algorithm that is used to phonetically encode the values. The possible options are "soundex", "nysiis", "metaphone" or "match_rating".
- **concat** (*bool, optional*) – Remove whitespace before phonetic encoding.
- **encoding** (*str, optional*) – If bytes are given, this encoding is used to decode. Default is 'utf-8'.
- **decode_error** (*{'strict', 'ignore', 'replace'}, optional*) – Instruction on what to do if a byte Series is given that contains characters not of the given *encoding*. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

Returns *pandas.Series* – A Series with phonetic encoded values.

preprocessing.**phonetic_algorithms = ['soundex', 'nysiis', 'metaphone', 'match_rating']**

CHAPTER 6

1. Indexing

The indexing module is used to make pairs of records. These pairs are called candidate links or candidate matches. There are several indexing algorithms available such as blocking and sorted neighborhood indexing. See the following references for background information about indexation.

The indexing module can be used for both linking and duplicate detection. In case of duplicate detection, only pairs in the upper triangular part of the matrix are returned. This means that the first record in each record pair is the largest identifier. For example, *("A2", "A1")*, *(5, 2)* and *("acb", "abc")*. The following image shows the record pairs for a complete set of record pairs.

6.1 `recordlinkage.Index` object

class recordlinkage.**Index**(*algorithms=[]*)

Class to make an index of record pairs.

Parameters **algorithms** (*list*) – A list of index algorithm classes. The classes are based on *recordlinkage.base.BaseIndexAlgorithm*

Example

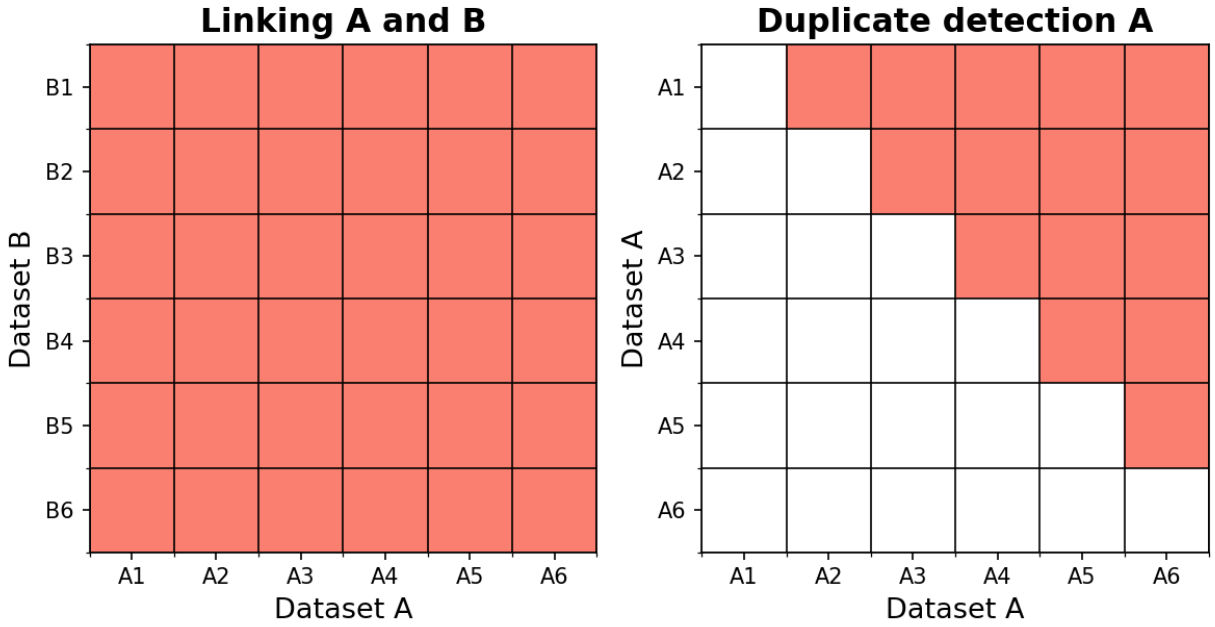
Consider two historical datasets with census data to link. The datasets are named `census_data_1980` and `census_data_1990`:

```
indexer = recordlinkage.Index()
indexer.block(left_on='first_name', right_on='givenname')
indexer.index(census_data_1980, census_data_1990)
```

add(*model*)

Add a index method.

This method is used to add index algorithms. If multiple algorithms are added, the union of the record pairs from the algorithm is taken.



Parameters **model** (*list, class*) – A (list of) index algorithm(s) from *recordlinkage.index*.

index(*x*, *x_link=None*)

Make an index of record pairs.

Parameters

- **x** (*pandas.DataFrame*) – A pandas DataFrame. When *x_link* is None, the algorithm makes record pairs within the DataFrame. When *x_link* is not empty, the algorithm makes pairs between *x* and *x_link*.
- **x_link** (*pandas.DataFrame, optional*) – A second DataFrame to link with the DataFrame x.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

full()

Add a 'full' index.

Shortcut of *recordlinkage.index.Full*:

```
from recordlinkage.index import Full

indexer = recordlinkage.Index()
indexer.add(Full())
```

block(**args*, ***kwargs*)

Add a block index.

Shortcut of *recordlinkage.index.Block*:

```
from recordlinkage.index import Block

indexer = recordlinkage.Index()
indexer.add(Block())
```

sortedneighbourhood(**args*, ***kwargs*)
Add a Sorted Neighbourhood Index.

Shortcut of *recordlinkage.index.SortedNeighbourhood*:

```
from recordlinkage.index import SortedNeighbourhood

indexer = recordlinkage.Index()
indexer.add(SortedNeighbourhood())
```

random(**args*, ***kwargs*)
Add a random index.

Shortcut of *recordlinkage.index.Random*:

```
from recordlinkage.index import Random

indexer = recordlinkage.Index()
indexer.add(Random())
```

6.2 Algorithms

The Python Record Linkage Toolkit contains basic and advanced indexing (or blocking) algorithms to make record pairs. The algorithms are Python classes. Popular algorithms in the toolkit are:

- *recordlinkage.index.Full*,
- *recordlinkage.index.Block*,
- *recordlinkage.index.SortedNeighbourhood*

The algorithms are available in the submodule *recordlinkage.index*. Import the algorithms in the following way (use blocking algorithm as example):

```
from recordlinkage.index import Block
```

The full reference for the indexing algorithms in the toolkit is given below.

class recordlinkage.index.**Full**(***kwargs*)
Class to generate a 'full' index.

A full index is an index with all possible combinations of record pairs. In case of linking, this indexation method generates the cartesian product of both DataFrame's. In case of deduplicating DataFrame A, this indexation method are the pairs defined by the upper triangular matrix of the A x A.

Parameters ****kwargs** – Additional keyword arguments to pass to *recordlinkage.base.BaseIndexAlgorithm*.

Note: This indexation method can be slow for large DataFrame's. The number of comparisons scales quadratic. Also, not all classifiers work well with large numbers of record pairs were most of the pairs are distinct.

---

index(*x*, *x_link=None*)
Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters

- **x** (*pandas.DataFrame*) – A pandas DataFrame. When *x_link* is None, the algorithm makes record pairs within the DataFrame. When *x_link* is not empty, the algorithm makes pairs between *x* and *x_link*.
- **x_link** (*pandas.DataFrame, optional*) – A second DataFrame to link with the DataFrame x.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

class recordlinkage.index.**Block**(*left_on=None*, *right_on=None*, ***kwargs*)
Make candidate record pairs that agree on one or more variables.

Returns all record pairs that agree on the given variable(s). This method is known as *blocking*. Blocking is an effective way to make a subset of the record space (A * B).

Parameters

- **left_on** (*label, optional*) – A column name or a list of column names of dataframe A. These columns are used to block on.
- **right_on** (*label, optional*) – A column name or a list of column names of dataframe B. These columns are used to block on. If 'right_on' is None, the *left_on* value is used. Default None.
- ****kwargs** – Additional keyword arguments to pass to *recordlinkage.base.BaseIndexAlgorithm*.

Examples

In the following example, the record pairs are made for two historical datasets with census data. The datasets are named `census_data_1980` and `census_data_1990`.

```
>>> indexer = recordlinkage.BlockIndex(on='first_name')
>>> indexer.index(census_data_1980, census_data_1990)
```

index(*x*, *x_link=None*)
Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters

- **x** (*pandas.DataFrame*) – A pandas DataFrame. When *x_link* is None, the algorithm makes record pairs within the DataFrame. When *x_link* is not empty, the algorithm makes pairs between *x* and *x_link*.
- **x_link** (*pandas.DataFrame, optional*) – A second DataFrame to link with the DataFrame x.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

class recordlinkage.index.**SortedNeighbourhood**(*left_on=None*, *right_on=None*, *window=3*, *sorting_key_values=None*, *block_on=[]*, *block_left_on=[]*, *block_right_on=[]*, ***kwargs*)

Make candidate record pairs with the SortedNeighbourhood algorithm.

This algorithm returns record pairs that agree on the sorting key, but also records pairs in their neighbourhood. A large window size results in more record pairs. A window size of 1 returns the blocking index.

The Sorted Neighbourhood Index method is a great method when there is relatively large amount of spelling mistakes. Blocking will fail in that situation because it excludes to many records on minor spelling mistakes.

Parameters

- **left_on** (*label, optional*) – The column name of the sorting key of the first/left dataframe.
- **right_on** (*label, optional*) – The column name of the sorting key of the second/right dataframe.
- **window** (*int, optional*) – The width of the window, default is 3
- **sorting_key_values** (*array, optional*) – A list of sorting key values (optional).
- **block_on** (*label*) – Additional columns to apply standard blocking on.
- **block_left_on** (*label*) – Additional columns in the left dataframe to apply standard blocking on.
- **block_right_on** (*label*) – Additional columns in the right dataframe to apply standard blocking on.
- ****kwargs*** – Additional keyword arguments to pass to *recordlinkage.base.BaseIndexAlgorithm*.

Examples

In the following example, the record pairs are made for two historical datasets with census data. The datasets are named `census_data_1980` and `census_data_1990`.

```
>>> indexer = recordlinkage.SortedNeighbourhoodIndex(
        'first_name', window=9
    )
>>> indexer.index(census_data_1980, census_data_1990)
```

When the sorting key has different names in both dataframes:

```
>>> indexer = recordlinkage.SortedNeighbourhoodIndex(
        left_on='first_name', right_on='given_name', window=9
    )
>>> indexer.index(census_data_1980, census_data_1990)
```

index(*x*, *x_link=None*)

Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters

- **x** (*pandas.DataFrame*) – A pandas DataFrame. When *x_link* is None, the algorithm makes record pairs within the DataFrame. When *x_link* is not empty, the algorithm makes pairs between *x* and *x_link*.
- **x_link** (*pandas.DataFrame, optional*) – A second DataFrame to link with the DataFrame x.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

class recordlinkage.index.**Random**(*n*, *replace=True*, *random_state=None*, ***kwargs*)

Class to generate random pairs of records.

This class returns random pairs of records with or without replacement. Use the random_state parameter to seed the algorithm and reproduce results. This way to make record pairs is useful for the training of unsupervised learning models for record linkage.

Parameters

- **n** (*int*) – The number of record pairs to return. In case replace=False, the integer n should be bounded by 0 < n <= n_max where n_max is the maximum number of pairs possible.
- **replace** (*bool, optional*) – Whether the sample of record pairs is with or without replacement. Default: True
- **random_state** (*int or numpy.random.RandomState, optional*) – Seed for the random number generator (if int), or numpy.RandomState object.
- ****kwargs** – Additional keyword arguments to pass to *recordlinkage.base.BaseIndexAlgorithm*.

index(*x*, *x_link=None*)

Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters

- **x** (*pandas.DataFrame*) – A pandas DataFrame. When *x_link* is None, the algorithm makes record pairs within the DataFrame. When *x_link* is not empty, the algorithm makes pairs between *x* and *x_link*.
- **x_link** (*pandas.DataFrame, optional*) – A second DataFrame to link with the DataFrame x.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

6.3 User-defined algorithms

A user-defined algorithm can be defined based on *recordlinkage.base.BaseIndexAlgorithm*. The *recordlinkage.base.BaseIndexAlgorithm* class is an abstract base class that is used for indexing algorithms. The classes

- *recordlinkage.index.Full*
- *recordlinkage.index.Block*
- *recordlinkage.index.SortedNeighbourhood*
- *recordlinkage.index.Random*

are inherited from this abstract base class. You can use BaseIndexAlgorithm to create a user-defined/custom algorithm.

To create a custom algorithm, subclass the *recordlinkage.base.BaseIndexAlgorithm*. In the subclass, overwrite the *recordlinkage.base.BaseIndexAlgorithm._link_index()* method in case of linking two datasets. This method accepts two (tuples of) pandas.Series objects as arguments. Based on these Series objects, you create record pairs. The record pairs need to be returned in a 2-level pandas.MultiIndex object. The pandas.MultiIndex.names are the name of index of DataFrame A and name of the index of DataFrame B respectively. Overwrite the *recordlinkage.base.BaseIndexAlgorithm._dedup_index()* method in case of finding link within a single dataset (deduplication). This method accepts a single (tuples of) pandas.Series objects as arguments.

The algorithm for linking data frames can be used for finding duplicates as well. In this situation, DataFrame B is a copy of DataFrame A. The Pairs class removes pairs like (record_i, record_i) and one of the following (record_i, record_j) (record_j, record_i) under the hood. As result of this, only unique combinations are returned. If you do have a specific algorithm for finding duplicates, then you can overwrite the _dedup_index method. This method accepts only one argument (DataFrame A) and the internal base class does not look for combinations like explained above.

class recordlinkage.base.**BaseIndexAlgorithm**(*verify_integrity=True*, *suffixes=('_1', '_2')*)

Base class for all index algorithms.

BaseIndexAlgorithm is an abstract class for indexing algorithms. The method *_link_index()*

Parameters

- **verify_integrity** (*bool*) – Verify the integrity of the input dataframe(s). The index is checked for duplicate values.
- **suffixes** (*tuple*) – If the names of the resulting MultiIndex are identical, the suffixes are used to distinguish the names.

Example

Make your own indexation class:

```
class CustomIndex(BaseIndexAlgorithm):

    def _link_index(self, df_a, df_b):

        # Custom index for linking.

        return ...

    def _dedup_index(self, df_a):

        # Custom index for duplicate detection, optional.

        return ...
```

Call the class in the same way:

```
custom_index = CustomIndex():
custom_index.index()
```

_link_index(*df_a*, *df_b*)

Build an index for linking two datasets.

Parameters

- **df_a** (*(tuple of) pandas.Series*) – The data of the left DataFrame to build the index with.
- **df_b** (*(tuple of) pandas.Series*) – The data of the right DataFrame to build the index with.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index values of two records.

_dedup_index(*df_a*)

Build an index for duplicate detection in a dataset.

This method can be used to implement an algorithm for duplicate detection. This method is optional if method *_link_index()* is implemented.

Parameters **df_a** (*(tuple of) pandas.Series*) – The data of the DataFrame to build the index with.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index values of two records. The records are sampled from the lower triangular part of the matrix.

index(*x*, *x_link=None*)

Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters

- **x** (*pandas.DataFrame*) – A pandas DataFrame. When *x_link* is None, the algorithm makes record pairs within the DataFrame. When *x_link* is not empty, the algorithm makes pairs between *x* and *x_link*.
- **x_link** (*pandas.DataFrame, optional*) – A second DataFrame to link with the DataFrame x.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

6.4 Examples

```
import recordlinkage as rl
from recordlinkage.datasets import load_febrl4
from recordlinkage.index import Block

df_a, df_b = load_febrl4()

indexer = rl.Index()
indexer.add(Block('given_name', 'given_name'))
indexer.add(Block('surname', 'surname'))
indexer.index(df_a, df_b)
```

Equivalent code:

```
import recordlinkage as rl
from recordlinkage.datasets import load_febrl4

df_a, df_b = load_febrl4()
```

(continues on next page)

(continued from previous page)

```
indexer = rl.Index()
indexer.block('given_name', 'given_name')
indexer.block('surname', 'surname')
index.index(df_a, df_b)
```

This example shows how to implement a custom indexing algorithm. The algorithm returns all record pairs of which the given names starts with the letter 'W'.

```
import recordlinkage
from recordlinkage.datasets import load_febrl4

df_a, df_b = load_febrl4()

from recordlinkage.base import BaseIndexAlgorithm

class FirstLetterWIndex(BaseIndexAlgorithm):
    """Custom class for indexing"""

    def _link_index(self, df_a, df_b):
        """Make pairs with given names starting with the letter 'w'."""

        # Select records with names starting with a w.
        name_a_w = df_a[df_a['given_name'].str.startswith('w') == True]
        name_b_w = df_b[df_b['given_name'].str.startswith('w') == True]

        # Make a product of the two numpy arrays
        return pandas.MultiIndex.from_product(
            [name_a_w.index.values, name_b_w.index.values],
            names=[df_a.index.name, df_b.index.name]
        )

indexer = FirstLetterWIndex()
candidate_pairs = indexer.index(df_a, df_b)

print ('Returns a', type(candidate_pairs).__name__)
print ('Number of candidate record pairs starting with the letter w:', len(candidate_
↪pairs))
```

The custom index class below does not restrict the first letter to 'w', but the first letter is an argument (named *letter*). This letter can is initialized during the setup of the class.

```
class FirstLetterIndex(BaseIndexAlgorithm):
    """Custom class for indexing"""

    def __init__(self, letter):
        super(FirstLetterIndex, self).__init__()

        # the letter to save
        self.letter = letter

    def _link_index(self, df_a, df_b):
        """Make record pairs that agree on the first letter of the given name."""

        # Select records with names starting with a 'letter'.
        a_startswith_w = df_a[df_a['given_name'].str.startswith(self.letter) == True]
```

(continues on next page)

(continued from previous page)

```
        b_startswith_w = df_b[df_b['given_name'].str.startswith(self.letter) == True]

        # Make a product of the two numpy arrays
        return pandas.MultiIndex.from_product(
            [a_startswith_w.index.values, b_startswith_w.index.values],
            names=[df_a.index.name, df_b.index.name]
        )
```

CHAPTER 7

2. Comparing

A set of informative, discriminating and independent features is important for a good classification of record pairs into matching and distinct pairs. The *recordlinkage.Compare* class and its methods can be used to compare records pairs. Several comparison methods are included such as string similarity measures, numerical measures and distance measures.

7.1 recordlinkage.Compare object

class recordlinkage.**Compare**(*features=[]*, *n_jobs=1*, *indexing_type='label'*, ***kwargs*)

Class to compare record pairs with efficiently.

Class to compare the attributes of candidate record pairs. The `Compare` class has methods like `string`, `exact` and `numeric` to initialise the comparing of the records. The `compute` method is used to start the actual comparing.

Example

Consider two historical datasets with census data to link. The datasets are named `census_data_1980` and `census_data_1990`. The MultiIndex `candidate_pairs` contains the record pairs to compare. The record pairs are compared on the first name, last name, sex, date of birth, address, place, and income:

```
# initialise class
comp = recordlinkage.Compare()

# initialise similarity measurement algorithms
comp.string('first_name', 'name', method='jarowinkler')
comp.string('lastname', 'lastname', method='jarowinkler')
comp.exact('dateofbirth', 'dob')
comp.exact('sex', 'sex')
comp.string('address', 'address', method='levenshtein')
comp.exact('place', 'place')
comp.numeric('income', 'income')
```

(continues on next page)

(continued from previous page)

```
# the method .compute() returns the DataFrame with the feature vectors.
comp.compute(candidate_pairs, census_data_1980, census_data_1990)
```

Parameters

- **features** (*list*) – List of compare algorithms.
- **n_jobs** (*integer, optional (default=1)*) – The number of jobs to run in parallel for comparing of record pairs. If -1, then the number of jobs is set to the number of cores.
- **indexing_type** (*string, optional (default='label')*) – The indexing type. The MultiIndex is used to index the DataFrame(s). This can be done with pandas `.loc` or with `.iloc`. Use the value 'label' to make use of `.loc` and 'position' to make use of `.iloc`. The value 'position' is only available when the MultiIndex consists of integers. The value 'position' is much faster.

features
A list of algorithms to create features.

Type list

add(*model*)
Add a compare method.

This method is used to add compare features.

Parameters **model** (*list, class*) – A (list of) compare feature(s) from *recordlinkage.compare*.

compute(*pairs*, *x*, *x_link=None*)
Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.DataFrame* – A pandas DataFrame with feature vectors, i.e. the result of comparing each record pair.

compare_vectorized(*comp_func*, *labels_left*, *labels_right*, **args*, ***kwargs*)
Compute the similarity between values with a callable.

This method initialises the comparing of values with a custom function/callable. The function/callable should accept numpy.ndarray's.

Example

```
>>> comp = recordlinkage.Compare()
>>> comp.compare_vectorized(custom_callable, 'first_name', 'name')
>>> comp.compare(PAIRS, DATAFRAME1, DATAFRAME2)
```

Parameters

- **comp_func** (*function*) – A comparison function. This function can be a built-in function or a user defined comparison function. The function should accept numpy.ndarray's as first two arguments.
- **labels_left** (*label, pandas.Series, pandas.DataFrame*) – The labels, Series or DataFrame to compare.
- **labels_right** (*label, pandas.Series, pandas.DataFrame*) – The labels, Series or DataFrame to compare.
- ***args** – Additional arguments to pass to callable comp_func.
- ****kwargs** – Additional keyword arguments to pass to callable comp_func. (keyword 'label' is reserved.)
- **label** (*(list of) label(s)*) – The name of the feature and the name of the column. IMPORTANT: This argument is a keyword argument and can not be part of the arguments of comp_func.

exact (**args*, ***kwargs*)

Compare attributes of pairs exactly.

Shortcut of *recordlinkage.compare.Exact*:

```
from recordlinkage.compare import Exact

indexer = recordlinkage.Compare()
indexer.add(Exact())
```

string (**args*, ***kwargs*)

Compare attributes of pairs with string algorithm.

Shortcut of *recordlinkage.compare.String*:

```
from recordlinkage.compare import String

indexer = recordlinkage.Compare()
indexer.add(String())
```

numeric (**args*, ***kwargs*)

Compare attributes of pairs with numeric algorithm.

Shortcut of *recordlinkage.compare.Numeric*:

```
from recordlinkage.compare import Numeric

indexer = recordlinkage.Compare()
indexer.add(Numeric())
```

geo (**args*, ***kwargs*)

Compare attributes of pairs with geo algorithm.

Shortcut of *recordlinkage.compare.Geographic*:

```
from recordlinkage.compare import Geographic

indexer = recordlinkage.Compare()
indexer.add(Geographic())
```

date (**args*, ***kwargs*)
Compare attributes of pairs with date algorithm.

Shortcut of *recordlinkage.compare.Date*:

```
from recordlinkage.compare import Date

indexer = recordlinkage.Compare()
indexer.add(Date())
```

7.2 Algorithms

class recordlinkage.compare.**Exact**(*left_on*, *right_on*, *agree_value=1*, *disagree_value=0*, *missing_value=0*, *label=None*)
Compare the record pairs exactly.

This class is used to compare records in an exact way. The similarity is 1 in case of agreement and 0 otherwise.

Parameters

- **left_on** (*str or int*) – Field name to compare in left DataFrame.
- **right_on** (*str or int*) – Field name to compare in right DataFrame.
- **agree_value** (*float, str, numpy.dtype*) – The value when two records are identical. Default 1. If 'values' is passed, then the value of the record pair is passed.
- **disagree_value** (*float, str, numpy.dtype*) – The value when two records are not identical.
- **missing_value** (*float, str, numpy.dtype*) – The value for a comparison with a missing value. Default 0.

compute (*pairs*, *x*, *x_link=None*)
Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

class recordlinkage.compare.**String**(*left_on*, *right_on*, *method='levenshtein'*, *threshold=None*, *missing_value=0.0*, *label=None*)
Compute the (partial) similarity between strings values.

This class is used to compare string values. The implemented algorithms are: 'jaro','jarowinkler', 'levenshtein', 'damerau_levenshtein', 'qgram' or 'cosine'. In case of agreement, the similarity is 1 and in case of complete disagreement it is 0. The Python Record Linkage Toolkit uses the *jellyfish* package for the Jaro, Jaro-Winkler, Levenshtein and Damerau- Levenshtein algorithms.

Parameters

- **left_on** (*str or int*) – The name or position of the column in the left DataFrame.
- **right_on** (*str or int*) – The name or position of the column in the right DataFrame.
- **method** (*str, default 'levenshtein'*) – An approximate string comparison method. Options are ['jaro', 'jarowinkler', 'levenshtein', 'damerau_levenshtein', 'qgram', 'cosine', 'smith_waterman', 'lcs']. Default: 'levenshtein'
- **threshold** (*float, tuple of floats*) – A threshold value. All approximate string comparisons higher or equal than this threshold are 1. Otherwise 0.
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.

compute(*pairs*, *x*, *x_link=None*)

Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

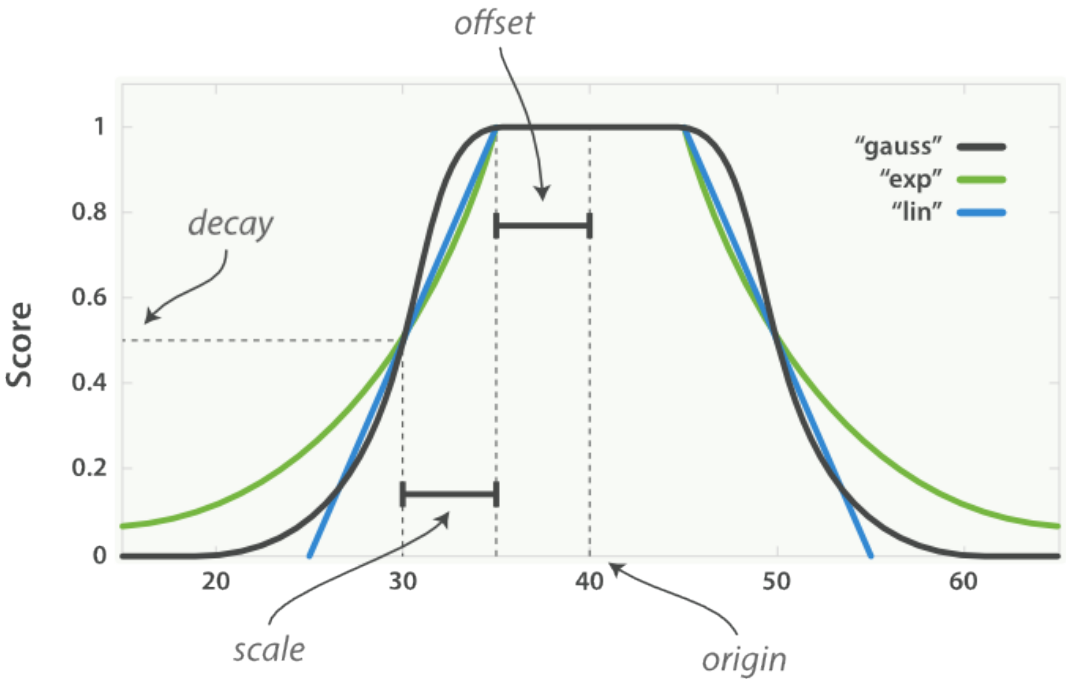
- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

class recordlinkage.compare.**Numeric**(*left_on*, *right_on*, *method='linear'*, *offset=0.0*, *scale=1.0*, *origin=0.0*, *missing_value=0.0*, *label=None*)

Compute the (partial) similarity between numeric values.

This class is used to compare numeric values. The implemented algorithms are: 'step', 'linear', 'exp', 'gauss' or 'squared'. In case of agreement, the similarity is 1 and in case of complete disagreement it is 0. The implementation is similar with numeric comparing in ElasticSearch, a full- text search tool. The parameters are explained in the image below (source ElasticSearch, The Definitive Guide)



Parameters

- **left_on** (*str or int*) – The name or position of the column in the left DataFrame.
- **right_on** (*str or int*) – The name or position of the column in the right DataFrame.
- **method** (*float*) – The metric used. Options 'step', 'linear', 'exp', 'gauss' or 'squared'. Default 'linear'.
- **offset** (*float*) – The offset. See image above.
- **scale** (*float*) – The scale of the numeric comparison method. See the image above. This argument is not available for the 'step' algorithm.
- **origin** (*float*) – The shift of bias between the values. See image above.
- **missing_value** (*numpy.dtype*) – The value if one or both records have a missing value on the compared field. Default 0.

Note: Numeric comparing can be an efficient way to compare date/time variables. This can be done by comparing the timestamps.

---

compute(*pairs*, *x*, *x_link=None*)

Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.

- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

class recordlinkage.compare.**Geographic**(*left_on_lat*, *left_on_lng*, *right_on_lat*, *right_on_lng*, *method=None*, *offset=0.0*, *scale=1.0*, *origin=0.0*, *missing_value=0.0*, *label=None*)

Compute the (partial) similarity between WGS84 coordinate values.

Compare the geometric (haversine) distance between two WGS- coordinates. The similarity algorithms are 'step', 'linear', 'exp', 'gauss' or 'squared'. The similarity functions are the same as in `recordlinkage.comparing.Compare.numeric()`

Parameters

- **left_on_lat** (*tuple*) – The name or position of the latitude in the left DataFrame.
- **left_on_lng** (*tuple*) – The name or position of the longitude in the left DataFrame.
- **right_on_lat** (*tuple*) – The name or position of the latitude in the right DataFrame.
- **right_on_lng** (*tuple*) – The name or position of the longitude in the right DataFrame.
- **method** (*str*) – The metric used. Options 'step', 'linear', 'exp', 'gauss' or 'squared'. Default 'linear'.
- **offset** (*float*) – The offset. See Compare.numeric.
- **scale** (*float*) – The scale of the numeric comparison method. See Compare.numeric. This argument is not available for the 'step' algorithm.
- **origin** (*float*) – The shift of bias between the values. See Compare.numeric.
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.

compute(*pairs*, *x*, *x_link=None*)

Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

class recordlinkage.compare.**Date**(*left_on*, *right_on*, *swap_month_day=0.5*, *swap_months='default'*, *errors='coerce'*, *missing_value=0.0*, *label=None*)

Compute the (partial) similarity between date values.

Parameters

- **left_on** (*str or int*) – The name or position of the column in the left DataFrame.
- **right_on** (*str or int*) – The name or position of the column in the right DataFrame.
- **swap_month_day** (*float*) – The value if the month and day are swapped. Default 0.5.
- **swap_months** (*list of tuples*) – A list of tuples with common errors caused by the translating of months into numbers, i.e. October is month 10. The format of the tuples is (month_good, month_bad, value). Default : swap_months = [(6, 7, 0.5), (7, 6, 0.5), (9, 10, 0.5), (10, 9, 0.5)]
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.0.

compute(*pairs*, *x*, *x_link=None*)

Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

class recordlinkage.compare.**Variable**(*left_on=None*, *right_on=None*, *missing_value=0.0*, *label=None*)

Add a variable of the dataframe as feature.

Parameters

- **left_on** (*str or int*) – The name or position of the column in the left DataFrame.
- **right_on** (*str or int*) – The name or position of the column in the right DataFrame.
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.0.

compute(*pairs*, *x*, *x_link=None*)

Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

class recordlinkage.compare.**VariableA**(*on=None*, *missing_value=0.0*, *label=None*)
Add a variable of the left dataframe as feature.

Parameters

- **on** (*str or int*) – The name or position of the column in the left DataFrame.
- **normalise** (*bool*) – Normalise the outcome. This is needed for good result in many classification models. Default True.
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.0.

compute(*pairs*, *x*, *x_link=None*)
Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

class recordlinkage.compare.**VariableB**(*on=None*, *missing_value=0.0*, *label=None*)
Add a variable of the right dataframe as feature.

Parameters

- **on** (*str or int*) – The name or position of the column in the right DataFrame.
- **normalise** (*bool*) – Normalise the outcome. This is needed for good result in many classification models. Default True.
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.0.

compute(*pairs*, *x*, *x_link=None*)
Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

class recordlinkage.compare.**Frequency**(*left_on=None*, *right_on=None*, *normalise=True*, *missing_value=0.0*, *label=None*)

Compute the (relative) frequency of each variable.

Parameters

- **left_on** (*str or int*) – The name or position of the column in the left DataFrame.
- **right_on** (*str or int*) – The name or position of the column in the right DataFrame.
- **normalise** (*bool*) – Normalise the outcome. This is needed for good result in many classification models. Default True.
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.0.

compute(*pairs*, *x*, *x_link=None*)

Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

class recordlinkage.compare.**FrequencyA**(*on=None*, *normalise=True*, *missing_value=0.0*, *label=None*)

Compute the frequency of a variable in the left dataframe.

Parameters

- **on** (*str or int*) – The name or position of the column in the left DataFrame.
- **normalise** (*bool*) – Normalise the outcome. This is needed for good result in many classification models. Default True.
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.0.

compute(*pairs*, *x*, *x_link=None*)

Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

class recordlinkage.compare.**FrequencyB**(*on=None*, *normalise=True*, *missing_value=0.0*, *label=None*)

Compute the frequency of a variable in the right dataframe.

Parameters

- **on** (*str or int*) – The name or position of the column in the right DataFrame.
- **normalise** (*bool*) – Normalise the outcome. This is needed for good result in many classification models. Default True.
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.0.

compute(*pairs*, *x*, *x_link=None*)

Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

7.3 User-defined algorithms

A user-defined algorithm can be defined based on *recordlinkage.base.BaseCompareFeature*. The *recordlinkage.base.BaseCompareFeature* class is an abstract base class that is used for compare algorithms. The classes

- *recordlinkage.compare.Exact*
- *recordlinkage.compare.String*
- *recordlinkage.compare.Numeric*
- *recordlinkage.compare.Date*

are inherited from this abstract base class. You can use BaseCompareFeature to create a user-defined/custom algorithm. Overwrite the abstract method *recordlinkage.base.BaseCompareFeature._compute_vectorized()* with the compare algorithm. A short example is given here:

```
from recordlinkage.base import BaseCompareFeature

class CustomFeature(BaseCompareFeature):

    def _compute_vectorized(s1, s2):
        # algorithm that compares s1 and s2
```

(continues on next page)

(continued from previous page)

```
        # return a pandas.Series
        return ...

feat = CustomFeature()
feat.compute(pairs, dfA, dfB)
```

A full description of the *recordlinkage.base.BaseCompareFeature* class:

class recordlinkage.base.**BaseCompareFeature**(*labels_left*, *labels_right*, *args=()*, *kwargs={}*, *label=None*)

Base abstract class for compare feature engineering.

Parameters

- **labels_left** (*list, str, int*) – The labels to use for comparing record pairs in the left dataframe.
- **labels_right** (*list, str, int*) – The labels to use for comparing record pairs in the right dataframe (linking) or left dataframe (duplicate detection).
- **args** (*tuple*) – Additional arguments to pass to the *_compare_vectorized* method.
- **kwargs** (*tuple*) – Keyword additional arguments to pass to the *_compare_vectorized* method.
- **label** (*list, str, int*) – The identifying label(s) for the returned values.

compute(*pairs*, *x*, *x_link=None*)

Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of duplicate detection.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

_compute(*left_on*, *right_on*)

Compare the data on the left and right.

BaseCompareFeature._compute() and *BaseCompareFeature.compute()* differ on the accepted arguments. *_compute* accepts indexed data while *compute* accepts the record pairs and the DataFrame's.

Parameters

- **left_on** (*(tuple of) pandas.Series*) – Data to compare with *right_on*
- **right_on** (*(tuple of) pandas.Series*) – Data to compare with *left_on*

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

_compute_vectorized(**args*)
Compare attributes (vectorized)

Parameters ***args** (*pandas.Series*) – pandas.Series' as arguments.

Returns *pandas.Series, pandas.DataFrame, numpy.ndarray* – The result of comparing record pairs (the features). Can be a tuple with multiple pandas.Series, pandas.DataFrame, numpy.ndarray objects.

Warning: Do not change the order of the pairs in the MultiIndex.

7.4 Examples

7.4.1 Example: High level usage

```
import recordlinkage as rl

comparer = rl.Compare()
comparer.string('name_a', 'name_b', method='jarowinkler', threshold=0.85, label='name
↪')
comparer.exact('sex', 'gender', label='gender')
comparer.date('dob', 'date_of_birth', label='date')
comparer.string('str_name', 'streetname', method='damerau_levenshtein', threshold=0.7,
↪ label='streetname')
comparer.exact('place', 'placename', label='placename')
comparer.numeric('income', 'income', method='gauss', offset=3, scale=3, missing_
↪value=0.5, 'label'='income')
comparer.compute(pairs, dfA, dfB)
```

7.4.2 Example: Low level usage

```
import recordlinkage as rl
from recordlinkage.compare import Exact, String, Numeric, Date

comparer = rl.Compare([
    String('name_a', 'name_b', method='jarowinkler', threshold=0.85, label='name')
    Exact('sex', 'gender', label='gender')
    Date('dob', 'date_of_birth', label='date')
    String('str_name', 'streetname', method='damerau_levenshtein', threshold=0.7,␣
↪label='streetname')
    Exact('place', 'placename', label='placename')
    Numeric('income', 'income', method='gauss', offset=3, scale=3, missing_value=0.5,
↪'label'='income')
])
comparer.compute(pairs, dfA, dfB)
```

The following examples give a feeling on the extensibility of the toolkit.

7.4.3 Example: User-defined algorithm 1

The following code defines a custom algorithm to compare zipcodes. The algorithm returns 1.0 for record pairs that agree on the zipcode and returns 0.0 for records that disagree on the zipcode. If the zipcodes disagree but the first two numbers are identical, then the algorithm returns 0.5.

```
import recordlinkage as rl
from recordlinkage.base import BaseCompareFeature

class CompareZipCodes(BaseCompareFeature):

    def _compute_vectorized(self, s1, s2):
        """Compare zipcodes.

        If the zipcodes in both records are identical, the similarity
        is 0. If the first two values agree and the last two don't, then
        the similarity is 0.5. Otherwise, the similarity is 0.
        """

        # check if the zipcode are identical (return 1 or 0)
        sim = (s1 == s2).astype(float)

        # check the first 2 numbers of the distinct comparisons
        sim[(sim == 0) & (s1.str[0:2] == s2.str[0:2])] = 0.5

        return sim

comparer = rl.Compare()
comparer.extact('given_name', 'given_name', 'y_name')
comparer.string('surname', 'surname', 'y_surname')
comparer.add(CompareZipCodes('postcode', 'postcode', label='y_postcode'))
comparer.compute(pairs, dfA, dfB)
```

```
0.0    71229
0.5     3166
1.0     2854
Name: sim_postcode, dtype: int64
```

Note: See *recordlinkage.base.BaseCompareFeature* for more details on how to subclass.

7.4.4 Example: User-defined algorithm 2

As you can see, one can pass the labels of the columns as arguments. The first argument is a column label, or a list of column labels, found in the first DataFrame (`postcode` in this example). The second argument is a column label, or a list of column labels, found in the second DataFrame (also `postcode` in this example). The `recordlinkage.Compare` class selects the columns with the given labels before passing them to the custom algorithm/function. The `compare` method in the `recordlinkage.Compare` class passes additional (keyword) arguments to the custom function.

Warning: Do not change the order of the pairs in the MultiIndex.

```
import recordlinkage as rl
from recordlinkage.base import BaseCompareFeature

```

(continues on next page)

(continued from previous page)

```
class CompareZipCodes(BaseCompareFeature):

    def __init__(self, left_on, right_on, partial_sim_value, *args, **kwargs):
        super(CompareZipCodes, self).__init__(left_on, right_on, *args, **kwargs)

        self.partial_sim_value = partial_sim_value

    def _compute_vectorized(self, s1, s2):
        """Compare zipcodes.

        If the zipcodes in both records are identical, the similarity
        is 0. If the first two values agree and the last two don't, then
        the similarity is 0.5. Otherwise, the similarity is 0.
        """

        # check if the zipcode are identical (return 1 or 0)
        sim = (s1 == s2).astype(float)

        # check the first 2 numbers of the distinct comparisons
        sim[(sim == 0) & (s1.str[0:2] == s2.str[0:2])] = self.partial_sim_value

        return sim

comparer = rl.Compare()
comparer.extact('given_name', 'given_name', 'y_name')
comparer.string('surname', 'surname', 'y_surname')
comparer.add(CompareZipCodes('postcode', 'postcode',
                             'partial_sim_value'=0.2, label='y_postcode'))
comparer.compute(pairs, dfA, dfB)
```

7.4.5 Example: User-defined algorithm 3

The Python Record Linkage Toolkit supports the comparison of more than two columns. This is especially useful in situations with multi-dimensional data (for example geographical coordinates) and situations where fields can be swapped.

The FEBRL4 dataset has two columns filled with address information (`address_1` and `address_2`). In a naive approach, one compares `address_1` of file A with `address_1` of file B and `address_2` of file A with `address_2` of file B. If the values for `address_1` and `address_2` are swapped during the record generating process, the naive approach considers the addresses to be distinct. In a more advanced approach, `address_1` of file A is compared with `address_1` and `address_2` of file B. Variable `address_2` of file A is compared with `address_1` and `address_2` of file B. This is done with the single function given below.

```
import recordlinkage as rl
from recordlinkage.base import BaseCompareFeature

class CompareAddress(BaseCompareFeature):

    def _compute_vectorized(self, s1_1, s1_2, s2_1, s2_2):
        """Compare addresses.

        Compare addresses. Compare address_1 of file A with
        address_1 and address_2 of file B. The same for address_2
        of dataset 1.

```

(continues on next page)

(continued from previous page)

```
        """

        return ((s1_1 == s2_1) | (s1_2 == s2_2) | (s1_1 == s2_2) | (s1_2 == s2_1)).
↪astype(float)

comparer = rl.Compare()

# naive
comparer.add(CompareAddress('address_1', 'address_1', label='sim_address_1'))
comparer.add(CompareAddress('address_2', 'address_2', label='sim_address_2'))

# better
comparer.add(CompareAddress(('address_1', 'address_2'),
                            ('address_1', 'address_2'),
                            label='sim_address'
))

features = comparer.compute(pairs, dfA, dfB)
features.mean()
```

The mean of the cross-over comparison is higher.

```
sim_address_1    0.02488
sim_address_2    0.02025
sim_address      0.03566
dtype: float64
```

CHAPTER 8

3. Classification

8.1 Classifiers

Classification is the step in the record linkage process were record pairs are classified into matches, non-matches and possible matches [*Christen2012*]. Classification algorithms can be supervised or unsupervised (with or without training data).

See also:

8.1.1 Supervised

class recordlinkage.**LogisticRegressionClassifier**(*coefficients=None*, *intercept=None*, ***kwargs*)

Logistic Regression Classifier.

This classifier is an application of the logistic regression model (wikipedia). The classifier partitions candidate record pairs into matches and non-matches.

This algorithm is also known as Deterministic Record Linkage.

The LogisticRegressionClassifier classifier uses the `sklearn.linear_model.LogisticRegression` classification algorithm from SciKit-learn as kernel.

Parameters

- **coefficients** (*list, numpy.array*) – The coefficients of the logistic regression.
- **intercept** (*float*) – The interception value.
- ****kwargs** – Additional arguments to pass to `sklearn.linear_model.LogisticRegression`.

kernel

The kernel of the classifier. The kernel is `sklearn.linear_model.LogisticRegression` from SciKit-learn.

Type sklearn.linear_model.LogisticRegression

coefficients
The coefficients of the logistic regression.

Type list

intercept
The interception value.

Type float

fit(*comparison_vectors*, *match_index=None*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors (or features) to train the model with.
- **match_index** (*pandas.MultiIndex*) – A pandas.MultiIndex object with the true matches. The MultiIndex contains only the true matches. Default None.

Note: A note in case of finding links within a single dataset (for example duplicate detection). Unsure that the training record pairs are from the lower triangular part of the dataset/matrix. See detailed information here: link.

---

fit_predict(*comparison_vectors*, *match_index=None*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors.
- **match_index** (*pandas.MultiIndex*) – The true matches.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

learn(**args*, ***kwargs*)
[DEPRECATED] Use 'fit_predict'.

predict(*comparison_vectors*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – Dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

prob(*comparison_vectors*, *return_type=None*)
Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. (default 'series')

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

class recordlinkage.**NaiveBayesClassifier**(*binarize=None*, *alpha=0.0001*, *use_col_names=True*, ***kwargs*)

Naive Bayes Classifier.

The Naive Bayes classifier (wikipedia) partitions candidate record pairs into matches and non-matches. The classifier is based on probabilistic principles. The Naive Bayes classification method has a close mathematical connection with the Fellegi and Sunter model.

Note: The NaiveBayesClassifier classifier differs of the Naive Bayes models in SciKit-learn. With binary input vectors, the NaiveBayesClassifier behaves like `sklearn.naive_bayes.BernoulliNB`.

Parameters

- **binarize** (*float or None, optional (default=None)*) – Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to consist of multilevel vectors.
- **alpha** (*float*) – Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing). Default 1e-4.
- **use_col_names** (*bool*) – Use the column names of the pandas.DataFrame to identify the parameters. If False, the column index of the feature is used. Default True.

kernel
The kernel of the classifier. The kernel is `sklearn.naive_bayes.BernoulliNB` from SciKit-learn.

Type sklearn.naive_bayes.BernoulliNB

log_p
Log match probability as described in the FS framework.

Type float

log_m_probs
Log probability P(x_i=1|Match) as described in the FS framework.

Type np.ndarray

log_u_probs
Log probability P(x_i=1|Non-match) as described in the FS framework.

Type np.ndarray

log_weights
Log weights as described in the FS framework.

Type np.ndarray

p
Match probability as described in the FS framework.

Type float

m_probs
Probability P(x_i=1|Match) as described in the FS framework.

Type np.ndarray

u_probs
Probability P(x_i=1|Non-match) as described in the FS framework.

Type np.ndarray

weights
Weights as described in the FS framework.

Type np.ndarray

fit(*X*, **args*, ***kwargs*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors (or features) to train the model with.
- **match_index** (*pandas.MultiIndex*) – A pandas.MultiIndex object with the true matches. The MultiIndex contains only the true matches. Default None.

Note: A note in case of finding links within a single dataset (for example duplicate detection). Unsure that the training record pairs are from the lower triangular part of the dataset/matrix. See detailed information here: link.

---

fit_predict(*comparison_vectors*, *match_index=None*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors.
- **match_index** (*pandas.MultiIndex*) – The true matches.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

learn(**args*, ***kwargs*)
[DEPRECATED] Use 'fit_predict'.

log_m_probs
Log probability P(x_i=1|Match) as described in the FS framework

log_p
Log match probability as described in the FS framework

log_u_probs
Log probability P(x_i=1|Non-match) as described in the FS framework

log_weights
Log weights as described in the FS framework

m_probs
Probability P(x_i=1|Match) as described in the FS framework

p
Match probability as described in the FS framework

predict(*comparison_vectors*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – Dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

prob(*comparison_vectors*, *return_type=None*)
Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. (default 'series')

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

u_probs
Probability P(x_i=1|Non-match) as described in the FS framework

weights
Weights as described in the FS framework

class recordlinkage.**SVMClassifier**(**args*, ***kwargs*)
Support Vector Machines Classifier

The Support Vector Machine classifier (wikipedia) partitions candidate record pairs into matches and non-matches. This implementation is a non-probabilistic binary linear classifier. Support vector machines are supervised learning models. Therefore, SVM classifiers need training- data.

The SVMClassifier classifier uses the `sklearn.svm.LinearSVC` classification algorithm from SciKit-learn as kernel.

Parameters ****kwargs** – Arguments to pass to `sklearn.svm.LinearSVC`.

kernel
The kernel of the classifier. The kernel is `sklearn.svm.LinearSVC` from SciKit-learn.

Type sklearn.svm.LinearSVC

fit(*comparison_vectors*, *match_index=None*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors (or features) to train the model with.
- **match_index** (*pandas.MultiIndex*) – A pandas.MultiIndex object with the true matches. The MultiIndex contains only the true matches. Default None.

Note: A note in case of finding links within a single dataset (for example duplicate detection). Unsure that the training record pairs are from the lower triangular part of the dataset/matrix. See detailed information here: link.

---

fit_predict(*comparison_vectors*, *match_index=None*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors.
- **match_index** (*pandas.MultiIndex*) – The true matches.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

learn(**args*, ***kwargs*)
[DEPRECATED] Use 'fit_predict'.

predict(*comparison_vectors*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – Dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

prob(**args*, ***kwargs*)
Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. (default 'series')

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

8.1.2 Unsupervised

class recordlinkage.**ECMClassifier**(*init='jaro'*, *binarize=None*, *max_iter=100*, *atol=0.0001*, *use_col_names=True*, **args*, ***kwargs*)

Expectation/Conditional Maxisation classifier (Unsupervised).

Expectation/Conditional Maximisation algorithm used to classify record pairs. This probabilistic record linkage algorithm is used in combination with Fellegi and Sunter model. This classifier doesn't need training data (unsupervised).

Parameters

- **init** (*str*) – Initialisation method for the algorithm. Options are: 'jaro' and 'random'. Default 'jaro'.
- **max_iter** (*int*) – The maximum number of iterations of the EM algorithm. Default 100.
- **binarize** (*float or None, optional (default=None)*) – Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.
- **atol** (*float*) – The tolerance between parameters between each interation. If the difference between the parameters between the iterations is smaller than this value, the algorithm is considered to be converged. Default 10e-4.
- **use_col_names** (*bool*) – Use the column names of the pandas.DataFrame to identify the parameters. If False, the column index of the feature is used. Default True.

kernel
The kernel of the classifier.

Type recordlinkage.algorithms.em_sklearn.ECM

log_p
Log match probability as described in the FS framework.

Type float

log_m_probs
Log probability P(x_i=1|Match) as described in the FS framework.

Type np.ndarray

log_u_probs
Log probability P(x_i=1|Non-match) as described in the FS framework.

Type np.ndarray

log_weights
Log weights as described in the FS framework.

Type np.ndarray

p
Match probability as described in the FS framework.

Type float

m_probs
Probability P(x_i=1|Match) as described in the FS framework.

Type np.ndarray

u_probs
Probability P(x_i=1|Non-match) as described in the FS framework.

Type np.ndarray

weights
Weights as described in the FS framework.

Type np.ndarray

References

Herzog, Thomas N, Fritz J Scheuren and William E Winkler. 2007. Data quality and record linkage techniques. Vol. 1 Springer.

Fellegi, Ivan P and Alan B Sunter. 1969. "A theory for record linkage." Journal of the American Statistical Association 64(328):1183–1210.

Collins, M. "The Naive Bayes Model, Maximum-Likelihood Estimation, and the EM Algorithm". http://www.cs.columbia.edu/~mcollins/em.pdf

---

fit_predict(*comparison_vectors*, *match_index=None*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors.
- **match_index** (*pandas.MultiIndex*) – The true matches.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

learn(**args*, ***kwargs*)
[DEPRECATED] Use 'fit_predict'.

log_m_probs
Log probability P(x_i=1|Match) as described in the FS framework

log_p
Log match probability as described in the FS framework

log_u_probs
Log probability P(x_i=1|Non-match) as described in the FS framework

log_weights
Log weights as described in the FS framework

m_probs
Probability P(x_i=1|Match) as described in the FS framework

p
Match probability as described in the FS framework

predict(*comparison_vectors*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – Dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

prob(*comparison_vectors*, *return_type=None*)
Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. (default 'series')

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

u_probs
Probability P(x_i=1|Non-match) as described in the FS framework

weights
Weights as described in the FS framework

fit(*X*, **args*, ***kwargs*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors (or features) to train the model with.
- **match_index** (*pandas.MultiIndex*) – A pandas.MultiIndex object with the true matches. The MultiIndex contains only the true matches. Default None.

Note: A note in case of finding links within a single dataset (for example duplicate detection). Unsure that the training record pairs are from the lower triangular part of the dataset/matrix. See detailed information here: link.

---

class recordlinkage.**KMeansClassifier**(*match_cluster_center=None*, *non-match_cluster_center=None*, ***kwargs*)
KMeans classifier.

The K-means clusterings algorithm (wikipedia) partitions candidate record pairs into matches and non-matches. Each comparison vector belongs to the cluster with the nearest mean.

The K-means algorithm is an unsupervised learning algorithm. The algorithm doesn't need trainings data for fitting. The algorithm is calibrated for two clusters: a match cluster and a non-match cluster). The centers of these clusters can be given as arguments or set automatically.

The KMeansClassifier classifier uses the `sklearn.cluster.KMeans` clustering algorithm from SciKit-learn as kernel.

Parameters

- **match_cluster_center** (*list, numpy.array*) – The center of the match cluster. The length of the list/array must equal the number of comparison variables. If None, the match cluster center is set automatically. Default None.
- **nonmatch_cluster_center** (*list, numpy.array*) – The center of the non-match (distinct) cluster. The length of the list/array must equal the number of comparison variables. If None, the non-match cluster center is set automatically. Default None.
- ****kwargs** – Additional arguments to pass to `sklearn.cluster.KMeans`.

kernel
The kernel of the classifier. The kernel is `sklearn.cluster.KMeans` from SciKit-learn.

Type sklearn.cluster.KMeans

match_cluster_center
The center of the match cluster.

Type numpy.array

nonmatch_cluster_center
The center of the nonmatch (distinct) cluster.

Type numpy.array

Note: There are better methods for linking records than the k-means clustering algorithm. This algorithm can be useful for an (unsupervised) initial partition.

---

prob(**args*, ***kwargs*)
Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. (default 'series')

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

fit(*comparison_vectors*, *match_index=None*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors (or features) to train the model with.
- **match_index** (*pandas.MultiIndex*) – A pandas.MultiIndex object with the true matches. The MultiIndex contains only the true matches. Default None.

Note: A note in case of finding links within a single dataset (for example duplicate detection). Unsure that the training record pairs are from the lower triangular part of the dataset/matrix. See detailed information here: link.

---

fit_predict(*comparison_vectors*, *match_index=None*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors.
- **match_index** (*pandas.MultiIndex*) – The true matches.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

learn (**args*, ***kwargs*)
[DEPRECATED] Use 'fit_predict'.

predict (*comparison_vectors*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – Dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

8.2 Adapters

Adapters can be used to wrap a machine learning models from external packages like ScitKit-learn and Keras. For example, this makes it possible to classify record pairs with an neural network developed in Keras.

class recordlinkage.adapters.**SKLearnAdapter**
SciKit-learn adapter for record pair classification.

SciKit-learn adapter for record pair classification with SciKit-learn models.

```
# import ScitKit-Learn classifier
from sklearn.ensemble import RandomForestClassifier

# import BaseClassifier from recordlinkage.base
from recordlinkage.base import BaseClassifier
from recordlinkage.adapters import SKLearnClassifier
from recordlinkage.datasets import binary_vectors

class RandomForest(SKLearnClassifier, BaseClassifier):

    def __init__(*args, **kwargs):
        super(self, RandomForest).__init__()

        # set the kernel
        kernel = RandomForestClassifier(*args, **kwargs)


# make a sample dataset
```

(continues on next page)

(continued from previous page)

```
features, links = binary_vectors(10000, 2000, return_links=True)

# initialise the random forest
cl = RandomForest(n_estimators=20)
cl.fit(features, links)

# predict the matches
cl.predict(...)
```

class recordlinkage.adapters.**KerasAdapter**

Keras adapter for record pair classification.

Keras adapter for record pair classification with Keras models.

Example of a Keras model used for classification.

```
from tensorflow.keras import layers
from recordlinkage.base import BaseClassifier
from recordlinkage.adapters import KerasAdapter

class NNClassifier(KerasAdapter, BaseClassifier):
    """Neural network classifier."""
    def __init__(self):
        super(NNClassifier, self).__init__()

        model = tf.keras.Sequential()
        model.add(layers.Dense(16, input_dim=8, activation='relu'))
        model.add(layers.Dense(8, activation='relu'))
        model.add(layers.Dense(1, activation='sigmoid'))
        model.compile(
            optimizer=tf.train.AdamOptimizer(0.001),
            loss='binary_crossentropy',
            metrics=['accuracy']
        )

        self.kernel = model

# initialise the model
cl = NNClassifier()
# fit the model to the data
cl.fit(X_train, links_true)
# predict the class of the data
cl.predict(X_pred)
```

8.3 User-defined algorithms

Classifiers can make use of the *recordlinkage.base.BaseClassifier* for algorithms. ScitKit-learn based models may want *recordlinkage.adapters.SKLearnAdapter* as subclass as well.

class recordlinkage.base.**BaseClassifier**

Base class for classification of records pairs.

This class contains methods for training the classifier. Distinguish different types of training, such as supervised and unsupervised learning.

learn (**args*, ***kwargs*)
[DEPRECATED] Use 'fit_predict'.

fit (*comparison_vectors*, *match_index=None*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors (or features) to train the model with.
- **match_index** (*pandas.MultiIndex*) – A pandas.MultiIndex object with the true matches. The MultiIndex contains only the true matches. Default None.

Note: A note in case of finding links within a single dataset (for example duplicate detection). Unsure that the training record pairs are from the lower triangular part of the dataset/matrix. See detailed information here: link.

---

fit_predict (*comparison_vectors*, *match_index=None*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors.
- **match_index** (*pandas.MultiIndex*) – The true matches.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

predict (*comparison_vectors*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – Dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. Use recordlinkage.options instead. Use the option *recordlinkage.set_option('classification.return_type', 'index')* instead.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

prob (*comparison_vectors*, *return_type=None*)
Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.
- **return_type** (*str*) – Deprecated. (default 'series')

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

Probabilistic models can use the Fellegi and Sunter base class. This class is used for the *recordlinkage.ECMClassifier* and the *recordlinkage.NaiveBayesClassifier*.

class recordlinkage.classifiers.**FellegiSunter**(*use_col_names=True*, **args*, ***kwargs*)

Fellegi and Sunter (1969) framework.

Meta class for probabilistic classification algorithms. The Fellegi and Sunter class is used for the *recordlinkage.NaiveBayesClassifier* and *recordlinkage.ECMClassifier*.

Parameters **use_col_names** (*bool*) – Use the column names of the pandas.DataFrame to identify the parameters. If False, the column index of the feature is used. Default True.

log_p
Log match probability as described in the FS framework.

Type float

log_m_probs
Log probability P(x_i=1|Match) as described in the FS framework.

Type np.ndarray

log_u_probs
Log probability P(x_i=1|Non-match) as described in the FS framework.

Type np.ndarray

log_weights
Log weights as described in the FS framework.

Type np.ndarray

p
Match probability as described in the FS framework.

Type float

m_probs
Probability P(x_i=1|Match) as described in the FS framework.

Type np.ndarray

u_probs
Probability P(x_i=1|Non-match) as described in the FS framework.

Type np.ndarray

weights
Weights as described in the FS framework.

Type np.ndarray

References

Fellegi, Ivan P and Alan B Sunter. 1969. "A theory for record linkage." Journal of the American Statistical Association 64(328):1183–1210.

---

log_p
Log match probability as described in the FS framework

log_m_probs
Log probability P(x_i=1|Match) as described in the FS framework

log_u_probs
Log probability P(x_i=1|Non-match) as described in the FS framework

log_weights
Log weights as described in the FS framework

p
Match probability as described in the FS framework

m_probs
Probability P(x_i=1|Match) as described in the FS framework

u_probs
Probability P(x_i=1|Non-match) as described in the FS framework

weights
Weights as described in the FS framework

8.4 Examples

Unsupervised learning with the ECM algorithm. [See example on Github.](https://github.com/J535D165/recordlinkage/examples/unsupervised_learning.py)

8.5 Network

The Python Record Linkage Toolkit provides network/graph analysis tools for classification of record pairs into matches and distinct pairs. The toolkit provides the functionality for one-to-one linking and one-to-many linking. It is also possible to detect all connected components which is useful in data deduplication.

class recordlinkage.**OneToOneLinking**(*method='greedy'*)
[EXPERIMENTAL] One-to-one linking

A record from dataset A can match at most one record from dataset B. For example, (a1, a2) are records from A and (b1, b2) are records from B. A linkage of (a1, b1), (a1, b2), (a2, b1), (a2, b2) is not one-to-one connected. One of the results of one-to-one linking can be (a1, b1), (a2, b2).

Parameters **method** (*str*) – The method to solve the problem. Only 'greedy' is supported at the moment.

Note: This class is experimental and might change in future versions.

---

compute(*links*)
Compute the one-to-one linking.

Parameters **links** (*pandas.MultiIndex*) – The pairs to apply linking to.

Returns *pandas.MultiIndex* – A one-to-one matched MultiIndex of record pairs.

class recordlinkage.**OneToManyLinking**(*level=0*, *method='greedy'*)
[EXPERIMENTAL] One-to-many linking

A record from dataset A can link multiple records from dataset B, but a record from B can link to only one record of dataset A. Use the *level* argument to switch A and B.

Parameters

- **level** (*int*) – The level of the MultiIndex to have the one relations. The options are 0 or 1 (incication the level of the MultiIndex). Default 0.
- **method** (*str*) – The method to solve the problem. Only 'greedy' is supported at the moment.

Example

Consider a MultiIndex with record pairs constructed from datasets A and B. To link a record from B to at most one record of B, use the following syntax:

> one_to_many = OneToManyLinking(0) > one_to_many.compute(links)

To link a record from B to at most one record of B, use:

> one_to_many = OneToManyLinking(1) > one_to_many.compute(links)

Note: This class is experimental and might change in future versions.

---

compute(*links*)

Compute the one-to-many matching.

Parameters **links** (*pandas.MultiIndex*) – The pairs to apply linking to.

Returns *pandas.MultiIndex* – A one-to-many matched MultiIndex of record pairs.

class recordlinkage.**ConnectedComponents**

[EXPERIMENTAL] Connected record pairs

This class identifies connected record pairs. Connected components are especially used in detecting duplicates in a single dataset.

Note: This class is experimental and might change in future versions.

---

compute(*links*)

Return the connected components.

Parameters **links** (*pandas.MultiIndex*) – The links to apply one-to-one matching on.

Returns *list of pandas.MultiIndex* – A list with pandas.MultiIndex objects. Each MultiIndex object represents a set of connected record pairs.

CHAPTER 9

4. Evaluation

Evaluation of classifications plays an important role in record linkage. Express your classification quality in terms accuracy, recall and F-score based on `true positives`, `false positives`, `true negatives` and `false negatives`.

recordlinkage.**reduction_ratio**(*links_pred*, **total*)
Compute the reduction ratio.

The reduction ratio is 1 minus the ratio candidate matches and the maximum number of pairs possible.

Parameters

- **links_pred** (*int, pandas.MultiIndex*) – The number of candidate record pairs or the pandas.MultiIndex with record pairs.
- ***total** (*pandas.DataFrame object(s)*) – The DataFrames are used to compute the full index size with the full_index_size function.

Returns *float* – The reduction ratio.

recordlinkage.**true_positives**(*links_true*, *links_pred*)
Count the number of True Positives.

Returns the number of correctly predicted links, also called the number of True Positives (TP).

Parameters

- **links_true** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The true (or actual) links.
- **links_pred** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The predicted links.

Returns *int* – The number of correctly predicted links.

recordlinkage.**true_negatives**(*links_true*, *links_pred*, *total*)
Count the number of True Negatives.

Returns the number of correctly predicted non-links, also called the number of True Negatives (TN).

Parameters

- **links_true** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The true (or actual) links.
- **links_pred** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The predicted links.
- **total** (*int, pandas.MultiIndex*) – The count of all record pairs (both links and non-links). When the argument is a pandas.MultiIndex, the length of the index is used.

Returns *int* – The number of correctly predicted non-links.

recordlinkage.**false_positives**(*links_true*, *links_pred*)

Count the number of False Positives.

Returns the number of incorrect predictions of true non-links. (true non- links, but predicted as links). This value is known as the number of False Positives (FP).

Parameters

- **links_true** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The true (or actual) links.
- **links_pred** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The predicted links.

Returns *int* – The number of false positives.

recordlinkage.**false_negatives**(*links_true*, *links_pred*)

Count the number of False Negatives.

Returns the number of incorrect predictions of true links. (true links, but predicted as non-links). This value is known as the number of False Negatives (FN).

Parameters

- **links_true** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The true (or actual) links.
- **links_pred** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The predicted links.

Returns *int* – The number of false negatives.

recordlinkage.**confusion_matrix**(*links_true*, *links_pred*, *total=None*)

Compute the confusion matrix.

The confusion matrix is of the following form:

	Predicted Positives	Predicted Negatives
True Positives	True Positives (TP)	False Negatives (FN)
True Negatives	False Positives (FP)	True Negatives (TN)

The confusion matrix is an informative way to analyse a prediction. The matrix can used to compute measures like precision and recall. The count of true prositives is [0,0], false negatives is [0,1], true negatives is [1,1] and false positives is [1,0].

Parameters

- **links_true** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The true (or actual) links.
- **links_pred** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The predicted links.

- **total** (*int, pandas.MultiIndex*) – The count of all record pairs (both links and non-links). When the argument is a pandas.MultiIndex, the length of the index is used. If the total is None, the number of True Negatives is not computed. Default None.

Returns *numpy.array* – The confusion matrix with TP, TN, FN, FP values.

Note: The number of True Negatives is computed based on the total argument. This argument is the number of record pairs of the entire matrix.

---

recordlinkage.**precision**(*links_true*, *links_pred*)

Compute the precision.

The precision is given by TP/(TP+FP).

Parameters

- **links_true** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The true (or actual) collection of links.
- **links_pred** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The predicted collection of links.

Returns *float* – The precision

recordlinkage.**recall**(*links_true*, *links_pred*)

Compute the recall/sensitivity.

The recall is given by TP/(TP+FN).

Parameters

- **links_true** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The true (or actual) collection of links.
- **links_pred** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The predicted collection of links.

Returns *float* – The recall

recordlinkage.**accuracy**(*links_true*, *links_pred*, *total*)

Compute the accuracy.

The accuracy is given by (TP+TN)/(TP+FP+TN+FN).

Parameters

- **links_true** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The true (or actual) collection of links.
- **links_pred** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The predicted collection of links.
- **total** (*int, pandas.MultiIndex*) – The count of all record pairs (both links and non-links). When the argument is a pandas.MultiIndex, the length of the index is used.

Returns *float* – The accuracy

recordlinkage.**specificity**(*links_true*, *links_pred*, *total*)

Compute the specificity.

The specificity is given by TN/(FP+TN).

Parameters

- **links_true** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The true (or actual) collection of links.
- **links_pred** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The predicted collection of links.
- **total** (*int, pandas.MultiIndex*) – The count of all record pairs (both links and non-links). When the argument is a pandas.MultiIndex, the length of the index is used.

Returns *float* – The specificity

recordlinkage.**fscore**(*links_true*, *links_pred*)

Compute the F-score.

The F-score is given by 2*(precision*recall)/(precision+recall).

Parameters

- **links_true** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The true (or actual) collection of links.
- **links_pred** (*pandas.MultiIndex, pandas.DataFrame, pandas.Series*) – The predicted collection of links.

Returns *float* – The fscore

Note: If there are no pairs predicted as links, this measure will raise a ZeroDivisionError.

---

recordlinkage.**max_pairs**(*shape*)

[DEPRECATED] Compute the maximum number of record pairs possible.

recordlinkage.**full_index_size**(**args*)

Compute the number of records in a full index.

Compute the number of records in a full index without building the index itself. The result is the maximum number of record pairs possible. This function is especially useful in measures like the *reduction_ratio*.

Deduplication: Given a DataFrame A with length N, the full index size is N*(N-1)/2. Linking: Given a DataFrame A with length N and a DataFrame B with length M, the full index size is N*M.

Parameters ***args** (*int, pandas.MultiIndex, pandas.Series, pandas.DataFrame*) – A pandas object or a int representing the length of a dataset to link. When there is one argument, it is assumed that the record linkage is a deduplication process.

Examples

Use integers: >>> full_index_size(10) # deduplication: 45 pairs >>> full_index_size(10, 10) # linking: 100 pairs

or pandas objects >>> full_index_size(DF) # deduplication: len(DF)*(len(DF)-1)/2 pairs >>> full_index_size(DF, DF) # linking: len(DF)*len(DF) pairs

CHAPTER 10

Datasets

The Python Record Linkage Toolkit contains several open public datasets. Four datasets were generated by the developers of Febrl. In the future, we are developing tools to generate your own datasets.

recordlinkage.datasets.**load_krebsregister**(*block=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], missing_values=None, shuffle=True*)

Load the Krebsregister dataset.

This dataset of comparison patterns was obtained in a epidemiological cancer study in Germany. The comparison patterns were created by the Institute for Medical Biostatistics, Epidemiology and Informatics (IMBEI) and the University Medical Center of Johannes Gutenberg University (Mainz, Germany). The dataset is available for research online.

"The records represent individual data including first and family name, sex, date of birth and postal code, which were collected through iterative insertions in the course of several years. The comparison patterns in this data set are based on a sample of 100.000 records dating from 2005 to 2008. Data pairs were classified as "match" or "non-match" during an extensive manual review where several documentarists were involved. The resulting classification formed the basis for assessing the quality of the registry's own record linkage procedure.

In order to limit the amount of patterns a blocking procedure was applied, which selects only record pairs that meet specific agreement conditions. The results of the following six blocking iterations were merged together:

- Phonetic equality of first name and family name, equality of date of birth.
- Phonetic equality of first name, equality of day of birth.
- Phonetic equality of first name, equality of month of birth.
- Phonetic equality of first name, equality of year of birth.
- Equality of complete date of birth.
- Phonetic equality of family name, equality of sex.

This procedure resulted in 5.749.132 record pairs, of which 20.931 are matches. The data set is split into 10 blocks of (approximately) equal size and ratio of matches to non-matches."

Parameters

- **block** (*int, list*) – An integer or a list with integers between 1 and 10. The blocks are the blocks explained in the description.
- **missing_values** (*object, int, float*) – The value of the missing values. Default NaN.
- **shuffle** (*bool*) – Shuffle the record pairs. Default True.

Returns *(pandas.DataFrame, pandas.MultiIndex)* – A pandas.DataFrame with comparison vectors and a pandas.MultiIndex with the indices of the matches.

recordlinkage.datasets.**load_febrl1**(*return_links=False*)

Load the FEBRL 1 dataset.

The Freely Extensible Biomedical Record Linkage (Febrl) package is distributed with a dataset generator and four datasets generated with the generator. This function returns the first Febrl dataset as a pandas.DataFrame.

"This data set contains 1000 records (500 original and 500 duplicates, with exactly one duplicate per original record."

Parameters **return_links** (*bool*) – When True, the function returns also the true links.

Returns *pandas.DataFrame* – A pandas.DataFrame with Febrl dataset1.csv. When return_links is True, the function returns also the true links. The true links are all links in the lower triangular part of the matrix.

recordlinkage.datasets.**load_febrl2**(*return_links=False*)

Load the FEBRL 2 dataset.

The Freely Extensible Biomedical Record Linkage (Febrl) package is distributed with a dataset generator and four datasets generated with the generator. This function returns the second Febrl dataset as a pandas.DataFrame.

"This data set contains 5000 records (4000 originals and 1000 duplicates), with a maximum of 5 duplicates based on one original record (and a poisson distribution of duplicate records). Distribution of duplicates: 19 originals records have 5 duplicate records 47 originals records have 4 duplicate records 107 originals records have 3 duplicate records 141 originals records have 2 duplicate records 114 originals records have 1 duplicate record 572 originals records have no duplicate record"

Parameters **return_links** (*bool*) – When True, the function returns also the true links.

Returns *pandas.DataFrame* – A pandas.DataFrame with Febrl dataset2.csv. When return_links is True, the function returns also the true links. The true links are all links in the lower triangular part of the matrix.

recordlinkage.datasets.**load_febrl3**(*return_links=False*)

Load the FEBRL 3 dataset.

The Freely Extensible Biomedical Record Linkage (Febrl) package is distributed with a dataset generator and four datasets generated with the generator. This function returns the third Febrl dataset as a pandas.DataFrame.

"This data set contains 5000 records (2000 originals and 3000 duplicates), with a maximum of 5 duplicates based on one original record (and a Zipf distribution of duplicate records). Distribution of duplicates: 168 originals records have 5 duplicate records 161 originals records have 4 duplicate records 212 originals records have 3 duplicate records 256 originals records have 2 duplicate records 368 originals records have 1 duplicate record 1835 originals records have no duplicate record"

Parameters **return_links** (*bool*) – When True, the function returns also the true links.

Returns *pandas.DataFrame* – A pandas.DataFrame with Febrl dataset3.csv. When return_links is True, the function returns also the true links. The true links are all links in the lower triangular part of the matrix.

recordlinkage.datasets.**load_febrl4**(*return_links=False*)

Load the FEBRL 4 datasets.

The Freely Extensible Biomedical Record Linkage (Febrl) package is distributed with a dataset generator and four datasets generated with the generator. This function returns the fourth Febrl dataset as a pandas.DataFrame.

"Generated as one data set with 10000 records (5000 originals and 5000 duplicates, with one duplicate per original), the originals have been split from the duplicates, into dataset4a.csv (containing the 5000 original records) and dataset4b.csv (containing the 5000 duplicate records) These two data sets can be used for testing linkage procedures."

Parameters **return_links** (*bool*) – When True, the function returns also the true links.

Returns *(pandas.DataFrame, pandas.DataFrame)* – A pandas.DataFrame with Febrl dataset4a.csv and a pandas dataframe with Febrl dataset4b.csv. When return_links is True, the function returns also the true links.

recordlinkage.datasets.**binary_vectors**(*n, n_match, m=[0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9], u=[0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1], random_state=None, return_links=False, dtype=<class 'numpy.int8'>*)

Generate random binary comparison vectors.

This function is used to generate random comparison vectors. The result of each comparison is a binary value (0 or 1).

Parameters

- **n** (*int*) – The total number of comparison vectors.
- **n_match** (*int*) – The number of matching record pairs.
- **m** (*list, default [0.9] * 8, optional*) – A list of m probabilities of each partially identifying variable. The m probability is the probability that an identifier in matching record pairs agrees.
- **u** (*list, default [0.9] * 8, optional*) – A list of u probabilities of each partially identifying variable. The u probability is the probability that an identifier in non-matching record pairs agrees.
- **random_state** (*int or numpy.random.RandomState, optional*) – Seed for the random number generator with an integer or numpy RandomState object.
- **return_links** (*bool*) – When True, the function returns also the true links.
- **dtype** (*numpy.dtype*) – The dtype of each column in the returned DataFrame.

Returns *pandas.DataFrame* – A dataframe with comparison vectors.

CHAPTER 11

Miscellaneous

recordlinkage.**index_split**(*index*, *chunks*)

Function to split pandas.Index and pandas.MultiIndex objects.

Split `pandas.Index` and `pandas.MultiIndex` objects into chunks. This function is based on `numpy.array_split()`.

Parameters

- **index** (*pandas.Index, pandas.MultiIndex*) – A pandas.Index or pandas.MultiIndex to split into chunks.
- **chunks** (*int*) – The number of parts to split the index into.

Returns *list* – A list with chunked pandas.Index or pandas.MultiIndex objects.

recordlinkage.**get_option**(*pat*)

Retrieves the value of the specified option.

The available options with its descriptions:

classification.return_type [str] The format of the classification result. The value 'index' returns the classification result as a pandas.MultiIndex. The MultiIndex contains the predicted matching record pairs. The value 'series' returns a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones. [default: index] [currently: index]

indexing.pairs [str] Specify the format how record pairs are stored. By default, record pairs generated by the toolkit are returned in a pandas.MultiIndex object ('multiindex' option).

Valid values: 'multiindex' [default: multiindex] [currently: multiindex]

Parameters **pat** (*str*) – Regexp which should match a single option. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. x.y.z.option_name), your code may break in future versions if new options with similar names are introduced.

Returns **result** (*the value of the option*)

Raises OptionError : if no such option exists

recordlinkage.**set_option**(*pat*, *value*)
Sets the value of the specified option.

The available options with its descriptions:

classification.return_type [str] The format of the classification result. The value 'index' returns the classification result as a pandas.MultiIndex. The MultiIndex contains the predicted matching record pairs. The value 'series' returns a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones. [default: index] [currently: index]

indexing.pairs [str] Specify the format how record pairs are stored. By default, record pairs generated by the toolkit are returned in a pandas.MultiIndex object ('multiindex' option).

Valid values: 'multiindex' [default: multiindex] [currently: multiindex]

Parameters

- **pat** (*str*) – Regexp which should match a single option. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. x.y.z.option_name), your code may break in future versions if new options with similar names are introduced.
- **value** – new value of option.

Returns *None*

Raises OptionError if no such option exists

recordlinkage.**reset_option**(*pat*)
Reset one or more options to their default value.

Pass "all" as argument to reset all options.

The available options with its descriptions:

classification.return_type [str] The format of the classification result. The value 'index' returns the classification result as a pandas.MultiIndex. The MultiIndex contains the predicted matching record pairs. The value 'series' returns a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones. [default: index] [currently: index]

indexing.pairs [str] Specify the format how record pairs are stored. By default, record pairs generated by the toolkit are returned in a pandas.MultiIndex object ('multiindex' option).

Valid values: 'multiindex' [default: multiindex] [currently: multiindex]

Parameters **pat** (*str/regex*) – If specified only options matching *prefix** will be reset. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. x.y.z.option_name), your code may break in future versions if new options with similar names are introduced.

Returns *None*

recordlinkage.**describe_option**(*pat*, *_print_desc=False*)
Prints the description for one or more registered options.

Call with not arguments to get a listing for all registered options.

The available options with its descriptions:

classification.return_type [str] The format of the classification result. The value 'index' returns the classification result as a pandas.MultiIndex. The MultiIndex contains the predicted matching record pairs. The value 'series' returns a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones. [default: index] [currently: index]

indexing.pairs [str] Specify the format how record pairs are stored. By default, record pairs generated by the toolkit are returned in a pandas.MultiIndex object ('multiindex' option).

Valid values: 'multiindex' [default: multiindex] [currently: multiindex]

Parameters

- **pat** (*str*) – Regexp pattern. All matching keys will have their description displayed.
- **_print_desc** (*bool, default True*) – If True (default) the description(s) will be printed to stdout. Otherwise, the description(s) will be returned as a unicode string (for testing).

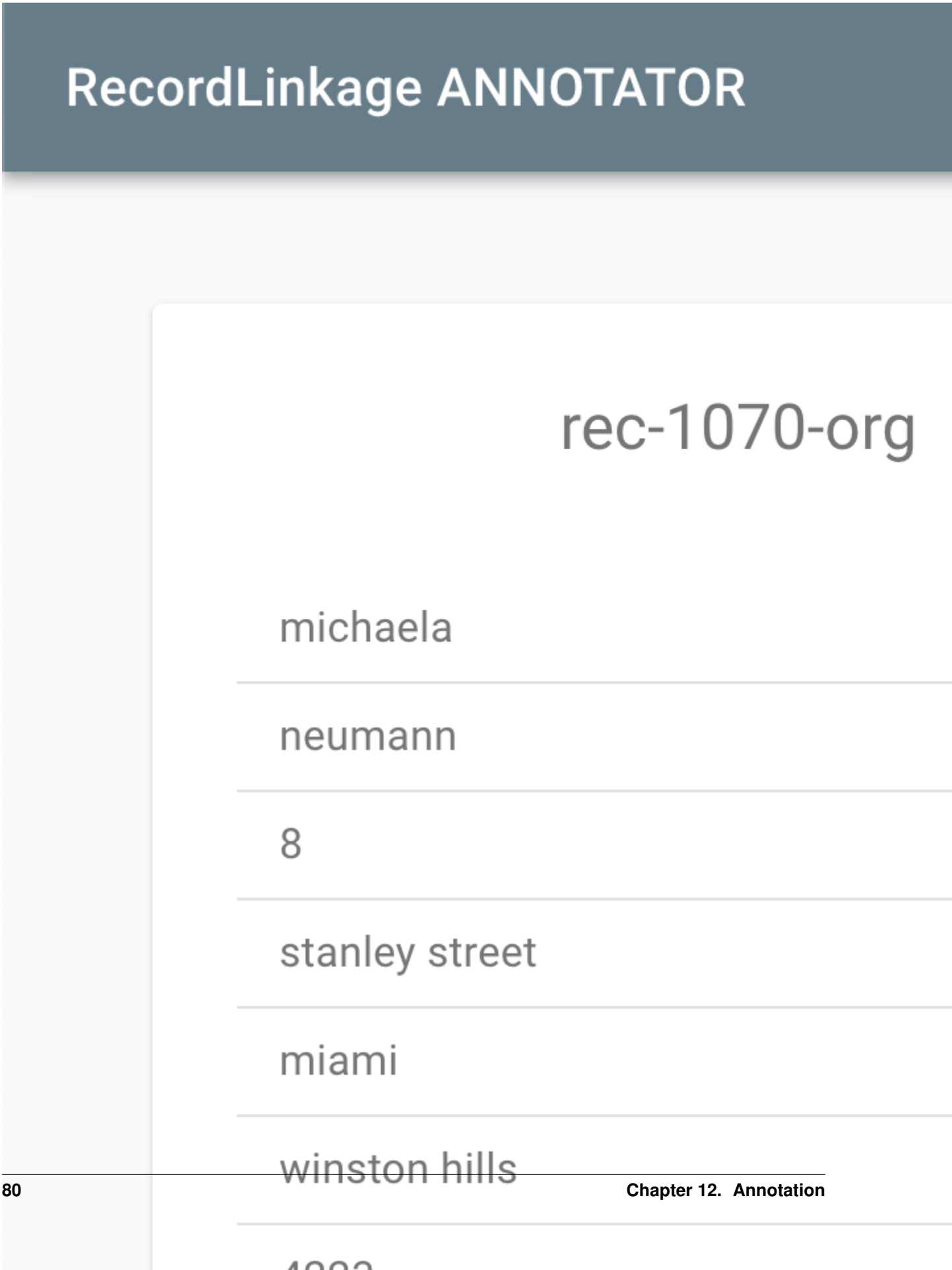
Returns

- *None by default, the description(s) as a unicode string if _print_desc*
- *is False*

CHAPTER 12

Annotation

Manually labeled record pairs are useful in training and validation tasks. Training data is usually not available in record linkage applications because it is highly dataset and sample-specific. The Python Record Linkage Toolkit comes with a browser-based user interface for manually classifying record pairs. A hosted version of RecordLinkage ANNOTATOR can be found on Github.


RecordLinkage ANNOTATOR
rec-1070-org
michaela
neumann
8
stanley street
miami
winston hills

12.1 Generate annotation file

The RecordLinkage ANNOTATOR software requires a structured annotation file. The required schema of the annotation file is open. The function *recordlinkage.write_annotation_file()* can be used to render and save an annotation file. The function can be used for both linking and deduplication purposes.

recordlinkage.**write_annotation_file**(*fp*, *pairs*, *df_a*, *df_b=None*, *dataset_a_name=None*, *dataset_b_name=None*, **args*, ***kwargs*)

Render and export annotation file.

This function renders and annotation object and stores it in a json file. The function is a wrapper around the *AnnotationWrapper* class.

Parameters

- **fp** (*str*) – The path to the annotation file.
- **pairs** (*pandas.MultiIndex*) – The record pairs to annotate.
- **df_a** (*pandas.DataFrame*) – The data frame with full record information for the pairs.
- **df_b** (*pandas.DataFrame*) – In case of data linkage, this is the second data frame. Default None.
- **dataset_a_name** (*str*) – The name of the first data frame.
- **dataset_b_name** (*str*) – In case of data linkage, the name of the second data frame. Default None.

12.1.1 Linking

This is a simple example of the code to render an annotation file for linking records:

```
import recordlinkage as rl
from recordlinkage.index import Block
from recordlinkage.datasets import load_febrl4

df_a, df_b = load_febrl4()

blocker = Block("surname", "surname")
pairs = blocker.index(df_a, df_b)

rl.write_annotation_file(
    "annotation_demo_linking.json",
    pairs[0:50],
    df_a,
    df_b,
    dataset_a_name="Febrl4 A",
    dataset_b_name="Febrl4 B"
)
```

12.1.2 Deduplication

This is a simple example of the code to render an annotation file for duplicate detection:

```
import recordlinkage as rl
from recordlinkage.index import Block
from recordlinkage.datasets import load_febrl1

df_a = load_febrl1()

blocker = Block("surname", "surname")
pairs = blocker.index(df_a)

rl.write_annotation_file(
    "annotation_demo_dedup.json",
    pairs[0:50],
    df_a,
    dataset_a_name="Febrl1 A"
)
```

12.2 Manual labeling

Go to RecordLinkage ANNOTATOR or start the server yourself.

Choose the annotation file on the landing screen or use the drag and drop functionality. A new screen shows the first record pair to label. Start labeling data the manually. Use the button *Match* for record pairs belonging to the same entity. Use *Distinct* for record pairs belonging to different entities. After all records are labeled by hand, the result can be saved to a file.

12.3 Export/read annotation file

After labeling all record pairs, you can export the annotation file to a JSON file. Use the function *recordlinkage.read_annotation_file()* to read the results.

```
import recordlinkage as rl

result = rl.read_annotation_file('my_annotation.json')
print(result.links)
```

The function *recordlinkage.read_annotation_file()* reads the file and returns an *recordlinkage.annotation.AnnotationResult* object. This object contains links and distinct attributes that return a pandas.MultiIndex object.

recordlinkage.**read_annotation_file**(*fp*)

Read annotation file.

This function can be used to read the annotation file and extract the results like the linked pairs and distinct pairs.

Parameters **fp** (*str*) – The path to the annotation file.

Returns *AnnotationResult* – An AnnotationResult object.

Example

Read the links from an annotation file:

```
> annotation = read_annotation_file("result.json")
> print(annotation.links)
```

class recordlinkage.annotation.**AnnotationResult**(*pairs=[]*, *version=1*)

Result of (manual) annotation.

Parameters

- **pairs** (*list*) – Raw data of each record pair in the annotation file.
- **version** (*str*) – The version number corresponding to the file structure.

links

Return the links.

Returns *pandas.MultiIndex* – The links stored in a pandas MultiIndex.

distinct

Return the distinct pairs.

Returns *pandas.MultiIndex* – The distinct pairs stored in a pandas MultiIndex.

unknown

Return the unknown or unlabalbled pairs.

Returns *pandas.MultiIndex* – The unknown or unlabalbled pairs stored in a pandas MultiIndex.

classmethod from_dict(*d*)

Create AnnotationResult from dict

Parameters **d** (*dict*) – The annotation file as a dict.

Returns *AnnotationResult* – An AnnotationResult object.

classmethod from_file(*fp*)

Create AnnotationResult from file

Parameters **fp** (*str*) – The path to the annotation file.

Returns *AnnotationResult* – An AnnotationResult object.

CHAPTER 13

Classification algorithms

In the context of record linkage, classification refers to the process of dividing record pairs into matches and non-matches (distinct pairs). There are dozens of classification algorithms for record linkage. Roughly speaking, classification algorithms fall into two groups:

- **supervised learning algorithms** - These algorithms make use of trainings data. If you do have trainings data, then you can use supervised learning algorithms. Most supervised learning algorithms offer good accuracy and reliability. Examples of supervised learning algorithms in the *Python Record Linkage Toolkit* are *Logistic Regression*, *Naive Bayes* and *Support Vector Machines*.
- **unsupervised learning algorithms** - These algorithms do not need training data. The *Python Record Linkage Toolkit* supports *K-means clustering* and an *Expectation/Conditional Maximisation* classifier.

First things first

The examples below make use of the Krebs register (German for cancer registry) dataset. The Krebs register dataset contains comparison vectors of a large set of record pairs. For each record pair, it is known if the records represent the same person (match) or not (non-match). This was done with a massive clerical review. First, import the recordlinkage module and load the Krebs register data. The dataset contains 5749132 compared record pairs and has the following variables: first name, last name, sex, birthday, birth month, birth year and zip code. The Krebs register contains `len(krebs_true_links) == 20931` matching record pairs.

[2]:
```
import recordlinkage as rl
from recordlinkage.datasets import load_krebsregister

krebs_X, krebs_true_links = load_krebsregister(missing_values=0)
krebs_X
```

[2]:
```
             cmp_firstname1  cmp_firstname2  cmp_lastname1  cmp_lastname2  \
id1   id2
22161 38467         1.00000             0.0        0.14286            0.0
38713 75352         0.00000             0.0        0.57143            0.0
13699 32825         0.16667             0.0        0.00000            0.0
22709 37682         0.28571             0.0        1.00000            0.0
2342  69060         0.25000             0.0        0.12500            0.0
...                     ...             ...            ...            ...
```

(continues on next page)

(continued from previous page)

```
52124 53629         1.00000             0.0         0.28571             0.0
30007 76846         0.75000             0.0         0.00000             0.0
50546 59461         0.75000             0.0         0.00000             0.0
43175 62151         1.00000             0.0         0.11111             0.0
11651 57925         1.00000             0.0         0.00000             0.0

             cmp_sex  cmp_birthday  cmp_birthmonth  cmp_birthyear  cmp_zipcode
id1   id2
22161 38467        1           0.0             1.0            0.0          0.0
38713 75352        1           0.0             0.0            0.0          0.0
13699 32825        0           1.0             1.0            1.0          0.0
22709 37682        1           0.0             0.0            0.0          0.0
2342  69060        1           1.0             1.0            1.0          0.0
...              ...           ...             ...            ...          ...
52124 53629        1           0.0             0.0            1.0          0.0
30007 76846        1           1.0             0.0            0.0          0.0
50546 59461        1           0.0             1.0            0.0          0.0
43175 62151        1           0.0             1.0            0.0          0.0
11651 57925        1           0.0             0.0            1.0          0.0

[5749132 rows x 9 columns]
```

Most classifiers can not handle comparison vectors with missing values. To prevent issues with the classification algorithms, we convert the missing values into disagreeing comparisons (using argument missing_values=0). This approach for handling missing values is widely used in record linkage applications.

[3]:
```
krebs_X.describe().T
```

[3]:
```
                    count     mean      std  min      25%      50%      75%  \
cmp_firstname1  5749132.0  0.71278  0.38884  0.0  0.28571  1.00000  1.00000
cmp_firstname2  5749132.0  0.01623  0.12520  0.0  0.00000  0.00000  0.00000
cmp_lastname1   5749132.0  0.31563  0.33423  0.0  0.10000  0.18182  0.42857
cmp_lastname2   5749132.0  0.00014  0.01008  0.0  0.00000  0.00000  0.00000
cmp_sex         5749132.0  0.95500  0.20730  0.0  1.00000  1.00000  1.00000
cmp_birthday    5749132.0  0.22443  0.41721  0.0  0.00000  0.00000  0.00000
cmp_birthmonth  5749132.0  0.48879  0.49987  0.0  0.00000  0.00000  1.00000
cmp_birthyear   5749132.0  0.22272  0.41607  0.0  0.00000  0.00000  0.00000
cmp_zipcode     5749132.0  0.00552  0.07407  0.0  0.00000  0.00000  0.00000

                max
cmp_firstname1  1.0
cmp_firstname2  1.0
cmp_lastname1   1.0
cmp_lastname2   1.0
cmp_sex         1.0
cmp_birthday    1.0
cmp_birthmonth  1.0
cmp_birthyear   1.0
cmp_zipcode     1.0
```

13.1 Supervised learning

As described before, supervised learning algorithms do need training data. Training data is data for which the true match status is known for each comparison vector. In the example in this section, we consider that the true match status of the first 5000 record pairs of the Krebs register data is known.

[4]:
```
golden_pairs = krebs_X[0:5000]
golden_matches_index = golden_pairs.index & krebs_true_links # 2093 matching pairs
```

13.1.1 Logistic regression

The `recordlinkage.LogisticRegressionClassifier` classifier is an application of the logistic regression model. This supervised learning method is one of the oldest classification algorithms used in record linkage. In situations with enough training data, the algorithm gives relatively good results.

[5]:
```
# Initialize the classifier
logreg = rl.LogisticRegressionClassifier()

# Train the classifier
logreg.fit(golden_pairs, golden_matches_index)
print ("Intercept: ", logreg.intercept)
print ("Coefficients: ", logreg.coefficients)
```

```
Intercept:  -6.298043571006437
Coefficients:  [ 4.90452843e-01  1.21640484e-01  2.15040485e+00 -2.84818101e-03
 -1.79712465e+00  9.61085558e-01  6.72610441e-02  1.03408608e+00
  4.30556110e+00]
```

[6]:
```
# Predict the match status for all record pairs
result_logreg = logreg.predict(krebs_X)

len(result_logreg)
```

[6]:
```
20150
```

[7]:
```
rl.confusion_matrix(krebs_true_links, result_logreg, len(krebs_X))
```

[7]:
```
array([[  19884,    1047],
       [    266, 5727935]])
```

[8]:
```
# The F-score for this prediction is
rl.fscore(krebs_true_links, result_logreg)
```

[8]:
```
0.96804
```

The predicted number of matches is not much more than the 20931 true matches. The result was achieved with a small training dataset of 5000 record pairs.

In (older) literature, record linkage procedures are often divided in **deterministic record linkage** and **probabilistic record linkage**. The Logistic Regression Classifier belongs to deterministic record linkage methods. Each feature/variable has a certain importance (named weight). The weight is multiplied with the comparison/similarity vector. If the total sum exceeds a certain threshold, it as considered to be a match.

[9]:
```
intercept = -9
coefficients = [2.0, 1.0, 3.0, 1.0, 1.0, 1.0, 1.0, 2.0, 3.0]

logreg = rl.LogisticRegressionClassifier(coefficients, intercept)

# predict without calling LogisticRegressionClassifier.fit
result_logreg_pretrained = logreg.predict(krebs_X)
print (len(result_logreg_pretrained))
```

```
21303
```

[10]:
```
rl.confusion_matrix(krebs_true_links, result_logreg_pretrained, len(krebs_X))
```

[10]:
```
array([[  20857,      74],
       [    446, 5727755]])
```

[11]:
```
# The F-score for this classification is
rl.fscore(krebs_true_links, result_logreg_pretrained)
```

[11]:
```
0.98769
```

For the given coefficients, the F-score is better than the situation without trainings data. Surprising? No (use more trainings data and the result will improve)

13.1.2 Naive Bayes

In contrast to the logistic regression classifier, the Naive Bayes classifier is a probabilistic classifier. The probabilistic record linkage framework by Fellegi and Sunter (1969) is the most well-known probabilistic classification method for record linkage. Later, it was proved that the Fellegi and Sunter method is mathematically equivalent to the Naive Bayes method in case of assuming independence between comparison variables.

[12]:
```
# Train the classifier
nb = rl.NaiveBayesClassifier(binarize=0.3)
nb.fit(golden_pairs, golden_matches_index)

# Predict the match status for all record pairs
result_nb = nb.predict(krebs_X)

len(result_nb)
```

[12]:
```
19837
```

[13]:
```
rl.confusion_matrix(krebs_true_links, result_nb, len(krebs_X))
```

[13]:
```
array([[  19825,    1106],
       [     12, 5728189]])
```

[14]:
```
# The F-score for this classification is
rl.fscore(krebs_true_links, result_nb)
```

[14]:
```
0.97258
```

13.1.3 Support Vector Machines

Support Vector Machines (SVM) have become increasingly popular in record linkage. The algorithm performs well there is only a small amount of training data available. The implementation of SVM in the Python Record Linkage Toolkit is a linear SVM algorithm.

[15]:
```
# Train the classifier
svm = rl.SVMClassifier()
svm.fit(golden_pairs, golden_matches_index)

# Predict the match status for all record pairs
```

(continues on next page)

(continued from previous page)

```
result_svm = svm.predict(krebs_X)

len(result_svm)
```

[15]:
```
20839
```

[16]:
```
rl.confusion_matrix(krebs_true_links, result_svm, len(krebs_X))
```

[16]:
```
array([[  20825,     106],
       [     14, 5728187]])
```

[17]:
```
# The F-score for this classification is
rl.fscore(krebs_true_links, result_svm)
```

[17]:
```
0.99713
```

13.2 Unsupervised learning

In situations without training data, unsupervised learning can be a solution for record linkage problems. In this section, we discuss two unsupervised learning methods. One algorithm is K-means clustering, and the other algorithm is an implementation of the Expectation-Maximisation algorithm. Most of the time, unsupervised learning algorithms take more computational time because of the iterative structure in these algorithms.

13.2.1 K-means clustering

The K-means clustering algorithm is well-known and widely used in big data analysis. The K-means classifier in the Python Record Linkage Toolkit package is configured in such a way that it can be used for linking records. For more info about the K-means clustering see Wikipedia.

[18]:
```
kmeans = rl.KMeansClassifier()
result_kmeans = kmeans.fit_predict(krebs_X)

# The predicted number of matches
len(result_kmeans)
```

[18]:
```
371525
```

The classifier is now trained and the comparison vectors are classified.

[19]:
```
rl.confusion_matrix(krebs_true_links, result_kmeans, len(krebs_X))
```

[19]:
```
array([[  20797,     134],
       [ 350728, 5377473]])
```

[ ]:
```
rl.fscore(krebs_true_links, result_kmeans)
```

```
0.10598
```

13.2.2 Expectation/Conditional Maximization Algorithm

The ECM-algorithm is an Expectation-Maximisation algorithm with some additional constraints. This algorithm is closely related to the Naive Bayes algorithm. The ECM algorithm is also closely related to estimating the parameters

in the Fellegi and Sunter (1969) framework. The algorithms assume that the attributes are independent of each other. The Naive Bayes algorithm uses the same principles.

[ ]:
```
# Train the classifier
ecm = rl.ECMClassifier(binarize=0.8)
result_ecm = ecm.fit_predict(krebs_X)

len(result_ecm)
```

[ ]:
```
rl.confusion_matrix(krebs_true_links, result_ecm, len(krebs_X))
```

[ ]:
```
# The F-score for this classification is
rl.fscore(krebs_true_links, result_ecm)
```

CHAPTER 14

Performance

Performance plays an important role in record linkage. Record linkage problems scale quadratically with the size of the dataset(s). The number of record pairs can be enormous and so are the number of comparisons. The Python Record Linkage Toolkit can be used for large scale record linkage applications. Nevertheless, the toolkit is developed with experimenting in first place and performance on the second place. This page provides tips and tricks to improve the performance.

Do you know more tricks? Let us know!

14.1 Indexing

14.1.1 Block on multiple columns

Blocking is an effective way to increase the performance of your record linkage. If the performance of your implementation is still poor, decrease the number of pairs by blocking on multiple variables. This implies that the record pair is agrees on two or more variables. In the following example, the record pairs agree on the given name **and** surname.

```
from recordlinkage.index import Block
indexer = Block(left_on=['first_name', 'surname'],
                                right_on=['name', 'surname'])
pairs = indexer.index(dfA, dfB)
```

You might exclude more links then desired. This can be solved by repeating the process with different blocking variables.

```
indexer = recordlinkage.Index()
indexer.block(left_on=['first_name', 'surname'],
              right_on=['name', 'surname'])
indexer.block(left_on=['first_name', 'age'],
              right_on=['name', 'age'])
pairs = indexer.index(dfA, dfB)
```

Note: Sorted Neighbourhood indexing supports, besides the sorted neighbourhood, additional blocking on variables.

14.1.2 Make record pairs

The structure of the Python Record Linkage Toolkit has a drawback for the performance. In the indexation step (the step in which record pairs are selected), only the index of both records is stored. The entire records are not stored. This results in less memory usage. The drawback is that the records need to be queried from the data.

14.2 Comparing

14.2.1 Compare only discriminating variables

Not all variables may be worth comparing in a record linkage. Some variables do not discriminate the links of the non-links or do have only minor effects. These variables can be excluded. Only discriminating and informative should be included.

14.2.2 Prevent string comparisons

String similarity measures and phonetic encodings are computationally expensive. Phonetic encoding takes place on the original data, while string simililatiry measures are applied on the record pairs. After phonetic encoding of the string variables, exact comparing can be used instead of computing the string similarity of all record pairs. If the number of candidate pairs is much larger than the number of records in both datasets together, then consider using phonetic encoding of string variables instead of string comparison.

14.2.3 String comparing

Comparing strings is computationally expensive. The Python Record Linkage Toolkit uses the package `jellyfish` for string comparisons. The package has two implementations, a C and a Python implementation. Ensure yourself of having the C-version installed (`import jellyfish.cjellyfish` should not raise an exception).

There can be a large difference in the performance of different string comparison algorithms. The Jaro and Jaro-Winkler methods are faster than the Levenshtein distance and much faster than the Damerau-Levenshtein distance.

14.2.4 Indexing with large files

Sometimes, the input files are very large. In that case, it can be hard to make an index without running out of memory in the indexing step or in the comparing step. `recordlinkage` has a method to deal with large files. It is fast, although is not primary developed to be fast. SQL databases may outperform this method. It is especially developed for the useability. The idea was to splllit the input files into small blocks. For each block the record pairs are computed. Then iterate over the blocks. Consider full indexing:

```
import recordlinkage
import numpy

cl = recordlinkage.index.Full()

for dfB_subset in numpy.split(dfB):
```

(continues on next page)

(continued from previous page)

```
    # a subset of record pairs
    pairs_subset = cl.index(dfA, dfB_subset)

    # Your analysis on pairs_subset here
```

CHAPTER 15

Contributing

Thanks for your interest in contributing to the Python Record Linkage Toolkit. There is a lot of work to do. See Github for the contributors to this package.

The workflow for contributing is as follows:

- clone https://github.com/J535D165/recordlinkage.git
- Make a branch with your modifications/contributions
- Write tests
- Run all tests
- Do a pull request

15.1 Testing

Install *pytest*:

```
pip install pytest
```

Run the following command to test the package

```
python -m pytest tests/
```

15.2 Performance

Performance is very important in record linkage. The performance is monitored for all serious modifications of the core API. The performance monitoring is performed with Airspeed Velocity (asv).

Install Airspeed Velocity:

```
pip install asv
```

Run the following command from the root of the repository to test the performance of the current version of the package:

```
asv run
```

Run the following command to test all versions since tag v0.6.0

```
asv run --skip-existing-commits v0.6.0..master
```

CHAPTER 16

Release notes

16.1 Version 0.14.0

- Drop Python 2.7 and Python 3.4 support. (#91)
- Upgrade minimal pandas version to 0.23.
- Simplify the use of all cpus in parallel mode. (#102)
- Store large example datasets in user home folder or use environment variable. Before, example datasets were stored in the package. (see issue #42) (#92)
- Add support to write and read annotation files for recordlinkage ANNOTATOR. See the docs and https://github.com/J535D165/recordlinkage-annotator for more information.
- Replace *.labels* by *.codes* for *pandas.MultiIndex* objects for newer versions of pandas (>0.24). (#103)
- Fix totals for pandas.MultiIndex input on confusion matrix and accuracy metrics. (see issue #84) (#109)
- Initialize Compare with (a list of) features (Bug). (#124)
- Various updates in relation to deprecation warnings in third-party libraries such as sklearn, pandas and networkx.

Bibliography

[christen2012] Christen, P. (2012). Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer Science & Business Media.

[christen2008] Christen, P. (2008). Febrl - A Freely Available Record Linkage System with a Graphical User Interface.

[Christen2012] Christen, Peter. 2012. Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer Science & Business Media.

Index

Symbols

A

B

C

D

E

F

G

I

K

L

V

W