Brains Copy Paste Documentation

Release 0.2

Sébastien Lerique

May 30, 2017

Contents

This software is a toolchain for the analysis of mutations in quotes when they propagate through the blog- and news-spaces, measuring some ways in which we alter quotes when we write blog posts. It was developed for a research paper based on the MemeTracker quotes database [LeriqueRoth16], and is released under the GPLv3 license.

This documentation will walk you through the steps to install and run the complete analysis to reproduce the results of the paper, and gives you access to all the tools used during exploration of the data. Hey, these are the days of open science!

We'll be referring to several concepts defined and discussed in the paper, so it might be helpful if you read it first.

CHAPTER 1

Documentation contents

Setup

Setting up the environment for the analyses is a bit involved.

If by chance you know how to use Docker (or are willing to learn – it's super useful and pretty easy!), the easiest way around this really is to use the prebuilt container which has everything included. To do so read the *Quick setup using Docker* section.

Otherwise, or if you want more control over the setup, go to the *Manual setup* section, which walks you through the full show.

Once you're done (either with Docker or with the manual setup), you have access to all the analysis tools. Among other things, this lets you reproduce the figures in the paper.

Quick setup using Docker

If you have Docker installed, just run:

```
docker run -it wehlutyk/brainscopypaste bash
```

That command will download the complete container (which might take a while since it bundles all the necessary data) and start a session in the container. You see a normal shell prompt, which looks like this:

```
brainscopypaste@3651c3dbcc4d:/$
```

Keep a note of the hexadecimal number after the `@` sign (it will be different for you), we'll use it later on to restart this session. It's the ID of your container instance.

Now, in that same shell, start the PostgreSQL server:

```
sudo service postgresql start
```

Then, `cd` into the analysis' home directory and run anything you want from the *Usage* section:

```
cd /home/brainscopypaste
brainscopypaste <any-analysis-command>
# -> the container computes...
brainscopypaste <another-analysis-command>
# -> the container does more computing...
```

Once you're done, just type `exit` (or Ctrl-D) to quit as usual in a terminal.

To restart the same container next time (and not a new instance, which will not now about any analyses you may have run), use your last container's ID:

```
docker start -i <instance-id>
```

(You can also find a more human-readable name associated to that container ID by running `docker ps -a`.)

Now if you're not a fan of Docker, you want see the detailed environment to use it yourself, or for any other reason you want to manually set up the analysis environment, keep reading below.

Manual setup

There are a few packages to install to get up and running.

We'll assume you're using a Debian/Ubuntu system from now on. If it's not the case, do this on a virtual machine with Debian/Ubuntu, or figure out yourself how to do it on your own system (be it OS X, Windows, or any other OS).

The installation breaks down into 6 steps:

1. *Install preliminary dependencies*
2. *Create and configure the environment*
3. *Configure the database*
4. *Install TreeTagger*
5. *Install datasets*
6. *Check everything works*

Note: This software was tested on Python 3.5 ONLY (which is what the docker container uses). Any other version might (and probably will) generate unexpected errors.

Now let's get started.

Install preliminary dependencies

First, there's a bunch of packages we're going to need: among them are virtualenv and virtualenvwrapper to isolate the environment, PostgreSQL for database handling, a LaTeX distribution for math rendering in figures, and some build-time dependencies. To get all the necessary stuff in one fell swoop, run:

```
sudo apt-get install virtualenv virtualenvwrapper \
    postgresql postgresql-server-dev texlive texlive-latex-extra \
    pkg-config python3-dev build-essential \
    libfreetype6-dev libpng12-0 libpng12-dev tk-dev
```

Then close and reopen your terminal (this loads the virtualenvwrapper scripts at startup).

Create and configure the environment

Now clone the main repository and `cd` into it:

```
git clone https://github.com/wehlutyk/brainscopypaste
cd brainscopypaste
```

Next, create a Python 3 virtual environment, and install the dependencies:

```
# Create the virtual environment
mkvirtualenv -p $(which python3) brainscopypaste

# Install NumPy first, which is required for the second line to work
pip install $(cat requirements.txt | grep "^numpy")
pip install -r requirements.txt
# Finally install the `brainscopypaste` command-line tool
pip install --editable .
```

While these instructions should be pretty foolproof, installing some of the dependencies (notably Matplotlib) can be a bit complicated. If you run into problems, look at the Matplotlib installation instructions. Another solution is to use the Anaconda distribution (but you have to juggle with nested anaconda and virtualenv environments in that case).

Note: All further shell commands are assumed to be running inside this new virtual environment. It is activated automatically after the `mkvirtualenv` command, but you can activate it manually in a new shell by running `workon brainscopypaste`.

Configure the database

First, the default configuration for PostgreSQL on Ubuntu requires a password for users other than `postgres` to connect, so we're going to change that to make things simpler: edit the `/etc/postgresql/<postgres-version>/main/pg_hba.conf` file (in my case, I run `sudo nano /etc/postgresql/9.5/main/pg_hba.conf`), and find the following lines, usually at the end of the file:

```
# "local" is for Unix domain socket connections only
local   all             all                                     peer
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
```

Change the last column of those three lines to `trust`, so they look like this:

```
# "local" is for Unix domain socket connections only
local   all             all                                     trust
# IPv4 local connections:
host    all             all             127.0.0.1/32            trust
# IPv6 local connections:
host    all             all             ::1/128                 trust
```

This configures PostgreSQL so that any user in the local system can connect as any database user. Then, restart the database service to apply the changes:

```
sudo service postgresql restart
```

Finally, create the user and databases used by the toolchain:

```
psql -c 'create user brainscopypaste;' -U postgres
psql -c 'create database brainscopypaste;' -U postgres
```

```
psql -c 'alter database brainscopypaste owner to brainscopypaste;' -U postgres
psql -c 'create database brainscopypaste_test;' -U postgres
psql -c 'alter database brainscopypaste_test owner to brainscopypaste;' -U postgres
```

Note: If you'd rather keep passwords for your local connections, then set a password for the `brainscopypaste` database user we just created, and put that password in the `DB_PASSWORD` variable of the *Database credentials* section of `brainscopypaste/settings.py`.

Install TreeTagger

TreeTagger is used to extract POS tags and lemmas from sentences, so is needed for all mining steps. Install it by running:

```
./install_treetagger.sh
```

Note: TreeTagger isn't packaged for usual GNU/Linux distributions, and the above script will do the install locally for you. If you're running another OS, you'll have to adapt the script to download the proper executable. See the project website for more information.

Install datasets

The analyses use the following datasets for mining and word feature extraction:

- WordNet data
- CMU Pronunciation Dictionary data
- Free Association Norms
- Age-of-Acquisition Norms
- CLEARPOND data
- MemeTracker dataset

You can install all of these in one go by running:

```
./install_datasets.sh
```

Note: Age-of-Acquisition Norms are in fact already included in the cloned repository, because they needed to be converted from `xslx` to `csv` format (which is a pain to do in Python).

Check everything works

The toolchain has an extensive test suite, which you should now be able to run. Still in the main repository with the virtual environment activated, run:

```
py.test
```

This should take about 5-10 minutes to complete (it will skip a few tests since we haven't computed all necessary features yet).

If you run into problems, say some tests are failing, try first rerunning the test suite (the language detection module introduces a little randomness, leading a few tests to fail sometimes), then double check all the instructions above to make sure you followed them well. If the problem persists please create an issue on the repository's bugtracker, because you may have found a bug!

If everything works, congrats! You're good to go to the next section: *Usage*.

Usage

This section explains how to re-run the full analysis (including what is described in the paper). The general flow for the analysis is as follows:

1. *Preload all necessary data*, which consists of the following 3 steps:
   (a) *Load the MemeTracker data into the database*
   (b) *Preprocess the MemeTracker data*
   (c) *Load and compute word features*
2. *Analyse substitutions mined by one model*, which consists of the following 2 steps:
   (a) *Mine for substitutions*
   (b) *Run the analysis notebooks*

Once you did that for a particular substitution model, you can do the *Analysis exploring all mining models*.

Now let's get all this running!

Preload all necessary data

The first big part is to load and preprocess all the bits necessary for the analysis. Let's go:

Load the MemeTracker data into the database

The MemeTracker data comes in a text-based format, which isn't suitable for the analysis we want to perform. So the first thing we do is load it into a PostgreSQL database. First, make sure the database service is running:

```
sudo service postgresql start
```

Then, from inside the analysis' repository (with the virtual environment activated if you're not using Docker — see the *Setup* section if you're lost here), tell the toolchain to load the MemeTracker data into the database:

```
brainscopypaste load memetracker
```

This might take a while to complete, as the MemeTracker data takes up about 1GB and needs to be processed for the database. The command-line tool will inform you about its progress.

Preprocess the MemeTracker data

Now, the data we just loaded contains quite some noise. Our next step is to filter out all the noise we can, to work on a cleaner data set overall. To do so, run:

```
brainscopypaste filter memetracker
```

This is also a bit long (but, as usual, informs you of the progress).

Load and compute word features

The final preloading step is to compute the features we'll use on words involved in substitutions. This comes after loading and filtering the MemeTracker data, since some features (like word frequency) are computed on the filtered MemeTracker data itself. To load all the features, run:

```
brainscopypaste load features
```

Now you're ready to mine substitutions and plot the results.

Analyse substitutions mined by one model

So first, choose a substitution model (read the paper for more information on this). If you want to use the model detailed in the paper, just follow the instructions below.

Mine for substitutions

To mine for all the substitutions that the model presented in the paper detects, run:

```
brainscopypaste mine substitutions Time.discrete Source.majority Past.last_bin Durl.
↪all 1
```

This will iterate through the MemeTracker data, detect all substitutions that conform to the main model presented in the paper, and store them in the database.

Head over to the *Command-line interface* reference for more details about what the arguments in this command mean.

Run the analysis notebooks

Once substitutions are mined, results are obtained by running the Jupyter notebooks located in the `notebooks/` folder. To do so, still in the same terminal, run:

```
jupyter notebook
```

Which will open the Jupyter file browser in your web browser.

Then click on the `notebooks/` folder, and open any analysis notebook you want and run it. All the figures presenting results in the paper come from these notebooks.

Note: If you used another substitution model than the one used above, you must correct the corresponding `model = Model(...)` line in the `distance.ipynb`, `susceptibility.ipynb`, and `variation.ipynb` notebooks.

Analysis exploring all mining models

Part of the robustness of the analysis comes from the fact that results are reproducible across substitution models. To compute the results for all substitution models, you must first mine all the possible substitutions. This can be done with the following command:

```
for time in discrete continuous; do \
  for source in majority all; do \
    for past in last_bin all; do \
      for durl in all exclude_past; do \
        for maxdistance in 1 2; do \
          echo "\n-----\n\nDoing Time.$time Source.$source Past.$past Durl.$durl
↪$maxdistance"; \
          brainscopypaste mine substitutions Time.$time Source.$source Past.$past␣
↪Durl.$durl $maxdistance; \
        done; \
      done; \
    done; \
  done; \
done;
```

(This will take a loooong time to complete. The `Time.continuous|discrete Source.all Past.all Durl.all 1|2` models especially, will use a lot of RAM.)

Once substitutions are mined for all possible models (or a subset of those), you can run notebooks for each model directly in the command-line (i.e. without having to open each notebook in the browser) with the `brainscopypaste variant <model-parameters> <notebook-file>` command. It will create a copy of the notebook you asked for, set the proper `model = Model(...)` line in it, run it and save it in the `data/notebooks/` folder. All the figures produced by that notebook will also be saved in the `data/figures/<model> -<notebook>/` folder.

So to run the whole analysis for all models, after mining for all models, run:

```
for time in discrete continuous; do \
  for source in majority all; do \
    for past in last_bin all; do \
      for durl in all exclude_past; do \
        for maxdistance in 1 2; do \
          echo "\n-----\n\nDoing Time.$time Source.$source Past.$past Durl.$durl
↪$maxdistance"; \
          brainscopypaste variant Time.$time Source.$source Past.$past Durl.$durl
↪$maxdistance notebooks/distance.ipynb; \
          brainscopypaste variant Time.$time Source.$source Past.$past Durl.$durl
↪$maxdistance notebooks/susceptibility.ipynb; \
          brainscopypaste variant Time.$time Source.$source Past.$past Durl.$durl
↪$maxdistance notebooks/variation.ipynb; \
        done; \
      done; \
    done; \
  done; \
done;
```

Needless to say, this plus mining will take at least a couple days to complete.

If you want to know more to try and hack on the analysis on the notebooks, head over to the *Reference*.

Reference

Contents:

Command-line interface

CLI tool for stepping through the analysis.

Once you have the environment properly set up (see *Setup*), invoke this tool with `brainscopypaste <command>`.

The documentation for this tool can be explored using `brainscopypaste --help` or `brainscopypaste <command> --help`. If you are viewing these docs in the browser, you will only see docstrings for convenience functions in the module. The other docstrings appear in the source code, but are best explored by calling the tool with `--help`.

`brainscopypaste.cli.`**`_drop_features`**`()`
Drop computed features from the filesystem.

`brainscopypaste.cli.`**`cliobj`**`()`
Convenience function to launch the CLI tool, used by the *setup.py* script.

`brainscopypaste.cli.`**`confirm`**`(`*fillin*`)`
Ask the user to confirm they want to drop the content described by *fillin*.

Database models

Database models and related utilities.

This module defines the database structure underlying storage for the analysis. This consists in models that get turned into PostgreSQL tables by SQLAlchemy, along with a few utility classes, functions and exceptions around them.

Cluster and *Quote* represent respectively an individual cluster or quote from the MemeTracker data set. *Url* represents a quote occurrence, and those are stored as attributes of *Quote*s (as opposed to in their own table). *Substitution* represents an individual substitution mined with a given substitution *Model*.

Each model (except *Url*, which doesn't have its own table) inherits the *BaseMixin*, which defines the table name, *id* field, and provides a common *clone()* method.

On top of that, models define a few computed properties (using the *utils.cache* decorator) which provide useful information that doesn't need to be stored directly in the database (storing that in the database would make first access faster, but introduces more possibilities of inconsistent data if updates don't align well). *Cluster* and *Substitution* also inherit functionality from the *mine*, *filter* and *features* modules, which you can inspect for more details.

Finally, this module defines *save_by_copy()*, a useful function to efficiently import clusters and quotes in bulk into the database.

class `brainscopypaste.db.`**`ArrayOfEnum`**`(`*item_type*, *as_tuple=False*, *dimensions=None*, *zero_indexes=False*`)`
Bases: `sqlalchemy.dialects.postgresql.base.ARRAY`

ARRAY of ENUMs column type, which is not directly supported by DBAPIs.

This workaround is provided by SQLAchemy's documentation.

class `brainscopypaste.db.`**`BaseMixin`**
Bases: `object`

Common mixin for all models defining a table name, an *id* field, and a *clone()* method.

clone(***fields*)
Clone a model instance, excluding the original *id* and optionally setting some fields to values provided as arguments.

Give the fields to override as keyword arguments, their values will be set on the cloned instance. Any field that is not a known table column is ignored.

id = Column(None, Integer(), table=None, primary_key=True, nullable=False)
Primary key for the table.

class brainscopypaste.db.**Cluster**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base, *brainscopypaste.db.BaseMixin*, *brainscopypaste.filter.ClusterFilterMixin*, *brainscopypaste.mine.ClusterMinerMixin*

Represent a MemeTracker cluster of quotes in the database.

Attributes below are defined as class attributes or *cache*d methods, but they appear as instance attributes when you have an actual cluster instance. For instance, if *cluster* is a *Cluster* instance, *cluster.size* will give you that instance's *size*.

See also:

filter.ClusterFilterMixin, *mine.ClusterMinerMixin*

filtered
Boolean indicating whether this cluster is part of the filtered (and kept) set of clusters or not.

format_copy()
Create a string representing the cluster in a `cursor.copy_from()` or *_copy()* call.

format_copy_columns = ('id', 'sid', 'filtered', 'source')
Tuple of column names that are used by *format_copy()*.

frequency
Complete number of occurrences of all the quotes in the cluster (i.e. counting url frequencies).

Look at *size_urls* for a count that ignores url frequencies.

quotes
List of *Quote*s in this cluster (this is a dynamic relationship on which you can run queries).

sid
Id of the cluster that originated this instance, i.e. the id as it appears in the MemeTracker data set.

size
Number of quotes in the cluster.

size_urls
Number of urls of all the quotes in the cluster (i.e. not counting url frequencies)

Look at *frequency* for a count that takes url frequencies into account.

source
Source data set from which this cluster originated. Currently this is always *memetracker*.

span
Span of the cluster (as a `timedelta`), from first to last occurrence.

Raises ValueError

If no urls are defined on any quotes of the cluster.

urls
Unordered list of *Url*s of all the quotes in the cluster.

class brainscopypaste.db.**ModelType**(**args*, ***kwargs*)
Bases: sqlalchemy.sql.type_api.TypeDecorator

Database type representing a substitution *Model*, used in the definition of *Substitution*.

impl
alias of String

process_bind_param(*value*, *dialect*)
Convert a *Model* to its database representation.

process_result_value(*value*, *dialect*)
Create a *Model* instance from its database representation.

class brainscopypaste.db.**Quote**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base, *brainscopypaste.db.BaseMixin*

Represent a MemeTracker quote in the database.

Attributes below are defined as class attributes or *cache*d methods, but they appear as instance attributes when you have an actual quote instance. For instance, if *quote* is a *Quote* instance, *quote.size* will give you that instance's *size*.

Note that children *Url*s are stored directly inside this model through lists of url attributes, where a given url is defined by items at the same index in the various lists. This is an internal detail, and you should use the *urls* attribute to directly get a list of *Url* objects.

add_url(*url*)
Add a *Url* to the quote.

The change is not automatically saved. If you want to persist this to the database, you should do it inside a session and commit afterwards (e.g. using *session_scope()*).

Parameters **url** : *Url*

The url to add to the quote.

Raises **SealedException**

If the *urls* attribute has already been accessed; since that attribute is *cache*d, adding a url to the quote would invalidate the value.

add_urls(*urls*)
Add a list of *Url*s to the quote.

As for *add_url()*, the changes are not automatically saved. If you want to persist this to the database, you should do it inside a session and commit afterwards (e.g. using *session_scope()*).

Parameters **urls** : list of *Url*s

The urls to add to the quote.

Raises **SealedException**

If the *urls* attribute has already been accessed; since that attribute is *cache*d, adding urls to the quote would invalidate the value.

cluster
Parent *Cluster*.

cluster_id
Parent cluster id.

filtered
Boolean indicating whether this quote is part of the filtered (and kept) set of quotes or not.

format_copy()
Create a string representing the quote and all its children urls in a `cursor.copy_from()` or *_copy()* call.

format_copy_columns = ('id', 'cluster_id', 'sid', 'filtered', 'string', 'url_timestamps', 'url_frequencies', 'url_url_type
Tuple of column names that are used by *format_copy()*.

frequency
Complete number of occurrences of the quote (i.e. counting url frequencies).

Look at *size* for a count that ignores url frequencies.

lemmas
List of the lemmas in the quote's *string*.

Raises ValueError

If the quote's *string* is *None*.

sid
Id of the quote that originated this instance, i.e. the id as it appears in the MemeTracker data set.

size
Number of urls in the quote.

Look at *frequency* for a count that takes url frequencies into account.

span
Span of the quote (as a `timedelta`), from first to last occurrence.

Raises ValueError

If no urls are defined on the quote.

string
Text of the quote.

substitutions_destination
List of *Substitution*s for which this quote is the destination (this is a dynamic relationship on which you can run queries).

substitutions_source
List of *Substitution*s for which this quote is the source (this is a dynamic relationship on which you can run queries).

tags
List of TreeTagger POS tags of the tokens in the quote's *string*.

Raises ValueError

If the quote's *string* is *None*.

tokens
List of the tokens in the quote's *string*.

Raises ValueError

If the quote's *string* is *None*.

url_frequencies
List of *int*s representing the frequencies of children urls (i.e. how many times the quote string appears at each url).

url_timestamps
List of `datetime`s representing the times at which children urls appear.

url_url_types
List of *url_type*s representing the types of the children urls.

url_urls
List of *str*s representing the URIs of the children urls.

urls
Unordered list of *Url*s of the quote; use this to access urls of the quote, instead of the *url_** attributes.

exception brainscopypaste.db.**SealedException**
Bases: Exception

Exception raised when trying to edit a model on which *cache*d methods have already been accessed.

class brainscopypaste.db.**Substitution**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base, *brainscopypaste.db.BaseMixin*, *brainscopypaste.mine.SubstitutionValidatorMixin*, *brainscopypaste.features.SubstitutionFe*

Represent a substitution in the database from one *Quote* to another.

A substitution is the replacement of a word from one quote (or a substring of that quote) in another quote. It is defined by a *source quote*, an *occurrence* of a *destination quote*, the *position of a substring* in the source quote string, the *position of the replaced word* in that substring, and the *substitution model* that detected the substitution in the data set.

Attributes below are defined as class attributes or *cache*d methods, but they appear as instance attributes when you have an actual substitution instance. For instance, if *substitution* is a *Substitution* instance, *substitution.tags* will give you that instance's *tags*.

See also:

mine.SubstitutionValidatorMixin, *features.SubstitutionFeaturesMixin*

destination
Destination *Quote* for the substitution.

destination_id
Id of the destination quote for the substitution.

lemmas
Tuple of lemmas of the replaced and replacing words.

model
Substitution detection *Model* that detected this substitution.

occurrence
Index of the destination *Url* in the destination quote.

position
Position of the replaced word *in the substring of the source quote* (which is also the position in the destination quote).

source
Source *Quote* for the substitution.

source_id
Id of the source quote for the substitution.

start
Index of the beginning of the substring in the source quote.

tags
Tuple of TreeTagger POS tags of the replaced and replacing words.

tokens
Tuple of the replaced and replacing words (the tokens here are the exact replaced and replacing words).

class brainscopypaste.db.**Url**(*timestamp*, *frequency*, *url_type*, *url*, *quote=None*)
Bases: object

Represent a MemeTracker url in a *Quote* in the database.

The url *occurrence* is defined below as a *cache*d method, but it appears as an instance attribute when you have an actual url instance. For instance, if *url* is a *Url* instance, *url.occurrence* will give you that url's *occurrence*.

Note that *Url*s are stored directly inside *Quote* instances, and don't have a dedicated database table.

Attributes

quote	(*Quote*) Parent quote.
timestamp	(datetime) Time at which the url occurred.
frequency	(int) Number of times the quote string appears at this url.
url_type	(*url_type*) Type of this url.
url	(str) URI of this url.

_Url__key()
Unique identifier for this url, used to compute e.g. equality between two *Url* instances.

occurrence
Index of the url in the list of urls of the parent *Quote*.

Raises **ValueError**

If the url's *quote* attribute is *None*.

brainscopypaste.db.**_copy**(*string*, *table*, *columns*)
Execute a PostgreSQL COPY command.

COPY is one of the fastest methods to import data in bulk into PostgreSQL. This function executes this operation through the raw psycopg2 cursor object.

Parameters **string** : file-like object

Contents of the data to import into the database, formatted for the COPY command (see PostgreSQL's documentation for more details). Can be an io.StringIO if you don't want to use a real file in the filesystem.

table : str

Name of the table into which the data is imported.

columns : list of str

List of the column names encoded in the *string* parameter. When *string* is produced using *Quote.format_copy()* or *Cluster.format_copy()* you can use the corresponding *Quote.format_copy_columns* or *Cluster.format_copy_columns* for this parameter.

See also:

save_by_copy, *Quote.format_copy*, *Cluster.format_copy*

brainscopypaste.db.**save_by_copy**(*clusters*, *quotes*)
Import a list of clusters and a list of quotes into the database.

This function uses PostgreSQL's COPY command to bulk import clusters and quotes, and prints its progress to stdout.

Parameters **clusters** : list of *Cluster*s

List of clusters to import in the database.

quotes : list of *Quote*s

List of quotes to import in the database. Any clusters they reference should be in the *clusters* parameter.

See also:

load.MemeTrackerParser.parse

brainscopypaste.db.**url_type** = Enum('B', 'M', name='url_type', metadata=MetaData(bind=None))
sqlalchemy.types.Enum of possible types of *Url*s from the MemeTracker data set.

Features

Features for words in substitutions.

This module defines the *SubstitutionFeaturesMixin* which is used to augment *Substitution*s with convenience methods that give access to feature values and related computed values (e.g. sentence-relative feature values and values for composite features).

A few other utility functions that load data for the features are also defined.

class brainscopypaste.features.**SubstitutionFeaturesMixin**
Bases: object

Mixin for *Substitution*s adding feature-related functionality.

Methods in this class fall into 3 categories:

- Raw feature methods: they are *memoized()* class methods of the form *cls._feature_name(cls, word=None)*. Calling them with a *word* returns either the feature value of that word, or *np.nan* if the word is not encoded. Calling them with *word=None* returns the set of words encoded by that feature (which is used to compute e.g. averages over the pool of words encoded by that feature). Their docstring (which you will see below if you're reading this in a web browser) is the short name used to identify e.g. the feature's column in analyses in notebooks. These methods are used internally by the class, to provide the next category of methods.
- Useful feature methods that can be used in analyses: *features()*, *feature_average()*, *source_destination_features()*, *components()*, and *component_average()*. These methods use the raw feature methods (previous category) and the utility methods (next category) to compute feature or composite values (eventually relative to sentence) on the source or destination words or sentences.
- Private utility methods: *_component()*, *_source_destination_components()*, *_average()*, *_static_average()*, *_strict_synonyms()*, *_substitution_features()*, and *_transformed_feature()*. These methods are used by the previous category of methods.

Read the source of the first category (raw features) to know how exactly an individual feature is computed. Read the docstrings (and source) of the second category (useful methods for analyses) to learn how to use this class in analyses. Read the docstrings (and source) of the third category (private utility methods) to learn how the whole class assembles its different parts together.

classmethod **_aoa**(*word=None*)
age of acquisition

_average (*func*, *source_synonyms*)
Compute the average value of *func* over the words it codes, or over the synonyms of this substitution's source word.

If *source_synonyms* is *True*, the method computes the average feature of the synonyms of the source word of this substitution. Otherwise, it computes the average over all words coded by *func*.

The method is *memoized()* since it is called so often.

Parameters **func** : function

The function to average. Calling *func()* must return the pool of words that the function codes. Calling *func(word)* must return the value for *word*.

source_synonyms : bool

If *True*, compute the average *func* of the synonyms of the source word in this substitution. If *False*, compute the average over all coded words.

Returns float

Average *func* value.

classmethod _betweenness (*word=None*)
betweenness

classmethod _clustering (*word=None*)
clustering

classmethod _component (*n*, *pca*, *feature_names*)
Get a function computing the *n*-th component of *pca* using *feature_names*.

The method is *memoized()* since it is called so often.

Parameters **n** : int

Index of the component in *pca* that is to be computed.

pca : sklearn.decomposition.PCA

PCA instance that was computed using the features listed in *feature_names*.

feature_names : tuple of str

Tuple of feature names used in the computation of *pca*.

Returns **component** : function

The component function, with signature *component(word=None)*. Call *component()* to get the set of words encoded by that component (which is the set of words encoded by all features in *feature_names*). Call *component(word)* to get the component value of *word* (or *np.nan* if *word* is not coded by that component).

Examples

Get the first component of "dog" in a PCA with very few words, using features *aoa*, *frequency*, and *letters_count*:

```
>>> mixin = SubstitutionFeaturesMixin()
>>> feature_names = ('aoa', 'frequency', 'letters_count')
>>> features = list(map(mixin._transformed_feature,
...                     feature_names))
>>> values = np.array([[f(w) for f in features]
```

```
...                     for w in ['bird', 'cat', 'human']])
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> pca.fit(values)
>>> mixin._component(0, pca, feature_names)('dog')
-0.14284518091970733
```

classmethod _degree(*word=None*)
degree

classmethod _frequency(*word=None*)
frequency

classmethod _letters_count(*word=None*)
#letters

classmethod _orthographic_density(*word=None*)
orthographic nd

classmethod _pagerank(*word=None*)
pagerank

classmethod _phonemes_count(*word=None*)
<#phonemes>

classmethod _phonological_density(*word=None*)
phonological nd

_source_destination_components(*n*, *pca*, *feature_names*)
Compute the *n*-th component of pca for all words in source and destination sentences of this substitution.

The method is *memoized()* since it is called so often.

Parameters **n** : int

Index of the component in *pca* that is to be computed.

pca : sklearn.decomposition.PCA

PCA instance that was computed using the features listed in *feature_names*.

feature_names : tuple of str

Tuple of feature names used in the computation of *pca*.

Returns **source_components** : array of float

Array of component values for each word in the source sentence of this substitution. Non-coded words appear as *np.nan*.

destination_components : array of float

Array of component values for each word in the destination sentence of this substitution. Non-coded words appear as *np.nan*.

static _static_average(*func*)
Static version of *_average()*, without the *source_synonyms* argument.

The method is *memoized()* since it is called so often.

classmethod _strict_synonyms(*word*)
Get the set of synonyms of *word* through WordNet, excluding *word* itself; empty if nothing is found.

_substitution_features(*name*)
Compute feature *name* for source and destination words of this substitution.

Feature values are transformed as explained in *_transformed_feature()*.

The method is *memoized()* since it is called so often.

Parameters **name** : str

Name of the feature for which to compute source and destination values.

Returns tuple of float

Feature values of the source and destination words of this substitution.

classmethod _syllables_count(*word=None*)
<#syllables>

classmethod _synonyms_count(*word=None*)
<#synonyms>

classmethod _transformed_feature(*name*)
Get a function computing feature *name*, transformed as defined by __features__.

Some features have a very skewed distribution (e.g. exponential, where a few words are valued orders of magnitude more than the vast majority of words), so we use their log-transformed values in the analysis to make them comparable to more regular features. The __features__ attribute (which appears in the source code but not in the web version of these docs) defines which features are transformed how. Given a feature *name*, this method will generate a function that proxies calls to the raw feature method, and transforms the value if necessary.

This method is *memoized()* for speed, since other methods call it all the time.

Parameters **name** : str

Name of the feature for which to create a function, without preceding underscore; for instance, call *cls._transformed_feature('aoa')* to get a function that uses the *_aoa()* class method.

Returns **feature** : function

The feature function, with signature *feature(word=None)*. Call *feature()* to get the set of words encoded by that feature. Call *feature(word)* to get the transformed feature value of *word* (or *np.nan* if *word* is not coded by that feature).

Examples

Get the transformed frequency value of "dog":

```
>>> mixin = SubstitutionFeaturesMixin()
>>> logfrequency = mixin._transformed_feature('frequency')
>>> logfrequency('dog') == np.log(mixin._frequency('dog'))
True
```

component_average(*n*, *pca*, *feature_names*, *source_synonyms=False*, *sentence_relative=None*)
Compute the average, over all coded words or synonyms of this substitution's source word, of the *n*-th component of *pca* using *feature_names*, possibly sentence-relative.

If *source_synonyms* is *True*, the method computes the average component of the synonyms of the source word of this substitution. Otherwise, it computes the average over all words coded by the component.

If *sentence_relative* is not *None*, it indicates a NumPy function used to aggregate word components in the source sentence of this substitution; this method then returns the component average minus that aggregate value. For instance, if *sentence_relative='median'*, this method returns the average component minus the median component value in the source sentence (words valued at *np.nan* are ignored).

The method is `memoized()` since it is called so often.

Parameters **n** : int

Index of the component in *pca* that is to be computed.

pca : `sklearn.decomposition.PCA`

`PCA` instance that was computed using the features listed in *feature_names*.

feature_names : tuple of str

Tuple of feature names used in the computation of *pca*.

source_synonyms : bool, optional

If *True*, compute the average component of the synonyms of the source word in this substitution. If *False* (default), compute the average over all coded words.

sentence_relative : str, optional

If not *None* (which is the default), return average component relative to component values of the source sentence of this substitution aggregated by this function; must be a name for which *np.nan<sentence_relative>* exists.

Returns float

Average component, of all coded words or of synonyms of the substitution's source word (depending on *source_synonyms*), relative to an aggregated source sentence value if *sentence_relative* specifies it.

components(*n*, *pca*, *feature_names*, *sentence_relative=None*)

Compute the *n*-th components of *pca* for source and destination words of this substitution, possibly sentence-relative.

If *sentence_relative* is not *None*, it indicates a NumPy function used to aggregate word components in the source and destination sentences of this substitution; this method then returns the source/destination word component values minus the corresponding aggregate value. For instance, if *sentence_relative='median'*, this method returns the source word component minus the median of the source sentence, and the destination word component minus the median of the destination sentence (words valued at *np.nan* are ignored).

The method is `memoized()` since it is called so often.

Parameters **n** : int

Index of the component in *pca* that is to be computed.

pca : `sklearn.decomposition.PCA`

`PCA` instance that was computed using the features listed in *feature_names*.

feature_names : tuple of str

Tuple of feature names used in the computation of *pca*.

sentence_relative : str, optional

If not *None* (which is the default), return components relative to values of their corresponding sentence aggregated by this function; must be a name for which *np.nan<sentence_relative>* exists.

Returns tuple of float

Components (possibly sentence-relative) of the source and destination words of this substitution.

feature_average(*name*, *source_synonyms=False*, *sentence_relative=None*)

Compute the average of feature *name* over all coded words or over synonyms of this substitution's source word, possibly sentence-relative.

If *source_synonyms* is *True*, the method computes the average feature of the synonyms of the source word of this substitution. Otherwise, it computes the average over all words coded by the feature.

If *sentence_relative* is not *None*, it indicates a NumPy function used to aggregate word features in the source sentence of this substitution; this method then returns the feature average minus that aggregate value. For instance, if *sentence_relative='median'*, this method returns the average feature minus the median feature value in the source sentence (words valued at *np.nan* are ignored).

The method is *memoized()* since it is called so often.

Parameters **name** : str

Name of the feature for which to compute an average.

source_synonyms : bool, optional

If *True*, compute the average feature of the synonyms of the source word in this substitution. If *False* (default), compute the average over all coded words.

sentence_relative : str, optional

If not *None* (which is the default), return average feature relative to feature values of the source sentence of this substitution aggregated by this function; must be a name for which *np.nan<sentence_relative>* exists.

Returns float

Average feature, of all coded words or of synonyms of the substitution's source word (depending on *source_synonyms*), relative to an aggregated source sentence value if *sentence_relative* specifies it.

features(*name*, *sentence_relative=None*)

Compute feature *name* for source and destination words of this substitution, possibly sentence-relative.

Feature values are transformed as explained in *_transformed_feature()*.

If *sentence_relative* is not *None*, it indicates a NumPy function used to aggregate word features in the source and destination sentences of this substitution; this method then returns the source/destination word feature values minus the corresponding aggregate value. For instance, if *sentence_relative='median'*, this method returns the source word feature minus the median of the source sentence, and the destination word feature minus the median of the destination sentence (words valued at *np.nan* are ignored).

The method is *memoized()* since it is called so often.

Parameters **name** : str

Name of the feature for which to compute source and destination values.

sentence_relative : str, optional

If not *None* (which is the default), return features relative to values of their corresponding sentence aggregated by this function; must be a name for which *np.nan<sentence_relative>* exists.

Returns tuple of float

Feature values (possibly sentence-relative) of the source and destination words of this substitution.

source_destination_features(*name*, *sentence_relative=None*)

Compute the feature values for all words in source and destination sentences of this substitution, possibly sentence-relative.

Feature values are transformed as explained in *_transformed_feature()*.

If *sentence_relative* is not *None*, it indicates a NumPy function used to aggregate word features in the source and destination sentences of this substitution; this method then returns the source/destination feature values minus the corresponding aggregate value. For instance, if *sentence_relative='median'*, this method returns the source sentence feature values minus the median of that same sentence, and the destination sentence feature values minus the median of that same sentence (words valued at *np.nan* are ignored).

The method is *memoized()* since it is called so often.

Parameters **name** : str

Name of the feature for which to compute source and destination values.

sentence_relative : str, optional

If not *None* (which is the default), return features relative to values of their corresponding sentence aggregated by this function; must be a name for which *np.nan*<*sentence_relative*> exists.

Returns **source_features** : array of float

Array of feature values (possibly sentence-relative) for each word in the source sentence of this substitution. Non-coded words appear as *np.nan*.

destination_features : array of float

Array of feature values (possibly sentence-relative) for each word in the destination sentence of this substitution. Non-coded words appear as *np.nan*.

brainscopypaste.features.**_get_aoa**()

Get the Age-of-Acquisition data as a dict.

The method is *memoized()* since it is called so often.

Returns dict

Association of words to their average age of acquisition. *NA* values in the originating data set are ignored.

brainscopypaste.features.**_get_clearpond**()

Get CLEARPOND neighbourhood density data as a dict.

The method is *memoized()* since it is called so often.

Returns dict

Dict with two keys: *orthographic* and *phonological*. *orthographic* contains a dict associating words to their orthographic neighbourhood density (CLEARPOND's *OTAN* column). *phonological* contains a dict associating words to their phonological neighbourhood density (CLEARPOND's *PTAN* column).

brainscopypaste.features.**_get_pronunciations**()

Get the CMU pronunciation data as a dict.

The method is *memoized()* since it is called so often.

Returns dict

Association of words to their list of possible pronunciations.

Filtering

Filter clusters and quotes to clean to MemeTracker dataset.

This module defines the *ClusterFilterMixin* mixin which adds filtering capabilities to *Cluster*, and the *filter_clusters()* function which uses that mixin to filter the whole MemeTracker dataset. A few other utility functions are also defined.

exception brainscopypaste.filter.**AlreadyFiltered**
Bases: Exception

Exception raised when trying to filter a dataset that has already been filtered.

class brainscopypaste.filter.**ClusterFilterMixin**
Bases: object

Mixin for *Cluster*s adding the *filter()* method used in *filter_clusters()*.

filter()
Filter this *Cluster* and its children *Quote*s to see if they're worth keeping.

First, iterate through all the children *Quote*s of the cluster, seeing if each one of them is worth keeping. A *Quote* is discarded if it has no urls, less than *MT_FILTER_MIN_TOKENS*, spans longer than *MT_FILTER_MAX_DAYS*, or is not in English. Any *Quote* that has none of those problems will be kept.

If after this filtering there are no *Quote*s left, or the *Cluster* made of the remaining *Quote*s still spans longer than *MT_FILTER_MAX_DAYS*, the cluster and all its quotes will be discarded and *None* is returned. If not, a new *Cluster* is created with *cluster.filtered* = *True* and *cluster.id* = *original_cluster.id* + *filter_cluster_offset()*. That new cluster points to copies of all the kept *Quote*s, with *quote.filtered* = *True* and *quote.id* = *original_quote.id* + *filter_quote_offset()*. All those models (new cluster and new quotes) should later be saved to the database (the method does not do it for you), e.g. by running this method inside a *session_scope()*.

Returns **cluster** : *Cluster* or None

The filtered cluster pointing to filtered quotes, or *None* if it is to be discarded.

Raises **AlreadyFiltered**

If this cluster is already filtered (i.e. *filtered* is *True*).

brainscopypaste.filter.**_top_id**(*id*)
Get the smallest power of ten three orders of magnitude greater than *id*.

Used to compute *filter_cluster_offset()* and *filter_quote_offset()*.

brainscopypaste.filter.**filter_cluster_offset**()
Get the offset to add to filtered *Cluster* ids.

A filtered *Cluster*'s id will be its original *Cluster*'s id plus this offset. The function is *memoized()* since it is called so often.

brainscopypaste.filter.**filter_clusters**(*limit=None*)
Filter the whole MemeTracker dataset by copying all valid *Cluster*s and *Quote*s and setting their *filtered* attributes to *True*.

Iterate through all the MemeTracker *Cluster*s, and filter each of them to see if it's worth keeping. If a *Cluster* is to be kept, the function creates a copy of it and all of its kept *Quote*s, marking them as filtered. Progress of this operation is printed to stdout.

Once the operation finishes, a VACUUM and an ANALYZE operation are run on the database so that it recomputes its optimisations.

Parameters **limit** : int, optional

If not *None*, stop filtering after *limit* clusters have been seen (useful for testing purposes).

Raises **AlreadyFiltered**

If there are already some filtered *Cluster*s or *Quote*s stored in the database (indicating another filtering operation has already been completed, or started and aborted).

brainscopypaste.filter.**filter_quote_offset**()

Get the offset to add to filtered *Quote* ids.

A filtered *Quote*'s id will be its original *Quote*'s id plus this offset. The function is *memoized()* since it is called so often.

Data loading

Load data from various datasets.

This module defines functions and classes to load and parse dataset files. *load_fa_features()* loads Free Association features (using *FAFeatureLoader*) and *load_mt_frequency_and_tokens()* loads MemeTracker features. Both save their computed features to pickle files for later use in analyses. *MemeTrackerParser* parses and loads the whole MemeTracker dataset into the database and is used by *cli*.

class brainscopypaste.load.**FAFeatureLoader**

Bases: *brainscopypaste.load.Parser*

Loader for the Free Association dataset and features.

This class defines a method to load the FA norms (*_norms()*), utility methods to compute the different variants of graphs that can represent the norms (*_norms_graph()*, *_inverse_norms_graph()*, and *_undirected_norms_graph()*) or to help feature computation (*_remove_zeros()*), and public methods that compute features on the FA data (*degree()*, *pagerank()*, *betweenness()*, and *clustering()*). Use a single class instance to compute all FA features.

_inverse_norms_graph

Get the Free Association directed graph with inverted weights.

This graph is useful for computing e.g. *betweenness()*, where link strength should be considered an inverse cost (i.e. a stronger link is easier to cross, instead of harder).

memoized() for performance of the class.

Returns networkx.DiGraph()

The FA inversely weighted directed graph.

_norms

Parse the Free Association Appendix A files into *self.norms*.

After loading, *self.norms* is a dict containing, for each (lowercased) cue, a list of tuples. Each tuple represents a word referenced by the cue, and is in format *(word, ref, weight)*: *word* is the referenced word; *ref* is a boolean indicating if *word* has been normed or not; *weight* is the strength of the referencing.

memoized() for performance of the class.

_norms_graph

Get the Free Association weighted directed graph.

memoized() for performance of the class.

Returns networkx.DiGraph()

The FA weighted directed graph.

classmethod _remove_zeros (*feature*)
Remove key-value pairs where value is zero, in dict *feature*.

Modifies the provided *feature* dict, and does not return anything.

Parameters **feature** : dict

Any association of key-value pairs where values are numbers. Usually a dict of words to feature values.

_undirected_norms_graph
Get the Free Association weighted undirected graph.

When a pair of words is connected in both directions, the undirected link between the two words receives the sum of the two directed link weights. This is used to compute e.g. *clustering()*, which is defined on the undirected (but weighted) FA graph.

memoized() for performance of the class.

Returns networkx.Graph()

The FA weighted undirected graph.

betweenness()
Compute betweenness centrality for words coded by Free Association.

Returns **betweenness** : dict

The association of each word to its betweenness centrality. FA link weights are considered as inverse cost in the computation (i.e. a stronger link is easier to cross). Words with betweenness zero are removed from the dict.

clustering()
Compute clustering coefficient for words coded by Free Association.

Returns **clustering** : dict

The association of each word to its clustering coefficient. FA link weights are taken into account in the computation, but direction of links is ignored (if words are connected in both directions, the link weights are added together). Words with clustering coefficient zero are removed from the dict.

degree()
Compute in-degree centrality for words coded by Free Association.

Returns **degree** : dict

The association of each word to its in-degree. Each incoming link counts as 1 (i.e. link weights are ignored). Words with zero incoming links are removed from the dict.

header_size = 4
Size (in lines) of the header in files to be parsed.

pagerank()
Compute pagerank centrality for words coded by Free Association.

Returns **pagerank** : dict

The association of each word to its pagerank. FA link weights are taken into account in the computation. Words with pagerank zero are removed from the dict.

class brainscopypaste.load.**MemeTrackerParser**(*filename*, *line_count*, *limit=None*)

Bases: *brainscopypaste.load.Parser*

Parse the MemeTracker dataset into the database.

After initialisation, the *parse()* method does all the job. Its internal work is done by the utility methods *_parse()*, *_parse_cluster_block()* and *_parse_line()* (for actual parsing), *_handle_cluster()*, *_handle_quote()* and *_handle_url()* (for parsed data handling), and *_check()* (for consistency checking).

Parameters **filename** : str

Path to the MemeTracker dataset file to parse.

line_count : int

Number of lines in *filename*, to help in showing a progress bar. Should be computed beforehand with e.g. `wc -l <filename>`, so python doesn't need to load the complete file twice.

limit : int, optional

If not *None* (default), stops the parsing once *limit* clusters have been read. Useful for testing purposes.

_check()

Check the consistency of the database with *self._checks*.

The original MemeTracker dataset specifies the number of quotes and frequency for each cluster, and the number of urls and frequency for each quote. This information is saved in *self._checks* during parsing. This method iterates through the whole database of saved *Cluster*s and *Quote*s to check that their counts correspond to what the MemeTracker dataset says (as stored in *self._checks*).

Raises **ValueError**

If any count in the database differs from its specification in *self._checks*.

_handle_cluster(*fields*)

Handle a list of cluster fields to create a new *Cluster*.

The newly created *Cluster* is appended to *self._objects['clusters']*, and corresponding fields are created in *self._checks*.

Parameters **fields** : list of str

List of fields defining the new cluster, as returned by *_parse_line()*.

_handle_quote(*fields*)

Handle a list of quote fields to create a new *Quote*.

The newly created *Quote* is appended to *self._objects['quotes']*, and corresponding fields are created in *self._checks*.

Parameters **fields** : list of str

List of fields defining the new quote, as returned by *_parse_line()*.

_handle_url(*fields*)

Handle a list of url fields to create a new *Url*.

The newly created *Url* is stored on *self._quote* which holds the currently parsed quote.

Parameters **fields** : list of str

List of fields defining the new url, as returned by *_parse_line()*.

_parse()
Do the actual MemeTracker file parsing.

Initialises the parsing tracking variables, then delegates each new cluster block to *_parse_cluster_block()*. Parsed clusters and quotes are stored as *Cluster*s and *Quote*s in *self._objects* (to be saved later in *parse()*). Frequency and url counts for clusters and quotes are saved in *self._checks* for later checking in *parse()*.

_parse_cluster_block()
Parse a block of lines representing a cluster in the source MemeTracker file.

The *Cluster* itself is first created from *self._cluster_line* with *_handle_cluster()*, then each following line is delegated to *_handle_quote()* or *_handle_url()* until exhaustion of this cluster block. During the parsing of this cluster, *self._cluster* holds the current cluster being filled and *self._quote* the current quote (both are cleaned up when the method finishes). At the end of this block, the method increments *self._clusters_read* and sets *self._cluster_line* to the line defining the next cluster, or *None* if the end of file or *self.limit* was reached.

Raises **ValueError**

If *self._cluster_line* is not a line defining a new cluster.

classmethod **_parse_line**(*line*)
Parse *line* to determine if it's a cluster-, quote- or url-line, or anything else.

Parameters **line** : str

A line from the MemeTracker dataset to parse.

Returns **tipe** : str in {'cluster', 'quote', 'url'} or None

The type of object that *line* defines; *None* if unknown or empty line.

fields : list of str

List of the tab-separated fields in *line*.

header_size = 6
Size (in lines) of the header in the MemeTracker file to be parsed.

parse()
Parse the whole MemeTracker file, save, optimise the database, and check for consistency.

Parse the MemeTracker file with *_parse()* to create *Cluster* and *Quote* database entries corresponding to the dataset. The parsed data is then persisted to database in one step (with *save_by_copy()*). The database is then VACUUMed and ANALYZEd (with *execute_raw()*) to force it to recompute its optimisations. Finally, the consistency of the database is checked (with *_check()*) against number of quotes and frequency in each cluster of the original file, and against number of urls and frequency in each quote of the original file. Progress is printed to stdout.

Note that if *self.limit* is not *None*, parsing will stop after *self.limit* clusters have been read.

Once the parsing is finished, *self.parsed* is set to *True*.

Raises **ValueError**

If this instance has already run a parsing.

class brainscopypaste.load.**Parser**
Bases: object

Mixin for file parsers providing the *_skip_header()* method.

Used by *FAFeatureLoader* and *MemeTrackerParser*.

_skip_header()
Skip *self.header_size* lines in the file *self._file*.

brainscopypaste.load.**load_fa_features**()
Load the Free Association dataset and save all its computed features to pickle files.

FA degree, pagerank, betweenness, and clustering are computed using the *FAFeatureLoader* class, and saved respectively to *DEGREE*, *PAGERANK*, *BETWEENNESS* and *CLUSTERING*. Progress is printed to stdout.

brainscopypaste.load.**load_mt_frequency_and_tokens**()
Compute MemeTracker frequency codings and the list of available tokens.

Iterate through the whole MemeTracker dataset loaded into the database to count word frequency and make a list of tokens encountered. Frequency codings are then saved to *FREQUENCY*, and the list of tokens is saved to *TOKENS*. The MemeTracker dataset must have been loaded and filtered previously, or an excetion will be raised (see *Usage* or *cli* for more about that). Progress is printed to stdout.

Substitution mining

Mine substitutions with various mining models.

This module defines several classes and mixins to mine substitutions in the MemeTracker dataset with a series of different models.

Time, *Source*, *Past* and *Durl* together define how a substitution *Model* behaves. *Interval* is a utility class used internally in *Model*. The *ClusterMinerMixin* mixin builds on this definition of a substitution model to provide *ClusterMinerMixin.substitutions()* which iterates over all valid substitutions in a *Cluster*. Finally, *mine_substitutions_with_model()* brings *ClusterMinerMixin* and *SubstitutionValidatorMixin* (which checks for spam substitutions) together to mine for all substitutions in the dataset for a given *Model*.

class brainscopypaste.mine.**ClusterMinerMixin**
Bases: object

Mixin for *Cluster*s that provides substitution mining functionality.

This mixin defines the *substitutions()* method (based on the private *_substitutions()* method) that iterates through all valid substitutions for a given *Model*.

classmethod _substitutions(*source*, *durl*, *model*)
Iterate through all substitutions from *source* to *durl* considered valid by *model*.

This method yields all the substitutions between *source* and *durl* when *model* allows for multiple substitutions.

Parameters **source** : *Quote*

Source for the substitutions.

durl : *Url*

Destination url for the substitutions.

model : *Model*

Model that validates the substitutions between *source* and *durl*.

substitutions(*model*)
Iterate through all substitutions in this cluster considered valid by *model*.

Multiple occurrences of a sentence at the same url (url "frequency") are ignored, so as not to artificially inflate results.

Parameters **model** : *Model*

Model for which to mine substitutions in this cluster.

Yields **substitution** : *Substitution*

All the substitutions in this cluster considered valid by *model*. When *model* allows for multiple substitutions between a quote and a destination url, each substitution is yielded individually. Any substitution yielded is attached to this cluster, so if you use this in a *session_scope()* substitutions will be saved automatically unless you explicitly rollback the session.

class brainscopypaste.mine.**Durl**

Bases: enum.Enum

Type of quotes accepted as substitution destinations.

_member_type_

alias of object

all = <Durl.all: 1>

All quotes are potential destinations for substitutions.

exclude_past = <Durl.exclude_past: 2>

Excluded past rule: only quotes that do not appear in what *Time* and *Past* define as "the past" can be the destination of a substitution.

class brainscopypaste.mine.**Interval**(*start*, *end*)

Bases: object

Time interval defined by *start* and *end* datetimes.

Parameters **start** : :class:datetime.datetime

The interval's start (or left) bound.

end : :class:datetime.datetime

The interval's end (or right) bound.

Raises **Exception**

If *start* is strictly after *end* in time.

Examples

Test if a datetime is in an interval:

```
>>> from datetime import datetime
>>> itv = Interval(datetime(2016, 7, 5, 12, 15, 5),
...                datetime(2016, 7, 9, 13, 30, 0))
>>> datetime(2016, 7, 8) in itv
True
>>> datetime(2016, 8, 1) in itv
False
```

_Interval__key()

Unique identifier for this interval, used to compute e.g. equality between two *Interval* instances.

class brainscopypaste.mine.**Model**(*time*, *source*, *past*, *durl*, *max_distance*)

Bases: object

Substitution mining model.

A mining model is defined by the combination of one parameter for each of *Time*, *Source*, *Past*, *Durl*, and a maximum hamming distance between source string (or substring) and destination string. This class represents such a model. It defines a couple of utility functions used in *ClusterMinerMixin* (*find_start()* and *past_surls()*), and a *validate()* method which determines if a given substitution conforms to the model. Other methods, prefixed with an underscore, are utilities for the methods cited above.

Parameters **time** : *Time*

Type of time defining how occurrence bins of the model are positioned.

source : *Source*

Type of quotes that the model accepts as substitution sources.

past : *Past*

How far back does the model look for substitution sources.

durl : *Durl*

Type of quotes that the model accepts as substitution destinations.

max_distance : int

Maximum number of substitutions between a source string (or substring) and a destination string that the model will detect.

Raises **Exception**

If *max_distance* is more than half of *MT_FILTER_MIN_TOKENS*.

_Model__key()
Unique identifier for this model, used to compute e.g. equality between two *Model* instances.

_distance_start(*source*, *durl*)
Get a *(distance, start)* tuple indicating the minimal distance between *source* and *durl*, and the position of *source*'s substring that achieves that minimum.

This is in fact an alias for what the model considers to be valid transformations and how to define them, but provides proper encapsulation of concerns.

_ok(**args*, ***kwargs*)
Dummy method used when a validation should always pass.

_past(*cluster*, *durl*)
Get an *Interval* representing what this model considers to be the past before *durl*.

See *Time* and *Past* to understand what this interval looks like. This method is *memoized()* for performance.

_validate_base(*source*, *durl*)
Check that *source* has at least one occurrence in what this model considers to be the past before *durl*.

_validate_distance(*source*, *durl*)
Check that *source* and *durl* differ by no more than *self.max_distance*.

_validate_durl(*source*, *durl*)
Check that *durl* is an acceptable substitution destination occurrence for this model.

This method proxies to the proper validation method, depending on the value of *self.durl*.

_validate_durl_exclude_past(*source*, *durl*)
Check that *durl* verifies the excluded past rule.

_validate_source(*source*, *durl*)
Check that *source* is an acceptable substitution source for this model.

This method proxies to the proper validation method, depending on the value of *self.source*.

_validate_source_majority(*source*, *durl*)
Check that *source* verifies the majority rule.

bin_span = datetime.timedelta(1)
Span of occurrence bins the model makes.

drop_caches()
Drop the caches of all *memoized()* methods of the class.

find_start(*source*, *durl*)
Get the position of the substring of *source* that achieves minimal distance to *durl*.

past_surls(*cluster*, *durl*)
Get the list of all *Url*s that are in what this model considers to be the past before *durl*.

This method is *memoized()* for performance.

validate(*source*, *durl*)
Test if potential substitutions from *source* quote to *durl* destination url are valid for this model.

This method is *memoized()* for performance.

Parameters **source** : *Quote*

Candidate source quote for substitutions; the substitutions can be from a substring of *source.string*.

durl : *Url*

Candidate destination url for the substitutions.

Returns bool

True if the proposed source and destination url are considered valid by this model, *False* otherwise.

class brainscopypaste.mine.**Past**
Bases: enum.Enum

How far back in the past can a substitution find its source.

_member_type_
alias of object

all = <Past.all: 1>
The past is everything: substitution sources can be in any bin preceding the destination occurrence (which is an interval that can end at midnight before the destination occurrence when using *Time.discrete*).

last_bin = <Past.last_bin: 2>
The past is the last bin: substitution sources must be in the bin preceding the destination occurrence (which can end at midnight before the destination occurrence when using *Time.discrete*).

class brainscopypaste.mine.**Source**
Bases: enum.Enum

Type of quotes accepted as substitution sources.

_member_type_
alias of object

all = <Source.all: 1>
All quotes are potential sources for substitutions.

majority = <Source.majority: 2>
Majority rule: only quotes that are the most frequent in the considered past bin can be the source of substitutions (note that several quotes in a single bin can have the same maximal frequency).

class brainscopypaste.mine.**SubstitutionValidatorMixin**
Bases: object

Mixin for *Substitution* that adds validation functionality.

A non-negligible part of the substitutions found by *ClusterMinerMixin* are spam or changes we're not interested in: minor spelling changes, abbreviations, changes of articles, symptoms of a deleted word that appear as substitutions, etc. This class defines the *validate()* method, which tests for all these cases and returns whether or not the substitution is worth keeping.

validate()
Check whether or not this substitution is worth keeping.

class brainscopypaste.mine.**Time**
Bases: enum.Enum

Type of time that determines the positioning of occurrence bins.

_member_type_
alias of object

continuous = <Time.continuous: 1>
Continuous time: bins are sliding, end at the destination occurrence, and start *Model.bin_span* before that.

discrete = <Time.discrete: 2>
Discrete time: bins are aligned at midnight, end at or before the destination occurrence, and start *Model.bin_span* before that.

brainscopypaste.mine.**_get_wordnet_words**()
Get the set of all words known by WordNet.

This is the set of all lemma names for all synonym sets in WordNet.

brainscopypaste.mine.**mine_substitutions_with_model**(*model*, *limit=None*)
Mine all substitutions in the MemeTracker dataset conforming to *model*.

Iterates through the whole MemeTracker dataset to find all substitutions that are considered valid by *model*, and save the results to the database. The MemeTracker dataset must have been loaded and filtered previously, or an excetion will be raised (see *Usage* or *cli* for more about that). Mined substitutions are saved each time the function moves to a new cluster, and progress is printed to stdout. The number of substitutions seen and the number of substitutions kept (i.e. validated by *SubstitutionValidatorMixin.validate()*) are also printed to stdout.

Parameters **model** : *Model*

The substitution model to use for mining.

limit : int, optional

If not *None* (default), mining will stop after *limit* clusters have been examined.

Raises **Exception**

If no filtered clusters are found in the database, or if there already are some substitutions from model *model* in the database.

Tagger

Utilities

Miscellaneous utilities.

class brainscopypaste.utils.**Namespace**(*init_dict*)
Bases: object

Convert a dict to a namespace by creating a class out of it.

Parameters **init_dict** : dict

The dict you wish to turn into a namespace.

exception brainscopypaste.utils.**NotFoundError**
Bases: Exception

Signal a file or directory can't be found.

class brainscopypaste.utils.**Stopwords**
Bases: object

Detect if a word is a stopword.

Prefer using this module's *stopwords* instance of this class for stopword-checking.

_load()
Read and load the underlying stopwords file.

class brainscopypaste.utils.**cache**(*method*, *name=None*)
Bases: object

Compute an attribute's value and cache it in the instance.

This is meant to be used as a decorator on class methods, to turn them into cached computed attributes: the value is computed the first time you access the attribute, and this decorator then replaces the method with the computed value. Any subsequent access gives you the cached value immediately.

Taken from the Python Cookbook (Denis Otkidach).

brainscopypaste.utils.**execute_raw**(*engine*, *statement*)
Execute the raw SQL statement *statement* on SQLAlchemy engine *engine*.

Useful to run ANALYZE or VACUUM operations on the database.

Parameters **engine** : sqlalchemy.engine.Engine

The engine to run *statement* on.

statement : str

A valid SQL statement for *engine*.

brainscopypaste.utils.**find_parent_rel_dir**(*rel_dir*)
Find a relative directory in parent directories.

Searches for directory *rel_dir* in all parent directories of the current directory.

Parameters **rel_dir** : string

The relative directory to search for.

Returns **d** : string

Full path to the first found directory.

Raises **NotFoundError**

If no relative directory is found in the parent directories.

brainscopypaste.utils.**grouper**(*iterable*, *n*, *fillvalue=None*)
Iterate over *n*-wide slices of *iterable*, filling the last slice with *fillvalue*.

See *grouper_adaptive()* for a version of this that doesn't fill the last slice.

brainscopypaste.utils.**grouper_adaptive**(*iterable*, *n*)
Iterate over *n*-wide slices of *iterable*, ending the last slice once *iterable* is empty.

See *grouper_adaptive()* for a version of this that fills the last slice with a value of your choosing.

brainscopypaste.utils.**hamming**(*s1*, *s2*)
Compute the hamming distance between strings or lists *s1* and *s2*.

brainscopypaste.utils.**init_db**(*echo_sql=False*)
Connect to the database and bind *db*'s *Session* object to it.

Uses the *DB_USER* and *DB_PASSWORD* credentials to connect to PostgreSQL database *DB_NAME*. It binds the *Session* object in *db* to this engine, and returns the engine object. Note that once this is done, you can directly use *session_scope()* since it uses the right *Session* object.

Parameters **echo_sql** : bool, optional

If *True*, print to stdout all SQL commands sent to the engine; defaults to *False*.

Returns sqlalchemy.engine.Engine

The engine connected to the database.

brainscopypaste.utils.**is_int**(*s*)
Test if *s* is a string that represents an integer; returns *True* if so, *False* in any other case.

brainscopypaste.utils.**is_same_ending_us_uk_spelling**(*w1*, *w2*)
Test if *w1* and *w2* differ by only the last two letters inverted, as in *center*/*centre* (words must be at least 4 letters).

brainscopypaste.utils.**iter_parent_dirs**(*rel_dir*)
Iterate through parent directories of current working directory, appending *rel_dir* to those successive directories.

brainscopypaste.utils.**langdetect**(*sentence*)
Detect the language of *sentence*.

brainscopypaste.utils.**levenshtein**(*s1*, *s2*)
Compute the levenshtein distance between strings or lists *s1* and *s2*.

brainscopypaste.utils.**memoized**(*f*)
Decorate a function to cache its return value the first time it is called.

If called later with the same arguments, the cached value is returned (not reevaluated).

brainscopypaste.utils.**mkdirp**(*folder*)
Create *folder* if it doesn't exist.

brainscopypaste.utils.**mpl_palette**(*n_colors*, *variation='Set2'*)
Get any seaborn palette as a usable matplotlib colormap.

brainscopypaste.utils.**session_scope**()
Provide an SQLAlchemy transactional scope around a series of operations.

Wrap your SQLAlchemy operations (queries, insertions, modifications, etc.) in a `with session_scope() as session` block to deal with sessions easily. Changes are committed when the block finishes. If an exception occurrs in the block, the session is rolled back and the exception propagated.

brainscopypaste.utils.**stopwords = <brainscopypaste.utils.Stopwords object>**
Instance of *Stopwords* to be used for stopword-testing.

brainscopypaste.utils.**subhamming**(*s1*, *s2*)
Compute the minimum hamming distance between *s2* and all sublists of *s1* as long as *s2*, returning *(distance, sublist start in s1)*.

brainscopypaste.utils.**sublists**(*s*, *l*)
Get all sublists of *s* of length *l*.

brainscopypaste.utils.**unpickle**(*filename*)
Load a pickle file at path *filename*.

This function is *memoized()* so a file is only loaded the first time.

Settings

Settings for the whole analysis are defined in the *brainscopypaste.settings* module, and should be accessed through the *brainscopypaste.conf* module as explained below.

Accessing settings: *brainscopypaste.conf*

Manage settings from the *settings* module, allowing overriding of some values.

Use the *settings* class instance from this module to access settings from any other module: `from brainscopypaste.conf import settings`. Note that only uppercase variables from the *settings* module are taken into account, the rest is ignored.

class brainscopypaste.conf.**Settings**
Bases: object

Hold all settings for the analysis, managing and proxying access to the *settings* module.

Only uppercase variables from the *settings* module are taken into account, the rest is ignored. This class also lets you override values with a context manager to make testing easier. See the *override()* and *file_override()* methods for more details.

Use the *settings* instance of this class to access a singleton version of the settings for the whole analysis. Overridden values then appear overridden to all other modules (i.e. for all accesses) until the context manager is closed.

_override(*name*, *value*)
Override *name* with *value*, after some checks.

The method checks that *name* is an uppercase string, and that it exists in the known settings. Use this when writing a context manager that wraps the operation in try/finally blocks, then restores the default behaviour.

Parameters **name** : str

Uppercase string denoting a known setting to be overridden.

value : object

Value to replace the setting with.

Raises **ValueError**

If *name* is not an uppercase string or is not a known setting name.

_setup()
Set uppercase variables from the *settings* module as attributes on this instance.

file_override(**names*)
Context manager that overrides a file setting by pointing it to an empty temporary file for the duration of the context.

Some values in the *settings* module are file paths, and you might want to easily override the *contents* of that file for a block of code. This method lets you do just that: it will create a temporary file for a setting you wish to override, point that setting to the new empty file, and clean up once the context closes. This is a shortcut for *override()* when working on files whose contents you want to override.

Parameters **names** : list of str

List of setting names you want to override with temporary files.

Raises **ValueError**

If any member of *names* is not an uppercase string or is not a known setting name.

See also:

override

Examples

Override the Age-of-Acquisition source file to e.g. test code that imports it as a word feature:

```
>>> from brainscopypaste.conf import settings
>>> with settings.file_override('AOA'):
...     with open(settings.AOA, 'w') as aoa:
...         # Write test content to the temporary AOA file.
...     # Test your code on the temporary AOA content.
>>> # `settings.AOA` is back to default here.
```

override(**names_values*)
Context manager that overrides setting values for the duration of the context.

Use this method to override one or several setting values for a block of code, then have those settings go back to their default value. Very useful when writing tests.

Parameters **names_values** : list of tuples

List of *(name, value)* tuples defining which settings to override with what value. Setting names must already exist (you can't use this to create a new entry).

Raises **ValueError**

If any of the *name* values in *names_values* is not an uppercase string or is not a known setting name.

See also:

file_override

Examples

Override MemeTracker filter settings for the duration of a test:

```
>>> from brainscopypaste.conf import settings
>>> with settings.override(('MT_FILTER_MIN_TOKENS', 2),
...                        ('MT_FILTER_MAX_DAYS, 50)):
...     # Here: some test code using the overridden settings.
>>> # `settings` is back to default here.
```

brainscopypaste.conf.**settings = <brainscopypaste.conf.Settings object>**
Instance of the *Settings* class that should be used to access settings. See that class's documentation for more information.

Defining settings: *brainscopypaste.settings*

Definition of the overall settings for the analysis.

Edit this module to permanently change settings for the analysis. Do NOT directly import this module if you want to access these settings from inside some other code; to do so see *conf.settings* (which also lets you temporarily override settings).

All uppercase variables defined in this module are considered settings, the rest is ignored.

See Also

brainscopypaste.conf.Settings

brainscopypaste.settings.**AOA = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkouts/latest/**
Path to the file containing word age of acquisition data.

brainscopypaste.settings.**BETWEENNESS = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/check**
Path to the pickle file containing word betweeness centrality values.

brainscopypaste.settings.**CLEARPOND = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkou**
Path to the file containing word neighbourhood density data.

brainscopypaste.settings.**CLUSTERING = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkc**
Path to the pickle file containing word clustering coefficient values.

brainscopypaste.settings.**DB_NAME = 'brainscopypaste'**
Name of the PostgreSQL database used to store analysis data.

brainscopypaste.settings.**DB_NAME_TEST = 'brainscopypaste_test'**
Name of the PostgreSQL database used to store test data.

brainscopypaste.settings.**DB_PASSWORD = ''**
PostgreSQL connection user password.

brainscopypaste.settings.**DB_USER = 'brainscopypaste'**
PostgreSQL connection user name.

brainscopypaste.settings.**DEGREE = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkouts/la**
Path to the pickle file containing word degree centrality values.

brainscopypaste.settings.**FA_SOURCES = ['/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/check**
List of files making up the Free Association data.

brainscopypaste.settings.**FIGURE = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkouts/la**
Template for the file path to a figure from the main analysis that is to be saved.

brainscopypaste.settings.**FIGURE_VARIANTS = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste**
Template for the folder containing all the figures of a notebook variant with a specific substitution-detection model.

brainscopypaste.settings.**FREQUENCY = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkou**
Path to the pickle file containing word frequency values.

brainscopypaste.settings.**MT_FILTER_MAX_DAYS = 80**
Maximum number of days a quote or a cluster can span to be kept by the MemeTracker filter.

brainscopypaste.settings.**MT_FILTER_MIN_TOKENS = 5**
Minimum number of tokens a quote must have to be kept by the MemeTracker filter.

brainscopypaste.settings.**MT_LENGTH = 8357595**
Number of lines in the *MT_SOURCE* file (pre-computed with `wc -l <memetracker-file>`); used by *MemeTrackerParser*.

brainscopypaste.settings.**MT_SOURCE = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkou**
Path to the source MemeTracker data set.

brainscopypaste.settings.**NOTEBOOK = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkout**
Template for the file path to a notebook variant with a specific substitution-detection model.

brainscopypaste.settings.**PAGERANK = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkout**
Path to the pickle file containing word pagerank centrality values.

brainscopypaste.settings.**STOPWORDS = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkou**
Path to the file containing the list of stopwords.

brainscopypaste.settings.**TOKENS = '/home/docs/checkouts/readthedocs.org/user_builds/brainscopypaste/checkouts/la**
Path to the pickle file containing the list of known tokens.

brainscopypaste.settings.**TREETAGGER_TAGDIR = 'treetagger'**
TreeTagger library folder.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

b

Symbols

A

B

C

D

Q

S

T

U

V