One year of Menéame

Ferdinando Papale (s121035), Albert Fernandez de la Peña (s112213), Jose Luis Bellod Cisneros (s101605)

***Abstract*—Menéame[1] is a Spanish social news website based on community partecipation, where users can submit content discovered on the Internet, vote and comment other people submissions. In our project we intend to investigate different aspects of the users and the stories shared in 1 year.**

I. Introduction

This report presents the analysis carried over a dataset of 12.000 news send by their users to Menéame during the last year. Moreover, we present the architecture and the implementation of the complete python project that has been coded to obtain the news, clean them, analyse them and present them using different visualizations in a website. The source code of the project can be found in github[2].

II. Overview of the Analysis

The development of the project can be divided in different phases.

1) Information regarding 1 year of shared content of Menéame has been scraped from the website and inserted in a CouchDB database. The year is comprised between September 2012 and September 2013;
2) The saved content has been cleaned and prepared for the subsequent analysis;
3) Independent analysis have been conducted on the dataset, considering three main fields: analysis of the network of users, topic analysis of the content and sentiment analysis of the content;
4) The results are presented in a website.[3]

In order to improve the separation of concerns and reusability, the code has been structured in packages and modules, that contain the different functions used during the analysis. These modules are then imported in the main files to perform different actions. In the appendix, it is shown an overview of the program design, showing both the modules and the scripts that use them.

After the initial scraping and cleaning, the different analysis have been conducted on the dataset, saved in a CouchDB[4] database. We have decided to use this database as its NoSQL nature better fits to the nature of the data. Besides, its document based model allows an easier management and manipulation of the information.

III. Implementation

In this section we will overview the different parts of the implementation.

A. Scraping

The initial step of the project was to obtain a properly formatted dataset containing the news of Menéame. For this purpose, a script able to scrap parametrized news was necessary.

Moreover, the whole project has been conceived as a general tool for analyzing any kind of social news website. In such websites, there is usually a page where the news are listed, and then another page where users comment on a specific new. For this purpose, a scraper interface, the `Scraper` class has been coded, as well as the general methods needed for any scraper, and the methods specific to Menéame (or Reddit), are left empty for implementation. In our case, we also have coded the `MeneameScraper`, which inherits from `Scraper` and implements the methods `extract_news` and `extract_comments`, needed for extracting the news and the related comments. The interface and the relative classes are contained in the module `scrapers.scraper`

For handling the HTTP requests, the python library `requests`[5] has been used, as well as `BeautifulSoup`[6] and `feedparser`[7] for the parsing of the news and the comments.

Furthermore, a CLI tool has been coded as an interface to the previously described classes. The tool, found in `scraper.py`, reads the following parameters from the command line arguments and stores the scraped content in the specified directory location in a JSON format:

- output directory of the scraped news and comments.
- start page (numeric value).
- timeout (time between two requests, default 0 seconds).
- range (range of news to scrap: one day, one week, etc.)

The argument parsing has been done using `docopt`[8], as it allows a human and nice way to define the argument parameters and default values, as well as document them.

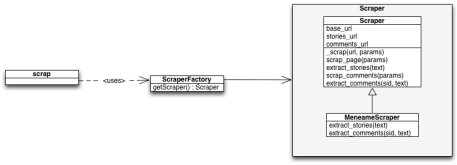


Figure 1. UML diagram of the scraper.

Once the news have been downloaded and stored in a directory, the script `to_database.py`, would read the specified directory and store them to a given (by parameter) database location.

[1] http://www.meneame.net/
[2] https://github.com/albertfdp/dtu-data-mining
[3] Meneapp: http://meneapp.appspot.com
[4] CouchDBhttp://couchdb.apache.org/
[5] Requests www.crummy.com/software/BeautifulSoup/bs4/doc/
[6] BeautifulSoup www.crummy.com/software/BeautifulSoup/bs4/doc/
[7] feedparser https://pypi.python.org/pypi/feedparser
[8] docopt: https://github.com/docopt/docopt

B. Cleaning

The cleaning script, `cleaner.py`, has been used to prepare the data for the subsequent analysis. In particular the aim of the script is two-fold:

- Update the database with a cleaned version of the content of the articles comments, with all the html tags stripped;
- Update the database with a dictionary of commenters for each article. In particular, the keys of the dictionary will be commenters, while the values will be the number of comments.

Those two operations are necessary for the subsequent analysis and are supported by the functions contained in the module `scrapers.cleaner`.

C. Sentiment Analysis

This section describes the implementation of the sentiment analysis module, and how it has been used to conduct a sentiment analysis of the news and the comments.

As a Spanish site, the content is mostly in Spanish, although it is not unusual to see news or comments in English. Therefore, we have used the Spanish ANEW [1] to compute the average valence and arousal of the description and of the comments of a specific new.

We have divided the code into two separate scripts, to make it general and reusable for different implementations. Moreover, we also have built `sentiment.py` script as a CLI tool, which listens for the following user input:

- database: the couchdb database where the news are stored.
- ANEW file: the CSV file where to read the valences and arousals for each word.

We use then `sentiment.py` to read the user input, construct a dictionary by mapping word with their valence and arousal, and iterate through all the news in the database and computing their sentiment using the module `sentiments.sentiments`. After the sentiment of a new and its comment has been computed, it is updated in the database.

The module, has a single function `get_sentiment`, which receives the dictionary and the text to be analized. Then, it maps each word to its corresponding valence and arousal, and computes the average for the given text.

In order to properly analyze the text, it has been previously tokenized using `nltk`, as well as stemmed using SnowballStemmer of the same library.

D. Topic Analysis

For the Topic Analysis, the tool we have selected is GenSim[9], mainly developed by Radim Rehurek. It deals with topic modeling with implementations of Latent Dirichlet Allocation (LDA), Latent Semantic Analysis (LSA), Term Frequency-Inverse Document Frequency (TF-IDF), Random Projections and Hierarchical Dirichlet Process. We have used LDA, because we wanted to check how well the method performed and whether the topics were accurate or not. According to GenSim's documentation: "Unlike LSA, the topics coming from LDA are easier to interpret". Therefore, this was our choice for the news.

The source for our topic analysis is the description of each article. Our first attempt to extract topics used all the news as the input corpus. After checking the results of LDA, we decided to reduce the size of the corpus for the training, because it was difficult to interpret the inferred topics, probably due to the high number of documents and the short length of the text analyzed for each new.

Prior to select the tokens that will be used as input for the model, we have to divide the description of the text, first into sentence and then into tokens. We modifided the Penn TreeBankWord [10] sentence tokenizer from the `nltk` python package to divide spanish sentence of a given text (mainly we add the possibility of sentence to start with the symbols "¡" and "¿", for exclamation and interrogation)

For the purpose of reducing the corpus, we decided to divide it into slices of 10 days. We grouped all the news that occurred in a 10-day time window as input for LDA. After the initial test with the topics, we decided to use named entities and names, as the tokens for the analysis. We discovered that using all the words from the text produced noisy topics difficult to interpret. Named entities and names were chosen because this type of words are very easy to interpret, and to relate with things that happened in the past.

This approach needed to use, first a Part-of-Speech tagger, to assign part-of-speech tags (verbs, names, determiners...) to each word, and then a Chunker to tag a set of words with a named-entity tag like "PERSON". For example, "The President of Spain" could be tagged as "PERSON". We used the code provided in the book "Natural Language Processing with Python" [2] for the Part-of-Speech tagger (chapter 5)[11] and the Chunker (chapter 7)[12], with some modifications to adapt it to the spanish language. In the function `tags_since_dt()` found in the module `train_CHUNK_tagger`, we modifided the determiner from where we check the named entities, from "DT" to "DA", according to the corpus used for the training). For the training we used the CoNLL-2002[13] corpus, both for tagging and chunking. Both the code for the chunker and the tagger can be found in `topics.train_chunk_tagger` and `topics.train_pos_tagger` respectively.

We discovered that if the number of news for the LDA model was small or the news had very similar words, the inferred topics were very similar to each other. This was due the fact that we had a fixed number of topics (35) per slice (Meneame provides 35 categories to each new). To prevent this, our approach was to perform a similarity check between all the topics in a time slice, usign Hellinger distance[14]. Once we have similarity score of each topic, we group them together if the score between two topics is higher than a threshold (0.7).

[9] http://radimrehurek.com/gensim/

[10] http://nltk.org/_modules/nltk/tokenize/treebank.html#TreebankWordTokenizer

[11] http://nltk.org/book/ch05.html

[12] http://nltk.org/book/ch07.html

[13] http://www.clips.ua.ac.be/conll2002/ner/data/

[14] http://en.wikipedia.org/wiki/Hellinger_distance

The module `topics.topic_analysis.py` contains the code necessary to query the database to get the news, divide the corpus into slices of 10 days, infer the topics to the news, and update the database assigning the most relevant topic to each new.

The script `topic.py` has been implemented as a CLI tool so the user can select the size of the time slices, the number of topics, the database and the tagger and the chunker.

Finally, once we have all the topics found for each time slice, we store them in the database, ready for the other scripts to use and for the ulterior visualization in the web page using the script `topic_assign.py`

E. Network Analysis

In order to conduct the network analysis we have used the `igraph`[15] module, assisted by `matplotlib` for visualization.

We have decided to use the `igraph` library, over a more pure python module such as `networkx`, because its core functionalities are implemented in C/C++, allowing greater performances[16], of significant importance due to the size of our network.

The disadvantage of this module is that it is not as direct to use as `networkx`, because both the vertices and the edges are indexed by integers, implying a more oblique access to the elements of the graph. Another peculiar aspect is that the edge list is re-indexed every time a edge is inserted[17], thus this library is not well suited for graphs that have to be modified often but, fortunately, this is not our case.

For the reasons presented above, to create the graph we first create the data structures for the vertexes, the edges and the necessary attributes, and only then create an `igraph.Graph` object.

The creation and the analysis of the network are contained respectively, in the scripts `network_creation.py` and `network_analysis.py`, that use the functions contained in the module `network.network`

In order to conduct the analysis, we defined the network of users of meneame as follows:

- Each vertex is a user that has commented in one of the articles we analyzed. Each vertex has an attribute, "comments", that represents the total number of comments an user has posted
- There is an edge between two users if they posted in the same article. Each edge has a weight representing the number of articles in which they posted together.

After the network has been created, it is pickled for subsequent analysis.

F. Web serving

In order to visualize the results of the different parts of this project, a web server has been implemented.[18].

[15]http://igraph.sourceforge.net/index.html
[16]http://graph-tool.skewed.de/performance
[17]http://stackoverflow.com/questions/13974279/igraph-why-is-add-edge-function-so-slow-ompared-to-add-edges
[18]Meneapp: http://meneapp.appspot.com

The server is hosted on the Google App Engine technology, and implemented using the python-based GAE framework.

The following technologies have been used for serving the websites aside from the included already with Google App Engine:

- `Bootstrap`[19]: for the website' CSS.
- `Jinja2` [20]: python library which provides a friendly template language for generating the HTML.
- `D3` [21]: javascript library used for generating the visualizations.
- `Crossfilter` [22]: javascript library used for crossfiltering and brunching the different charts.

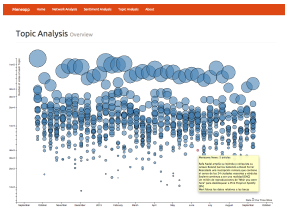


Figure 2. Topic page in the web site.

IV. Documentation

Our project has been thoroughly documented, both at a module and at a function level, in the form of docstrings. In order to create an accessible documentation we have used `sphinx`[23], a tool to create easy-to-read python projects documentation. An html version of the documentation has been included in the website.

V. Testing

As our project is script-oriented, the necessity of testing our code is not as high as in a full-fledged python library or program. Besides, large parts of our analysis rely on external libraries, that we suppose have been thoroughly tested. Nevertheless, we have decided to use the `nose` module [24], an useful extension to unittest, to test several parts of our projects. The modules which serve the basic functionality and produce the more important computations have been tested.

VI. Results

In this section we will outline some of the results that we have obtained with our analysis.

A. Sentiment Analysis

The sentiment analysis has shown some interesting results on the scraped data. The results and visualizations of them are shown in the website[25].

[19]Bootstrap: http://getbootstrap.com/
[20]Jinja2 http://jinja.pocoo.org/docs/
[21]D3 d3js.org/
[22]Crossfilter d3js.org/
[23]http://sphinx-doc.org/index.html
[24]http://nose.readthedocs.org/en/latest/
[25]http://meneapp.appspot.com/sentiment

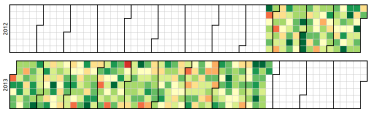


Figure 3. Sentiment by day

In order to properly visualize the sentiment along the time, the average valence and arousal have been computed by date, as any other time ranges would make it too flat to see anything interesting.

We have seen that the news and their comments have a higher valence and arousal during the winter months and lower values during the summer, especially during July. Moreover, we also have observed that the weekends tend to have lower average values of them too.

B. Network Analysis

The network analysis provided some interesting results, with the meneame network containing a total of 19823 users and about 9 milions connections between them included in one single connected component.

In order to see if the six-degrees of separation theory hold in this network, we computed the shortest path length between any two users in the network, and we have found a maximum length of only 3, and an average length of 1.98. This confirms the theory and implies all the users in the network very connected. Not surprisingly, we have found also a very high clustering, with an average global clustering coefficient of 0.36.

We have also investigated wether the network can be considered scale-free. To do so we plotted its degree distribution, that can be seen in figure 5. As we can observe, the degree distribution has a peculiar shape, due to the way in which we defined edges. Infact, the minimum observable degree is equal to the minimum number of comments posted in an article, in our case 5. The degree distribution does not strictly follow a power-law, thus we cannot clearly state whether this can be considered a scale-free network or not.

Another interesting aspect that we have found is the distribution of the weights of the edges, that can be seen in figure 4. In this case the distribution undoubtely follows a power law, with the most edges having a low value of weight, and a few edges having a value higher of 400.

We also tested for communities inside the network using the infomap algorithm [3] by Rosvall and Bergstrom that appears to be one of the best performing [4]. We found only one community, but this is not surprising, giving the high clustering level.

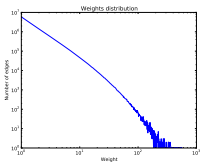


Figure 4. Edges weights distribution of the meneame network.

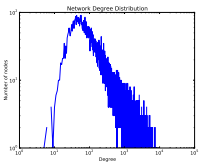


Figure 5. Degree distribution of the meneame network

C. Topic Analysis

The main results for the Topic Analysis can be seen in figure 2. The main reason of using this visualiztion is to check easily wether the topics found with LDA make sense or not. Inspecting the topics (and the news associated with them) we can conclude that some of the topics are very noise, meaing that the news belonging to some of them are from different topics (politics, sports...). On the other hand, some of the topics with the highest number of votes (and articles associated) were very easy to identify.

VII. Development Process

The devlopment of the project has been straightforward. Once the dataset has been obtained with scraping and cleaned, the database has been copied by each of the member of the group and used for the different analysis.

The code has been developed in collaboration between the different group members, using `git` as version control system and github[26] as hosting platform. The code has been made public as well.

Overall, we have decided to follow the official guidelines for code style and, thus, we have checked our code using both `pep8` and `pylint`. Nevertheless, especially when dealing with `pylint`, we did not strictly followed the given comments, and decided to use common sense. Additionally, `pylint`, as a static tool, cannot recognize the presence of dynamically set attributes in classes and, thus, will give unnecessary errors in some cases.

VIII. Future work

There are different way in which our work can be continued and expanded. For instance, different scrapers can be implemented for different social news website, such as Reddit. For the Topic analysis, one solution to improve the qualit of the topics could be to use, not the description, but the content of the article mentioned in the new. For this purpose, some services like Readability [27] allow you to retrieve just the content of a webpage. Testing other topic analysis methods like LSA could give us an idea of which method is best suited for the analysis we performed.

References

[1] J. Redondo, I. Fraga, I. Padrón, and M. Comesaña, "The spanish adaptation of anew (affective norms for english words)," *Behavior research methods*, vol. 39, no. 3, pp. 600–605, 2007.

[2] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. Beijing: O'Reilly, 2009. [Online]. Available: http://www.nltk.org/book

[3] M. Rosvall and C. T. Bergstrom, "Maps of random walks on complex networks reveal community structure," *Proc. Natl. Acad. Sci. USA*, p. 1118, 2008.

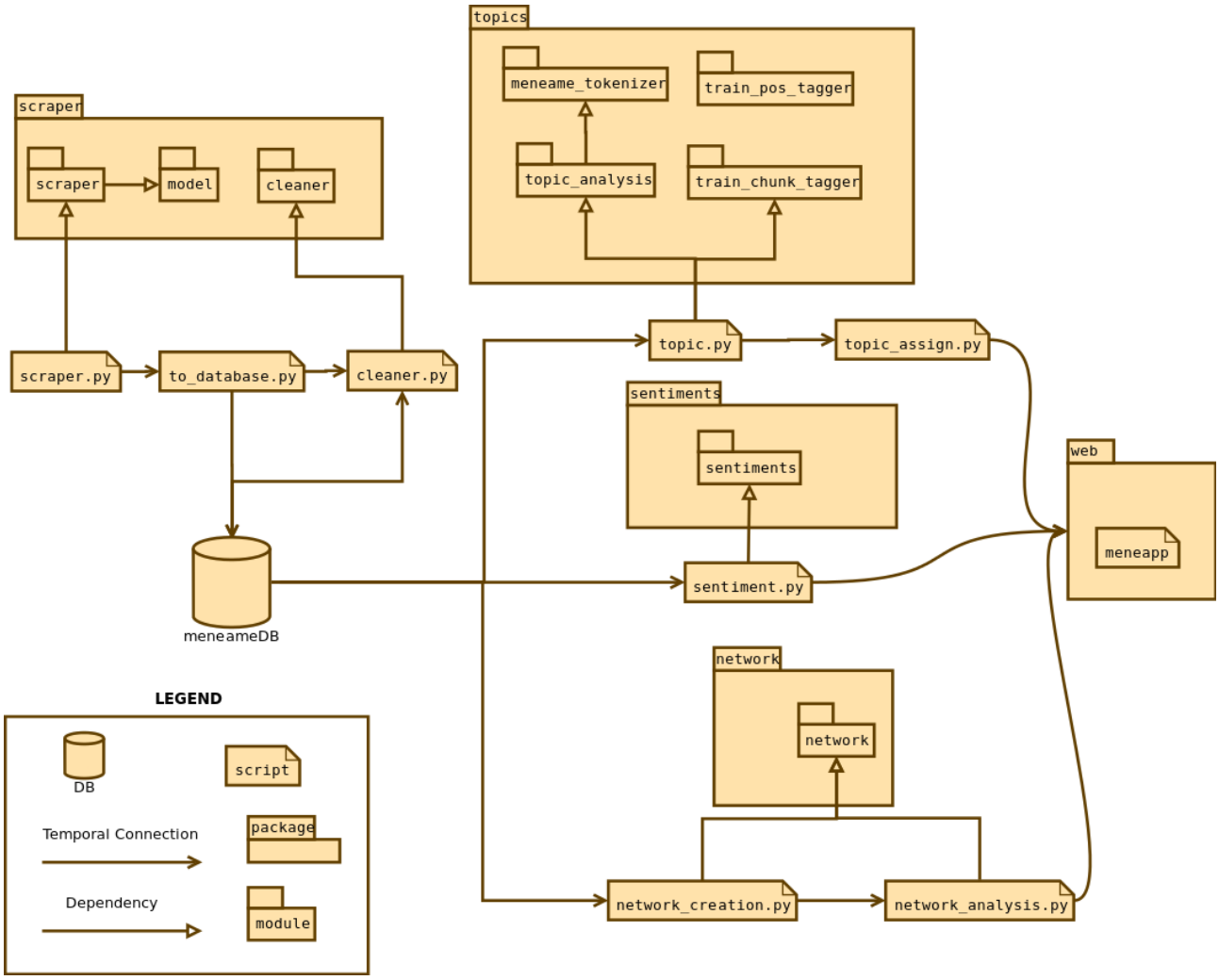
[4] S. Fortunato and A. Lancichinetti, "Community detection algorithms: A comparative analysis: Invited presentation, extended abstract," in *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. ICST, Brussels, Belgium, Belgium: ICST, 2009, pp. 27:1–27:2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1698822.1698858

[26] https://github.com/albertfdp/dtu-data-mining

[27] http://www.readability.com/

APPENDIX

Diagram of the implementation design



```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
This script aim is two-fold:

* Update the database with a cleaned version of the content of the
    articles comments, with all the html tags stripped;
* Update the database with a dictionary of commenters for each article.
    In particular, the keys of the dictionary will be commenters, while the
    values will be the number of comments.

Those two operations are necessary for the subsequent analysis.
"""
from scraper.cleaner import strip_html_tags
import couchdb
import logging
import sys


def main():
    """Main function. """
    logging.basicConfig(filename='cleaner.log',
                        level=logging.DEBUG, filemode='w')
    logging.basicConfig(format='%(asctime)s %(message)s')

    #Connection with the database
    try:
        couch = couchdb.Server()
        news_db = couch['meneame']
    except couchdb.ResourceNotFound, err:
        logging.exception("Connection error with the database! \n")
        logging.exception(err)
        sys.exit(1)

    logging.info('Connection to the databases estabilished')

    for element in news_db:

        logging.info('Considering article ' + element)
        news = dict(news_db.get(element))

        comments = news['comments']
        commenters = dict()

        for comment in comments:
            user = comment['user']

            #Adding to the commenters list
            commenters.setdefault(user, 0)
            commenters[user] = commenters[user] + 1

            #Adding the cleaned version of the comment
            cleaned_comment = strip_html_tags(comment['summary'])
            comment['cleaned_summary'] = cleaned_comment

        news['commenters'] = commenters
        news_db.save(news)


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

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
The aim of the script is to train a Part of Speech (PoS) tagger.

* The train set for the PoS tagger
* For the purpose of the training an external maximun entropy model (megam)\
    is used.

After the PoS tagger has been created, it is pickled for subsequent use.
"""

import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import logging
import pickle


# define logging configuration
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(levelname)-8s %(message)s')


def main():
    """ Main function. """
    # Regular expression used as a backoff tagger
    regex = nltk.RegexpTagger
    (
        [
            (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),
            (
                r'('+'|'.join(stopwords.words('spanish')) + ')$', 'STOP'
            ),
            (
                r'(?:www\.|(?!www))[^\s\.]+\.[^\s]{2,}|www\.[^\s]+\.[^\s]{2,}',
                'URL'
            ),
            (r'[0-9]+/[0-9]+/[0-9]+', 'DATE'),
            (r'([^A-Za-z0-9])+', 'PUNCT'),
            (r'\xbf', 'Faa'),
            (r'\xa1', 'Fat'),
            (r'.*', 'N_N')  # weird tokens (default)
        ]
    )

    # Create training set from the Conll2002 Spanish corpus
    train_set = []
    for text in nltk.corpus.conll2002.tagged_sents('esp.train'):
        train_set.append([(word.lower(), tag) for word, tag in text])

    logging.info('Training Unigram Tagger...')
    unigram_tagger = nltk.UnigramTagger(train_set, backoff=regex)
    logging.info('Training Bigram Tagger...')
    tagger_da = nltk.BigramTagger(train_set, backoff=unigram_tagger)

    logging.info('Pickling Part of Speech Tagger...')
    pickle.dump(tagger_da, open("tmp/pos_tagger.p", "wb"))


def test_pos_tagger():
    """
        Test PoS tagger.
    """

    pos_tag = pickle.load(open("topics/tmp/pos_tagger.p", "rb"))
    string = "El presidente del congreso"
    tokens = [token.lower() for token in word_tokenize(string)]
```

```
    result = pos_tag.tag(tokens)

    assert result == [('el', 'DA'), ("presidente", 'NC'), ('del', 'SP'),
                      ('congreso', 'NC')]


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

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""The aim of this package is to deal with topic analysis of the meneame\
news database.
"""
```

```
# -*- coding: utf-8 -*-

# Natural Language Toolkit: Tokenizers
#
# Copyright (C) 2001-2013 NLTK Project
# Author: Edward Loper <edloper@gradient.cis.upenn.edu>
#         Michael Heilman <mheilman@cmu.edu> \
#\ (re-port from http://www.cis.upenn.edu/~treebank/tokenizer.sed)
#
# URL: <http://nltk.sourceforge.net>
# For license information, see LICENSE.TXT
#
# Modifications in lines 84 and 85 added by Jose Luis Bellod Cisneros

r"""

Penn Treebank Tokenizer

The Treebank tokenizer uses regular expressions to tokenize text as in Penn\
Treebank.
This implementation is a port of the tokenizer sed script written by Robert\
McIntyre
and available at http://www.cis.upenn.edu/~treebank/tokenizer.sed.
"""

import re
from nltk.tokenize.api import TokenizerI



class TreebankWordTokenizer(TokenizerI):
    """
    The Treebank tokenizer uses regular expressions to tokenize text as in \
    Penn Treebank.
    This is the method that is invoked by ``word_tokenize()``.  It assumes \
    that the
    text has already been segmented into sentences, e.g. using \
    ``sent_tokenize()``.

    This tokenizer performs the following steps:

    - split standard contractions, e.g. ``don't`` -> ``do n't``
    - treat most punctuation characters as separate tokens
    - split off commas and single quotes, when followed by whitespace
    - separate periods that appear at the end of line

        >>> from topics.meneame_tokenizer import TreebankWordTokenizer
        >>> s = '''Good muffins cost $3.88\\nin New York.Please buy me\\ntwo'''
        >>> TreebankWordTokenizer().tokenize(s)
        ['Good', 'muffins', 'cost', '$', '3.88', 'in', 'New', 'York.',
        'Please', 'buy', 'me', 'two']
        >>> s = "They'll save and invest more."
        >>> TreebankWordTokenizer().tokenize(s)
        ['They', "'ll", 'save', 'and', 'invest', 'more', '.']
    """

    # List of contractions adapted from Robert MacIntyre's tokenizer.
    CONTRACTIONS2 = [re.compile(r"(?i)\b(can)(not)\b"),
                     re.compile(r"(?i)\b(d)('ye)\b"),
                     re.compile(r"(?i)\b(gim)(me)\b"),
                     re.compile(r"(?i)\b(gon)(na)\b"),
                     re.compile(r"(?i)\b(got)(ta)\b"),
                     re.compile(r"(?i)\b(lem)(me)\b"),
                     re.compile(r"(?i)\b(mor)('n)\b"),
                     re.compile(r"(?i)\b(wan)(na) ")]
    CONTRACTIONS3 = [re.compile(r"(?i) ('t)(is)\b"),
                     re.compile(r"(?i) ('t)(was)\b")]
    CONTRACTIONS4 = [re.compile(r"(?i)\b(whad)(dd)(ya)\b"),
```

```
                    re.compile(r"(?i)\b(wha)(t)(cha)\b")]

    def tokenize(self, text):

        #starting quotes
        text = re.sub(r'^\"', r'``', text)
        text = re.sub(r'(``)', r' \1 ', text)
        text = re.sub(r'([ (\[{<])"', r'\1 `` ', text)

        #punctuation
        text = re.sub(r'([:,])([^\d])', r' \1 \2', text)
        text = re.sub(r'\.\.\.', r' ... ', text)
        text = re.sub(r'[;@#$%&]', r' \g<0> ', text)
        text = re.sub(r'([^\.])(\.)([\]\)}>"\']*)\s*$', r'\1 \2\3 ', text)
        text = re.sub(r'[?!]', r' \g<0> ', text)

        text = re.sub(r"([^'])' ", r"\1 ' ", text)

        #parens, brackets, etc.
        text = re.sub(r'[\]\[\(\)\{\}\<\>]', r' \g<0> ', text)

        #Spanish 'Â¿' and 'Â¡'
        text = re.sub(r'\xbf', r' \g<0> ', text) # Â¿
        text = re.sub(r'\xa1', r' \g<0> ', text) # Â¡

        #add extra space to make things easier
        text = " " + text + " "

        #ending quotes
        text = re.sub(r'"', " '' ", text)
        text = re.sub(r'(\S)(\'\')', r'\1 \2 ', text)

        text = re.sub(r"([^' ])('[sS]|'[mM]|'[dD]|') ", r"\1 \2 ", text)
        text = re.sub(r"([^' ])('ll|'LL|'re|'RE|'ve|'VE|n't|N'T) ", r"\1 \2 ",
                      text)

        for regexp in self.CONTRACTIONS2:
            text = regexp.sub(r' \1 \2 ', text)
        for regexp in self.CONTRACTIONS3:
            text = regexp.sub(r' \1 \2 ', text)

        # We are not using CONTRACTIONS4 since
        # they are also commented out in the SED scripts
        # for regexp in self.CONTRACTIONS4:
        #     text = regexp.sub(r' \1 \2 \3 ', text)

        return text.split()


if __name__ == "__main__":
    import doctest
    doctest.testmod(optionflags=doctest.NORMALIZE_WHITESPACE)
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
The aim of the script is to train a chunk tokenizer to detect named entities\
    such as persons, locations and organizations in a given document.

    (For the purpose of the training an external maximun entropy model (megam)\
    is used.

After the chunker has been created, it is pickled for subsequent use.
"""

import nltk
import logging
import pickle

MEGAM_FOLDER = 'topics/megam_0.92/megam'

try:
    nltk.config_megam(MEGAM_FOLDER)
except LookupError:
    nltk.config_megam('megam_0.92/megam')


# define logging configuration
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(levelname)-8s %(message)s')


class BigramChunker(nltk.ChunkParserI):
    """This class defines a bigram chunker"""
    def __init__(self, train_sents):
        """Construct a new BigramChunker instance.
            :param train_sents: Array of sentences with named entities tagged

        """
        train_data = [[(t, c) for w, t, c in nltk.chunk.tree2conlltags(sent)]
                      for sent in train_sents]
        self.tagger = nltk.UnigramTagger(train_data)

    def parse(self, sentence):
        """Converts the tag sequence provided by the tagger back into a\
        chunk tree

        :param sentence: Array of word and part of speech tag

        """
        pos_tags = [pos for (word, pos) in sentence]
        tagged_pos_tags = self.tagger.tag(pos_tags)
        chunktags = [chunktag for (pos, chunktag) in tagged_pos_tags]
        conlltags = [(word, pos, chunktag) for ((word, pos), chunktag)
                     in zip(sentence, chunktags)]
        return nltk.chunk.util.conlltags2tree(conlltags)


class ConsecutiveNPChunkTagger(nltk.TaggerI):
    """This class defines a chunker to tag named entities"""
    def __init__(self, train_sents):
        """Construct a new ConsecutiveNPChunkTagger instance.

        :param train_sents: Array of sentences with named entities tagged

        """
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
```

```
                featureset = npchunk_features(untagged_sent, i)
                train_set.append((featureset, tag))
                history.append(tag)

        self.classifier = nltk.MaxentClassifier.train(
            train_set, algorithm='megam', trace=0)

    def tag(self, sentence):
        """Tag the provided sentence with named entities.

        :param sentence: Array of word and part of speech tag

        """
        history = []
        for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)


class ConsecutiveNPChunker(nltk.ChunkParserI):
    """This class is a wrapper around the tagger class that turns it\
     into a chunker"""

    def __init__(self, train_sents):
        """Construct a new ConsecutiveNPChunker instance.

        :param train_sents: Array of sentences with named entities tagged

        """
        tagged_sents = [[((w.lower(), t), c) for (w, t, c) in
                         nltk.chunk.tree2conlltags(sent)]
                        for sent in train_sents]
        self.tagger = ConsecutiveNPChunkTagger(tagged_sents)

    def parse(self, sentence):
        """Converts the tag sequence provided by the tagger back into a\
        chunk tree

        :param sentence: Array of word and part of speech tag

        """
        tagged_sents = self.tagger.tag(sentence)
        conlltags = [(w, t, c) for ((w, t), c) in tagged_sents]
        return nltk.chunk.util.conlltags2tree(conlltags)


def npchunk_features(sentence, i):
    """Feature extractor

    :param sentence: Array of word and part of speech tag
    :param i: Index of the actual word

    """
    word, pos = sentence[i]
    if i == 0:
        prevword, prevpos = "<START>", "<START>"
    else:
        prevword, prevpos = sentence[i-1]

    if i == len(sentence)-1:
        nextword, nextpos = "<END>", "<END>"
    else:
        nextword, nextpos = sentence[i+1]
    return {"pos": pos,
            "word": word,
            "prevpos": prevpos,
```

```
            "nextpos": nextpos,
            "prevpos+pos": "%s+%s" % (prevpos, pos),
            "pos+nextpos": "%s+%s" % (pos, nextpos),
            "tags-since-dt": tags_since_dt(sentence, i)
            }


def tags_since_dt(sentence, i):
    """Creates a string describing the set of all part-of-speech tags\
    that have been encountered since the most recent determiner.

    :param sentence: Array of word and part of speech tag
    :param i: Index of the actual word

    """
    tags = set()
    for word, pos in sentence[:i]:
        if pos == 'DA':
            tags = set()
        else:
            tags.add(pos)
    return '+'.join(sorted(tags))


def test_chunk_tagger():
    """
        Test Chunk tagger.
    """

    from nltk.tokenize import word_tokenize

    logging.info('Loading PoS tagger')
    pos_tag = pickle.load(open("tmp/pos_tagger.p", "rb")).tag
    logging.info('Loading Chunk tagger')
    chunk_tag = pickle.load(open("tmp/chunk_tagger.p", "rb")).parse
    loc = nltk.Tree('LOC', [(u'santander', u'NC')])
    org = nltk.Tree('ORG', [(u'izquierda', u'NC')])

    test_tree = nltk.Tree('S', [org,
    (u'unida', u'AQ'),
    (u'de', u'SP'),
    loc,
    (unicode('presentÃ³', 'utf-8'), u'VMI'),
    (u'hoy', u'RG'),
    (u'su', u'DP'),
    (u'nuevo', u'AQ'),
    (unicode('boletÃ-n', 'utf-8'), u'NC'),
    (u'trimestral', u'AQ')])

    string = unicode(
        """Izquierda Unida de Santander presentÃ³ hoy su nuevo boletÃ-n\
 trimestral""",
        'utf-8'
    )
    tokens = [token.lower() for token in word_tokenize(string)]
    pos_tokens = pos_tag(tokens)
    result = chunk_tag(pos_tokens)

    assert result == test_tree


def main():
    """ Main function. """

    logging.info('Starting training chunker...')
    train_set = nltk.corpus.conll2002.chunked_sents('esp.train')
    chunker_es = ConsecutiveNPChunker(train_set)
```

```
    logging.info('Pickling chunker...')
    pickle.dump(chunker_es, open("tmp/chunk_tagger.p", "wb"))

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

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
The aim of the script is to query the news data base in order to extract the\
description of each new, group them in an 10-day time slice window and infer\
the latent topics.

* The PoS tagger and the Chunk tagger are unpickled for later use.\
* The scripts pickles:\
     - The Topic Distribution (topic_dist): Topic_id + words per topics\
     - The Topic Slices (topic_slice): Slice_Date + array of topic_ids\

"""

import sys
import time
from dateutil import parser
import couchdb
import networkx as nx
from collections import defaultdict
import nltk
from nltk.corpus import stopwords
from gensim import corpora, models
from topics.meneame_tokenizer import TreebankWordTokenizer
import logging
import numpy as np
from topics.train_chunk_tagger import ConsecutiveNPChunker
from topics.train_chunk_tagger import ConsecutiveNPChunkTagger


# define logging configuration
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(levelname)-8s %(message)s')


def tokenize_text(time_slice, sent_tokenizer):
    """
    Tokenize all the text included in time_slice and returns\
    arrays of tokens

    :param time_slice: Array of texts and texts ids

    """

    texts = []
    # Get topics for the previous timeslice
    objects_ids = []
    for text, text_id in time_slice:
        words = []
        for sentence in sent_tokenizer.tokenize(text):
            tokens = [
                token.lower()
                for token in TreebankWordTokenizer().tokenize(sentence)
            ]
            words.extend(tokens)
        objects_ids.append(text_id)
        texts.append(words)

    return texts


def test_tokenize_text():
    """
    Test the tokenize_text function

    """

    sent_es_token = nltk.data.load('tokenizers/punkt/spanish.pickle')
```

```
    result = tokenize_text([["Example to tokenize", 1]], sent_es_token)
    assert result == [["example", "to", "tokenize"]]


def extract_entities(texts, pos_tagger, chunk_tagger):
    """
    Extract entities (named entities, like persons and organizations,\
    and nouns) from arrays of tokens

    :param texts: Array of tokens
    :param pos_tagger: Part of Speech tagger
    :param chunk_tagger: Chunk tagger

    """

    nouns = ['NC', 'N_N', 'NP']
    entities = []
    stop_w = stopwords.words('spanish')

    for tokens in texts:
        words = []
        pos_tokens = pos_tagger(tokens)
        named_tokens = chunk_tagger(pos_tokens)
        for token in named_tokens:
            if type(token) == nltk.tree.Tree:
                words.append(
                    ' '.join(
                        [
                            i[0] for i in token.leaves()
                            if len(i[0]) > 2 and len(i[0]) < 20
                        ]
                    )
                )
            else:
                if len(token[0]) <= 2 or len(token[0]) >= 20:
                    continue
                if token[1] not in nouns:
                    continue
                if token[0].lower().strip('.').encode('utf-8') in stop_w:
                    continue

                words.append(token[0])

        entities.append(words)

    return entities


def hellinger_distance(prob_a, prob_b):
    """
    Calculates the Hellinger distance between p and q.

    :param p: Probability distribution over a vocabulary
    :param q: Probability distribution over a vocabulary


    """
    i = [t[0] for t in sorted(prob_a, key=lambda word: word[1])]
    j = [t[0] for t in sorted(prob_b, key=lambda word: word[1])]

    return 1 - np.sqrt(np.sum((np.sqrt(i) - np.sqrt(j)) ** 2)) / np.sqrt(2)


def test_hellinger_distance():
    """
    Test the hellinger_distance function

    """
```

```
    prob_a = [(0.25, 'A'), (0.25, 'B'), (0.25, 'C'), (0.25, 'D')]
    prob_b = [(0.25, 'A'), (0.25, 'B'), (0.25, 'C'), (0.25, 'D')]
    result = hellinger_distance(prob_a, prob_b)
    assert result == 1.0

    prob_a = [(0.25, 'A'), (0.25, 'B'), (0.25, 'C'), (0.20, 'D'), (0.05, 'E')]
    prob_b = [(0.2, 'A'), (0.2, 'B'), (0.2, 'C'), (0.2, 'D'), (0.2, 'E')]
    result = hellinger_distance(prob_a, prob_b)
    assert result == 0.82917960675006308


def merging_topics(topic_dist, topic_list, threshold, sim_metric):
    """
    Calculates similarities between a list of topics and merge them if their\
    similarity is greater than a threshold

    :param topic_dist: Dictionary of topic ids and associated words
    :param topic_list: Array of topics as probability distributions
    :param threshold: Value to check if two topics are similar
    :param sim_metric: Function to compare topics

    """

    # Get similarties between topics
    topics_score = defaultdict(list)
    index_a = 0
    for topic_a in topic_list:
        index_b = 0
        for topic_b in topic_list:
            if index_a != index_b:
                score = sim_metric(topic_a, topic_b)
                topics_score[index_a].append({index_b: score})
            index_b += 1
        index_a += 1

    # Check similarity between topics grouping similar topics (score > 0.7)
    # making edges between them in a network
    graph = nx.Graph()
    for topic_key in topics_score:
        # Order according to max similarity
        score_list = sorted(
            topics_score[topic_key],
            key=lambda x: x.items()[0][1],
            reverse=True
        )
        for topic_value in score_list:
            if topic_value.values()[0] > threshold:
                graph.add_edge(topic_value.keys()[0], topic_key)
            else:
                graph.add_edge(topic_key, topic_key)

    components = nx.connected_components(graph)

    topic_dict = {}
    topic_ids_in_slice = []

    # All the topics in a component represent the same topic
    for i in components:
        # All the topics in the component have the same id
        # id = len(topic_dist)
        for topic_id in i:
            topic_dict[topic_id] = len(topic_dist)

        # We always choose the first topic of the component
        # as representative of all the topics in the component
        topic_dist[len(topic_dist)] = topic_list[i[0]]
        topic_ids_in_slice.append(len(topic_dist))
```

```
    return (topic_dict, topic_ids_in_slice)


def get_topics(tagger_es, chunker_es, news_db, window_size=10, num_topics=35):
    """Main function"""

    ini = time.time()
    couch = couchdb.Server()
    try:
        topic_db = couch.create('meneame_topic_db')
    except Exception:
        topic_db = couch['meneame_topic_db']

    logging.info('Loading Spanish Sent Tokenizer')
    sent_es_token = nltk.data.load('tokenizers/punkt/spanish.pickle')

    logging.info('Creating news corpus...')
    news = []
    for post in news_db:
        new = dict(news_db.get(post))
        news.append(
            {
                'published': parser.parse(new['published']),
                'description': new['description'],
                '_id': new['_id']
            }
        )

    # Order news by date
    ordered_news = sorted(news, key=lambda element: element['published'])
    last_date = ''
    index = 0
    mssg_id = 0
    len_query = len(ordered_news)
    topic_dist = dict()
    time_slices = defaultdict(list)
    topic_slices = {}

    logging.info('Creating time slices...')
    for post in ordered_news:
        # Group articles in a "window_size"-day window
        if index == 0:  # first time
            last_date = post['published']
            index += 1

        if (post['published'] - last_date).days < window_size:
            if post['description'] is not None:
                time_slices[last_date].append(
                    [post['description'], post['_id']]
                )
                sys.stderr.write(
                    'ids:%d [%f]\r' %
                    (mssg_id, float(mssg_id*100)/float(len_query))
                )
                mssg_id += 1
        else:
        # New 10-day timeslice
            # Creating tokens
            texts = tokenize_text(time_slices[last_date], sent_es_token)

            # Extract entities
            entities = extract_entities(texts, tagger_es, chunker_es)

            # Discard tokens that appear once
            tokens = sum(entities, [])
            lonely_tokens = set(
                word for word in set(tokens) if tokens.count(word) == 1
            )
```

```
            texts = [
                [word for word in entity if word not in lonely_tokens]
                for entity in entities
            ]

            dictionary = corpora.Dictionary(texts)
            corpus = [dictionary.doc2bow(text) for text in texts]

            # Creating TF-IDF corpus
            tfidf = models.TfidfModel(corpus)
            corpus_tfidf = tfidf[corpus]

            # Create LDA model
            model = models.ldamodel.LdaModel(
                corpus_tfidf,
                id2word=dictionary,
                num_topics=num_topics,
                update_every=1,
                chunksize=20,
                passes=1
            )

            topic_list = []
            # Extract Topics from model
            for i in range(num_topics):
                # Get all the words of the probability distribution
                topic = model.show_topic(i, len(dictionary))
                topic_list.append(topic)

            # Merge topics if their similarity score is greater than 0.7
            topic_dict, topic_ids_in_slice = merging_topics(
                topic_dist,
                topic_list,
                0.7,
                hellinger_distance
            )

            topic_dist_per_slice = [new for new in model[corpus_tfidf]]
            #Once we have the topics, assign them to the news in the DB
            for text, id_new in time_slices[last_date]:
                # Order topics for each new according to the most
                # representative, and select it
                best_topic = sorted(
                    topic_dist_per_slice[index],
                    key=lambda t: t[1],
                    reverse=True
                )[0]
                topic_db.save(
                    {
                        "article_id": id_new,
                        "topic_id": topic_dict[best_topic[0]],
                        "slice_id": len(topic_slices),
                        "slice_date": last_date.strftime('%d/%m/%Y')
                    }
                )

            topic_slices[last_date] = topic_ids_in_slice

            # Update new time_lice date
            last_date = post['published']
            if post['description'] is not None:
                time_slices[last_date].append(
                    [post['description'], post['_id']]
                )
            sys.stderr.write(
                'ids:%d [%f]\r' %
                (mssg_id, float(mssg_id*100)/float(len_query))
            )
```

```
            mssg_id += 1

    fin = time.time()
    logging.info('Time consumed: %f' % ((float(fin)-float(ini))/60/60))
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
The aim of the script is to build the network of users of meneame, defined
as follows:

* Each vertex is a user that has commented in one of the articles we\
    analyzed. Each *vertex* has an attribute, "comments", that represents\
    the total number of comments an user has posted;
* There is an *edge* between two users if they posted in the same article.\
    Each edge has a weight representing the number of articles in which they\
    posted together.

After the network has been created, it is pickled for subsequent analysis.
"""

#import igraph as ig
import couchdb
import sys
import itertools
import collections
from network.network import create_graph


def main():
    """ Main function. """
    #Connection with the database
    try:
        couch = couchdb.Server()
        news_db = couch['meneame']
    except couchdb.ResourceNotFound:
        print "Connection error with the database!"
        sys.exit(1)

    vertices_g = collections.Counter()
    edges_g = collections.Counter()

    #Populating vertices and edges collections
    for article in news_db:
        item = dict(news_db.get(article))['commenters']
        vertices_g.update(**item)
        edges_g.update(itertools.combinations(item.keys(), 2))

    graph = create_graph(vertices_g, edges_g)
    graph.simplify(combine_edges='sum')

    graph.write_pickle('meneame_network.pickle')

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

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
The aim of this script is to analyze in various ways the meneame network.
"""
import igraph as ig
import os
import sys
from network.network import general_analysis, community_analysis
from network.network import save_degree_distribution
from network.network import save_weights_distribution

IMAGES_FOLDER = "images/"
NETWORK_FILE = "meneame_network.pickle"


def main():
    """Main function"""

    if not os.path.exists(NETWORK_FILE):
        print "Input network file is not present!"
        sys.exit(1)

    graph = ig.load(NETWORK_FILE)
    graph['name'] = 'meneame'

    general_analysis(graph)

    community_analysis(graph)

    save_degree_distribution(graph, IMAGES_FOLDER)
    save_weights_distribution(graph, IMAGES_FOLDER)


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

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Meneame scraper

Usage:
    scrap.py [options]

Options:
    -h, --help      show this screen.
    --version       show version.
    -o DIRECTORY    output directory [default: meneame].
    --start START   specify start page [default: 0].
    --timeout TIME  specify timeout after each request [default: 0].
    --range RANGE   specify the range. valid options\
 (24h | 48h | week | month | year | all) [default: 24h].

"""
import logging
import json
from scraper.scraper import ScraperFactory
from docopt import docopt
from time import sleep
import os

SCRAPER_TYPE = 'meneame'
SCRAPER_BASE_URL = 'http://www.meneame.net/'
SCRAPER_NEWS_URL = 'topstories.php'
SCRAPER_COMMENTS_URL = 'comments_rss2.php'

# define logging configuration
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s %(levelname)-8s %(message)s')


def download_news(output, start=0, time_range=1, pause=1):
    """
        Download news from meneame, and store them in the\
        specified output directory.

            :param output: the output directory
            :param start: the number of page where to \
            start at.
            :param time_range: the time range to scrap.
            :param pause: the number of seconds to wait\
            after each request.

    """
    output_dir = os.path.join(output, 'raw')

    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    current_page = start  # page counter
    more_news = True

    meneame = ScraperFactory.factory(SCRAPER_TYPE, SCRAPER_BASE_URL,
                                     SCRAPER_NEWS_URL,
                                     SCRAPER_COMMENTS_URL)

    while more_news:
        stories = meneame.scrap_page(params={
            'range': time_range,
            'page': current_page})
        logging.info('Page %s: [%s]',
                     current_page,
                     ",".join([str(story.id) for story in stories]))
        for story in stories:
```

```
            filepath = os.path.join(output_dir, '%s.json' % story.id)
            if not os.path.exists(filepath):
                logging.debug('Downloading comments for %s', story.id)
                story.comments, story.published = meneame.scrap_comments(
                    params={'id': story.id})
                logging.debug('Writing %s ...', filepath)
                filename = open(filepath, 'w')
                json.dump(story.to_dict(), filename)
                filename.close()
                sleep(pause)
            else:
                logging.warning('File %s already exists...', story.id)

        if not stories:
            more_news = False

        current_page += 1


if __name__ == '__main__':
    ARGS = docopt(__doc__, version='Meneame Scraper 1.0')

    # transform the time range to index
    try:
        TIMERANGE = ['24h', '48h',
                     'week', 'month', 'year',
                     'all'].index(ARGS['--range'])
    except ValueError:
        logging.error('Error reading time range, not valid. Using \
            default value [24h].')
        TIMERANGE = 0

    download_news(output=ARGS['-o'],
                  start=int(ARGS['--start']),
                  time_range=TIMERANGE,
                  pause=int(ARGS['--timeout']))
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
        Scraper module

        Contains Scraper classes for scraping and cleaning from\
        social news websites. Currently it also implements a\
        scraper for MenÃ©ame.
"""
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
This module contains the necessary functions for cleaning the articles.
"""

from bs4 import BeautifulSoup


def strip_html_tags(html):
    """Strip out all html tags from the input text.

        :param html: input html string
        :returns: the input string stripped of all html tags
    """
    return ''.join(BeautifulSoup(html).findAll(text=True))


def test_strip_html_tags():
    """Test for the strip_html_tags function."""
    text = list()
    stripped = list()

    text.append("This is a text without HTML")
    stripped.append("This is a text without HTML")

    text.append("This text contains <b>one</b> tag")
    stripped.append("This text contains one tag")

    text.append("This text <i>contains</i> <b>two</b> tags")
    stripped.append("This text contains two tags")

    text.append("This <br>text <i>contains</i> <b><i>multiple</i></b> tags")
    stripped.append("This text contains multiple tags")

    for i in range(len(text)):
        assert strip_html_tags(text[i]) == stripped[i]
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
    Python classes for that model the data used in the scripts.
"""
class Story(object):
    """
        A story
    """
    def __init__(self, sid=None, votes=None, clicks=None, title=None,
        url=None, author=None, description=None, published=None, comments=None):
        self.id = sid
        self.votes = votes
        self.clicks = clicks
        self.title = title
        self.url = url
        self.author = author
        self.description = description
        self.published = published
        self.comments = comments

    def to_dict(self):
        """
            Converts the class to a dictionary to be handled as a JSON
        """
        out = {}
        out['_id'] = str(self.id)
        out['votes'] = self.votes
        out['clicks'] = self.clicks
        out['title'] = self.title
        out['url'] = self.url
        out['author'] = self.author
        out['description'] = self.description
        out['published'] = self.published

        comments = []
        for comment in self.comments:
            comments.append(comment.to_dict())
        out['comments'] = comments

        return out


class Comment(object):
    """
        A comment
    """
    def __init__(self, story, order=None, karma=None, user=None,
        votes=None, cid=None, published=None, summary=None):
        self.story = story
        self.order = order
        self.karma = karma
        self.user = user
        self.votes = votes
        self.id = cid
        self.published = published
        self.summary = summary

    def to_dict(self):
        """
            Converts the class to a dictionary to be handled as a JSON
        """
        out = {}
        out['order'] = self.order
        out['karma'] = self.karma
        out['user'] = self.user
        out['votes'] = self.votes
        out['id'] = self.id
```

```
        out['published'] = self.published
        out['summary'] = self.summary
        return out
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
    Module for scraping meneame stories
"""
from model import Story, Comment
import requests
import logging
import feedparser
from bs4 import BeautifulSoup
import re
import os


class ScraperFactory(object):
    """
    ScraperFactory: returns a Scraper implementation.

    """
    def factory(scrapper_type, base_url, stories_url, comments_url):
        """
            Returns a scrapper implementation.
        """
        if scrapper_type == 'meneame':
            return MeneameScraper(base_url, stories_url, comments_url)
    factory = staticmethod(factory)


class Scraper(object):
    """
        Abstract class Scraper which provides the basic methods for\
        scraping a social news website. It relies on subclasses for\
        specific methods.
    """

    def __init__(self, base_url, stories_url, comments_url):
        self.base_url = base_url
        self.stories_url = stories_url
        self.comments_url = comments_url

    @staticmethod
    def _scrap(url, params):
        """
            Private method. Should not be called directly. Reads an\
            HTTP page given a URL and some params.

                :param url: url to do an HTTP request.
                :param params: a dictionary of GET parameters.
        """
        try:
            logging.debug('requesting %s [%s]', url, params)
            req = requests.get(url, params=params)
        except requests.exceptions.ConnectionError:
            logging.error('error in the connection, skipping [params=%s]',
                          ','.join(params))
            return None
        if req.status_code is not 200:
            return None
        logging.debug('scrapped page %s ...', req.url)
        return req

    def scrap_page(self, params):
        """
            Requests a given HTTP page given the provided GET\
            parameters. Returns a list of Stories.

                :param params: a dictionary of GET parameters.
```

```
        """
        req = self._scrap(self.base_url + self.stories_url, params)
        if req is None:
            return None
        return self.extract_stories(req.text)

    def extract_stories(self, text):
        """
            Extract stories from the given HTTP text. Should be\
            implemented by subclasses of this Scraper class.

                :param text: a string read from an HTTP request.

        """
        pass

    def scrap_comments(self, params):
        """
            Requests a given HTTP page given the provided GET\
            parameters. Returns a list of Comments.

                :param params: a dictionary of GET parameters.

        """
        req = self._scrap(self.base_url + self.comments_url, params)
        if req is None:
            return None
        return self.extract_comments(params['id'], req.text)

    def extract_comments(self, sid, text):
        """
            Extract comments from the given HTTP text. Should be\
            implemented by subclasses of this Scraper class.

                :param sid: the id of the new of these comments.
                :param text: a string read from an HTTP request.

        """
        pass


class MeneameScraper(Scraper):
    """
        Implementation of the class Scraper. Provides methods for\
        obtaining meename stories and comments.
    """

    def __init__(self, base_url, stories_url, comments_url):
        super(MeneameScraper,
              self).__init__(base_url, stories_url, comments_url)

    def extract_stories(self, text):
        """
            Implementation of extract stories from the given \
            HTTP text. Returns a list of Stories.

                :param text: a string read from an HTTP request.

        """
        parsed_stories = []

        soup = BeautifulSoup(text)
        stories = soup.find_all('div', {'class': 'news-body'})

        for story in stories:
            # build a dict with all the relevant attributes
            meneame_story = Story()
```

```
            # number of votes
            id_temp = story.find('div', {'class': 'votes'})
            if id_temp:
                meneame_story.votes = int(id_temp.a.string)
            else:
                meneame_story.votes = 0

            try:
                # extract the id
                id_regex = re.match(r'a-votes-(\d*)', id_temp.a['id'])
                if id_regex:
                    meneame_story.id = int(id_regex.group(1))
            except AttributeError:
                logging.error('Could not read id for new, skipping ...')
                continue

            if story.h2 is not None:
                meneame_story.title = story.h2.a.string
                meneame_story.url = story.h2.a['href']
            else:
                meneame_story.title = ""
                meneame_story.url = ""

            # number of clicks
            clicks = story.find('div', {'class': 'clics'})
            if clicks is not None:
                clicks_regex = re.match(r'\s*(\d+)\s.*', clicks.string)
                if clicks_regex:
                    meneame_story.clicks = int(clicks_regex.group(1))
                else:
                    logging.error('Error reading clicks for story %s',
                                  meneame_story.id)
                    meneame_story.clicks = 0
            else:
                meneame_story.clicks = 0

            # extract the user id
            user_a = story.find('a', {'class': 'tooltip'})
            try:
                user_regex = re.match(r'\/user\/(.*)', user_a['href'])
                if user_regex:
                    meneame_story.author = user_regex.group(1)
            except (TypeError, ValueError):
                logging.error('Error reading user for story %s',
                              meneame_story.id)
                meneame_story.user = ""

            # extract description
            try:
                meneame_story.description = story.contents[8]
            except IndexError:
                logging.error('Error reading description for story %s',
                              meneame_story.id)
                meneame_story.description = " "

            parsed_stories.append(meneame_story)
        return parsed_stories

    def extract_comments(self, sid, text):
        """
        Extracts comments from a XML string. The parsing is done
        using feedparser library.

            :param sid: the id of the new.
            :param text: the RSS xml.

    """
        parsed = feedparser.parse(text)
```

```
        try:
            published = parsed.feed.published
        except AttributeError:
            published = parsed.feed.updated

        comments = []
        for comment in parsed.entries:
            meneame_comment = Comment(sid)
            meneame_comment.order = comment['meneame_order']
            meneame_comment.karma = comment['meneame_karma']
            meneame_comment.user = comment['meneame_user']
            meneame_comment.votes = comment['meneame_votes']
            meneame_comment.id = comment['meneame_comment_id']
            try:
                meneame_comment.published = comment.published
            except AttributeError:
                meneame_comment.published = comment.updated
            meneame_comment.summary = comment.summary
            comments.append(meneame_comment)

        return comments, published


def test_extract_stories():
    """
        Test for the extract_stories function.
    """

    meneame = ScraperFactory.factory('meneame',
                                     'http://www.meneame.net/',
                                     'topstories.php',
                                     'comments_rss2.php')

    test_data = open(os.path.join(os.path.dirname(__file__),
                     'test_data.html')).read()
    stories = meneame.extract_stories(test_data)

    # assert there are 15 stories parsed
    assert len(stories) is 15

    # assert that each story has id, author, description, ....
    story = stories[0]
    assert story.id == 2066791
    assert story.title == u"La PolicÃ-a intenta cerrar Canal 9 \
y los trabajadores lo impiden. #RTVVnoestanca "
    assert story.votes == 711
    assert story.clicks == 2848
    assert story.url == u"https://www.youtube.com/watch?v=c6mX4owi1fY"
    assert story.author == u"ninyobolsa"


def test_extract_comments():
    """
        Test for the extract_comments function.
    """
    meneame = ScraperFactory.factory('meneame',
                                     'http://www.meneame.net/',
                                     'topstories.php',
                                     'comments_rss2.php')

    test_data = open(os.path.join(
                     os.path.dirname(__file__), 'test_comments.xml')).read()
    comments, published = meneame.extract_comments(2067716, test_data)
    assert len(comments) is 77
    assert published == u'Sat, 30 Nov 2013 00:31:00 +0000'
    comment = comments[0]
    assert comment.order == u'77'
    assert comment.karma == u'18'
```

```
    assert comment.user == u'Lucer'
    assert comment.published == u'Sat, 30 Nov 2013 00:31:00 +0000'
    assert comment.id == u'13918274'
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
        The sentiment analysis module

        Provides the required methods and tools to perform\
        sentiment analysis to texts.

"""
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
    Module that computes the sentiment for a given text and a given
    map with word => (valence, arousal)
"""
from nltk.tokenize import PunktWordTokenizer
from nltk.stem import SnowballStemmer

STEMMER = SnowballStemmer('spanish')


def get_sentiment(sentiment_db, txt):
    """
        Returns a tuple with the valence and arousal strength \
        based on the input text. Returns null in case it cannot \
        be computed.

            :param sentiment_db: the file ANEW
            :param txt: the sentence to be analysed.

    """
    words = PunktWordTokenizer().tokenize(txt)
    try:
        sentiments = map(lambda word: sentiment_db.get(
            STEMMER.stem(word), None), words)
        sentiments = filter(None, sentiments)
    except IndexError:
        sentiments = None
    if sentiments:
        valences = [s['valence'] for s in sentiments if s is not None]
        arousals = [s['arousal'] for s in sentiments if s is not None]
        valence = float(sum(valences))/len(valences)
        arousal = float(sum(arousals))/len(arousals)
        return valence, arousal
    else:
        return None


def test_get_sentiments():
    """
        Test get_sentiment function.
    """

    import os
    import csv

    csvfile = open(os.path.join(os.path.dirname(__file__),
                   'test_anew.csv'), 'r')

    reader = csv.reader(csvfile, delimiter=',')
    reader.next()  # skip headers
    anew = dict([(STEMMER.stem(unicode(row[2], 'utf-8')),
                  {'valence': float(row[3]),
                   'arousal': float(row[5])}) for row in reader])

    text = """En realidad los filÃ³sofos griegos importantes \
odiaban la democracia (ENG)"""

    valence, arousal = get_sentiment(anew, text.decode('utf-8'))
    assert valence == 1.74
    assert arousal == 7.01
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""The aim of this package is to deal with network creation and analysis\
 of the meneame network.
"""
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
This module contains the functions for the network analysis and creation.
"""

import igraph as ig
import collections
import matplotlib.pyplot as plt


def test_create_graph():
    """Testing function for create_graph()
    """
    vertices = {'user1': 12, 'user2': 8, 'user3': 7, 'user4': 4}
    vertices = collections.Counter(vertices)
    edges = {('user1', 'user2'): 23,
             ('user1', 'user3'): 12, ('user2', 'user4'): 11}
    edges = collections.Counter(edges)
    test_graph = create_graph(vertices, edges)

    for v in test_graph.vs:
        assert v['comments'] == vertices[v['name']]

    for e in test_graph.es:
        source = test_graph.vs[e.source]['name']
        target = test_graph.vs[e.target]['name']
        ed = tuple(sorted((source, target)))
        assert edges[ed] == e['weight']


def create_graph(vertices, edges):
    """Return the graph object, given the edges and vertices collections.

    :param vertices: collection of vertices, where each element is in the
        \format "username: number_of_comments"
    :param edges: collection of edges, where each element is in the format\
        (username1, username2): weight, where the weight is the number\
        of articles in which the two users have commented together\
    :returns: the igraph object

    The creation of the graph is done only on the final step of the
    function, due to the way igraph deals with the edges.
    Infact, building first the graph and then the edges would result in a
    very inefficient code, as the edges are re-indexed every time a new edge
    is added.
    """
    usernames = vertices.keys()
    comments = vertices.values()
    users_dic = {name: idx for (idx, name) in enumerate(usernames)}
    n_users = len(usernames)

    edges_list = [(users_dic[el1],
                   users_dic[el2]) for (el1, el2) in edges.keys()]
    weights_list = edges.values()

    vertex_attrs = {'name': usernames, 'comments': comments}
    edge_attrs = {'weight': weights_list}
    return ig.Graph(n=n_users, edges=edges_list, vertex_attrs=vertex_attrs,
                    edge_attrs=edge_attrs)


def save_degree_distribution(graph, image_folder):
    """Save the degree distribution of the input graph to file.

        :param graph: the input igraph object
        :image_folder: the path to the folder where to save the images
    """
```

```
    degree_dist = graph.degree_distribution()
    x = [el[0] for el in degree_dist.bins()]
    y = [el[2] for el in degree_dist.bins()]

    title = "Network Degree Distribution"
    xlabel = "Degree"
    ylabel = "Number of nodes"
    filename = graph['name'] + "_degree_distribution"
    save_log_histogram(x, y, title, xlabel, ylabel, filename, image_folder)


def save_weights_distribution(graph, image_folder):
    """Saves the weigth distribution to file.

        :param graph: the input igraph object
        :image_folder: the path to the folder where to save the images
    """
    weights = graph.es['weight']

    co = collections.Counter(weights)
    x = [el for el in co]
    y = [co[el] for el in co]

    title = "Weights distribution"
    xlabel = "Weight"
    ylabel = "Number of edges"
    filename = graph['name'] + "_weights_distribution"
    save_log_histogram(x, y, title, xlabel, ylabel, filename, image_folder)


def save_log_histogram(x, y, title, xlabel, ylabel, filename, image_folder):
    """Create a loglog histogram from the input x and y and saves it to\
    file. Each bin has an unitary size.

        :param x: bin positions
        :param y: count values
        :param title: title of the histogram
        :param xlabel: label of the x axis
        :param ylabel: label of the y axis
        :param filename: name of the saved file

    """
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(x, y)
    ax.set_xscale('log')
    ax.set_yscale('log')
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    plt.title(title)
    plt.savefig(image_folder + filename + '.png')
    plt.savefig(image_folder + filename + '.pdf')


def community_analysis(graph):
    """Analyze the communities of the input graph.

        :param graph: the input igraph object

    """
    print "\nCOMMUNITY ANALYSIS\n"

    communities = graph.community_infomap(edge_weights='weight',
                                          vertex_weights='comments',
                                          trials=10)
    print "Infomap  ", communities.summary()
    print "Sizes of communities: ", communities.sizes()
    single_community_analysis(communities)
```

```
def single_community_analysis(communities):
    """Analyze the communities found. The analysis is done only if more\
    than one community has been found.

        :param graph: the input VertexClustering object representing the\
        communities

    """
    n_comm = len(communities.sizes())
    if n_comm > 1:
        for idx in range(n_comm):
            print "\n COMMUNITY NUMBER ", idx, "\n"
            general_analysis(communities.subgraph(idx))


def general_analysis(graph):
    """Print to screen some basic information regarding the input graph.

        :param graph: the input igraph object

    """
    comments = graph.vs['comments']
    weights = graph.es['weight']

    print "GENERAL ANALYSIS\n"

    print "Number of users: ", graph.vcount()
    print "Number of links: ", graph.ecount()

    print "\nMax. number of comment per user: ", max(comments)
    print "Min. number of comment per user: ", min(comments)
    print "Average number of comments: ",
    print float(sum(comments)) / len(comments)

    print "\nMax. value of weight: ", max(weights)
    print "Min. value of weight: ", min(weights)
    print "Average weight: ", float(sum(weights)) / len(weights)

    print "\nClustering coefficient: ", graph.transitivity_undirected()

    components = graph.components()
    print "\nNumber of connected components: ", len(components.sizes())

    hist = graph.path_length_hist()
    lengths, paths = [[el[i] for el in list(hist.bins())] for i in [0, 2]]
    print "\nPath length distibution: "
    for i in range(len(lengths)):
        print paths[i], " paths with length ", lengths[i]

    print "\nAverage path length: ",
    splengths = sum([lengths[i] * paths[i] for i in range(len(lengths))])
    print splengths / sum(paths)
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Meneame sentiments

Usage:
    sentiment.py [--input FILE] [--database DB]

Options:
    -h, --help      show this screen.
    --version       show version.
    --database DB   database where the text is stored [default: meneame].
    --input ANEW    input ANEW file
"""
import csv
import logging
import couchdb
from nltk.stem import SnowballStemmer
from sentiments.sentiments import get_sentiment
from docopt import docopt
import os
import sys

# define logging configuration
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(levelname)-8s %(message)s')


def main(input_file, dbname):
    """
        Main function. Connects to a database and reads a\
        CSV with the arousal and valence. Uses the sentiment \
        library to compute the sentiment of a new.

          :param input_file: the ANEW file
          :param dbname: the name of the database

    """

    # read ANEW file
    if not os.path.exists(input_file):
        logging.error('File %s does not exist', input_file)
        sys.exit(1)
    else:
        csvfile = open(input_file, 'r')
        reader = csv.reader(csvfile, delimiter=',')
        reader.next()  # skip headers
        stemmer = SnowballStemmer('spanish')
        anew = dict([(stemmer.stem(unicode(row[2], 'utf-8')),
                     {'valence': float(row[3]),
                      'arousal': float(row[5])}) for row in reader])

    couch = couchdb.Server()
    database = couch[dbname]
    logging.info('Established connection with the db %s', dbname)

    for element in database:
        doc = database.get(element)

        comments = " ".join([comment['cleaned_summary']
                             for comment in doc['comments']])
        description = " ".join([database.get(element)['title'],
                                doc['description']])

        sentiment_comments = get_sentiment(anew, comments)
        sentiment_description = get_sentiment(anew, description)

        if sentiment_comments is not None and sentiment_description is not None:
            logging.info('%s val: %.2f - %.2f aro: %.2f - %.2f : %s',
```

```
                            doc.id, sentiment_comments[0],
                            sentiment_description[0],
                            sentiment_comments[1],
                            sentiment_description[1],
                            doc['title'])
                doc['sentiments'] = {'comments':
                                     {'valence': sentiment_comments[0],
                                      'arousal': sentiment_comments[1]},
                                      'description':
                                     {'valence': sentiment_description[0],
                                      'arousal': sentiment_description[1]}}
                database.save(doc)

            else:
                logging.warn('%s could not be analyzed. skiping ...',
                             database.get(element)['title'])

if __name__ == '__main__':
    ARGS = docopt(__doc__, version='Meneame Scraper 1.0')
    main(ARGS['--input'], ARGS['--database'])
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
The aim of the script is to query the news data base in order to merge the\
news with each of the topics assigned to them.

The final dataset containing all the information is saved in a JSON file.
"""


import couchdb
try:
    import jsonlib2 as json
except ImportError:
    import json
import logging

# define logging configuration
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(levelname)-8s %(message)s')


def main():
    """Main function"""

    couch = couchdb.Server()
    topics_db = couch['meneame_topic_db']
    news_db = couch['meneame']
    logging.info('Loading topic distribution...')

    logging.info('Retrieving news from DB...')
    news = {}
    for post in news_db:
        new = dict(news_db.get(post))
        news[new['_id']] = {
            'description': new['description'],
            'title': new['title'],
            'votes': new['votes']
        }

    logging.info('Merging news and topics...')
    for topic in topics_db:
        aux = dict(topics_db.get(topic))
        data = news[aux['article_id']]

        data['topic_id'] = aux['topic_id']
        data['slice_id'] = aux['slice_id']
        data['slice_date'] = aux['slice_date']

        news[aux['article_id']] = data

    logging.info('Generating JSON files...')
    json.dump(news, open('web/meneapp/assets/data/topic_news.json', 'w'))


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

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
The aim of this script is to generate a JSON file to feed the\
visualizations of the website.

"""
import couchdb
import json
import logging
import datetime
from collections import defaultdict

def main():
    """
        Main function. Connects to the database and generates\
        a JSON file with the average sentiment per day.
    """
    logging.info('Connecting to database ....')
    couch = couchdb.Server()
    database = couch['sentiments']
    logging.info('Established connection to db ...')

    valence = defaultdict(list)
    arousal = defaultdict(list)
    title = defaultdict(list)
    votes = defaultdict(list)

    for row in database.view('sentiments/sentiment'):
        date = datetime.date(row.key[0], row.key[1]+1, row.key[2])
        valence[date.isoformat()].append(row.value[2])
        arousal[date.isoformat()].append(row.value[1])
        title[date.isoformat()].append(row.value[3])
        votes[date.isoformat()].append(int(row.value[4]))

    average = []
    for date, valences in valence.iteritems():
        average.append({'date': date, 'valence': sum(valences)/len(valences),
            'arousal': sum(arousal[date])/len(arousal[date]),
            'votes': sum(votes[date])/len(votes[date]),
            'title': '\n'.join(title[date])})

    filename = open('sentiment_description.json', 'w')
    json.dump(average, filename)
    filename.close()


if __name__ == '__main__':
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s %(levelname)-8s %(message)s')
    main()
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
The aim of this script is to load the articles json files to the couchdb\
database. The id in the db is set to be the same as the identifier of the\
article on the meneame webiste.

Usage:
    to_database.py [--input MENEAME] [--db DATABASE]

Options:
    -h, --help      show this screen.
    --version       show version.
    --input INPUT   input folder.
    --db DATABASE    output database [default: meneame].

"""
from docopt import docopt
import os
import couchdb
import json
import logging
import sys


def main(path, dbname):
    """ Main function. """
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s %(levelname)-8s %(message)s')

    #Connection with the database
    try:
        couch = couchdb.Server()
        database = couch[dbname]
    except couchdb.ResourceNotFound, err:
        logging.exception("Connection error with the database! \n")
        logging.exception(err)
        sys.exit(1)

    logging.info('Connection to the server estabilished')

    if not os.path.exists(path):
        logging.error('Input folder %s does not exists', path)
        sys.exit(1)

    json_news = [doc for doc in os.listdir(path) if doc.endswith(".json")]

    for news in json_news:
        news_document = json.load(open(os.path.join(path, news)))
        doc_id = str(news_document['_id'])
        news_document['_id'] = doc_id
        if database.get(doc_id) is None:
            database.save(news_document)
            logging.info('Document {0} saved'.format(doc_id))
        else:
            logging.info('Document {0} was already saved'.format(doc_id))


if __name__ == '__main__':
    ARGS = docopt(__doc__, version='Meneame Scraper 1.0')
    main(ARGS['--input'], ARGS['--db'])
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Meneame topic

Usage:
    topic.py [--window Number] [--topic Number] [--dbname DB]

Options:
    -h, --help      show this screen.
    --version       show version.
    --window Number     size of the time slice
    --topic Number      number of topics for the analysis
    --dbname DB   database where the text is stored [default: meneame].
"""

from topics.topic_analysis import get_topics
from topics.train_chunk_tagger import ConsecutiveNPChunker
from topics.train_chunk_tagger import ConsecutiveNPChunkTagger
import logging
import couchdb
from docopt import docopt
import pickle


# define logging configuration
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(levelname)-8s %(message)s')


def main(window, topics, dbname):
    """
        Main function.

          :param window: size of the time-slice
          :param topics: number of topics for the analysis
          :param dbname: the name of the database

    """

    couch = couchdb.Server()
    database = couch[dbname]
    logging.info('Established connection with the db %s', database)

    logging.info('Loading Part of Speech tagger')
    tagger = pickle.load(open("topics/tmp/pos_tagger.p", "rb")).tag

    logging.info('Loading Chunk tagger')
    chunker = pickle.load(open("topics/tmp/chunk_tagger.p", "rb")).parse

    get_topics(tagger, chunker, database, int(window), int(topics))

if __name__ == '__main__':
    ARGS = docopt(__doc__, version='Meneame Topics 1.0')
    main(ARGS['--window'], ARGS['--topic'], ARGS['--dbname'])
```

```
import os
import sys

# credit:  Nick Johnson of Google
sys.path.append(os.path.join(os.path.dirname(__file__), 'lib'))
```

```
import webapp2
import jinja2
import os

JINJA_ENVIRONMENT = jinja2.Environment(
    loader=jinja2.FileSystemLoader(os.path.join(os.path.dirname(__file__), '../view')))

class TimeHandler(webapp2.RequestHandler):

        def get(self):

                template_values = {
                        'title': 'meneapp'
                }
```

```
import webapp2
import jinja2
import os

JINJA_ENVIRONMENT = jinja2.Environment(
    loader=jinja2.FileSystemLoader(os.path.join(os.path.dirname(__file__), '../view')))

class NetworkHandler(webapp2.RequestHandler):

    def get(self):

        template_values = {
            'title': 'Meneame',
                        'project_name': 'Meneapp'
        }

        template = JINJA_ENVIRONMENT.get_template('network.html')
        self.response.write(template.render(template_values))
```

```
import webapp2
import jinja2
import os

JINJA_ENVIRONMENT = jinja2.Environment(
    loader=jinja2.FileSystemLoader(os.path.join(os.path.dirname(__file__), '../view')))

class TopicsHandler(webapp2.RequestHandler):

    def get(self):

        template_values = {
            'title': 'Meneame',
                        'project_name': 'Meneapp'
        }

        template = JINJA_ENVIRONMENT.get_template('topics.html')
        self.response.write(template.render(template_values))
```

```
import webapp2
import jinja2
import os
import hashlib

JINJA_ENVIRONMENT = jinja2.Environment(
    loader=jinja2.FileSystemLoader(os.path.join(os.path.dirname(__file__), '../view')))

class AboutHandler(webapp2.RequestHandler):

    def get(self):

        template_values = {
            'title': 'Meneame',
                        'project_name': 'Meneapp',
                        'albert_hash': hashlib.md5('albertfdp@gmail.com').hexdigest(),
                        'ferdinando_hash': hashlib.md5('ferdinando.papale@gmail.com').hexdi
gest(),
                        'jose_hash': hashlib.md5('th0rg4l@gmail.com').hexdigest()
        }

        template = JINJA_ENVIRONMENT.get_template('about.html')
        self.response.write(template.render(template_values))
```

```
import webapp2
import jinja2
import os

JINJA_ENVIRONMENT = jinja2.Environment(
    loader=jinja2.FileSystemLoader(os.path.join(os.path.dirname(__file__), '../view')))

class SentimentHandler(webapp2.RequestHandler):

    def get(self):

        template_values = {
            'title': 'Meneame',
                        'project_name': 'Meneapp'
        }

        template = JINJA_ENVIRONMENT.get_template('sentiment.html')
        self.response.write(template.render(template_values))
```

```
import webapp2
import jinja2
import os

JINJA_ENVIRONMENT = jinja2.Environment(
    loader=jinja2.FileSystemLoader(os.path.join(os.path.dirname(__file__), '../view')))

class MainHandler(webapp2.RequestHandler):

    def get(self):

        template_values = {
            'title': 'Meneame',
                        'project_name': 'Meneapp'
        }

        template = JINJA_ENVIRONMENT.get_template('index.html')
        self.response.write(template.render(template_values))
```

```
#!/usr/bin/env python
#
# Copyright 2007 Google Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
import webapp2
import fix_path
from control import main, time, about, sentiment, network, topics

app = webapp2.WSGIApplication([
    ('/', main.MainHandler),
    ('/index', main.MainHandler),
        ('/time', time.TimeHandler),
        ('/sentiment', sentiment.SentimentHandler),
        ('/network', network.NetworkHandler),
        ('/topics', topics.TopicsHandler),
        ('/about', about.AboutHandler)
], debug=True)
```