Speciale

Fra Hades
of
Doom

Bjarke Mønsted

Pretentious
quote.

- Famous Person, Born-Died

Fra Hades
of
Doom

Author	My Name
Advisor	His Name
Co-Advisor	Her Name

Ting, som KU siger der skal stå her

CMOL

Center for Models of Life

Submitted to the University of Great Justice
June 14, 2014

Acknowledgements

Thank you! Thank you all!

Contents

English abstract

WORDS! SOOOOO MANY WORDS!

Dansk sammenfatning

ORD! SAAAAAAAAAA MANGE ORD

Part I

SOCIAL FABRIC PROJECT

CHAPTER 1

Phone Activity and Quantitative Data Extraction

The main objective of part I of this thesis is to investigate behavioural patterns in the phone activities of the participants in the Social Fabric Project, and to predict various traits of the users based only on their phone logs. Throughout the part, I'll provide brief examples of usage for the software I've written to process the large amount of data available and to apply various prediction schemes to it, while the source code itself is included in the appendix.

My first objective was to investigate how various phone activities correlate with each other temporally, i.e. how a given user's probability for e.g. receiving a call increases or decreases around other activities such as moving around physically. This is the topic of section 1.1.

Next, I set out to replicate some recent research resultsclaiming that people's phone activities predict certain psychological traits. In the most general terms, then, the task consists of predicting a collection of numbers or labels denoted Y based on a set of corresponding data points X. The topic of section 1.2 is the extraction of the many-dimensional data points or *feature vectors* X from the phone logs of the participants, while section 1.3 gives a brief description of the psychological traits Y. Finally, section 1.4 derives an often used linear classification method known as Linear Discriminant Analysis or Fisher's Discriminant and provides a discussion of why it fails for the present dataset, which serves to motivate the more sophisticated prediction schemes introduced in chapter 2.

kilder!!!

1.1 Temporal Correlations in Activity

One category of interesting quantities is the predictability of mobile phone behaviour from recorded behaviour at different times, i.e. the influence of certain events deduced from a user's Bluetooth, GPS or call log data on the tendency of some event to happen in the near past or future. A simple example would be to determine how much placing or receiving a call increases or decreases the probability of a user placing or receiving another call in the period following the first call.

This analysis was performed by comparing an 'activity' signal with a 'background' signal in the following fashion: For each user, the time period around each call is sliced into bins and the each of the remaining calls placed in the bin corresponding to the time between the two calls. Once divided by the total number of calls for the user, this is the activity signal. The background is obtained in a similar fashion but comparing each call in a given user's file with calls in the remaining users' files.

This involves repeatedly binning the time around certain events and then determining in which bin to place other events; a situation in which confusion may arise easily and errors may be hard to identify. To accommodate this, I started out by writing a custom array class designed to greatly simplify the binning procedure. This class called features the following:

- Methods to bin the time around a given event and determine determine which bin a given event falls into. This is useful to implement in the class itself as one then avoids having to continually worry about which bin an event fits into, and as it ensures that bin placement errors can only arise in one small piece of code which can then be tested rigorously.

- Attributes that keep track of the number of events that didn't fit into any bin, and of the current centre of the array, which can then be manipulated to move the array and a method to use this to return a normalized list of the bins.

In short, the binarray can be visualized as a collection of enumerated buckets that can be moved so as to center it on some event and then let other events 'drip' into the buckets. The code for this class is included in A.1.1. In general, objects can be converted to byte streams and stored using Python's pickle module, but as that tends to be both slow and insecure, I generally used json to save my objects. This poses a slight problem as some data types, such as tuples, and custom classes in general are not json serializable. I got around this by writing some recursive helper methods to help store the relevant information about arbitrary nested combinations of some such objects and to help reconstruct said objects again. These are also included in section A.1.1. As an example of usage, the following code constructs a Binarray, centers it around the present time, and generates a number of timestamps which are then placed in the event. It is then saved to a file using the helper method previously described.

```
from time import time
from random import randint
#Create Binarray with interval +/- one hour and bin size ten minutes.
ba = Binarray(interval = 60*60, bin_size = 10*60)
#Center it on the present
now = int(time())
ba.center = now
#Generate some timestamps around the present
new_times = [now + randint(-60*60, 60*60) for _ in xrange(100)]
for tt in new_times:
    ba.place_event(tt)

#Save it
with open('filename.sig', 'w') as f:
    dump(ba, f)
```

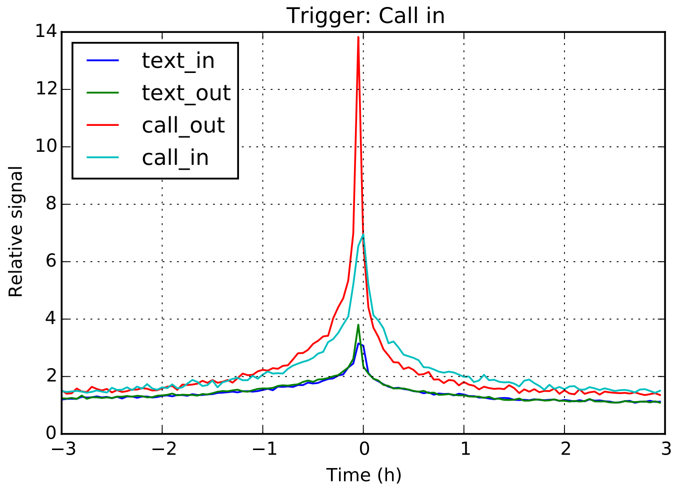
This data will be visualized by plotting the relative signal from the activity of some event, such as in- or outgoing calls or texts, over the background (simply A/B) around another type of event hypothesized to trigger the activity.

1.1.1 Influence of phone calls

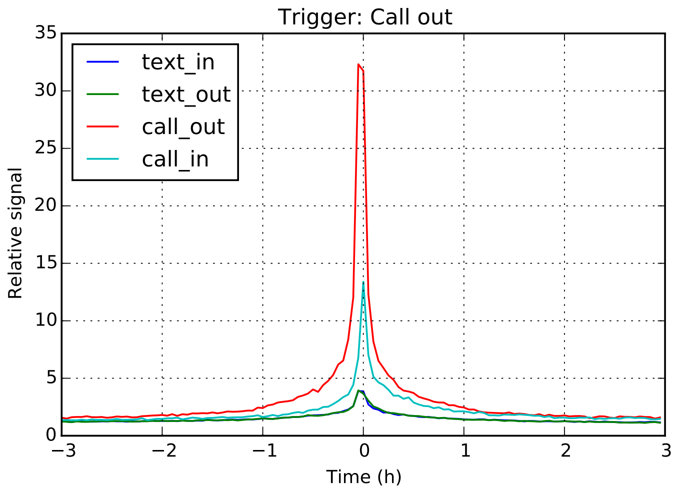
I first investigated the effects of incoming and outgoing calls as triggers for other phone activities. The call logs were stored in a format where each line represents a hashmap with quantities such as call time or call duration mapping to their corresponding value. Below is an example of one such line, were any personal data have been replaced by a random number or hexadecimal string of equal length.


```
{
    "timestamp": 6335212287,
    "number": "c4bdd708b1d7b82e349780ee1e7875caa600c579",
    "user": "ea42a1dbe422f83b0178d158f154f4",
    "duration": 483,
    "type": 2,
    "id": 45687
}
```


the text logs are similar except for the missing duration entry. Computing the relative signal in Binarrays centered on each incoming and outgoing



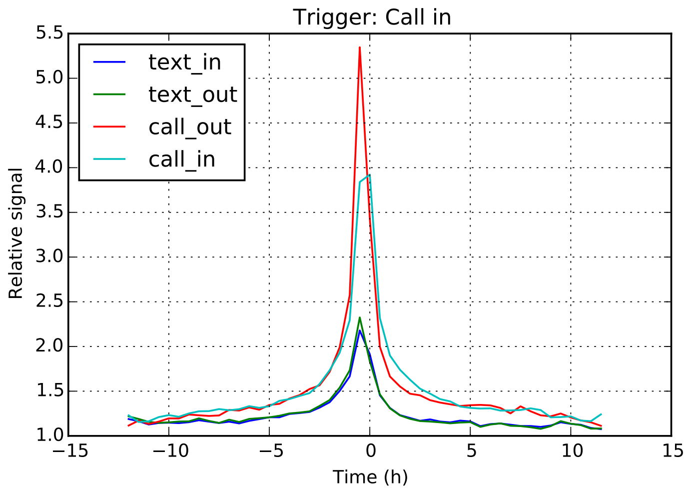
(a) Relative activity of events triggered by incoming calls.



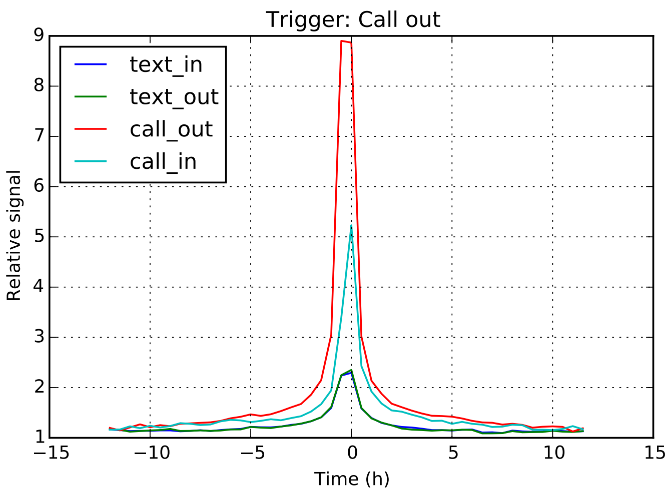
(b) Relative activity of events triggered by outgoing calls.

Figure 1.1: Comparison of the increased activity caused by incoming and outgoing calls over an interval of ± 3 hours around an event with bins of three minutes.

call using bin sizes of three and thirty minutes resulted in the plots shown in figures 1.1 and 1.2, respectively. As the figures clearly show, the all four activities increase significantly for the average user around incoming and outgoing calls.



(a) Relative activity of events triggered by incoming calls.



(b) Relative activity of events triggered by outgoing calls.

Figure 1.2: Comparison of the increased activity caused by incoming and outgoing calls over an interval of ± 12 hours around an event with bins of thirty minutes.

1.1.2 Influence of GPS activity

The raw format of the users' GPS logs looks similar to those of the call and text logs:


```
{
    "timestamp": 8058876274,
    "lon": 6.45051654,
    "user": "0c28e8f4ad9619bca1e5ea4167e10a",
    "provider": "gps",
    "lat": 28.20527041,
    "id": 6429902,
    "accuracy": 39.4
}
```


An analysis similar to that of described in section 1.1.1 was carried out using GPS phone data as triggers. I chose to define a user as being 'active' if they travelled at an average speed of 0.5 m/s between two consecutive GPS log entries, while discarding measurements closely following each other. The reason for this is that the uncertainty on the location measurements could yield false measurements of high average speeds when the measurements are not temporally separated. A lot of the measurements turned out to be grouped somewhat tightly - for instance, approximately 80% of the time intervals were below 100 s. This occurs because the Social Fabric data not only actively records its users' locations with some set interval, but also passively records the location when another app requests it, so when users spend time on apps that need to continually update their position such as Google Maps, a location log entry is written every second. The distribution of intervals between consecutive GPS measurements is shown in figure 1.3. A typical uncertainty on civilian GPS devices is at most 100 m[21], so because I choose to consider a user active if they travel at a mean speed of 0.5 m/s, and based on the time spacings shown in figure 1.3, I chose to discard measurements separated by less than 500 s.

An analysis like that of section 1.1.1 reveals that a user's phone activity is significantly increased around times when they are on the move, as shown in figure 1.4. Note the asymmetry of the signal, especially visible in figure 1.4(a). After a measurement of a user being active, the signal dies off about two and a half hours into the future, whereas it persists much longer into the past. Concretely, this means that people's phone activity (tendency to call or text) becomes uncorrelated with their physical activity after roughly two and a half hours, whereas their tendency to move around is increased for much longer time after calling or texting.

The relative signal in figure 1.4(b) appears to be increasing at around ±24 h, which would seem reasonable assuming people have slightly

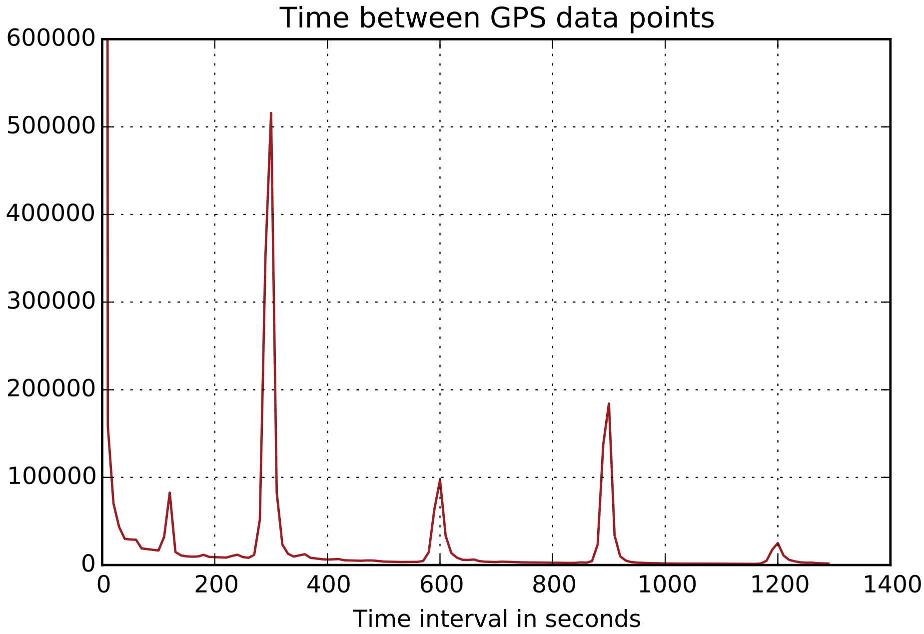
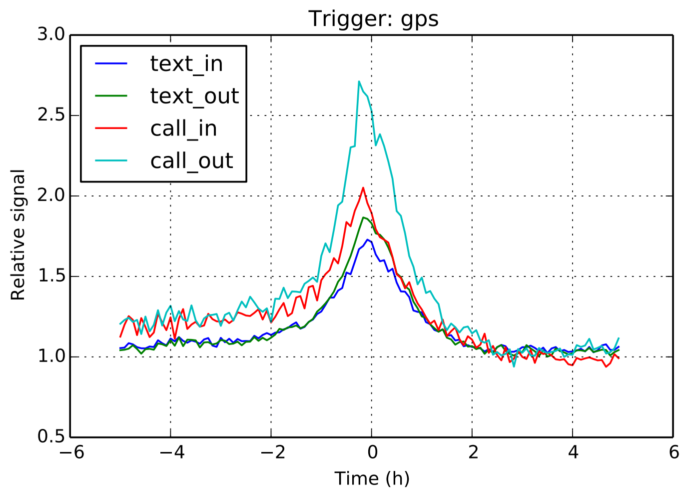
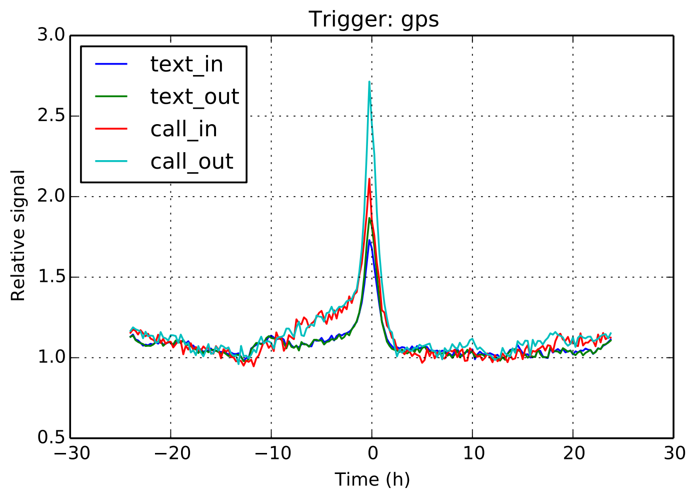


Figure 1.3: Plot of typical temporal spacings between consecutive GPS measurements.

different sleep schedules - if a person is on the move and hence more likely to place a call at time $t = 0$, they're slightly more likely than the general user to be on the move around $t = \pm 24$ h. Figure 1.5 shows the same signal extended to ± 36 h where slight bumps are visible 24 hours before and after activity.



(a) Interval: 5 hours. Bin size: 5 minutes.



(b) Interval: 24 hours. Bin size: 15 minutes.

Figure 1.4: Relative increase of activities triggered by GPS activity.

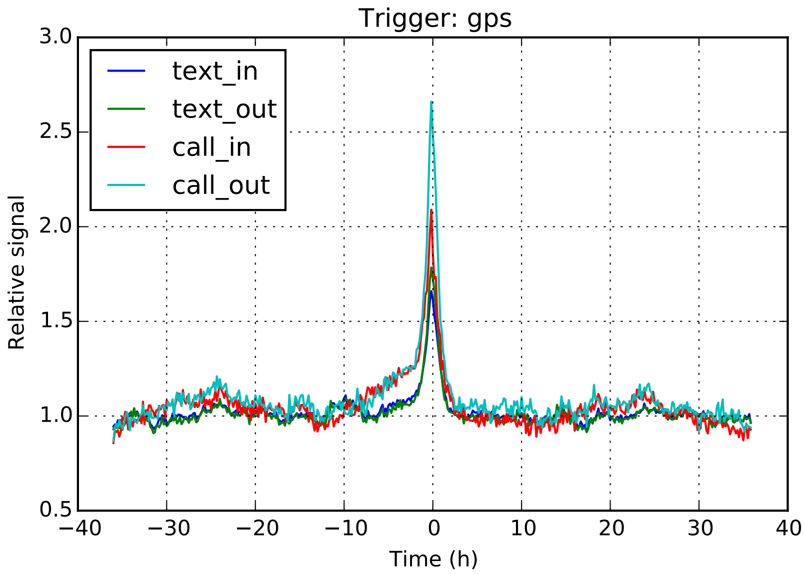


Figure 1.5: GPS-triggered activity increase over an interval of 36 hours using a bin size of 10 minutes.

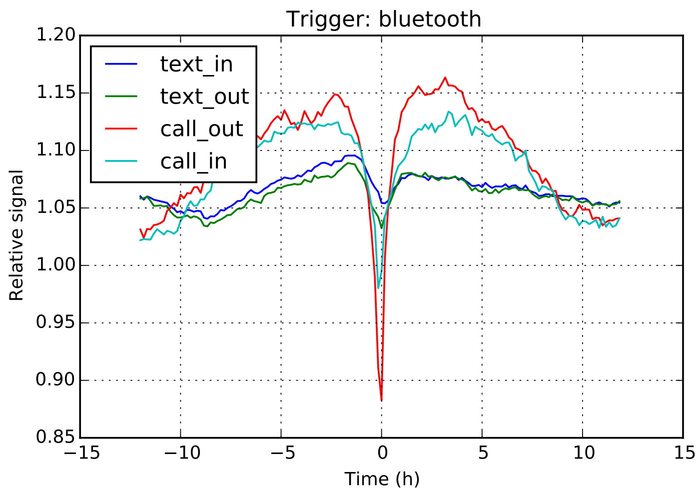
1.1.3 Influence of Bluetooth signal

The following is a randomized entry in a user's bluetooth log.

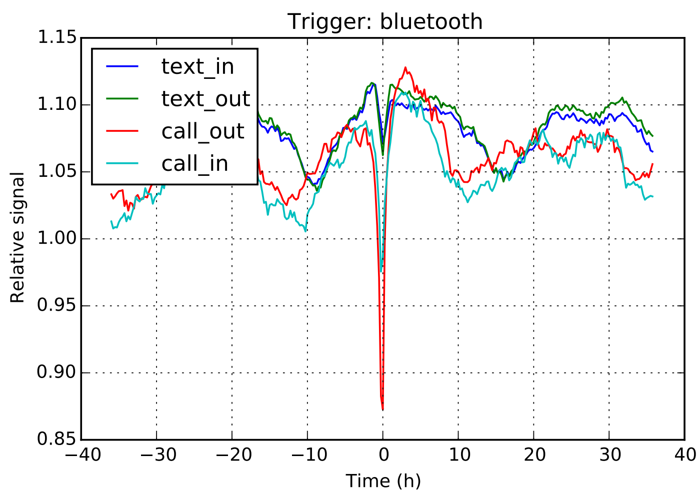

```
{
    "name": "d5306a3672b7a0b8f9696d294ec4b731",
    "timestamp": 6870156680,
    "bt_mac": "1f158ae269d69efa5bb4794ee2a0b2dd68bd3a9badfeaf70f258ad3c74b0c09b",
    "class": 1317046,
    "user": "41cdb7ecaaaec3d33391ed063e7fa2",
    "rssi": −76,
    "id": 4139043
}
```


The 'bt_mac' entry is the MAC-adress of the device which the Bluetooth receiver in the user's phone has registered, so it is reasonable to assume several different MAC addresses occur at several consecutive timestamps. I call the number of repeated MAC adresses needed for a user to be considered social the 'social threshold'. Figures 1.6, 1.7 and 1.8 show the increased activity around times when users were considered social with a threshold of 1, 2 and 4 repeated pings.

Contrary to the previous analyses, phone activities decreased somewhat when users were social. As stated, each of these analyses were fairly similar, I've only explicitly included the code used to extract and save

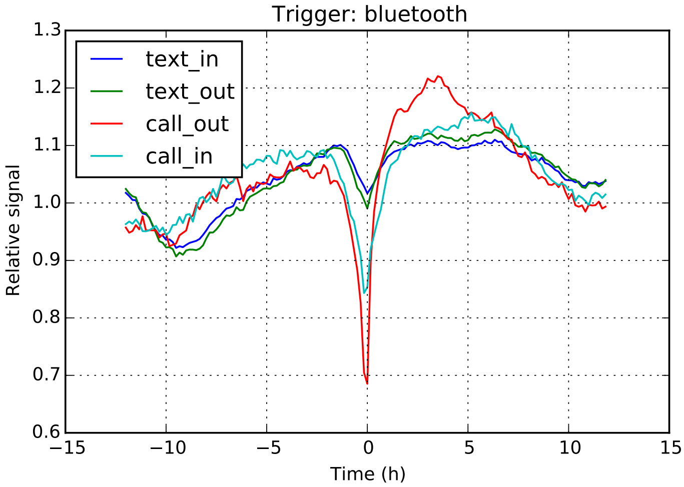


(a) Interval: 12 hours, Bin size: 10 minutes.

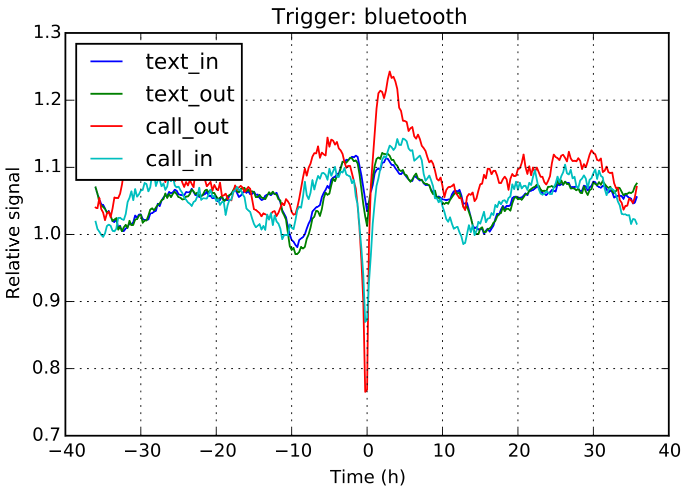


(b) Interval: 36 hours, Bin size: 15 minutes.

Figure 1.6: The effect on phone activity of sociality as measured by the user's Bluetooth signal. The threshold used for being considered social as one repeated signal.

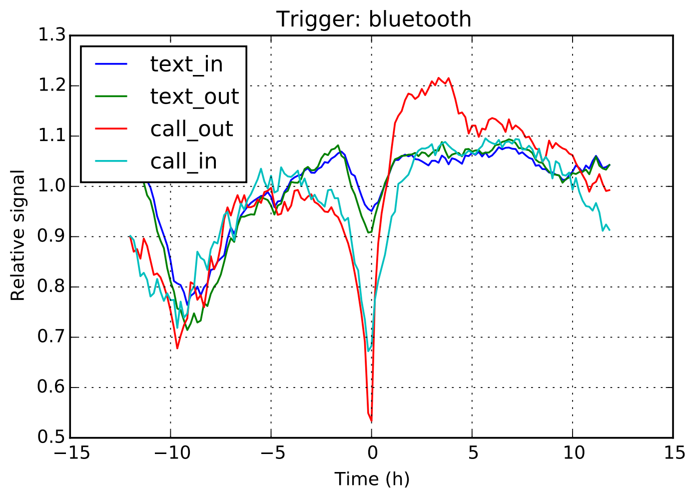


(a) Interval: 12 hours, Bin size: 10 minutes.

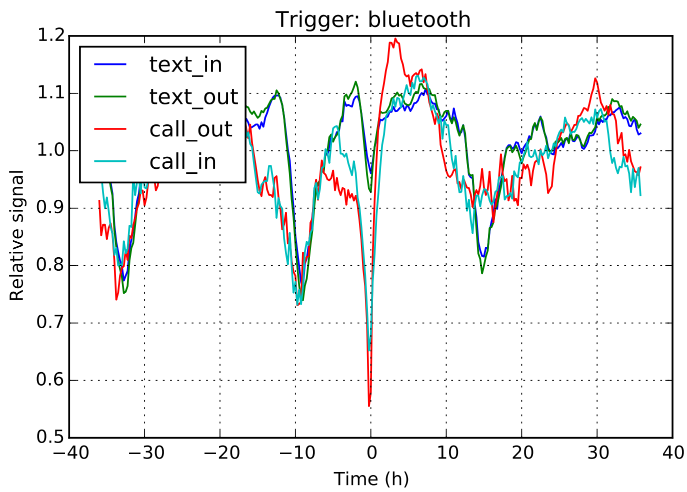


(b) Interval: 36 hours, Bin size: 15 minutes.

Figure 1.7: The effect on phone activity of sociality as measured by the user's Bluetooth signal. The threshold used for being considered social as two repeated signals.



(a) Interval: 12 hours, Bin size: 10 minutes.



(b) Interval: 36 hours, Bin size: 15 minutes.

Figure 1.8: The effect on phone activity of sociality as measured by the user's Bluetooth signal. The threshold used for being considered social as four repeated signals.

Bluetooth data, as well as the code used to load the data and generate figures 1.6 through 1.8. This code is included in section A.1.2.

1.2 Extraction of Input Data

The predictive powers of mobile phone behaviour on the user's psychological profile is currently an area of active research. As part of my thesis work, I have tried to predict the psychological profiles of the SFP participants using various machine learning methods on the available phone logs.

1000 kilder!!!

The software I've written first preprocesses the phone logs to extract various relevant parameters, then collects the parameters and psychological profile scores for each user to serve as input and output, respectively, for the various learning methods. Many of the parameters are chosen following a recent article by de Montjoye et al[6]. The following contains an outline and brief explanation of the extracted parameters.

This section contains a list of the extracted parameters used for psychological profiling along with a brief description of the extraction process when necessary. The preprocessing code is included in section A.1.3.

1.2.1 Simple Call/Text Data

The most straightforward data type is the timestamps from a given user's call/text logs. Six of the parameters used were simply the standard deviation and median of the times between events in the logs for each user's call log, text log, and the combination thereof, excluding time gaps of more than three says on the assumption that it would indicate a user being on vacation or otherwise having a period of telephone inactivity. The entropy S_u of each of the three was also included simply by computing the sum

$$S_u = -\sum_c \frac{n_c}{n_t} \ln_2 \frac{n_c}{n_t}, \tag{1.1}$$

where c denotes a given contact and n_t the total number of interactions, and n_c the number of interactions with the given contact. The number of contacts, i.e. the number of unique phone numbers a given user had contacted by means of calls, texts, and the combination thereof, was also extracted along with the total number of the various kinds of interactions and the contact to interaction ratios. The response rates, defined as the rate of missed calls and incoming texts, respectively, that a given user replied to within an hour, where also determined along with the text latency defined as the median test response time. Finally the percentage calls and texts that were outgoing was determined as well as the fraction of call interactions that took places during the night, defined as between 22-08.

1.2.2 Location Data

A number of parameters based on the spacial dynamics of the user were also extracted. Among these is the radius of gyration, meaning simply the radius of the smallest enclosing circle enclosing all the registered locations of the user on the given day, and the distance travelled per day. I chose to extract the median and standard deviation of each, filtering out the radii that exceeded 500km so as to keep information about long distance travels in the distance parameter and information about travel within a given region in the radius of gyration parameter.

Cluster Analysis

One parameter which has strong links[6] to psychological traits is the number of locations in which the user typically spends time, and the entropy of their visits to that location. Hence, the task at hand is to identify dense clusters of GPS coordinates for each user. This is a typical example of a task which is very intuitive and quickly approximated by humans, but is extremely computationally expensive to solve exactly. Concretely, the problem of finding the optimal division of n data points into K clusters is formulated as minimizing the 'score' defined as

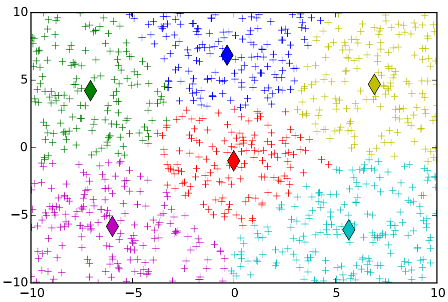
$$S = \sum_{K} \sum_{x_n \in C_k} |x_n - c_k|^2, \tag{1.2}$$

where c_k denotes the centroid of the cluster C_k. Each point x_n is assigned to the cluster corresponding to the nearest centroid. The usual way of approaching this problem is to use Lloyd's algorithm, which consists of initializing the centroids randomly, assigning each point to the cluster corresponding to the centroid which is nearest, then moving each centroid to the center of its points and repeating the last two steps until convergence. As this isn't guaranteed to converge on the global minimum of (1.2), the process can be repeated a number of times, keeping only the result with the lowest value of S. I accomplished this by writing a small Python module to perform various variations of Lloyd's algorithm and to produce plots of the resulting clusters. The code is included in section A.1.5.

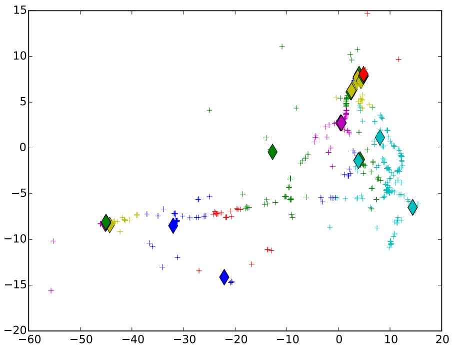
This allows one to implement Lloyd's algorithm and visualize its result easily, as the code allows automatic plotting of the result from the algorithm while automatically selecting different colors for the various clusters. As an example, the following code snippet generates 1000 random points, runs Lloyd's algorithm to determine clusters and saves a plot of the results.

```
points = [[random.uniform(-10,10), random.uniform(-10,10)] for _ in xrange(10**3)]
clusters = lloyds(X = points, K = 6, runs = 1)
draw_clusters(clusters = clusters, filename = 'lloyds_example.pdf')
```

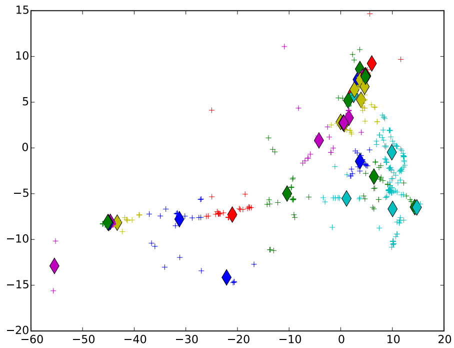
This results in the following visualization:



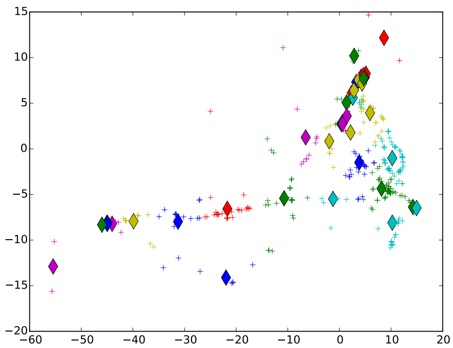
I chose to modify the algorithm slightly on the following basis: Usually, the algorithm takes as its initial centroids a random sample of the data. I'll call this 'sample' initialization. This leads to a greater number of clusters being initialized in the areas with an increased density of data points, meaning that centroids will be highly cluttered at first, 'fighting' over the dense regions of data points then slowly spreading out. A few such iterations are shown in figure 1.9. However, this method is dangerous: The goal is to identify locations in which a user spends much of their time, i.e. in which more than some threshold of their GPS pings originated, and this initialization is likely to 'cut' the more popular locations into several clusters, neither of which contains more data points than the threshold. One example might be the DTU campus, which is a risk of being divided into several locations with too few data points in each, giving the false impression that user doesn't visit the campus that often. To avoid this effect, I implemented another initialization, 'scatter', in which the clusters start out on points select randomly from the entire range of x, y-values in the user's dataset. This turned out to not only solve the problem described above, but also converge much quicker and reach a slightly lower score as define in (1.2). A few such iterations are shown in figure 1.10. The difference in end results for the two methods is exemplified in figure 1.11. While this works great for users who stay in or around Copenhagen, it will cause problems for people who travel a lot. A user who has visited Australia, for instance, will have their initial clusters spread out across



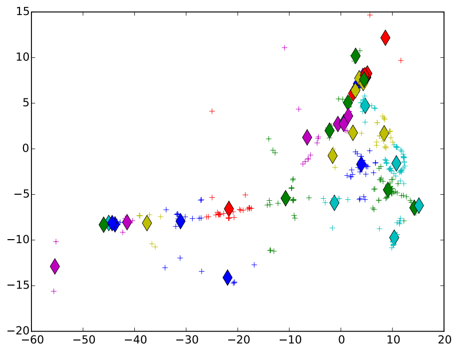
(a)



(b)



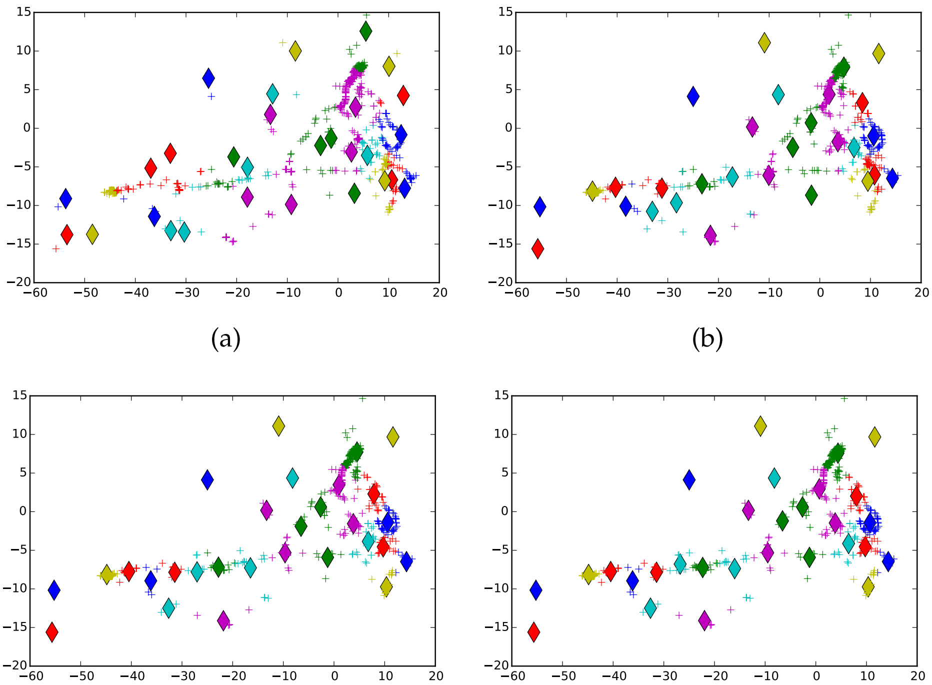
(c)



(d)

Figure 1.9: A few iterations of Lloyd's algorithm using 'sample' initialization. The axes denote the distance in km to some typical location for the user. Note that clusters are initially cluttered, then slowly creep away from the denser regions.

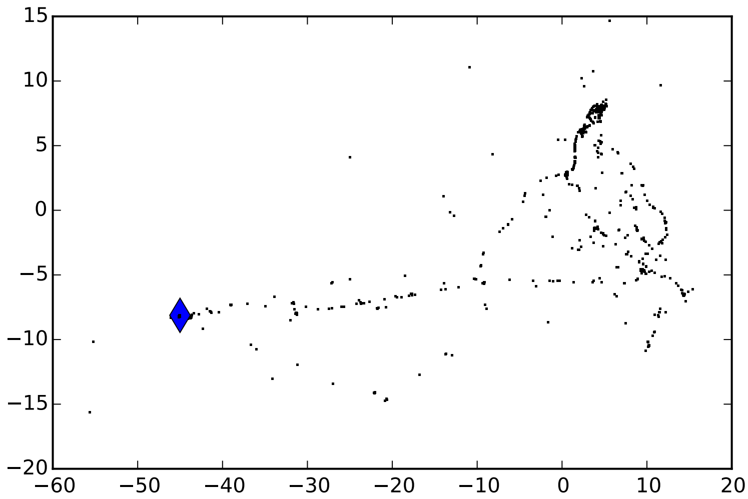
the globe, and it's highly likely that one them will end up representing all of Denmark. I ended up simply running both versions and keeping the result yielding the highest amount of locations.



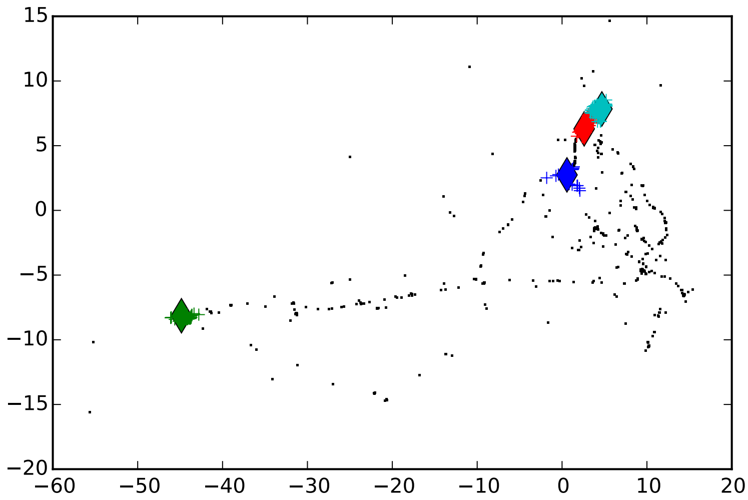
(a) (b)

(c) (d)

Figure 1.10: A few iterations of Lloyd's algorithm using 'scatter' initialization. The axes denote the distance in km to some typical location for the user. Note that clusters are initially randomly spread across the entire range of x, y-values and converge quickly to a local minimum for (1.2).



(a) Sample initialization.



(b) Scattered initialization.

Figure 1.11: Comparison of the final results of the two initialization methods using 100 initial clusters, a threshold of 5% of the data points before a cluster is considered a popular location and running the algorithm 10 times and keeping the best result. Clusters containing more than 5% of the total amount of data points are in color, whereas the remaining points are black dots.

1.2.3 Time Series Analysis

Another interesting aspect to include is what one somewhat qualitatively might call behavioural regularity - some measure of the degree in which a user's phone activities follow a regular pattern. Quantifying this turns out to take a bit of work. First of all, any user's activity would be expected to closely follow the time of day, so the timestamps of each user's outgoing texts and calls are first converted into 'clock times' meaning simply the time a regular clock in Copenhagen's time zone would display at the given time. This process is fairly painless when using e.g. the UTC time standard, which does not observe daylight saving time (DST), but some subtleties arise in countries that do use DST, as this makes the map from Unix/epoch time to clock time 'almost bijective' - when changing *away* from DST, two consecutive hours of unix time map to the same clock time period (02:00 to 03:00), whereas that same clock period is skipped when changing *to* DST. The most commonly used Python libraries for datetime arithmetic accommodate this by including a dst boolean in their datetime objects when ambiguity might arise, however I simply mapped the timestamps to clock times and ignored the fact twice a year, one time bin will artificially contain contributions from one hour too many or few. One resulting histogram is shown in figure 1.12.

Tilføj lidt om AR-serier når du har bogen!!!

1.2.4 Facebook Data

Unfortunately, the only available Facebook data was a list of each user's friends, so the only contribution of each user's Facebook log was the number of friends the user had.

1.2.5 Bluetooth Data

I extracted a number of different features from each user's Bluetooth log file. First, I set a threshold for when a given user is considered social, as described in section 1.1.3. I chose to use a threshold of two. I then tried to estimate how much time each user spends in the physical company of others in the following way: for each time stamp in the user's Bluetooth log, I checked if the user was social or not and assumed that this status was the same until the following log entry, unless the delay was more than two hours. The rationale behind this is to avoid skewing the measurements if a user turns off their phone for extended periods of time. Otherwise, e.g. studying with a few friends at DTU, turning off your phone and going on vacation for two weeks would give the false impression that the user

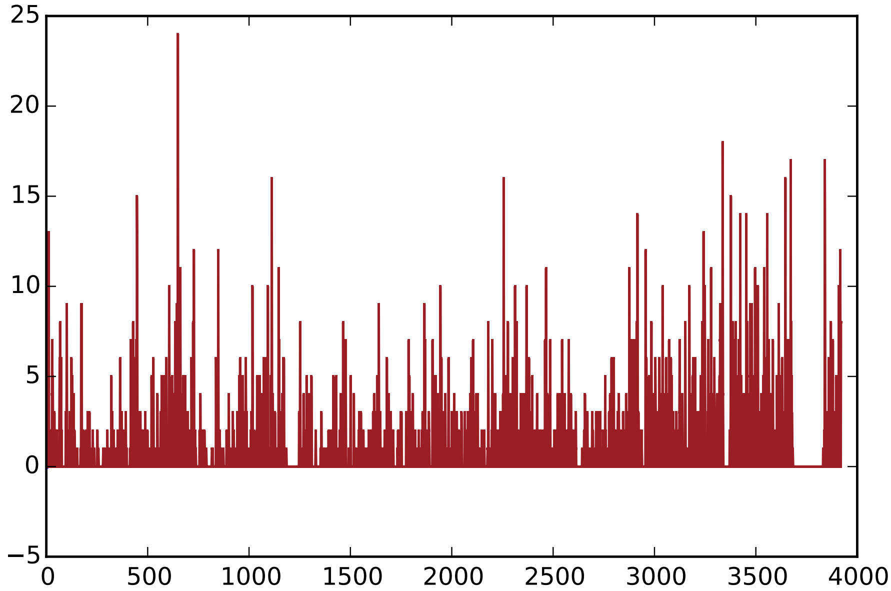


Figure 1.12: Histogram of a user's outgoing calls and texts with a bin size of six hours.

were highly social for a long period of time. I then recorded the fraction of times the user was estimated as being social in this fashion.

Finally, I also wanted some measure of the degrees to which a user's social behaviour follows a pattern. I looked for temporal patterns by fitting AR-series and computing autocorrelation coefficients for each user's social behaviour as described in section 1.2.3. I also chose to compute a 'social entropy' much like (1.1), but weighted by the time the user spends with each acquaintance:

$$E = -\sum_i f_i \ln_2(f_i), \tag{1.3}$$

$$f_i = \frac{\text{time spent with } i}{\sum_j \text{time spent with } j}. \tag{1.4}$$

Note that the denominator of (1.4) is not equal to the total amount of time spent being social, as the contribution from each log entry is multiplied by the number of people present.

1.3 Output Data

The main emphasis of this part of the thesis is on predicting so-called *Big Five* personality traits. This section contains a brief description of those, following[7]. **Extraversion** signifies how extroverted and sociable a person is. People with high extraversion scores are supposed to be more eager to seek the company of others. **Agreeableness** is supposed to be a measure of how sympathetic or cooperative a person is, whereas **conscientiousness** denotes constraint, self discipline, level of organization etc.. **Neuroticism** signify the tendency to experience mood swings, and is complementary to emotional stability. Finally, **Openness**, also called 'openness to experience', or 'inquiring intellect' in earlier works, signifies thoughtfulness, imagination and so on. These five are collectively referred to as the 'big five' or 'OCEAN' after their initials.

I addition to the above, I also had access to a range self-explanatory traits about the participants such as their gender, whether they smoke etc.

1.4 Linear Discriminant Analysis & Gender Prediction

Linear discriminant analysis is basically a dimensionality reduction technique developed by Fisher in 1936 [8] for separating data points into two or more classes. The general idea is to project a collection of data points in n-dimensional variable space, onto the line or hyperplane which maximizes the separation between classes. Representing data points in n-space by vectors denoted x, the objective is to find a vector ω such that separation between the projected data points on it

$$y = \omega^T x \tag{1.5}$$

is maximized.

To break down the derivation of this method, I will first define a convenient distance measure used to optimize the separation between classes, then solve the resulting optimization problem. For clarity, I'll only describe the case of projection of two classes onto one dimension (i.e. using 'line' rather than 'hyperplane' and so on), although the method generalizes easily.

1.4.1 A measure of separation for projected Gaussians

If the projected data points for two classes a and b follow distributions $\mathcal{N}_a$ and $\mathcal{N}_b$, which are standard Gaussians, $\mathcal{N}_i(x) = \mathcal{N}(x; \mu_i, \sigma_i^2)$, the joint probability distribution for the distance between the projections will be the convolution

$$P(x) = \int_{-\infty}^{\infty} \mathcal{N}_a(y) \cdot \mathcal{N}_b(x-y)\,\mathrm{d}y. \tag{1.6}$$

Computing this for a Gaussian distribution,

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{1.7}$$

becomes easier with the convolution theorem, which I'll derive in the following.

Denoting convolution by $*$ and Fourier transforms by

$$\mathcal{F}(f) = \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} f(x) \cdot e^{-i\omega x}\,\mathrm{d}x, \tag{1.8}$$

the convolution theorem is derived as follows:

$$\begin{aligned}
\mathcal{F}(f*g) &= \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} \int_{\mathbb{R}^n} f(y) \cdot g(x-y)\,\mathrm{d}y\, e^{-i\omega x}\,\mathrm{d}\omega, && (1.9)\\
&= \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} f(y) \int_{\mathbb{R}^n} g(x-y) e^{-i\omega x}\,\mathrm{d}y\,\mathrm{d}\omega, && (1.10)\\
&= \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} f(y) \int_{\mathbb{R}^n} g(z) e^{-i\omega(z+y)}\,\mathrm{d}z\,\mathrm{d}\omega, && (1.11)\\
&= \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} f(y) e^{-i\omega y} \int_{\mathbb{R}^n} g(z) e^{-i\omega z}\,\mathrm{d}z\,\mathrm{d}\omega, && (1.12)\\
&\boxed{\mathcal{F}(f*g) = (2\pi)^{n/2} \mathcal{F}(f) \cdot \mathcal{F}(g),} && (1.13)
\end{aligned}$$

where the factor in front of the usual form of the theorem $\mathcal{F}(f*g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$ stems from the convention of using angular frequency in Fourier transforms, as in (1.8), rather than

$$\mathcal{F}(f) = \int_{\mathbb{R}^n} f(x) \cdot e^{-2\pi i \nu x}\,\mathrm{d}x. \tag{1.14}$$

Using this, the convolution of two Gaussians can be calculated as

$$\mathcal{N}_a * \mathcal{N}_b = (2\pi)^{n/2} \mathcal{F}^{-1} \left(\mathcal{F}(\mathcal{N}_a) \cdot \mathcal{F}(\mathcal{N}_b)\right). \tag{1.15}$$

The required Fourier transform can be massaged into a nicer form by displacing the coordinate system and cancelling out terms with odd parity:

$$\begin{aligned}
\mathcal{F}(\mathcal{N}(x)) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \cdot e^{-i\omega x} \,\mathrm{d}x, \\
&= \frac{1}{2\pi\sigma} \int_{-\infty}^{\infty} e^{-\frac{x^2}{2\sigma^2}} e^{-i\omega(x+\mu)} \,\mathrm{d}x, \\
&= \frac{1}{2\pi\sigma} e^{-i\omega\mu} \int_{-\infty}^{\infty} e^{-\frac{x^2}{2\sigma^2}} (\cos(\omega x) + i\sin(\omega x)) \,\mathrm{d}x, \\
&= \underbrace{\frac{1}{2\pi\sigma} e^{-i\omega\mu}}_{a} \underbrace{\int_{-\infty}^{\infty} e^{-\frac{x^2}{2\sigma^2}} \cos(\omega x) \,\mathrm{d}x}_{I(\omega)} . \qquad (1.16)
\end{aligned}$$

Noting that $I(\omega)$ reduces to an ordinary Gaussian integral at $\omega = 0$ so $I(0) = \sqrt{2\pi}\sigma$, this can be solved with a cute application of Feynman's trick:

$$\begin{aligned}
\frac{\partial I}{\partial \omega} &= -\int_{-\infty}^{\infty} x e^{-\frac{x^2}{2\sigma^2}} \sin(\omega x) \,\mathrm{d}x, \\
&= \int_{-\infty}^{\infty} \sigma^2 \frac{\partial}{\partial x}\left(e^{-\frac{x^2}{2\sigma^2}}\right) \sin(\omega x) \,\mathrm{d}x, \\
&= \sigma^2 e^{-\frac{x^2}{2\sigma^2}} \sin(\omega x)\Big|_{-\infty}^{\infty} - \omega \int_{-\infty}^{\infty} \sigma^2 e^{-\frac{x^2}{2\sigma^2}} \cos(\omega x) \,\mathrm{d}x, \\
&= -\omega\sigma^2 I(\omega) \Leftrightarrow \\
I(\omega) &= C e^{-\sigma^2\omega^2/2}, \\
I(0) = C &= \sqrt{2\pi}\sigma, \\
I(\omega) &= \sqrt{2\pi}\sigma e^{-\sigma^2\omega^2/2}.
\end{aligned}$$

Plugging this into (1.16) gives the result

$$\mathcal{F}(\mathcal{N}) = \frac{1}{\sqrt{2\pi}} e^{-i\omega\mu} e^{-\sigma^2\omega^2/2}. \qquad (1.17)$$

This can be used in conjunction with (1.13) to obtain

$$\begin{aligned}
\mathcal{F}(\mathcal{N}_a * \mathcal{N}_b) &= \sqrt{2\pi} \frac{1}{\sqrt{2\pi}} e^{-i\omega\mu_a} e^{-\sigma_a^2\omega^2/2} \cdot \frac{1}{\sqrt{2\pi}} e^{-i\omega\mu_b} e^{-\sigma_b^2\omega^2/2}, \qquad (1.18) \\
&= \frac{1}{\sqrt{2\pi}} e^{-i\omega(\mu_a-\mu_b)} e^{-(\sigma_a^2+\sigma_b^2)\omega^2/2}, \qquad (1.19)
\end{aligned}$$

which is recognized as the transform of another Gaussian describing the separation with $\mu_s = \mu_a - \mu_b$ and $\sigma_s^2 = \sigma_a^2 + \sigma_b^2$, so taking the inverse Fourier transformation gives the convolution

$$\mathcal{N}_a * \mathcal{N}_b = \frac{1}{\sqrt{2\pi}\sigma_s} e^{-\frac{(x-\mu_S)^2}{2\sigma_S^2}}. \tag{1.20}$$

Hence, a reasonable measure of the separation of two projected distributions is

$$d = \frac{(\mu_a - \mu_b)^2}{\sigma_a^2 + \sigma_b^2}. \tag{1.21}$$

1.4.2 Optimizing separation

To maximize the separation, the numerator and denominator, respectively, of (1.21) can be rewritten in terms of w in the following way (using $\widetilde{\mu}_i$ to denote projected means) and simplified by introducing scattering matrices:

$$\begin{aligned}
(\widetilde{\mu}_a - \widetilde{\mu}_b)^2 &= \left(w^T (\mu_a - \mu_b)\right)^2, && (1.22)\\
&= w^T (\mu_a - \mu_b)(\mu_a - \mu_b)^T w, && (1.23)\\
&= w^T S_B w, && (1.24)
\end{aligned}$$

and

$$\begin{aligned}
\widetilde{\sigma}_i^2 &= \sum_{y \in i} \frac{1}{N} (y - \widetilde{\mu}_i)^2, && (1.25)\\
&= w^T \sum_{y \in i} (x - \mu_i)(x - \mu_i)^T w, && (1.26)\\
&= w^T S_i w, && (1.27)\\
\widetilde{\sigma}_a^2 + \widetilde{\sigma}_b^2 &= w^T S_W w, && (1.28)
\end{aligned}$$

having introduced the between-class and within-class scatter matrices S_B and S_W by

$$\begin{aligned}
S_B &= (\mu_a - \mu_b)(\mu_a - \mu_b)^T, && (1.29)\\
S_i &= \sum_{y \in i} (x - \mu_i)(x - \mu_i)^T, && (1.30)\\
S_W &= S_a + S_b. && (1.31)
\end{aligned}$$

Hence, the objective is to solve

$$\frac{\mathrm{d}}{\mathrm{d}w}J(w) = \frac{\mathrm{d}}{\mathrm{d}w}\left(\frac{w^T S_B w}{w^T S_W w}\right) = 0, \tag{1.32}$$

$$\frac{\frac{\mathrm{d}[w^T S_B w]}{\mathrm{d}w} w^T S_W w - w^T S_B w \frac{\mathrm{d}[w^T S_W w]}{\mathrm{d}w}}{(w^T S_W w)^2} = 0, \tag{1.33}$$

$$2S_B w \cdot w^T S_W w - w^T S_B w \cdot 2S_W w = 0, \tag{1.34}$$

$$S_B w - \frac{w^T S_B w \cdot S_W w}{w^T S_W w} = 0, \tag{1.35}$$

$$S_B w - S_W w J(w) = 0, \tag{1.36}$$

$$S_B w = S_W w J(w), \tag{1.37}$$

$$S_W^{-1} S_B w = J(w) w. \tag{1.38}$$

The optimal projection vector w^* which satisfies this is

Vær lige sikker på at du forstår det her.

$$w^* = S_W^{-1}(\mu_a - \mu_b). \tag{1.39}$$

Figure 1.13 shows a visualization of this that I generated by drawing (x, y) points from two random distributions to simulate two distinct classes of points. If the distributions are independent and Gaussian, the projections will also form Gaussian distributions, and the probability of a new point belonging to e.g. class a given its coordinates d can be estimated using Bayesian probability

$$P(a|d) = \frac{P(d|a)P(a)}{P(d|a)p(a) + P(d|b)P(b)}, \tag{1.40}$$

where $P(a)$ and $P(b)$ are simply the prior probabilities for encountering the respective classes, and the conditional probabilities, e.g. $P(d|a)$ are simply given by the value of the projected Gaussian $\mathcal{N}(x'; \widetilde{\mu}_a, \widetilde{\sigma}_a)$ at the projected coordinate x'. In practise, even when the points are not independent or Gaussian, so that (1.40) is not a precise estimate of the probability of the point representing a given class, the class with the highest posteriori according to (1.40) still often turns out to be a good guess.

This method accurately predicted the gender of 79.8% of the participants, which is not particularly impressive as 77.3% of participants were male, so a classifier that assumes that every participant is male would have a comparable success rate. An immediate source of concern is the assumption of linearity: It is possible that the data is ordered in such a way that it is possible to separate data points fairly well based on

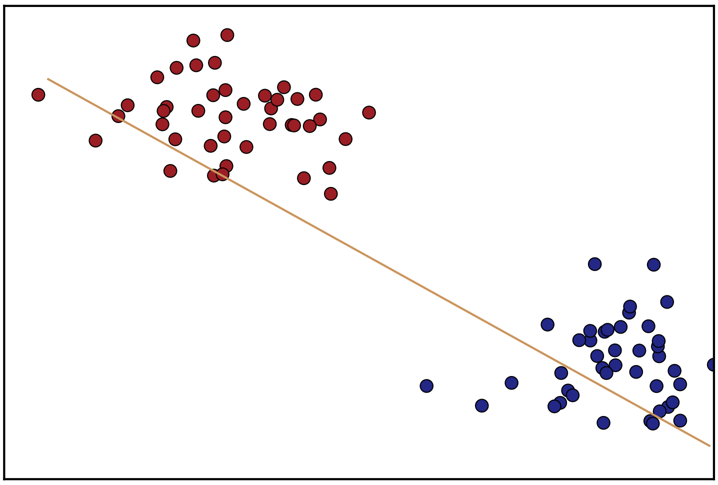


Figure 1.13: Two collections of points drawn from independent Gaussian distributions, representing class a and class b. If the points are projected onto the straight line, which is given by (1.39), the separation between the peaks representing the two classes is maximized.

gender or some psychological trait, just not using a linear classifier. As an extreme example of this, figure 1.14 shows a situation where the points representing one class are grouped together in an 'island' in the middle, isolating them from points representing the remaining class. While it is clear that there's a pattern here, a linear classifier fails to predict classes more precisely than their ratio. Support Vector Machines, or SVMs are another linear classification technique which can be generalized to detect patterns like that in figure 1.14. This is described in section 2.1

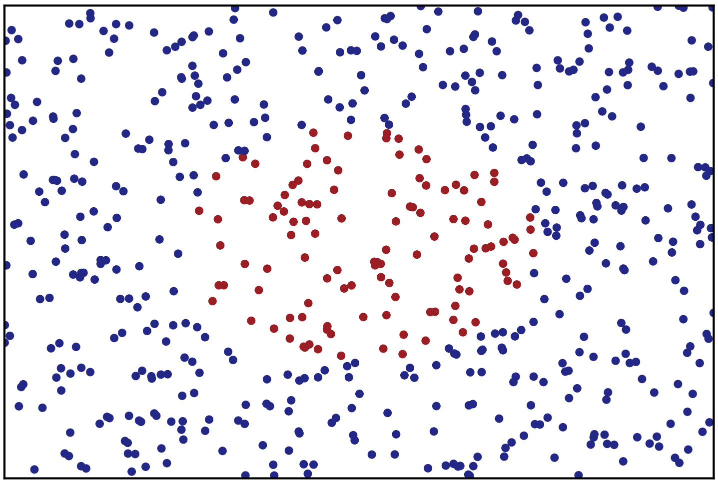


Figure 1.14: An example of data points representing class a are clearly discernible from those of class b, yet a linear Fisher classifier fails to predict the classes more precisely than the ratio of b to a.

CHAPTER 2

Psychological Profiling & Machine Learning

Machine learning is currently a strong candidate for prediction of psychological profiles from phone data. This chapter describes the application of the quantitative data described in setion 1.2 and various machine learning schemes, starting with support vector machines (SVMs).

1000 kilder!!!

Uddyb når der er flere modeller.

2.1 Support Vector Machines

The purpose of this section is to introduce SVMs and attempt to apply them to the data obtained in 1.2. The introduction is mainly based on introductory texts by Marti Hearst [11] and Christopher Burges [5]. SVMs in their simplest form (*simplest* meaning using a linear kernel, which I'll explain shortly) can be thought of as a slight variation on the linear classifier described in section 1.4. However, where LDA finds a line such that the distribution of the points representing various classes projected onto the line is maximized, the aim of SVMs is to establish the hyperplane that represents the best possible slicing of the feature space into regions containing only points corresponding to the different classes. A simple example of this is shown in figure 2.1. Using labels ±1 to denote classes, the problem may be stated as trying to guess the mapping from an N-dimensional data space to classes $f : \mathbb{R}^N \to \{\pm 1\}$ based on a set of training data in $\mathbb{R}^N \otimes \{\pm 1\}$. I'll describe separately the properties of

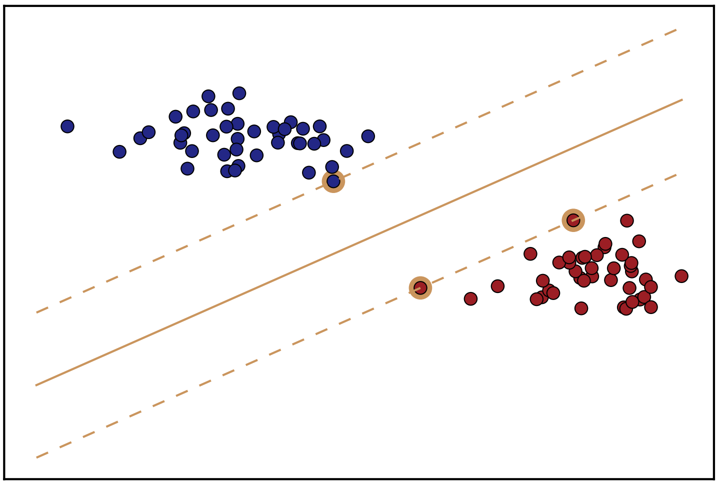


Figure 2.1: The same points as those shown in figure 1.13, except points in class a and class b are now pictured along with their maximally separating hyperplane.

this maximally separating hyperplane, how it is obtained, and how the method is generalized to non-linear classification problems as the 'island' illustrated in figure 1.14.

The well-known equation for a plane is obtained by requiring that its normal vector $\mathbf{w}$ be orthogonal to the vector from some point in the plane $\mathbf{p}$ to any point $\mathbf{x}$ contained in it:

$$\mathbf{w} \cdot (\mathbf{x} - \mathbf{p}) = 0. \tag{2.1}$$

The left hand side of (2.1) gives zero for points in the plane and positive or negative values when the point is displaced in the same or opposite direction as the normal vector, respectively. Hence, $\operatorname{sign}(\mathbf{w} \cdot (\mathbf{x} - \mathbf{p}))$ may be taken as the decision function. It is clear from (2.1) that the normal vector may be scaled without changing the actual plane (of course the decision function is inverted if a negative value is chosen), so $\mathbf{w}$ is usually rescaled such that

$$\mathbf{w} \cdot (\mathbf{x} - \mathbf{p}) = \mathbf{w} \cdot \mathbf{x} + b = \pm 1, \tag{2.2}$$

for the points that are closest to the separating plane. Those points located on the margin are encircled in figure 2.1. In general then, the meaning of the sign and magnitude of

$$\mathbf{w} \cdot \mathbf{x} + b \tag{2.3}$$

will be the predicted class and a measure of prediction confidence, respectively, for new data points. Finally, note that $\mathbf{w}$ can be expanded in terms of the data points that are on the margin in figure 2.1 as

$$\mathbf{w} = \sum_i v_i \mathbf{x}_i, \tag{2.4}$$

these $\mathbf{x}_i$, the position vectors of the margin points in data space, are the 'support vectors' that lend their name to the method.

2.1.1 Obtaining the Maximally Separating Hyperplane

Assuming first that it is possible to slice the data space into two regions that contain only points corresponding to one class each, and that the plane's normal vector has already been rescaled according to (2.2), the following inequalities hold:

$$\begin{aligned} \mathbf{x}_i \cdot \mathbf{w} + b \geq 1, y_i = +1, \\ \mathbf{x}_i \cdot \mathbf{w} + b \leq -1, y_i = -1. \end{aligned} \tag{2.5}$$

Multiplying by y_i, both simply become

$$y_i \left(\mathbf{x}_i \cdot \mathbf{w} + b \right) - 1 \geq 0. \tag{2.6}$$

The distance between the separating plane and each of the margins in figure 2.1 is $1/|\mathbf{w}|$, so in order to maximize the separation, $|\mathbf{w}|$ must be minimized. For mathematical convenience, $\frac{1}{2}|\mathbf{w}|^2$, rather than $|\mathbf{w}|$ is included in the Lagrangian, which then becomes

$$L = \frac{1}{2}|\mathbf{w}|^2 - \sum_i \alpha_i y_i \left(\mathbf{x}_i \cdot \mathbf{w} + b - 1 \right), \tag{2.7}$$

must be minimized with the constraints

$$\alpha_i \geq 0, \tag{2.8}$$

$$\frac{\partial L}{\partial \alpha_i} = 0. \tag{2.9}$$

A result from convex optimization theory known as Wolfe Duality[20] states that one may instead maximize the above Lagrangian subject to

$$\nabla_w L = \frac{\partial L}{\partial b} = 0, \tag{2.10}$$

which gives conditions

$$\mathbf{w} = \sum_j \alpha_j y_j \mathbf{x}_j, \tag{2.11}$$

$$\sum_j \alpha_j y_j = 0. \tag{2.12}$$

These can be plugged back into (2.7) to obtain

$$L_D = \frac{1}{2}\sum_i \sum_j \alpha_i y_i \alpha_j y_j \mathbf{x}_i \cdot \mathbf{x}_j - \sum_i \alpha_i y_i \Big(\mathbf{x}_i \cdot \sum_j \alpha_j y_j \mathbf{x}_j + b\Big) + \sum_i \alpha_i, \tag{2.13}$$

$$L_D = -\frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \mathbf{x}_j + \sum_i \alpha_i. \tag{2.14}$$

A problem with this is that eqs 2.5 can only be satisfied in the completely separable case, although it is easy to imagine an example in which a classifier performs well but not flawlessly on the training set. For instance, if two points, one from each class, in figure 2.1 were permuted, the classifier shown in the plot would still do a very good job, but eqs. 2.5 would not be satisfiable, causing the method to fail. This is remedied by introducing slack variables[3]

$$\begin{aligned} \mathbf{x}_i \cdot \mathbf{w} + b &\geq 1 - \xi_i, \quad y_i = +1, \\ \mathbf{x}_i \cdot \mathbf{w} + b &\leq -(1 - \xi_i), \quad y_i = -1, \\ \xi_i &\geq 0, \end{aligned} \tag{2.15}$$

which allows the algorithm to misclassify. This should come without a cost to the overall Lagrangian, or one would just end up classifying randomly, so a 'cost term', $C \cdot \sum_i \xi_i$ is added as well. The value of C (as well as a few similar parameters which have yet to be introduced) is usually determined experimentally by simply varying it across some space of possible values and choosing the value resulting in the best performance - see for instance figure 2.3. In the case of the misclassification cost parameter C, low values will result in low performance, whereas too large values will result in overfitting. The above can be rewritten exactly as previously, except another set of non-negative Lagrange multipliers μ_i are added to (2.7) to ensure positivity of the ξ_i, resulting in

$$L = \frac{1}{2}|\mathbf{w}|^2 + C \cdot \sum_i \xi_i - \sum_i \alpha_i y_i \left(\mathbf{x}_i \cdot \mathbf{w} + b - 1 + \xi_i\right) - \sum_i \mu_i \xi_i. \tag{2.16}$$

This results in the same dual Lagrangian L_D as before, but with an upper bound on the α_i:

$$0 \leq \alpha_i \leq C. \tag{2.17}$$

The methods outlined above can also be used to solve regression, rather than classification, problems[19]. The training data will then be in $\mathbb{R}^{N+1}$ rather than in $\mathbb{R}^N \otimes \{\pm 1\}$, and the value of the decision function in (2.3) is predicted instead of using only its sign. The criterion of correct classification from (2.6) is replaced by the demand that predictions be within some tolerated margin ϵ of the true value of the training point y_i, so (2.5) becomes

$$-\epsilon \leq \mathbf{x}_i \cdot \mathbf{w} + b - y_i \leq \epsilon \tag{2.18}$$

so when slack variables ξ_i and ξ_i^* (for the lower and upper bound, respectively) like in (2.15) are introduced, the Lagrangian from (2.16) becomes

$$L = \frac{1}{2}|\mathbf{w}|^2 + C \sum_i \left(\xi_i + \xi_i^* \right), \tag{2.19}$$

with constraints

$$\mathbf{x}_i \cdot \mathbf{w} + b - y_i \geq -\left(\epsilon + \xi_i\right), \tag{2.20}$$

$$\mathbf{x}_i \cdot \mathbf{w} + b - y_i \leq \epsilon + \xi_i^*, \tag{2.21}$$

$$\xi_i, \xi^* \geq 0. \tag{2.22}$$

The main point to be emphasized here is that the training data $\mathbf{x}_i$ only enter into the dual Lagrangian of (2.14) as inner products. This is essential when extending the SVM model to nonlinear cases, which is the subject of the following section.

2.1.2 Generalizing to the non-linear case

The fact that the data $\mathbf{x}_i$ only occur as inner products in (2.14) makes one way of generalizing to non-linearly separable datasets straightforward: Referring back to figure 1.14, one might imagine bending the plane containing the data points by curling the edges outwards in a third dimension after which a two-dimensional plane could separate the points very well. In general, this means applying some mapping

$$\Phi : \mathbb{R}^l \to \mathbb{R}^h, \quad h > l, \tag{2.23}$$

to the $\mathbf{x}_i$ (l and h are for low and high, respectively). For example, one could look for a mapping such that the new inner product becomes

$$\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j) = \left(\mathbf{x}_i \cdot \mathbf{x}_j\right)^2. \tag{2.24}$$

I'll describe the components of each vector separately, so I'm going to change notation to let the subscripts denote coordinates and using $\mathbf{x}$ and $\mathbf{y}$ as two arbitrary feature vectors, where the latter shouldn't be confused with the class labels used earlier. As an example, in two dimensions the above becomes

$$(\mathbf{x}\cdot\mathbf{y})^2 = \left(\sum_{i=1}^{2} x_i y_i\right)^2 = x_1^2 y_1^2 + 2x_1 y_1 x_2 y_2 + x_2^2 y_2^2, \tag{2.25}$$

meaning that one possibility for Φ is

$$\Phi : \mathbf{x} \mapsto \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1 x_2 \\ x_2^2 \end{pmatrix} \tag{2.26}$$

This can be generalized to d-dimensional feature vectors and to taking the n'th power rather than the square using the multinomial theorem:

$$\left(\sum_{i=1}^{d} x_i\right)^n = \sum_{\sum_{i=1}^{d} k_i = n} \frac{n!}{\prod_{l=1}^{d} k_l!} \prod_{j=1}^{d} x_j^{k_j}, \tag{2.27}$$

where the subscript $\sum_{i=1}^{d} k_i = n$ simply means that the sum goes over any combination of d non-negative integers k_i that sum to n. I wish to rewrite this slightly for two reasons: to simplify the notation in order to make a later proof more manageable, and to help quantify how quickly the number of dimensions in the output space grows to motivate a trick to avoid these explicit mappings.

As stated, the sum on the RHS of (2.27) runs over all combinations of d integers which sum to n. This can be simplified by introducing a function K, which simply maps

$$K : n, d \mapsto \left\{ \{k\} \in \mathbb{N}^d \,\middle|\, \sum_{i=1}^{d} k_i = n \right\}, \tag{2.28}$$

and denoting each of those collections $\{k\}_i$ so each of the coefficients in (2.27) can be written

$$\frac{n!}{\prod_{i=1}^{d} k_i!} = C_{\{k\}}. \tag{2.29}$$

Then, (2.27) becomes

$$\left(\sum_{i=1}^{d} x_i\right)^n = \sum_{K(n,d)} C_{\{k\}} \prod_{j=1}^{d} x_j^{k_j} \tag{2.30}$$

To show how quickly the dimensions of the required embedding space grows, note that the dimension is equal to the number of terms in the sum above, i.e.

$$\dim(\mathbb{R}^h) = |K(n,d)| = \left| \left\{ \{k\} \in \mathbb{N}^d \;\middle|\; \sum_{i=1}^{d} k_i = n \right\} \right|, \tag{2.31}$$

which can be computed using a nice trick known from enumerative combinatorics.

Consider the case where $n = 5$ and $d = 3$. $K(5,3)$ then contains all sets of 3 integers summing to 5, such as $1,3,1$ or $0,1,4$. Each of these can be uniquely visualized as 5 unit values distributed into 3 partitions in the following fashion:

$$\circ \mid \circ \; \circ \; \circ \mid \circ,$$
$$\mid \circ \mid \circ \; \circ \; \circ \; \circ,$$

and so on. It should be clear that you need n $\circ$-symbols and $d-1$ $|$ separators. The number of possible such combinations, and hence the dimensionality of the embedding space, is then

$$\binom{n+d-1}{n} = \frac{(n+d-1)!}{n!(d-1)!}. \tag{2.32}$$

This number quickly grows to be computationally infeasible, which motivates one to look for a way to compute the inner product in the embedded space without performing the explicit mapping itself. This is the point of the so-called 'kernel trick', which I'll introduce in the following.

The idea of the kernel trick is that since only the inner products between feature vectors in the embedded space are required, one might as well look for some function K of the original feature vectors which gives the same scalar as the inner product in the embedded space, i.e.

$$K(\mathbf{x}, \mathbf{y}) = \Phi(\mathbf{x}) \cdot \Phi(\mathbf{y}). \tag{2.33}$$

In the polynomial case treated above, the correspondence between the kernel function $K(\mathbf{x}, \mathbf{y})$ and the explicit mapping Φ is straightforward:

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^n, \tag{2.34}$$

$$\Phi(\mathbf{x}) = \sum_{K(n,d)} \sqrt{C_{\{k\}}} \prod_{j=1}^{d} x_j^{k_j}, \tag{2.35}$$

so that (2.33) is true by the multinomial theorem and the above considerations. However, situations arise in which the explicit mapping Φ isn't directly obtainable, and the correspondence of the kernel function to inner products in higher dimensional spaces is harder to demonstrate. This is the subject of the following section.

Radial Basis Functions

One commonly used kernel function is the RBF, or radial basis function, kernel:

$$K(\mathbf{x}, \mathbf{y}) = \mathrm{e}^{|\mathbf{x}-\mathbf{y}|^2/2\sigma}. \tag{2.36}$$

Burges [5] shows that the polynomial kernel is valid, so I'll show how the argument extends to the RBF kernel in the following.

Mercer's condition[18] states that for a kernel function $K(\mathbf{x}, \mathbf{y})$, there exists a corresponding Hilbert space $\mathcal{H}$ and a mapping Φ as specified earlier, iff any L^2-normalizable function $g(\mathbf{x})$ satisfies

$$\int K(\mathbf{x}, \mathbf{y}) g(\mathbf{x}) g(\mathbf{y})\, \mathrm{d}\mathbf{x}\, \mathrm{d}\mathbf{y} \geq 0. \tag{2.37}$$

This can be shown be rewriting (2.36) as

$$K(\mathbf{x}, \mathbf{y}) = \mathrm{e}^{(\mathbf{x}-\mathbf{y})\cdot(\mathbf{x}-\mathbf{y})/2\sigma} = \mathrm{e}^{|\mathbf{x}|^2/2\sigma} \mathrm{e}^{|\mathbf{y}|^2/2\sigma} \mathrm{e}^{-\mathbf{x}\cdot\mathbf{y}/\sigma}, \tag{2.38}$$

and expanding the last term in $(\mathbf{x} \cdot \mathbf{y})$ as

$$\mathrm{e}^{-\mathbf{x}\cdot\mathbf{y}/\sigma} = \sum_{i=0}^{\infty} \frac{(-1)^i}{i!\sigma^i} (\mathbf{x} \cdot \mathbf{y})^i, \tag{2.39}$$

but using (2.30) on the dot product gives

$$(\mathbf{x} \cdot \mathbf{y})^i = \left(\sum_{j=1}^{d} x_j y_j \right)^i = \sum_{K(i,d)} C_{\{k\}} \prod_{j=1}^{d} x_j^{k_j} y_j^{k_j} \tag{2.40}$$

so the Taylor expansion becomes

$$\mathrm{e}^{-\mathbf{x}\cdot\mathbf{y}/\sigma} = \sum_{i=0}^{\infty} \sum_{K(i,d)} \frac{(-1)^i}{i!\sigma^i} C_{\{k\}} \prod_{j=1}^{d} x_j^{k_j} y_j^{k_j}, \tag{2.41}$$

which can be plugged back into (2.38) to yield

$$K(\mathbf{x}, \mathbf{y}) = \sum_{i=0}^{\infty} \sum_{K(i,d)} \frac{(-1)^i}{i!\sigma^i} C_{\{k\}} \mathrm{e}^{|\mathbf{x}|^2/2\sigma} \mathrm{e}^{|\mathbf{y}|^2/2\sigma} \prod_{j=1}^{d} x_j^{k_j} y_j^{k_j}. \tag{2.42}$$

The underlying reason for these algebraic shenanigans is that (2.42) is clearly separable so that the integral in (2.37) from Mercer's condition becomes

$$\int K(\mathbf{x},\mathbf{y})g(\mathbf{x})g(\mathbf{y})\,\mathrm{d}\mathbf{x}\,\mathrm{d}\mathbf{y} \tag{2.43}$$

$$=\sum_{i=0}^{\infty}\sum_{K(i,d)}\frac{(-1)^i}{i!\sigma^i}C_{\{k\}}\int_{\mathbb{R}^{2d}}\mathrm{e}^{|\mathbf{x}|^2/2\sigma}\mathrm{e}^{|\mathbf{y}|^2/2\sigma}\prod_{j=1}^{d}x_j^{k_j}y_j^{k_j}g(\mathbf{x})g(\mathbf{y})\,\mathrm{d}\mathbf{x}\,\mathrm{d}\mathbf{y} \tag{2.44}$$

$$=\sum_{i=0}^{\infty}\sum_{K(i,d)}\frac{(-1)^i}{i!\sigma^i}C_{\{k\}}\left(\int_{\mathbb{R}^d}\mathrm{e}^{|\mathbf{x}|^2/2\sigma}\prod_{j=1}^{d}x_j^{k_j}g(\mathbf{x})\,\mathrm{d}\mathbf{x}\right)\cdot\left(\int_{\mathbb{R}^d}\mathrm{e}^{|\mathbf{y}|^2/2\sigma}\prod_{j=1}^{d}y_j^{k_j}g(\mathbf{y})\,\mathrm{d}\mathbf{y}\right) \tag{2.45}$$

$$=\sum_{i=0}^{\infty}\sum_{K(i,d)}\frac{(-1)^i}{i!\sigma^i}C_{\{k\}}\left(\int_{\mathbb{R}^d}\mathrm{e}^{|\mathbf{x}|^2/2\sigma}\prod_{j=1}^{d}x_j^{k_j}g(\mathbf{x})\,\mathrm{d}\mathbf{x}\right)^2 \tag{2.46}$$

$$\geq 0. \tag{2.47}$$

Hence, radial basis functions satisfy Mercer's condition and the kernel described above can be plugged into the dual Lagrangian from (2.14) to obtain

$$L_D=-\frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \mathrm{e}^{|\mathbf{x}_i-\mathbf{x}_j|^2/2\sigma}+\sum_i\alpha_i, \tag{2.48}$$

which must be maximized subject to the same constraints as earlier. The concrete optimization procedure is complicated and already implemented in most machine learning libraries, so I choose not to go into details with that, but instead to demonstrate the effectiveness of the RBF kernel approach on the non-linear-separable points that were generated earlier. figure 2.2 shows the points again, along with the *decision frontier* i.e. the curve which separates regions in which points are classified into separate classes. The danger of overfitting should be clear from figure 2.2. If the cost of misclassification C and the sharpness of the RBFs, usually denoted by $\gamma = 2/\sigma$ are set sufficiently high, the algorithm will simply end up with a tiny decision boundary around every training point of class a, resulting in flawless classification on the training set, but utter failure on new data. The typical way of evaluating this is to perform k-fold validation, meaning that the available data is k equal parts and the SVM is consecutively trained on $k-1$ parts and tested on the remaining. A variant of this, which my code uses, is stratified k-fold validation, which only differs in that the data is partitioned so as to keep the ratio between the different classes in each parts as close to equal as possible.

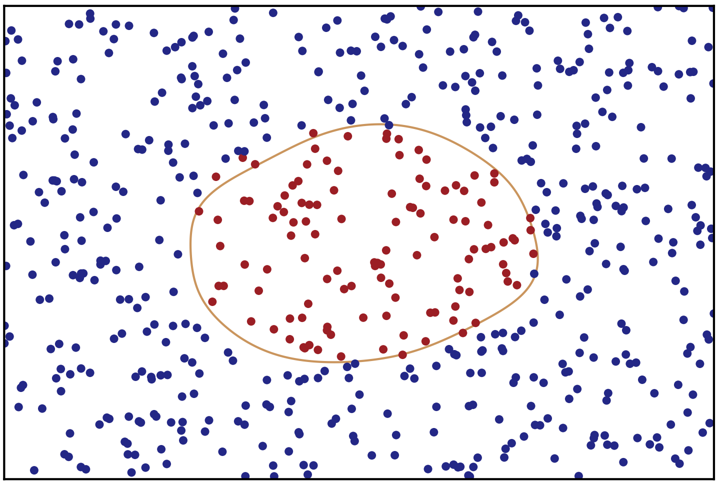


Figure 2.2: The 'island' scenario of figure 1.14 revisited. The points representing class a and class b have been mapped to a higher-dimensional space in which it is possible to construct a separating hyperplane whose decision frontier is also shown.

The γ parameter is often fixed by performing a grid search similar to that discussed earlier. Figure 2.3 shows the resulting heat map from a grid search.

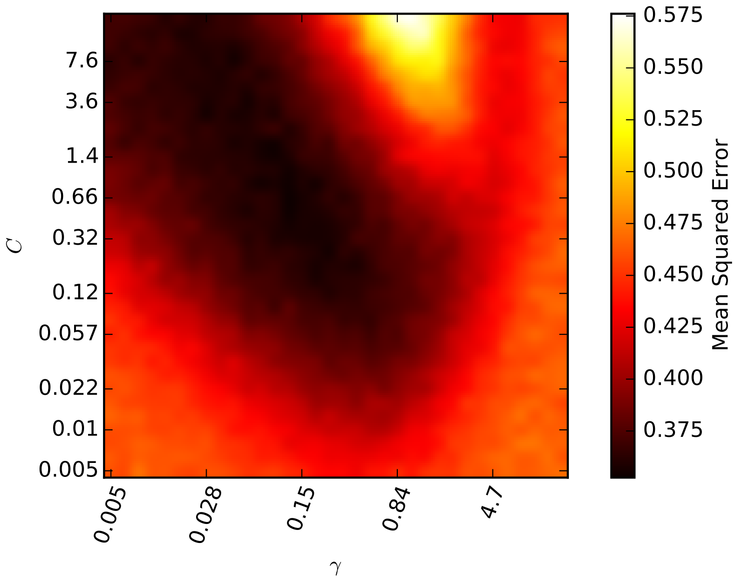


Figure 2.3: Result of a grid search for the optimal combination of values for the cost parameter C and the sharpness γ of the Gaussian kernel function giving optimal values of $C = 0.52$ and $\gamma = 0.12$.

2.1.3 Implementing a custom weighted radial basis function kernel

In general, the features extracted as described in section 1.2 are too numerous for efficient implementations of most machine learning schemes. [6] work around this by determining the linear correlation between each feature and the target values, and then discarding features whose correlation is below some threshold and letting the rest contribute equally in the SVM. They never explicate this threshold but state that they include features 'significantly related' to the target values, which convention would suggest means a threshold of 0.05. [15] have demonstrated significant improvements in SVM performance by assigning variable importances to each feature, so I implemented a modified kernel function where I don't just discard features with linear correlations below some threshold but also assign to each remaining feature a normalized weight given by its correlation with the output variable.

I do this by defining a diagonal matrix $\hat{M}$ where the i,ith element is the linear correlation coefficient between feature i and the target variable, and using the matrix as a metric in the exponent of the usual radial basis functions so the kernel $K(\mathbf{x},\mathbf{y})$ becomes

$$e^{\gamma|\mathbf{x}-\mathbf{y}|^2} \rightarrow e^{\gamma(\mathbf{x}-\mathbf{y})^T\hat{M}(\mathbf{x}-\mathbf{y})} \tag{2.49}$$

Note that this does not change the validity of the proof I gave in section 2.1.2, so this weighted RBF kernel also satisfies Mercer's condition and may hence be used as a kernel function. It turned out to be nontrivial to write an implementation of this which had a syntax consistent with that of the default methods available in the library I used, and which provided a similarly simple way to grid search over its parameters (C and γ as in the usual RBF kernel SVM along with a threshold parameter). I ended up solving this by writing a metamethod to provide a kernel matrix based on an input list of variable importances as well as a default sharpness parameter denoted γ.

```
def make_kernel(importances, gamma = 1.0):
    '''Returns a weighted radial basis function (WRBF) kernel which can be
     passed to an SVM or SVR from the sklearn module.

     Parameters:
     -----------------------
     importances : list
       The importance of each input feature. The value of element i can mean
       e.g. the linear correlation between feature i and target variable y.
       None means feature will be weighted equally.

     gamma : float
       The usual gamma parameter denoting inverse width of the gaussian used.
     '''
    def kernel(x,y, *args, **kwargs):
        d = len(importances) #number of features
        impsum = sum([imp**2 for imp in importances])
        if not impsum == 0:
            normfactor = 1.0/np.sqrt(impsum)
        else:
            normfactor = 0.0
        #Metric to compute distance between points
        metric = dok_matrix((d,d), dtype = np.float64)
        for i in xrange(d):
            metric[i,i] = importances[i]*normfactor
        #
        result = np.zeros(shape = (len(x), len(y)))
        for i in xrange(len(x)):
            for j in xrange(len(y)):
                diff = x[i] - y[j]
                dist = diff.T.dot(metric*diff)
                result[i,j] = np.exp(-gamma*dist)
        return result
    return kernel
```

With that in place, it was a simple matter to implement classifiers and regressors by inheriting from the default classes and overriding the constructor methods like this:

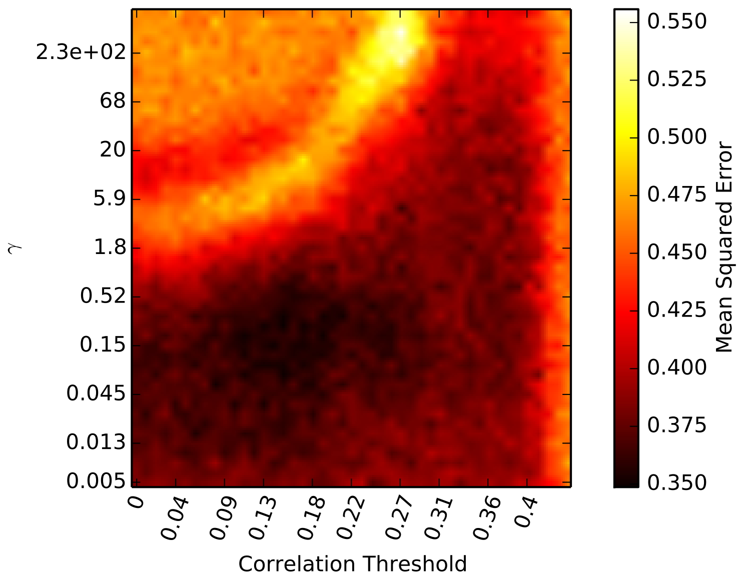


Figure 2.4: Heat map resulting from a grid search over the parameter space of the linear correlation threshold and the default sharpness γ of the radial basis functions showing optimal values of 0.25 and 0.15 for γ and the threshold, respectively.

```python
class WRBFR(svm.SVR):
    '''Weighted radial basis function support vector regressor.'''
    def __init__(self, importances, C = 1.0, epsilon = 0.1,
                 gamma = 0.0):
    kernel = make_kernel(importances = importances, gamma = gamma)
    super(WRBFR, self).__init__(C = C, epsilon = epsilon, kernel = kernel)
```

Figures 2.4 and 2.5 show heat maps resulting from grid searches over the threshold parameter versus γ and C, respectively. Interestingly, the ideal correlation threshold seems to be well above the value used in [6].

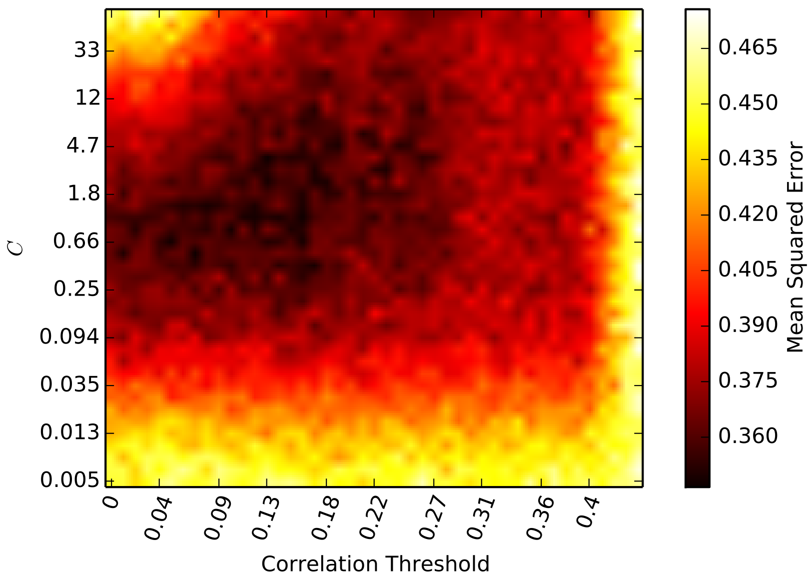


Figure 2.5: Heat map resulting from a grid search over the parameter space of the linear correlation threshold and the cost parameter C of the radial basis functions showing optimal values of 3.6 and 0.13 for γ and the threshold, respectively.

2.1.4 Statistical subtleties

An important note should be made here about some often neglected subtleties relating to uncertainties. Physicists often deal with measurements that can assumed to be independently drawn from a normal distribution $\mathcal{N}(x_i; \mu, \sigma^2)$ due to the central limit theorem. With a large number of measurements n, the standard deviation of a sample

$$\sigma^2 = \frac{1}{N}\sum_i^N (x_i - \mu)^2 , \tag{2.50}$$

converges as $N \to \infty$ to the maximum likelihood, minimum variance unbiased estimator for the true variance of the underlying distribution with unknown mean

$$\hat{\sigma}^2 = \frac{1}{N-1}\sum_i^N (x_i - \mu)^2 . \tag{2.51}$$

The standard deviation σ and the width of the underlying gaussian $\hat{\sigma}^2$ can then often be used interchangeably. This tempts some people into the questionable habit of always assuming that the sample standard deviance can be used as the 68% confidence interval of their results.

When using a K-fold validation scheme, the performance scores for the various folds cannot be assumed to be independently drawn from an underlying distribution, as the test set of one fold is used in the training sets of the remaining folds. In fact, it has been shown [1] that there is no unbiased estimator for the variance of the performance estimated using K-fold validation. However, as K-fold validation is more effective than keeping the test, and training data separate, which can be shown using Jensen's inequality along with some basic properties of expectation values [2], I'll mostly use K-fold regardless. As the standard deviation still provides a qualitative measure of the consistency of the model's performance, I'll still use the sample STD in a usual fashion, such as error bars, unless otherwise is specified, but the reader should keep in mind that these do not indicate precise uncertainties whenever K-fold validation has been involved.

2.2 Decision Trees & Random Forests

Another popular machine learning scheme is that of random forests, which consist of an ensemble of decision trees. A decision tree is a very intuitive method for classification problems which can be visualized as a kind of flow chart in the following fashion. As usual, the problem consists of a set of feature vectors $\mathbf{x}_i$ and a set of corresponding class labels y_i. A decision tree then resembles a flowchart starting at the root of the tree, at each node splitting into branches and finally branching into leaves at which all class labels should be identical. At each node, part of the feature vector is used to split the dataset into parts. This resembles the 'twenty questions' game, in which one participant thinks of a famous person and another attempts to guess who it is by asking a series of yes/no-questions, each one splitting the set of candidates in two parts. In this riddle game and in decision tree learning, there are good and bad questions (asking whether the person was born on March 14th, 1879 is a very bad first question, for instance). There are several ways of quantifying how 'good' a yes/no-question, corresponding to a partitioning of the dataset, is.

On metric for this is the Gini Impurity Index I_G, which is computed by summing over each class label:

$$I_G = \sum_i f_i (1 - f_i) = 1 - \sum_i f_i^2, \tag{2.52}$$

where f_i denotes the fraction of the set the consists of class y_i. Using this as a metric, the best partitioning is the one which results in the largest drop in total Gini impurity following a branching. Another metric is the information gain measured by comparing the entropy before a split with a weighted average of the entropy in the groups resulting from the split. Denoting the fractions of the various classes in the parent group, i.e. before splitting, by f_i and the two child groups by a_i and b_i, the information gain is

$$I_E = -\sum_i f_i \log_2 f_i + \frac{n_a}{N}\sum_i a_i \log_2 a_i + \frac{n_b}{N}\sum_i b_i \log_2 b_i. \qquad (2.53)$$

However, if too many such nodes are added to a decision tree, overfitting, i.e. extreme accuracies on training data but poor performance on new data, becomes a problem. This can be remedied by instead predicting with a majority vote, or averaging in the case of regression problems, from an ensemble of randomized decision trees called a random forest. The main merits of random forests are their accuracy and ease of use, and their applications as auxiliary methods in other machine learning schemes, which I'll elaborate on shortly.

The individual trees in a random forest are grown using a randomly selected subset of the training data for each tree. The data used to construct a given tree is referred to as 'in bag', whereas the remaining training data is referred to as 'out of bag' (OOB) for the given tree. At each node, a set number of features is randomly selected and the best possible branching, cf. the above considerations, is determined. The only parameters that must be tweaked manually are the number of trees in the forest, number of features to include in each branching, and the maximum tree depth. While other variables such as the metric for determining branching quality as described above, may be customized, those aren't essential to achieve a decent predictor, which is robust in regard to both overfitting and irrelevant parameters.[4]

There doesn't seem to be a single universally accepted way of adjusting these parameters, so I chose a somewhat pragmatic approach of simply checking how well various choices for each parameter performed on a randomly selected trait. For instance, figure 2.6 shows how well a random forest predicted the tertiles of participants' extroversion as a function of the fraction of available features each tree was allowed to include in each branching. This was done using a large number of trees ($n = 1000$) and using each of the two metrics described earlier. The number of features used pr split doesn't seem to have any significant effect on performance, and as the entropy metric seems to perform as well or slightly better than Gini impurity, I decided to stick to that. A similar plot of the performance

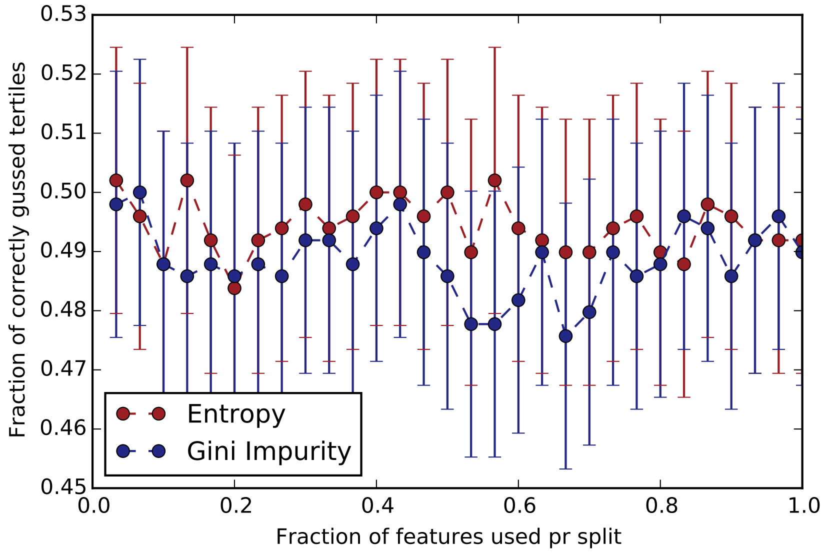


Figure 2.6: Performance of a random forest with 1000 decision trees using various fractions of the available features in each branching using both the entropy and the Gini impurity metric to determine the optimal branching. The number of features seems not to play a major role, and the entropy metric seems to perform slightly better in general.

of various numbers of decision trees in the forest is shown in figure 2.7. The performance seems to stagnate around 100 trees, and remain constant after that, so I usually used at least 500 trees to make sure to get the optimal performance, as runtime wasn't an issue.

The robustness to irrelevant features and overfitting described earlier also plays a role in the application of random forests in conjunction with other schemes. SVMs as described in section 2.1 can be sensitive to irrelevant data[15]. There exist off-the-shelf methods, such as recursive feature elimination (RFE)[10], for use with linear SVMs, but to my knowledge, there is no 'standard' way to eliminate irrelevant features when using a non-linear kernel. However, it is possible to use a random forest approach to obtain the relative importance of the various features and then use only the most important ones in another machine learning scheme which is less tolerant to the inclusion of irrelevant data. The relative importance of feature j can be estimated by first constructing a random forest and evaluating its performance s, then randomly permuting the values of feature j across the training sample and measure the damage it does to

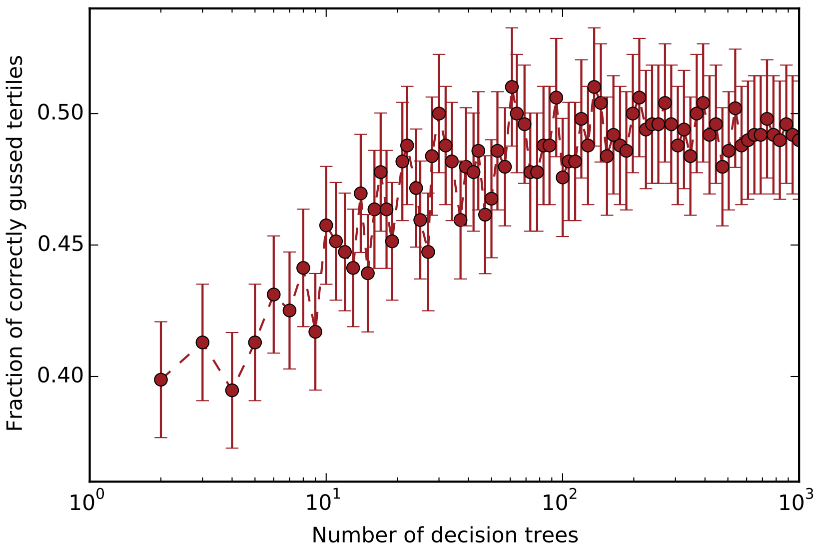


Figure 2.7: Example of a random forest performance versus number of decision trees. Performance seems to increase steadily until about 100 trees, then stagnate.

the performance of the forest by comparing with the permuted score s_p. The ratio of the mean to standard deviation of those differences:

$$w_j = \frac{\langle s - s_p \rangle}{\mathrm{std}(s - s_p)} \tag{2.54}$$

Random forests also provide a natural measure of similarity between data points. Given data points i and j these can be plugged into all their OOB decision trees, or a random subset thereof, and the fraction of the attempts in which both end up at the same leaf can be taken as a measure of similarity. This can be used to generate a proximity matrix for the data points, and it can be used as a metric for determining the nearest neighbours of a new point in conjunction with a simple nearest neighbour classifier.

2.3 Nearest Neighbour-classifiers

Do iiit!

Skriv en masse om den smart random forest NN-model.

2.4 Results

2.4.1 Big Five Personality Traits

HARJ

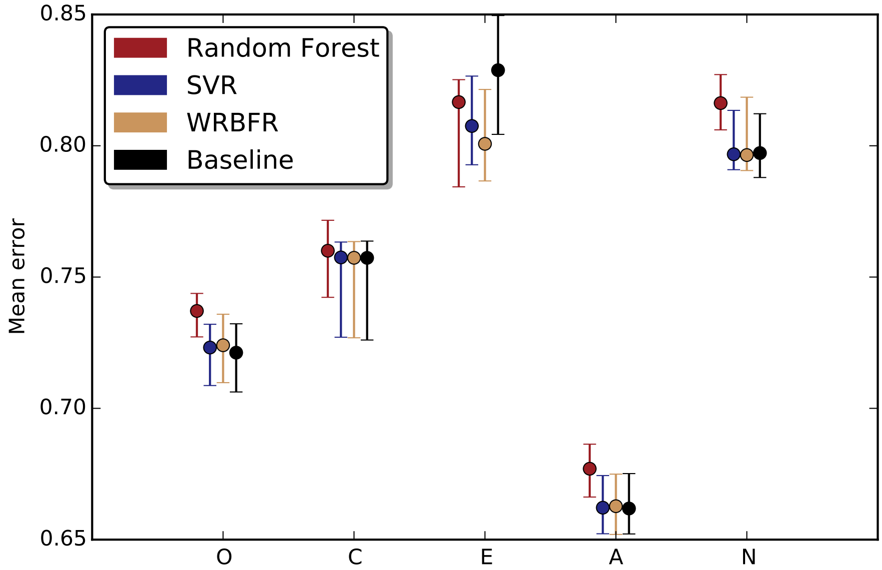


Figure 2.8: Comparison of performance of models using random forest, support vector regression and weighted radial basis function regressor with a **baseline model** which always predicts the mean of the training sample. The y axis shows the mean error of each model and the error bars show the 95-percentile around the median scores obtained by running on 1000 bootstrap samples.

2.4.2 Miscellaneous Traits

Opdater med ML

Her står der ting om figur 2.9.

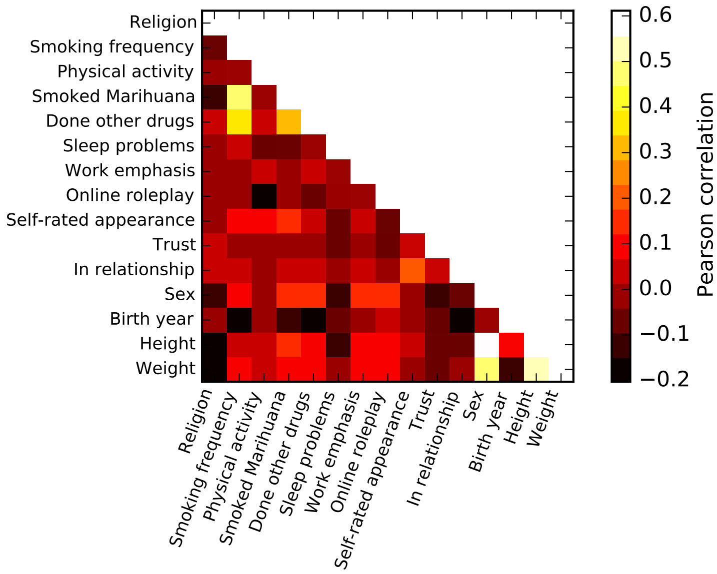


Figure 2.9: Triangle of Pearson correlation coefficients for various miscellaneous traits.

Part II

WIKIPEDIA-BASED EXPLICIT SEMANTIC ANALYSIS

Chapter 3

Wikipedia-based Explicit Semantic Analysis

NATURAL language processing has long been both a subject of interest and a source of great challenges in the field of artificial intelligence. ref The difficulty varies greatly depending with the different language processing tasks; certain problems, such as text categorization, are relatively straightforward to convert to a purely mathematical problem, which in turn can be solved by a computer, whereas other problems, such as computing semantic relatedness, necessitates a deeper understanding of a given text, and thus poses a greater problem. This sections aims firstly to give a brief introduction to some of the most prominent techniques used in language processing in order to explain my chosen method of explicit semantic analysis (ESA), and secondly to explain in detail my practical implementation of an ESA-based text interpretation scheme.

3.1 Methods

This section outlines a few methods used in natural language processing, going into some detail on ESA while touching briefly upon related techniques.

3.1.1 Bag-of-Words

ref

An example of a categorization problem is the 'bag of words' approach, which has seen use in spam filters.Here, text fragments are treated as unordered collections of words drawn from various bags, which in the case of spam filters would be undesired mails (spam) and desired mails (ham). By analysing large amounts of regular mail and spam, the probability of drawing each of the words constituting a given text from each bag can be computed, and the probability of the given text fragment representing draws from each bag can be computed using Bayesian statistics.

More formally, the text T is represented as a collection of words $T = \{w_1, w_2, \ldots, w_n\}$, and the probability of T actually representing draws from bag j is hence

$$P(B_j|T) = \frac{P(T|B_j)P(B_j)}{P(T)}, \tag{3.1}$$

$$= \frac{\prod_i P(w_i|B_j)P(B_j)}{\sum_j \prod_i P(w_i|B_j)P(B_j)}, \tag{3.2}$$

ref

for an arbitrary number of bags labelled by j. This method is simple and powerful whenever a text is expected to fall in one of several discrete categories (such as spam filters or language detection). However, for more complex tasks it proves lucrative to attempt instead to assign some kind of meaning to text fragments rather than to consider them analogous to lottery numbers or marbles. This notion of meaning will be elaborated on shortly, as it varies depending on the method of choice, but the overall idea is to ascribe to words a meaning which depends not only on the word itself, but also on the connection between the word and and existing repository of knowledge. The reader may think of this as mimicking the reading comprehension of humans. In itself, the word 'dog' for instance, contains a mere 24 bits of information if stored with a standard encoding, yet a human reader immediately associates a rich amount of existing knowledge to the word, such as dogs being mammals, related to wolves, being a common household pet, etc. The objective of both explicit and latent semantic analysis is to establish a high-dimensional 'concept space' in which words and text fragments are represented as vectors. The difference between explicit and latent semantic analysis is the method used to obtain said concepts, a explained in the following sections.

3.1.2 Semantic Analysis

Salton et al proposed in their 1975 paper *A Vector Space Model for Automatic Indexing*[16] an approach where words and text fragments are mapped with a linear transformation to vectors in a high-dimensional concept space,

$$T \to |V\rangle = \sum_i v_i |i\rangle, v_i \in \mathbb{R}, \tag{3.3}$$

where a similarity measure of two texts can be defined as the inner product of two normalized such vectors,

$$S(V, W) = \langle \hat{V} | \hat{W} \rangle = \frac{\sum_i v_i w_i}{\left(\sum_i v_i^2\right)\left(\sum_i w_i^2\right)}, \tag{3.4}$$

and the cosine quasi-distance can be considered as a measure of semantic distance between texts:

$$D(V, W) = 1 - S(V, W). \tag{3.5}$$

This approach has later seen use in the methods of Latent Semantic Analysis (LSA) and Explicit Semantic Analysis (ESA). Both methods can be said to mimic human cognition in the sense that the transformation from (3.3) is viewed as a mapping of a text fragment to a predefined *concept space* and thus, processing of texts relies heavily on external repositories of knowledge.

The difference between LSA and ESA is how the concept space is established. Although I have used solely ESA for this project, I will give an extremely brief overview of LSA for completeness following (Landauer 1998 [12]). LSA constructs its concept space by first extracting every unique word encountered in a large collection of text corpora and essentially uses the leading eigenvectors (i.e. corresponding to the largest eigenvalues) of the word-word covariance matrix as the basis vectors of its conceptual space. This is the sense in which the concept are latent - rather than interpret text in terms of explicit concepts, such as 'healthcare', LSA would discover correlations between words such as 'doctor', 'surgery' etc. and consider that a latent concept. Owing to the tradeoff between performance and computational complexity, only about 400 such vectors are kept[14]. In psychology, LSA has been proposed as a possible model of fundamental human language acquisition as it provides computers a way of estimating e.g. word-word relatedness (a task which LSA does decently) using nothing but patterns discovered in the language it encounters[12].

In contrast, the concepts in ESA correspond directly to certain parts of the external text corpora one has employed to construct a semantic analyser. Concretely, the matrix playing the role of the reduced covariance matrix in LSA has columns corresponding to each text corpus used and rows corresponding to individual terms or words, with the value of each matrix element denoting some measure of relatedness between the designated word and concept. I have used the English Wikipedia, so naturally each concept consists of an article, although the process could easily be tuned to be more or less fine-grained and associate instead each concept with e.g. a subsection or a category, respectively. Of course, a wholly different collection of texts could also be used - for instance a version of ESA more suited to compare the style or period of literary works could be constructed using a large collection of literature such as the Gutenberg Project. However, with no prior knowledge of the subject matter of the text to be analysed, Wikipedia seems like a good all-round solution considering its versatility and the massive numbers of volunteers constantly keeping it up to date.

I wish to point out two advantages of ESA over LSA. First, it is more successful, at least using the currently available text corpora and implementation techniques. A standard way of evaluating the performance of a natural language processing program is to measure the correlation between relatedness scores assigned to pairs of words or text fragments by the computer and by human judges. In both disciplines, ESA has outperformed LSA since it was first implemented ([9], p 457).

Second, the concepts employed in ESA are directly understandable by a human reader, whereas the concepts in LSA correspond to the leading moments of the covariance matrix. For example, to test whether the first semantic analyser built by my program behaved reasonably, I fed it a snippet of a news article on CNN with the headline "*In Jerusalem, the 'auto intifada' is far from an uprising*". This returned an ordered list of top scoring concepts as follows: "Hamas, Second Intifada, Palestinian National Authority, Shuafat, Gaza War (2008-09), Jerusalem, Gaza Strip, Arab Peace Initiative, Yasser Arafat, Israel, West Bank, Temple Mount, Western Wall, Mahmoud Abbas", which seems a very reasonable output.

3.2 Constructing a Semantic Analyser

The process of applying ESA to a certain problem may be considered as the two separate subtasks of first a very computationally intensive construction af the machinery required to perform ESA, followed by the application of said machinery to some collection of texts. For clarity, I'll limit the present section to the details of the former subtask while description its application and results in section 3.3.

The construction itself is divided into three steps which are run in succession to create the desired machinery. The following is a very brief overview of these steps, each of which is elaborated upon in the following subsections.

1. First, a full Wikipedia XML-dump[1] is parsed to a collection of files each of which contains the relevant information on a number of articles. This includes the article contents in plaintext along with some metadata such as designated categories, inter-article link data etc.

2. Then, the information on each concept (article) is evaluated according to some predefined criteria, and concepts thus deemed inferior are purged from the files. Furthermore, two list are generated and saved, which map unique concepts and words, respectively, to an integer, the combination of which is to designate the relevant row and column in the final matrix. For example, the concept 'Horse' corresponded to column 699221 in my matrix, while the word 'horse' corresponded to row 11533476.

3. Finally, a large sparse matrix containing relevance scores for each word-concept pair is built and, optionally, pruned (a process used to remove 'background noise' from common words as explained in section 3.2.3)

These steps are elaborated upon in the following.

3.2.1 XML Parsing

The Wikipedia dump comes in a rather large (~50GB unpacked for the version I used) XML file which must be parsed to extract each article's contents and relevant information. This file is essentially a very long list of nested fields where the data type in each field is denoted by an XML tag, such as `<text> blabla </text>`. A very simplified example of a field for one Wikipedia article is shown in 3.1 The content of each field has already been sanitised by Wikipedia so that if for instance the symbol '<' is entered into an article, it is instead represented as '&lt' in the XML file. To this end, I wrote a SAX parser, which processes the dump sequentially to accommodate its large size. When running, the parser walks through the file and sends the various elements it encounters to a suitable processing function depending on the currently open XML tag. For example, when a 'title' tag is encountered, a callback method is triggered which assigns

[1] These are periodically released at http://dumps.wikimedia.org/enwiki/

```
<page>
  < title >Horse</title>
  <ns>0</ns>
  <id>12</id>
  <revision>
    <id>619093743</id>
    <parentid>618899706</parentid>
    <timestamp>2014−07−30T07:26:05Z</timestamp>
    <contributor>
      <username>Eduen</username>
      <id>7527773</id>
    </contributor>
    <text xml:space="preserve">
===Section title===
[[Image:name_of_image_file] Image caption]]]
Lots of educational text containing, among other things links to [[other article | text to display ]].
</text>
    <sha1>n57mnhttuhxxpq1nanak3zhmmmcl622</sha1>
    <model>wikitext</model>
    <format>text/x−wiki</format>
  </revision>
</page>
```

Snippet 3.1: A simplified snippet, of a Wikipedia XML dump.

a column number to the article title and adds it to the list of processed articles. For the callback method for processing the 'text' fields, I used a bit of code from the pre-existing Wikiextractor project to remove Wiki markup language (such as links to other articles being displayed with square brackets and the like) from the content. This code is included in section A.2.5. The remainder of the parser is my work, and is included in section A.2.1. Throughout this process, the parser keeps a list of unique words encountered as well as outgoing link information for each article. These lists, along with the article contents, are saved to files each time a set number of articles have been processed. The link information is also kept in a hashmap with target articles as keys and a set articles linking to the target as values. The point of this is to reduce the computational complexity of the link processing as detailed in the following section.

3.2.2 Index Generation

The next step reads in the link information previously saved and adds it as ingoing link information in the respective article content files. The point of this approach is that link information is initially saved as a hashmap so the link going to a given article can be found quickly, rather that having to search for outgoing links in every other article to determine the ingoing links to each article, which would be of $\mathcal{O}(n^2)$ complexity.

Following that, articles with sufficiently few words and/or ingoing/-outgoing links are discarded an index lists for the remaining articles and words are generated to associate a unique row/column number with each word/concept pair. The code performing the step described here is included in section A.2.2.

3.2.3 Matrix Construction

This final step converts the information compiled in the previous steps into a very large sparse matrix. The program allows for this to be done in 'chunks' in order to avoid insane RAM usage. Similarly, the matrix is stored in segments with each file containing a set number of rows in order to avoid loading the entire matrix to interpret a short text.

The full matrix is initially constructed using a DOK (dictionary of keys) sparse format in which the i, jth element simply counts the number of occurrences of word i in the article corresponding to concept j. This is denoted $\text{count}(w_i, c_j)$. The DOK format as a hashmap using tuples (i, j) as keys and the corresponding matrix elements as values and is the fastest format available for element-wise construction. The matrix is subsequently converted to CSR (compressed sparse row) format, which allows faster operations on rows which performs much quicker when computing TF-IDF (term frequency - inverse document frequency) scores and extracting concept vectors from words, i.e. when accessing separate rows corresponding to certain words.

Each non-zero entry is then converted to a TF-IDF score according to

$$T_{ij} = \left(1 + \ln\left(\text{count}(w_i, c_j)\right)\right) \ln\left(\frac{n_c}{df_i}\right), \tag{3.6}$$

where n_c is the total number of concepts and

$$df_i = |\{c_k, w_i \in c_k\}| \tag{3.7}$$

is the number of concepts whose corresponding article contains the ith word. Thus, the first part of (3.6), $1 + \ln\left(\text{count}(w_i, c_j)\right)$ is the *text frequency* term, as it increases with the frequency of word i in document j. Similarly, $\ln\left(\frac{n_c}{df_i}\right)$ in (3.6) is the *inverse document frequency* term as it decreases with the frequency of documents containing word i. Thus, the TF-IDF score as somewhat complement to entropy in that it goes to zero as the fraction Giver det mening?
of documents containing word i goes to 1, and takes its highest values if

word i occurs with high frequency in only a few documents[13]. While (3.6) is not the only expression to have those properties, empirically it tends to achieve superior results in information retrieval[17].

Each row is then L^2 normalized (divided by their Euclidean norm):

$$T_{ij} \rightarrow \frac{T_{ij}}{\sqrt{\sum_i T_{ij}^2}}. \tag{3.8}$$

Finally, each row is pruned to reduce spurious associations between concepts and articles with a somewhat uniform occurrence rate. This was done in practice by following the pragmatic approach of Gabrilovich [9] of sorting the entries of each row, move a sliding window across the entries, truncating when the falloff drops below a set threshold and finally reversing the sorting. The result of this step is the matrix which computes the interpretation vectors as described in 3.1.2. The code is included in section A.2.3.

Hearst nævner noget offentligt tilgængeligt Reuters-data som folk øver tekstklassifikation på. Det kunne være ret sjovt. Det kunne være sjovt at lave 'semantisk nearest neighbor'

3.3 Applications & Results

Having constructed a necessary machinery, I wrote a small Python module to provide an easy-to-use interface with the output from the computations described earlier. The code for this is included in section A.2.4. The module consists mainly of a SemanticAnalyser class, which loads in the previously mentioned index lists and provides methods for various computations such as estimating the most relevant concepts for a text, determining semantic distance etc. For example, the following code will create a semantic analyser instance and use it to guess the topic of the input string:

```
sa = SemanticAnalyser()
sa.interpret_text("Physicist from Austria known for the theory of relativity")
```

This returns a sorted list of the basis concepts best matching the input string, where the first element is of course 'Albert Einstein'. The SemanticAnalyser class contains equally simple methods to interpret a target text file or keyboard input, to calculate the semantic similarity or cosine distance between texts, and to compute interpretations vectors from a text.

The same module contains a TweetHarvester class which I wrote in order to obtain a large number of tweets to test the semantic analyser on, as tweets are both numerous and timestamped, which allows investigations of the temporal evolution of tweets matching a given search term. The

TweetHarvester class provides an equally simply interface - for instances, the 100 most recent tweets regarding a list of companies can be mined and printed by typing

```
terms = ['google', 'carlsberg', 'starbucks']
th = TweetHarvester()
th.mine(terms, 100)
print th.harvested_tweets
```

in addition to actively 'mining' for tweets matching a given query, the class can also passively 'listen' for tweets while automatically saving its held tweets to a generated date-specific filename once a set limit of held tweets is exceeded:

```
th = TweetHarvester(tweets_pr_file = 100)
th.listen(terms)
```

The downloaded tweets are stored as tweet objects which contain a built in method to convert to a JSON-serializable hashmap, an example of which is provided in 3.2. As can be seen in the example, the tweet object contains not only the tweets textual content but also a wide range of metadata such as the hashtags contained in the tweet, users mentioned, time of creation, language etc. Using this, I wrote a script that harvested tweets mentioning some selected brand names for about 5 months which resulted in a massive dataset of 370 million tweets. This section demonstrates some applications of the methods outlined earlier to this dataset.

```
{u'contributors': None,
 u'coordinates': None,
 u'created_at': u'Wed Jun 03 12:16:23 +0000 2015',
 u'entities': {u'hashtags': [],
               u'symbols': [],
               u'urls': [],
               u'user_mentions': [{u'id': 630908729,
                                   u'id_str': u'630908729',
                                   u'indices': [0, 12],
                                   u'name': u'Alexandra White ',
                                   u'screen_name': u'lexywhite86'}]},
 u'favorite_count': 0,
 u'favorited': False,
 u'geo': None,
 u'id': 606071930282745857L,
 u'id_str': u'606071930282745857',
 u'in_reply_to_screen_name': u'lexywhite86',
 u'in_reply_to_status_id': 605991263129714688L,
 u'in_reply_to_status_id_str': u'605991263129714688',
 u'in_reply_to_user_id': 630908729,
 u'in_reply_to_user_id_str': u'630908729',
 u'is_quote_status': False,
 u'lang': u'en',
 u'metadata': {u'iso_language_code': u'en', u'result_type': u'recent'},
 u'place': None,
 u'retweet_count': 0,
 u'retweeted': False,
 u'source': u'<a href="http://twitter.com/download/iphone" rel="nofollow">Twitter for iPhone
     </a>',
 u'text': u'@lexywhite86 if carlsberg did mornings \U0001f602',
 u'truncated': False,
 u'user': {u'contributors_enabled': False,
           u'created_at': u'Thu Apr 05 13:48:00 +0000 2012',
           u'description': u'',
           u'favourites_count': 169,
           u'follow_request_sent': False,
           u'followers_count': 110,
           u'friends_count': 268,
           u'geo_enabled': False,
           u'id': 545986280,
           u'id_str': u'545986280',
           u'lang': u'en',
           u'listed_count': 0,
           u'location': u'',
           u'name': u'Robert Murphy',
           u'notifications': False,
           u'protected': False,
           u'screen_name': u'7Robmurphy',
           u'statuses_count': 1650,
           u'time_zone': None,
           u'url': None,
           u'utc_offset': None,
           u'verified': False}}
```


Snippet 3.2: Example of a downloaded tweet.

3.3.1 Trend Discovery and Monitoring

A simple application of the methods outlined above is a purely exploratory analysis. As an example, using code included in section A.2.6, I extracted the 10 concepts most closely related to every tweet mentioning Carlsberg for each week over a period of a few months. This gave a bar chart like figure 3.1 for each of those weeks. Figure 3.1 implies some significant relation between Carlsberg and Raidió Teilifís Éireann blablabla!!!

Erstat med det med den irske radio

Another application is to manually select a few concepts of interest and then monitor how the number of tweets strongly related to them develops over time. For example I did an exploratory analysis as described above and occasionally saw the concept 'Kim Little' surface. This turned out to be a young football player signed to a team sponsored by Carlsberg, and when I produced a series of bar charts like that shown in figure 3.2 and combined them into an animation, the bar representing Little tended to peak around dates on which news stories featuring her or her team could be found. Of course these method are only effective insofar the concepts of interest actually have an associated Wikipedia article. However, these concepts are merely the basis vectors of the semantic space in which an arbitrary text can be represented, so this procedure can be extended fairly elegantly to estimate the impact of e.g. a press release on social media.

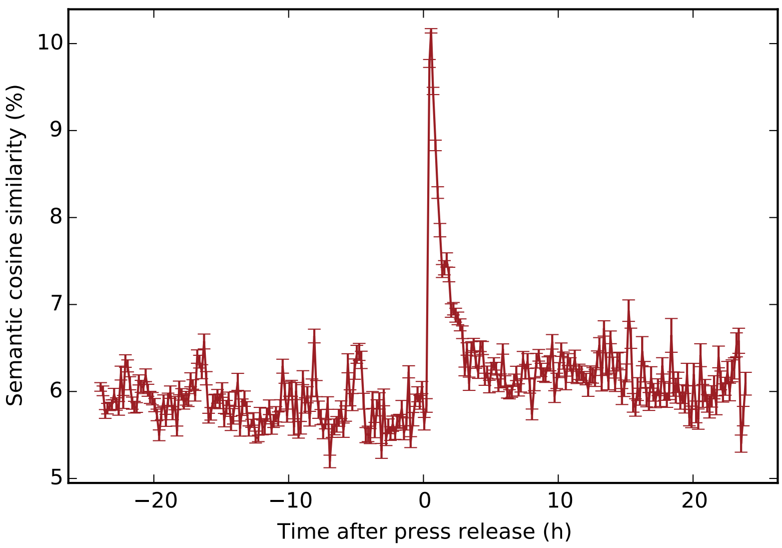


Figure 3.1: Sæt det rigtige ind når du kommer hjem!!!

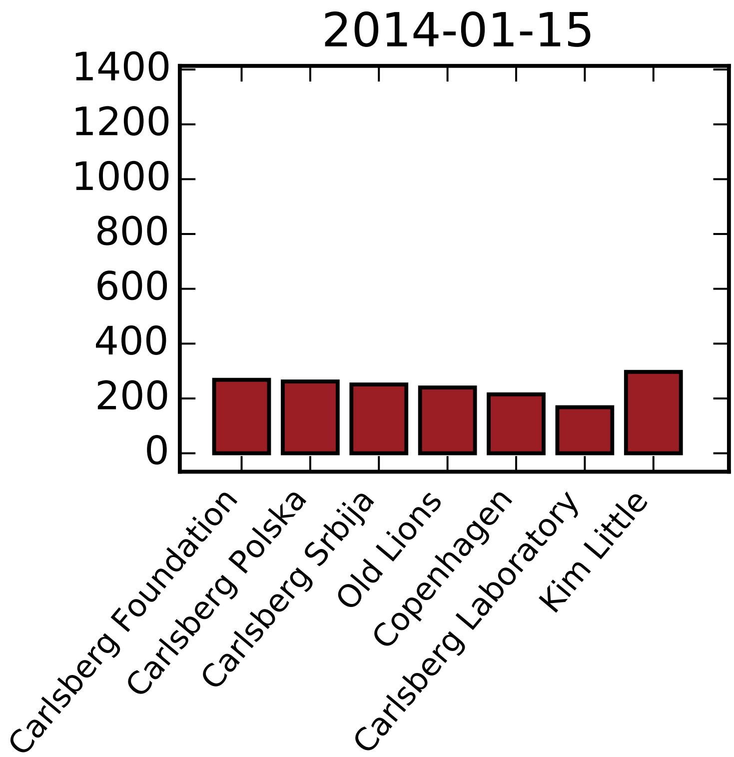


Figure 3.2: Bar chart showing the concepts most related to tweets about Carlsberg for January 15th 2014.

This is the subject of section 3.3.2.

3.3.2 Measuring Social Media Impact

One possible application of the software I wrote to perform ESA is to provide a quantitative measure of the impact of some event on social media. For instance, some corporation or organization might be interested in learning precisely effectively a campaign or press release has reached the general public as it is expressed by social media. While my tweet harvesting script was running, Google posted a blog entry[2] on their experiments with producing psychedelic images with deep neural network called *Deep Dreams* which received widespread attention on social media.

Using the text from the Deep Dreams blog post as a reference text, I converted each English tweet about Google from a period around the blog post to a semantic vector using (3.3) and computed their semantic cosine similarity with the reference text as described by (3.4). Figure 3.3 shows this behaviour in the time around the release of the blog post. A measure of impact should of course take into account the rate at which new tweets occur. Just as one would expect, not only semantic relatedness, but also tweet frequency increase drastically around an interesting event. To make the signal from figure 3.3 independent of the normal tweet frequency, yet sensitive to changes in it, I first find the difference in semantic relatedness between the signal and a reference signal obtained long before the event to be investigated, then modulate the signal by multiplying each bin with the activity A given by the ratio between the current and baseline tweet frequency. As normal Twitter activity tends to vary greatly over a 24 hour period, I obtained a baseline tweet frequency by computing the average tweet rate, weighted by the activity, for each discreet time interval of the day, and then repeating that signal into the period after the event of interest. The baseline along with the observed tweet rate is shown in figure 3.4(a). Modulating in this fashion and computing the difference between the observed signal and the baseline gives an expression of the 'impact rate' of a given time bin, which can be integrated to obtain a measure of the total impact which is independent of the average tweet rate before publication, which is practical if one wishes to compare e.g. the success of media campaigns for companies or organizations of varying sizes. This is shown in figure 3.5. If one wishes to include the number of tweets posted in the impact measure, the result can be computed and visualized nicely by again obtaining a baseline cosine similarity from a period prior to publication and then considering e.g. the cumulative deviation from that following the release, as shown in figure 3.6.

[2] The original blog post is available at http://googleresearch.blogspot.dk/2015/06/inceptionism-going-deeper-into-neural.html

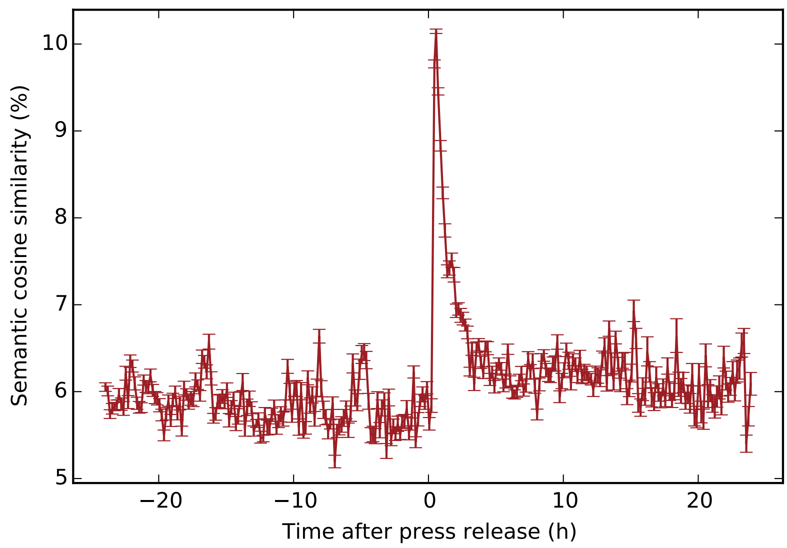
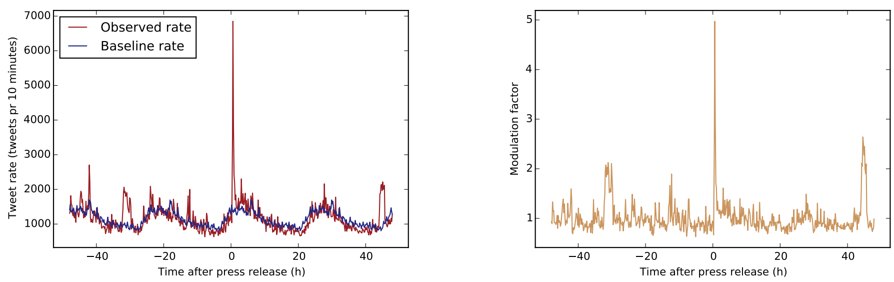


Figure 3.3: Graph of the mean semantic cosine similarity of tweets around the Deep Dreams press release. There is a clear peak around $t = 0$ and the similarity appears increased in the period following the release. The error bars represent the true standard deviation of the sample mean $\sigma/\sqrt{N}$ for each time bin, each representing a 10-minute interval.



(a) (b)

Figure 3.4: Tweet activity around the time the Deep Dreams blog entry was posted. The signal tweet rate increases with a factor of about 5 relative to the baseline rate around the post. Figure 3.4(b) shows the resulting modulation factor to the signal from in figure 3.3.

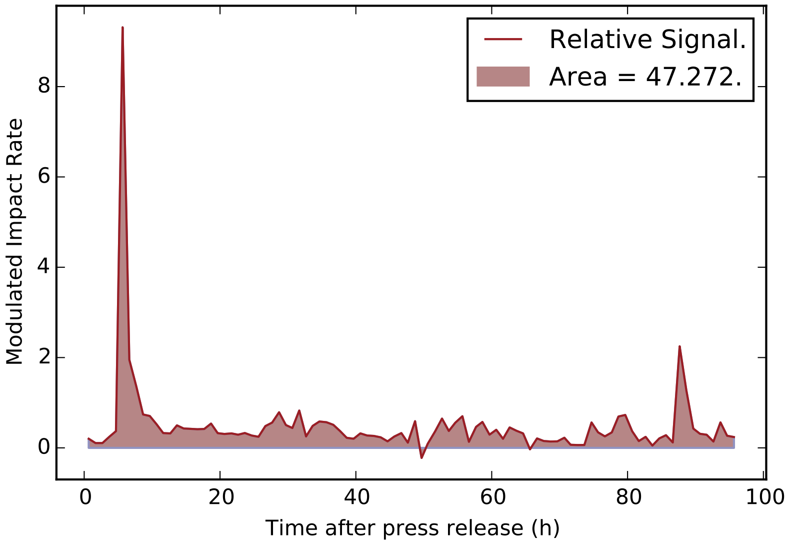


Figure 3.5: A measure of the total impact on Twitter by a press release can be obtained by integrating over the difference between a signal corresponding of the average tweet cosine similarity modulated by the relative activity, and a corresponding baseline. This measure does not depend on the typical Twitter activity and so can be used to compare the effectiveness of campaigns of varying size.

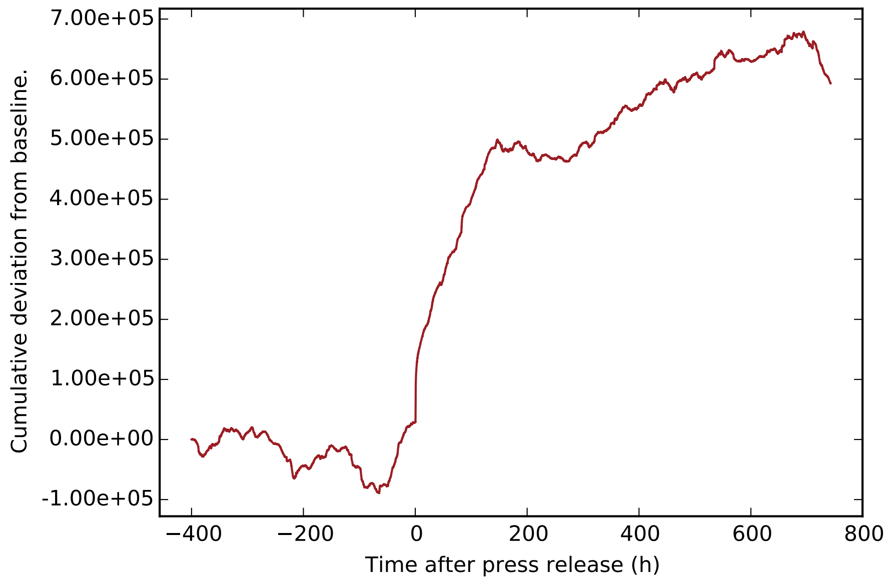


Figure 3.6: The cumulative deviation from the mean cosine similarity between the Deep Dreams post and tweets in the period around its release.

Appendix A

Appendix

A.1 Social Fabric-related Code

This section contains the code referred to in part I of the thesis.

A.1.1 Phonetools

```
# -*- coding: utf-8 -*-
"""
Created on Tue Dec 02 15:12:29 2014

@author: Bjarke
"""

import datetime
from pkg_resources import resource_filename
from ast import literal_eval as LE
import numpy as np
import json
import os

def _global_path(path):
    '''Helper method to ensure that data files belonging to the social_fabric
     module are available both when importing the module and when running
     individual parts from it for testing.
     usage: Always use _global_path(somefile) rather than somefile'''
    if __name__ == '__main__':
        return path
    else:
        return resource_filename('social_fabric', path)

def make_filename(prefix, interval, bin_size, ext=''):
    '''Returns a filename like "call_out-intervalsize-binsize_N", where N is
     an int so files aren't accidentally overwritten.
     I've been known to do that.'''
    stem = "%s-int%s-bin%s" % (prefix, interval, bin_size)
    n = 0
    ext = '.'+ext
    attempt = stem+ext
    while os.path.isfile(attempt):
        n+=1
        attempt = stem+"_"+str(n)+ext
    return attempt

def unix2str(unixtime):
    '''Converts timestamp to datetime object'''
    dt = datetime.datetime.fromtimestamp(unixtime)
    return str(dt)

#Converts bluetooth MAC-addresses to users
with open(_global_path('user_mappings/bt_to_user.txt'), 'r') as f:
    bt2user = LE(f.read())

#Converts phone 'number' code to users
with open(_global_path('user_mappings/phonenumbers.txt'), 'r') as f:
    number2user = LE(f.read())

#Converts IDs from psychological profile to  users
```

```
with open(_global_path('user_mappings/user_mapping.txt'), 'r') as f:
    psych2user = {}
    #This is in tab-separated values, for some reason.
    for line in f.read().splitlines():
        (ID, usercode) = line.split('\t')
        assert ID.startswith('user_')
        psych2user[ID] = usercode

#Converts users to info on their psych profiles
with open(_global_path('user_mappings/user2profile.json'), 'r') as f:
    user2profile = json.load(f)

def is_valid_call(call_dict):
    '''Determine whether an entry is 'valid', i.e. make sure user isn't
     calling/texting themselves, which people apparently do...'''
    caller = call_dict['user']
    try:
        receiver = number2user[call_dict['number']]
    except KeyError:
        receiver = None
    return caller != receiver

def readncheck(path):
    '''Reads in all valid call info from the file at path'''
    try:
        with open(path,'r') as f:
            raw = [LE(line) for line in f.readlines()]
    except IOError:
        return []  #no file :(
    #File read. Return proper calls
    return [call for call in raw if is_valid_call(call)]

class Binarray(list):
    '''Custom array type to automatically bin the time around a set center
     and place elements in each bin.
     The array can be centered using Binarray.center = <some time>.
     After centering, timestamps can be placed in bins around the center with
     Binarray.place_event(<some other time>).'''

    def __init__(self, interval = 3*60**2, bin_size = 3*60, center = None,
                 initial_values = None):
        '''
         Args:
         ----------------------
         interval : int
           Total number of seconds covered by the Binarray.

         bin_size : int
           Width of each bin measured in seconds. The total interval must be
           an integer multiplum of the bin size.

         center : int
           Where to center the Binarray. If the array is centered at time t,
           any event placed in it will placed in a bin depending on how long
           before or after t the event occured.

         initial_values : list
           List of values to start the Binarray with. Default is zeroes.'''


        #Make sure interval is an integer multiplum of bins
        if not interval % bin_size == 0:
```

```
            suggest = interval - interval % bin_size
            error = "Interval isn't an integer multiple of bin size. \
              Consider changing interval to %s." % suggest
            raise ValueError(error)

        #Set parameters
        self.bin_size = bin_size
        self.interval = interval
        self.size = 2*int(interval/bin_size)
        self.centerindex = int(self.size//2)
        self.center = center
        #Keep track of how many events missed the bins completely
        self.misses = 0
        #Call parent constructor
        if not initial_values:
            startlist = [0]*self.size
        else:
            if not len(initial_values) == self.size:
                msg = '''Array of start value must have length %d. Tried to
                  instantiate with length of %d.''' % (self.size,
                                                       len(initial_values))
                raise ValueError(msg)
            startlist = initial_values
        super(Binarray, self).__init__(startlist)


    def place_event(self, position):
        '''Places one count in the appropriate bin if event falls within
          <interval> of <center>. Returns True on success.'''
        if self.center == None:
            raise TypeError('Center must be set!')
        delta = position - self.center
        #Check if event is outside current interval
        if np.abs(delta) >= self.interval:
            self.misses += 1
            return False
        #Woo, we're in the correct interval
        index = int(delta//self.bin_size) #relative to middle of array
        self[self.centerindex + index] += 1
        return True

    def normalized(self):
        '''Returns a normalized copy of the array's contents.'''
        events = sum(self) + self.misses
        #Use numpy vectorized function for increased speed.
        f = np.vectorize(lambda x: x*1.0/events if events else 0)
        return f(self)

    def _todict(self):
        '''Helper method to allow dumping to JSON format.'''
        attrs = ['misses', 'interval', 'bin_size']
        d = {att : self.__getattribute__(att) for att in attrs}
        d['values'] = list(self)
        d['type'] = 'binarray'
        return d


def _dumphelper(obj):
    '''Evil recursive helper method to convert various nested objects to
     a JSON-serializeable format.
     This should only be called by the dump method!'''
    if isinstance(obj, Binarray):
```

```
        d = obj._todict()
        return _dumphelper(d)
    elif isinstance(obj, tuple):
        hurrayimhelping = [_dumphelper(elem) for elem in obj]
        return {'type' : 'tuple', 'values' : hurrayimhelping}
    elif isinstance(obj, dict):
        temp = {'type' : 'dict'}
        contents = [{'key' : _dumphelper(key), 'value' : _dumphelper(value)}
                    for key, value in obj.iteritems()]
        temp['contents'] = contents
        return temp
    #Do nothing if obj is an unrecognized type. Let JSON raise errors.
    else:
        return obj

def _hook(obj):
    '''Evil recursive object hook method to reconstruct various nested
     objects from a JSON dump.
     This should only be called by the load method!'''
    if isinstance(obj, (unicode, str)):
        try:
            return _hook(LE(obj))
        except ValueError: #happens for simple strings that don't need eval
            return obj
    elif isinstance(obj, dict):
        if not 'type' in obj:
            raise KeyError('Missing type info')
        if obj['type'] == 'dict':
            contents = obj['contents']
            d = {_hook(e['key']) : _hook(e['value']) for e in contents}
            #Make sure we also catch nested expressions
            if 'type' in d:
                return _hook(d)
            else:
                return d
        elif obj['type'] == 'binarray':
            instance = Binarray(initial_values = obj['values'],
                                bin_size = obj['bin_size'],
                                interval = obj['interval'])
            instance.misses = obj['misses']
            for key, val in obj.iteritems():
                if key == 'values':
                    continue
                instance.__setattr__(key, val)
            return instance

        elif obj['type'] == 'tuple':
            #Hook elements individually, then convert back to tuple
            restored = [_hook(elem) for elem in obj['values']]
            return tuple(restored)
        else:
            temp = {}
            for k, v in obj.iteritems():
                k = _hook(k)
                temp[k] = _hook(v)
            return temp
        #
    #Do nothing if obj is an unrecognized type
    else:
        return obj

def load(file_handle):
```

```
    '''Reads in json serialized nested combinations of dicts, binarrays
     and tuples.'''
    temp = json.load(file_handle, encoding='utf-8')
    return _hook(temp)

def dump(obj, file_handle):
    '''json serializes nested combinations of dicts, binarrays
     and tuples.'''
    json.dump(_dumphelper(obj), file_handle, indent=4, encoding='utf-8')


if __name__=='__main__':
    from time import time
    from random import randint
    #Create Binarray with interval +/- one hour and bin size ten minutes.
    ba = Binarray(interval = 60*60, bin_size = 10*60)
    #Center it on the present
    now = int(time())
    ba.center = now
    #Generate some timestamps around the present
    new_times = [now + randint(-60*60, 60*60) for _ in xrange(100)]
    for tt in new_times:
        ba.place_event(tt)

    #Save it
    with open('filename.sig', 'w') as f:
        dump(ba, f)

    print ba
```

A.1.2 Code Communication Dynamics

Code for extracting Bluetooth signals

```
# -*- coding: utf-8 -*-
from __future__ import division

import os
import glob
import itertools
import random
from ast import literal_eval as LE
from social_fabric.phonetools import readncheck, Binarray, dump, make_filename

#==========================================================================
# Parameters
#==========================================================================

#Grap all the files we need to read
userfile_path = "userfiles/" #linux


#Interval of interest and bin size (seconds)
interval = 12*60**2
bin_size = 10*60

#How many other users contribute to background for a given user
number_of_background_samples = 1

#Number of users to analyse. Use None or 0 to include everyone.
max_users = 2

#Number of repeated signals required to be considered social
social_threshold = 2

#==========================================================================
# Functions
#==========================================================================

def call_type(n):
    if n == 1:
        return 'call_in'
    elif n == 2:
        return 'call_out'
    else:
        return None

def text_type(n):
    if n == 1:
        return 'text_in'
    elif n == 2:
        return 'text_out'
    else:
        return None


def user2social_times(user):
    '''Converts user code to a list of times when the user was social i.e. had
     two or more repeated Bluetooth signals.'''
```

```
    with open(userfile_path+user+"/bluetooth_log.txt", 'r') as f:
        raw = [LE(line) for line in f.readlines()]
    #List of times when user was social
    social_times = []
    #Temporary variables
    current_time = 0
    previous_time = 0
    current_users = []
    previous_users = []
    for ind in xrange(len(raw)-1):
        signal = raw[ind]
        new_time = signal['timestamp']
        #Check if line represents a new signal and if so, update values
        if new_time != current_time:
            #Determine if previous signal was social and append to results
            overlap = set(previous_users).intersection(set(current_users))
            if len(overlap) >= social_threshold:
                social_times.append(previous_time)
            #Update variables
            previous_time = current_time
            previous_users = current_users
            current_users = []
            current_time = new_time
        new_user = signal['name']
        if new_user=='-1' or not new_user:
            continue
        else:
            current_users.append(new_user)
        #
    return social_times

#==============================================================================
# Time to crunch some numbers
#==============================================================================

user_folders = [f for f in glob.glob(userfile_path+"/*")
                if os.path.isfile(f+"/call_log.txt")
                and os.path.isfile(f+"/sms_log.txt")
                and os.path.isfile(f+"/bluetooth_log.txt")]
users = [folder.split(userfile_path)[1] for folder in user_folders]

if not users:
    raise IOError('Found no users. Check userfile path.')

if max_users:
    users = users[:max_users]

trigger = 'bluetooth'
events = ['call_out', 'call_in', 'text_in', 'text_out']
pairs = [p for p in itertools.product([trigger], events)]

activity = {p : Binarray(interval,bin_size) for p in pairs}
background = {p : Binarray(interval,bin_size) for p in pairs}


#Read in data
call_data = {user : readncheck(userfile_path+user+"/call_log.txt")
             for user in users}
text_data = {user : readncheck(userfile_path+user+"/sms_log.txt")
             for user in users}

count = 0
```

```

for user in users:
    count += 1
    print "Analyzing user %s out of %s. Code: %s" % (count, len(users), user)

    #Get user data
    user_calls = call_data[user]
    user_texts = text_data[user]
    user_social_times = user2social_times(user)

    if not user_social_times:
        continue

    #Get background data
    others = []
    other_social_times = []
    while len(others) < number_of_background_samples:
        temp = random.choice(users)
        if not (temp in others or temp == user):
            newstuff = user2social_times(temp)
            if not newstuff:
                continue
            others.append(temp)
            other_social_times += newstuff

    #Determine the interval in which we have data on the current user
    first = min(user_social_times)
    last = max(user_social_times)

    #==========================================================================
    #      #Establish activity signal
    #==========================================================================
    for time in user_social_times:
        for e in events:
            activity[(trigger, e)].center = time

        for user_call in user_calls:
            event = call_type(user_call['type'])
            if not event:
                continue
            time = user_call['timestamp']
            activity[(trigger, event)].place_event(time)

        for user_text in user_texts:
            event = text_type(user_text['type'])
            if not event:
                continue
            time = user_text['timestamp']
            activity[(trigger, event)].place_event(time)

    #==========================================================================
    #      #Establish background signal
    #==========================================================================
    for other_time in other_social_times:
        #Reposition the relevant binarrays
        if not first <= other_time <= last:
            continue
        for e in events:
            background[(trigger, e)].center = other_time

        #Determine call background
        for user_call in user_calls or []:
```

```
            event = call_type(user_call['type'])
            if not event:
                continue
            time = user_call['timestamp']
            background[(trigger,event)].place_event(time)

        #Determine text background
        for user_text in user_texts or []:
            event = text_type(user_text['type'])
            if not event:
                continue
            time = user_text['timestamp']
            background[(trigger,event)].place_event(time)
        #
    #
#

#===========================================================================
# Done. Save signals
#===========================================================================

#Make a filename for the output file
filename = make_filename(prefix = trigger, interval=interval,
                         bin_size=bin_size, ext = 'json')
with open(filename, 'w') as f:
    dump((activity, background), f)

print "Saved to "+filename
```

Code for loading and plotting Bluetoot data

```
# −*− coding: utf−8 −*−
from __future__ import division

import glob
import numpy as np
import matplotlib.pyplot as plt

from social_fabric.phonetools import (load, make_filename)

#==========================================================================
display = True
#These are just default − they're updated when data is read
interval = 12*60**2
bin_size = 10*60
#==========================================================================


#Numpy−compliant method to convert activity and background to relative signal
get_signal = np.vectorize(lambda act, back : act * 1.0/back if back else 0)

def read_data(filename):
    '''Reads in data to plot. Updates the interval and bin sizes parameters.
     Returns data to be plotted as a hashmap with the following structure:
     {trigger : {event : signal}}.'''
    data = {}
    with open(filename, 'r') as f:
        (a,b) = load(f)
    first = True
    for key in a.keys():
        #Update interval and bin info
        if first:
            global interval
            global bin_size
            interval = a[key].interval
            bin_size = a[key].bin_size
            first = False

        (trigger, event) = key
        act = a[key].normalized()
        back = b[key].normalized()
        signal = get_signal(list(act), list(back))
        if not trigger in data:
            data[trigger] = {}
        data[trigger][event] = signal
    return data


def make_plot(trigger, signals):
    '''Generate a plot of relative user activity signal.'''
    legendstuff = []
    for event, vals in signals.iteritems():
        t = np.arange(−interval, interval, bin_size)
        t = map(lambda x: x*1.0/3600, t)
        legendstuff.append(event)
        s = list(vals)
        plt.plot(t, s)
    plt.legend(legendstuff, loc='upper left')
    plt.xlabel('Time (h)')
    plt.ylabel('Relative signal')
    plt.title("Trigger: " + trigger)
    plt.grid(True)
```

```
    saveas = make_filename(trigger, interval, bin_size, ext = 'pdf')
    plt.savefig(saveas)
    print "Saved to "+saveas
    if display:
        plt.show()

if __name__ == '__main__':
    # Read in data
    filenames = glob.glob('bluetooth-int43200-bin600.json')
    for filename in filenames:
        data = read_data(filename)
        for trigger, signals in data.iteritems():
            make_plot(trigger, signals)
        #
```

A.1.3 Preprocessing

```python
# -*- coding: utf-8 -*-
from __future__ import division

import os
import glob
import numpy as np
from ast import literal_eval as LE
from social_fabric.phonetools import readncheck, user2profile
from social_fabric.secrets import pushme
from social_fabric.smallestenclosingcircle import make_circle
from social_fabric import lloyds
from collections import Counter
import math
from datetime import datetime, timedelta
import pytz
import json
from statsmodels.tsa import ar_model

#==============================================================================
# Parameters
#==============================================================================

#Grap all the files we need to read
#userfile_path = "c:\\ userfiles \\" #@Windows
userfile_path = "/lscr_paper/amoellga/Data/Telefon/userfiles/" #linux

#Filename to save output to.
output_filename = 'data_single_thread_numedgps.json'

#Number of users to analyse. Use None or 0 to include everyone.
max_users = None

#Specific user codes to analyze for debugging purposes. Empty means include all
exclusive_users = []

#Conversion factors from degrees to meters (accurate around Copenhagen)
longitude2meters = 111319
latitude2meters = 110946

#In meters. These improve convergence of the stochastic SEC algorithm.
x_offset = 1389425.2238223257
y_offset = 6181209.0059678229

required = ['bluetooth_log.txt', 'call_log.txt', 'facebook_log.txt',
            'gps_log.txt', 'sms_log.txt']

user_folders = [f for f in glob.glob(userfile_path+"*")
                if all(os.path.isfile(f+"/"+stuff) for stuff in required)]
users = [folder.split(userfile_path)[1] for folder in user_folders]

if not users:
    raise IOError('Found no users. Check userfile path.')

if exclusive_users:
    users = list(set(exclusive_users).intersection(users))

if max_users:
```

```
    users = users[:max_users]

# Number of wallclock hours pr bin when fitting autoregressive series
hours_pr_ar_bin = 6
assert 24%hours_pr_ar_bin == 0

# Number of hours pr bin when compute daily rythm entropy
hours_pr_daily_rythm_bin = 1
assert 24%hours_pr_daily_rythm_bin == 0

# Time zone information
cph_tz = pytz.timezone('Europe/Copenhagen')

#Which data kinds to include
include_calls = True
include_ar = True
include_gps = True
include_network = False #Allow geolocation from network data
include_bluetooth = True
include_facebook = True

#Whether to include not-a-number values in final output
allow_nan = True

#Threshold values to discard users with insufficient data
minimum_number_of_texts = 10
minimum_number_of_calls = 5
minimum_number_of_gps_points = 100
minimum_number_of_facebook_friends = 1

#N_pings required to be considered social
bluetooth_social_threshold = 2

#Whether to output plots of cluster analysis
plot_clusters = False

#===========================================================================
# Define helper methods
#===========================================================================

def get_distance(p, q):
    return math.sqrt(sum([(p[i]-q[i])**2 for i in xrange(len(p))]))

def is_sorted(l):
    return all([l[i+1] >= l[i] for i in xrange(len(l)-1)])

def get_entropy(event_list):
    '''Takes a list of contacts from in/ or outgoing call/text events and
     computes its entropy. event_list must be simply a list of user codes
     corresponding to events.'''
    n = len(event_list)
    counts = Counter(event_list)
    ent = sum([-v/n * math.log(v/n, 2) for v in counts.values()])
    return ent

def next_time(dt, deltahours):
    '''Accepts a datetime object and returns the next datetime object at which
     the 'hour' count modulo deltahours is zero.
     For example, deltahours = 6 gives the next time clock time is
     0, 6, 12 or 18.
     Sounds simple but is pretty annoying due to daylight saving time and so on,
     so take care not to mess with this.'''
```

```python
    base = datetime(dt.year, dt.month, dt.day, dt.hour)
    interval = timedelta(hours = deltahours - dt.hour%deltahours)
    naive_guess = base + interval
    return cph_tz.localize(naive_guess)

def next_midnight(dt):
    '''Takes a datetime object and returns dt object of following midnight'''
    base = datetime(dt.year, dt.month, dt.day)
    naive = base + timedelta(days = 1)
    return cph_tz.localize(naive)

def epoch2dt(timestamp):
    '''Converts unix timestamp into a pytz timezone-aware datetimeobject.'''
    utc_time = datetime.utcfromtimestamp(timestamp)
    smart_time = cph_tz.fromutc(utc_time)
    return smart_time

def sort_dicts_by_key(dictlist, key):
    '''Takes a list of dicts and returns the same list sorted by its
     key-entries.'''
    decorated = [(d[key], d) for d in dictlist]
    decorated.sort()
    return [d for (k, d) in decorated]

def get_autocovar_coefficient(X, lag):
    '''Returns the autocovariance coefficient for the input series at the
     input lag.'''
    mu = np.mean(X)
    temp = sum((X[i] - mu)*(X[i+lag]-mu) for i in xrange(len(X)-lag))
    return temp*1.0/len(X)

def get_autocorrelation_coefficients(series, lags):
    '''Determines the autocorrelation coefficients of input series at
     each of the input lags. Uses .r_k = c_k/c_0.
     Accepts a list of lags or an int in which case it returns lags up to
     and including the input.'''
    if isinstance(lags, int):
        lags = range(lags+1)
    c0 = get_autocovar_coefficient(series, 0)
    inv = 1.0/c0
    return [inv*get_autocovar_coefficient(series, lag) for lag in lags]

def make_time_series(dts, hours_pr_ar_bin):
    '''Takes a sorted list of datetime objects and converts to a time series
     where each entry denotes the number of events in the corresponding bin.'''
    first_time = next_time(dts[0], hours_pr_ar_bin)
    for i in xrange(len(dts)):
        if dts[i] >= first_time:
            dts = dts[i:]
            break
        #
    last_time = next_time(first_time, hours_pr_ar_bin)
    time_series = []

    summer = 0
    for dt in dts:
        while not first_time <= dt < last_time:
            time_series.append(summer)
            summer = 0
            first_time = last_time
            last_time = next_time(last_time, hours_pr_ar_bin)
        summer += 1
```

```
    return time_series


def timestamps2daily_entropy(timestamps, hours_pr_bin):
        '''Constructs a histogram of hour-values of the imput timestamps
         and computes its entropy.'''
        if not 24%hours_pr_bin == 0:
            raise ValueError("24 must be divisible by hours_pr_bin.")
        bins = {}
        for timestamp in timestamps:
            hour = epoch2dt(timestamp).hour
            _bin = int(hour/hours_pr_bin)
            try:
                bins[_bin] += 1
            except KeyError:
                bins[_bin] = 1
            #
        total = sum(bins.values())
        entropy = sum([-v/total*math.log(v/total, 2) for v in bins.values()])
        return entropy

#Make sure we have a blank file to write to.
open(output_filename, 'w').close()
assert os.stat(output_filename).st_size == 0 #Check that it worked.

user_counter = 0

for user in users:
    user_counter += 1
    msg = "Processing user %s of %s: %s" % (user_counter, len(users), user)
    print msg
    if user_counter % 100 == 0:
        pushme(msg)

    #Dict to hold all data extracted on current user
    data = {}

    #Try to load user psych profile- Discard user if they're not in file
    try:
        profile = user2profile[user]
    except KeyError:
        print "No psychological data on user."
        continue # No questionaire data. Shouldn't happen.

    #Read in calls/texts
    calls = readncheck(userfile_path+user+"/call_log.txt")

    if len(calls) < minimum_number_of_calls:
        print "too few calls"
        continue
    texts = readncheck(userfile_path+user+"/sms_log.txt")
    if len(texts) < minimum_number_of_texts:
        print "too few texts"
        continue

    #Extract list of times for calls, texts and combination
    call_times = sorted([d['timestamp'] for d in calls])
    text_times = sorted([d['timestamp'] for d in texts])
    call_text_times = sorted(call_times + text_times)

    #Get calls from the first three months
    tmin = call_times[0]
```

```
    tmax = call_times[0] + 60*60*24*30*3
    early_calls = [c['number'] for c in calls if tmin<=c['timestamp']<=tmax]

    #Repeat for texts
    tmin = text_times[0]
    tmax = text_times[0] + 60*60*24*30*3
    early_texts = [t['body'] for t in texts if tmin<=t['timestamp']<=tmax]

    #Get number of unique contacts for first 3 months and append to data.
    uniques = len(set(early_calls + early_texts))
    data['n_contacts_first_three_months'] = uniques

    #Compute daily entropy for calls and texts
    data['call_daily_entropy'] = timestamps2daily_entropy(call_times,
                                            hours_pr_daily_rythm_bin)
    data['text_daily_entropy'] = timestamps2daily_entropy(text_times,
                                            hours_pr_daily_rythm_bin)

    #Compute median and std for call durations
    call_durations = [c['duration'] for c in calls if not c['duration'] == 0]
    data['call_duration_med'] = np.median(call_durations)
    data['call_duration_std'] = np.std(call_durations)

#==============================================================================
# Crunch time-series info
#==============================================================================
    if include_ar:
        #Grap a sorted list of only times of events caused by user
        outgoing_stuff = sorted([d['timestamp'] for d in calls
                            if d['type'] == 2] + [d['timestamp']
                            for d in texts if d['type'] == 2])

        n_params = int(24*7/hours_pr_ar_bin + 1) #1 week plus 1 extra bin

        #-Fit time series and extract parameters
        try:
            #Convert into timezone aware datetime objects
            dts = [epoch2dt(timestamp) for timestamp in outgoing_stuff]

            time_series = make_time_series(dts, hours_pr_ar_bin)

            model = ar_model.AR(time_series)
            result = model.fit(n_params)
            #Grab parameters from fitted model
            params = result.params[1:]
            while len(params) < n_params:
                params.append(float('nan'))
        except:
            if not allow_nan:
                continue
            else:
                params = [float('nan') for _ in xrange(n_params)]

        #Append AR-coefficients to user data
        count = 0
        for par in params:
            count += 1
            name = "outgoing_activity_AR_coeff_"+str(count)
            data[name] = par

        #Get autocorrelation coefficients as well
        try:
```

```
            accs = get_autocorrelation_coefficients(time_series, n_params)[:1]
        except:
            if not allow_nan:
                continue
            accs = [float('nan') for _ in xrange(n_params + 1)]

        #Append autocorrelation coefficients to user data
        for i in xrange(len(accs)):
            name = "outgoing_activity_acc_"+str(i)
            data[name] = accs[i]

        #Repeat with incoming signals. Might be interesting.
        incoming_stuff = sorted([d['timestamp'] for d in calls
                                if d['type'] == 1] + [d['timestamp']
                                for d in texts if d['type'] == 1])

        try:
            # Convert into timezone aware datetime objects
            dts = [epoch2dt(timestamp) for timestamp in incoming_stuff]

            time_series = make_time_series(dts, hours_pr_ar_bin)
            model = ar_model.AR(time_series)
            result = model.fit(n_params)
            params = result.params[1:]
            while len(params) < n_params:
                params.append(float('nan'))
        except:
            if not allow_nan:
                continue
            else:
                params = [float('nan') for _ in xrange(n_params)]

        # Name each of them and append to user data
        count = 0
        for par in params:
            count += 1
            name = "incoming_activity_AR_coeff_"+str(count)
            data[name] = par

        #Get autocorrelation coefficients as well
        try:
            accs = get_autocorrelation_coefficients(time_series, n_params)[:1]
        except:
            if not allow_nan:
                continue
            accs = [float('nan') for _ in xrange(n_params + 1)]

        for i in xrange(len(accs)):
            name = "incoming_activity_acc_"+str(i)
            data[name] = accs[i]


#===========================================================================
#       Crunch call/text info
#===========================================================================

    if include_calls:
        #Add values to temporary data map.
        d = {'call' : call_times, 'text' : text_times, 'ct' : call_text_times}
        for label, times in d.iteritems():
            timegaps = [times[i+1] - times[i] for i in xrange(len(times)-1)]
            timegaps = filter(lambda x: x < 259200, timegaps)  #3 days, tops
```

```
        data[label+'_iet_med'] = np.median(timegaps)
        data[label+'_iet_std'] = np.std(timegaps)

    #Generate lists of the contact for each text/call event
    call_numbers = [call['number'] for call in calls]
    text_numbers = [text['address'] for text in texts]
    ct_numbers = call_numbers + text_numbers

    #Compute entropy and add to data
    data['call_entropy'] = get_entropy(call_numbers)
    data['text_entropy'] = get_entropy(text_numbers)
    data['ct_entropy'] = get_entropy(ct_numbers)

    #Compute contact list info
    call_contacts = Counter([c['number'] for c in calls
                        if c['type'] == 2]).keys()
    text_contacts = Counter([t['address'] for t in texts
                        if t['type'] == 2]).keys()
    #Grap number of contacts
    n_call_contacts = len(call_contacts)
    n_text_contacts = len(text_contacts)
    n_ct_contacts = len(set(call_contacts).union(set(text_contacts)))

    #Add to data map
    data['n_call_contacts'] = n_call_contacts
    data['n_text_contacts'] = n_text_contacts
    data['n_ct_contacts'] = n_ct_contacts

    #Compute and add contact/interaction ratio (cir)
    data['call_cir'] = n_call_contacts/len(calls)
    data['text_cir'] = n_text_contacts/len(texts)
    data['ct_cir'] = n_ct_contacts/(len(calls) + len(texts))

    #Add data on number of interactions
    data['n_calls'] = len(calls)
    data['n_texts'] = len(texts)
    data['n_ct'] = len(calls + texts)

    #Determine percentage of calls/texts that were initiated by user.
    initiated_calls = len([c for c in calls if c['type'] == 2])
    data['call_percent_initiated'] = initiated_calls/len(calls)
    initiated_texts = len([t for t in texts if t['type'] == 2])
    data['call_percent_initiated'] = initiated_texts/len(texts)

    #Determine call response rate.
    with open(userfile_path+user+"/call_log.txt", 'r') as f:
        all_calls = [LE(line) for line in f.readlines()]
    #Make sure the call data is sorted
    if not is_sorted([c['timestamp'] for c in all_calls]):
        all_calls = sort_dicts_by_key(all_calls, 'timestamp')

    '''Check for unanswered called that are replied to within an hour.
     This is performed in the following fashion: iterate through all the
     calls. If a call is unanswered, add it to "holding" list. If a call
     from holding matches the current call, it counts as a reply.
     If the time of the current call is more than hour after a held call,
     it is discarded'''
    missed = 0
    replied = 0
    holding = []
    for call in all_calls:
        if call['type']==3 or call['type']==1 and call['duration']==0:
```

```
                holding.append(call)
                missed += 1
            else:
                for held_call in holding:
                    #Drop calls that have been held for too long
                    if call['timestamp'] − held_call['timestamp'] > 3600:
                        holding.remove(held_call)
                    #Check if given call is a resonse
                    elif (call['type'] == 2
                          and call['number'] == held_call['number']):
                        holding.remove(held_call)
                        replied += 1
                    #
                #
            #
        data['call_response_rate'] = replied/(missed+replied) if replied else 0

        #Determine text response rate
        missed = 0
        replied = 0
        holding = []
        response_times = []
        if not is_sorted([t['timestamp'] for t in texts]):
            texts = sort_dicts_by_key(texts, 'timestamp')

        for text in texts:
            #Make sure incoming text is not from a user already held
            if text['type'] == 1:
                if not holding or all([text['address'] !=
                                       t['address'] for t in holding]):
                    #It's good − append it
                    holding.append(text)
                    missed += 1
                #
            else:
                for held_text in holding:
                    if text['timestamp'] − held_text['timestamp'] > 3600:
                        holding.remove(held_text)
                    #Check if text counts as reply
                    elif (text['type'] == 2
                    and text['address'] == held_text['address']):
                        holding.remove(held_text)
                        replied += 1
                        dt = text['timestamp'] − held_text['timestamp']
                        response_times.append(dt)
                    #
                #
            #
        data['text_response_rate'] = replied/(missed+replied) if replied else 0
        data['text_latency'] = np.median(response_times)

        #Check percentage of calls taken place in during the night
        count = 0
        for call in calls:
            hour = epoch2dt(call['timestamp']).hour
            if not (8 <= hour <22):
                count += 1
            #
        data['call_night_activity'] = count/len(calls)

        #Compute % of calls/texts outgoing from user. This works because true=1
        data['call_outgoing'] = sum([c['type'] == 2 for c in calls])/len(calls)
```

```
        data['text_outgoing'] = sum([t['type'] == 2 for t in texts])/len(texts)

#==============================================================================
# Crunch location data
#==============================================================================

    if include_gps:
        with open(userfile_path+user+"/gps_log.txt", 'r') as f:
            raw = [LE(line) for line in f.readlines()]

        if not is_sorted([l['timestamp'] for l in raw]):
            raw = sort_dicts_by_key(raw, 'timestamp')

        #We only want measurements taken at least 500s apart.
        prev = 0
        gps_data = []
        allowed_providers = ['gps', 'network'] if include_network else ['gps']
        for line in raw:
            now = line['timestamp']
            if line['provider'] in allowed_providers and now - prev >= 500:
                #Convert coordinates to km and note it down
                x = (longitude2meters*line['lon'] - x_offset)*0.001
                y = (latitude2meters*line['lat'] - y_offset)*0.001
                gps_data.append({'point' : (x,y), 'timestamp' : now,
                                 'smarttime' : epoch2dt(now)})
                prev = now
            #
        #ignore user if there aren't enough data
        if not len(gps_data) >= minimum_number_of_gps_points:
            continue

        # We want to investigate each day saparately so start at midnight.
        first_midnight = next_midnight(gps_data[0]['smarttime'])
        for i in xrange(len(gps_data)):
            if gps_data[i]['smarttime'] >= first_midnight:
                gps_data = gps_data[i:]
                break


        #Generate list of radii of smallest enclosing circle , SEC, for each day
        current_points = []
        prev = gps_data[0]['timestamp']
        radii = []

        distances = []
        early_day = gps_data[0]['smarttime']
        late_day = next_midnight(early_day)
        for datum in gps_data:
            now = datum['smarttime']
            while not early_day <= now < late_day:
                if len(current_points) > 2:
                    crds = [p['point'] for p in current_points]
                    circle = make_circle(crds)
                    r = circle[2] if circle else 0
                    if circle and r > 0:
                        if r <= 500:
                            radii.append(r)
                    distances.append(sum([get_distance(crds[i], crds[i+1])
                                     for i in xrange(len(crds)-1)]))
                # Reset counters and update bins
                current_points = []
                early_day = late_day
```

```
            late_day = next_midnight(late_day)
        current_points.append(datum)


    data['radius_of_gyration_med'] = np.median(radii)
    data['radius_of_gyration_std'] = np.std(radii)
    data['travel_med'] = np.median(distances)
    data['travel_std'] = np.std(distances)

    #Run Lloyd's algorithm to identify clusters
    #Determine which points are stationary - less movement than 100m
    stationary_data = []
    try:
        for i in xrange(1,len(gps_data)-1):
            a,b,c = tuple([gps_data[ind]['point'] for ind in
                           [i-1, i, i+1]])
            if (get_distance(a, b) < 0.1 and get_distance(b, c) < 0.1):
                stationary_data.append(gps_data[i])
            #
        initial_clusters = 50
        threshold_percent = 0.05
        points = [elem['point'] for elem in stationary_data]

        minimum_points = int(threshold_percent*len(points))


        clusters_scatter = lloyds.lloyds(points, initial_clusters, runs=3,
                                         init='scatter')

        clusters_sample = lloyds.lloyds(points, initial_clusters, runs=3,
                                        init='sample')

        #Determine most succesful method
        locs = lambda c: [p for p in c.values() if len(p)>=minimum_points]
        if len(locs(clusters_scatter)) >= len(locs(clusters_sample)):
            method = 'scatter'
            best = clusters_scatter
        else:
            method = 'sample'
            best = clusters_sample

        # Number of places which survive cutoff (true = 1, so just sum)
        n_places = sum([len(pl) >= minimum_points for pl in best.values()])

        #Output plots of clustering
        if plot_clusters:
            if not os.path.isdir('pics'):
                os.mkdir('pics')
            for ext in ['.pdf', '.png']:
                filename = 'pics/'+user+"_"+method+ext
                lloyds.draw_clusters(clusters = best,
                                     threshold = minimum_points,
                                     show = False,
                                     filename = filename)
    except:
        if not allow_nan:
            continue
        n_places = float('nan')
    data['n_places'] = n_places

    #Compute location entropy and add to data
    try:
```

```
        n_points = sum(len(location) for location in best.values())
        data['location_entropy'] = sum([-len(p)/n_points*math.log(len(p)/n_points)
                                       for p in best.values()])
    except ValueError:
        data['location_entropy'] = float('nan')

    '''Guess where people live. Probably where they spend weeknights...
     It's important to avoid selection bias here (people probably turn off
     their phone when sleeping at home but not while partying at DTU, which
     means fewer data points at their actual home).
     This is rectified by excluding points that aren't logged monday to
     thursday and only recording one 'late' or 'early' data point pr date.
     These points are labelled 'weird' and are used to determine the user's
     home.'''
    try:
        weirdpoints = []
        latedays = []
        earlydays = []
        for datum in stationary_data:
            now = datum['smarttime']
            #ignore weekends
            if now.weekday() > 3:
                continue
            thisdate = (now.year, now.month, now.day)
            if now.hour >= 20 and not thisdate in latedays:
                weirdpoints.append(datum['point'])
                latedays.append(thisdate)
            elif now.hour <= 7 and not thisdate in earlydays:
                weirdpoints.append(datum['point'])
                earlydays.append(thisdate)

        best_score = 0
        home = None
        for key, val in best.iteritems():
            score = len(set(val).intersection(weirdpoints))
            if score > best_score:
                home = key
                best_score = score
            #
        # Estimate how much user spends at home
        ordered_gps = sort_dicts_by_key(gps_data, 'timestamp')
        is_home = lambda p: get_distance(home, p) <= 0.200
        time_home = 0
        time_away = 0
        for i in xrange(len(ordered_gps)-1):
            a = ordered_gps[i]
            b = ordered_gps[i+1]
            dt = b['timestamp'] - a['timestamp']
            if dt > 7200:
                continue
            elif is_home(a['point']) and is_home(b['point']):
                time_home += dt
            elif (not is_home(a['point'])) and (not is_home(b['point'])):
                time_away += dt
            #
        data['home_away_time_ratio'] = time_home/time_away
    except:
        if not allow_nan:
            continue
        data['home_away_time_ratio'] = float('nan')
```

```
#==========================================================================
# Facebook data
#==========================================================================

    if include_facebook:
        with open(userfile_path+user+'/facebook_log.txt', 'r') as f:
            n = len(f.readlines())
            if n < minimum_number_of_facebook_friends:
                continue
            data['number_of_facebook_friends'] = n

#==========================================================================
# Bluetooth data
#==========================================================================

    if include_bluetooth:
        with open(userfile_path+user+"/bluetooth_log.txt", 'r') as f:
            raw = [LE(line) for line in f.readlines()]
        #Make sure data is sorted chronologically
        if not is_sorted([entry['timestamp'] for entry in raw]):
            raw = sort_dicts_by_key(raw, 'timestamp')
        #List of times when user was social
        social_times = []
        total_social_time = 0
        total_time = 0
        #maps from each other user encountered to time spend with said user
        friend2time_spent = {}
        #Temporary variables
        current_time = 0
        previous_time = 0
        current_users = []
        previous_users = []
        for signal in raw:
            new_time = signal['timestamp']
            #Check if line represents a new signal and if so, update values
            if new_time != current_time:
                dt = new_time - current_time
                #Determine number of pings
                overlap = set(previous_users).intersection(set(current_users))
                if len(overlap) >= bluetooth_social_threshold:
                    social_times.append(previous_time)
                    if dt <= 7200:
                        total_social_time += dt
                        total_time += dt
                        for friend in overlap:
                            try:
                                friend2time_spent[friend] += dt
                            except KeyError:
                                friend2time_spent[friend] = dt
                    #
                elif dt <= 7200:
                    total_time += dt
                #Update variables
                previous_time = current_time
                previous_users = current_users
                current_users = []
                current_time = new_time
            new_user = signal['name']
            if new_user=='-1' or not new_user:
                continue
            else:
                current_users.append(new_user)
```

```
            #
        # Add fraction of time spent social to output
        data['fraction_social_time'] = total_social_time/total_time
        # Compute social entropy
        normfac = 1.0/sum(friend2time_spent.values())
        ent = sum(-t*normfac*math.log(t*normfac)
                  for t in friend2time_spent.values())
        data['social_entropy'] = ent

        data['bluetooth_daily_entropy']=timestamps2daily_entropy(social_times,
                                                    hours_pr_daily_rythm_bin)

        #Ensure time span is suficcient to make a time series
        if not (social_times[-1] - social_times[0] > 24*3600*7
                +1+3600*hours_pr_ar_bin):
            continue

        #Fit AR-series and append parameters to output
        try:
            dts = [epoch2dt(timestamp) for timestamp in social_times]
            time_series = make_time_series(dts, hours_pr_ar_bin)
            model = ar_model.AR(time_series)
            n_params = int(24*7/hours_pr_ar_bin + 1) #1 week plus 1 extra bin
            result = model.fit(maxlag = None, ic = None)
            params = result.params#[1:]
            while len(params) < n_params:
                params.append(float('nan'))
        except:
            if not allow_nan:
                continue
            params = [float('nan') for _ in xrange(n_params)]

        count = 0
        for par in params:
            count += 1
            name = "bluetooth_activity_AR_coeff_"+str(count)
            data[name] = par

        # Compute autocorrelation coeffs and append to output
        try:
            accs = get_autocorrelation_coefficients(time_series, n_params)[:1]
        except:
            if not allow_nan:
                continue
            accs = [float('nan') for _ in xrange(n_params + 1)]
        for i in xrange(len(accs)):
            name = "bluetooth_activity_acc_"+str(i)
            data[name] = accs[i]

#==============================================================================
# Wrap up user
#==============================================================================

    # Double check thata doesn't containing nan values
    if any(np.isnan(value) for value in data.values()) and not allow_nan:
        continue #Discard user due to insufficient data

    #Collect results
    final = {'user' : user, 'data' : data, 'profile' : profile}

    with open(output_filename, 'a') as f:
        json.dump(final, f)
```

```
        f.write("\n")

#Done.
pushme("Data extraction done.")
```

A.1.4 Social Fabric Code

```
# -*- coding: utf-8 -*-
"""This module aims to allow sharing of some common methods and settings
when testing and tweaking various machine learning schemes.
Always import settings and the like from here!"""

from __future__ import division
import abc
from collections import Counter
import itertools
import json
import math
import matplotlib.colors as mcolors
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import multiprocessing
import numpy as np
import os
import random
from scipy.sparse import dok_matrix
from sklearn import svm
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.cross_validation import (cross_val_score, LeaveOneOut, KFold,
                                      StratifiedKFold)
import sys
import traceback

BASE = os.path.dirname(__file__)

oldhat = (35/256,39/256,135/256)
nude = (203/256,150/256,93/256)
wine = (110/256,14/256,14/256)
moerkeroed = (156/256,30/256,36/256)


#Empirically determined optimal hyperparameters for SVRs and RFs
with open(BASE+'/svr_parameters.json', 'r') as f:
    svr_parameters = json.load(f)

wrbf_parameters = svr_parameters #updater!!!

with open(BASE+'/rf_parameters.json', 'r') as f:
    rf_parameters = json.load(f)

def _make_colormap(seq):
    """Return a LinearSegmentedColormap
     seq: a sequence of floats and RGB-tuples. The floats should be increasing
     and in the interval (0,1).
     """
    seq = [(None,) * 3, 0.0] + list(seq) + [1.0, (None,) * 3]
    cdict = {'red': [], 'green': [], 'blue': []}
```

```python
    for i, item in enumerate(seq):
        if isinstance(item, float):
            r1, g1, b1 = seq[i - 1]
            r2, g2, b2 = seq[i + 1]
            cdict['red'].append([item, r1, r2])
            cdict['green'].append([item, g1, g2])
            cdict['blue'].append([item, b1, b2])
    return mcolors.LinearSegmentedColormap('CustomMap', cdict)

color_map = _make_colormap([oldhat, moerkeroed, 0.33, moerkeroed, nude, 0.67, nude])

big_five = ['openness', 'conscientiousness', 'extraversion', 'agreeableness',
            'neuroticism']

#_default_features = ["n_texts",
#                     " ct_iet_std ",
#                     " call_cir ",
#                     "call_entropy",
#                     " text_cir ",
#                     "n_calls ",
#                     "text_latency ",
#                     "call_outgoing",
#                     "fraction_social_time ",
#                     "text_outgoing",
#                     " call_iet_std ",
#                     "n_text_contacts",
#                     " call_night_activity ",
#                     "call_iet_med",
#                     "outgoing_activity_AR_coeff_2",
#                     "text_entropy",
#                     " ct_cir ",
#                     "text_response_rate",
#                     "n_ct_contacts",
#                     "social_entropy",
#                     "n_call_contacts ",
#                     "n_ct",
#                     " text_iet_std ",
#                     "ct_iet_med",
#                     "ct_entropy",
#                     "text_iet_med",
#                     "call_response_rate",
#                     "number_of_facebook_friends"]

_default_features = ['call_iet_med', 'text_iet_med', 'social_entropy',
'call_entropy', 'travel_med', 'n_places', 'text_latency',
'call_night_activity']

def split_ntiles(values, n):
    '''Determines the values that separate the imput list into n equal parts.
     this is a generalization of the notion of median (in the case n = 2) or
     quartiles (n=4).
     Usage: ntiles([5,6,7], 2) gives [6] for instance.'''
    result = []
    for i in xrange(1,n):
        percentile = 100/n * i
        result.append(np.percentile(values, percentile,
                                    interpolation='linear'))
    return result

def determine_ntile(value, ntiles):
    '''Determines which n-tile the input value belongs to.
     Usage: determine_ntile([7,9,13], 10) gives 2 (third quartile).
```

```
    This uses zero indexing so data split into e.g. quartiles will give results
    like 0,1,2,3 - NOT 1,2,3,4.'''
    #Check if value is outside either extreme, meaning n−tile 1 or n.
    if value >= ntiles[−1]:
        return len(ntiles) #Remember the length is n−1
    elif value < ntiles[0]:
        return 0 #Values was in the first n−tile
    # Define possible region and search for where value is between two elements
    left = 0
    right = len(ntiles)−2
    #Keep checking th middle of the region and updating region
    ind = (right + left)//2
    while not ntiles[ind] <= value < ntiles[ind + 1]:
        #Check if lower bound tile is on the left
        if value < ntiles[ind]:
            right = ind − 1
        else:
            left = ind + 1
        ind = (right + left)//2
    # Being between ntiles 0 and 1, means second n−tile and so on.
    return ind + 1

def assign_labels(Y, n):
    '''Accepts a list and an int n and returns a list of discrete labels
    corresponding to the ntile each original y-value was in.'''
    ntiles = split_ntiles(Y, n)
    labels = [determine_ntile(y, ntiles) for y in Y]
    return labels

def normalize_data(list_):
    '''Normalizes input data to the range [-1, 1]'''
    lo, hi = min(list_), max(list_)
    if lo == hi:
        z = len(list_)*[0]
        return z
    else:
        return [2*(val−lo)/(hi − lo) − 1 for val in list_]


def read_data(filename, trait, n_classes = None, normalize = True,
              features='default', interpolate = True):
    '''This reads in a preprocessed datafile, splits psych profile data into
    n classes if specified, filters desired psychological traits and
    features and returns as a tuple (X,Y, indexdict), which can be fed to a'
    number of off-the-shelf ML schemes.
    If trait=='Sex', female and male are converted to 0 and 1, respectively.
    indexdict maps each element of the feature vectors to their label, as in
    {42 : 'distance_travelled_pr_day'} etc.

    Args:
      filename : str
        Name of the file containing the data.
      trait : str
        The psychological trait to extract data on.
      n_classes : int
        Number of classes to split data into. Default is None,
        i.e. just keep the decimal values. Ignored if trait == 'Sex', as data
        only has two discreet values.
      normalize : bool
        Whether to hard normalize data to [-1, 1].
      features : str/list
        Which features to read in. Can also be 'all'
```

```
        or 'default', meaning the ones I've pragmatically found to be
        reasonable.
      interpolate : bool:
        Whether to replace NaN's with the median value of
        the feature in question.'''
    if trait == 'sex':
        n_classes = None
    #Read in the raw data
    with open(filename, 'r') as f:
        raw = [json.loads(line) for line in f.readlines()]
    #Get list of features to be included - everything if nothing's specified
    included_features = []
    if features == 'default':
        included_features = _default_features
    elif features == 'all':
        included_features = raw[0]['data'].keys() if features=='all' else features
    else:
        included_features = features

    #Remove any features that only have NaN values.
    for i in xrange(len(included_features)-1,-1,-1):
        feat = included_features[i]
        if all(math.isnan(line['data'][feat]) for line in raw):
            del included_features[i]


    # ----- Handle feature vectors -----
    # Dict mapping indices to features
    indexdict = {ind : feat for ind, feat in enumerate(included_features)}
    # N_users x N_features array to hold data
    rows = len(raw)
    cols = len(included_features)
    X = np.ndarray(shape = (rows, cols))
    for i in xrange(rows):
        line = raw[i]
        #Construct data matrix
        for j, feat in indexdict.iteritems():
            val = line['data'][feat]
            X[i, j] = val
        #
    #Replace NaNs with median values
    if interpolate:
        for j in xrange(cols):
            #Get median of feature j
            med = np.median([v for v in X[:, j] if not math.isnan(v)])
            if math.isnan(med):
                raise ValueError('''Feature %s contains only NaN's and should
                                 have been removed.''' % indexdict[j])
            for i in xrange(rows):
                if math.isnan(X[i,j]):
                    X[i,j] = med

    if normalize:
        for j in xrange(cols):
            col = X[:, j]
            X[:, j] = normalize_data(col)

    # ----- Handle class info -----
    trait_values = []
    for line in raw:
        #Add value of psychological trait
        psych_trait = line['profile'][trait]
```

```
        if trait == 'Sex':
            if psych_trait == 'Female':
                psych_trait = 0
            elif psych_trait == 'Male':
                psych_trait = 1
            else:
                raise ValueError('My code is binary gender normative, sorry.')
        trait_values.append(psych_trait)
    Y = []
    if n_classes == None:
        Y = trait_values
    else:
        ntiles = split_ntiles(trait_values, n_classes)
        Y = [determine_ntile(tr, ntiles) for tr in trait_values]

    return (X, Y, indexdict)

def get_strong_indices(X, Y, threshold):
    '''Returns the indices of the features vectors X whose linear correlation
     with Y is above the imput threshold.'''
    correlations = get_correlations(X, Y)
    indices = [i for i in xrange(len(correlations))
               if correlations[i] >= threshold]
    return indices

def reduce_data(X, indices):
    '''Return feature vectors including only the feature numbers in the input
     list of indices.'''
    reduced_X = np.array([[x[i] for i in indices] for x in X])
    return reduced_X

def plot_stuff(input_filename, output_filename=None, color=moerkeroed):
    with open(input_filename, 'r') as f:
        d = json.load(f)
    x = d['x']
    y = d['y']
    yerr = d['mean_stds']
    plt.plot(x,y, color=color, linestyle='dashed',
             marker='o')
    plt.errorbar(x, y, yerr=yerr, linestyle="None", marker="None",
                 color=color)
    if output_filename:
        plt.savefig(output_filename)

def get_TPRs_and_FPRs(X, Y, forest = None, verbose = False):
    '''Accepts a list of feature vectors and a list of labels and returns a
     tuple of true positive and false positive rates (TPRs and FPRs,
     respectively) for various confidence thresholds.'''
    kf = LeaveOneOut(n=len(Y))

    results = []
    thresholds = []

    counter = 0
    for train, test in kf:
        counter += 1
        if counter % 10 == 0 and verbose:
            print "Testing on user %s of %s..." % (counter, len(Y))

        result = {}
        train_data = [X[i] for i in train]
        train_labels = [Y[i] for i in train]
```

```
        test_data = [X[i] for i in test]
        test_labels = [Y[i] for i in test]

        if not forest:
            forest = RandomForestClassifier(
                                            n_estimators = 1000,
                                            n_jobs=-1,
                                            criterion='entropy')

        forest.fit(train_data, train_labels)
        result['prediction'] = forest.predict(test_data)[0]
        result['true'] = test_labels[0]
        confidences = forest.predict_proba(test_data)[0]
        result['confidences'] = confidences
        thresholds.append(max(confidences))

        results.append(result)

    #ROC curve stuff - false and true positive rates
    TPRs = []
    FPRs = []

    unique_thresholds = sorted(list(set(thresholds)), reverse=True)

    for threshold in unique_thresholds:
        tn = 0
        fn = 0
        tp = 0
        fp = 0
        for result in results:
            temp = result['prediction']
            if temp == 1 and result['confidences'][1] >= threshold:
                pred = 1
            else:
                pred = 0
            if pred == 1:
                if result['true'] == 1:
                    tp += 1
                else:
                    fp += 1
                #
            elif pred == 0:
                if result['true'] == 0:
                    tn += 1
                else:
                    fn += 1
                #
            #
        TPRs.append(tp/(tp + fn))
        FPRs.append(fp/(fp + tn))
    return (TPRs, FPRs)

def make_roc_curve(TPRs, FPRs, output_filename = None):
    '''Accepts a list of true and false positive rates (TPRs and FPRs,
     respectively) and generates a ROC-curve.'''
    predcol = moerkeroed
    basecol = oldhat
    fillcol = nude

    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)

```

```
    TPRs = [0] + TPRs + [1]
    FPRs = [0] + FPRs + [1]

    area = 0.0
    for i in xrange(len(TPRs)-1):
        dx = FPRs[i+1] - FPRs[i]
        y = 0.5*(TPRs[i] + TPRs[i+1])
        under_curve = dx*y
        baseline = dx*0.5*(FPRs[i] + FPRs[i+1])
        area += under_curve - baseline

    baseline = FPRs
    ax.fill_between(x = FPRs, y1 = TPRs, y2 = baseline, color = fillcol,
                    interpolate = True, alpha=0.8)
    ax.plot(baseline, baseline, color = basecol, linestyle = 'dashed',
            linewidth = 1.0, label = 'Baseline')
    ax.plot(FPRs, TPRs, color=predcol, linewidth = 1,
                     label = 'Prediction')

    plt.xlabel('False positive rate.')
    plt.ylabel('True positive rate.')

    handles, labels = ax.get_legend_handles_labels()
    hest = mpatches.Patch(color=fillcol)

    labels += ['Area = %.3f' % area]
    handles += [hest]
    ax.legend(handles, labels, loc = 'lower right')
#     plt.legend(handles = [tp_line, base])
    if output_filename:
        plt.savefig(output_filename)
    plt.show()

def rank_features(X, Y, forest, indexdict, limit = None):
    '''Ranks the features of a given dataset and classifier.
     indexdict should be a map from indices to feature names like
     {0 : 'average_weigth'} etc.
     if limit is specified, this method returns only the top n ranking features.
     Returns a dict like {'feature name' : (mean importances, std)}.'''
    importances = forest.feature_importances_
    stds=np.std([tree.feature_importances_ for tree in forest.estimators_],
                axis=0)
    indices = np.argsort(importances)[::-1]
    if limit:
        indices = indices[:limit]
    d = {indexdict[i] : (importances[i], stds[i]) for i in indices}
    return d

def check_performance(X, Y, clf, strata = None):
    '''Checks forest performance compared to baseline.'''
    N_samples = len(Y)
    #Set up validation indices
    if not strata:  #Do leave-one-out validation
        skf = KFold(N_samples, n_folds=N_samples, shuffle = False)
    else:  #Do stratified K-fold
        skf = StratifiedKFold(Y, n_folds=strata)

    #Evaluate classifier performance
    scores = []
    for train, test in skf:
        train_data = [X[ind] for ind in train]
        train_labels = [Y[ind] for ind in train]
```

```
        test_data = [X[ind] for ind in test]
        test_labels = [Y[ind] for ind in test]

        #Check performance of input forest
        clf.fit(train_data, train_labels)
        score = clf.score(test_data, test_labels)
        scores.append(score)

    #Compute baseline
    most_common_label = max(Counter(Y).values())
    baseline = float(most_common_label)/N_samples

    #Compare results with prediction baseline
    score_mean = np.mean(scores)/baseline
    score_std = np.std(scores)/baseline

    return (score_mean, score_std)

def check_regressor(X, Y, reg, strata = None):
    '''Checks the performance of a regressor against mean value baseline.'''
    N_samples = len(Y)
    #Set up validation indices
    if not strata:  #Do leave-one-out validation
        skf = KFold(N_samples, n_folds=N_samples,
                                    shuffle = False)
    else:  #Do stratified  K-fold
        skf = StratifiedKFold(Y, n_folds=strata)

    #Evaluate performance
    model_abs_errors = []
    baseline_abs_errors = []
    for train, test in skf:
        train_data = [X[ind] for ind in train]
        train_labels = [Y[ind] for ind in train]
        test_data = [X[ind] for ind in test]
        test_labels = [Y[ind] for ind in test]

        #Check performance of input forest
        reg.fit(train_data, train_labels)
        base = np.mean(train_labels)
        for i in xrange(len(test_data)):
            pred = reg.predict(test_data[i])
            true = test_labels[i]
            model_abs_errors.append(np.abs(pred - true))
            baseline_abs_errors.append(np.abs(base - true))
        #
    return (np.mean(model_abs_errors), np.mean(baseline_abs_errors))

class _RFNN(object):
    __metaclass__ = abc.ABCMeta
    '''Abstract class for random forest nearest neighbor predictors.
     This should never be instantiated.'''

    def __init__(self, forest, n_neighbors):
        self.forest = forest
        self.n_neighbors = n_neighbors
        self.X = None
        self.Y = None

    def fit(self, X, Y):
        '''Fits model to training data.
```

```
         Args
         ------------
         X : List
           List of training feature vectors.

         Y : List
           List of training labels or values to be predicted.'''
        if not len(X) == len(Y):
            raise ValueError("Training input and output lists must have "
                          "same length.")
        if not self.n_neighbors <= len(X):
            raise ValueError("Fewer data points than neighbors.")

        self.forest.fit(X, Y)
        self.X = X
        self.Y = Y

    def _rf_similarity(self, a, b):
        '''Computes a similarity measure for two points using a trained random
         forest classifier.'''
        if self.X == None or self.Y == None:
            raise NotImplementedError("Model has not been fittet to data yet.")

        #Feature vectors must be single precision.
        a = np.array([a], dtype = np.float32)
        b = np.array([b], dtype = np.float32)
        hits = 0
        tries = 0
        for estimator in self.forest.estimators_:
            tries += 1
            tree = estimator.tree_
            # Check whether the points end up on the same leaf for this tree
            if tree.apply(a) == tree.apply(b):
                hits += 1
            #
        return hits/tries

    def find_neighbors(self, point):
        '''Determine the n nearest nieghbors for the given point.
         Returns a list of n tuples like (yval, similarity).
         The tuples are sorted descending by similarity.'''
        if self.X == None or self.Y == None:
            raise NotImplementedError("Model has not been fittet to data yet.")

        #Get list of tuples like (y, similarity) for the n 'nearest' points
        nearest = [(None,float('-infinity')) for _ in xrange(self.n_neighbors)]
        for i in xrange(len(self.X)):
            similarity = self._rf_similarity(self.X[i], point)
            # update top n list if more similar than the furthest neighbor
            if similarity > nearest[-1][1]:
                nearest.append((self.Y[i], similarity))
                nearest.sort(key = lambda x: x[1], reverse = True)
                del nearest[-1]
            #
        return nearest

    #Mandatory methods - must be overridden
    @abc.abstractmethod
    def predict(self, point):
        pass

    @abc.abstractmethod
```

```
    def score(self, X, Y):
        pass

def _reservoir_sampler(start = 1):
    '''Generator of the probabilities need to do reservoir sampling. The point
     it that this can be used to iterate through a list, discarding each element
     for the following element with probability P_n and ending up with a random
     element from the list.'''
    n = start
    while True:
        p = 1/n
        r = random.uniform(0,1)
        if r < p:
            yield True
        else:
            yield False
        n += 1

class RFNNClassifier(_RFNN):
    '''Random Forest Nearest Neighbor Classifier.

     Parameters
     ----
     n_neighbors : int
       Number of neighbors to consider.

     forest : RandomForestClassifier
       The forest which will provide a
       distance measure on which determine nearest neighbors.

     weighting : str
       How to weigh the votes of different neighbors.
       'equal' means each neighbor has an equivalent vote.
       'linear' mean votes are weighed by their similarity to the input point.
     '''
    def predict(self, point):
        '''Predicts the label of a given point.'''
        neighbortuples = self.find_neighbors(point)
        if self.weighting == 'equal':
            #Simple majority vote. Select randomly if it's a tie.
            predictions = [t[0] for t in neighbortuples]
            best = 0
            winner = None
            switch = _reservoir_sampler(start = 2)
            for label, votes in Counter(predictions).iteritems():
                if votes > best:
                    best = votes
                    winner = label
                    switch = _reservoir_sampler(start = 2)
                elif votes == best:
                    if switch.next():
                        winner = label
                    else:
                        pass
                else:
                    pass
                #
            return winner

        #Weigh votes by their similarity to the input point
        elif self.weighting == 'linear':
            #The votes are weighted by their similarity
```

```
            d = {}
            for yval, similarity in neighbortuples:
                try:
                    d[yval] += similarity
                except KeyError:
                    d[yval] = similarity
            best = float('-infinity')
            winner = None
            for k, v in d.iteritems():
                if v > best:
                    best = v
                    winner = k
                else:
                    pass
            return winner

    def score(self, X, Y):
        if not len(X) == len(Y):
            raise ValueError("Training data and labels must have same length.")

        hits = 0
        n = len(X)
        for i in xrange(n):
            pred = self.predict(X[i])
            if pred == Y[i]:
                hits += 1
            #
        return hits/n


    def __init__(self, forest = None, n_neighbors = 3, weighting = 'equal',
                 n_jobs = 1):
        #Make sure we have a forest * classifier *
        if forest == None:
            forest = RandomForestClassifier(n_estimators = 1000,
                                            criterion = 'entropy',
                                            n_jobs = n_jobs)
        if not isinstance(forest, RandomForestClassifier):
            raise TypeError("Forest must be a classifier")

        self.weighting = weighting

        #Call parent constructor
        super(RFNNClassifier, self).__init__(forest, n_neighbors)

class RFNNRegressor(_RFNN):
    '''Random Forest Nearest Neighbor Regressor.

     Parameters
     ----
     n_neighbors : int
       Number of neighbors to consider.

     forest : RandomForestRegressor
       The forest which will provide a
       distance measure on which determine nearest neighbors.

     weighting : str
       How to weigh the votes of different neighbors.
       'equal' means each neighbor has an equivalent weight.
       'linear' mean votes are weighed by their similarity to the input point.
     '''
```

```
    def predict(self, point):
        # lists of the y vaues and similarities of nearest neighbors
        neighbortuples = self.find_neighbors(point)
        yvals, similarities = zip(*neighbortuples)

        # Weigh each neighbor y value equally is that's how we roll
        if self.weighting == 'equal':
            weight = 1.0/len(yvals)
            result = 0.0
            for y in yvals:
                result += y*weight
            return result
            #
        # Otherwise, weigh neighbors by similarity
        elif self.weighting == 'linear':
            weight = 1.0/(sum(similarities))
            result = 0.0
            for i in xrange(len(yvals)):
                y = yvals[i]
                similarity = similarities[i]
                result += y*similarity*weight
            return result


    def score(self, X, Y):
        if not len(X) == len(Y):
            raise ValueError("X and Y must be same length.")
        errors = [Y[i] - self.predict(X[i]) for i in xrange(len(X))]
        return np.std(errors)


    def __init__(self, forest = None, n_neighbors = 3, weighting = 'equal'):
        #Check forest type.
        if forest == None:
            forest = RandomForestRegressor(n_estimators = 1000, n_jobs = -1)
        if not isinstance(forest, RandomForestRegressor):
            raise TypeError("Must use Random Forest Regressor to initialize.")
        # Set params
        self.weighting = weighting
        # Done. Call parent constructor
        super(RFNNRegressor, self).__init__(forest, n_neighbors)


class _BaselineRegressor(object):
    '''Always predicts the mean of the training set.'''
    def __init__(self, guess=None):
        self.guess = guess
    def fit(self, xtrain, ytrain):
        '''Find the average of input lidt of target values and guess on that
         from now on.'''
        self.guess = np.mean(ytrain)
    def predict(self, x):
        return self.guess


class _BaselineClassifier(object):
    '''Always predicts the most common label in the training set'''
    def __init__(self, guess=None):
        self.guess = guess
    def fit(self, xtrain, ytrain):
        '''Find the most common label and guess on that from now on.'''
        countmap = Counter(ytrain)
```

```
        best = 0
        for label, count in countmap.iteritems():
            if count > best:
                best = count
                self.guess = int(label)
            #
        #
    def predict(self, x):
        return self.guess


def _worker(X, Y, score_type, train_percentage, classifier, clf_args, n_groups,
            replace, threshold):
    '''Worker method for parallelizing bootstrap evaluations.'''
    #Create bootstrap sample
    try:
        rand = np.random.RandomState() #Ensures PRNG works in children
        indices = rand.choice(xrange(len(X)), size = len(X), replace = replace)
        xsample = [X[i] for i in indices]
        ysample = [Y[i] for i in indices]

        #Generate training and testing set
        cut = int(train_percentage*len(X))
        xtrain = xsample[:cut]
        xtest = xsample[cut:]
        ytrain = ysample[:cut]
        ytest = ysample[cut:]

        #Discard features with too low correlation with output vectors
        inds = get_strong_indices(X=xtrain, Y=ytrain, threshold = threshold)
        xtrain = reduce_data(xtrain, inds)
        xtest = reduce_data(xtest, inds)

        #Create regressor if we're doing regression
        if classifier == 'RandomForestRegressor':
            clf = RandomForestRegressor(**clf_args)
        elif classifier == 'SVR':
            clf = svm.SVR(**clf_args)
        elif classifier == 'baseline_mean':
            clf = _BaselineRegressor()
        elif classifier == 'WRBFR':
            importances = get_correlations(xtrain, ytrain)
            clf_args['importances'] = importances
            clf = WRBFR(**clf_args)

        #Create classifier and split dataset into labels
        elif classifier == 'RandomForestClassifier':
            clf = RandomForestClassifier(**clf_args)
            ysample = assign_labels(ysample, n_groups)
            ytrain = ysample[:cut]
            ytest = ysample[cut:]
        elif classifier == 'SVC':
            clf = svm.SVC(**clf_args)
            ysample = assign_labels(ysample, n_groups)
            ytrain = ysample[:cut]
            ytest = ysample[cut:]
        elif classifier == 'baseline_most_common_label':
            clf = _BaselineClassifier()
            ysample = assign_labels(ysample, n_groups)
            ytrain = ysample[:cut]
            ytest = ysample[cut:]
        elif classifier == 'WRBFC':
```

```
            importances = get_correlations(xtrain, ytrain)
            clf_args['importances'] = importances
            clf = WRBFC(**clf_args)
            ysample = assign_labels(ysample, n_groups)
            ytrain = ysample[:cut]
            ytest = ysample[cut:]

        #Fail if none of the above classifiers were specified
        else:
            raise ValueError('Regressor or classifier not defined.')

        #Fit the classifier or regressor
        clf.fit(xtrain, ytrain)

        #Compute score and append to output list
        if score_type == 'mse':
            scores = [(ytest[i] - clf.predict(xtest[i]))**2
                  for i in xrange(len(xtest))]
            return np.mean(scores)

        elif score_type == 'fraction_correct':
            n_correct = sum([ytest[i] == int(clf.predict(xtest[i]))
                          for i in xrange(len(ytest))])
            score = n_correct/len(ytest)
            return score
        elif score_type == 'over_baseline':
            #Get score
            score = sum([ytest[i] == int(clf.predict(xtest[i]))
                          for i in xrange(len(ytest))])
            #Get baseline
            baselineclf = _BaselineClassifier()
            baselineclf.fit(xtrain, ytrain)
            baseline = sum([ytest[i] == baselineclf.predict(xtest[i])
                          for i in xrange(len(ytest))])
            return score/baseline

        #Fail if none of the above performance metrics were specified
        else:
            raise ValueError('Score type not defined.')

        #Job's done!
        return None
    except:
        raise Exception("".join(traceback.format_exception(*sys.exc_info())))


def bootstrap(X, Y, classifier, score_type = 'mse', train_percentage = 0.8,
              clf_args = {}, iterations = 1000, n_groups = 3, n_jobs = 1,
              replace = False, threshold = 0.0):
    '''Performs bootstrap resampling to evaluate the performance of some
    classifier or regressor. Note that this takes the *complete dataset* as
    arguments as well as arguments specifying which predictor to use and which
    function to estimate the distribution of.
    This seems to be the most straightforward generalizable implementation
    which can be parallelized, as passing e.g. the scoring function directly
    clashed with the mechanisms implemented to work around the GIL for
    multiprocessing for obscure reasons.

    Parameters:
    ----------------
    X : list
      All feature vectors in the complete dataset.
```

```

    Y : list
      All 'true' labels or output values in the complete dataset.

    classifier : str
      Which classifier to use to predict the test set. Allowed values:
      'RandomForestRegressor', 'baseline_mean', 'SVR',
      'RandomForestClassifier', 'SVC', 'baseline_most_common_label, WRBFR,
      WRBFC'

    score_type : str
      String signifying which function to estimate the distribution of.
      Allowed values: 'mse', 'fraction_correct', 'over_baseline'

    train_percentage : float
      The percentage [0:1] of each bootstrap sample to be used for training.

    clf_args : dict
      optional arguments to the constructor method of the regressor/classifier.

    iterations : int
      Number of bootstrap samples to run.

    n_jobs : int
      How many cores (maximum) to use.

    replace : bool
      Whether to sample with replacement when obtaining the bootstrap samples.

    threshold : float
      features whose linear correlations with vectors in the output space
      are below this value are discarded.
    '''

    if not len(X) == len(Y):
        raise ValueError("X and Y must have equal length.")

    #Arguments to pass to worker processes
    d = {'X' : X, 'Y' : Y, 'train_percentage' : train_percentage,
         'classifier' : classifier, 'clf_args' : clf_args,
         'score_type' : score_type, 'n_groups' : n_groups,
         'replace' : replace, 'threshold' : threshold}

    #Make job queue
    pool = multiprocessing.Pool(processes = n_jobs)
    jobs = [pool.apply_async(_worker, kwds = d) for _ in xrange(iterations)]
    pool.close() #run
    pool.join()  #Wait for remaining jobs

    #Make sure no children died too early
    if not all(job.successful() for job in jobs):
        raise RuntimeError('Some jobs failed.')

    return [j.get() for j in jobs]

def get_correlations(X, Y, absolute = True):
    '''Given a list of feature vectors X and labels or values Y, returns a list
     of correlation coefficients for each dimension of the feature vectors.'''
    n_feats = len(X[0])
    correlations = []
    for i in xrange(n_feats):
        temp = np.corrcoef([x[i] for x in X], Y)
```

```
        correlation = temp[0,1]
        if math.isnan(correlation):
            correlation = 0
        correlations.append(correlation)
    if absolute:
        correlations = [np.abs(c) for c in correlations]
    return correlations


def make_kernel(importances, gamma = 1.0):
    '''Returns a weighted radial basis function (WRBF) kernel which can be
     passed to an SVM or SVR from the sklearn module.

     Parameters:
     -----------------------
     importances : list
       The importance of each input feature. The value of element i can mean
       e.g. the linear correlation between feature i and target variable y.
       None means feature will be weighted equally.

     gamma : float
       The usual gamma parameter denoting inverse width of the gaussian used.
    '''
    def kernel(x,y, *args, **kwargs):
        d = len(importances) #number of features
        impsum = sum([imp**2 for imp in importances])
        if not impsum == 0:
            normfactor = 1.0/np.sqrt(impsum)
        else:
            normfactor = 0.0
        #Metric to compute distance between points
        metric = dok_matrix((d,d), dtype = np.float64)
        for i in xrange(d):
            metric[i,i] = importances[i]*normfactor
        #
        result = np.zeros(shape = (len(x), len(y)))
        for i in xrange(len(x)):
            for j in xrange(len(y)):
                diff = x[i] - y[j]
                dist = diff.T.dot(metric*diff)
                result[i,j] = np.exp(-gamma*dist)
        return result
    return kernel

class WRBFR(svm.SVR):
    '''Weighted radial basis function support vector regressor.'''
    def __init__(self, importances, C = 1.0, epsilon = 0.1,
                 gamma = 0.0):
        kernel = make_kernel(importances = importances, gamma = gamma)
        super(WRBFR, self).__init__(C = C, epsilon = epsilon, kernel = kernel)

class WRBFC(svm.SVC):
    '''Weighted radial basis function support vector classifier.'''
    def __init__(self, importances, C = 1.0, gamma = 0.0):
        kernel = make_kernel(importances = importances, gamma = gamma)
        super(WRBFC, self).__init__(C = C, kernel = kernel)

if __name__ == '__main__':

    random.seed(42)
    X, Y, ind_dict = read_data('../data.json', trait = 'extraversion',
```

```
#                           features = ['call_iet_med', 'text_iet_med', 'social_entropy', 'call_entropy', 'travel_med', 'n_places', 'text_latency', 'call_night_activity']
                            features = 'all'
                            )
    corrs = get_correlations(X, Y)


#     for i in xrange(len(corrs)):
#         print corrs[i], ind_dict[i]

    cut = int(0.6*len(X))
    xtrain = X[:cut]
    ytrain = Y[:cut]
    xtest = X[cut:]
    ytest = Y[cut:]

    inds = get_strong_indices(xtrain, ytrain, threshold = 0.08)
    ind_dict = {i : ind_dict[inds[i]] for i in xrange(len(inds))}
    xtrain = reduce_data(xtrain, inds)
    xtest = reduce_data(xtest, inds)

    importances = get_correlations(xtrain, ytrain)
    clf = WRBFR(importances = importances, C = 26.5, epsilon=0.11, gamma = 0.23)
    clf_baseline = _BaselineRegressor()
#     clf = svm.SVR(C = 26.5, epsilon=0.11, gamma = 0.23)
    clf.fit(xtrain, ytrain)
    clf_baseline.fit(xtrain, ytrain)
    scores_baseline = []
    scores = []
    for i in xrange(len(xtest)):
        pred = clf.predict(xtest[i])
        pred_baseline = clf_baseline.predict(xtest[i])
        print pred
        scores.append((pred - ytest[i])**2)
        scores_baseline.append((pred_baseline - ytest[i])**2)
    print np.mean(scores)**0.5
    print np.mean(scores_baseline)**0.5

#     X =     [[1,2,7,0],[3,1,6,0.01],[6,8,1,0],[10,8,2,0.01]]
#     Y = [1,1,0,0]
#

#     corrs = get_correlations(X, Y)
#     print corrs
#
#     C = 70
#     gamma = 3.75
#
#     kernel = make_kernel(corrs, 0.0)
#
#     clf = svm.SVC(kernel = kernel)
#     clf = svm.SVC()
#     clf.fit(xtrain, ytrain)
#
#     hits = 0
#
#     for i in xrange(len(xtest)):
#         if clf.predict(xtest[i]) == ytest[i]:
#             hits += 1
#         #
#     print 100.0*hits/len(ytest)
#
```

```
#    for i in xrange(len(corrs)):
#        print ind_dict[i], corrs[i]
#
#    test = WRBFC(threshold=0.1, gamma = 0.2, importances=corrs)
#    test.fit(xtrain, ytrain)
#
#    preds = []
#    for i in xrange(len(ytest)):
#        pred = test.predict(xtest[i])[0]
#        print pred, ytest[i]
#        preds.append(pred)
#
#    print len(Counter(preds).keys())

#    print len(X[0])
##     print i
#    print [el for el in i.values() if 'init' in el]
```

A.1.5 Lloyd's Algorithm

```
# -*- coding: utf-8 -*-
from __future__ import division

import numpy as np
import matplotlib
matplotlib.use('Agg') #ugly hack to allow plotting from terminal
import matplotlib.pyplot as plt
import random
from copy import deepcopy

def _dist(p,q):
    return sum([(p[i]-q[i])**2 for i in xrange(len(p))])

def _lloyds_single_run(X, K, max_iterations, init):
    # Initialize with a subset of the data
    if init == 'sample':
        initials = random.sample(X, K)
    # Or initialize with random points across the same range as data
    elif init == 'scatter':
        vals = zip(*X)
        xmin = min(vals[0])
        xmax = max(vals[0])
        ymin = min(vals[1])
        ymax = max(vals[1])
        initials = [(random.uniform(xmin, xmax),
                     random.uniform(ymin, ymax)) for _ in xrange(K)]
    # Or yell RTFM at user
    else:
        raise ValueError('Invalid initialization mode!')

    #Contruct hashmap mapping integers up to K to centroids
    centroids = dict(enumerate(initials))
    converged = False
    iterations = 0

    while not converged and iterations < max_iterations:
        clusters = {i : [] for i in xrange(K)}
```

```
        #Make sure clusters and centroids have identical keys, or we're doomed.
        assert set(clusters.keys()) == set(centroids.keys())
        prev_centroids = deepcopy(centroids)

        ### STEP ONE -update clusters
        for x in X:
            #Check distances to all centroids
            bestind = -1
            bestdist = float('inf')
            for ind, centroid in centroids.iteritems():
                dist = _dist(x, centroid)
                if dist < bestdist:
                    bestdist = dist
                    bestind = ind
                #
            clusters[bestind].append(x)

        ### STEP TWO -update centroids
        for ind, points in clusters.iteritems():
            if not points:
                pass  #Cluster's empty - nothing to update
            else:
                centroids[ind] = np.mean(points, axis = 0)

        ### We're converged when all old centroids = new centroids.
        converged = all([_dist(prev_centroids[k],centroids[k]) == 0
                    for k in xrange(K)])
        iterations += 1
        #
    return {tuple(centroids[i]) : clusters[i] for i in xrange(K)}

def lloyds(X, K, runs = 1, max_iterations = float('inf'), init = 'sample'):
    '''Runs Lloyd's algorithm to identify K clusters in the dataset X.
    X is a list of points like [[x1,y1],[x2,y2]---].
    Returns a hash of centroids mapping to points in the corresponding cluster.
    The objective is to minimize the sum of distances from each centroid to
    the points in the corresponding cluster. It might only converge on a local
    minimum, so the configuration with the lowest score (sum of distances) is
    returned.
    init denotes initialization mode, which can be 'sample', using a randomly
    select subset of the input data, or 'scatter', using random points selected
    from the same range as the data as initial centroids.

    Parameters
    ----------------
    X : array_like
      list of points. 2D example: [[3,4],[3.4, 7.2], ...]

    K : int
      Number of centroids

    runs : int
      Number of times to run the entire algorithm. The result with the lowest
      score will be returned.

    max_iterations : int or float
      Number of steps to allow each run. Default if infinit, i.e. the algorithm
      runs until it's fully converged.

    init : str
      Initialization mode. 'sample' means use a random subset of the data as
      starting centroids. 'scatter' means place starting centroids randomly in
```

```
        the entire x-y range of the dataset.

      Returns
      --------------
      result : dict
        A dictionary in which each key is a tuple of coordinated corresponding to
        a centroid, and each value is a list of points belonging to that cluster.
        '''

    record = float('inf')
    result = None
    for _ in xrange(runs):
        clusters = _lloyds_single_run(X, K, max_iterations = max_iterations,
                                      init = init)
        #Determine how good the clusters came out
        score = 0
        for centroid, points in clusters.iteritems():
            score += sum([_dist(centroid, p) for p in points or [] ])
        if score < record:
            result = clusters
            record = score
        #
    return result


def _makecolor():
    i = 0
    cols = ['b', 'g', 'r', 'c', 'm', 'y']
    while True:
        yield cols[i]
        i = (i+1)%len(cols)


def draw_clusters(clusters, threshold = 0, show = True, filename = None):
    '''Accepts a dict mapping cluster centroids to cluster points and makes
     a color-coded plot of them. Clusters containing fewer points than the
     threshold are plottet in black.'''
    colors = _makecolor()
    plt.figure()
    for centroid, points in clusters.iteritems():
        if not points:
            continue
        if len(points) < threshold:
            style = ['k,']
        else:
            color = colors.next()
            style = [color+'+']
            #Plot centroids
            x,y = centroid
            plt.plot(x,y, color = color, marker = 'd', markersize = 12)
        #plot points
        plt.plot(*(zip(*points)+style))
    if filename:
        plt.savefig(filename, bbox_inches = 'tight')
    if show:
        plt.show()


if __name__ == '__main__':
    points = [[random.uniform(-10,10), random.uniform(-10,10)] for _ in xrange(10**3)]
    clusters = lloyds(X = points, K = 6, runs = 1)
    draw_clusters(clusters = clusters, filename = 'lloyds_example.pdf')
```

A.1.6 Smallest Enclosing Circle

```
# -*- coding: utf-8 -*-
#
# Smallest enclosing circle
#
# Copyright (c) 2014 Project Nayuki
# http://www.nayuki.io/page/smallest-enclosing-circle
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program (see COPYING.txt).
# If not, see <http://www.gnu.org/licenses/>.
#

import math, random


# Data conventions: A point is a pair of floats (x, y). A circle is a triple of floats (center x, center y, radius).

#
# Returns the smallest circle that encloses all the given points. Runs in expected O(n) time, randomized.
# Input: A sequence of pairs of floats or ints, e.g. [(0,5), (3.1,-2.7)].
# Output: A triple of floats representing a circle.
# Note: If 0 points are given, None is returned. If 1 point is given, a circle of radius 0 is returned.
#
def make_circle(points):
    '''Accepts list of points as tuples and returns (x, y, r).'''
    # Convert to float and randomize order
    shuffled = [(float(p[0]), float(p[1])) for p in points]
    random.shuffle(shuffled)

    # Progressively add points to circle or recompute circle
    c = None
    for (i, p) in enumerate(shuffled):
        if c is None or not _is_in_circle(c, p):
            c = _make_circle_one_point(shuffled[0 : i + 1], p)
    return c


# One boundary point known
def _make_circle_one_point(points, p):
    c = (p[0], p[1], 0.0)
    for (i, q) in enumerate(points):
        if not _is_in_circle(c, q):
            if c[2] == 0.0:
```

```
                c = _make_diameter(p, q)
            else:
                c = _make_circle_two_points(points[0 : i + 1], p, q)
    return c


# Two boundary points known
def _make_circle_two_points(points, p, q):
    diameter = _make_diameter(p, q)
    if all(_is_in_circle(diameter, r) for r in points):
        return diameter

    left = None
    right = None
    for r in points:
        cross = _cross_product(p[0], p[1], q[0], q[1], r[0], r[1])
        c = _make_circumcircle(p, q, r)
        if c is None:
            continue
        elif cross > 0.0 and (left is None or _cross_product(p[0], p[1], q[0], q[1], c[0], c[1]) > _cross_product(
            p[0], p[1], q[0], q[1], left[0], left[1])):
            left = c
        elif cross < 0.0 and (right is None or _cross_product(p[0], p[1], q[0], q[1], c[0], c[1]) < _cross_product(
            p[0], p[1], q[0], q[1], right[0], right[1])):
            right = c
    return left if (right is None or (left is not None and left[2] <= right[2])) else right


def _make_circumcircle(p0, p1, p2):
    # Mathematical algorithm from Wikipedia: Circumscribed circle
    ax = p0[0]; ay = p0[1]
    bx = p1[0]; by = p1[1]
    cx = p2[0]; cy = p2[1]
    d = (ax * (by - cy) + bx * (cy - ay) + cx * (ay - by)) * 2.0
    if d == 0.0:
        return None
    x = ((ax * ax + ay * ay) * (by - cy) + (bx * bx + by * by) * (cy - ay) + (cx * cx + cy * cy) * (ay - by)) / d
    y = ((ax * ax + ay * ay) * (cx - bx) + (bx * bx + by * by) * (ax - cx) + (cx * cx + cy * cy) * (bx - ax)) / d
    return (x, y, math.hypot(x - ax, y - ay))


def _make_diameter(p0, p1):
    return ((p0[0] + p1[0]) / 2.0, (p0[1] + p1[1]) / 2.0, math.hypot(p0[0] - p1[0], p0[1] - p1[1]) / 2.0)


_EPSILON = 1e-12

def _is_in_circle(c, p):
    return c is not None and math.hypot(p[0] - c[0], p[1] - c[1]) < c[2] + _EPSILON


# Returns twice the signed area of the triangle defined by (x0, y0), (x1, y1), (x2, y2)
def _cross_product(x0, y0, x1, y1, x2, y2):
    return (x1 - x0) * (y2 - y0) - (y1 - y0) * (x2 - x0)

#pts = [(0.0, 0.0), (6.0, 8.0)]
#
#test = make_circle(pts)
#
#print test
```

A.2 Source Code for Explicit Semantic Analysis

This section contains code pertaining to part II of the thesis.

A.2.1 Parser

```
# -*- coding: utf-8 -*-
'''Parses a full Wikipedia XML-dump and saves to files containing
a maximum of 1000 articles.
In the end, each file is saved as a JSON file containing entries like:
{{
    'concept':
    {
    'text': <article contents>,
    'links_in' : Set of links TO the article in question,
    'links_out' : Set of links FROM the article in question,
    }
}
Although links_in is added by the generate_indices script.
Also saved are dicts for keeping track of word and concept indices when
building a large sparse matrix for the semantic interpreter.
The file structure is like {'word blah' : index blah}'''

import re
import xml.sax as SAX
import wikicleaner
import os
import glob
import shared
import sys

DEFAULT_FILENAME = 'medium_wiki.xml'

def canonize_title(title):
    # remove leading whitespace and underscores
    title = title.strip(' _')
    # replace  sequencesences of whitespace and underscore chars with a single space
    title = re.compile(r'[\s_]+').sub(' ', title)
    #remove forbidden characters
    title = re.sub('[?/\\\*\"\']','',title)
    return title.title()

#Import shared parameters
from shared import extensions, temp_dir

#Cleanup
for ext in extensions.values():
    for f in glob.glob(temp_dir + '*'+ext):
        os.remove(f)

def filename_generator(folder):
    '''Generator for output filenames'''
    if not os.path.exists(folder):
        os.makedirs(folder)
    count = 0
    while True:
        filename = folder+"content"+str(count)
```

```python
        count += 1
        yield filename

make_filename = filename_generator(temp_dir)

#Format {right title : redirected title }, e.g. {because : ([cuz, cus])}
redirects = {}

#Minimum number of links/words required to keep an article.
from shared import min_links_out, min_words

#Open log file for writing and import logging function
logfile = open(os.path.basename(__file__)+'.log', 'w')
log = shared.logmaker(logfile)

class WikiHandler(SAX.ContentHandler):
    '''ContentHandler class to process XML and deal with the WikiText.
     It works basically like this:
     It traverses the XML file, keeping track of the type of data being read and
     adding any text to its input buffer. When event handlers register a page
     end, the page content is processed, the processed content is placed in the
     output buffer, and the input buffer is flushed.
     Whenever a set number of articles have been processed, the output buffer is
     written to a file. The point of this approach is to
     limit memory consumption.'''

    def __init__(self):
        SAX.ContentHandler.__init__(self)
        self.current_data = None
        self.title = ''
        self.input_buffer = []
        self.output_buffer = {}
        self.article_counter = 0
        self.links = []
        self.categories = []
        self.redirect = None
        self.verbose = False
        #Harvest unique words here
        self.words = set([])
        #keeps track of ingoing article links. format {to : set([from])}
        self.linkhash = {}

    def flush_input_buffer(self):
        '''Deletes info on the currently processed article.
         This is called when a page end event is registered.'''
        self.input_buffer = []
        self.current_data = None
        self.title = ''
        self.links = []
        self.categories = []
        self.redirect = None

    def flush_output_buffer(self):
        '''Flushes data gathered so far to a file and resets.'''
        self.output_buffer = {}
        self.words = set([])
        self.linkhash = {}

    def startElement(self, tag, attrs):
        '''Eventhandler for element start - keeps track of current datatype.'''
        self.current_data = tag
        #Informs the parser of the redirect destination of the article
```

```python
        if tag == "redirect":
            self.redirect = attrs['title']
            return None

    def endElement(self, name):
        '''Eventhandler for element end. This causes the parser to process
         its input buffer when a pageend is encountered.'''
        #Process content after each page
        if name == 'page':
            self.process()
        #Write remaining data at EOF.
        elif name == 'mediawiki':
            self.writeout()

    def characters(self, content):
        '''Character event handler. This simply passes any raw text from an
         article field to the input buffer and updates title info.'''
        if self.current_data == 'text':
            self.input_buffer.append(content)
        elif self.current_data == 'title' and not content.isspace():
            self.title = content

    def process(self):
        '''Process input buffer contents. This converts wikilanguage to
         plaintext, registers link information and checks if content has
         sufficient words and outgoing links (ingoing links can't be checked
         until the full XML file is processed).'''

        #Ignore everything else if article redirects
        if self.redirect:
            self.flush_input_buffer()
            return None
            global redirects
            try:
                redirects[self.title].add(self.redirect)
            except KeyError:
                redirects[self.title] = set([self.redirect])
            self.flush_input_buffer()
            return None

        #Redirects handled - commence processing
        print "processing: "+self.title.encode('utf8')
        #Combine buffer content to a single string
        text = ''.join(self.input_buffer).lower()

        #Find and process link information
        link_regexp = re.compile(r'\[\[(.*?)\]')
        links = re.findall(link_regexp, text) #grap stuff like [[<something>]]
        #Add links to the parsers link hash
        for link in links:
            #Check if link matches a namespace, e.g. 'file:something.png'
            if any([ns+':' in link for ns in wikicleaner.namespaces]):
                continue #Proceed to next link
            #Namespaces done, so remove any colons:
            link = link.replace(':', '')
            if not link:
                continue #Some noob could've written an empty link...
            #remove chapter designations/displaytext - keep article title
            raw = re.match(r'([^\|\#]*)', link).group(0)
            title = canonize_title(raw)
            #note down that current article has outgoing link to 'title'
            self.links.append(title)
```

```
            #also note that 'title' has incoming link from here
            try:
                self.linkhash[title].add(self.title) #maps target->sources
            except KeyError:
                self.linkhash[title] = set([self.title])

        #Disregard current article if it contains too few links
        if len(self.links) < min_links_out:
            self.flush_input_buffer()
            return None

        #Cleanup text
        text = wikicleaner.clean(text)
        article_words = text.split()

        #Disregard article if it contains too few words
        if len(article_words) < min_words:
            self.flush_input_buffer()
            return None

        #Update global list of unique words
        self.words.update(set(article_words))

        #Add content to output buffer
        output = {
            'text' : text,
            #Don't use category info for now
            #'categories' : self.categories,
            'links_out' : self.links
        }
        self.output_buffer[self.title] = output
        self.article_counter += 1

        #Flush output buffer to file
        if self.article_counter%1000 == 0:
            self.writeout()

        #Done, flushing buffer
        self.flush_input_buffer()
        return None

    def writeout(self):
        '''Writes output buffer contents to file'''
        #Generate filename and write to file
        filename = make_filename.next()
        #Write article contents to file
        with open(filename+extensions['content'], 'w') as f:
            shared.dump(self.output_buffer, f)

        #Store wordlist as files
        with open(filename+extensions['words'], 'w') as f:
            shared.dump(self.words, f)

        #Store linkhash in files
        with open(filename+extensions['links'], 'w') as f:
            shared.dump(self.linkhash, f)

        if self.verbose:
            log("wrote "+filename)

        #Empty output buffer
        self.flush_output_buffer()
```

```python
        return None

if __name__ == "__main__":
    if len(sys.argv) == 2:
        file_to_parse = sys.argv[1]
    else:
        file_to_parse = DEFAULT_FILENAME

    #Create and configure content handler
    test = WikiHandler()
    test.verbose = True

    #Create a parser and set handler
    ATST = SAX.make_parser()
    ATST.setContentHandler(test)

    #Let the parser walk the file
    log("Parsing started...")
    ATST.parse(file_to_parse)
    log("...Parsing done!")

    #Attempt to send notification that job is done
    if shared.notify:
        try:
            shared.pushme(sys.argv[0]+' completed.')
        except:
            log("Job's done. Push failed.")

    logfile.close()
```

A.2.2 Index Generator

```
# -*- coding: utf-8 -*-
'''This finishes preproccessing of the output from the XML parser.
This script reads in link data and removes from the content files those
concepts that have too few incoming links. Information on incoming links
is saved to each content file.
Finally, index maps for words and approved concepts are generated and saved.'''

from __future__ import division
import glob
import gc
import shared
import os
import sys

logfile = open(os.path.basename(__file__)+'.log', 'w')
log = shared.logmaker(logfile)

#Import shared parameters
from shared import extensions, temp_dir, min_links_in, matrix_dir

def listchopper(l):
    '''Generator to chop lists into chunks of a predefined length'''
    n = shared.link_chunk_size
    ind = 0
    while ind < len(l):
        yield l[ind:ind+n]
        ind += n

def main():
    #Import shared parameters and verify output dir exists
    if not os.path.exists(temp_dir):
        raise IOError

#===============================================================================
#      Read in link data and update content files accordingly
#===============================================================================

    #Get list of files containing link info and chop it up
    linkfiles = glob.glob(temp_dir + '*'+extensions['links'])
    linkchunks = listchopper(linkfiles)

    linkfiles_read = 0
    for linkchunk in linkchunks:
        #Hash mapping each article to a set of articles linking to it
        linkhash = {}

        for filename in linkchunk:
            with open(filename, 'r') as f:
                newstuff = shared.load(f)
            #Add link info to linkhash
            for target, sources in newstuff.iteritems():
                try:
                    linkhash[target].update(set(sources))
                except KeyError:
                    linkhash[target] = set(sources)

            #Log status
```

```
            linkfiles_read += 1
            log("Read " + filename + " - " +
                str(100*linkfiles_read/len(linkfiles))[:4] + " % of link data.")

        log("Chunk finished - updating content files")
        #Update concept with newly read link data
        contentfiles = glob.glob(temp_dir + '*'+extensions['content'])
        contentfiles_read = 0
        for filename in contentfiles:
            #Read file. Content is like {'article title' : {'text' : blah}}
            with open(filename, 'r') as f:
                content = shared.load(f)

            #Search linkhash for links going TO concept
            for concept in content.keys():
                try:
                    sources = linkhash[concept]
                except KeyError:
                    sources = set([])  #Missing key => zero incoming links

                #Update link info for concept
                try:
                    content[concept]['links_in'] = set(content[concept]['links_in'])
                    content[concept]['links_in'].update(sources)
                except KeyError:
                    content[concept]['links_in'] = sources

            #Save updated content
            with open(filename, 'w') as f:
                shared.dump(content, f)

            contentfiles_read += 1
            if contentfiles_read % 100 == 0:
                log("Fixed " + str(100*contentfiles_read/len(contentfiles))[:4]
                    + "% of content files")
        pass  #Proceed to next link chunk

#==============================================================================
#       Finished link processing
#       Remove unworthy concepts and combine concept/word lists.
#==============================================================================

    #What, you think memory grows on trees?
    del linkhash
    gc.collect()

    #Set of all approved concepts
    concept_list = set([])

    #Purge inferior concepts (with insufficient incoming links)
    for filename in contentfiles:
        #Read in content file
        with open(filename, 'r') as f:
            content = shared.load(f)

        for concept in content.keys():
            entry = content[concept]
            if 'links_in' in entry and len(entry['links_in']) >= min_links_in:
                concept_list.add(concept)
            else:
                del content[concept]
```

```
            with open(filename, 'w') as f:
                shared.dump(content, f)

    log("Links done - saving index files")

    #Make sure output dir exists
    if not os.path.exists(matrix_dir):
        os.makedirs(matrix_dir)

    #Generate and save a concept index map. Structure: {concept : index}
    concept_indices = {n: m for m,n in enumerate(concept_list)}
    with open(matrix_dir+'concept2index.ind', 'w') as f:
        shared.dump(concept_indices, f)

    #Read in all wordlists and combine them.
    words = set([])
    for filename in glob.glob(temp_dir + '*'+extensions['words']):
        with open(filename, 'r') as f:
            words.update(shared.load(f))

    #Generate and save a word index map. Structure: {word : index}
    word_indices = {n: m for m,n in enumerate(words)}
    with open(matrix_dir+'word2index.ind', 'w') as f:
        shared.dump(word_indices, f)

    log("Wrapping up.")
    #Attempt to notify that job is done
    if shared.notify:
        try:
            shared.pushme(sys.argv[0]+' completed.')
        except:
            log("Job's done. Push failed.")

    logfile.close()

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

A.2.3 Matrix Builder

```python
# -*- coding: utf-8 -*-
'''Builds a huge sparse matrix of Term frequency/Inverse Document Frequency
(TFIDF) of the previously extracted words and concepts.
First a matrix containing simply the number of occurrences of word i in the
article corresponding to concept j is build (in DOK format as that is faster
for iterative construction), then the matrix is converted to sparse row format
(CSR), TFIDF values are computed, each row is normalized and finally pruned.'''

from __future__ import division
import scipy.sparse as sps
import numpy as np
from collections import Counter
import glob
import shared
import sys
import os

def percentof(small, large):
    return str(100*small/large) + "%"

logfile = open(os.path.basename(__file__)+'.log', 'w')
log = shared.logmaker(logfile)

#import shared parameters
from shared import (extensions, matrix_dir, prune, temp_dir, column_chunk_size,
                    row_chunk_size, datatype)

def main():
    #Cleanup
    for f in glob.glob(matrix_dir + '/*'+extensions['matrix']):
        os.remove(f)

    #Set pruning parameters
    window_size = shared.window_size
    cutoff = shared.cutoff

    #Read in dicts mapping words and concepts to their respective indices
    log("Reading in word/index data")
    word2index = shared.load(open(matrix_dir+'word2index.ind', 'r'))
    concept2index = shared.load(open(matrix_dir+'concept2index.ind', 'r'))
    log("...Done!")

#==============================================================================
#      Construct count matrix in small chunks
#==============================================================================

    #Count words and concepts
    n_words = len(word2index)
    n_concepts = len(concept2index)

    #Determine matrix dimensions
    matrix_shape = (n_words, n_concepts)

    #Allocate sparse matrix. Dict-of-keys should be faster for iterative
    #construction. Convert to csr for fast row operations later.
    mtx = sps.dok_matrix(matrix_shape, dtype = datatype)

```

```python
def matrix_chopper(matrix, dim):
    '''Generator to split a huge matrix into small submatrices, which can
     then be stored in individual files.
     This is handy both when constructing the matrix (building the whole
     matrix without saving to files in the process takes about 50 gigs RAM),
     and when applying it, as this allows one to load only the submatrix
     relevant to a given word.'''
    ind = 0
    counter = 0
    rows = matrix.get_shape()[0]
    while ind < rows:
        end = min(ind+dim, rows)
        #Return pair of submatrix number and the submatrix itself
        yield counter, sps.vstack([matrix.getrow(i)\
                                   for i in xrange(ind, end)], format = 'csr')
        counter += 1
        ind += dim

def writeout():
    '''Saves the matrix as small submatrrices in separate files.'''
    for n, submatrix in matrix_chopper(mtx, row_chunk_size):
        filename = matrix_dir+str(n)+extensions['matrix']
        #Update submatrix if it's already partially calculated
        log("Writing out chunk %s" % n)
        try:
            with open(filename, 'r') as f:
                submatrix = submatrix + shared.mload(f)
            #
        except IOError:
            pass #File doesn't exist yet, so no need to change mtx

        #Dump the submatrix to file
        with open(filename, 'w') as f:
            shared.mdump(submatrix, f)
    return None

log("Constructing matrix.")
filelist = glob.glob(temp_dir + '*'+extensions['content'])
files_read = 0
for filename in filelist:
    with open(filename, 'r') as f:
        content = shared.load(f)

    #Loop over concepts (columns) as so we don't waste time with rare words
    for concept, entry, in content.iteritems():
        #This is the column index (concept w. index j)
        j = concept2index[concept]

        #Convert concept 'countmap' like so: {word : n}
        wordmap = Counter(entry['text'].split()).iteritems()

        #Add them all to the matrix
        for word, count in wordmap:
            #Find row index of the current word
            i = word2index[word]

            #Add the number of times word i occurs in concept j to the matrix
            mtx[i,j] = count
        #
    #Update file count
    files_read += 1
    log("Processed content file no. %s of %s - %s"
```

```
            % (files_read, len(filelist)-1, percentof(files_read, len(filelist))))

        if files_read % column_chunk_size == 0:
            mtx = mtx.tocsr()
            writeout()
            mtx = sps.dok_matrix(matrix_shape)
        #

    #Convert matrix to CSR format and write to files.
    mtx = mtx.tocsr()
    writeout()

#==============================================================================
# Count matrix/matrices constructed - computing TF-IDF
#==============================================================================

    log("Done - computing TF-IDF")

    #Grap list of matrix files (containing the submatrices from before)
    matrixfiles = glob.glob(matrix_dir + "*" + extensions['matrix'])
    words_processed = 0 #for logging purposes

    for filename in matrixfiles:
        with open(filename, 'r') as f:
            mtx = shared.mload(f)

        #Number of words in a submatrix
        n_rows = mtx.get_shape()[0]

        for w in xrange(n_rows):
            #Grap non-zero elements from the row corresonding to word w
            row = mtx.data[mtx.indptr[w] : mtx.indptr[w+1]]
            if len(row) == 0:
                continue

            #Make a vectorized function to convert a full row to TF-IDF
            f = np.vectorize(lambda m_ij: (1+np.log(m_ij))*
                             np.log(n_concepts/len(row)))

            #Map all elements to TF-IDF and update matrix
            row = f(row)

            #Normalize the row
            assert row.dtype.kind == 'f' #Non floats round to zero w/o warning
            normfact = 1.0/np.linalg.norm(row)
            row *= normfact

            #Start inverted index pruning
            if prune:
                #Number of documents containing w
                n_docs = len(row)

                #Don't prune if the windows exceeds the array bounds (duh)
                if window_size < n_docs:

                    #Obtain list of indices such that row[index] is sorted
                    indices = np.argsort(row)[::-1]

                    #Generate a sorted row
                    sorted_row = [row[index] for index in indices]

                    #Go through sorted row and truncate when pruning condition is met
```

```
                    for i in xrange(n_docs-window_size):
                        if sorted_row[i+window_size] >= cutoff*sorted_row[i]:
                            #Truncate, i.e. set the remaining entries to zero
                            sorted_row[i:] = [0]*(n_docs-i)
                            break
                        else:
                            pass

                    #Unsort to original positions
                    for i in xrange(n_docs):
                        row[indices[i]] = sorted_row[i]

                #Update matrix
                mtx.data[mtx.indptr[w] : mtx.indptr[w+1]] = row

                #Log it
                words_processed += 1
                if words_processed % 10**3 == 0:
                    log("Processing word %s of %s - %s" %
                        (words_processed, n_words,
                         percentof(words_processed, n_words)))

            #Keep it sparse - no need to store zeroes
            mtx.eliminate_zeros()
            with open(filename, 'w') as f:
                shared.mdump(mtx, f)

        log("Done!")

        #Notify that the job is done
        if shared.notify:
            try:
                shared.pushme(sys.argv[0]+' completed.')
            except:
                log("Job's done. Push failed.")

        logfile.close()
        return None

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

A.2.4 Library for Computational Linguistics

```python
# -*- coding: utf-8 -*-
'''Small module for computational linguistics applied to Twitter.
The main classes are a TweetHarvester, which gathers data from Twitters' API,
and a SemanticAnalyser, which relies on the previously constructed TFIDF
matrices.'''

from __future__ import division
from scipy import sparse as sps
from collections import Counter
from numpy.linalg import norm
import re
import shared
import tweepy
from datetime import date
import json
import time
import sys
import codecs
import os
from pprint import pprint
sys.stdout = codecs.getwriter('utf8')(sys.stdout)
sys.stderr = codecs.getwriter('utf8')(sys.stderr)

#==============================================================================
# This stuff defines a twitter 'harvester' for downloading Tweets
#==============================================================================

#Import credentials for accessing Twitter API
from supersecretstuff import consumer_key, consumer_secret, access_token, access_token_secret
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)

class listener(tweepy.StreamListener):
    '''Listener class to access Twitter stream.'''
    #What to do with a tweet (override later)
    def process(self, content):
        print content
        return None

    def on_status(self, status):
        self.process(status)
        return True

    def on_error(self, status):
        print status

# Exception to be raised when the Twitter API messes up. Happens occasionally.
class IncompleteRead(Exception):
    pass

class TweetHarvester(object):
    '''Simple class to handle tweet harvest.
     Harvest can be performed actively or passively, i.e. using the 'mine'
     method to gather a fixed number of tweets or using the 'listen' method
     to stream tweets matching a given search term.
     Harvested tweets are sent to the process method which by default simply
     stores them inside the object.'''
```

```

    def __init__(self, max_tweets=-1, verbose = False, tweets_pr_file = 10**5):
        #Set parameters
        self.max_tweets = max_tweets #-1 for unlimited stream
        self.verbose = verbose
        self.tweets_pr_file = tweets_pr_file

        #Internal parameters to keep track of harvest status
        self.files_saved = 0
        self.harvested_tweets = []
        self.current_filenumber = 0
        self.current_date = date.today()

    def filename_maker(self):
        #Update counter and date if neccessary
        if not self.current_date == date.today():
            self.current_date = date.today()
            self.current_filenumber = 0
        else:
            pass #Date hasn't changed. Proceed.
        filename = str(self.current_date) + "-data%s.json" % self.current_filenumber
        self.current_filenumber += 1
        return filename

    #Simple logging function
    def log(self, text):
        string = text+" at "+time.asctime()+"\n"
        if self.verbose:
            print string
        with open('tweetlog.log', 'a') as logfile:
            logfile.write(string)
        #Must return true so I can log errors without breaking the stream.
        return True

    def listen(self, search_term):
        #Make a listener
        listener = tweepy.StreamListener()
        #Override relevant methods.
        listener.on_status = self.process
        listener.on_error = lambda status_code: self.log("Error: "+status_code)
        listener.on_timeout = lambda: self.log("Timeout.")

        twitterStream = tweepy.Stream(auth, listener)
        twitterStream.filter(track=search_term)

    def mine(self, search_term, n = None):
        '''Mine a predefined number of tweets using input search word'''
        if n == None:
            n = self.max_tweets

        api = tweepy.API(auth)
        tweets = tweepy.Cursor(api.search, q=search_term).items(n)
        for tweet in tweets:
            self.process(tweet)

    def process(self, tweet):
        self.harvested_tweets.append(tweet)
        if self.verbose:
            print "Holding %s tweets." % len(self.harvested_tweets)

        #Write to file if buffer is full
        if len(self.harvested_tweets) == self.tweets_pr_file:
```

```
            self.writeout()

        #Check if limit has been reached (returning false cuts off listener)
        return not (len(self.harvested_tweets) == self.max_tweets)

    def writeout(self):
        filename = self.filename_maker()
        with open(filename,'w') as outfile:
            outfile.writelines([json.dumps(t._json)+"\n"
                            for t in self.harvested_tweets])

        self.harvested_tweets = []
        self.files_saved += 1
        #Log event
        s = "Saved %s files" % self.files_saved
        self.log(s)


#==============================================================================
# Defines stuff to analyse text using an already constructed interpretation
# matrix.
#==============================================================================

from shared import matrix_dir, row_chunk_size, extensions

class SemanticAnalyser(object):
    '''Analyser class using Explicit Semantic Analysis (ESA) to process
     text fragments. It can compute semantic (pseudo) distance and similarity,
     as well'''
    def __init__(self, matrix_filename = 'matrix.mtx'):
        #Hashes for word and concept indices
        with open(matrix_dir+'word2index.ind', 'r') as f:
            self.word2index = shared.load(f)
        with open(matrix_dir+'concept2index.ind', 'r') as f:
            self.concept2index = shared.load(f)
        self.index2concept = {i : c for c, i in self.concept2index.iteritems()}

        #Count number of words and concepts
        self.n_words = len(self.word2index)
        self.n_concepts = len(self.concept2index)

    def clean(self, text):
        text = re.sub('[^\w\s\d\'\-]','', text)
        text = text.lower()

        return text

    def interpretation_vector(self, text):
        '''Converts a text fragment string into a row vector where the i'th
         entry corresponds to the total TF-IDF score of the text fragment
         for concept i'''

        #Remove mess (quotes, parentheses etc) from text
        text = self.clean(text)

        #Convert string to hash like {'word' : no. of occurrences}
        countmap = Counter(text.split()).iteritems()

        #Interpretation vector to be returned
        result = sps.csr_matrix((1, self.n_concepts), dtype = float)

        #Add word count in the correct position of the vector
```

```
        for word, count in countmap:
            try:
                ind = self.word2index[word]
                #Which file to look in
                file_number = int(ind/row_chunk_size)
                filename = matrix_dir+str(file_number)+extensions['matrix']

                #And which row to extract
                row_number = ind % row_chunk_size

                #Do it! Do it naw!
                with open(filename, 'r') as f:
                    temp = shared.mload(f)
                    result = result + count*temp.getrow(row_number)
            except KeyError:
                pass    #No data on this word -> discard

        #Done. Return row vector as a 1x#concepts CSR matrix
        return result

    def interpret_text(self, text, display_concepts = 10):
        '''Attempts to guess the core concepts of the given text fragment'''
        #Compute the interpretation vector for the text fragment
        vec = self.interpretation_vector(text)

        #Magic, don't touch
        top_n = vec.data.argsort()[:len(vec.data)-1-display_concepts:-1]

        #List top scoring concepts and their TD-IDF
        concepts = [self.index2concept[vec.indices[i]] for i in top_n]
        return concepts
#        scores = [vec.data[i] for i in top_n]
#        #Return as dict {concept : score}
#        return dict(zip(concepts, scores))

    def interpret_file(self, filename):
        with open(filename, 'r') as f:
            data = self.clean(f.read())
        return self.interpret_text(data)

    def interpret_input(self):
        text = raw_input("Enter text fragment: ")
        topics = self.interpret_text(text)
        print "Based on your input, the most probable topics of your text are:"
        print topics[:self.display_concepts]

    def scalar(self, v1, v2):
        #Compute their inner product and make sure it's a scalar
        dot = v1.dot(v2.transpose())
        assert dot.shape == (1,1)

        if dot.data:
            scal = dot.data[0]
        else:
            scal = 0    #Empty sparse matrix means zero

        #Normalize and return
        sim = scal/(norm(v1.data)*norm(v2.data))
        return sim

    def cosine_similarity(self, text1, text2):
        '''Determines cosine similarity between input texts.
```

```
         Returns float in [0,1]'''

        #Determine intepretation vectors
        v1 = self.interpretation_vector(text1)
        v2 = self.interpretation_vector(text2)

        #Compute the normalized dot product and return
        return self.scalar(v1, v2)


    def cosine_distance(self, text1, text2):
        return 1-self.cosine_similarity(text1, text2)

if __name__ == '__main__':
    th = TweetHarvester(verbose=True, max_tweets=10)
    th.mine('carlsberg', n=10)
    temp = [t._json for t in th.harvested_tweets if t._json['lang'] == 'en']
    js = temp[4]
    with open('tweet_example.json', 'w') as f:
        pprint(js, stream=f)

#     if len(sys.argv) > 1:
#         fn = sys.argv[1]
#     else:
#         fn = 'interpret_me.txt'
#         with open(fn, 'r') as f:
#             data = f.read()
#         #
#     data = sa.clean(data)
#     guesses = sa.interpret_text(data)
#
#     if len(sys.argv) > 2:
#         output_filename = sys.argv[2]
#     else:
#         output_filename = 'guesses.txt'
#     with open(output_filename, 'w') as f:
#         for line in guesses:
#             f.write(line.encode('utf8'))
#             f.write('\n')
```

A.2.5 Wikicleaner

```
# -*- coding: utf-8 -*-
import re
from htmlentitydefs import name2codepoint

namespaces = set(['help', 'file talk', 'module', 'topic', 'mediawiki',
'wikipedia talk', 'file', 'user talk', 'special', 'category talk', 'category',
'media', 'wikipedia', 'book', 'draft', 'book talk', 'template', 'help talk',
'timedtext', 'mediawiki talk', 'portal talk', 'portal', 'user', 'module talk',
'template talk', 'education program talk', 'education program',
'timedtext talk', 'draft talk', 'talk'])

def dropNested(text, openDelim, closeDelim):
    '''Helper function to match nested expressions which may cause problems
     example: {{something something {{something else}} and something third}}
     cannot be easily matched with a regexp to remove all occurrences.
```

```
     Copied from the WikiExtractor project.'''
    openRE = re.compile(openDelim)
    closeRE = re.compile(closeDelim)
    # partition text in separate blocks { } { }
    matches = []                # pairs (s, e) for each partition
    nest = 0                    # nesting level
    start = openRE.search(text, 0)
    if not start:
        return text
    end = closeRE.search(text, start.end())
    next = start
    while end:
        next = openRE.search(text, next.end())
        if not next:            # termination
            while nest:         # close all pending
                nest -=1
                end0 = closeRE.search(text, end.end())
                if end0:
                    end = end0
                else:
                    break
            matches.append((start.start(), end.end()))
            break
        while end.end() < next.start():
            # { } {
            if nest:
                nest -= 1
                # try closing more
                last = end.end()
                end = closeRE.search(text, end.end())
                if not end:     # unbalanced
                    if matches:
                        span = (matches[0][0], last)
                    else:
                        span = (start.start(), last)
                    matches = [span]
                    break
            else:
                matches.append((start.start(), end.end()))
                # advance start, find next close
                start = next
                end = closeRE.search(text, next.end())
                break           # { }
        if next != start:
            # { { }
            nest += 1
    # collect text outside partitions
    res = ''
    start = 0
    for s, e in matches:
        res += text[start:s]
        start = e
    res += text[start:]
    return res

def unescape(text):
    '''Removes HTML or XML character references and entities
     from a text string.
     @return nice text'''
    def fixup(m):
        text = m.group(0)
        code = m.group(1)
```

```
            return text
        try:
            if text[1] == "#":  # character reference
                if text[2] == "x":
                    return unichr(int(code[1:], 16))
                else:
                    return unichr(int(code))
            else:                # named entity
                return unichr(name2codepoint[code])
        except UnicodeDecodeError:
            return text # leave as is

    return re.sub("&#?(\w+);", fixup, text)

def drop_spans(matches, text):
    """Drop from text the blocks identified in matches"""
    matches.sort()
    res = ''
    start = 0
    for s, e in  matches:
        res += text[start:s]
        start = e
    res += text[start:]
    return res

###Compile regexps for text cleanup:
#Construct patterns for elements to be discarded:
discard_elements = set([
        'gallery', 'timeline', 'noinclude', 'pre',
        'table', 'tr', 'td', 'th', 'caption',
        'form', 'input', 'select', 'option', 'textarea',
        'ul', 'li', 'ol', 'dl', 'dt', 'dd', 'menu', 'dir',
        'ref', 'references', 'img', 'imagemap', 'source'
        ])
discard_element_patterns = []
for tag in discard_elements:
    pattern = re.compile(r'<\s*%s\b[^>]*>.*?<\s*/\s*%s>' % (tag, tag), re.DOTALL | re.IGNORECASE)
    discard_element_patterns.append(pattern)

#Construct patterns to recognize HTML tags
selfclosing_tags = set([ 'br', 'hr', 'nobr', 'ref', 'references' ])
selfclosing_tag_patterns = []
for tag in selfclosing_tags:
    pattern = re.compile(r'<\s*%s\b[^/]*/\s*>' % tag, re.DOTALL | re.IGNORECASE)
    selfclosing_tag_patterns.append(pattern)

#Construct patterns for tags to be ignored
ignored_tags = set([
        'a', 'b', 'big', 'blockquote', 'center', 'cite', 'div', 'em',
        'font', 'h1', 'h2', 'h3', 'h4', 'hiero', 'i', 'kbd', 'nowiki',
        'p', 'plaintext', 's', 'small', 'span', 'strike', 'strong',
        'sub', 'sup', 'tt', 'u', 'var',
])
ignored_tag_patterns = []
for tag in ignored_tags:
    left = re.compile(r'<\s*%s\b[^>]*>' % tag, re.IGNORECASE)
    right = re.compile(r'<\s*/\s*%s>' % tag, re.IGNORECASE)
    ignored_tag_patterns.append((left, right))

#Construct patterns to recognize math and code
placeholder_tags = {'math':'formula', 'code':'codice'}
placeholder_tag_patterns = []
```

```
for tag, repl in placeholder_tags.items():
    pattern = re.compile(r'<\s*%s(\s*| [^>]+?)>.*?<\s*/\s*%s\s*>' % (tag, tag), re.DOTALL | re.IGNORECASE)
    placeholder_tag_patterns.append((pattern, repl))

#HTML comments
comment = re.compile(r'<!--.*?-->', re.DOTALL)

#Wikilinks
wiki_link = re.compile(r'\[\[([^[]*?)(?:\|([^[]*?))?\]\](\w*)')
parametrized_link = re.compile(r'\[\[.*?\]\]')

#External links
externalLink = re.compile(r'\[\w+.*? (.*?)\]')
externalLinkNoAnchor = re.compile(r'\[\w+[&\]]*\]')

#Bold/italic text
bold_italic = re.compile(r"'''''([^']*?)'''''")
bold = re.compile(r"'''(.*?)'''")
italic_quote = re.compile(r"''\"(.*?)\"''")
italic = re.compile(r"''([^']*)''")
quote_quote = re.compile(r'""(.*?)""')

#Spaces
spaces = re.compile(r' {2,}')

#Dots
dots = re.compile(r'\.{4,}')

#Sections
section = re.compile(r'(==+)\s*(.*?)\s*\1')

# Match preformatted lines
preformatted = re.compile(r'^ .*?$', re.MULTILINE)

#Wikilinks
def make_anchor_tag(match):
    '''Recognizes links and returns only their anchor. Example:
     <a href="www.something.org">Link text</a> -> Link text'''
    link = match.group(1)
    colon = link.find(':')
    if colon > 0 and link[:colon] not in namespaces:
        return ''
    trail = match.group(3)
    anchor = match.group(2)
    if not anchor:
        if link[:colon] in namespaces:
            return '' #Don't keep stuff like "category: shellfish"
        anchor = link
    anchor += trail
    return anchor

def clean(text):
    '''Outputs an article in plaintext from its format in the raw xml dump.'''
    # Drop transclusions (template, parser functions)
    # See: http:// www.mediawiki.org/wiki/Help:Templates
    text = dropNested(text, r'{{', r'}}')
    # Drop tables
    text = dropNested(text, r'{\|', r'\|}')

    # Convert wikilinks links to plaintext
    text = wiki_link.sub(make_anchor_tag, text)
    # Drop remaining links
```

```
    text = parametrized_link.sub('', text)

    # Handle external links
    text = externalLink.sub(r'\1', text)
    text = externalLinkNoAnchor.sub('', text)

    #Handle text formatting
    text = bold_italic.sub(r'\1', text)
    text = bold.sub(r'\1', text)
    text = italic_quote.sub(r'&quot;\1&quot;', text)
    text = italic.sub(r'&quot;\1&quot;', text)
    text = quote_quote.sub(r'\1', text)
    text = text.replace("''''", '').replace("'''", '&quot;')

    ################ Process HTML ###############

    # turn into HTML
    text = unescape(text)

    # do it again (&amp;nbsp;)
    text = unescape(text)

    # Collect spans

    matches = []
    # Drop HTML comments
    for m in comment.finditer(text):
            matches.append((m.start(), m.end()))

    # Drop self-closing tags
    for pattern in selfclosing_tag_patterns:
        for m in pattern.finditer(text):
            matches.append((m.start(), m.end()))

    # Drop ignored tags
    for left, right in ignored_tag_patterns:
        for m in left.finditer(text):
            matches.append((m.start(), m.end()))
        for m in right.finditer(text):
            matches.append((m.start(), m.end()))

    # Bulk remove all spans
    text = drop_spans(matches, text)

    # Cannot use dropSpan on these since they may be nested
    # Drop discarded elements
    for pattern in discard_element_patterns:
        text = pattern.sub('', text)

    # Expand placeholders
    for pattern, placeholder in placeholder_tag_patterns:
        index = 1
        for match in pattern.finditer(text):
            text = text.replace(match.group(), '%s_%d' % (placeholder, index))
            index += 1

    ##############################################

    # Drop preformatted
    # This can't be done before since it may remove tags
    text = preformatted.sub('', text)
```

```
# Cleanup text
text = text.replace('\t', ' ')
text = spaces.sub(' ', text)
text = dots.sub('...', text)
text = re.sub(u' (,:\.\)\]»)', r'\1', text)
text = re.sub(u'(\[\(«) ', r'\1', text)
text = re.sub(r'\n\W+?\n', '\n', text) # lines with only punctuations
text = text.replace(',,', ',').replace(',.', '.')

#Handle section headers, residua etc .
page = []
headers = {}
empty_section = False

for line in text.split('\n'):

    if not line:
        continue
    # Handle section titles
    m = section.match(line)
    if m:
        title = m.group(2)
        lev = len(m.group(1))
        if title and title[-1] not in '!?':
            title += '.'
        headers[lev] = title
        # drop previous headers
        for i in headers.keys():
            if i > lev:
                del headers[i]
        empty_section = True
        continue
    # Handle page title
    if line.startswith('++'):
        title = line[2:-2]
        if title:
            if title[-1] not in '!?':
                title += '.'
            page.append(title)
    # handle lists
    elif line[0] in '*#:;':
        continue
    # Drop residuals of lists
    elif line[0] in '{|' or line[-1] in '}':
        continue
    # Drop irrelevant lines
    elif (line[0] == '(' and line[-1] == ')') or line.strip('.-') == '':
        continue
    elif len(headers):
        items = headers.items()
        items.sort()
        for (i, v) in items:
            page.append(v)
        headers.clear()
        page.append(line) # first line
        empty_section = False
    elif not empty_section:
        page.append(line)

text = ''.join(page)

#Remove quote tags.
```

```
    text = text.replace("&quot;", '')

    #Get rid of parentheses, punctuation and the like
    text = re.sub('[^\w\s\d\'\-]','', text)
    return text
```

A.2.6 Application Examples

This section contains code used for the examples of applications of explicit semantic analysis described in section 3.3.

Trend Discovery and Monitoring

This is the script used to generate the results seen in section 3.3.1.

```
# -*- coding: utf-8 -*-
from __future__ import division

import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt
import sys
sys.path.insert(0, '../../')
import codecs
sys.stdout = codecs.getwriter('utf8')(sys.stdout)
sys.stderr = codecs.getwriter('utf8')(sys.stderr)
from collections import Counter
from cunning_linguistics import SemanticAnalyser as SA
import json
import time
import datetime
from shared import moerkeroed#, wine, oldhat, nude

INPUT_FILENAME = 'carlsberg_short.txt'
#INPUT_FILENAME = 'carlsberg_filtered_tweets.txt'
OUTPUT_DIR = 'animation/'
EXT = '.pdf'
SHOW = True
USE_ONE_IN = 1 # e.g. if 5 only every 5th tweet will be used. 1 for all
PROCESS = False #Whether to process data from scratch or read in old data


#INPUT_FILENAME = 'carlsberg_filtered_tweets.txt'
MAX_TWEETS = float('inf')

TOP_N = 10 #Number of best matched concepts to include in analysis
IGNORE = ['Carlsberg Group'] #Concepts to ignore
#If not empty, track only concepts in this list
TRACK = ['Carlsberg Foundation',
         'Carlsberg Polska',
         'Carlsberg Srbija',
         'Old Lions',
         'Copenhagen',
         'Carlsberg Laboratory',
#          'Raidió Teilifís Éireann',
```

```
        'Kim Little']

_display = TOP_N + len(IGNORE)

_now = time.time()

def tweet2epoch(t):
    '''Extracts the time of input tweets creation and returns as datetime.'''
    epoch = time.mktime(time.strptime(t['created_at'],'%a %b %d %H:%M:%S +0000 %Y'))
    return epoch

def epoch2dtindex(epoch, span = 'day'):
    dt = datetime.datetime.fromtimestamp(epoch)
    year, week, weekday = datetime.date.isocalendar(dt)
    day, month = dt.day, dt.month
    if span == 'week':
        return "%s-%02d" % (year, week)
    elif span == 'day':
        return "%s-%02d-%02d" % (year, month, day)

def pad_labels(X, limit = 15):
    '''Limits label length to <limit> characters.'''
    for i in xrange(len(X)):
        if len(X[i]) <= limit:
            continue
        else:
            X[i] = X[i][:limit]+"..."

def process():
    tweets = []
    n_read = 0
    with open(INPUT_FILENAME, 'r') as f:
        for line in f.readlines():
            tweet = json.loads(line)
            n_read += 1
            if not n_read % USE_ONE_IN == 0:
                continue
            if tweet['retweeted'] or not tweet['lang'] == 'en':
                continue
            else:
                tweets.append({'created_at' : tweet2epoch(tweet),
                               'text' : tweet['text']})
                if len(tweets) >= MAX_TWEETS:
                    break
                #
            #
        #

    # Create a semantic analyser
    sa = SA()

    data = {}   #This will map index yyyy_ww to cobined list of top concepts
    counter = 0
    for tweet in tweets:
        concepts = sa.interpret_text(tweet['text'], display_concepts = _display)
        filtered = [c for c in concepts if not c in IGNORE][:TOP_N]
        index = epoch2dtindex(tweet['created_at'])
        # If tracking, include only tracked concepts
        if TRACK:
            filtered = [c for c in filtered if c in TRACK]
        try:
            data[index] += filtered
```

```
            except KeyError:
                data[index] = filtered
            counter += 1
            print "Processed %d of %d tweets..." % (counter, len(tweets))

        #Change data into {index : (sorted topics, their counts)}
        for index, _list in data.iteritems():
            # Most often used concepts in entire period coresponding to index
            d = dict(Counter(_list))
            top_concepts = sorted(d, key = d.get)[::−1][:TOP_N]
            #If not tracking specific concepts, just save top n concepts and counts
            if not TRACK:
                X = sorted(top_concepts)
                Y = [d[x] for x in X]
            #If tracking, use NaN for concepts that didn't occur in top n.
            else:
                X = TRACK
                Y = [d[x] if x in top_concepts else float('nan') for x in TRACK]
            data[index] = (X, Y)


        return data

if __name__ == '__main__':
    if PROCESS:
        data = process()

        with open('processed_data.json', 'w') as f:
            json.dump(data, f, indent = 4)
        #
    else:
        with open('processed_data.json', 'r') as f:
            data = json.load(f)

#===========================================================================
# Plotting ...
#===========================================================================

    #Set pylab params to make plots look similar
    ymin = float('inf')
    ymax = float('-inf')
    for _,Y in data.values():
        if not Y:
            continue
        thismin = min(Y)
        thismax = max(Y)
        if thismin < ymin:
            ymin = thismin
        if thismax > ymax:
            ymax = thismax

    ymin = 0  # Don't truncate
    yair = (ymax − ymin)*0.05

    xair = 0.5
    xmin = 0
    xmax = len(TRACK) if TRACK else TOP_N


    for index, datum in data.iteritems():
        #Enforce axes 'n stuff
        fig = plt.figure(figsize = (3.25,3))
```

```
        ax = fig.add_subplot(111)
        plt.axis((xmin - xair, xmax + xair, ymin - yair, ymax + yair))
        ax.set_autoscale_on(False)

        X, Y = datum
        pad_labels(X, limit = 20)
        plt.xticks([n+0.4 for n in xrange(len(X))], X, rotation = 50,
                   ha='right', fontsize = 8)
        plt.bar(range(len(X)), Y, color = moerkeroed)
        plt.gcf().subplots_adjust(bottom = 0.42, left = 0.3)
        plt.title(index)

        plt.savefig(OUTPUT_DIR+index+EXT, dpi = 600)

        if SHOW:
            plt.show()

        plt.close()


    print "Runtime (m): ", (time.time()-_now)/60
```

Social Media Impact

This is the code used to extract and analyse the data described in section 3.3.2.

```
# -*- coding: utf-8 -*-
'''This script computes the semantic vector corresponding to an input reference
text, loads in tweets contained in files saved in the specified period
(change filelist to whatever files contain your tweets), and then computes
the cosine similarity of each tweet with the reference text and saves the
result.'''
from __future__ import division

import sys
sys.path.insert(0, '../../')
from cunning_linguistics import SemanticAnalyser
import datetime
from dateutil.parser import parse
import json
from matplotlib import pylab as plt
import numpy as np
import pytz
import glob
import time
import multiprocessing


REFERENCE = 'reference_google.txt'
PUBLISHED = datetime.datetime(2015, 06, 17)  #Date ref was published
EARLY = 16  #Number of days to include around pub date
LATE = 42
OUTPUT_FILENAME = 'deep_dreams.txt'
KEYWORD = 'google'
N_JOBS = 8

```

```

#15-06-17
with open(REFERENCE, 'r') as f:
    reference_text = f.read()

now = time.time()

epoch = datetime.datetime(1970,1,1, tzinfo = pytz.utc)
def dt2epoch(dt):
    utc_date = dt.astimezone(pytz.utc)
    delta = utc_date - epoch
    return delta.total_seconds()

def percentof(small, large):
    return str(100*small/large) + "%"

#Get canonical date string
def timeparse(string):
    dt = parse(string)
    (y,m,d) = (str(dt.year), str(dt.month).zfill(2), str(dt.day).zfill(2))
    return "%s-%s-%s" % (y,m,d)

with open(REFERENCE, 'r') as f:
    reference_text = f.read()

#required entries in tweets
ineedthese = ['lang', 'text', 'created_at']

def worker(filename):
    Y = []
    X = []
    #Make an analyser
    sa = SemanticAnalyser()
    reference_vector = sa.interpretation_vector(reference_text)
    with open(filename, 'r') as f:
        tweets = [json.loads(line) for line in f.readlines()]
    for tweet in tweets:
        if not all(stuff in tweet.keys() for stuff in ineedthese):
            continue
        if not tweet['lang'] == 'en':
            continue
        text = tweet['text'].lower()
        if not KEYWORD in text:
            continue
        t = dt2epoch(parse(tweet['created_at']))
        this_vector = sa.interpretation_vector(text)
        similarity = sa.scalar(this_vector, reference_vector)
        if np.isnan(similarity):
            continue
        X.append(t)
        Y.append(similarity)
        #
    print "Processed file: ", filename
    d = {'X' : X, 'Y': Y}
    return d

if __name__ == '__main__':
    filelist = set([])

    for w in np.arange(-EARLY, LATE+1, 1):
        delta = datetime.timedelta(days = w)
        dt = PUBLISHED + delta
```

```
        prefix = dt.strftime("%Y-%m-%d")
        pattern = 'tweets/' + prefix + '-data*'
        filelist.update((glob.glob(pattern)))

    pool = multiprocessing.Pool(processes = N_JOBS)
    jobs = [pool.apply_async(worker, kwds = {'filename' : fn})
            for fn in filelist]
    pool.close()  #run
    pool.join()  #Wait for  remaining jobs

    #Make sure no children died too early
    if not all(job.successful() for job in jobs):
        raise RuntimeError('Some jobs failed.')

    X = []
    Y = []
    for d in [j.get() for j in jobs]:
        X += d['X']
        Y += d['Y']
    inds = np.argsort(X)
    X = [X[i] for i in inds]
    Y = [Y[i] for i in inds]

    with open(OUTPUT_FILENAME, 'w') as f:
        json.dump(X, f)
        f.write('\n')
        json.dump(Y, f)

#     plt.plot(X, Y)
#     plt.show()
```

Bibliography

[1] Yoshua Bengio and Yves Grandvalet. No Unbiased Estimator of the Variance of K-Fold Cross-Validation. 5:1089–1105, 2004.

[2] Avrim Blum, Adam Kalai, and John Langford. Beating the Hold-out: Bounds for {K}-fold and Progressive Cross-Validation. *Proceedings of the 12th Annual Conference on Computational Learning Theory*, (c):203–208, 1999.

[3] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[4] L Breiman. Random forests. *Machine learning*, pages 5–32, 2001.

[5] Cjc Christopher J C Burges. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[6] Yves Alexandre De Montjoye, Jordi Quoidbach, Florent Robic, and Alex Pentland. Predicting personality using novel mobile phone-based metrics. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7812 LNCS:48–55, 2013.

[7] JM John M Digman. Personality structure: Emergence of the five-factor model. *Annual review of psychology*, 41:417–440, 1990.

[8] RA Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 1936.

[9] E Gabrilovich and S Markovitch. Computing Semantic Relatedness Using Wikipedia-based Explicit Semantic Analysis. *IJCAI*, 2007.

[10] Isabelle Guyon. Gene Selection for Cancer Classification. pages 389–422, 2002.

[11] Marti a. Hearst, Susan T. Dumais, Edgar Osuna, John Platt, and Bernhard Schölkopf. Support vector machines. *IEEE Intelligent Systems and their Applications*, 13(4):18–28, 1998.

[12] Thomas K Landauer, Peter W. Foltz, and Darrell Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25(2-3):259–284, 1998.

[13] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. An Introduction to Information Retrieval. *Online*, (c):569, 2009.

[14] Preslav Nakov. Latent semantic analysis of textual data. *Proceedings of the conference on Computer systems and technologies - CompSysTech '00*, pages 5031–5035, 2000.

[15] AD Pietersma. Kernel Learning in Support Vector Machines using Dual-Objective Optimization. *Proceedings of the 23rd ...*, 2011.

[16] G. Salton, a. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[17] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988.

[18] AJ Smola and B Schölkopf. *Learning with kernels*. 1998.

[19] AJ Smola and B Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 2004.

[20] Philip Wolfe. A duality theorem for nonlinear programming. *Quarterly of applied mathematics*, 19(3):239–244, 1961.

[21] PA Zandbergen and SJ Barbeau. Positional accuracy of assisted gps data from high-sensitivity gps-enabled mobile phones. *Journal of Navigation*, 2011.