Correlation between weather and sentiment analysis on Twitter

Alex Pellegrini (s132199) & Woody Rousseau (s131642)

***Abstract*—We describe a webservice application which provides several means of visualization of the correlation between the weather and sentiment analysis results, based on data mined tweets which include localization information. We find that only taking into account these two aspects is not enough to obtain a high correlation.**

I. Introduction

Many studies such as [1] aimed at establishing a correlation between the weather and the mood or other human cognition aspects. It is now well known that many more factors need to be taken into account in order to obtain results corroborating that there is indeed a correlation: the time spent outside, and personal life events which tend to have a much larger impact on the mood than the weather. This is also due to the fact that these studies were unable to acquire enough data to reduce the importance of those personal factors. With that flaw in mind, we focus in our study on data mining techniques to establish such a correlation, by gathering tweets that contain localization information, by finding weather information on that localization and by combining it with a sentiment analysis of the tweet. The data mining tasks are performed using Python and two different APIs: Twitter's and OpenWeatherMap's. The presentation of our results is also made using Python, and more specifically a Flask microframework powered webservice. Those two main tasks run in a parallel fashion using threads.

II. Methods

A. Webserver

The webservice relies on Flask microframework, defining several routes: one for each visualization methods available, one for a homepage from which all those methods can be accessed, as well as HTTP error handlers (404 and 500). The code for running the webserver is included in the `server.py`, the script that needs to be launched to run the full application.

B. Visualization Methods

Three visualization methods are available, and on each page, a start button and a stop button are made available to start and stop the data mining of tweets.

- The route `list_data` displays a table containing all acquired tweets as well as their analysis results and updated using websockets powered by gevent-socketio as they are data mined. Using websockets allows the server to let the client know that some of the content needs to be updated (in this case, new acquired tweets), triggering an update.
- The route `plot` displays a scatter plot with all tweets, using NumPy and matplotlib and our `plotting.py` module. The x-axis correspond to the sentiment value (continuous) and the y-axis to the weather value (discrete), both scaled to be between 0.0 and 1.0. Because we have discrete weather values, a mean sentiment value is computed for each of these weather values. Polynomial (degree 1) curve fitting is then applied to these data points and its result is plotted as well. The image can be updated by clicking on a button, which triggers an Asynchronous JavaScript and XML (AJAX) request, requesting an update from the server.
- The route `map` displays a map powered by Google Map's Javascript API and centered around the United States. A weather layer as well as a heatmap layer based on a simple correlation score[1] is added to the map. Objects added to the layer contain a latitude, longitude and a weight information corresponding to the correlation score. Obviously, only tweets gathered since the start button was first clicked are displayed on the heatmap, in order not to include data coming from a period with a different weather. New tweets are added to the heatmap again using websockets.

C. Data Mining

Data mining tasks are performed on a separate thread and use the *TweetWeather* class from `tweetweather.py`, in order to keep everything packed in a single Python script while being able to run the webserver and data mine tweets at the same time. The thread is flagged as a "daemon thread" to allow keyboard interruptions to kill the process. For most of the used APIs, app credentials are provided so that the program runs "out of the box", even though they should in principle remain personal. All tweets are saved in a sqlite database.

- Tweepy allows an easy Python access to Twitter's API: the initial request only gathers tweets in English, and they are then filtered to remove tweets that do not include localization. When the rate limit is exceeded, the thread is paused and resumes fifteen minutes later, following Twitter's policy.
- OpenWeatherMap is used to give a weather score to the acquired localization of a tweet. A manually created dictionary maps weather icons describing the weather to

[1] $\text{score} = |\text{weatherValue} - \text{sentimentValue}|$

a value between 0.0 and 1.0, ranging from a thunderstorm (0.0) to clear skies (1.0), with snow and night weather (without rain) considered neutral (0.5).

D. Sentiment Analysis

For the sentiment analysis we created a large word-values list containing more than 9400 word-value pairs that we load in a single dictionary, so that a value associated to a certain word can be acquired with an $O(1)$ complexity. The values fit in the $[-5, 5]$ range. The list is a merge of the AFINN word-value list and Alex Davies's list which was created by using a learning algorithm and customized to be scaled in the $[-5, 5]$ interval. The goal is to have words with very high and very low values get a big weight within the overall evaluation, as they are supposed to be quite rare. In order to do that, the "probability" of a word is defined as the probability of its value following a gaussian distribution (properly fitted to the range).

The analysis in itself is performed by splitting the body of each gathered tweet, and evaluating each word with the dictionary mentioned above.

The probability function mentioned above is computed using the likelihood function of a gaussian distribution formula where the mean and deviation are $\mu = mean(list.values())$ and $\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i-\mu)^2}{N}}$. The probability is found in the following way:

$$P(w) = \exp\left(-\frac{(value(w)-\mu)^2}{2\sigma^2}\right) \quad (1)$$

The value of each word is computed by getting the actual value from the list and dividing it by the probability above:

$$value(w) = \frac{value(w)}{P(w)} \quad (2)$$

The values in each category (positive, negative and neutral) are summed (pos, neg, neu), and the number of words in each category is also saved $(pos.size(), neg.size(), neu.size())$. Finally, a global sum of the values of all words (whatever their category may be) is computed $(globalSum)$.

With these information, we can "properly" weight each word inside the tweet. The overall value of a tweet is given:

$$value(tweet) = \frac{\sum_{ctg\in(pos,neg,neu)} ctg \times ctg.size()}{globalSum} \quad (3)$$

A `threshold` value was defined in order to limit the values of tweets, which could be very large. This allows the program to scale tweet values in the interval $[0.0, 1.0]$:

$$value(tweet) = \frac{value(tweet) + threshold}{2 \times threshold} \quad (4)$$

NumPy and the standard `math` modules are used to perform these computations.

E. Development process details

The application was developed on two computers running Mac OSX 10.9 with Python 2.7.5 (in order not to run with library compatibility with Python 3 issues). As for text editors, emacs and Sublime Text were used instead of any IDE. git was the chosen revision control system since the beginning of the development process, uploading and maintaining the repository on Bitbucket, a popular code cloud service.

The whole module is quite easy to install, as all dependencies can be installed thanks to the `setup.py` file which is included in the project.

III. Results

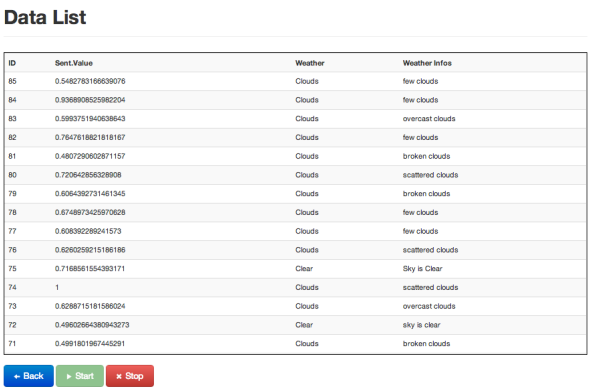

Fig. 1. Websockets updated table with tweets

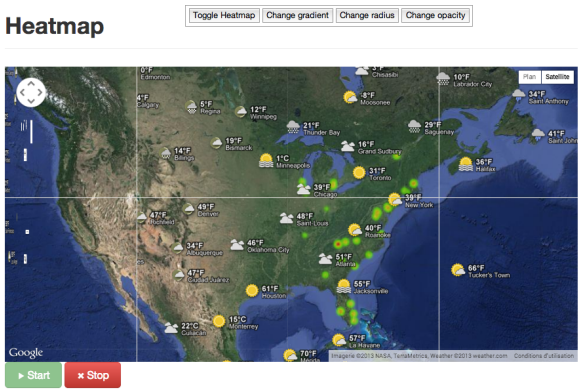

Fig. 2. Websockets updated heatmap with weather layer

Figure 1, Figure 2 and Figure 3 show the three visualization methods. The heatmap does show some points with high weights (high correlation tweets) which have small areas and yet very red centers, and points with low weights (low correlation tweets) which have similar areas but are only colored in green. The scatter plot is useful as it allows a global vision of all data, but also shows that our hypothesis is far from being correct. When many points are acquired, the fitted model is entirely different from the ideal line. This shows some issues with the weather dictionary and the sentiment analysis.

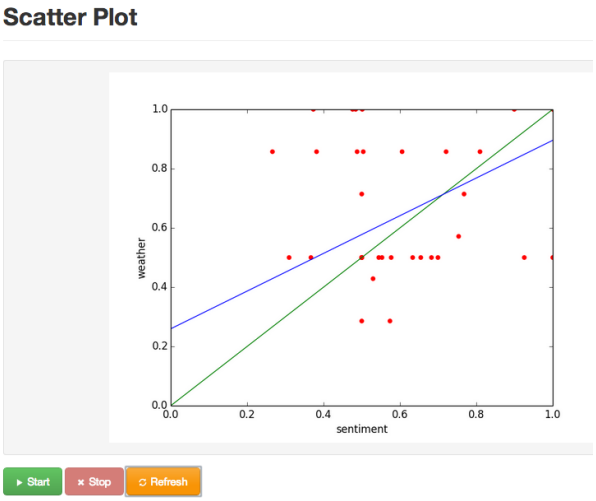

Fig. 3. Scatter plot (red points), ideal line (green), fitted model (blue)

A. Code checking

We used pylint as well as pep8 to check our coding quality. Only minor issues (such as a few too long lines) are left after using these tools. These tools made us aware of some "pythonic" conventions, such as using list comprehensions instead of `map`.

B. Testing

Testing was performed using py.test. Most aspects of our code was not testable as it runs in real time as it gathers tweets. We however focused on the unit testing of our sentiment analysis, on the plotting, as well as on the provided webservice, using Flask's test client. All tests are runnable from the root directory, through the `python setup.py test` command. In total, 12 tests are created and gathered in three different classes, placed in `tests/test_server.py`, `tests/test_analysis.py` and `tests/test_plotting.py`. Coverage was performed using pytest-cov, a py.test plugin with the following command:

```
py.test --cov-report term-missing --cov
server/ tests/
```

It returns a coverage of 99% for `analysis.py`, of 89% for `plotting.py` and of 52% for `server.py` (`tweetweather.py` was not tested, for reasons explained above). The low coverage for `server.py` (and to a lesser extent for `plotting.py`) is due to the fact that some methods require an interaction with the data mining thread, with the database or with the javascript part of the application (websockets).

C. Profiling

Profiling was performed by timing the data mining, analyzing, and the saving of the data and an overall time of the whole process using the `time.py` module, surrounding each function with the following statement:

```
elapsed = time.time()
method.call()
elapsed = time.time() - elapsed
```

This data was saved in a text file using a simple unix redirection (i.e. >>) and analyzed by a new python script (`profile.py`) that for each overall time (i.e. each group of tweets gathered from the twitter stream) computes how much of that time was spent on the above functions. What comes out is that on an overall timing, almost 50% of the time is spent on gathering new tweets using the API:

- gathering: 48.4456012566
- analysis: 1.24803740403
- save: 1.24540493406
- weather: 1.07298472832
- connection: 0.666942267582

save is the time spent to save data into the `sqlite` database, *weather* is the time used to gather weather information, and *connection* is the time spent to connect to the database each time.

IV. Discussion

There are obviously limitations to the application:

- For the data mining, it takes a long time to get a large database (English tweets including localization are scarce). The weather values are also discrete, and they are quite arbitrary. Overall, it would have been a good idea to include more intermediate weather values, and to have many people rate them on a scale from 0.0 to 1.0, taking the means of the results of this study to create the weather dictionary.
- For the Flask application, a good idea would have been to use a cloud service to grow a shared database.
- Obviously, testing and coverage are quite limited by the nature of the application. However, we could have tested the database if we had used Flask's sqlalchemy.
- As for profiling, we did not have the time to implement several versions of our code (for example with several alternative algorithms or data structures) and it would have been interesting to benchmark these different methods.

V. Conclusion

We have built a data mining webservice application. Our original correlation hypothesis was not confirmed, but the application being properly structured, documented and tested, it is thus quite scalable: it could include more complex sentiment analysis or a more complete weather dictionary. The next step would be to establish a state of the art on this correlation, and try to include more factors (time spent outside for instance), although they might be hard to find on Twitter.

References

[1] M. C. Keller, B. L. Fredrickson, O. Ybarra, S. Côté, K. Johnson, J. Mikels, A. Conway, and T. Wager, "A warm heart and a clear head the contingent effects of weather on mood and cognition," *Psychological Science*, vol. 16, no. 9, pp. 724–731, 2005.

Appendix A
Code listings

Listings


```
"""
server.py is a python module which implements a flask http
server needed by the whole web application.
"""

import sys
from flask import Flask, render_template, jsonify, Response, request
from socketio.server import SocketIOServer
from socketio import socketio_manage
from socketio.namespace import BaseNamespace
from pysqlite2 import dbapi2 as db
import gevent
import os
from analysis import Analyzer
from plotting import ScatterPlot
import urllib2
from tweetweather import TweetWeather

PATH = os.path.join('.', os.path.dirname(__file__), '../')
sys.path.append(PATH)

PORT = 5000

app = Flask(__name__)


@app.route('/')
def home():
    """
    The home page is a static page
    allowing the choice of the visualization tool
    """
    return render_template('hello.html')


@app.route('/list_data')
def list_data():
    """
    This page displays all objects in a table
    updated as they are mined.
    The user can start and stop the data mining
    using buttons.
    """
    data = []
    if os.path.exists('data.sqlite'):
        cur = db.connect('data.sqlite').cursor()
        cur.execute('SELECT id,sentimentValue,weather,infos FROM tweets ORDER BY id DESC')
        data = cur.fetchall()
    return render_template('list.html', data=data)


@app.route('/map')
def display_map():
    """
    This page displays a map centered around the United States
    containing a weather layer and a heatmap with arriving objects.
    The user can start and stop the data mining
    using buttons.
    """
    return render_template('map.html')


@app.route("/plot")
def plot():
    """
    This page displays a scatter plot of all gathered tweets
    The x-axis is the sentiment value.
    The y-axis is the weather value.
    The closer points are to the 'identity' line,
```

```
70      the closer they fit our hypothesis
71      """
72      scatter_plot = ScatterPlot(1)
73      scatter_plot.load_data()
74      img_data = scatter_plot.get_image_data()
75      refresh = request.args.get('refresh', 0, type=int)
76      if refresh:
77          return img_data
78      return render_template('plot.html', data=img_data)
79
80
81  @app.route('/socket.io/<path:remaining>')
82  def socketio(request):
83      """
84      This route configures the WebSocket
85      used to let the client know that new objects
86      were mined
87      """
88      try:
89          socketio_manage(request.environ,
90                          {'/new_posts': BaseNamespace},
91                          request)
92      except:
93          app.logger.error("Exception while handling socketio connection",
94                           exc_info=True)
95      return Response()
96
97
98  def check_conn():
99      """
100     Checks whether a working Internet connection is available
101     """
102     try:
103         urllib2.urlopen('http://74.125.228.100')  # Google IP (no DNS lookup)
104         return True
105     except urllib2.URLError:
106         pass
107     return False
108
109
110 @app.route('/start')
111 def start():
112     """
113     Starts the data mining thread if an internet connection is available
114     """
115     if check_conn():
116         tw_thread.start()
117         return jsonify('true')
118     else:
119         tw_thread.connexion_lost("Absent Internet Access")
120         return jsonify('false')
121
122
123 @app.route('/stop')
124 def stop():
125     """
126     Stops the data mining thread
127     """
128     tw_thread.stop()
129     return jsonify('true')
130
131
132 @app.errorhandler(404)
133 def page_not_found(exc):
134     """
135     404 error handler
136     used if a non existant route
137     is requested
138     """
139     return render_template('404.html'), 404
140
141
142 @app.errorhandler(500)
143 def page_not_found(exc):
144     """
145     500 error handler
146     used if there is a server error
147     """
148     return render_template('500.html'), 500
149
150
151 if __name__ == '__main__':
152     analyzer = Analyzer()
153     server = SocketIOServer(('', PORT), app, resource="socket.io")
154     tw_thread = TweetWeather(server, analyzer, name="Tweet-Weather-Thread")
155     tw_thread.daemon = True
```

```
    gevent.spawn(tw_thread.new_post, server)
    gevent.spawn(tw_thread.connexion_lost, server)
    print "Application Started: http://localhost:5000"
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        tw_thread.stop()
        server.stop()
        sys.exit()
```

```
"""Perform the data mining tasks through Twitter and OpenWeatherMap's APIs"""

import threading
from pysqlite2 import dbapi2 as db
import simplejson as jsn
import urllib
import tweepy
import os
import time

# secret twitter app credentials
CONSUMER_KEY = "Zlf11aZTxrnydXBMZfeA"
CONSUMER_SECRET = "rNSask1WRb8mLbbzTZo6vAHB27EwNRmy4AA5c3G04"
ACCESS_KEY = "1889545957-BFTycJVNsAgtlfdKbalV1rwTJqoGGhj0iTxIo6k"
ACCESS_SECRET = "NmfEez4FykN1iGZYUfjYzUvUIksNne2xi6Ovo9Wq00"
WEATHER_APPID = "&APPID=4e04cba42b432a01c4226e186f3d23d2"

'''
Dictionary used to map a weather icon to a
weather "score" from 0 (worse) to 7 (best).
Night weathers (with no rain) as well as snow are considered neutral.
The dictionary is normalized to fit in the
[0,1] range.
'''
WEATHER_DICT = {'13d': 3.5, '11d': 0, '09d': 1,
                '10d': 2, '50d': 3, '04d': 4,
                '03d': 5, '02d': 6, '01d': 7,
                '13n': 3.5, '11n': 0, '09n': 1,
                '10n': 2, '50n': 3, '04n': 3.5,
                '03n': 3.5, '02n': 3.5, '01n': 3.5}
WEATHER_DICT = {k: float(v)/7 for (k, v) in WEATHER_DICT.iteritems()}


class TweetWeather(threading.Thread):
    """
    TweetWeather inherits the Python Thread class.

    Indeed, data mining tasks need to be performed in a separate thread,
    to keep the server running.

    Each new object is stored in the database, and sent to the client
    through a WebSocket.
    """

    def __init__(self, server, analyzer, name=''):
        """
        Checks if there is a working Internet access
        """
        threading.Thread.__init__(self)
        self.name = name
        self.server = server
        self.analyzer = analyzer
        self.root_weather_url = "http://openweathermap.org/data/2.5/weather?lat=%s&lon=%s"
        auth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
        auth.set_access_token(ACCESS_KEY, ACCESS_SECRET)
        self.api = tweepy.API(auth)  # Initialization of the tweeter API
        self.terminated = False

    def new_post(self, *args):
        """
        Sends the new object in a packet to the client through a WebSocket
        """
        pkt = dict(type="event", name="new_post",
                   args=args, endpoint="/new_posts")
        for _, socket in self.server.sockets.iteritems():
            socket.send_packet(pkt)

    def connexion_lost(self, *args):
        """
        Sends a notification packet to let the client know
        that the internet connexion is lost.
        """
        pkt = dict(type="event", name="connexion_lost",
                   args=args, endpoint="/new_posts")
        for _, socket in self.server.sockets.iteritems():
            socket.send_packet(pkt)
```

```
77
78      def run(self):
79          init_database()
80          self.gather_tweets()
81          self.__init__(self.server, self.analyzer, name=self.name)
82
83      def parse_text(self, status):
84          """
85          preforms a very basic sentiment analysis on
86          a single tweet by comparing the most
87          significative words found on the afinn
88          word-value list. Furthermore gets the
89          weather conditions from "http://openweathermap.org"
90          of the location where the tweet has been written
91          and saves the result in the database.
92          """
93          conn = db.connect('data.sqlite')
94          cursor = conn.cursor()
95          score = self.analyzer.analyze(status.text)
96          weather_url = self.root_weather_url % tuple(
97              [str(x) for x in status.coordinates['coordinates']])
98          response = urllib.urlopen(weather_url)
99          try:
100             weather = jsn.load(response)
101         except jsn.JSONDecodeError:
102             print('Program --> Tweet not saved due to invalid weather json')
103         else:
104             if 'weather' in weather.keys():
105                 main = weather['weather'][0]
106                 #print(main['main'], status.text, score)
107
108                 correlation_score = abs(score-WEATHER_DICT[main['icon']])
109                 cursor.execute("INSERT INTO tweets(sentimentValue,"
110                                "weatherValue, correlationScore, weather,"
111                                "latitude, longitude, infos)"
112                                "VALUES(?, ?, ?, ?, ?, ?, ?)",
113                                [score, WEATHER_DICT[main['icon']],
114                                 correlation_score, main['main'],
115                                 status.coordinates['coordinates'][1],
116                                 status.coordinates['coordinates'][0],
117                                 main['description']])
118                 self.new_post(score, main['main'], main['description'],
119                               status.coordinates['coordinates'][1],
120                               status.coordinates['coordinates'][0],
121                               correlation_score)
122                 conn.commit()
123         conn.close()
124
125     def gather_tweets(self):
126         """
127         Performs data mining on tweets which have localization information
128         using the Twitter API (and the tweepy wrapper)
129         """
130         print('Fetching, localizing and analyzing Twitter '
131               'stream data ( could take a while due to '
132               'the few geotagged tweets ) ... ')
133         filtered_tweets = []
134         query = 'lang:en'
135         tweet_pages = tweepy.Cursor(self.api.search,
136                                     q=query, lang='en',
137                                     count=100, result_type="recent",
138                                     include_entities=True).pages()
139         while True:
140             try:
141                 tweets = next(tweet_pages)
142             except tweepy.error.TweepError as exc:
143                 if exc.message[0]['code'] == 88:  # Rate Limit Exceeded
144                     print "Rate Limit Exceeded. Waiting for 15 minutes."
145                     time.sleep(60*15)
146                 tweets = next(tweet_pages)
147             except KeyboardInterrupt:
148                 self.stop()
149
150             filtered_tweets = [tweet for tweet in tweets if tweet.coordinates]
151             if not filtered_tweets:  # No tweet with coordinates on that page
152                 continue
153             for filtered_tweet in filtered_tweets:
154                 self.parse_text(filtered_tweet)
155             if self.terminated:
156                 break
157
158     def stop(self):
159         """
160         Stops the thread
161         """
162         self.terminated = True
```

```
163
164
165 def init_database():
166     """
167     Initializes an sqlite database where evaluated tweets
168     will be saved the table created has an id primary key
169     attribute, a main value for the weather and a short
170     description
171     """
172     if not os.path.exists('data.sqlite'):
173         print("Initializing sqlite database for further analysis ...")
174         conn = db.connect('data.sqlite')
175         cursor = conn.cursor()
176         cursor.execute("CREATE TABLE tweets(" +
177                        "id integer PRIMARY KEY AUTOINCREMENT," +
178                        "sentimentValue real NOT NULL," +
179                        "weatherValue real NOT NULL," +
180                        "correlationScore real NOT NULL," +
181                        "weather VARCHAR(255) NOT NULL," +
182                        "latitude REAL NOT NULL," +
183                        "longitude REAL NOT NULL," +
184                        "infos VARCHAR(255) )")
185         conn.commit()
186         conn.close()
187         print(">> Done <<")
188     else:
189         print("Connecting to sqlite database")
190         print(">> Done <<")
```

```
1  """
2  Module needed by the application to analyze a single tweet
3  and give it a sentiment score
4  """
5
6  import numpy as np
7  import math
8  import urllib
9  import os
10 import re
11
12
13 class Analyzer(object):
14     """
15     Analyzer is used to give a real value to the sentiment found
16     in a tweet text.
17     """
18     def __init__(self):
19         """
20         Analyzer constructor, urls and external data
21         management hard coded.
22         """
23         import zipfile
24         self.comp_list = {}
25         self.afinn_url = "http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/6010/zip/imm6010.zip"
26         self.url = "https://dl.dropbox.com/u/3773091/Twitter%20Sentiment/Twitter%20sentiment%20analysis.zip"
27         urllib.urlretrieve(self.afinn_url, 'word_list.zip')
28         word_list_zip = zipfile.ZipFile('word_list.zip')
29         self.comp_list = {unicode(k, 'utf-8'): int(v)  # reads AFINN list
30                           for (k, v) in [line.split('\t') for line in open(word_list_zip.extract('AFINN/AFINN-111.txt'))]}
31         script_dir = os.path.dirname(__file__)
32         with open(os.path.join(script_dir, 'my_list.txt'), 'r') as comp_file:  # reads larger list
33             for line in comp_file:
34                 data = line.split('\t')
35                 self.comp_list[data[0]] = int(float(data[1].strip())) - 5
36
37         with open(os.path.join(script_dir, 'emoticons.csv'), 'r') as smiles:  # reads emoticons file
38             for line in smiles:
39                 data = line.split('\t')
40                 self.comp_list[data[0]] = int(data[1].strip())
41
42         # clean temporary files on the fly
43         values = np.array(self.comp_list.values())
44         self.mean = np.mean(values)
45         self.deviation = math.sqrt(sum([pow(x - self.mean, 2) for x in values])
46                                    / float(len(values)))
47         os.remove('word_list.zip')
48         os.remove('AFINN/AFINN-111.txt')
49         os.rmdir('AFINN')
50
51     def analyze(self, tweet):
52         """
53         Analyzes the body of a tweet by comparing words to the
54         AFINN word-value list and using a gaussian distribution
55         to compute the weight of each word
56         """
57         emoticons_groups = re.findall(r"([0-9'\&\-\.\/\(\)=:;]+)|((?::|;|=)(?:-)?(?:\)|D|P))|(<3)", tweet)
```

```
        emoticons = [x[0] for x in emoticons_groups if x[0] != '']  # we have three groups in our regexp so we need to check everyone c
        emoticons.extend([x[1] for x in emoticons_groups if x[1] != ''])
        emoticons.extend([x[2] for x in emoticons_groups if x[2] != ''])
        data = [self.comp_list.get(word, 0) for word in tweet.split('␣')]
        data.extend([self.comp_list.get(e, 0) for e in emoticons])
        ctg_count = {'positive': 0, 'negative': 0, 'neutral': 0}  # dict containing the number of positive negative and neutral words i
        ctg_total = {'positive': 0.0, 'negative': 0.0, 'neutral': 0.0}  # dict containing the sum respectively for positive negative ar
        threshold = 22.5

        # computes categories cardinality and global sum of values of each word in tweet
        vals = self.categories_cardinality(tweet, ctg_count)
        # weights each category
        tot_pos, tot_neg, tot_neu = self.weight_categories(data, ctg_total, ctg_count)
        if vals:
            total = (sum([tot_pos, tot_neg, tot_neu]) / vals) + threshold
        else:
            total = (sum([tot_pos, tot_neg, tot_neu])) + threshold
        if total > 2*threshold:
            total = 2*threshold
        elif total < 0:
            total = 0
        else:
            pass
        return total / (2*threshold)

    def weight_categories(self, data, ctg_total, ctg_count):
        """
        Computes the weight in terms of word value of each category
        of words ( positive, negative, neutral )
        """
        for value in data:
            if value > 0:
                ctg_total['positive'] = ctg_total['positive'] + (value / espone(value, self.mean, self.deviation))
            elif value < 0:
                ctg_total['negative'] = ctg_total['negative'] + (value / espone(value, self.mean, self.deviation))
            else:
                ctg_total['neutral'] = ctg_total['neutral'] + espone(value, self.mean, self.deviation)
        tot_pos = ctg_total['positive'] * ctg_count['positive']
        tot_neg = ctg_total['negative'] * ctg_count['negative']
        tot_neu = ctg_total['neutral'] * ctg_count['neutral']
        return tot_pos, tot_neg, tot_neu

    def categories_cardinality(self, tweet, ctg_count):
        """
        Computes the cardinality in terms of number of words belonging
        to each category ( positive, negative, neutral ) returns also
        the sum of the absolute values of each word.
        """
        vals = 0
        for word in (tweet.lower()).split('␣'):
            temp = self.comp_list.get(word, 100)
            if temp > 0 and temp < 100:
                ctg_count['positive'] = ctg_count.get('positive') + 1
                vals = vals + abs(temp)
            elif temp < 0:
                ctg_count['negative'] = ctg_count.get('negative') + 1
                vals = vals + abs(temp)
            elif temp == 0:
                ctg_count['neutral'] = ctg_count.get('neutral') + 1
                vals = vals + abs(temp)
        return vals


def espone(value, mean, deviation):
    """
    The "Likelihood" function using a gaussian probability
    function.
    """
    return math.exp(-(pow((value - mean), 2)) / (2*pow(deviation, 2)))
```

```
"""
Module needed by the application to generate the scatter plot
of the sentiment value with the weather value
"""

import os
from pysqlite2 import dbapi2 as db
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
from cStringIO import StringIO


class ScatterPlot(object):
    """
```

```
    This scatter plots includes the data points,
    an ideal correlation line (in green),
    and a fit of the data available (in blue)
    """

    def __init__(self, pol_order=1):
        self.pol_order = pol_order
        self.x = []
        self.y = []

    def load_data(self):
        """
        Gets all data available
        from the database
        """
        if os.path.exists('data.sqlite'):
            cur = db.connect('data.sqlite').cursor()
            cur.execute('SELECT sentimentValue, weatherValue FROM tweets'
                        ' WHERE sentimentValue <> 0 ORDER BY id DESC')
            all_fetched = cur.fetchall()
            self.x = [point[0] for point in all_fetched]
            self.y = [point[1] for point in all_fetched]
            return True
        return False

    def set_data(self, x, y):
        """
        Sets x and y from given Lists
        Used for testing purposes
        """
        self.x = list(x)
        self.y = list(y)

    def get_fit_function(self):
        """
        Fits a model following
        the polynomial order given to the class
        """
        mean_x = []
        mean_y = []
        for i in range(0, 8):
            indices = [ind for ind, val in enumerate(self.y) if val == float(i)/7]
            if indices:
                mean_y.append(float(i)/7)
                mean_x.append(np.mean([self.x[j] for j in indices]))
        try:
            fit_fn = np.poly1d(np.polyfit(mean_x, mean_y, self.pol_order))
        except TypeError:  # Empty Lists
            return None
        return fit_fn

    def get_image_data(self):
        """
        Prepares the figure and returns the data
        formatted in a base64 encoded String
        """
        fig = plt.figure()
        axis = fig.add_subplot(1, 1, 1)
        xs = np.linspace(0, 1, 8)
        axis.set_xlim([0, 1])
        axis.set_ylim([0, 1])

        axis.plot(xs, xs,
                  label='Perfect Correlation', color='green')
        fit_fn = self.get_fit_function()
        if (fit_fn):
            axis.plot(xs, fit_fn(xs),
                      label='Observed Correlation', color='blue')
        axis.scatter(self.x, self.y,
                     label='Data Points', color='red')
        plt.xlabel('sentiment')
        plt.ylabel('weather')
        str_io = StringIO()
        fig.savefig(str_io, format='png')
        plt.close("all")
        return str_io.getvalue().encode('base64')
```

```
import os
import sys
MY_PATH = os.path.dirname(os.path.abspath(__file__))
sys.path.insert(0, MY_PATH + '/../server')
import server


class TestFlaskServer(object):
    """
```

```
    Testing a part of the Flask webserver
    """

    def setup(self):
        """
        Called at the beginning of the test module
        Configures the Flask Test Client
        """
        server.app.config['TESTING'] = True
        self.app = server.app.test_client()

    def test_http_routes(self):
        """
        Tests that all routes deliver the
        pages without errors
        """
        response = self.app.get('/')
        assert response.status_code == 200
        response = self.app.get('/list_data')
        assert response.status_code == 200
        response = self.app.get('/map')
        assert response.status_code == 200
        response = self.app.get('/plot')
        assert response.status_code == 200

    def test_error_handlers(self):
        """
        Testing a random URL to
        catch a 404 HTTP errors
        """
        response = self.app.get('/randomurl')
        assert response.status_code == 404

    def test_plot(self):
        """
        Testing the reception of the image
        data when an ajax request is sent
        """
        response = self.app.get('/plot?refresh=1')
        assert len(response.data) > 0
```

```
import os
import sys
MY_PATH = os.path.dirname(os.path.abspath(__file__))
sys.path.insert(0, MY_PATH + '/../server')
from analysis import Analyzer


class TestAnalysis(object):
    """
    Testing the analysis module
    which deals with sentiment analysis
    """

    def test_espone(self):
        """
        Testing normal distribution
        value for 0 mean and 1 variance
        """
        from analysis import espone
        assert espone(0, 0, 1) == 1.0

    def test_analyze_empty(self):
        """
        Testing empty tweets
        and tweets including words not
        in the dictionary
        """
        ana = Analyzer()
        assert ana.analyze("") == 0.5
        assert ana.analyze("hzoehfsdl") == 0.5

    def test_analyze_bounds(self):
        """
        Testing the bounds of the tweets values
        """
        ana = Analyzer()
        assert ana.analyze("this␣is␣a␣test␣neutral␣tweet") <= 1.0
        assert ana.analyze("this␣is␣a␣test␣neutral␣tweet") >= 0.0

    def test_analyze_judgement(self):
        """
        Testing the proper judgement of the sentiment analysis:
        * positive and negative
        * best and worse tweet values
        """
```

```
        ana = Analyzer()
        assert ana.analyze(":)") > 0.5 and ana.analyze(":'(") < 0.5
        assert ana.analyze("yahoo yahoo yahoo") == 1.0
        assert ana.analyze("zzz zzz zzz zzz zzz") == 0.0

    def test_analyze_judgement_weight(self):
        """
        Testing the value order
        of arbitrary tweets
        """
        ana = Analyzer()
        assert ana.analyze("i am so happy, great day :D") > ana.analyze("i am so happy :D")
        assert ana.analyze("so sad, feeling depressed :'(") < ana.analyze("so depressed :'(")

    def test_categories_cardinality(self):
        """
        Testing the cardinality of the different
        categorie sums (positive, negative, neutral)
        """
        ana = Analyzer()
        ctg_count = {'positive': 0, 'negative': 0, 'neutral': 0}
        text = 'great day today lol ;) but still have to work'
        assert ana.categories_cardinality(text, ctg_count) == 15
        assert ctg_count['positive'] == 4  # great day lol ;)
        assert ctg_count['neutral'] == 1  # today
        assert ctg_count['negative'] == 2  # work still

    def test_categories_weight(self):
        """
        Testing the weights of the different
        categorie sums (positive, negative, neutral)
        """
        ana = Analyzer()
        ctg_total = {'positive': 0.0, 'negative': 0.0, 'neutral': 0.0}
        ctg_count = {'positive': 4, 'negative': 2, 'neutral': 1}
        data = [2, 3, 0, 2, 2, 0, -4, 0, 0, -2, 2]
        tot_pos, tot_neg, tot_neu = ana.weight_categories(data, ctg_total, ctg_count)
        assert (tot_pos, tot_neg, tot_neu) == (99.47646509317096, -49.392885301738836, 3.9750077625545726)
```

```
import os
import sys
import pytest
import random
MY_PATH = os.path.dirname(os.path.abspath(__file__))
sys.path.insert(0, MY_PATH + '/../server')
from plotting import ScatterPlot


class TestPlotting(object):
    """
    Testing the plotting module
    Since it requires a database,
    this part is not tested and
    fake data is inserted instead
    """

    def test_plotting_empty(self):
        """
        Testing the fit function
        and the image data when no data is available
        or when weather data are random
        and thus not among the allowed discrete values
        """
        plot = ScatterPlot(1)
        img_data = plot.get_image_data()
        assert not(plot.get_fit_function())
        plot.set_data([random.random(), random.random()], [random.random(), random.random()])
        assert not(plot.get_fit_function())
        assert img_data

    def test_plotting(self):
        """
        Testing the fit function
        and the image data when data are available
        """
        plot = ScatterPlot(1)
        plot.set_data([0, 1], [0, 1])
        img_data = plot.get_image_data()
        assert plot.get_fit_function()
        assert img_data
```

```
"""
profile.py performs the profiling of the different
sections of the application
"""
```

```
5
6   times = {
7       'gathering': 0.0,
8       'analysis': 0.0,
9       'push': 0.0,
10      'save': 0.0,
11      'weather': 0.0,
12      'connection': 0.0,
13      'overall': 0.0
14      }
15
16  total = {
17      'gathering': 0.0,
18      'analysis': 0.0,
19      'push': 0.0,
20      'save': 0.0,
21      'weather': 0.0,
22      'connection': 0.0,
23      'overall': 0.0
24      }
25
26
27  def run():
28      for key, _ in times.items():
29          times[key] = times[key] / times['overall']
30          total[key] = total[key] + times[key]
31
32
33  def init_times():
34      times = {
35          'gathering': 0.0,
36          'analysis': 0.0,
37          'push': 0.0,
38          'save': 0.0,
39          'weather': 0.0,
40          'connection': 0.0,
41          'overall': 0.0
42          }
43
44
45  def compute_overall():
46      for k in total.keys():
47          if k != 'overall':
48              total[k] = total[k] / total['overall']
49
50
51  def main():
52      with open('times.txt', 'r') as times_file:
53          for line in times_file:
54              data = line.split('␣:␣')
55              times[data[0]] = times.get(data[0], 0.0) + float(data[1])
56              if data[0] == 'overall':
57                  run()
58                  init_times()
59      compute_overall()
60      for k in total.keys():
61          if k != 'overall':
62              print(k+"␣:␣"+str(total[k]*100))
63
64
65  if __name__ == '__main__':
66      main()
```