dedupe Documentation

Release 1.9.9

Forest Gregg, Derek Eder, and contributors

Mar 09, 2020

CONTENTS

dedupe is a library that uses machine learning to perform de-duplication and entity resolution quickly on structured data.

If you're looking for the documentation for the Dedupe.io Web API, you can find that here: https://apidocs.dedupe.io/

dedupe will help you:

- **remove duplicate entries** from a spreadsheet of names and addresses
- **link a list** with customer information to another with order history, even without unique customer id's
- take a database of campaign contributions and **figure out which ones were made by the same person**, even if the names were entered slightly differently for each record

dedupe takes in human training data and comes up with the best rules for your dataset to quickly and automatically find similar records, even with very large databases.

CHAPTER ONE

IMPORTANT LINKS

- Documentation: https://docs.dedupe.io/
- Repository: https://github.com/dedupeio/dedupe
- Issues: https://github.com/dedupeio/dedupe/issues
- Mailing list: https://groups.google.com/forum/#!forum/open-source-deduplication
- Examples: https://github.com/dedupeio/dedupe-examples
- IRC channel, #dedupe on irc.freenode.net

CHAPTER TWO

TOOLS BUILT WITH DEDUPE

Dedupe.io A full service web service powered by dedupe for de-duplicating and find matches in your messy data. It provides an easy-to-use interface and provides cluster review and automation, as well as advanced record linkage, continuous matching and API integrations. See the product page and the launch blog post.

csvdedupe Command line tool for de-duplicating and linking CSV files. Read about it on Source Knight-Mozilla OpenNews.

CHAPTER

THREE

CONTENTS

3.1 Library Documentation

3.1.1 Dedupe Objects

class dedupe.**Dedupe**(*variable_definition*, *num_cores=None*, ***kwargs*)

Class for active learning deduplication. Use deduplication when you have data that can contain multiple records that can all refer to the same entity.

Parameters

- **variable_definition** (Sequence[Mapping]) – A list of dictionaries describing the variables will be used for training a model. See *Variable Definitions*
- **num_cores** (Optional[int]) – the number of cpus to use for parallel processing. If set to None, uses all cpus available on the machine. If set to 0, then multiprocessing will be disabled.

> **Warning:** If using multiprocessing on Windows or Mac OS X, then you must protect calls to the Dedupe methods with a `if __name__ == '__main__'` in your main module, see https://docs.python.org/3/library/multiprocessing.html#the-spawn-and-forkserver-start-methods

```
# initialize from a defined set of fields
variables = [{'field' : 'Site name', 'type': 'String'},
             {'field' : 'Address', 'type': 'String'},
             {'field' : 'Zip', 'type': 'String', 'has missing':True},
             {'field' : 'Phone', 'type': 'String', 'has missing':True}
             ]

deduper = dedupe.Dedupe(variables)
```

prepare_training(*data*, *training_file=None*, *sample_size=1500*, *blocked_proportion=0.9*, *original_length=None*)

Initialize the active learner with your data and, optionally, existing training data.

Sets up the learner.

Parameters

- **data** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records, where the keys are record_ids and the values are dictionaries with the keys being field names

- **training_file** (Optional[TextIO]) – file object containing training data
- **sample_size** (int) – Size of the sample to draw
- **blocked_proportion** (float) – The proportion of record pairs to be sampled from similar records, as opposed to randomly selected pairs. Defaults to 0.9.
- **original_length** (Optional[int]) – If `data` is a subsample of all your data, `original_length` should be the size of your complete data. By default, `original_length` defaults to the length of `data`.

```
matcher.prepare_training(data_d, 150000, .5)

# or
with open('training_file.json') as f:
    matcher.prepare_training(data_d, training_file=f)
```

uncertain_pairs()

Returns a list of pairs of records from the sample of record pairs tuples that Dedupe is most curious to have labeled.

This method is mainly useful for building a user interface for training a matching model.

```
> pair = matcher.uncertain_pairs()
> print(pair)
[({'name' : 'Georgie Porgie'}, {'name' : 'Georgette Porgette'})]
```

mark_pairs(*labeled_pairs*)

Add users labeled pairs of records to training data and update the matching model

This method is useful for building a user interface for training a matching model or for adding training data from an existing source.

Parameters **labeled_pairs** (TrainingData) – A dictionary with two keys, `match` and `distinct` the values are lists that can contain pairs of records

```
labeled_examples = {'match'    : [],
                    'distinct' : [({'name' : 'Georgie Porgie'},
                                   {'name' : 'Georgette Porgette'})]
                    }
matcher.mark_pairs(labeled_examples)
```

train(*recall=0.95*, *index_predicates=True*)

Learn final pairwise classifier and fingerprinting rules. Requires that adequate training data has been already been provided.

Parameters

- **recall** (float) – The proportion of true dupe pairs in our training data that that the learned fingerprinting rules must cover. If we lower the recall, there will be pairs of true dupes that we will never directly compare.

  recall should be a float between 0.0 and 1.0.

- **index_predicates** (bool) – Should dedupe consider predicates that rely upon indexing the data. Index predicates can be slower and take substantial memory.

write_training(*file_obj*)

Write a JSON file that contains labeled examples

Parameters **file_obj** (TextIO) – file object to write training data to

```
with open('training.json', 'w') as f:
    matcher.write_training(f)
```

write_settings (*file_obj*)

Write a settings file containing the data model and predicates to a file object

Parameters **file_obj** (BinaryIO) – file object to write settings data into

```
with open('learned_settings', 'wb') as f:
    matcher.write_settings(f)
```

cleanup_training ()

Clean up data we used for training. Free up memory.

partition (*data*, *threshold=0.5*)

Identifies records that all refer to the same entity, returns tuples containing a sequence of record ids and corresponding sequence of confidence score as a float between 0 and 1. The record_ids within each set should refer to the same entity and the confidence score is a measure of our confidence a particular entity belongs in the cluster.

This method should only used for small to moderately sized datasets for larger data, you need may need to generate your own pairs of records and feed them to *score()*.

Parameters

- **data** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records, where the keys are record_ids and the values are dictionaries with the keys being field names
- **threshold** (float) – Number between 0 and 1 (Default is 0.5). We will only consider put together records into clusters if the cophenetic similarity of the cluster is greater than the threshold.

  Lowering the number will increase recall, raising it will increase precision

```
> clusters = matcher.partition(data, threshold=0.5)
> print(duplicates)
[((1, 2, 3), (0.790, 0.860, 0.790)),
 ((4, 5), (0.720, 0.720)),
 ((10, 11), (0.899, 0.899))]
```

3.1.2 StaticDedupe Objects

class dedupe.**StaticDedupe** (*settings_file*, *num_cores=None*, ***kwargs*)

Class for deduplication using saved settings. If you have already trained a *Dedupe* object and saved the settings, you can load the saved settings with StaticDedupe.

Parameters

- **settings_file** (BinaryIO) – A file object containing settings info produced from the write_settings() method.
- **num_cores** (Optional[int]) – the number of cpus to use for parallel processing, defaults to the number of cpus available on the machine. If set to 0, then multiprocessing will be disabled.

> **Warning:** If using multiprocessing on Windows or Mac OS X, then you must protect calls to the Dedupe methods with a `if __name__ == '__main__'` in your main module, see https://docs.python.org/3/library/multiprocessing.html#the-spawn-and-forkserver-start-methods

```
with open('learned_settings', 'rb') as f:
    matcher = StaticDedupe(f)
```

partition (*data*, *threshold=0.5*)

Identifies records that all refer to the same entity, returns tuples containing a sequence of record ids and corresponding sequence of confidence score as a float between 0 and 1. The record_ids within each set should refer to the same entity and the confidence score is a measure of our confidence a particular entity belongs in the cluster.

This method should only used for small to moderately sized datasets for larger data, you need may need to generate your own pairs of records and feed them to *score()*.

Parameters

- **data** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records, where the keys are record_ids and the values are dictionaries with the keys being field names
- **threshold** (float) – Number between 0 and 1 (Default is 0.5). We will only consider put together records into clusters if the cophenetic similarity of the cluster is greater than the threshold.

  Lowering the number will increase recall, raising it will increase precision

```
> clusters = matcher.partition(data, threshold=0.5)
> print(duplicates)
[((1, 2, 3), (0.790, 0.860, 0.790)),
 ((4, 5), (0.720, 0.720)),
 ((10, 11), (0.899, 0.899))]
```

3.1.3 RecordLink Objects

class dedupe.**RecordLink** (*variable_definition*, *num_cores=None*, ***kwargs*)

Class for active learning record linkage.

Use RecordLinkMatching when you have two datasets that you want to join.

Parameters

- **variable_definition** (Sequence[Mapping]) – A list of dictionaries describing the variables will be used for training a model. See *Variable Definitions*
- **num_cores** (Optional[int]) – the number of cpus to use for parallel processing. If set to None, uses all cpus available on the machine. If set to 0, then multiprocessing will be disabled.

> **Warning:** If using multiprocessing on Windows or Mac OS X, then you must protect calls to the Dedupe methods with a `if __name__ == '__main__'` in your main module, see https://docs.python.org/3/library/multiprocessing.html#the-spawn-and-forkserver-start-methods

```
# initialize from a defined set of fields
variables = [{'field' : 'Site name', 'type': 'String'},
             {'field' : 'Address', 'type': 'String'},
             {'field' : 'Zip', 'type': 'String', 'has missing':True},
             {'field' : 'Phone', 'type': 'String', 'has missing':True}
             ]

deduper = dedupe.RecordLink(variables)
```

prepare_training(*data_1*, *data_2*, *training_file=None*, *sample_size=15000*, *blocked_proportion=0.5*, *original_length_1=None*, *original_length_2=None*)

Initialize the active learner with your data and, optionally, existing training data.

Parameters

- **data_1** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records from first dataset, where the keys are record_ids and the values are dictionaries with the keys being field names
- **data_2** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records from second dataset, same form as data_1
- **training_file** (Optional[TextIO]) – file object containing training data
- **sample_size** (int) – The size of the sample to draw. Defaults to 150,000
- **blocked_proportion** (float) – The proportion of record pairs to be sampled from similar records, as opposed to randomly selected pairs. Defaults to 0.5.
- **original_length_1** (Optional[int]) – If data_1 is a subsample of your first dataset, original_length_1 should be the size of the complete first dataset. By default, original_length_1 defaults to the length of data_1
- **original_length_2** (Optional[int]) – If data_2 is a subsample of your first dataset, original_length_2 should be the size of the complete first dataset. By default, original_length_2 defaults to the length of data_2

```
matcher.prepare_training(data_1, data_2, 150000)

with open('training_file.json') as f:
    matcher.prepare_training(data_1, data_2, training_file=f)
```

uncertain_pairs()

Returns a list of pairs of records from the sample of record pairs tuples that Dedupe is most curious to have labeled.

This method is mainly useful for building a user interface for training a matching model.

```
> pair = matcher.uncertain_pairs()
> print(pair)
[({'name' : 'Georgie Porgie'}, {'name' : 'Georgette Porgette'})]
```

mark_pairs(*labeled_pairs*)

Add users labeled pairs of records to training data and update the matching model

This method is useful for building a user interface for training a matching model or for adding training data from an existing source.

Parameters **labeled_pairs** (TrainingData) – A dictionary with two keys, match and distinct the values are lists that can contain pairs of records

```
labeled_examples = {'match'    : [],
                    'distinct' : [({'name' : 'Georgie Porgie'},
                                   {'name' : 'Georgette Porgette'})]
                    }
matcher.mark_pairs(labeled_examples)
```

train (*recall=0.95*, *index_predicates=True*)

Learn final pairwise classifier and fingerprinting rules. Requires that adequate training data has been already been provided.

Parameters

- **recall** (float) – The proportion of true dupe pairs in our training data that that the learned fingerprinting rules must cover. If we lower the recall, there will be pairs of true dupes that we will never directly compare.

  recall should be a float between 0.0 and 1.0.

- **index_predicates** (bool) – Should dedupe consider predicates that rely upon indexing the data. Index predicates can be slower and take substantial memory.

write_training (*file_obj*)

Write a JSON file that contains labeled examples

Parameters **file_obj** (TextIO) – file object to write training data to

```
with open('training.json', 'w') as f:
    matcher.write_training(f)
```

write_settings (*file_obj*)

Write a settings file containing the data model and predicates to a file object

Parameters **file_obj** (BinaryIO) – file object to write settings data into

```
with open('learned_settings', 'wb') as f:
    matcher.write_settings(f)
```

cleanup_training ()

Clean up data we used for training. Free up memory.

join (*data_1*, *data_2*, *threshold=0.5*, *constraint='one-to-one'*)

Identifies pairs of records that refer to the same entity.

Returns pairs of record ids with a confidence score as a float between 0 and 1. The record_ids within the pair should refer to the same entity and the confidence score is the estimated probability that the records refer to the same entity.

This method should only used for small to moderately sized datasets for larger data, you need may need to generate your own pairs of records and feed them to the *score()*.

Parameters

- **data_1** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records from first dataset, where the keys are record_ids and the values are dictionaries with the keys being field names
- **data_2** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records from second dataset, same form as data_1
- **threshold** (float) – Number between 0 and 1 (default is .5). We will consider records as potential duplicates if the predicted probability of being a duplicate is above the threshold.

Lowering the number will increase recall, raising it will increase precision

- **constraint** (Literal['one-to-one', 'many-to-one', 'many-to-many']) – What type of constraint to put on a join.

  'one-to-one' Every record in data_1 can match at most one record from data_2 and every record from data_2 can match at most one record from data_1. This is good for when both data_1 and data_2 are from different sources and you are interested in matching across the sources. If, individually, data_1 or data_2 have many duplicates you will not get good matches.

  'many-to-one' Every record in data_1 can match at most one record from data_2, but more than one record from data_1 can match to the same record in data_2. This is good for when data_2 is a lookup table and data_1 is messy, such as geocoding or matching against golden records.

  'many-to-many' Every record in data_1 can match multiple records in data_2 and vice versa. This is like a SQL inner join.

```
> links = matcher.join(data_1, data_2, threshold=0.5)
> print(list(links))
[((1, 2), 0.790),
 ((4, 5), 0.720),
 ((10, 11), 0.899)]
```

3.1.4 StaticRecordLink Objects

class dedupe.**StaticRecordLink**(*settings_file*, *num_cores=None*, ***kwargs*)

Class for record linkage using saved settings. If you have already trained a RecordLink instance, you can load the saved settings with StaticRecordLink.

Parameters

- **settings_file** (BinaryIO) – A file object containing settings info produced from the write_settings() method.
- **num_cores** (Optional[int]) – the number of cpus to use for parallel processing, defaults to the number of cpus available on the machine. If set to 0, then multiprocessing will be disabled.

Warning: If using multiprocessing on Windows or Mac OS X, then you must protect calls to the Dedupe methods with a `if __name__ == '__main__'` in your main module, see https://docs.python.org/3/library/multiprocessing.html#the-spawn-and-forkserver-start-methods

```
with open('learned_settings', 'rb') as f:
    matcher = StaticRecordLink(f)
```

join(*data_1*, *data_2*, *threshold=0.5*, *constraint='one-to-one'*)

Identifies pairs of records that refer to the same entity.

Returns pairs of record ids with a confidence score as a float between 0 and 1. The record_ids within the pair should refer to the same entity and the confidence score is the estimated probability that the records refer to the same entity.

This method should only used for small to moderately sized datasets for larger data, you need may need to generate your own pairs of records and feed them to the *score()*.

Parameters

- **data_1** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records from first dataset, where the keys are record_ids and the values are dictionaries with the keys being field names
- **data_2** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records from second dataset, same form as data_1
- **threshold** (float) – Number between 0 and 1 (default is .5). We will consider records as potential duplicates if the predicted probability of being a duplicate is above the threshold.

  Lowering the number will increase recall, raising it will increase precision
- **constraint** (Literal['one-to-one', 'many-to-one', 'many-to-many']) – What type of constraint to put on a join.

  'one-to-one' Every record in data_1 can match at most one record from data_2 and every record from data_2 can match at most one record from data_1. This is good for when both data_1 and data_2 are from different sources and you are interested in matching across the sources. If, individually, data_1 or data_2 have many duplicates you will not get good matches.

  'many-to-one' Every record in data_1 can match at most one record from data_2, but more than one record from data_1 can match to the same record in data_2. This is good for when data_2 is a lookup table and data_1 is messy, such as geocoding or matching against golden records.

  'many-to-many' Every record in data_1 can match multiple records in data_2 and vice versa. This is like a SQL inner join.

```
> links = matcher.join(data_1, data_2, threshold=0.5)
> print(list(links))
[((1, 2), 0.790),
 ((4, 5), 0.720),
 ((10, 11), 0.899)]
```

3.1.5 `Gazetteer` Objects

class dedupe.**Gazetteer**(*variable_definition*, *num_cores=None*, ***kwargs*)

Class for active learning gazetteer matching.

Gazetteer matching is for matching a messy data set against a 'canonical dataset'. This class is useful for such tasks as matching messy addresses against a clean list

Parameters

- **variable_definition** (Sequence[Mapping]) – A list of dictionaries describing the variables will be used for training a model. See *Variable Definitions*
- **num_cores** (Optional[int]) – the number of cpus to use for parallel processing. If set to None, uses all cpus available on the machine. If set to 0, then multiprocessing will be disabled.

> **Warning:** If using multiprocessing on Windows or Mac OS X, then you must protect calls to the Dedupe methods with a `if __name__ == '__main__'` in your main module, see https://docs.python.org/3/library/multiprocessing.html#the-spawn-and-forkserver-start-methods

```python
# initialize from a defined set of fields
variables = [{'field' : 'Site name', 'type': 'String'},
             {'field' : 'Address', 'type': 'String'},
             {'field' : 'Zip', 'type': 'String', 'has missing':True},
             {'field' : 'Phone', 'type': 'String', 'has missing':True}
             ]

matcher = dedupe.Gazetteer(variables)
```

prepare_training(*data_1*, *data_2*, *training_file=None*, *sample_size=15000*, *blocked_proportion=0.5*, *original_length_1=None*, *original_length_2=None*)

Initialize the active learner with your data and, optionally, existing training data.

Parameters

- **data_1** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records from first dataset, where the keys are record_ids and the values are dictionaries with the keys being field names
- **data_2** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records from second dataset, same form as data_1
- **training_file** (Optional[TextIO]) – file object containing training data
- **sample_size** (int) – The size of the sample to draw. Defaults to 150,000
- **blocked_proportion** (float) – The proportion of record pairs to be sampled from similar records, as opposed to randomly selected pairs. Defaults to 0.5.
- **original_length_1** (Optional[int]) – If data_1 is a subsample of your first dataset, original_length_1 should be the size of the complete first dataset. By default, original_length_1 defaults to the length of data_1
- **original_length_2** (Optional[int]) – If data_2 is a subsample of your first dataset, original_length_2 should be the size of the complete first dataset. By default, original_length_2 defaults to the length of data_2

```python
matcher.prepare_training(data_1, data_2, 150000)

with open('training_file.json') as f:
    matcher.prepare_training(data_1, data_2, training_file=f)
```

uncertain_pairs()

Returns a list of pairs of records from the sample of record pairs tuples that Dedupe is most curious to have labeled.

This method is mainly useful for building a user interface for training a matching model.

```python
> pair = matcher.uncertain_pairs()
> print(pair)
[({'name' : 'Georgie Porgie'}, {'name' : 'Georgette Porgette'})]
```

mark_pairs(*labeled_pairs*)

Add users labeled pairs of records to training data and update the matching model

This method is useful for building a user interface for training a matching model or for adding training data from an existing source.

Parameters **labeled_pairs** (TrainingData) – A dictionary with two keys, match and distinct the values are lists that can contain pairs of records

```
labeled_examples = {'match'    : [],
                    'distinct' : [({'name' : 'Georgie Porgie'},
                                   {'name' : 'Georgette Porgette'})]
                    }
matcher.mark_pairs(labeled_examples)
```

train (*recall=0.95*, *index_predicates=True*)

Learn final pairwise classifier and fingerprinting rules. Requires that adequate training data has been already been provided.

Parameters

- **recall** (float) – The proportion of true dupe pairs in our training data that that the learned fingerprinting rules must cover. If we lower the recall, there will be pairs of true dupes that we will never directly compare.

  recall should be a float between 0.0 and 1.0.

- **index_predicates** (bool) – Should dedupe consider predicates that rely upon indexing the data. Index predicates can be slower and take substantial memory.

write_training (*file_obj*)

Write a JSON file that contains labeled examples

Parameters **file_obj** (TextIO) – file object to write training data to

```
with open('training.json', 'w') as f:
    matcher.write_training(f)
```

write_settings (*file_obj*)

Write a settings file containing the data model and predicates to a file object

Parameters **file_obj** (BinaryIO) – file object to write settings data into

```
with open('learned_settings', 'wb') as f:
    matcher.write_settings(f)
```

cleanup_training ()

Clean up data we used for training. Free up memory.

index (*data*)

Add records to the index of records to match against. If a record in canonical_data has the same key as a previously indexed record, the old record will be replaced.

Parameters **data** (Mapping[Union[int, str], Mapping[str, Any]]) – a dictionary of records where the keys are record_ids and the values are dictionaries with the keys being field_names

unindex (*data*)

Remove records from the index of records to match against.

Parameters **data** (Mapping[Union[int, str], Mapping[str, Any]]) – a dictionary of records where the keys are record_ids and the values are dictionaries with the keys being field_names

search (*data*, *threshold=0.0*, *n_matches=1*, *generator=False*)

Identifies pairs of records that could refer to the same entity, returns tuples containing tuples of possible matches, with a confidence score for each match. The record_ids within each tuple should refer to potential matches from a messy data record to canonical records. The confidence score is the estimated probability that the records refer to the same entity.

Parameters

- **`data`** (`Mapping[Union[int, str], Mapping[str, Any]]`) – a dictionary of records from a messy dataset, where the keys are record_ids and the values are dictionaries with the keys being field names.
- **`threshold`** (`float`) – a number between 0 and 1 (default is 0.5). We will consider records as potential duplicates if the predicted probability of being a duplicate is above the threshold.

  Lowering the number will increase recall, raising it will increase precision
- **`n_matches`** (`int`) – the maximum number of possible matches from canonical_data to return for each record in data. If set to `None` all possible matches above the threshold will be returned. Defaults to 1
- **`generator`** (`bool`) – when `True`, match will generate a sequence of possible matches, instead of a list. Defaults to `False` This makes `match` a lazy method.

```
> matches = gazetteer.search(data, threshold=0.5, n_matches=2)
> print(matches)
[(((1, 6), 0.72),
  ((1, 8), 0.6)),
 (((2, 7), 0.72),),
 (((3, 6), 0.72),
  ((3, 8), 0.65)),
 (((4, 6), 0.96),
  ((4, 5), 0.63))]
```

3.1.6 `StaticGazetteer` Objects

class `dedupe.`**`StaticGazetteer`**(*settings_file*, *num_cores=None*, ***kwargs*)

Class for gazetter matching using saved settings.

If you have already trained a *`Gazetteer`* instance, you can load the saved settings with StaticGazetteer.

Parameters

- **`settings_file`** (`BinaryIO`) – A file object containing settings info produced from the `write_settings()` method.
- **`num_cores`** (`Optional[int]`) – the number of cpus to use for parallel processing, defaults to the number of cpus available on the machine. If set to 0, then multiprocessing will be disabled.

> **Warning:** If using multiprocessing on Windows or Mac OS X, then you must protect calls to the Dedupe methods with a `if __name__ == '__main__'` in your main module, see https://docs.python.org/3/library/multiprocessing.html#the-spawn-and-forkserver-start-methods

```
with open('learned_settings', 'rb') as f:
    matcher = StaticGazetteer(f)
```

`index`(*data*)

Add records to the index of records to match against. If a record in `canonical_data` has the same key as a previously indexed record, the old record will be replaced.

Parameters **data** (Mapping[Union[int, str], Mapping[str, Any]]) – a dictionary of records where the keys are record_ids and the values are dictionaries with the keys being field_names

unindex (*data*)

Remove records from the index of records to match against.

Parameters **data** (Mapping[Union[int, str], Mapping[str, Any]]) – a dictionary of records where the keys are record_ids and the values are dictionaries with the keys being field_names

search (*data*, *threshold=0.0*, *n_matches=1*, *generator=False*)

Identifies pairs of records that could refer to the same entity, returns tuples containing tuples of possible matches, with a confidence score for each match. The record_ids within each tuple should refer to potential matches from a messy data record to canonical records. The confidence score is the estimated probability that the records refer to the same entity.

Parameters

- **data** (Mapping[Union[int, str], Mapping[str, Any]]) – a dictionary of records from a messy dataset, where the keys are record_ids and the values are dictionaries with the keys being field names.
- **threshold** (float) – a number between 0 and 1 (default is 0.5). We will consider records as potential duplicates if the predicted probability of being a duplicate is above the threshold.

  Lowering the number will increase recall, raising it will increase precision
- **n_matches** (int) – the maximum number of possible matches from canonical_data to return for each record in data. If set to None all possible matches above the threshold will be returned. Defaults to 1
- **generator** (bool) – when True, match will generate a sequence of possible matches, instead of a list. Defaults to False This makes match a lazy method.

```
> matches = gazetteer.search(data, threshold=0.5, n_matches=2)
> print(matches)
[(((1, 6), 0.72),
  ((1, 8), 0.6)),
 (((2, 7), 0.72),),
 (((3, 6), 0.72),
  ((3, 8), 0.65)),
 (((4, 6), 0.96),
  ((4, 5), 0.63))]
```

blocks (*data*)

Yield groups of pairs of records that share fingerprints.

Each group contains one record from data_1 paired with the records from the indexed records that data_1 shares a fingerprint with.

Each pair within and among blocks will occur at most once. If you override this method, you need to take care to ensure that this remains true, as downstream methods, particularly *many_to_n()*, assumes that every pair of records is compared no more than once.

Parameters **data** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records, where the keys are record_ids and the values are dictionaries with the keys being field names

```
> pairs = matcher.pairs(data)
> print(list(pairs))
[[((1, {'name' : 'Pat', 'address' : '123 Main'}),
   (8, {'name' : 'Pat', 'address' : '123 Main'})),
  ((1, {'name' : 'Pat', 'address' : '123 Main'}),
   (9, {'name' : 'Sam', 'address' : '123 Main'}))
  ],
 [((2, {'name' : 'Sam', 'address' : '2600 State'}),
   (5, {'name' : 'Pam', 'address' : '2600 Stat'})),
  ((2, {'name' : 'Sam', 'address' : '123 State'}),
   (7, {'name' : 'Sammy', 'address' : '123 Main'}))
 ]]
```

`score`(*blocks*)

Scores groups of pairs of records. Yields structured numpy arrays representing pairs of records in the group and the associated probability that the pair is a match.

Parameters **`blocks`** (`Iterator[List[Tuple[Tuple[Union[int, str], Mapping[str, Any]], Tuple[Union[int, str], Mapping[str, Any]]]]]`) – Iterator of blocks of records

`many_to_n`(*score_blocks*, *threshold=0.0*, *n_matches=1*)

For each group of scored pairs, yield the highest scoring N pairs

Parameters

- **`score_blocks`** (`Iterable[ndarray]`) – Iterator of numpy structured arrays, each with a dtype of `[('pairs', id_type, 2), ('score', 'f4')]` where dtype is either a str or int, and score is a number between 0 and 1. The 'pairs' column contains pairs of ids of the records compared and the 'score' column should contains the similarity score for that pair of records.
- **`threshold`** (`float`) – Number between 0 and 1 (default is 0.0). We will consider records as potential duplicates if the predicted probability of being a duplicate is above the threshold.

  Lowering the number will increase recall, raising it will increase precision
- **`n_matches`** (`int`) – How many top scoring pairs to select per group

3.1.7 Lower Level Classes and Methods

With the methods documented above, you can work with data into the millions of records. However, if are working with larger data you may not be able to load all your data into memory. You'll need to interact with some of the lower level classes and methods.

See also:

The PostgreSQL and MySQL examples use these lower level classes and methods.

Dedupe and StaticDedupe

class `dedupe.`**`Dedupe`**

`fingerprinter`

Instance of *`dedupe.blocking.Fingerprinter`* class if the *`train()`* has been run, else `None`.

`pairs`(*data*)

Yield pairs of records that share common fingerprints.

Each pair will occur at most once. If you override this method, you need to take care to ensure that this remains true, as downstream methods, particularly *`cluster()`*, assumes that every pair of records is compared no more than once.

Parameters **`data`** – Dictionary of records, where the keys are record_ids and the values are dictionaries with the keys being field names

```
> pairs = matcher.pairs(data)
> print(list(pairs))
[((1, {'name' : 'Pat', 'address' : '123 Main'}),
  (2, {'name' : 'Pat', 'address' : '123 Main'})),
 ((1, {'name' : 'Pat', 'address' : '123 Main'}),
  (3, {'name' : 'Sam', 'address' : '123 Main'}))
 ]
```

`score`(*pairs*)

Scores pairs of records. Returns pairs of tuples of records id and associated probabilites that the pair of records are match

Parameters **`pairs`** (`Iterator[Tuple[Tuple[Union[int, str], Mapping[str, Any]], Tuple[Union[int, str], Mapping[str, Any]]]]`) – Iterator of pairs of records

`cluster`(*scores*, *threshold=0.5*)

From the similarity scores of pairs of records, decide which groups of records are all referring to the same entity.

Yields tuples containing a sequence of record ids and corresponding sequence of confidence score as a float between 0 and 1. The record_ids within each set should refer to the same entity and the confidence score is a measure of our confidence a particular entity belongs in the cluster.

Parameters

- **`scores`** (`ndarray`) – a numpy structured array with a dtype of `[('pairs', id_type, 2), ('score', 'f4')]` where dtype is either a str or int, and score is a number between 0 and 1. The 'pairs' column contains pairs of ids of the records compared and the 'score' column should contains the similarity score for that pair of records.

  For each pair, the smaller id should be first.

- **`threshold`** (`float`) – Number between 0 and 1. We will only consider put together records into clusters if the cophenetic similarity of the cluster is greater than the threshold.

  Lowering the number will increase recall, raising it will increase precision

  Defaults to 0.5.

```
> pairs = matcher.pairs(data)
> scores = matcher.scores(pairs)
> clusters = matcher.cluster(scores)
> print(list(clusters))
```

(continues on next page)

(continued from previous page)

```
[((1, 2, 3), (0.790, 0.860, 0.790)),
 ((4, 5), (0.720, 0.720)),
 ((10, 11), (0.899, 0.899))]
```

class dedupe.**StaticDedupe**

fingerprinter
Instance of *dedupe.blocking.Fingerprinter* class

pairs (*data*)
Same as *dedupe.Dedupe.pairs()*

score (*pairs*)
Same as *dedupe.Dedupe.score()*

cluster (*matches*, *threshold=0.5*)
Same as *dedupe.Dedupe.cluster()*

RecordLink and StaticRecordLink

class dedupe.**RecordLink**

fingerprinter
Instance of *dedupe.blocking.Fingerprinter* class if the *train()* has been run, else None.

pairs (*data_1*, *data_2*)
Yield pairs of records that share common fingerprints.

Each pair will occur at most once. If you override this method, you need to take care to ensure that this remains true, as downstream methods, particularly *one_to_one()*, and *many_to_one()* assumes that every pair of records is compared no more than once.

Parameters

- **data_1** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records from first dataset, where the keys are record_ids and the values are dictionaries with the keys being field names
- **data_2** (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records from second dataset, same form as data_1

```
> pairs = matcher.pairs(data_1, data_2)
> print(list(pairs))
[((1, {'name' : 'Pat', 'address' : '123 Main'}),
  (2, {'name' : 'Pat', 'address' : '123 Main'})),
 ((1, {'name' : 'Pat', 'address' : '123 Main'}),
  (3, {'name' : 'Sam', 'address' : '123 Main'}))
 ]
```

score (*pairs*)
Scores pairs of records. Returns pairs of tuples of records id and associated probabilites that the pair of records are match

Parameters **pairs** (Iterator[Tuple[Tuple[Union[int, str], Mapping[str, Any]], Tuple[Union[int, str], Mapping[str, Any]]]]) – Iterator of pairs of records

one_to_one(*scores*, *threshold=0.0*)

From the similarity scores of pairs of records, decide which pairs refer to the same entity.

Every record in data_1 can match at most one record from data_2 and every record from data_2 can match at most one record from data_1. See https://en.wikipedia.org/wiki/Injective_function.

This method is good for when both data_1 and data_2 are from different sources and you are interested in matching across the sources. If, individually, data_1 or data_2 have many duplicates you will not get good matches.

Yields pairs of record ids with a confidence score as a float between 0 and 1. The record_ids within the pair should refer to the same entity and the confidence score is the estimated probability that the records refer to the same entity.

Parameters

- **scores** (ndarray) – a numpy structured array with a dtype of `[('pairs', id_type, 2), ('score', 'f4')]` where dtype is either a str or int, and score is a number between 0 and 1. The 'pairs' column contains pairs of ids of the records compared and the 'score' column should contains the similarity score for that pair of records.
- **threshold** (float) – Number between 0 and 1 (default is 0.0). We will consider records as potential duplicates if the predicted probability of being a duplicate is above the threshold.

  Lowering the number will increase recall, raising it will increase precision

```
> pairs = matcher.pairs(data)
> scores = matcher.scores(pairs, threshold=0.5)
> links = matcher.one_to_one(scores)
> print(list(links))
[((1, 2), 0.790),
 ((4, 5), 0.720),
 ((10, 11), 0.899)]
```

many_to_one(*scores*, *threshold=0.0*)

From the similarity scores of pairs of records, decide which pairs refer to the same entity.

Every record in data_1 can match at most one record from data_2, but more than one record from data_1 can match to the same record in data_2. See https://en.wikipedia.org/wiki/Surjective_function

This method is good for when data_2 is a lookup table and data_1 is messy, such as geocoding or matching against golden records.

Yields pairs of record ids with a confidence score as a float between 0 and 1. The record_ids within the pair should refer to the same entity and the confidence score is the estimated probability that the records refer to the same entity.

Parameters

- **scores** (ndarray) – a numpy structured array with a dtype of `[('pairs', id_type, 2), ('score', 'f4')]` where dtype is either a str or int, and score is a number between 0 and 1. The 'pairs' column contains pairs of ids of the records compared and the 'score' column should contains the similarity score for that pair of records.
- **threshold** (float) – Number between 0 and 1 (default is 0.0). We will consider records as potential duplicates if the predicted probability of being a duplicate is above the threshold.

  Lowering the number will increase recall, raising it will increase precision

```
> pairs = matcher.pairs(data)
> scores = matcher.scores(pairs, threshold=0.5)
> links = matcher.many_to_one(scores)
> print(list(links))
[((1, 2), 0.790),
 ((4, 5), 0.720),
 ((7, 2), 0.623),
 ((10, 11), 0.899)]
```

class dedupe.**StaticRecordLink**

fingerprinter
Instance of *dedupe.blocking.Fingerprinter* class

pairs(*data_1*, *data_2*)
Same as *dedupe.RecordLink.pairs()*

score(*pairs*)
Same as *dedupe.RecordLink.score()*

one_to_one(*scores*, *threshold=0.0*)
Same as *dedupe.RecordLink.one_to_one()*

many_to_one(*scores*, *threshold=0.0*)
Same as *dedupe.RecordLink.many_to_one()*

Gazetteer and StaticGazetteer

class dedupe.**Gazetteer**

fingerprinter
Instance of *dedupe.blocking.Fingerprinter* class if the *train()* has been run, else None.

blocks(*data*)
Yield groups of pairs of records that share fingerprints.

Each group contains one record from data_1 paired with the records from the indexed records that data_1 shares a fingerprint with.

Each pair within and among blocks will occur at most once. If you override this method, you need to take care to ensure that this remains true, as downstream methods, particularly *many_to_n()*, assumes that every pair of records is compared no more than once.

Parameters data (Mapping[Union[int, str], Mapping[str, Any]]) – Dictionary of records, where the keys are record_ids and the values are dictionaries with the keys being field names

```
> pairs = matcher.pairs(data)
> print(list(pairs))
[[((1, {'name' : 'Pat', 'address' : '123 Main'}),
   (8, {'name' : 'Pat', 'address' : '123 Main'})),
  ((1, {'name' : 'Pat', 'address' : '123 Main'}),
   (9, {'name' : 'Sam', 'address' : '123 Main'}))
  ],
 [((2, {'name' : 'Sam', 'address' : '2600 State'}),
   (5, {'name' : 'Pam', 'address' : '2600 Stat'})),
  ((2, {'name' : 'Sam', 'address' : '123 State'}),
```

(continues on next page)

(continued from previous page)

```
  (7, {'name' : 'Sammy', 'address' : '123 Main'}))
]]
```

score (*blocks*)

Scores groups of pairs of records. Yields structured numpy arrays representing pairs of records in the group and the associated probability that the pair is a match.

Parameters **blocks** (`Iterator[List[Tuple[Tuple[Union[int, str], Mapping[str, Any]], Tuple[Union[int, str], Mapping[str, Any]]]]]`) – Iterator of blocks of records

many_to_n (*score_blocks*, *threshold=0.0*, *n_matches=1*)

For each group of scored pairs, yield the highest scoring N pairs

Parameters

- **score_blocks** (`Iterable[ndarray]`) – Iterator of numpy structured arrays, each with a dtype of `[('pairs', id_type, 2), ('score', 'f4')]` where dtype is either a str or int, and score is a number between 0 and 1. The 'pairs' column contains pairs of ids of the records compared and the 'score' column should contains the similarity score for that pair of records.
- **threshold** (`float`) – Number between 0 and 1 (default is 0.0). We will consider records as potential duplicates if the predicted probability of being a duplicate is above the threshold.

  Lowering the number will increase recall, raising it will increase precision
- **n_matches** (`int`) – How many top scoring pairs to select per group

class dedupe.**StaticGazeteer**

fingerprinter

Instance of *`dedupe.blocking.Fingerprinter`* class

blocks (*data*)

Same as *`dedupe.Gazetteer.blocks()`*

score (*blocks*)

Same as *`dedupe.Gazetteer.score()`*

many_to_n (*score_blocks*, *threshold=0.0*, *n_matches=1*)

Same as *`dedupe.Gazetteer.many_to_n()`*

`Fingerprinter` Objects

class dedupe.blocking.**Fingerprinter** (*predicates*)

Takes in a record and returns all blocks that record belongs to

__call__ (*records*, *target=False*)

Generate the predicates for records. Yields tuples of (predicate, record_id).

Parameters

- **records** (`Iterable[Tuple[Union[int, str], Mapping[str, Any]]]`) – A sequence of tuples of (record_id, record_dict). Can often be created by `data_dict.items()`.

- **target** (bool) – Indicates whether the data should be treated as the target data. This effects the behavior of search predicates. If target is set to True, an search predicate will return the value itself. If target is set to False the search predicate will return all possible values within the specified search distance.

  Let's say we have a LevenshteinSearchPredicate with an associated distance of 1 on a "name" field; and we have a record like {"name": "thomas"}. If the target is set to True then the predicate will return "thomas". If target is set to False, then the blocker could return "thomas", "tomas", and "thoms". By using the target argument on one of your datasets, you will dramatically reduce the total number of comparisons without a loss of accuracy.

```
> data = [(1, {'name' : 'bob'}), (2, {'name' : 'suzanne'})]
> blocked_ids = deduper.fingerprinter(data)
> print list(blocked_ids)
[('foo:1', 1), ..., ('bar:1', 100)]
```

index_fields

A dictionary of all the fingerprinter methods that use an index of data field values. The keys are the field names, which can be useful to know for indexing the data.

index(*docs*, *field*)

Add docs to the indices used by fingerprinters.

Some fingerprinter methods depend upon having an index of values that a field may have in the data. This method adds those values to the index. If you don't have any fingerprinter methods that use an index, this method will do nothing.

Parameters

- **docs** (Union[Iterable[str], Iterable[Iterable[str]]]) – an iterator of values from your data to index. While not required, it is recommended that docs be a unique set of of those values. Indexing can be an expensive operation.
- **field** (str) – fieldname or key associated with the values you are indexing

unindex(*docs*, *field*)

Remove docs from indices used by fingerprinters

Parameters

- **docs** (Union[Iterable[str], Iterable[Iterable[str]]]) – an iterator of values from your data to remove. While not required, it is recommended that docs be a unique set of of those values. Indexing can be an expensive operation.
- **field** (str) – fieldname or key associated with the values you are unindexing

reset_indices()

Fingeprinter indicdes can take up a lot of memory. If you are done with blocking, the method will reset the indices to free up. If you need to block again, the data will need to be re-indexed.

3.1.8 Convenience Functions

`dedupe.`**`console_label`**(*deduper*)

Train a matcher instance (Dedupe, RecordLink, or Gazetteer) from the command line. Example

```
> deduper = dedupe.Dedupe(variables)
> deduper.prepare_training(data)
> dedupe.console_label(deduper)
```

`dedupe.`**`training_data_dedupe`**(*data*, *common_key*, *training_size=50000*)

Construct training data for consumption by the func:`mark_pairs` method from an already deduplicated dataset.

Parameters

- **`data`** (`Mapping[Union[int, str], Mapping[str, Any]]`) – Dictionary of records where the keys are record_ids and the values are dictionaries with the keys being field names
- **`common_key`** (`str`) – The name of the record field that uniquely identifies a match
- **`training_size`** (`int`) – the rough limit of the number of training examples, defaults to 50000

Note: Every match must be identified by the sharing of a common key. This function assumes that if two records do not share a common key then they are distinct records.

`dedupe.`**`training_data_link`**(*data_1*, *data_2*, *common_key*, *training_size=50000*)

Construct training data for consumption by the func:`mark_pairs` method from already linked datasets.

Parameters

- **`data_1`** (`Mapping[Union[int, str], Mapping[str, Any]]`) – Dictionary of records from first dataset, where the keys are record_ids and the values are dictionaries with the keys being field names
- **`data_2`** (`Mapping[Union[int, str], Mapping[str, Any]]`) – Dictionary of records from second dataset, same form as data_1
- **`common_key`** (`str`) – The name of the record field that uniquely identifies a match
- **`training_size`** (`int`) – the rough limit of the number of training examples, defaults to 50000

Note: Every match must be identified by the sharing of a common key. This function assumes that if two records do not share a common key then they are distinct records.

`dedupe.`**`canonicalize`**(*record_cluster*)

Constructs a canonical representation of a duplicate cluster by finding canonical values for each field

Parameters **`record_cluster`** (`List[Mapping[str, Any]]`) – A list of records within a duplicate cluster, where the records are dictionaries with field names as keys and field values as values

3.2 Variable Definitions

3.2.1 Variable Types

A variable definition describes the records that you want to match. It is a dictionary where the keys are the fields and the values are the field specification. For example:-

```
[
    {'field': 'Site name', 'type': 'String'},
    {'field': 'Address', 'type': 'String'},
    {'field': 'Zip', 'type': 'String', 'has missing': True},
    {'field': 'Phone', 'type': 'String', 'has missing': True}
]
```

String Types

A `String` type field must declare the name of the record field to compare a `String` type declaration. The `String` type expects fields to be of class string.

`String` types are compared using affine gap string distance.

For example:-

```
{'field': 'Address', type: 'String'}
```

ShortString Types

A `ShortString` type field is just like `String` types except that dedupe will not try to learn a canopy blocking rule for these fields, which can speed up the training phase considerably.

Zip codes and city names are good candidates for this type. If in doubt, always use `String`.

For example:-

```
{'field': 'Zipcode', type: 'ShortString'}
```

Text Types

If you want to compare fields containing long blocks of text e.g. product descriptions or article abstracts, you should use this type. `Text` type fields are compared using the cosine similarity metric.

This is a measurement of the amount of words that two documents have in common. This measure can be made more useful as the overlap of rare words counts more than the overlap of common words.

If provided a sequence of example fields (i.e. a corpus) then dedupe will learn these weights for you. For example:-

```
{
 'field': 'Product description',
 'type': 'Text',
 'corpus' : [
         'this product is great',
         'this product is great and blue'
     ]
}
```

If you don’t want to adjust the measure to your data, just leave ‘corpus’ out of the variable definition entirely.

```
{'field': 'Product description', 'type': 'Text'}
```

Custom Types

A `Custom` type field must have specify the field it wants to compare, a type declaration of `Custom`, and a comparator declaration. The comparator must be a function that can take in two field values and return a number.

For example, a custom comparator:

```
def sameOrNotComparator(field_1, field_2) :
  if field_1 and field_2 :
      if field_1 == field_2 :
          return 0
      else:
          return 1
```

The corresponding variable definition:

```
{
    'field': 'Zip',
    'type': 'Custom',
    'comparator': sameOrNotComparator
 }
```

LatLong

A `LatLong` type field must have as the name of a field and a type declaration of `LatLong`. `LatLong` fields are compared using the Haversine Formula.

A `LatLong` type field must consist of tuples of floats corresponding to a latitude and a longitude.

```
{'field': 'Location', 'type': 'LatLong'}
```

Set

A `Set` type field is for comparing lists of elements, like keywords or client names. `Set` types are very similar to *Text Types*. They use the same comparison function and you can also let dedupe learn which terms are common or rare by providing a corpus. Within a record, a `Set` type field has to be hashable sequences like tuples or frozensets.

```
{
    'field': 'Co-authors',
    'type': 'Set',
    'corpus' : [
            ('steve edwards'),
            ('steve edwards', 'steve jobs')
        ]
 }
```

or

```
{'field': 'Co-authors', 'type': 'Set'}
```

Interaction

An `Interaction` field multiplies the values of the multiple variables. An `Interaction` variable is created with type declaration of `Interaction` and an `interaction variables` declaration.

The `interaction variables` field must be a sequence of variable names of other fields you have defined in your variable definition.

Interactions are good when the effect of two predictors is not simply additive.

```
[
    { 'field': 'Name', 'variable name': 'name', 'type': 'String' },
    { 'field': 'Zip', 'variable name': 'zip', 'type': 'Custom',
  'comparator' : sameOrNotComparator },
    {'type': 'Interaction', 'interaction variables': ['name', 'zip']}
]
```

Exact

`Exact` variables measure whether two fields are exactly the same or not.

```
{'field': 'city', 'type': 'Exact'}
```

Exists

`Exists` variables measure whether both, one, or neither of the fields are defined. This can be useful if the presence or absence of a field tells you something meaningful about the record.

```
{'field': 'first_name', 'type': 'Exists'}
```

Categorical

`Categorical` variables are useful when you are dealing with qualitatively different types of things. For example, you may have data on businesses and you find that taxi cab businesses tend to have very similar names but law firms don't. `Categorical` variables would let you indicate whether two records are both taxi companies, both law firms, or one of each.

Dedupe would represent these three possibilities using two dummy variables:

```
taxi-taxi      0 0
lawyer-lawyer  1 0
taxi-lawyer    0 1
```

A categorical field declaration must include a list of all the different strings that you want to treat as different categories.

So if you data looks like this:-

```
'Name'           'Business Type'
AAA Taxi         taxi
AA1 Taxi         taxi
Hindelbert Esq   lawyer
```

You would create a definition such as:

```
{
    'field': 'Business Type',
    'type': 'Categorical',
    'categories' : ['taxi', 'lawyer']
}
```

Price

`Price` variables are useful for comparing positive, non-zero numbers like prices. The values of `Price` field must be a positive float. If the value is 0 or negative, then an exception will be raised.

```
{'field': 'cost', 'type': 'Price'}
```

DateTime

`DateTime` variables are useful for comparing dates and timestamps. This variable can accept strings or Python datetime objects as inputs.

The `DateTime` variable definition accepts a few optional arguments that can help improve behavior if you know your field follows an unusual format:

- `fuzzy` - Use fuzzy parsing to automatically extract dates from strings like "It happened on June 2nd, 2018" (default `True`)
- `dayfirst` - Ambiguous dates should be parsed as dd/mm/yy (default `False`)
- `yearfirst`- Ambiguous dates should be parsed as yy/mm/dd (default `False`)

Note that the `DateTime` variable defaults to mm/dd/yy for ambiguous dates. If both `dayfirst` and `yearfirst` are set to `True`, then `dayfirst` will take precedence.

For example, a sample `DateTime` variable definition, using the defaults:

```
{
    'field': 'time_of_sale',
    'type': 'DateTime',
    'fuzzy': True,
    'dayfirst': False,
    'yearfirst': False
}
```

If you're happy with the defaults, you can simply define the `field` and `type`:

```
{'field': 'time_of_sale', 'type': 'DateTime'}
```

3.2.2 Optional Variables

Address Type

An `Address` variable should be used for United States addresses. It uses the usaddress package to split apart an address string into components like address number, street name, and street type and compares component to component.

For example:-

```
{'field': 'address', 'type': 'Address'}
```

Install the dedupe-variable-address package for `Address` Type.

Name Type

A `Name` variable should be used for a field that contains American names, corporations and households. It uses the probablepeople package to split apart an name string into components like give name, surname, generational suffix, for people names, and abbreviation, company type, and legal form for corporations.

For example:-

```
{'field': 'name', 'type': 'Name'}
```

Install the dedupe-variable-name package for `Name` Type.

Fuzzy Category

A `FuzzyCategorical` variable should be used for when you for categorical data that has variations.

Occupations are an example, where the you may have 'Attorney', 'Counsel', and 'Lawyer'. For this variable type, you need to supply a corpus of records that contain your focal record and other field types. This corpus should either be all the data you are trying to link or a representative sample.

For example:-

```
{
 'field': 'occupation',
 'type': 'FuzzyCategorical',
 'corpus' : [
        {'name' : 'Jim Doe', 'occupation' : 'Attorney'},
        {'name' : 'Jim Doe', 'occupation' : 'Lawyer'}
    ]
}
```

Install the dedupe-variable-fuzzycategory package for the `FuzzyCategorical` Type.

3.2.3 Missing Data

If the value of field is missing, that missing value should be represented as a `None` object.

```
[
     {'Name': 'AA Taxi', 'Phone': '773.555.1124'},
     {'Name': 'AA Taxi', 'Phone': None},
     {'Name': None, 'Phone': '773-555-1123'}
]
```

If you want to model this missing data for a field, you can set `'has missing' : True` in the variable definition. This creates a new, additional field representing whether the data was present or not and zeros out the missing data.

If there is missing data, but you did not declare `'has missing' : True` then the missing data will simply be zeroed out and no field will be created to account for missing data.

This approach is called 'response augmented data' and is described in Benjamin Marlin's thesis "Missing Data Problems in Machine Learning". Basically, this approach says that, even without looking at the value of the field comparisons, the pattern of observed and missing responses will affect the probability that a pair of records are a match.

This approach makes a few assumptions that are usually not completely true:

- Whether a field is missing data is not associated with any other field missing data.
- That the weighting of the observed differences in field A should be the same regardless of whether field B is missing.

If you define an an interaction with a field that you declared to have missing data, then `has missing : True` will also be set for the Interaction field.

Longer example of a variable definition:

```
[
    {'field': 'name', 'variable name' : 'name', 'type': 'String'},
    {'field': 'address', 'type': 'String'},
    {'field': 'city', 'variable name' : 'city', 'type': 'String'},
    {'field': 'zip', 'type': 'Custom', 'comparator' : sameOrNotComparator},
    {'field': 'cuisine', 'type': 'String', 'has missing': True}
    {'type': 'Interaction', 'interaction variables' : ['name', 'city']}
]
```

3.2.4 Multiple Variables comparing same field

It is possible to define multiple variables that all compare the same variable.

For example:-

```
[
    {'field': 'name', 'type': 'String'},
    {'field': 'name', 'type': 'Text'}
]
```

Will create two variables that both compare the 'name' field but in different ways.

3.2.5 Optional Edit Distance

For `String`, `ShortString`, `Address`, and `Name` fields, you can choose to use the a conditional random field distance measure for strings. This measure can give you more accurate results but is much slower than the default edit distance.

```
{'field': 'name', 'type': 'String', 'crf': True}
```

3.3 How it works

3.3.1 Matching Records

If you look at the following two records, you might think it's pretty clear that they are about the same person.

```
first name | last name | address                    | phone    |
---------------------------------------------------------------
bob        | roberts   | 1600 pennsylvania ave.     | 555-0123 |
Robert     | Roberts   | 1600 Pensylvannia Avenue |          |
```

However, I bet it would be pretty hard for you to explicitly write down all the reasons why you think these records are about the same Mr. Roberts.

Record similarity

One way that people have approached this problem is by saying that records that are more similar are more likely to be duplicates. That's a good first step, but then we have to precisely define what we mean for two records to be similar.

The default way that we do this in Dedupe is to use what's called a string metric. A string metric is an way of taking two strings and returning a number that is low if the strings are similar and high if they are dissimilar. One famous string metric is called the Hamming distance. It counts the number of substitutions that must be made to turn one string into another. For example, `roberts` and `Roberts` would have Hamming distance of 1 because we have to substitute `r` for `R` in order to turn `roberts` into `Roberts`.

There are lots of different string metrics, and we actually use a metric called the Affine Gap Distance, which is a variation on the Hamming distance.

Record by record or field by field

When we are calculating whether two records are similar we could treat each record as if it was a long string.

```
record_distance = string_distance('bob roberts 1600 pennsylvania ave. 555-0123',
                                  'Robert Roberts 1600 Pensylvannia Avenue')
```

Alternately, we could compare field by field

```
record_distance = (string_distance('bob', 'Robert')
                   + string_distance('roberts', 'Roberts')
                   + string_distance('1600 pennsylvania ave.', '1600 Pensylvannia␣
↪Avenue')
                   + string_distance('555-0123', ''))
```

The major advantage of comparing field by field is that we don't have to treat each field string distance equally. Maybe we think that its really important that the last names and addresses are similar but it's not as important that first name and phone numbers are close. We can express that importance with numeric weights, i.e.

```
record_distance = (0.5 * string_distance('bob', 'Robert')
                   + 2.0 * string_distance('roberts', 'Roberts')
                   + 2.0 * string_distance('1600 pennsylvania ave.', '1600␣
↪Pensylvannia Avenue')
                   + 0.5 * string_distance('555-0123', ''))
```

Setting weights and making decisions

Say we set our record_distance to be this weighted sum of field distances, just as we had above. Let's say we calculated the record_distance and we found that it was the beautiful number **8**.

That number, by itself, is not that helpful. Ultimately, we are trying to decide whether a pair of records are duplicates, and I'm not sure what decision I should make if I see an 8. Does an 8 mean that the pair of records are really similar or really far apart, likely or unlikely to be duplicates. We'd like to define the record distances so that we can look at the number and know whether to decide whether it's a duplicate.

Also, I really would rather not have to set the weights by hand every time. It can be very tricky to know which fields are going to matter and even if I know that some fields are more important I'm not sure how to quantify it (is it 2 times more important or 1.3 times)?

Fortunately, we can solve both problems with a technique called regularized logistic regression. If we supply pairs of records that we label as either being duplicates or distinct, then Dedupe will learn a set of weights such that the record distance can easily be transformed into our best estimate of the probability that a pair of records are duplicates.

Once we have learned these good weights, we want to use them to find which records are duplicates. But turns out that doing this the naive way will usually not work, and *we'll have to do something smarter*.

Active learning

In order to learn those weights, Dedupe needs example pairs with labels. Most of the time, we will need people to supply those labels.

But the whole point of Dedupe is to save people's time, and that includes making good use of your labeling time so we use an approach called Active Learning.

Basically, Dedupe keeps track of bunch unlabeled pairs and whether

1. the current learning blocking rules would cover the pairs
2. the current learned classifier would predict that the pairs are duplicates or are distinct

We maintain a set of the pairs where there is disagreement: that is pairs which classifier believes are duplicates but which are not covered by the current blocking rules, and the pairs which the classifier believes are distinct but which are blocked together.

Dedupe picks, at random from this disagreement set, a pair of records and asks the user to decide. Once it gets this label, it relearns the weights and blocking rules. We then recalculate the disagreement set.

Other field distances

We have implemented a number of field distance measures. See *the details about variables*.

3.3.2 Making Smart Comparisons

Say we have magic function that takes in a pair of records and always returns a `False` if a pair of records are distinct and `True` if a pair of records refer to the same person or organization.

Let's say that this function was pretty slow. It always took one second to return.

How long would it take to duplicate a thousand records?

Within a dataset of thousand records, there are $\frac{1{,}000 \times 999}{2} = 499{,}500$ unique pairs of records. If we compared all of them using our magic function it would take six days.

But, one second is a **long** time, let's say we sped it up so that we can make 10,000 comparisons per second. Now we can get through our thousand-record-long dataset in less than a minute.

Feeling good about our super-fast comparison function, let's take on a dataset of 100,000 records. Now there are $\frac{100{,}000 \times 99{,}999}{2} = 4{,}999{,}950{,}000$ unique possible pairs. If we compare all of them with our super-fast comparison function, it will take six days again.

If we want to work with moderately sized data, we have to find a way of making fewer comparisons.

Duplicates are rare

In real world data, nearly all possible pairs of records are not duplicates.

In this four-record example below, only two pairs of records are duplicates–(1, 2) and (3, 4), while there are four unique pairs of records that are not duplicates–(1,3), (1,4), (2,3), and (2,4). Typically, as the size of the dataset grows, the fraction of pairs of records that are duplicates gets very small very quickly.

first name	last name	address	phone	record_id
bob	roberts	1600 pennsylvania ave.	555-0123	1
Robert	Roberts	1600 Pensylvannia Avenue		2
steve	Jones	123 Cowabunga Lane	555-0000	3
Stephen	Janes	123 Cawabunga Ln	444-555-0000	4

If we could only compare records that were true duplicates, we wouldn't run into the explosion of comparisons. Of course, if we already knew where the true duplicates were, we wouldn't need to compare any individual records. Unfortunately we don't, but we do quite well if just compare records that are somewhat similar.

Blocking

Duplicate records almost always share *something* in common. If we define groups of data that share something and only compare the records in that group, or *block*, then we can dramatically reduce the number of comparisons we will make. If we define these blocks well, then we will make very few comparisons and still have confidence that will compare records that truly are duplicates.

This task is called blocking, and we approach it in two ways: predicate blocks and canopies.

Predicate blocks

A predicate block is a bundle of records that all share a feature – a feature produced by a simple function called a predicate.

Predicate functions take in a record field, and output a set of features for that field. These features could be "the first 3 characters of the field," "every word in the field," and so on. Records that share the same feature become part of a block.

Let's take an example. Let's use a "first 3 character" predicate on the **address field** below..

first name	last name	address	phone	record_id
bob	roberts	1600 pennsylvania ave.	555-0123	1
Robert	Roberts	1600 Pensylvannia Avenue		2
steve	Jones	123 Cowabunga Lane	555-0000	3
Stephen	Janes	123 Cawabunga Ln	444-555-0000	4

That leaves us with two blocks - The '160' block, which contains records 1 and 2, and the '123' block, which contains records 3 and 4.

```
{'160' : (1,2) # tuple of record_ids
 '123' : (3,4)
 }
```

Again, we're applying the "first three characters" predicate function to the address field in our data, the function outputs the following features – 160, 160, 123, 123 – and then we group together the records that have identical features into "blocks".

Others simple predicates Dedupe uses include:

- whole field
- token field
- common integer
- same three char start
- same five char start
- same seven char start
- near integers
- common four gram
- common six gram

Index Blocks

Dedupe also uses another way of producing blocks from searching and index. First, we create a special data structure, like an inverted index, that lets us quickly find records similar to target records. We populate the index with all the unique values that appear in field.

When blocking, for each record we search the index for values similar to the record's field. We block together records that share at least one common search result.

Index predicates require building an index from all the unique values in a field. This can take substantial time and memory. Index predicates are also usually slower than predicate blocking.

Combining blocking rules

If it's good to put define blocks of records that share the same 'city' field, it might be even better to block records that share *both* the 'city' field *and* the 'zip code' field. Dedupe tries these cross-field blocks. These combinations blocks are called disjunctive blocks.

Learning good blocking rules for given data

Dedupe comes with a long set of predicates, and when these are combined Dedupe can have hundreds of possible blocking rules to choose from. We will want to find a small set of these rules that covers every labeled duplicated pair but minimizes the total number pairs dedupe will have to compare.

While we approach this problem by using greedy algorithms, particularly Chvatal's Greedy Set-Cover algorithm.

3.3.3 Grouping Duplicates

Once we have calculated the probability that pairs of record are duplicates or not, we still have a kind of thorny problem because it's not just pairs of records that can be duplicates. Three, four, thousands of records could all refer to the same entity (person, organization, ice cream flavor, etc.,) but we only have pairwise measures.

Let's say we have measured the following pairwise probabilities between records A, B, and C.

```
A -- 0.6 -- B -- 0.6 -- C
```

The probability that A and B are duplicates is 60%, the probability that B and C are duplicates is 60%, but what is the probability that A and C are duplicates?

Let's say that everything is going perfectly and we can say there's a 36% probability that A and C are duplicates. We'd probably want to say that A and C should not be considered duplicates.

Okay, then should we say that A and B are a duplicate pair and C is a distinct record or that A is the distinct record and that B and C are duplicates?

Well… this is a thorny problem, and we tried solving it a few different ways. In the end, we found that **hierarchical clustering with centroid linkage** gave us the best results. What this algorithm does is say that all points within some distance of centroid are part of the same group. In this example, B would be the centroid - and A, B, C and would all be put in the same group.

Unfortunately, a more principled answer does not exist because the estimated pairwise probabilities are not transitive.

Clustering the groups depends on us setting a threshold for group membership – the distance of the points to the centroid. Depending on how we choose that threshold, we'll get very different groups, and we will want to choose this threshold wisely.

In recent years, there has been some very exciting research that solves the problem of turning pairwise distances into clusters, by avoiding making pairwise comparisons altogether. Unfortunately, these developments are not compatible with Dedupe's pairwise approach. See, Michael Wick, et.al, 2012. "A Discriminative Hierarchical Model for Fast Coreference at Large Scale" and Rebecca C. Steorts, et. al., 2013. "A Bayesian Approach to Graphical Record Linkage and De-duplication".

3.3.4 Choosing a Good Threshold

Dedupe can predict the *probability* that a pair of records are duplicates. So, how should we decide that a pair of records really are duplicates?

To answer this question we need to know something about Precision and Recall. Why don't you check out the Wikipedia page and come back here.

There's always a trade-off between precision and recall. That's okay. As long as we know how much we care about precision vs. recall, we can define an F-score that will let us find a threshold for deciding when records are duplicates *that is optimal for our priorities*.

Typically, the way that we find that threshold is by looking at the true precision and recall of some data where we know their true labels - where we know the real duplicates. However, we will only get a good threshold if the labeled examples are representative of the data we are trying to classify.

So here's the problem - the labeled examples that we make with Dedupe are not at all representative, and that's by design. In the active learning step, we are not trying to find the most representative data examples. We're trying to find the ones that will teach us the most.

The approach we take here is to take a random sample of blocked data, and then calculate the pairwise probability that records will be duplicates within each block. From these probabilities we can calculate the expected number of duplicates and distinct pairs, so we can calculate the expected precision and recall.

3.3.5 Special Cases

The process we have been describing is for the most general case–when you have a dataset where an arbitrary number of records can all refer to the same entity.

There are certain special cases where we can make more assumptions about how records can be linked, which if true, make the problem much simpler.

One important case we call Record Linkage. Say you have two datasets and you want to find the records in each dataset that refer to the same thing. If you can assume that each dataset, individually, is unique, then this puts a big constraint on how records can match. If this uniqueness assumption holds, then (A) two records can only refer to the same entity if they are from different datasets and (B) no other record can match either of those two records.

Problems with real-world data

Journalists, academics, and businesses work hard to get big masses of data to learn about what people or organizations are doing. Unfortunately, once we get the data, we often can't answer our questions because we can't tell who is who.

In much real-world data, we do not have a way of absolutely deciding whether two records, say `John Smith` and `J. Smith` are referring to the same person. If these were records of campaign contribution data, did a `John Smith` give two donations or did `John Smith` and maybe `Jane Smith` give one contribution apiece?

People are pretty good at making these calls, if they have enough information. For example, I would be pretty confident that the following two records are the about the same person.

```
first name | last name | address                   | phone    |
--------------------------------------------------------------
bob        | roberts   | 1600 pennsylvania ave.    | 555-0123 |
Robert     | Roberts   | 1600 Pensylvannia Avenue |          |
```

If we have to decide which records in our data are about the same person or organization, then we could just go through by hand, compare every record, and decide which records are about the same entity.

This is very, very boring and can takes a **long** time. Dedupe is a software library that can make these decisions about whether records are about the same thing about as good as a person can, but quickly.

3.4 Bibliography

- http://research.microsoft.com/apps/pubs/default.aspx?id=153478
- http://cs.anu.edu.au/~Peter.Christen/data-matching-book-2012.html
- http://www.umiacs.umd.edu/~getoor/Tutorials/ER_VLDB2012.pdf

3.4.1 New School

- Steorts, Rebecca C., Rob Hall and Stephen Fienberg. "A Bayesian Approach to Record Linkage and De-duplication" December 2013. http://arxiv.org/abs/1312.4645

Very beautiful work. Records are matched to latent individuals. O(N) running time. Unsupervised, but everything hinges on tuning hyperparameters. This work only contemplates categorical variables.

3.4.2 To Read

- Domingos and Domingos Multi-relational record linkage. http://homes.cs.washington.edu/~pedrod/papers/mrdm04.pdf
- An Entity Based Model for Coreference Resolution http://people.cs.umass.edu/~mwick/MikeWeb/Publications_files/wick09entity.pdf

CHAPTER

FOUR

FEATURES

- **machine learning** - reads in human labeled data to automatically create optimum weights and blocking rules
- **runs on a laptop** - makes intelligent comparisons so you don't need a powerful server to run it
- **built as a library** - so it can be integrated in to your applications or import scripts
- **extensible** - supports adding custom data types, string comparators and blocking rules
- **open source** - anyone can use, modify or add to it

CHAPTER FIVE

INSTALLATION

```
pip install dedupe
```

CHAPTER

SIX

USING DEDUPE

Dedupe is a library and not a stand-alone command line tool. To demonstrate its usage, we have come up with a few example recipes for different sized datasets for you (repo, as well as annotated source code:

- Small data deduplication
- Bigger data deduplication ~700K
- Record Linkage
- Postgres
- Patent Author Disambiguation

CHAPTER

SEVEN

ERRORS / BUGS

If something is not behaving intuitively, it is a bug, and should be reported. Report it here

CHAPTER

EIGHT

CONTRIBUTING TO DEDUPE

Check out dedupe repo for how to contribute to the library.

Check out dedupe-examples for how to contribute a useful example of using dedupe.

CHAPTER

NINE

CITING DEDUPE

If you use Dedupe in an academic work, please give this citation:

Gregg, Forest and Derek Eder. 2015. Dedupe. https://github.com/dedupeio/dedupe.

CHAPTER TEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX