Imperial College London

Department of Computing

Reality Mining In Twitter

by

Gabriela de Oliveira Penna Tavares

Submitted in partial fulfilment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

September 2012

Abstract

Twitter is arguably among the most popular social networks today, with over 140 million active users. It is characterised by a dual nature, since it can be used as both a social interaction environment and as an information dissemination site. Twitter's popularity has attracted not only individuals but also corporations seeking to promote their brands, and bots, which are computer programs that use Twitter to disseminate information and, in some cases, malicious content.

In this work, we have collected a large dataset of Twitter users and have employed this dataset to study the online behaviour of three different types of accounts: personal, managed and bot-controlled. We perform a statistical analysis of user profiles and create two different Machine Learning algorithms based on tweeting behaviour: a naive Bayes classifier, which classifies Twitter accounts into the aforementioned categories, and a probabilistic prediction model, which aims to predict the time of a user's next tweet.

We obtain some interesting results when comparing the account categories regarding three different properties, namely, inter-tweet delay, tweet frequency on different hours of the day, and tweet frequency on different days of the week, and find very distinct behaviours among the three different account classes. Our classifier performs an effective classification despite using only the first two of the properties studied. Finally, our predictive model generates statistically significant results, although it has so far exhibited a moderate correctness ratio and therefore still leaves room for future improvement.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Aldo Faisal, for his continuous guidance, support and enthusiasm throughout the development of this project.

I would also like to thank Dr. Luke Dickens for his valuable ideas and for helping me with the implementation of the Twitter crawler application.

As well, I would like to thank my beloved family for always believing in me and being by my side, even from far away.

Finally, I would like to thank my professors and my friends at Imperial College, without whom this past year would not have been half as much enlightening, or fun.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Online social networks play a major role in social interaction today [17]: they are an important medium for communication and exchange of information and data between their users. The informal setting of social media encourages users to frequently express their thoughts, opinions, and random, personal details of their lives [29, 34]. Moreover, recent research has shown that managing the impression of one's online presence is increasingly important [1]. Among the large number of online social networks that presently exist, Twitter stands out as a microblogging website in which users broadcast brief text updates, with up to 140 characters, called tweets. Another distinct feature of Twitter is that its user relationship is directed: user A may choose to follow user B and receive their updates, but user B is not required to follow user A back. This means that the flow of information on the Twitter network is directed from the source (author of the tweet) to its subscribers (followers) [12].

Although it is possible for Twitter users to have private accounts, which cannot be accessed by the general public, these constitute only a small portion of cases. Twitter is a public interaction network that contains freely available information about the lives of its millions of users. Previous studies, such as the one found in [23], have shown the main uses of Twitter to be the following: daily chatter, or posting information about one's personal life; conversations, in the form of direct tweets to specific users; information sharing, such as links to other web pages, photos and videos; and news reporting, such as commentary on news and current affairs. These different types of interaction give us a wide range of possibilities for using Twitter data when studying human behaviour.

There are many advantages to using data from Twitter for scientific research: it is free, abundant and relatively easy to obtain. Furthermore, another important characteristic of Twitter content is its timeliness: the messages are so brief that they are intrinsically associated with the moment they were posted, which is represented as a timestamp on the tweet. This means that tweets relate to a very narrow temporal window and thus constitute an extremely up-to-date view of users' information [7]. Although most tweets contain little informational value, the aggregation of millions of these messages can generate important knowledge about the population sample of Twitter users. Many studies have been developed that use data from Twitter to analyse public sentiment and opinion and their relation to social, political and economic measures. Some of these studies are briefly described in chapter 3.

While past research has largely focused on using tweets as a representation of collective behaviour, this project takes the perspective of Reality Mining, a field concerned with the analysis and discovery of patterns in large datasets pertaining to human social behaviour [15, 16], and aims to use Twitter data to study users individually and make predictions about them in real life. The main motivation behind this project is therefore the essential need to understand human behaviour and the human decision-making process. Because the dynamics of many social, technological and economic phenomena are driven by individual human actions, the quantitative understanding of human behaviour is a central question of modern science. In [3], Barabási argues that understanding the mechanisms that govern the timing of various human activities has significant scientific and commercial potential. For instance, models of human behaviour are indispensable for large-scale models of social organisation, such as urban models, the spread of epidemics, the development of

panic and financial market behaviour. Furthermore, human behavioural models are crucial for better resource allocation and pricing plans for telephone companies, to improve inventory and service allocation in retail, and to understand the bursts of ideas and memes emerging in communication and publication patterns.

The field of Computational Social Science [25, 19] is concerned with studying patterns of individual and group behaviours through digital and online data. In [25], Lazer *et al.* claim that the digital traces of our lives can be compiled into comprehensive pictures of human behaviour, with the potential to transform our understanding of our lives, organisations and societies. This data can enable, for instance, investigating the temporal dynamics of human communications, studying the evolution of social networks over time, analysing the spread of pathogens through populations, tracing the spread of political arguments and rumours online, among many other interesting studies. Data from social network websites, in particular, can help us understand the impact of a person's position in the network on their tastes, their moods and even their health. The present work is a contribution to the field of Computational Social Science in that it attempts to understand the behaviour of humans when deciding to post messages on Twitter, a decision that might be associated with many aspects of their real lives, such as their work routines, their friendship network and the process of forming and expressing their opinions.

The purpose of this project is therefore to obtain real-life information about Twitter users based solely on their tweeting patterns. We focus on the use of data that can be easily obtained through a web crawler and does not require parsing the contents of posts or obtaining questionnaire answers from the users, which can be both time-consuming and expensive. Moreover, with a large number of publishers, it is difficult to establish a standard for the presentation of information on Twitter, which can hamper content analysis. Analysing tweeting behaviour, on the other hand, can avoid this shortcoming because it focuses on how the content is sent rather than what the content is [26].

The first step in the development of this project was the collection of data from multiple Twitter users through a web crawler. For this purpose, we created Creepy Crawly, a Twitter crawler which allowed us to retrieve data in an efficient way while conforming to the request limit imposed by the Twitter API. After data collection, we studied tweeting patterns, such as the timestamps of each post and the time interval in between posts, and used this information to classify users into three different groups and to predict when their next tweets would be posted. By analysing users' behaviour on Twitter, we were able to learn about human behavioural patterns without having to look into the tweets' contents.

Since its creation in 2006, Twitter has become increasingly popular, and has reached over 140 million active users [10]. The popularity of Twitter makes it an important tool for marketing and business promotion [8], customer service [9], political campaigning [33] and even fuelling of civil revolutions [37]. As a result, identifying and understanding the behaviour of users behind Twitter accounts can be of great importance: it allows Twitter users, both humans and corporations, to know with whom they are interacting on the social network, and thus plan their interaction accordingly. Although many real life characteristics of users could potentially be extracted from their tweeting behaviour, such as gender, age group, friendship network and even personality, this project focuses on the simple goal of identifying what type of user is controlling a given account.

With the continuing expansion of Twitter, many companies and businesses have created corporate accounts in order to promote their products or services and gain visibility [10]. Furthermore, the website's popularity has attracted a large number of automated programs, known as bots, that access Twitter through its API and are allowed to perform the same activities as an actual person would [12]. Although some of these bots are harmless and merely provide automated information, such as news or blog updates, there are also ill-intentioned bots that spread spam and malicious content. The short tweet size limit, characteristic of Twitter, favours spammers because it requires links to be shortened, making them illegible. Consequently, identifying spammers on Twitter is a relevant issue, and for this purpose we create a naive Bayes classifier based on tweeting patterns to classify Twitter accounts as either managed (corporate), personal, or bot-controlled. This classification can be very helpful in the recognition and filtering of spammers and malicious accounts.

The main contributions of this project are the following: a vast literature survey, comprising past work in Computational Social Science and Reality Mining, and previous research that studies the structure of Twitter or uses Twitter data for a variety of purposes; the development of Creepy Crawly, a web crawler for data collection from Twitter; a statistical analysis of the data collected and a behavioural analysis of Twitter users; the implementation of two Bayes classifiers for Twitter accounts, the first one to distinguish between personal and managed accounts, and the second one to distinguish between personal, managed, and bot-controlled accounts; and finally, the implementation of a predictive model for estimating the time of a user's next tweet.

The remainder of this report is organised as follows. Chapter 2 contains a brief description of concepts and techniques that were relevant to the development of the project. The literature survey in chapter 3 is an analysis of past research related to human behaviour and to the use of online social network data. Chapter 4 describes the application developed for collection of data from Twitter, and chapter 5 contains a statistical analysis of the data that was obtained. Machine Learning methods and the algorithms implemented are explained is chapter 6, while developments and results obtained so far are described in chapter 7. Finally, chapter 8 concludes the report and contains suggestions for future work.

Chapter 2

Methodological Background

In this chapter, we briefly review the theoretical concepts that constitute the building blocks of both the literature survey, presented in chapter 3, and the development of the project itself. In the next sections, we cover some important topics in Probability and Statistics, Machine Learning, and Complex Networks.

2.1 Probability and Statistics

Basic Concepts

The study of Probability and Statistics [27, 13] is concerned with the analysis of random phenomena. We begin by reviewing the concept of a random variable, which numerically describes the outcome of an event occurring in a system subject to variations due to chance. If A is a discrete random variable, the probability that A takes an arbitrary value a is:

$$0 \leq P(A = a) \leq 1 \tag{2.1.1}$$

The probability of alternative values is given by the sum of the individual probabilities:

$$P(A = a \text{ or } A = a') = P(A = a) + P(A = a') \tag{2.1.2}$$

Intuitively, it is easy to conclude that the sum of probabilities of all possible values that A can take on is equal to one:

$$\sum_{\text{all possible a}} P(A = a) = 1 \tag{2.1.3}$$

For two random variables A and B, the joint probability $P(A = a, B = b)$ is the probability that both A has value a and B has value b. We can use joint probabilities to marginalise the probability of a random variable, i.e., to obtain the probability for variable A as the sum of joint probabilities over all possible outcomes of a different variable B:

$$P(A = a) = \sum_{\text{all possible b}} P(A = a, B = b) \tag{2.1.4}$$

The conditional probability $P(A = a|B = b)$ is the probability that A will take on a value a given that we know B has value b. The product rule defines the following property:

$$P(A = a, B = b) = P(A = a)P(B = b|A = a) = P(B = b)P(A = a|B = b) \tag{2.1.5}$$

If A and B are independent variables, then following three properties also apply:

$$P(A = a|B = b) = P(A = a) \tag{2.1.6}$$

$$P(B = b|A = a) = P(B = b) \tag{2.1.7}$$

$$P(A = a, B = b) = P(A = a)P(B = b) \tag{2.1.8}$$

From the product rule defined in 2.1.5, we can derive a central theorem in the study of Probability, called Bayes' theorem:

$$P(A = a|B = b) = \frac{P(A = a)P(B = b|A = a)}{P(B = b)} \propto P(A = a, B = b) \tag{2.1.9}$$

Intuitively, Bayes' theorem can be written as:

$$\text{posterior} \propto \text{likelihood} \times \text{prior} \tag{2.1.10}$$

where the prior, $p(A = a)$, is the initial degree of belief in $A = a$; the likelihood, $P(B = b|A = a)$, is the degree of belief in $B = b$ having accounted for $A = a$; and the posterior, $P(A = a|B = b)$, is the degree of belief in $A = a$ having accounted for $B = b$.

The Bernoulli distribution is a discrete probability distribution which takes value one with probability p and value zero with probability $1 - p$:

$$B(k = 1; p) = p \tag{2.1.11}$$
$$B(k = 0; p) = 1 - p$$

The binomial coefficient $\binom{n}{k}$ can be used to represent the number of ways in which we can choose k elements out of a set of n elements, i.e., it counts the number of possible k-size combinations using those n elements. It can be computed as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \binom{n}{n-k} \tag{2.1.12}$$

The probability mass function (PMF) gives us the probability that a discrete random variable x will be exactly equal to a given value. If x is a discrete random variable defined on a sample space S, then the PMF of x can be defined as:

$$f_x(a) = P(x = a) = P(\{s \in S|x(s) = a\}) \tag{2.1.13}$$

The total probability for all possible values of x must be equal to 1:

$$\sum_{a \in A} f_x(a) = 1 \tag{2.1.14}$$

For continuous random variables, the probability density function (PDF) describes the relative likelihood for a variable x to take on a given value, and is defined within range $[-\infty, \infty]$:

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{2.1.15}$$

The cumulative distribution function (CDF) describes the probability that a continuous random variable x with probability distribution $p(x)$ will be found at a value less than or equal to a:

$$P(x \leq a) = \int_{-\infty}^{a} p(x)dx \tag{2.1.16}$$

Intuitively, the CDF is the "area so far" function of the PDF.

The expectation of a random variable x under a probability distribution $p(x)$ is defined as:

$$\langle f(x) \rangle = \int_{-\infty}^{\infty} p(x)f(x)dx \tag{2.1.17}$$

Finally, the variance of a random variable x under a probability distribution is defined as:

$$\text{Var}(x) = \langle (x - \langle x \rangle)^2 \rangle = \langle x^2 \rangle - \langle x \rangle^2 \tag{2.1.18}$$

Poisson Distribution

The Poisson distribution, named after mathematician Siméon Denis Poisson, is a discrete distribution that describes the probability that a given number of events will occur in a fixed interval of time or space. This distribution assumes that events will occur with a known average rate λ and that the occurrence of each particular event is independent of the time elapsed since the last occurrence. Therefore, for a given interval, the Poisson distribution predicts the degree of spread around the known average rate of occurrence.

A random variable x has a Poisson distribution with average rate parameter λ if its PMF is given by:

$$f_x(k;\lambda) = P(x = k) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{2.1.19}$$

where the parameter λ is equal to both the expected value and the variance of x.

Power Law Distribution

The power law distribution expresses the probability of a continuous variable that varies as a power of that variable's value. This distribution is characterised by a fat tail that represents large but rare values that the random variable might take. A random variable is said to follow a power law if its probability density function has the form:

$$p(x) \propto L(x)x^{-\alpha} \tag{2.1.20}$$

where $\alpha > 1$ and $L(x)$ is a slowly varying function such that $\lim_{x\to\infty} L(cx)/L(x) = 1$. The form of function $L(x)$ controls the shape and extent of the distribution's tail. On a log-log plot, in which both axes are displayed in log scale, a power law distribution will have the form of a straight line with negative slope given by $-\alpha$.

Lognormal Distribution

The lognormal distribution is the continuous probability distribution of a random variable x, whose logarithm is normally distributed. This means that a variable x is lognormally distributed if $y = log(x)$ has a normal distribution. The probability density function of a lognormal distribution is given by:

$$p(x|\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x-\mu)^2}{2\sigma^2}}, \quad x > 0 \tag{2.1.21}$$

where μ and σ are the mean and the standard deviation, respectively, of the associated normal distribution. A lognormal random variable x is therefore defined as:

$$x = e^{\mu+\sigma z} \tag{2.1.22}$$

where z is a standard normal variable. The mean m and the variance v of x can be obtained as functions of μ and σ:

$$m = e^{(\mu+\sigma^2)/2} \tag{2.1.23}$$

$$v = e^{2\mu+\sigma^2}(e^{\sigma^2} - 1) \tag{2.1.24}$$

The lognormal distribution is applicable when the quantity of interest must be positive, since $log(x)$ exists only when x is positive. A random variable can be modelled as lognormal if it can be thought of as the multiplicative product of many independent, positive random variables.

Dirichlet Distribution

The multinomial distribution is a discrete distribution which gives the probability of choosing a collection of m items from a set of n items with repetitions, where the probabilities of choosing each item are given by $q_1, ..., q_n$. These probabilities are the parameters of the multinomial distribution.

The Dirichlet distribution is a family of continuous, multivariate probability distributions parametrized by a vector α of positive reals. It is the conjugate prior of the parameters of the multinomial distribution. The probability density function of the Dirichlet distribution for variables $q = (q_1, ..., q_n)$ with parameters $\alpha = (\alpha_1, ..., \alpha_n)$ is defined as:

$$p(q) = \text{Dirichlet}(q, \alpha) = \frac{1}{B(\alpha)} \prod_{i=1}^{n} q_i^{\alpha_i - 1} \tag{2.1.25}$$

where $q_1, ..., q_n \geq 0$, $\sum_{i=1}^{n} q_i = 1$ and $\alpha_1, ..., \alpha_n > 0$. The parameters α_i can be seen as prior observation counts for events governed by probabilities q_i. The normalisation constant $B(\alpha)$ is the multinomial beta function, which can be expressed in terms of the gamma function:

$$B(\alpha) = \frac{\prod_{i=1}^{n} \Gamma(\alpha_i)}{\Gamma(\prod_{i=1}^{n} \alpha_i)} \tag{2.1.26}$$

Intuitively, the probability density function of the Dirichlet distribution returns the belief that the probabilities of n rival events are q_i given that each event has been observed $\alpha_i - 1$ times.

Statistical Measures

We now review some statistical measures that can be used in the evaluation of statistical models. The first of these measures is the coefficient of determination R^2, which is a goodness-of-fit evaluation for regression and statistical models of prediction. It determines how well future outcomes are likely to be predicted by the predictive model. In a dataset with values y_i (observed values), and for which the model has predicted values f_i (predicted values), the variability of the dataset can be measured by the residual sum of squares, SS_{err}, and the total sum of squares, SS_{tot}:

$$SS_{err} = \sum_i (y_i - f_i)^2 \tag{2.1.27}$$

$$SS_{tot} = \sum_i (y_i - \overline{y})^2 \tag{2.1.28}$$

where the mean of y is given by

$$\overline{y} = \frac{1}{n} \sum_{i}^{n} y_i \tag{2.1.29}$$

The coefficient of determination is defined as:

$$R^2 = 1 - \frac{S_{err}}{S_{tot}} \tag{2.1.30}$$

The coefficient of determination is therefore a measure of the portion of variance in the data that is explained by the model, since the second term in its definition compares the unexplained variance (variance of the model's errors) with the total variance of the data. The value of R^2 ranges from 0 to 1, where a value of 1 means that the model perfectly predicts the data.

The Pearson Correlation Coefficient (PCC) is a value between -1 and 1 that measures the correlation, or linear dependence, between two variables X and Y. A value close or equal to 1 means that a linear equation can describe the relationship between X and Y quite well, with all data points lying close to a line for which Y increases as X decreases, while a value close or equal to -1 means that the data points lie close to a line for which Y decreases as X increases. A value close 0 implies that there is no linear correlation between the two variables.

The PCC is defined as the covariance of the two variables divided by the product of their standard deviations. For a population the PCC $\rho_{X,Y}$ between X and Y is given by:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{2.1.31}$$

where σ_X and σ_Y are the standard deviations of X and Y, respectively, and μ_X and μ_Y are the means of X and Y, respectively.

The PCC is a symmetric measure, i.e., $\rho_{X,Y} = \rho_{Y,X}$. It can be interpreted geometrically as the cosine of the angle between both possible regression lines, $y = f_x(x)$ and $x = f_y(y)$.

Granger causality is a statistical hypothesis test for determining whether one time series is useful in forecasting another, revealing causality between the two. Consider three time series X_1, X_2 and X_3, and two autoregressive linear models for X_1:

$$X_1(t) = \sum_{j=1}^{m} A_j X_1(t-j) + B_j X_2(t-j) + C_j X_3(t-j) + \varepsilon_{ABC}(t) \tag{2.1.32}$$

$$X_1(t) = \sum_{j=1}^{m} A_j X_1(t-j) + C_j X_3(t-j) + \varepsilon_{AC}(t) \tag{2.1.33}$$

If we find that the variance of ε_{ABC} is significantly less than the variance of ε_{AC}, this means that X_2 is relevant to the prediction of values of X_1, and we say that X_2 Granger-causes X_1. Granger causality therefore assesses the influence of one variable (X_2) on another variable (X_1) over and above the influence of the rest of the system (X_3).

In a test for Granger causality, we start by doing a regression of ΔY on lagged values of ΔY, where ΔY is the first difference of variable Y, equal to Y minus its one-period-prior value. Once the set of significant lagged values for ΔY is found, we add lagged values of ΔX to the regression and check whether any of these lagged values are retained in the regression according to t-tests and F-tests. Then we can conclude that there is no Granger causality between X and Y if and only if no lagged values of ΔX have been retained in the regression.

2.2 Machine Learning

Machine Learning [4] is the study of algorithms that can learn from data and automatically improve with experience. It is a subject closely related to Statistics and to Pattern Recognition, which is the inspection of complex patterns and probability distributions in large amounts of data. In this section, we revise a few topics in Machine Learning that were relevant to the development of the algorithms described in chapter 6.

Probabilistic Prediction

In deterministic prediction, the outcome of a variable is determined exactly, which means that the predictive model has complete confidence in the value predicted. In contrast, probabilistic prediction generates a "degree of belief" in each possible outcome of a variable. We can use probabilities to predict the outcome of a random variable A based on the probability distribution obtained from a training set. If $p(A)$ is the PDF obtained for A, we can simply infer that the outcome of A will be given by that distribution. To evaluate the prediction, a statistical measure such as the coefficient of determination can be used to compare the distribution obtained from the training set to the actual outcome of the variable.

In order to generate a deterministic prediction for the outcome of A, we can assume the value of A will be given by:

$$\text{outcome}(A) = \arg\max_{v} \{p(A=v)\} \tag{2.2.1}$$

where v represents all possible outcomes for A.

Naive Bayes Classifier

A naive Bayes classifier is a simple classifier model based on Bayes' theorem. It is called naive because it uses naive or strong independence assumptions: it assumes that the presence, or absence, of a particular feature of a class is unrelated to the presence, or absence, of any other feature of that class. Naive Bayes classifiers are trained in a supervised learning setting, in which the classes of all samples in the training dataset are known a priori.

The probability model for such a classifier can be defined as follows:

$$p(C|F_1, ..., F_n) = \frac{p(C)p(F_1, ..., F_n|C)}{p(F_1, ..., F_n)} \tag{2.2.2}$$

where C is a class and $F_1, ..., F_n$ are n features. The denominator in the equation above is not strictly relevant: since all features F_i are independent and their values are known, the value of $p(F_1, ..., F_n)$ will be effectively constant. Therefore, we can rewrite equation 2.2.2 as:

$$p(C|F_1, ..., F_n) = p(C)p(F_1, ..., F_n|C) = p(C, F_1, ..., F_n) \tag{2.2.3}$$

It can be shown that

$$p(C, F_1, ..., F_n) \propto p(C)p(F_1|C)p(F_2|C, F_1)...p(F_n|C, F_1, F_2, ..., F_{n-1}) \tag{2.2.4}$$

and, since each feature is conditionally independent of every other feature, we have:

$$p(C, F_1, ..., F_n) \propto p(C) \prod_{i=1}^{n} p(F_i|C) \tag{2.2.5}$$

Finally, the conditional distribution over class C is given by:

$$p(C|F_1, ..., F_n) = \frac{1}{Z} p(C) \prod_{i=1}^{n} p(F_i|C) \tag{2.2.6}$$

where Z is a scaling factor which is constant as long as the values of each feature variable are known. $p(C)$ is the class prior, and $p(F_i|C)$ is the probability of feature F_i given that the sample is from class C; these probabilities can be obtained from the training set. In order to classify a new sample, we can simply pick the class that is most probable, using the maximum a posteriori (MAP) decision rule:

$$\text{class}(f_1, ..., f_n) = \arg\max_c \left\{ p(C = c) \prod_{i=1}^{n} p(F_i = f_i|C = c) \right\} \tag{2.2.7}$$

EM Algorithm

The Expectation-Maximisation (EM) Algorithm is an iterative method used in Statistics to determine the maximum likelihood of parameters in a certain model. The algorithm's name is due to the two main steps it comprises: in the E step, we use the current estimated values for the parameters to compute the expectation of the log-likelihood; in the M step, we obtain a new estimate for the parameters by maximising the expected log-likelihood obtained in the E step.

Given a statistical model with observable variables X, discrete latent (hidden) variables Z and unknown parameters θ, the complete-data likelihood function is the probability of obtaining both observable and latent variables given the parameters, and can be defined as follows:

$$L(\theta; X, Z) = p(X, Z|\theta) \tag{2.2.8}$$

The maximum likelihood estimation (MLE) is the extent to which the parameters can explain the observed data, and is given by the marginal likelihood of the observed variables:

$$L(\theta; X) = p(X|\theta) = \sum_{Z} p(X, Z|\theta) \tag{2.2.9}$$

We assume that optimising $p(X|\theta)$ is difficult, but that optimising the complete-data likelihood function $p(X, Z|\theta)$ is easier.

For N observations of X, the joint probability of X and Z is:

$$p(X, Z) = \prod_{i=1}^{N} p(x_i|z_i)p(z_i) \tag{2.2.10}$$

Thus, the complete-data log-likelihood function is given by:

$$\ln\{p(X, Z|\theta)\} = \ln\{\prod_{i=1}^{N} p(x_i|z_i, \theta)p(z_i|\theta)\} = \sum_{i=1}^{N}\{\ln(p(x_i|z_i, \theta)) + \ln(p(z_i|\theta)\} \tag{2.2.11}$$

In order to obtain the parameters θ for the model, we first initialise these parameters, define a prior over the latent variables Z, $p(Z)$, and define the likelihood $p(X|Z)$. Because we do not have the complete data set $\{X, Z\}$, we cannot directly maximise the complete-data log likelihood function. Our knowledge of the latent variables Z is given by the posterior distribution $p(Z|X, \theta)$; therefore, we can use this posterior to obtain the expected value of the complete-data log likelihood.

In the E step, we use the current estimates for θ to obtain the posterior distribution of the latent variables, $p(Z|X, \theta)$. We then take the expectation of the complete-data log likelihood function, $\ln\{p(X, Z|\theta)\}$, with respect to the posterior distribution of Z. This expectation will be a function of θ:

$$Q(\theta|\theta^{(t)}) = E_{p(Z|X,\theta^{(t)})}[\ln\{p(X, Z|\theta)\}] = \sum_{Z} p(Z|X, \theta^{(t)}) \ln(p(X, Z|\theta)) \tag{2.2.12}$$

In the M step, we maximise the expectation of the joint probability obtained in the E step, with respect to the parameters θ, keeping the posterior statistics fixed:

$$\theta^{(t+1)} = \arg\max_{\theta} Q(\theta|\theta^{(t)}) \tag{2.2.13}$$

2.3 Complex Networks

The study of complex networks [18, 5] is concerned with the dynamics of large systems where the behaviour of individual elements is relatively simple and easy to understand, while the behaviour of the network as a whole is not. We can model such networks as graphs in which the vertices are the elements of the system and the edges represent the interactions between these elements. The applications of this model include, for instance, genetic networks, the nervous system, social networks, electric power grids, and the World Wide Web. Representing the complexity of a system as a network can be very helpful since it allows us to analyse the network's topology, providing insight into the organisational principles of that system. As an online social network, Twitter can be studied as a complex network in which users are represented as vertices and the 'follow' relationships between users are represented as directed edges.

Basic Concepts

In order to analyse and classify complex networks, we need to study their topology, i.e., we need to measure some properties of the network's connectivity structure that will allow us to understand its behaviour. An important measure is the average node degree, which is the average number of nodes that are connected to one specific node. Related to the average node degree is the node degree distribution, which is a sequence $\{k_i\}$ for $1 \leq i \leq n$, where n is the total of nodes, and where each k_i is the degree of node i. The degree distribution can also be seen as a histogram showing the proportion of nodes that have a specific degree. The shortest path length between two nodes i and

j is the minimum amount of edges forming a path that connects i to j; if there is no path linking these two nodes, then the value for this measure is set to infinity. Using the notion of shortest path length, we can define the network diameter as the maximum degree of separation between all pairs of nodes in the network, and the average distance as the average shortest path between all pairs of nodes. The local clustering coefficient for one particular node is the fraction of pairs of neighbours of that node that are connected by an edge, while the global clustering coefficient is the average of local clustering coefficients over the entire network.

Random Networks

A random network is the most basic kind of network, and it can be defined as a graph that has a uniform distribution of randomly assigned edges. For every pair of nodes i and j in such a network, the probability of finding an edge connecting them is p. A random network is fully determined by its number of nodes, n, and its average node degree, z. Using that information we can obtain the number of edges, $nz/2$, and the connection probability, p, which is the probability of finding an edge between two given nodes:

$$p = \frac{nz/2}{n(n-1)/2} = \frac{z}{n-1} \tag{2.3.1}$$

If D is the diameter of the random network, we have that z^2 is the number of next nearest neighbours of a node, and that $z^D \approx n$, since any given node has approximately z neighbours. Therefore, the network diameter D is proportional to $\log n$. It can also be shown that the average distance L is proportional to $\log(n)/\log(z)$. For a random network, the global clustering coefficient C is equal to the connection probability p, i.e., it's the probability that any two vertices are connected by an edge, and thus, for large random networks, C tends to zero as $n \to \infty$.

The generating model for a random network is the Erdős-Rènyi model, in which a link is drawn between each pair of nodes with equal probability and independently of other edges. We begin with n disconnected nodes and then add $nz/2$ edges at random between pairs of nodes, which means that the network will have average node degree z. For any network, if x_k is the number of nodes that have degree k, then the node degree distribution of the network is given by $p_k = x_k/n$. For Erdős-Rènyi graphs, we have:

$$p_k = \binom{n-1}{k} p^k (1-p)^{n-1-k} \tag{2.3.2}$$

where $p = z/(n-1)$ is the connection probability. When $n \to \infty$, the node degree distribution p_k is given by a Poisson distribution with mean z:

$$p_k \approx e^{-pn} \frac{(pn)^k}{k!} \approx e^{-z} \frac{z^k}{k!} \tag{2.3.3}$$

since $z \approx pn$. We can generate an Erdős-Rènyi graph that satisfies any desired degree distribution given by p_k, where $\sum_{k=0}^{\infty} p_k = 1$, through the following steps: we start with n disconnected nodes and assign k_i stubs (ends of edges) to each node i; we then iteratively choose pairs of stubs and connect them, until there are no disconnected stubs left. In consequence, the fraction of nodes with degree k will tend to p_k as $N \to \infty$.

The average number of neighbours of a node can be written in terms of the degree distribution:

$$z = \langle k \rangle = \sum_k k p_k \tag{2.3.4}$$

The conditional probability that a node i has degree k_i given that i and j are connected is:

$$P(k_i | i \leftrightarrow j) = \frac{P(k_i, i \leftrightarrow j)}{P(i \leftrightarrow j)} = \frac{P(i \leftrightarrow j | k_i) P(k_i)}{P(i \leftrightarrow j)} \tag{2.3.5}$$

where $P(k_i) = p_{k_i}$. The probability that nodes i and j are connected is $P(i \leftrightarrow j) = \langle k \rangle/(n-1)$, and $P(i \leftrightarrow j | k_i) = k_i/(n-1)$.

Small-world Networks

Random networks are not easily observed in the real world, since most networks display some sort of structure. On the other hand, small-world networks appear to be very common. A small-world network has three important features: it is sparse, it has a large global clustering coefficient, and it has a small average distance. Being sparse means that the number of existing edges is only a small fraction of the possible edges among the network's nodes. A large global clustering coefficient means that if a node i is connected to nodes j and k, then there is a higher probability that j and k are also connected to each other. Finally, a small average distance translates into a relatively small path between any two nodes in the network, when compared to the size of the network. Small-world networks frequently have hub nodes, which are connected to many different nodes, whereas Erdős-Rènyi random networks rarely have hubs. Hub nodes are responsible for collapsing the diameter of a small-world network.

We can use the Watts-Strogatz method to construct small-world networks. We start with a regular ring lattice with degree z; then, for each edge in the lattice, we remove one of its ends with probability p and reconnect it to a randomly selected node. The average degree is still z, but now different nodes can have different degrees. This method can preserve the large clustering coefficient of the original lattice, while decreasing the average distance at the same time. For $p \approx 0.1$, we obtain a small-world network as desired.

Most real networks are in fact growing networks. We can create a growing network by starting with a small graph, consisting of two doubly connected nodes. At each moment in time, starting at $t = 3$, we add a new node and connect it at random to one of the existing nodes. Therefore, at time t, we have exactly t nodes and t edges in the network. The average degree $\langle k \rangle$ is then defined as a function of t:

$$\langle k \rangle(t) = \frac{2t}{t} = 2 \tag{2.3.6}$$

Scale-free Networks

Scale-free networks are networks whose degree distribution, i.e., the distribution of edges per node, fits a power law:

$$P(k) \propto k^{-\gamma}, \gamma > 1 \tag{2.3.7}$$

The degree distribution is therefore fat-tailed and goes to zero slowly. In scale-free networks, the probability of a node being highly connected is higher than in a random graph. While scale-free networks are robust and exhibit high tolerance to random perturbations, they are also very sensitive to attacks on highly connected nodes. The failure of a hub node causes the network to break into isolated clusters, while the failure of random nodes affects mostly small-degree nodes and does not cause any major loss of connectivity. Such networks are called scale-free because no specific feature value stands out in the distribution structure. If we rescale by taking $k \to sk$, then the distribution remains the same: $P(k) \propto (sk)^{-\gamma} \propto k^{-\gamma}$.

Many real world networks satisfy a power law, such as the WWW documents, the citation network, and the network of actors with edges between actors who have casted in the same movie. A scale-free network can be built by the preferential growth algorithm proposed by Barabási and Albert: we modify the random growth method previously described by giving preference to connection to vertices with higher degree. In this algorithm, the probability of choosing a node for the insertion of a new link is proportional to the current degree of that node. Therefore, heavily linked nodes (hubs) will quickly accumulate even more links, while nodes with only a few links are unlikely to be chosen as the destination for a new link. This method generates a network that is scale-free and has low average length between nodes.

The Twitter Network

In [23], Java *et al.* present evidence that the Twitter network is scale-free and exhibits a small world feature. At the time of their study, they found that the clustering coefficient in Twitter was quite high at 0.106 while the network diameter was only 6, which indicates that Twitter, in the same way as many other social networks, is a small-world network. They also found that the network presents a high degree correlation and high reciprocity, indicating a large number of mutual acquaintances in the graph. This can be explained by the fact that new users often join the network after being invited by friends, and that new friends are added to the network by browsing through user profiles and adding other known acquaintances. Moreover, Java *et al.* found that both in-degree and out-degree distributions of the Twitter network follow a power law with slope close to -2.4, which is similar to the value found for the Web as a whole and the blogosphere.

Chapter 3

Research Background

We now present a review of past research papers that were relevant to the conception and development of our project. In this literature survey, we cover five different areas of research: Reality Mining, Computational Social Science, classification of Twitter accounts, measuring influence in social networks, sentiment analysis using Twitter data, and other miscellaneous applications of social network data.

3.1 Reality Mining

Reality mining is a field related to data mining that uses machine-sensed environmental data to study human behaviour. The term 'reality mining' was used by a research group in the MIT Media Lab when conducting a study with mobile phone data from 100 phones during the period of one academic year. This data includes call logs, Bluetooth devices in proximity, cell tower IDs, application usage and phone status, and was analysed in different ways, giving rise to several studies and publications.

In [16], Eagle *et al.* compare the mobile phone data with standard self-report survey data, focusing on the relationships between dyads of subjects. Three analyses were conducted: the relationship between self-report and behavioural data; whether there were behaviours identified in the mobile phone data that were characteristic of real life friendship; and the relationship between behavioural data and individual satisfaction. In order to analyse the dyadic variables, the authors used the nonparametric multiple regression quadratic assignment procedure (MRQAP), which is a standard technique for social network data.

When studying the relationship between self-report and behavioural data, the authors checked for recency and salience biases in recall of physical proximity. Recency means that memories are biased toward recent events, while salience means memories are biased toward more vivid events. In their data, recency is represented by the quantity of interactions in a fixed period preceding the survey, and salience by whether the other individual is a friend or non friend. Subjects were asked about their proximity with other subjects, and the answers were compared with average daily proximity based on Bluetooth scans. It was found that survey responses were biased in favour of recent behaviour, and that subjects recall of information about their interactions begins to degrade after approximately one week.

The authors obtained the average hour-by-hour levels of proximity between dyads of symmetric friends, asymmetric friends and non friends, which are shown in figure 3.1. To check for behaviours that were characteristic of friendship, they clustered the data according to proximity, location and time. Using factor analysis, they found that the two factors that captured the most variance in these variables were representative of the behaviour between work colleagues and between friends outside the work environment. With these factors, they were able to accurately predict self-reported friendships.

The authors then built two models to predict social integration in work groups, one based on self-reported friendship, and the other one based on the dyadic weights associated with the factor analysis previously performed. The predictors used in both models were number of friendships,

average proximity to friends while at work, and phone communication with friends while at work. They found that the inferred friendship network produced substantially identical results to the self-report model, and concluded that having friends at work predicted satisfaction with the work group, whereas calling friends while at work was associated with lack of satisfaction.

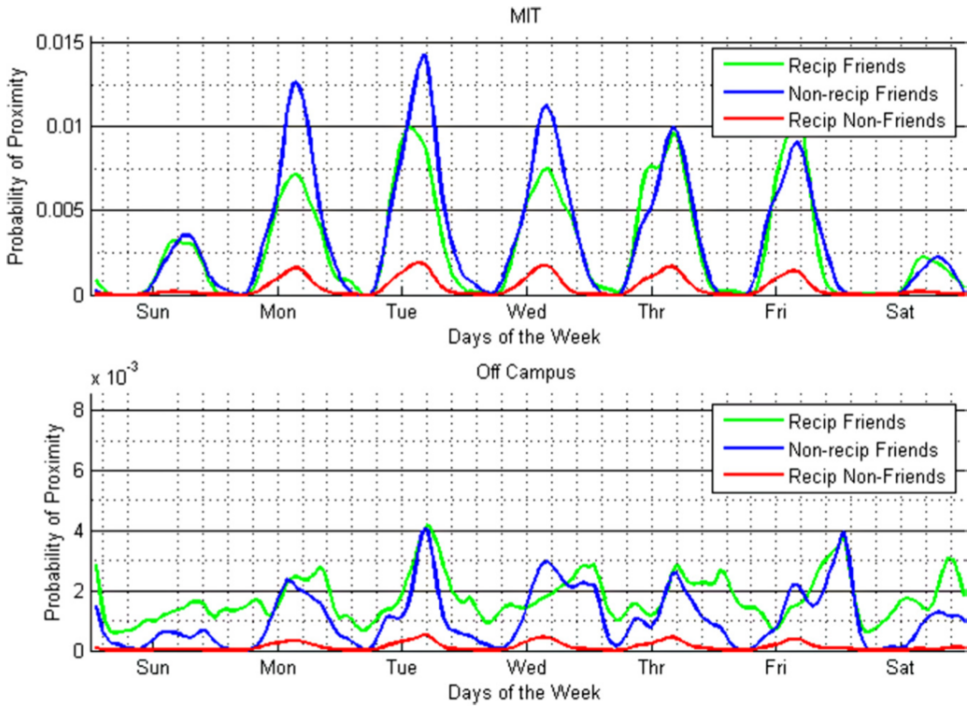


Figure 3.1: Probability of proximity between subjects, both at work and off campus for symmetric friends, asymmetric friends and non friends. This probability is calculated for each hour in the week and is generally much higher for friends than non friends. We can also see that symmetric friends spend more time together off campus in the evenings. Source: [16]

Another important study in Reality Mining using the same mobile phone data can be found in [15]. In this work, Eagle *et al.* obtain the principal components of the dataset in order to find structure and behavioural patterns. The principal components, referred to as 'eigenbehaviours', are the eigenvectors of the covariance matrix of the data, and were computed for each individual studied. A linear combination of an individual's eigenbehaviours was used to accurately reconstruct the behaviour from each day in the data, and also to predict the individual's subsequent behaviour. Furthermore, eigenbehaviours were used to characterise the behaviour of communities within the social network and to identify affiliations and relationships between individuals.

The data for each individual was represented as a two-dimensional array, where the dimensions are the days of data collection and the hours of each day. Each element of this array is one of the following labels corresponding to behaviour: 'home', 'elsewhere', 'work', 'no signal', and 'off'. The primary eigenbehaviours are a subset of vectors that best characterise the distribution of behaviours, and define the individual's low-dimensional behaviour space. The authors argue that an individual's primary eigenbehaviours represent a space upon which all vectors corresponding to individual days can be projected with different levels of accuracy. Figure 3.2 shows the projection of each day onto spaces created using an increasing number of eigenbehaviours.

In order to compute the eigenbehaviours of an individual, the authors obtained an average behaviour and then used it to obtain the deviation of each day from the mean. They performed principal components analysis on these vectors to generate a set of orthonormal vectors that corresponds to the eigenvectors of the covariance matrix with highest eigenvalues. They then used the resulting eigenbehaviours and weights generated from the first 12 hours of an individual's day to predict the subsequent 12 hours.

In addition, eigenbehaviours were computed for three communities in the social network, and

the data for each individual was projected into the behavioural space of each community. Individual affiliation was then measured by the Euclidian distance between the individual and the principal components of each community's behaviour space. Community eigenbehaviours were also used to determine the similarity of members, by identifying how accurately a member's behaviour could be approximated by the community's eigenbehaviours, and to determine how much each individual fit in a community, by measuring the distance between the individual's projection onto the community's behaviour space and the individual's original behaviour.

While these two studies in Reality Mining use mobile phone data, they could easily be modified and extended with the use of online social network data. They do require, however, the collection of real life data about the subjects, such as their real friendship networks and satisfaction level in the work environment.

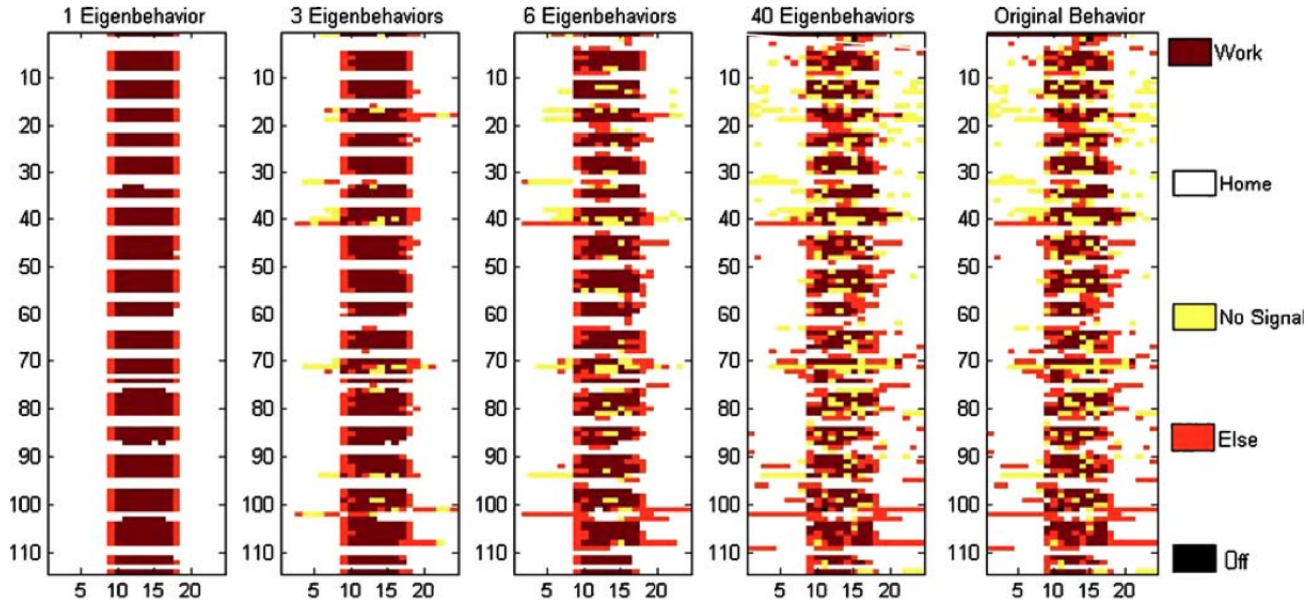


Figure 3.2: Behaviour approximation of a period of 115 days using a varying number of eigenbehaviours. As the number of eigenbehaviours increases, a better approximation is obtained. Source: [15]

3.2 Computational Social Science and Human Behaviour

As explained in chapter 1, Computational Social Science is a relatively new field of research concerned with the study of human behaviour, both individually and in groups, based on the analysis of digital and online data. Many interesting studies have arisen in this field. For instance, in [3], Barabási studies the communication between e-mail users in order to understand how humans prioritise their activities. While previous models of human activity had been largely based on Poisson processes, in which the time interval between two consecutive actions by the same individual follows an exponential distribution, the model supported by Barabási predicts that inter-event times are better approximated by a heavy-tailed or Pareto distribution. This model allows for very long periods of inactivity that separate bursts of intense activity, a behaviour characteristic of the timing of many human actions.

The author studied an e-mail dataset capturing the sender, recipient, time and size of each e-mail. He found that the distribution of time differences between consecutive e-mails sent by a user was best approximated by $P(\tau) \approx \tau^{-\alpha}$, with $\alpha \simeq 1$, which indicates a bursty, non-Poisson character: during a single session a user would send several emails in quick succession, followed by long periods of no e-mail activity. Figure 3.3 shows some of the findings in the dataset analysed. Barabási then continued to show that the bursty nature of human dynamics is a consequence of a

queuing process in decision making: when presented with multiple tasks, an individual chooses a task based on some priority parameter, and thus the waiting time of the various tasks will follow a Pareto distribution.

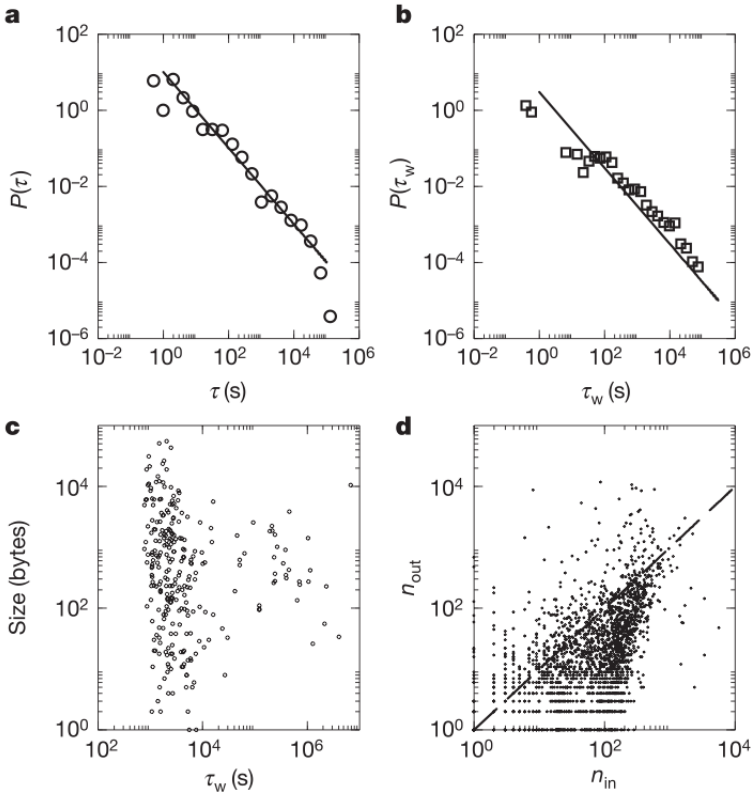


Figure 3.3: Heavy-tailed activity patterns in e-mail communications. (a) shows the distribution of time intervals between consecutive e-mails sent by a single user over three months. (b) shows the distribution of the time taken by the same user to reply to a received message. (c) shows a scatter plot of the waiting time and the size of each e-mail responded to by the user, indicating that file size and response time do not correlate. (d) shows a scatter plot of the number of e-mails sent and received by 3,188 users during three months. Source: [3]

Barabási explains that most human-initiated events require an individual to assess and prioritise different activities. When presented with a list of tasks, an agent assigns a priority parameter x to each task, which allows for a comparison between the urgency of different tasks. In most human activities, the individual executes the highest-priority item on the list first, and this selection mechanism is the probable source of the fat tails observed in human-initiated processes. The author built a priority list model to account for this behaviour, and found that both the analytical solution and the numerical simulations performed were in agreement with the empirical data, resulting in a power law tail with exponent $\alpha = 1$. He explains that the tail distribution is independent of the distribution from which the agent chooses the priorities, and this is the reason why, despite the diversity of human actions, most decision-driven processes develop a heavy tail.

The author proposes that the inter-event time distribution observed for e-mail communication is rooted in the uneven waiting times experienced by different tasks. In order to test this hypothesis, he determined the time a user would take to reply to each received message, and found that the waiting time distribution was best approximated by a heavy-tail with exponent $\alpha_w = 1$, which would explain the observed bursty e-mail activity patterns.

Barabási concludes that the observed fat-tailed activity distributions can be explained by the hypothesis that humans execute their tasks based on some perceived priority and set up queues that generate uneven waiting time distributions for different tasks. He argues that the fact that a wide range of human activity patterns follow non-Poisson statistics suggests that the observed bursty character reflects some fundamental feature of human dynamics. Finally, the author speculates

that Internet traffic must be driven by two separate processes: a heavy-tailed size distribution of the files sent by users, and the human decision-driven timing of various Internet activities that individuals engage in.

In [30], Oliveira and Barabási extended the study found in [3]. They studied Darwin's and Einstein's patterns of correspondence, shown in figure 3.4, and compared them with today's e-mail exchanges. For both scientists, they computed the response times, or the time interval between the date a letter was received and the date that a reply was sent. They found that the probability that a letter would be replied to in τ days was well approximated by a power law, $P(\tau) \approx \tau^{-\alpha}$, and thus followed the same scaling laws as current e-mail communication. The authors conclude that the famous scientists' late responses or resumed correspondences are a part of the universal scaling law, representing a fundamental pattern of human dynamics.

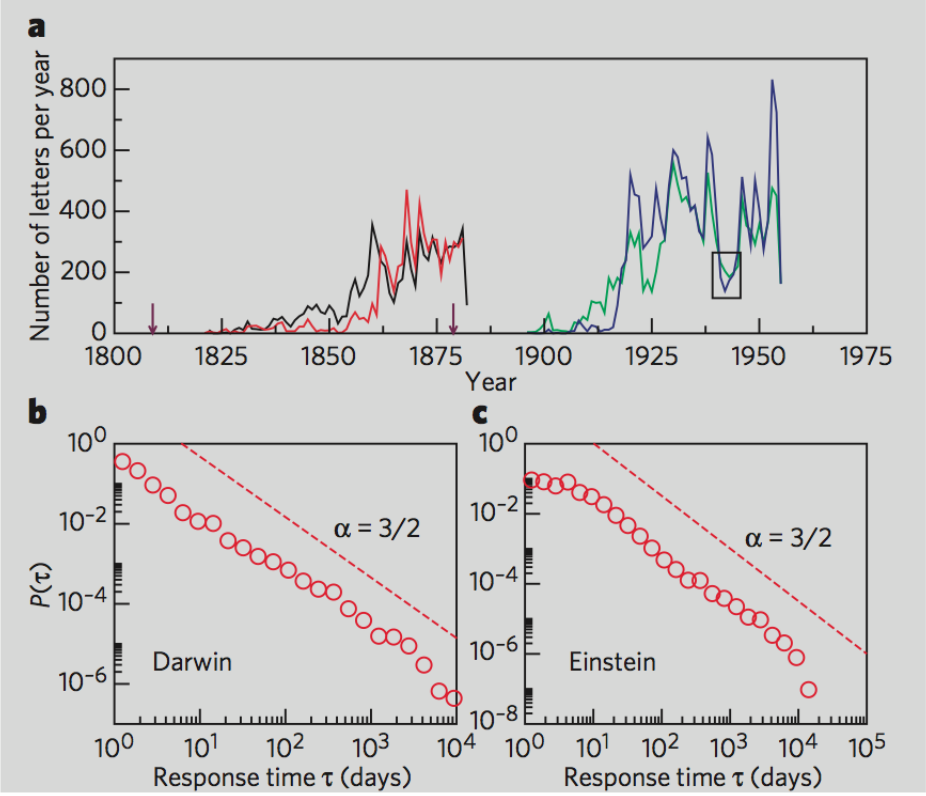


Figure 3.4: Correspondence patterns of Darwin and Einstein. (a) shows the historical record of letters sent (Darwin, red; Einstein, green) and received (Darwin, black; Einstein, blue) each year. (b) and (c) show the distribution of response times to letters by Darwin and Einstein, respectively. Source: [30]

Assuming that Darwin and Einstein prioritised correspondence in need of response, and that therefore the rate of letters arriving was higher than the rate of letters being answered, the authors obtained a value of $\alpha = 3/2$, which is different from the $\alpha = 1$ obtained for e-mail communications. The different scaling exponents describing the scientists' letter responses and current e-mail show that these two communication patterns belong to different universality classes. Oliveira and Barabási argue that this fact provides evidence for a new class of phenomena in human dynamics.

In [14], Dezsö *et al.* present yet another study on the bursty nature of human dynamics. In this work, they aim to understand the topology and dynamical features of rapidly changing networks such as the most visited portions of the World Wide Web, which change within hours through the rapid addition and removal of documents and links. They investigate the visitation patterns of a major news portal and show that the timing of the browsing process is non-Poisson, a fact that has a significant impact on the visitation history of web documents.

The authors used the log files of the news portal to collect the visitation pattern of each visitor during the period of one day. They explain that web portal networks consist of a stable

skeleton, representing the main structure of the portal, and a large number of news items that are documents only temporarily linked to the skeleton. While documents belonging to the skeleton present an approximately constant daily visitation pattern, which causes the cumulative number of accesses to increase linearly in time, the visitation of news documents is the highest right after their release and decreases in time, which causes a saturation in their cumulative visitation after several days. These distinct visitation patterns were used to distinguish, in an automated fashion, the pages belonging to the skeleton from the news documents.

Dezsö *et al.* find that, in general, the number of visits $n(t)$ to a news document follows a dampened periodic pattern, with the majority of visits happening within the first day, decaying to only 7% on the second day, then reaching a small but constant visitation beyond four days. These measurements indicate that the visitation does not decay exponentially, but its asymptotic behaviour is best approximated by a power law $n(t) \sim t^{-\beta}$, such that, while the bulk of visits takes place at small t, a considerable number of visits are recorded well beyond the document's release time. This visitation pattern can be explained by the uneven browsing patterns of individual users, shown in figure 3.5: for each user, numerous frequent downloads are followed by long periods of inactivity. This bursting, non-Poisson pattern, characteristic of human behaviour, causes the interval between consecutive requests by the same user to follow a power-law distribution.

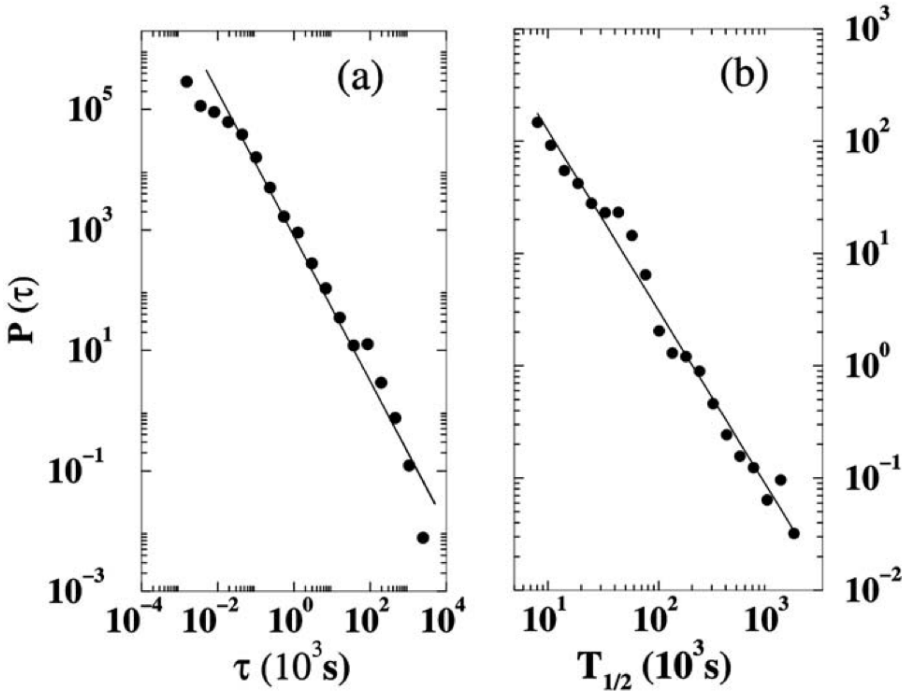


Figure 3.5: (a) shows the distribution of time intervals between two consecutive visits of users, with a slope of $\alpha = 1.2$. (b) shows the half-time distribution for individual news items, following a power law with exponent ≈ -1.5. Source: [14]

The authors then describe a model for the number of visits of a news document, $n(t)$, based on the browsing pattern of individual users. They found that the waiting time distribution is a power law where the exponents follow a Gaussian distribution with average exponent $\alpha_0 = 1.14$, which is very close to the value found for the power law of users' visits, $\alpha = 1.2$. Furthermore, they obtained the following relation between exponent α, related to the decay in news visitation, and β, related to the visitation pattern of individual users: $\beta = \alpha - 1$. They conclude their study by characterising the interest in news documents by their half-time, i.e., the time frame during which half of all visitors that eventually access it have visited. They found that the overall half-time distribution also follows a power law, which indicates that while most news items have a short lifetime, a few continue to be accessed well beyond their initial release.

The authors conclude that the decay laws identified in the study are likely generic, as they

do not depend on the content of pages, but are determined mainly by the users' visitation and browsing patterns. They conjecture that the same power law inter-event time behaviour is likely to be observed in the visitation of individual users to commercial sites, and might be also applicable to biological systems.

3.3 Identifying and Classifying Behaviour on Twitter

Identifying and classifying specific types of users on Twitter can be useful for a variety of reasons, from focusing advertisement and political campaigns, to filtering spam and malicious accounts. With a large occurrence of spamming and political campaigning on Twitter, recent research has focused on methods for identifying certain types of behaviour that are characteristic of spammers or propagandists.

In [26], Lumezanu *et al.* aim to understand how Twitter is used to spread propaganda. They study the Twitter behaviour of propagandists, users who consistently express the same opinion or ideology, and focus their work on hyperadvocates, who show a consistent lack of impartiality in their messages. Four publishing patterns were used in the study of hyperadvocates: sending high volumes of tweets in short periods of time, retweeting more than publishing original content, quickly retweeting, and colluding with other users to post similar content at the same time.

The data used consisted of tweets from two discussion groups: the 2010 Nevada Senate race, identified by the hashtag #nvsen, and the 2011 debt ceiling debate, identified by the hashtag #debtceiling. Figure 3.6 shows the volume of tweets and retweets in both communities. In order to classify users as either biased (hyperadvocates) or neutral, they clustered users with similar ideologies, who retweeted exclusively each other's messages. They began by randomly assigning each user to one of two clusters, corresponding to one set of related opinions. The algorithm was seeded with users whose political views were known, and then each user was reassigned to the cluster in which they had the most associated users, where the association consists of retweeting of each other's messages. If at least a certain fraction of the connections of a user were to users in the same cluster, then the user was considered a hyperadvocate; otherwise, they were considered neutral.

In the #nvsen group, the authors identified two distinct tweeting behaviours among hyperadvocates: high daily volumes of tweets and a high daily fraction of retweets. The same behaviours were not observed in the #debtceiling group. Behaviours assumed to be characteristic of hyperadvocates were determined as follows: a high volume of tweets per day was identified when a user published more tweets in one day than a predefined threshold θ, and the value of θ varied for each day; for identifying high fraction of retweets, the authors defined a user's repeater score as the ratio of the number of retweets to the total number of tweets; quick retweeting was characterised by short reaction time, defined as the difference between the time of a retweet and the time of the original post; and collusion was quantified by the number of users in the same community that sent duplicate or near-duplicate messages very close in time.

The authors conclude that the behaviours studied can help amplify the effect on hyperadvocacy on Twitter and that their presence or absence depends on the community being analysed. Although they make some interesting observations in the paper, Lumezanu *et al.* assume that the behaviours studied are characteristic of propagandists rather than extracting this information from the data structure. For this reason, we cannot conclude whether these behaviours are indeed the most representative attributes of propagandists.

Many organisations currently make use of Twitter accounts to promote their products and services. For them, a large number of followers is an advantage since it can be interpreted as popularity or credibility. With this in mind, some companies purchase fake followers, which are actually Twitter bots generated in large quantities by software programs.

In [10], Calzolari describes an algorithm to distinguish between Twitter accounts controlled by humans and those controlled by robots. Based on criteria selected by the author, this algorithm assigns "human behaviour" points and "bot behaviour" points to each user, then uses the points to classify them. The author picked 39 Twitter accounts owned by companies among the accounts

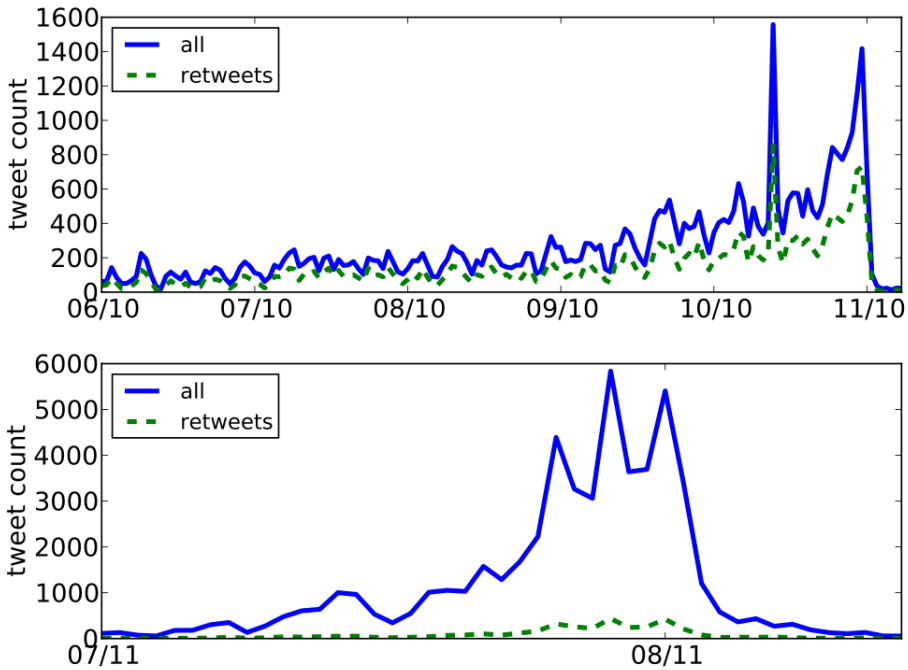


Figure 3.6: Number of tweets and retweets published each day in the #nvsen (top) and #debtceiling (bottom) communities. Source: [26]

with largest number of followers. For each of these companies, 10,000 randomly selected followers were analysed.

Some of the characteristics used by the algorithm to assign "human points" were: the profile contained a biography, the user had at least 30 followers, the user had used an iPhone or Android to log in to Twitter, at least one post by the user had been retweeted by others, among others. The characteristic associated with "bot behaviour" was that the user used only APIs. For each "human" characteristic that a user did not possess, a "bot point" was assigned, and conversely for "bot" characteristics. Users with "uncertain" behaviour, i.e. whose behaviour did not feature enough characteristics to identify them as either human or bot, and users with protected Twitter accounts, were not taken into account.

Calzolari found a large number of users with "bot behaviour" following certain companies, despite the fact that the algorithm used very conservative parameters when assigning "human" and "bot" points. The author argues that the remarkably different numbers of potential bots among the followers of different companies confirm that the algorithm works well. However, since the criteria that differentiates between "human" and "bot" behaviour were determined a priori by the author, there is no guarantee that they are actually representative of each behaviour. Furthermore, other than the fact that different companies obtained different results, nothing else confirms or validates the results generated by the algorithm.

In [2], Balasubramaniyan *et al.* develop a system to distinguish between legitimate users and malicious entities on Twitter, using both the behaviour and the true social network structure of each user. They begin by determining the users' true social networks (TSN) based on '@' mentions and retweets, then apply a PageRank-like algorithm to these networks in order to measure reputation values. They use the reputation values and behavioural traits to identify malicious entities, who are characterised by disseminating poor quality information (and therefore obtain low PageRank values) in an aggressive manner.

The authors collected data from users in different categories, such as science, sports, entertainment, etc., by using specific keywords and seed users who were related to each category. They also manually picked 10 users who engaged in malicious activities and added them to the dataset. The TSN's were constructed as follows: for each user, their friends were sorted based on who they mentioned the most and the 30 most mentioned friends were picked. For each of these friends, the

20 most mentioned friends were picked, and so on, until they had a 3-hop true social network for each user. For all users in the dataset and all users in their TSN's, the authors collected details including name, screen name, friend and follower lists, and the last 200 tweets. In order to identify malicious accounts, they continuously ran a script to check if Twitter had suspended any accounts in their dataset, and found that 222 accounts were eventually suspended. These accounts were then used to test the effectiveness of their classification approach.

To demonstrate the difficulties in detecting malicious accounts, they began by testing two simple mechanisms. The first one is measuring the tweet frequency of users (number of tweets per day), seeing as malicious entities would potentially tweet more aggressively than legitimate ones. However, they found no threshold to distinguish between the sets of all users and the set of malicious users, since there were some legitimate users at each tweet frequency. The second approach to identify malicious entities was checking if a large fraction of people that they mentioned did not interact among themselves. This was evaluated using the clustering coefficient (CC) of each user. Although legitimate users had much higher CC than malicious entities, they found that CC values were well spread across all values, meaning that malicious entities can appear as well connected as legitimate users.

The PageRank algorithm was implemented on top of the TSN, with the aim of ranking users based on their ability to either engage other users or provide meaningful information. They used a weighted PageRank model where the link from user A to user B was weighted based on the number of times A had either mentioned or retweeted B. Therefore users who were mentioned or retweeted by well ranked users, would become well ranked themselves. Since malicious entities try to get noticed, they produce large quantities of information and yet do not obtain high PageRank (PR^{TSN}) values, thus a combination of PR^{TSN} with behavioural characteristics would be a good indication of whether a user is malicious.

The authors characterised bad or aggressive behaviour as a combination of tweeting frequently, providing links to products and mentioning random users. Users with low PR^{TSN} would not be considered bad as long as their behaviour was good, and users with poor behaviour traits would not be considered bad as long as they had high PR^{TSN}. Four user categories were then defined: users with high PR^{TSN} and good behaviour were celebrities who had a significant number of users interacting with them; entities such as TweetMeme, who tweeted links many times a day, could afford to behave aggressively since they were also popular across a wide user base, having a high PR^{TSN}; legitimate users had moderate to low PR^{TSN} and good behaviour; lastly, malicious accounts were characterised by low PR^{TSN} and bad behaviour. In addition to these criteria, the authors flagged down users who had a high number of outside mentions as malicious entities. Finally, they observed that some malicious entities were harder to identify, since they had amassed a large social network and were not aggressive in posting links or tweeting frequently. However, these entities had many aggressively malicious accounts among their initial friends, and could therefore be identified based on this criterion.

The combined algorithm was implemented as follows. They began by computing the PR^{TSN}, which was the principle eigenvector of the social network transition matrix. Moreover, they calculated the link ratio LR (number of links over number of tweets) and the tweet frequency TF (number of tweets over time between last and first tweet) for each user. The combined rank for user i was given by:

$$CR_i = \frac{PR_i^{TSN}}{LR_i \times TF_i} \tag{3.3.1}$$

where the denominator $LR_i \times TF_i$ represents the number of tweets per day that have links, indicating how aggressively the user is trying to disseminate their information. Users with CR below a certain threshold were trying to propagate information aggressively but their information was deemed worthless by most users; they were therefore classified as malicious users. In order to catch more sophisticated malicious entities, the authors calculated a corrected combined rank, CR', using the number of initial friends who were identified as malicious based on CR. Lastly, they considered users who mentioned a large number of users outside their social network. Figure

3.7 shows a plot of user behaviour against PR^{TSN} for all accounts and shows the ones that were identified as spammers.

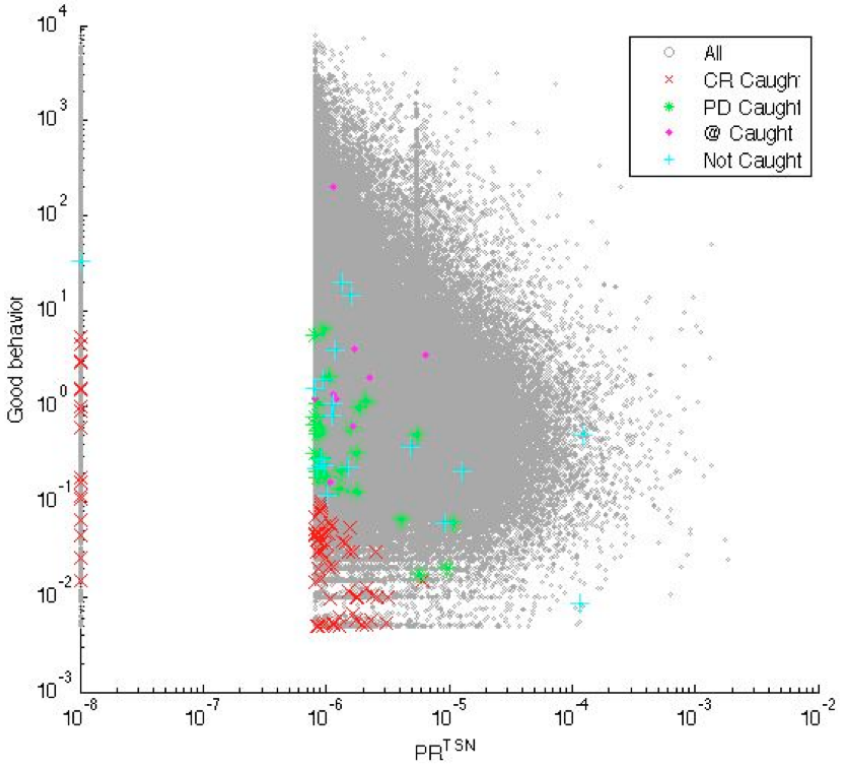


Figure 3.7: Plot showing user behaviour against PR^{TSN}. Different markers show the suspended accounts caught by different mechanisms: combined rank (CR), temporal pulldown (PD) and mentions (@). Source: [2]

The authors set the classification threshold such that 10% of users in the dataset were classified as potentially malicious. They found that all 10 seed malicious accounts were in the bottom 10%, along with 181 of the 222 accounts that were suspended by Twitter. They evaluated the effectiveness of their system by comparing it to two commercially available systems for grading Twitter users. While their algorithm classified all 10 seed malicious accounts correctly, the majority of these accounts were given high grades by both commercial systems, which confirms the validity of their system. Their algorithm has the advantages of using a measure of influence, given by the PageRank algorithm, in order to evaluate the legitimacy of users, and not requiring parsing the tweets' contents.

Another example of Twitter account classification can be found in [12]. Chu *et al.* observe the differences between Twitter accounts controlled by humans, bots, and cyborgs, which refer to either bot-assisted humans or human-assisted bots and interweave characteristics of both humans and bots. The authors study tweeting behaviour, tweet content and account properties in order to characterise the automation feature of Twitter accounts, then use this information to build a classifier for the three account categories, with the aim of assisting Twitter in managing the online community and helping humans recognise who they are tweeting with.

The authors collected data from over 500,000 Twitter users, then created a training dataset and a test dataset containing known samples of humans, bots and cyborgs. This was achieved by randomly choosing different samples and manually checking their logs and homepages in order to classify them. Some of the criteria used for this classification were tweet contents, user profile, number of friends and followers and the content of web pages pointed by posted URLs. Users were labeled as humans if there was evidence of original and intelligent content, and as bots if there was excessive automation in tweeting, abundant presence of spam or malicious URLs, aggressive following behaviour and lack of original or intelligent content. Users were labeled as cyborgs if there was evidence of both human and bot participation.

After analysing the collected data, the authors concluded that the cyborg category was characterised by a high tweet count, which can be attributed to the combination of both automatic and human behaviours. They also found that humans are more active during the regular workdays and less active during the weekend, while bots have roughly the same activity level every day of the week and cyborgs are most active on Mondays, then slowly decrease their activity during the week. Figure 3.8 shows the proportion of tweets posted by each class, per day of the week and per hour of the day. They further noticed that humans tend to tweet manually, bots are more likely to use auto piloted tools, and cyborgs display both behaviours. Finally, they observed that bots had the highest external URL ratios (the number of external URLs included in tweets over the number of tweets posted by an account), while the human URL ratio was the lowest.

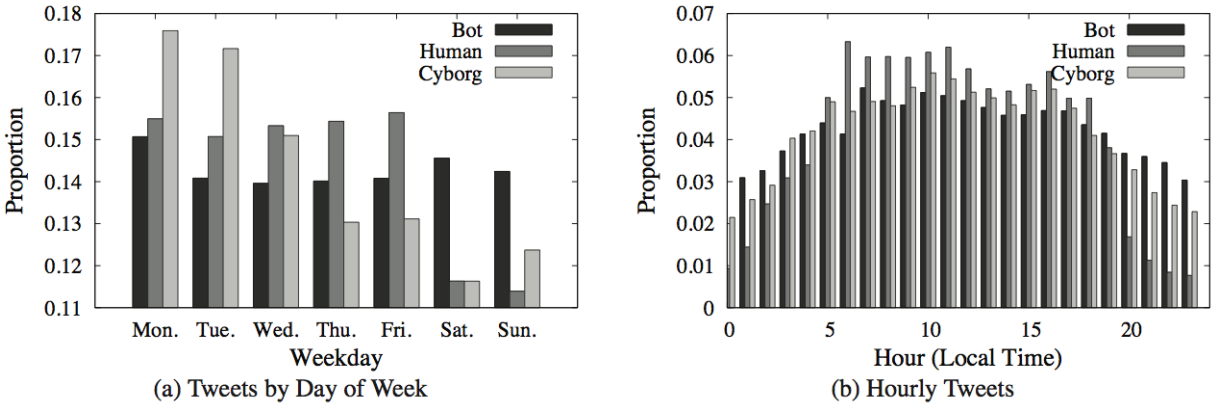


Figure 3.8: Proportion of tweets posted by each class, per week day and per hour. Source: [12]

The automated classification system developed by the authors consists of four components: an entropy component, which uses tweeting interval as a measure of behaviour complexity and detects the periodic and regular timing that is an indicator of automation; a machine-learning component, which uses tweet content to check whether text patterns contain spam or not; an account properties component, which employs account properties to detect potential bot or human characteristics; and a decision maker, based on Linear Discriminant Analysis (LDA), which uses a linear combination of the features generated by the first three components in order to classify the Twitter accounts.

The entropy component of the classifier detected periodic or regular timing of the messages posted by a Twitter user. A low entropy for inter-tweet delays indicates periodic behaviour, which is a sign of automation, whereas a high entropy indicates irregularity, which is a sign of human participation. The machine-learning component used the content of tweets to detect spam, which is usually a sign of automation. The authors used an implementation of a Bayesian classifier, CRM114, which decided that a message belonged to class spam if the probability of the class given the message, $P(C = spam|M)$, was over a certain threshold. Messages were represented as feature vectors, where each feature is one or more words and is assumed to be conditionally independent. The account properties component evaluated important properties of Twitter profiles, based on the data analysis previously performed: URL ratio, tweeting device makeup and followers to friends ratio. Finally, the decision maker used the features identified by the other three components to determine whether a user was a human, a bot, or a cyborg. A multiclass discriminant model was used to identify the most relevant features of the three classes. For each class, a classification function was created, and then the training dataset was used to decide feature weights in each function. Each sample was classified into the class with the highest classification score.

Chu *et al.* conclude that their classification system was able to accurately differentiate humans from bots: over 94% of humans and over 93% of bots were classified correctly. Distinguishing humans and bots from cyborgs, however, was more challenging and caused the system to generate slightly worse results, with only 82.8% of cyborgs being classified correctly. The authors also state that the classification system developed is resistant to possible evasion attempts made by bots, since the URL ratio and tweeting device makeup features are very hard for bots to evade. Despite

attaining a high correctness rate, the system has the limitation of heavily relying on processing the contents of tweets in order to identify them as spam, which can be an expensive process.

3.4 Measuring Influence

As a social network, Twitter allows for plenty of interaction among its users, and this setting generates relationships of influence between users who interact. Direct links between users determine the flow of information in the social network; on Twitter, whenever a user posts a tweet or retweet, the contents of these posts can be seen by the user's followers, therefore influencing them somehow.

IIn [11], Cha *et al.* study and compare three measures of influence on Twitter, namely in-degree, retweets and mentions, and examine how the three types of influential users perform in spreading popular news topics. Moreover, they investigate how a user's influence changes by topic and over time, and pinpoint behaviours that can make users gain influence over short periods of time.

Among the different types of influence, a user's in-degree, or their number of followers, is the most straight-forward. Regarding the other two types, while retweets are focused on the content of the original post, mentions are focused on the user being replied to. In order to compare user influence, the authors used one rank set for each type of influence, then used the relative order of the ranks as a measure of difference. To quantify how a user's rank varied across different types of influence, they used Spearman's rank correlation coefficient as a measure of the strength of the association between two rank sets.

After ranking the users, the authors found that there was little overlap between influence measures, as shown in figure 3.9. This indicates that each measure indeed captures a different type of influence. The top users in the rank showed strong correlation between their retweet influence and their mention influence, but in-degree was not related to the other measures. The authors thus argue that the most connected users are not necessarily the most influential when engaging their audience in conversations or having their messages spread.

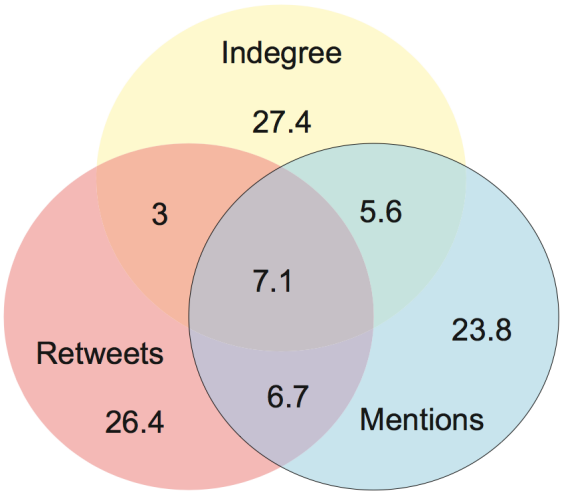


Figure 3.9: Venn diagram of the top 100 influential users across different measures. Numbers are normalised so that the whole diagram sums up to 100. Source: [11]

To investigate whether a user's influence varies by topic, the authors picked three popular topics and identified a set of keywords related to each topic, then searched for messages including those keywords. The measure of influence for a given topic was then computed as the count of retweets and mentions on that topic. It was found that the most influential users ranked consistently high amongst different topics. In oder to track the popularity of the most influential users over time, they counted the number of retweets and mentions of these users every 15 days over a period of 8 months, then computed the probability that a random tweet posted during a 15 day period would be a retweet or mention of each user. They found that organisation accounts were the most retweeted, while evangelists increased their influence by engaging users in conversation and getting

mentions. Finally, the authors computed the same probability for ordinary users with the aim of understanding what behaviours could make these users influential, and found that users who limit their tweets to a single topic had the largest increase in their influence scores.

The authors conclude that, although a user's in-degree represents their popularity, it is not related other notions of influence, such as engaging audience. Furthermore, users must keep great personal involvement in order to gain and maintain influence. The paper provides an interesting comparison of the different ways in which a user can influence the Twitter community around them.

We now look into some theoretical work about influence in a social community. In [32], Pan *et al.* describe a simple influence model, which is used to explain the influence amongst interacting agents in a social system. The term influence here means that the state of an agent A at time t is somewhat determined by the state of all agents in the system at time $t-1$. The authors also define a generalisation of the influence model, the dynamical influence model, which differs from the simple model in that its influence matrix changes over time. The paper also provides a few examples of applications of the influence model.

The simple influence model was defined as follows: in a system with C agents, or entities, each entity c is associated with a certain state h_t^c at time t. This state can be any of the elements in the finite set $\{1, ..., S\}$. This set can be the same for every entity or each entity can have its own finite set of states. The state of an entity is not observable, but it is possible to observe the signal O_t^c emitted by entity c at time t, which has a probability conditioned on the entity's state, $p(O_t^c|h_t^c)$. This probability function can be defined as a multinomial, for the discrete case, or as a gaussian, for the continuous case.

The influence between entities is the conditional dependence between an entity's state at time t and all other entities' states at time $t-1$:

$$p(h_t^{c'}|h_{t-1}^1, ..., h_{t-1}^C) \tag{3.4.1}$$

This probability was defined as follows:

$$p(h_t^{c'}|h_{t-1}^1, ..., h_{t-1}^C) = \sum_{c \in \{1,...,C\}} R_{c',c} \times p(h_t^{c'}|h_{t-1}^c) \tag{3.4.2}$$

where $R_{c',c}$ is the tie strength between c' and c (how much entity c influences entity c')and $p(h_t^{c'}|h_{t-1}^c)$ is the conditional probability between c' and c. $R_{c',c}$ is an element of matrix R, which is a $C \times C$ row stochastic matrix called the influence matrix. The conditional probability can be modelled by C^2 transition matrices of size $S \times S$, i.e., for each entity c, there would be C transition matrices $M^{c,c'}$ of size $S \times S$ defining the transitions between all possible states. However, a more economic way of modelling the conditional probability is to define two $S \times S$ matrices E^c and F^c for each entity c: while $E^c = M^{c,c}$ defines the transitions for entity c on itself, $F^c = M^{c,c'}$ defines the influence of c over any other entity c' (assuming that the influence of an entity is the same over all other entities, except itself).

In conclusion, the influence model described is a generative model with parameters R, $E^{1:C}$, $F^{1:C}$ and emission probabilities $p(O_t^c|h_t^c)$. These parameters can be learned by machine learning algorithms from the observations $O_{1:T}^1, ..., O_{1:T}^C$. The number of parameters grows quadratically with the number of entities C and with the number of states S, which makes the model more scalable and also reduces the risk of overfitting.

The authors also define the dynamical influence model, which is a generalisation of the influence model wherein the influence matrix R changes over time. This modification makes sense for social systems such as group discussion sessions and negotiations, where the influence between subjects will fluctuate. They define a finite set of different influence matrices, $\{R(1), ..., R(J)\}$, each representing an influence dynamical pattern between the entities. J is a hyper parameter that defines how many different influence matrices exist. The parameter $r_t \in \{1, ..., J\}$ specifies which influence matrix is active at time t. The conditional probability for entity c''s state is now defined as:

$$p(h_t^{c'}|h_{t-1}^1, ..., h_{t-1}^C) = \sum_{c \in \{1,...,C\}} R(r_t)_{c',c} \times p(h_t^{c'}|h_{t-1}^c) \tag{3.4.3}$$

To make sure that the influence matrix will change gradually, they define a prior for r_t:

$$p(r_{t+1}|r_t) \sim multi(V_{r_t,1}, ..., V_{r_t,J}) \tag{3.4.4}$$

where V is a parameter matrix constrained by hyper parameter $p^V > 0$. The prior for V is defined as:

$$(V_{r_t,1}, ..., V_{r_t,J}) \sim \text{Dirichlet}(10^0, 10^0, ..., 10^{p^V}, ..., 10^0) \tag{3.4.5}$$

These priors are such that if p^V is very large, r_{t-1} and r_t will be there same, and if p^V is very small, r_{t-1} will randomly switch to any value in $\{1, ..., J\}$ with equal probability. The authors do not justify the choice for a finite set of influence matrices. This raises the question of what results would be obtained if a continuously changing influence matrix was used instead.

The likelihood function for the model is defined as:

$$L(O_{1:T}^{1:C}, h_{1:T}^{1:C}, r_{1:T}|E^{1:C}, F^{1:C}, R(1:J), V) \tag{3.4.6}$$

$$= \prod_{t=2}^{T}\{p(r_t|r_{t-1}) \times \prod_{c=1}^{C}[p(O_t^c|h_t^c) \times p(h_t^c|h_{t-1}^{(1,...,C)}, r_t)]\} \times \prod_{c=1}^{C} p(O_1^c|h_1^c)p(h_1^c)p(r_1) \tag{3.4.7}$$

As in the simple influence model, the system parameters and latent variables for the dynamical influence model can be learned from observations through an inference process. The authors describe in the paper the steps for a variational E-M algorithm that can achieve this. The paper is then concluded with the presentation of various applications of the models described: a toy example with two binary time series, where the influence model is used to find structural changes in network dynamics, modelling of inter-personal influence and interaction on turn taking during group discussions in different settings, and a study of the flu spreading dynamics.

The simple influence model presented by Pan *et al.* is a direct application of the Hidden Markov Model (HMM), where the actual states are not visible but it is possible to observe outputs that are dependent on the states. The influence model is a special case because it involves multiple entities, with each having its own set of possible states, and thus the addition of an influence matrix to weigh the multiple transition probabilities. In the particular case of the dynamical influence model, there is also the modification that the influence matrix changes at each time step.

3.5 Sentiment Analysis with Twitter Data

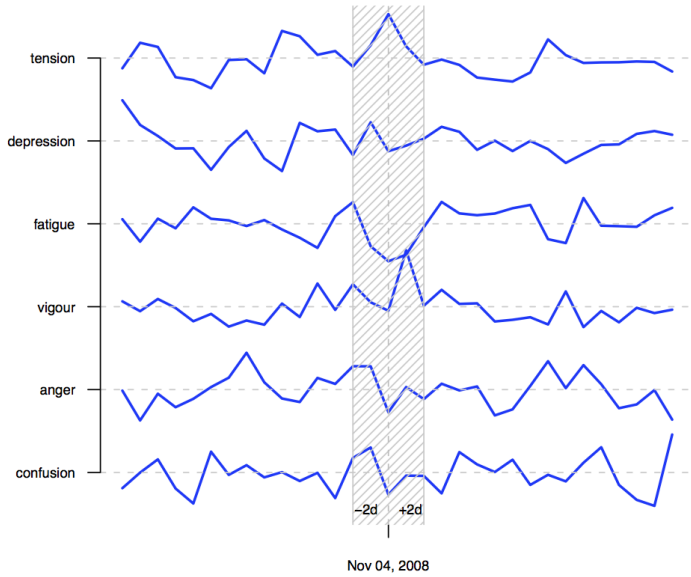
The purpose of sentiment analysis is to determine implicit mood or opinion contained in text according to the words and expressions used. Twitter data is ideal for sentiment analysis since it comprises messages posted by millions of different users every day and is fairly simple to obtain.

Social, political and economical events have a significant and immediate effect on public mood, and tweets often contain information about the mood of their authors, which means that a large collection of tweets published over a given time period can be used as a measure of public mood and indicate changes in its state. The work found in [7], published when Twitter was still in its early stages, is one of the first to perform sentiment analysis of Twitter messages. Bollen *et al.* explore how public mood patterns obtained from Twitter data relate to fluctuations in social and economic indicators. They performed sentiment analysis on tweets to measure the public mood's state using a six-dimensional psychometric instrument, then studied how this state relates to macroscopic socio-economic events such as drops in the stock market, rises in the oil prices and the outcome of a political election.

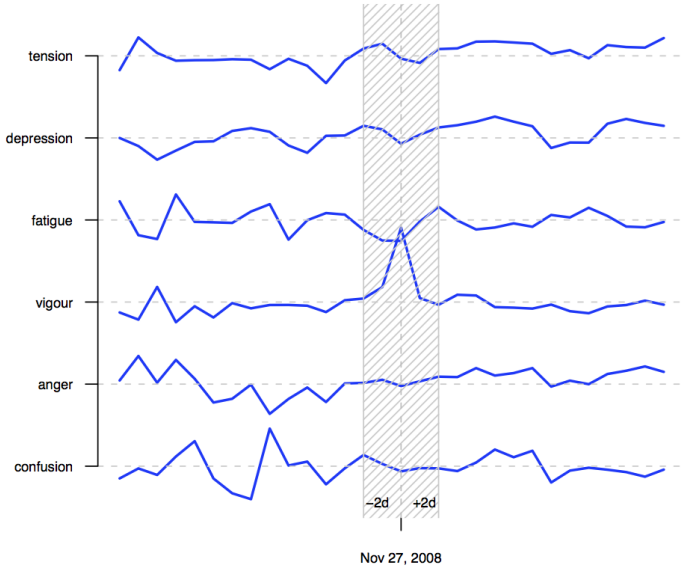
The authors manually selected events including major fluctuations in gas prices and stock market indices, international and US-based political events and natural disasters. Sentiment analysis of the Twitter data was performed using an extended version of the Profile Of Mood States psychometric instrument, POMS-ex, which measures six dimensions of mood: "tension", "depression", "anger", "vigour", "fatigue" and "confusion". The messages selected for the analysis were tweets that explicitly expressed individual sentiment, matching regular expressions such as "feel"

and “I’m”. The POMS-scoring function mapped each tweet to a six-dimensional mood vector by matching the terms extracted from the tweet to the set of POMS mood adjectives for each dimension. The aggregate mood vector for the set of tweets submitted on a particular day was the average of the mood vectors of all the tweets from that day, which was used to generate a time series of aggregated, daily mood vectors over a period of time.

The authors carried out two case studies. In the first one, they analysed public mood during the 2008 US presidential elections and found, among other things, that all mood levels dropped to nominal levels on the day of the outcome of the election, except for a significant spike in “vigour” and a large drop in “fatigue”, indicating positive sentiments about the election results. The second case study is regarding to the celebration of Thanksgiving, for which they found that all mood dimensions remained nearly at baseline levels except for “vigour”, which spiked significantly on Thanksgiving day, indicating a happy and active public mood. These findings are shown in figure 3.10



(a) US presidential elections



(b) Thanksgiving

Figure 3.10: Sparklines for public mood before, during and after the US presidential elections and Thanksgiving in 2008.

The paper also includes an analysis of the ability of large-scale economic indicators such as the Dow Jones Industrial Average (DJIA) and the West Texas Intermediate oil price (WTI) to influ-

ence public mood. During the time period used for data extraction, public sentiment fluctuated significantly due to many important events such as the presidential elections and the economical crisis. In order to assess the effect of changes in the DJIA and WTI on public mood levels, they defined four time periods in which DJIA underwent significant changes in value and examined the extent of mood changes across those four periods, interpreting results visually. To check for statistical relevance, they performed a Mann-Withney U-test over all possible combinations of the time series observed within the four time periods and found that all mood curves underwent statistically significant changes from one DJIA period to the next. An equivalent study was developed for WTI prices, for which they also found statistically significant changes in mood levels across the four WTI periods.

The paper is concluded with an analysis of the results found in the studies performed: while long-term fluctuations in indicators such as the DJIA and WTI seem to have a delayed, cumulative effect on public mood, short-term events such as the news cycle, national holidays and elections have a significant and immediate effect on public mood, generating the high variability observed in the mood time series extracted from the tweets.

In [6], Bollen *et al.* present another application of sentiment analysis of Twitter messages: studying and predicting social and economical measures. The authors claim that, assuming that emotions play a significant role in human decision-making and that financial decisions are significantly driven by emotion and mood, it is reasonable to use public mood and sentiment to study and predict stock market values. They therefore investigate whether measurements of collective mood states obtained from Twitter are correlated to the value of the Dow Jones Industrial Average (DJIA) over time, and whether the same public mood states can be used to predict changes in the DJIA closing values.

In order to measure variations in the public mood, the authors used tweets that contained explicit statements of their author's mood states. The Twitter data obtained was then analysed by two different tools: OpinionFinder (OF), which analyses the text contents of tweets and provides a positive vs. negative daily time series of public mood, and the Google Profile of Mood States (GPOMS), which similarly analyses tweets to generate a six-dimensional daily time series of public mood. A total of 7 public mood time series were obtained, one generated by OF and six generated by GPOMS, which were correlated to a time series of daily DJIA closing-values in order to assess their ability to predict changes in the DJIA over time. The authors performed a Granger causality analysis in which the DJIA values were correlated to GPOMS and OF values of the previous n days. Furthermore, a Self-Organising Fuzzy Neural Network (SOFNN) was used to test the hypothesis that the prediction accuracy of DJIA prediction models can be improved by including measurements of public mood.

To enable comparison, the OF and GPOMS time series were normalised such that they would fluctuate around a zero mean and be expressed on a scale of 1 standard deviation. In order to validate their ability to capture aspects of public mood, both tools were applied to tweets posted during a period that included the US presidential election in early November and Thanksgiving in late November. Both tools successfully identified the public's emotional response to these events.

The Granger causality technique was applied to both the time series produced by GPOMS and by OF vs. the DJIA time series in order to test whether one times series had predictive information about the other one or not. The authors compared the variance explained by two linear models, one using only lagged values of the DJIA series for prediction, and the other using the DJIA values but also the GPOMS and OF mood time series. They found, for instance, that when the "calm" time series and the DJIA were plotted together, they frequently overlapped or pointed in the same direction, suggesting that the "calm" mood dimension has predictive value with regards to the DJIA. Both time series are shown in figure 3.11.

The authors also compared the performance of a SOFNN model that predicts DJIA values based on two different inputs, one using only past DJIA values, and the other using DJIA values combined with various permutations of the mood time series. The forecasting accuracy was measured in terms of the average Mean Absolute Percentage Error (MAPE) and the direction accuracy (up or down). It was found that adding the data from OF to the input had no effect on prediction accuracy, and

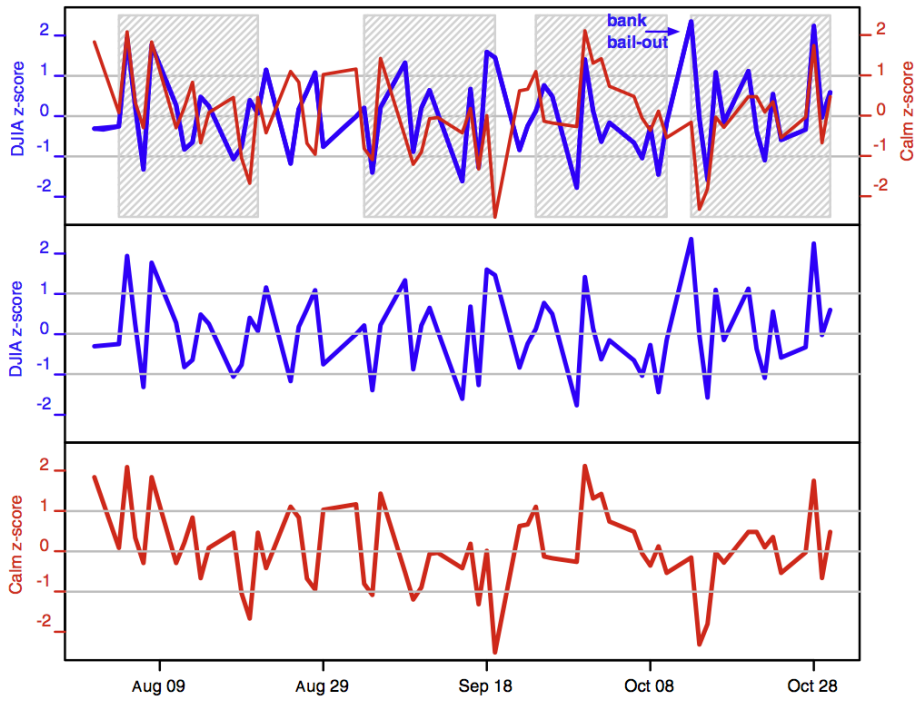


Figure 3.11: Overlap (top) and individual plots for the DJIA (middle) and GPOMS "calm" dimension (bottom) time series. Source: [6]

that adding data from the "calm" series resulted in the highest prediction accuracy. Finally, the authors assessed the statistical significance of their results by calculating the odds that they would occur by chance, and thus confirmed that the SOFNN direction accuracy was most likely not the result of chance or of selecting a particularly favourable test period.

Survey and polling methods aim to measure public opinion by asking questions to a random sample of people. In [29], O'Connor *et al.* use Twitter data to try to infer population attitudes in a similar fashion to how public opinion pollsters query a population. This method is faster and cheaper than traditional polls and allows for consideration of a greater variety of polling questions. The authors compare the sentiment measured from Twitter messages text analysis to measures of public opinion obtained from traditional polls.

The public opinion polls used in the paper measure consumer confidence, which refers to how optimistic the public feels about the economy and their personal finances, and political opinion, regarding both the presidential elections in the United States in 2008 and the presidential job approval rating for president Barack Obama over the year 2009. In order to assess the population's aggregate opinion based on Twitter data, two tasks were performed: message retrieval, which consisted of identifying the messages that were related to a specific topic, and opinion estimation, which involved determining whether the messages expressed positive or negative opinions about the topic.

The correlation between Twitter data and the polls results used was a goodness-of-fit metric for fitting slope and bias parameters in a linear least-squares model. In this model, a poll outcome was compared to the k-day text sentiment window that ended on the same day as the poll. In order to check whether changes in consumer confidence appeared in text sentiment measure before they appeared in polls, the authors introduced a lag hyperparameter L to the model, so that a poll outcome was compared against the text window ending L days before the poll outcome. With this modification, it was found that correlation is higher for text leading the poll and not the other way around, which suggests that text obtained from Twitter is a leading indicator.

For a forecasting analysis, the authors tested how well the text-based model could predict future values of the poll. The results obtained are shown in figure 3.12. It was found that text sentiment is a poor predictor of consumer confidence for data obtained in 2008 and early 2009, but a good predictor for data obtained from mid-2009, which suggests that different phenomena are being

captured by the text sentiment measure at different times.

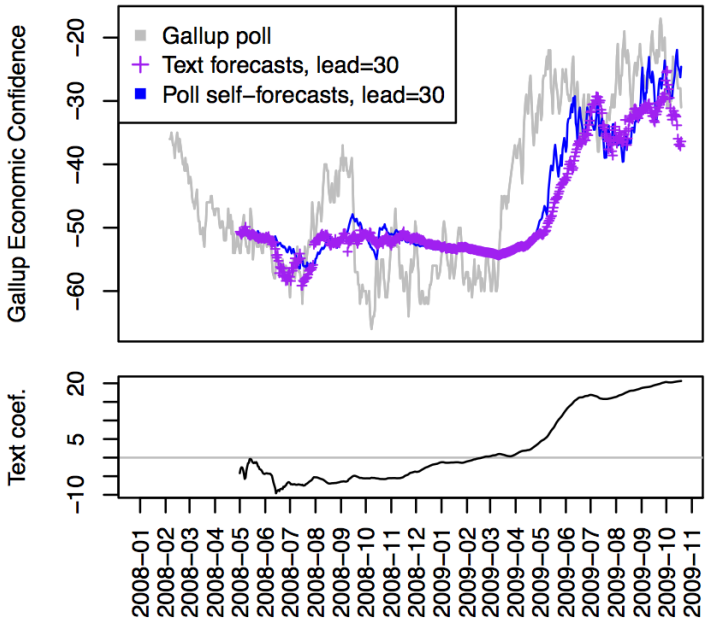


Figure 3.12: Rolling text-based forecasts (top) and text sentiment coefficients (bottom) for the text forecasting models over time. Source: [29]

The relatively simple sentiment detector based on Twitter data used by O'Connor *et al.* was able to replicate both the consumer confidence and the presidential job approval polls, which makes it a good supplement for the expensive and time-consuming traditional polling methods. The authors emphasise the advantages of using Twitter data, which can be gathered in a fast, inexpensive manner, and allows for studying measures regarding any topic that is mentioned in the posts.

3.6 Other Applications of Social Network Data

Social network data has been vastly used for research in recent years, and Twitter data has been particularly useful in studies concerning collective behaviour and opinion. The work developed in [34] is a good example of how Twitter data can be used to understand and make predictions about the behaviour of a population. In this particular study, Paul and Dredze use the data to investigate public health matters. They apply a probabilistic topic model called the Ailment Topic Aspect Model (ATAM) to the data and extend this model through the use of priors in several different applications: geographic syndromic surveillance, correlating behavioural risk factors with ailments, and correlating symptoms and treatments with ailments.

The authors used the ATAM model to translate tweets into structured disease information that can be used for public health metrics. For this work, ATAM was extended with the use of prior knowledge from WebMD.com articles, and the extended model was named ATAM+. The data used consists of over 2 billion tweets collected between May 2009 and October 2010. The ailments obtained from the models qualitatively matched their WebMD priors, but the discovered ailment clusters were often more general: for instance, the "breast cancer" prior resulted in a general "cancer" cluster.

In order to evaluate the outputs obtained from the models, the authors compared the ailment distributions obtained through ATAM+ with distributions estimated from the WebMD articles used to generate the priors. Each article was paired with its corresponding ailment in the model output, and then the Jenson-Shannon divergence was computed between the distributions for each ailment and the distributions for each WebMD disease. The divergence was used to compute a score that defined a ranking of articles for ailments and a ranking of ailments for articles; associations between an ailment and an article were considered correct when the correct answer had rank 1.

The results obtained showed that ATAM+ outperformed ATAM, which indicates that the priors produced more coherent ailments.

The article presents different applications of the ATAM+ model created, such as analysing temporal and geographic impacts on medical wellbeing, and investigating how the public treats illness. In one application, the authors used tweets with allergy-related keywords to compute the rate of the allergy ailment by american state from February to June 2010. Results obtained are shown in figure 3.13.

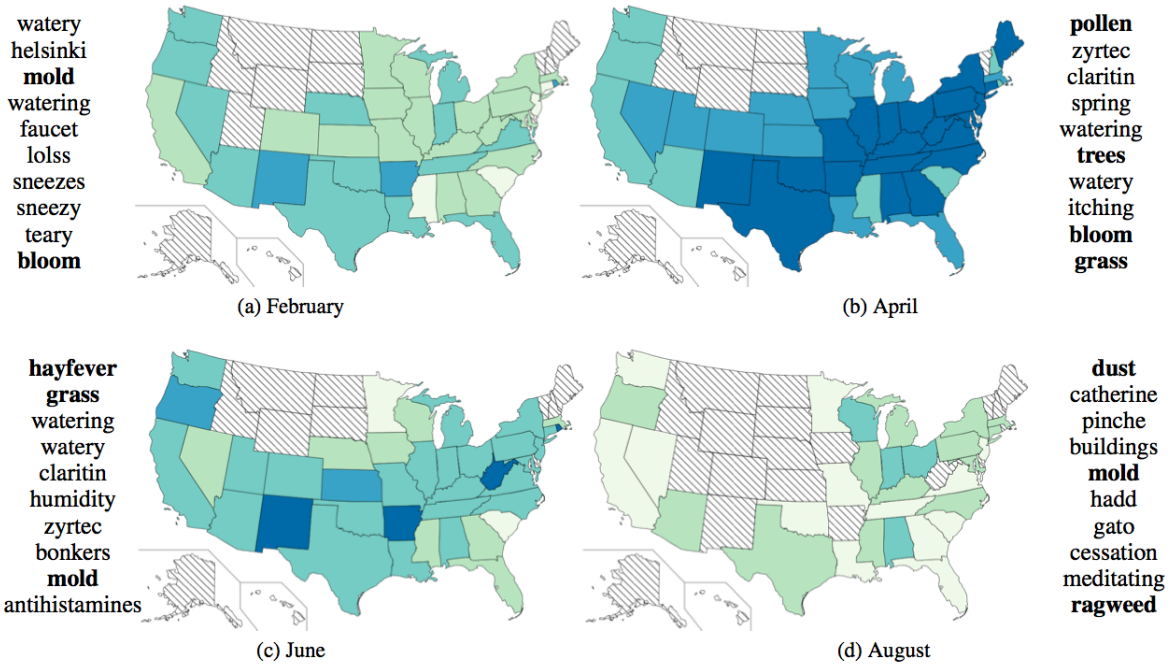


Figure 3.13: Rates of allergies by state discovered by ATAM+ during a period of four months in 2010. Darker shading indicates higher allergy rates, while diagonal shaded states lacked sufficient data. Source: [34]

The authors conclude the article by discussing some limitations of the use of Twitter for obtaining information on public health. While a variety of public health data can be automatically extracted from Twitter, resulting in different measures regarding the population in general, statistics that require a single user to post multiple tweets could not be obtained, since the vast majority of users only had a single tweet in the data used. The authors argue that this suggests that Twitter data is a better fit for population level metrics rather than understanding the individual behaviour of users. Paul and Dredze's paper is one example of many studies that use Twitter data to represent collective behaviour and opinion. The majority of these studies, however, does not attempt to use Twitter data for understanding individual behaviour, which is the main goal of our project.

While most previous work with social network data focuses on collective behaviour, there is a smaller segment that has explored the possibility of extracting individual level metrics from this data. A very recent study of individual behaviour using social network data can be found in [1]. Bachrach *et al.* use data extracted from Facebook to understand the correlations between features of a person's online profile and their real personality. Some of the features studied were the size and density of a person's friendship network, the number of uploaded photos and the number of events attended, among many others. The authors also performed multivariate regression to predict an individual's personality traits based on their Facebook profile. Users' personalities were evaluated through the standard Five Factor Model personality questionnaire, which is currently the most widespread and generally accepted model of personality.

In order to study the relation between a given Facebook feature and each personality trait, the authors clustered together users with similar features: users were partitioned into equal-sized, disjoint groups with increasing feature values, then the average values of personality trait scores

for each group were examined. The personality traits present in the Five Factor Model are the following: openness to experience, conscientiousness, extraversion, agreeableness and neuroticism. Openness was found to be positively correlated with number of users' likes, group associations and status updates; conscientiousness was negatively related to the number of likes and group membership, but positively related to number of uploaded photos; extraversion was correlated to the number of Facebook friends and more interaction with other users through groups; agreeableness was correlated with appearing in more pictures with other users and negatively correlated with the number of likes; finally, neuroticism was found to be positively correlated with number of likes and number of groups.

The authors tested the statistical significance of their findings by using a t-distribution test, to test against the null hypothesis of no correlation, and a Mann-Whitney-Wilcoxon test, to determine whether the top and bottom thirds of the Facebook users differed significantly in terms of personality trait scores. In order to predict a user's personality based on multiple profile features, they used a multivariate linear regression model, and evaluated the model using the coefficient of determination, R^2. They found that some personality traits can be predicted with reasonable accuracy using high-level Facebook features, and that the use of more sophisticated machine learning methods, such as tree based rule-sets and support vector machines, did not considerably increase the value of R^2. Bachrach *et al.* present in their paper an interesting use of social network data for Reality Mining and the study of individual behaviour. Nonetheless, one drawback of their study is that the data was not easy to obtain, since the questionnaires used in the Five Factor Model had to be requested from each of the users studied.

In this chapter, we have reviewed several research papers pertaining to Reality Mining, Computational Social Science and the use of Twitter data for a variety of applications. In the next few chapters, we will show how our project builds on some of these papers with a study of human behaviour through data retrieved from Twitter.

Chapter 4

The Creepy Crawly Software Application

The first task to be executed in our project was the collection of data from Twitter. This was achieved through the development of a Twitter crawler, a small computer program that browses Twitter and gathers its contents in a methodical, automated manner. We have affectionately named our Twitter crawler Creepy Crawly, due to its controlled and careful crawling process throughout the social network graph. Access to Twitter was possible due to the Twitter Application Programming Interface (API), a specification that allows communication between the crawler and Twitter itself. In this chapter, we give a brief description of the structure of Twitter, an explanation of the Twitter API, implementation details of the application developed for data collection and, lastly, we describe the data retrieval process and the database created for storing this data. Figure 4.1 shows a simple diagram of the complete system implemented.

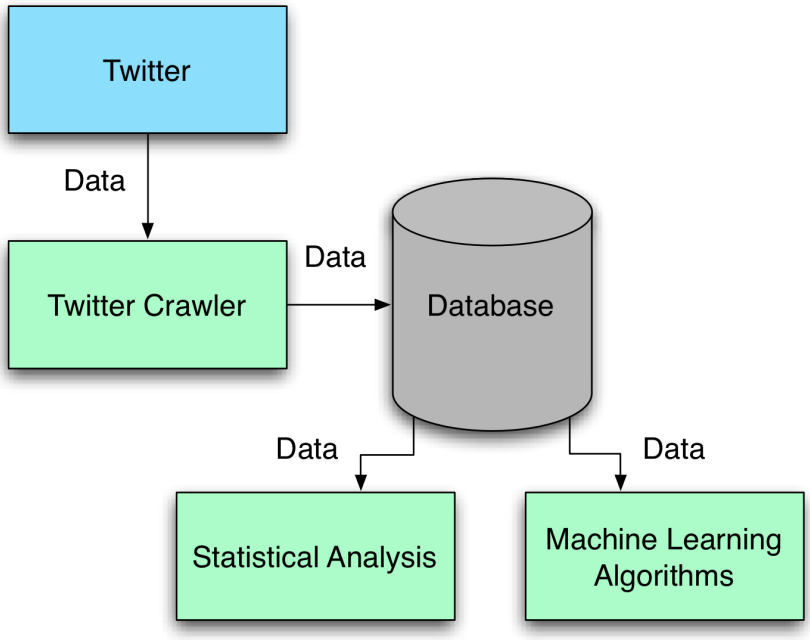


Figure 4.1: System overview diagram.

There is currently a large number of companies that sells data from social networks like Twitter. Furthermore, there are many different crawlers freely available online that allow for data collection. Nonetheless, we had several reasons to develop our own Twitter crawler and collect our own data:

- Data provided commercially by companies is usually expensive, and we wanted to take advantage of the fact that all data on Twitter is available for free;

- In order to collect data from specific account types, we had to be able to exactly specify the accounts to be tracked;
- The Twitter API has a restrictive request limit, which only allows clients to make 150 requests per hour;
- In the past, Twitter would "white list" some accounts in order to facilitate data collection by increasing the API request limit. However, this is no longer an option;
- Using multiple IP addresses to bypass the API's request limit and retrieve data is against Twitter's usage policy [1].

Building our own Twitter crawler allowed us to bypass Twitter's bandwidth restrictions by simply creating multiple accounts that individually respected the request limit. Moreover, we were able to choose the Twitter accounts we wanted to collect data from, which was a crucial factor for studying the behaviour of different types of account.

4.1 The Structure of Twitter

The primary structure of Twitter can be described as follows: each user can post text updates of up to 140 characters, called tweets, on their profile page. The collection of tweets posted by a user constitutes their timeline, with the most recent tweet appearing first on the profile page. Relationships between Twitter users are created by the process of following. When a user A follows another user B, A becomes B's follower and B becomes A's friend. B in turn can follow A to make the relationship mutual [2]. When user A chooses to follow user B, A will be able to see B's tweets on their homepage. The profile of a user on Twitter has therefore three main attributes: a set of tweets (their timeline), a set of followers, and a set of friends (users that they chose to follow).

User profile: The user profile on Twitter consists of the following main attributes: name, screen name (which is unique on the network), location, short biography, website and picture. Other attributes, such as the unique user id and time zone, can only be accessed through the Twitter API. Users can edit their profile information whenever they want.

Following other users: To start following another user, the user must go to the other's profile page and click on 'Follow'. On their home page, the user gets a feed of tweets from all the users they choose to follow, also called their friends.

Tweeting: From their home page, the user can compose a new tweet, which will be broadcast to all their followers. A tweet has a timestamp that describes the exact moment it was posted, and can include url's and media (photos, videos, etc.). The stream of the user's tweets, called their timeline, appears on their profile page, with the most recent tweet first.

Retweeting: A retweet (RT) is a reposting of someone else's tweet, constituting a way of spreading or popularising someone else's tweet. Any tweet that the user retweets will be broadcast to their followers, even if the followers do not follow the user who originally created the tweet. The retweet will also appear on the user's timeline in their profile page. To retweet someone else's post, the user must hover the mouse over it and click 'Retweet'.

Mentions: A user can mention another user in one of their tweets by using the symbol '@' followed by their user name, anywhere in the tweet. The mentioned user will receive a notification that they were mentioned.

Replies: To reply to a tweet, the user must hover the mouse over it and click 'Reply'. The screen name of the user being replying to will be automatically added to the beginning of the tweet. The replied user will receive a notification that they were replied to. This feature allows users to publicly exchange messages on Twitter.

Favorites: The user can save tweets that they particularly like in their favourites list by hovering the mouse over the tweets and clicking on 'Favorite'. The list of favourite tweets can be seen on the user's profile page.

[1] support.twitter.com/articles/18311-the-twitter-rules

Hashtags: A hashtag '#' is used to mark keywords or topics in a tweet. Using the hashtag symbol before a relevant keyword or phrase (with no spaces) in a tweet is useful because it categorises the tweet and will make it appear more easily in Twitter search. A hashtag can be added anywhere in the tweet.

Direct messages: Users can also exchange messages on Twitter privately using direct messages, which are similar to mentions but are private between the interacting users.

Trends: On their home page, the user can see a list of the topics and hashtags that are currently trending on Twitter.

4.2 The Twitter API

The Twitter API [2] consists of a large collection of resources that enable developers to access Twitter through their programs, without the use of a web browser. As described in its documentation, the API provides methods that can perform nearly every feature present in the Twitter website. In this project, we used only one component of this API, called the REST API, which has methods for dealing with timelines (collections of tweets ordered by timestamp), tweets, users, friends and followers, among other features.

The three main objects handled by the Twitter API are users, tweets and entities. The most important attributes of each of these objects are described in table 4.1.

(a) User object	(b) Tweet object	(c) Entities object
Attributes:	Attributes:	Attributes:
created_at	created_at	hashtags
description	entities	media
favourites_count	favorited	urls
followers_count	id	user_metions
friends_count	in_reply_to_screen_name	
id	in_reply_to_status_id	
language	in_reply_to_user_id	
location	retweet_count	
name	text	
screen_name	user	
status		
statuses_count		
time_zone		
url		
utc_offset		

Table 4.1: Twitter API objects.

The following API methods were used in the development of the Creepy Crawly application:

- UsersLookup: returns the profiles of users (list of User objects) given their id's;
- GetUser: returns the profile of a user (User object) given their screen name;
- GetFollowerIds: returns the id's of followers of a specific user, given their id;
- GetFriendIds: returns the id's of friends of a specific user, given their id;
- GetUserTimeline: returns the timeline (list of Tweet objects) for a specific user, given their id;

[2] dev.twitter.com/docs

- GetRetweets: returns the retweets of a status (list of Tweet objects), given the status id;
- GetFavorites: returns the user's favourite tweets (list of Tweet objects), given the user id.

The Twitter API also provides methods for dealing with the request limit it imposes, which are explained in more detail later in this chapter. In order to gain access to the API, one must first create a Twitter account, then request a consumer key, a consumer secret, an access token key and an access token secret. These unique values are used by Twitter to control access to the API.

4.3 System Design and Implementation

The Creepy Crawly application was developed in Python [3] script language, due to its simplicity, efficiency and fast implementation. Additionally, a wide variety of third-party libraries exist to support web access in Python applications. The libraries used in the implementation of our crawler were the following: python-twitter [4], oauth2 [5], httplib [6], json [7], psycopg [8], and pyparsing [9]. Python-twitter is a Python wrapper library for the Twitter API; oauth2 is an implementation of the Oauth protocol, which allows for secure authorisation in web applications; httplib implements the client side of the HTTP protocol; json is a JSON (JavaScript Object Notation) encoder and decoder library; psycopg is a PostgreSQL adapter for Python, which queries the PostgreSQL database; finally, pyparsing is a text parsing library which allows the user to create grammars directly in Python code. Details about these libraries are omitted from this paper, but their documentations can be obtained from their respective web links. The whole Creepy Crawly application consists of four Python modules to be described in this section: `crawler`, `rateLimiter`, `databaseAccess` and `errorReport`. Figure 4.2 shows the UML diagram for these modules.

[3] docs.python.org/tutorial/
[4] code.google.com/p/python-twitter/
[5] github.com/simplegeo/python-oauth2/
[6] docs.python.org/library/httplib.html
[7] docs.python.org/library/json.html
[8] initd.org/psycopg/
[9] pyparsing.wikispaces.com/

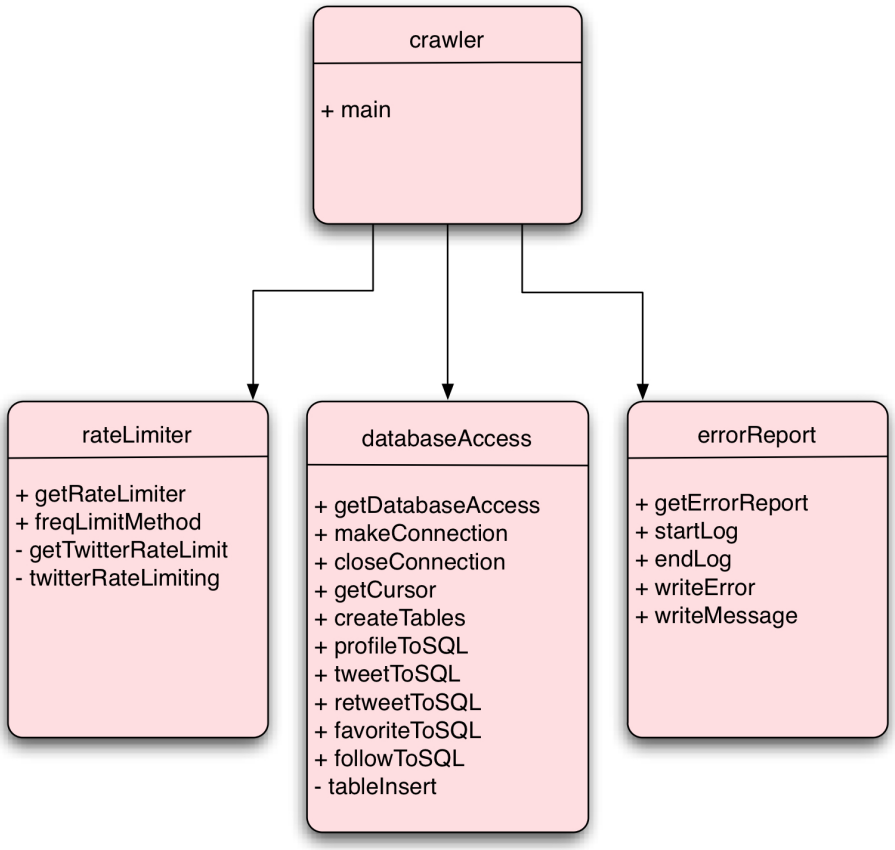


Figure 4.2: UML diagram of the Python modules.

Preventing Twitter Throttling

As previously mentioned, one significant shortcoming in using the Twitter API for data retrieval is its restrictive limit policy, which only allows clients to make 150 requests per hour. Even if a client makes calls to the API within the allowed limit, Twitter may throttle the account when too many calls are made repeatedly. For these reasons, the first step in the development of the crawler was creating a wrapper module for the python-twitter library, in order to add small time intervals in between requests to the API and thus prevent the account from being "black listed" by Twitter. Module `rateLimiter` was created for this purpose. Before making a call to the Twitter API, this module obtains from the API the minimum amount of time that the program must wait before making the next call without exceeding the rate limit. It then puts the program to sleep until the specified amount of time has passed, when it finally makes the call.

Access to the Database

In addition to the crawler application, it was necessary to create a database where all the data collected from Twitter was stored. This database was created in PostgreSQL and is described in the next section, Data Collection and Storage. Access to the database was implemented in module `databaseAccess`, in order to confine all database queries to just one module. This module contains methods to create tables and to insert all retrieved data into these tables. Furthermore, it includes two exception classes, `ConnectionClosed` and `CursorClosed`, which are raised when the connection or the cursor to the database are lost.

The Restricted and Unrestricted Crawlers

The main loop of the Creepy Crawly application was implemented in module `crawler`. This module uses `rateLimiter` to extract the data via Twitter API, then uses `databaseAccess` to store this data

into the database. Lastly, module `errorReport` is used to generate a log file describing exceptions and errors that may occur while the application is running, in addition to informing which profiles have already been processed. In order to differentiate between tweets and retweets in a user's timeline, module `crawler` uses a parsing mechanism which checks for the sequence "RT @[user screen name]" in the beginning of a message. This sequence characterises a retweet of anther user and, although not visible on the Twitter interface, it is present in the text obtained through the API. Two different versions of the crawler were implemented: unrestricted and restricted.

The unrestricted crawler has the goal of spreading as much as possible throughout the Twitter network, and for this purpose it performs Breadth-First Search (BFS) based crawling. The program is given the screen name of a seed user, from which the crawling process begins. The crawler adds this user to a queue and begins a loop to handle the contents of the queue: while the queue is not empty, the first user in the queue has their profile processed and their followers and friends also added to the queue. Processing a profile means sending all the information relating to that user to the database: profile attributes, timeline (up to 200 tweets), and follow connections. Follow connections are stored as directed edges in a graph, where one node corresponds to the follower and the other node corresponds to the followed user. Since some users on Twitter such as celebrities have an extremely large number of followers, we discarded users who had more than 100,000 followers so as to avoid slowing down the data collection process. The unrestricted crawler runs indefinitely, spreading throughout the Twitter network until the program is interrupted.

In the restricted version of the crawler, the program is given a full list of the screen names of users to process, and collects more detailed data than the unrestricted version. The crawler simply goes through the list and processes each user, adding their information to the database, and ceases to run as soon as the list has been fully processed. The information collected for each user consists of profile attributes, timeline (both tweets and retweets, up to 700 posts total), favourite tweets, friends and followers.

Parallelism and Performance

In order to retrieve the data in an efficient way, it was important to have multiple Python scripts running in parallel for long periods of time. However, this setting brought about some difficulties that had to be overcome. First, to make the system robust to possible run-time exceptions, such as errors from the Twitter API or loss of the connection to the database, we implemented careful exception handling for every call to the API and every database query. To avoid exceeding the API rate limit, multiple Twitter accounts were created, so that a separate account (and corresponding API key and access token) could be used in each running crawler. Finally, the crawlers were set up as long running processes in a virtual machine using the Ubuntu 10.04 operating system. This virtual machine was accessed via Secure Shell (SSH). In order to run the crawler application via command line, a very simple shell script named `runCrawler` was used, consisting of the following two lines:

```
#!/bin/sh
python ./crawler.py
```

And the following command was used to run the shell script and thus set the process running in the background, detached from the keyboard and the screen, and immune to hangup signals:

```
nohup ./runCrawler </dev/null >&/dev/null &
```

The performance of Creepy Crawly was evaluated through manual measurements. The unrestricted version was able to collect, on average, data from 20 Twitter accounts per hour, whereas

for the restricted version that rate was reduced to one user every 2 hours. This significant difference in performance is because the restricted crawler collects more detailed data and can extract up to 700 tweets from each user, compared to only 200 tweets in the unrestricted version. In addition, the parsing of tweets in the restricted crawler, used to differentiate between tweets and retweets, further retards the crawling process.

4.4 Data Collection and Storage

The data was retrieved from Twitter in June 2012, then aggregated and inserted into a PostgreSQL database. Two separate datasets were constructed: the uncategorised dataset was created through the unrestricted crawler, while the categorised dataset was created through the restricted crawler.

The uncategorised dataset has three main components: user profiles, tweets and social graph. The user profiles contain the following information about each Twitter user: unique user id, name, screen name, location, UTC (Coordinated Universal Time) offset, time zone, number of tweets in timeline, number of followers, number of friends, number of favourites, language, and date of profile creation. These attributes were implemented as columns in a table called *profiles*. Similarly, table *tweets* was created, containing the attributes for that component. Details about each table are shown in figure 4.3, a diagram of the tables found in the uncategorised dataset.

The user graph is a directed graph where each Twitter user is a node and the follow relationships between users are edges, i.e., there is an edge in the graph from node A to node B if and only if user A follows user B on Twitter. This graph is represented in table *social graph*, containing two columns: parent, where the user id of the follower is stored, and child, where the user id of the followed user is stored. Therefore, each row in the graph table corresponds to an edge in the graph.

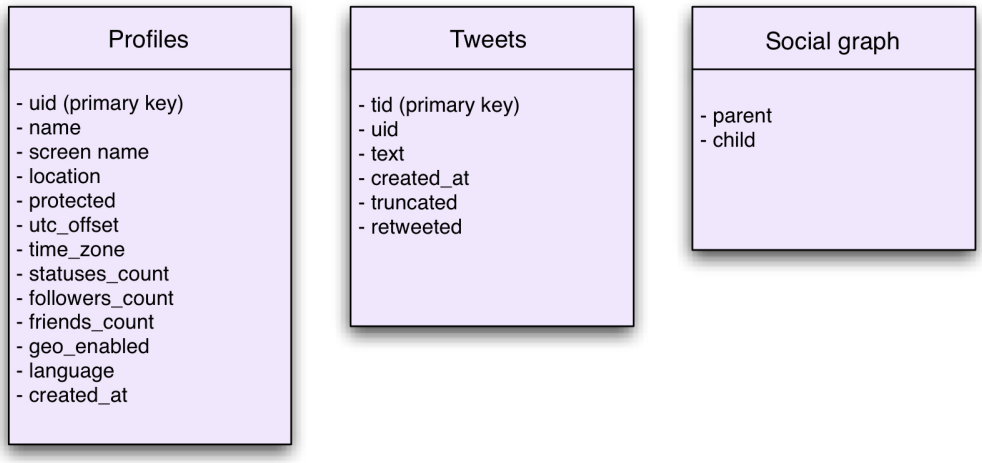


Figure 4.3: Diagram of tables in the uncategorised dataset.

The categorised dataset consists of three classes (Personal, Managed and Bot-controlled). The unrestricted crawler was run in three separate instances, each instance with a list of 100 Twitter accounts from one of the classes. These Twitter accounts were manually selected and had their profile information and timelines carefully analysed. They were then classified into one of the three classes, and the result was considered a ground truth dataset in the analyses that followed. All accounts chosen were unprotected (their contents were visible to the public) and were being actively used at the time when the data was collected.

Each account class contains six main components, namely, user profiles, tweets, retweets, favourites, friends and followers. In order to store this data, six tables for each class were created: *profiles*, *tweets*, *retweets*, *favourites*, *friends*, and *followers*. Tables *friends* and *followers* are similar and contain only two columns, each column representing one end of an edge in the network graph. In the *friends* table, the first column is the user id, and the second column is the friend id; in the *followers* table, the first column is the user id, and the second column is the follower id.

Details about each table are shown in figure 4.4, a diagram of the tables found in the categorised dataset.

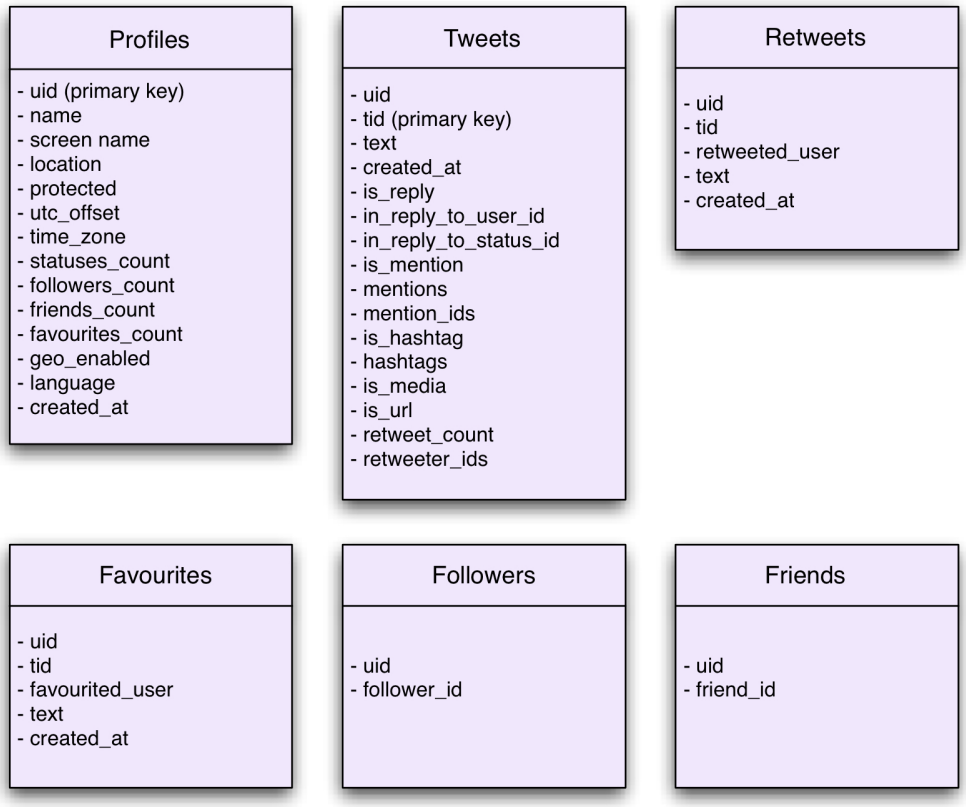


Figure 4.4: Diagram of tables in the categorised dataset.

In order to be analysed and used in the algorithms, the data was exported from the PostgreSQL database to text files ('.txt'), which can be read in the MATLAB environment.

In this chapter, we have described the implementation of our Twitter crawler application, Creepy Crawly, which successfully retrieves data from Twitter while respecting the API request limitations. The application was written in Python with the aid of various Python libraries. Two versions of the application were implemented: the unrestricted version starts from a seed user and continuously spreads across the Twitter network, while the restricted version collects data from a pre-determined set of accounts. In order to increase performance, several scripts were run in parallel on a virtual machine. Furthermore, a Postgresql database was created for data storage.

Chapter 5

Data Analysis

In this chapter, we present the statistical analyses performed for the two datasets retrieved through the Creepy Crawly application. The uncategorised dataset is the largest, consisting of 10,958 user accounts and over 890,000 tweets, and was therefore used for analysing general characteristics of Twitter accounts, such as number of tweets, number of followers, number of friends and tweet frequency. For these characteristics, a larger amount of data improves the results of the analysis. The categorised dataset, on the other hand, is much smaller, containing only 100 user accounts for each of the three classes. This dataset was used for analysing behavioural characteristics specific to each account class. The graph analysis and plot for the uncategorised dataset were produced with the Gephi platform. All other statistical analyses and plots were produced in the MATLAB environment.

5.1 Analysis of the Uncategorised Dataset

We begin our study of the uncategorised dataset with a brief analysis of the user network obtained, following the concepts presented in section 2.3. A directed graph containing all users and 'follow' links in the uncategorised dataset is shown in figure 5.1. We removed from the graph edges for which one end-node did not belong to the dataset. In this graph, node size and colour are proportional to degree (including both in and out edges). The network contains a total of 10,958 nodes and 34,440 edges. The average degree is 3.14, the network diameter is 19, and the average path length is 5.48. Graph analysis and plotting were performed with the aid of the igraph library [1] and the Gephi open-source platform. The graph was built through a Python script using the igraph Python interface, then exported as a '.graphml' file and finally plotted in Gephi. The layout of the graph was obtained through the use of the Force Atlas algorithm.

The number of tweets, number of followers and number of friends are three simple yet important properties of Twitter accounts, since they allow us to quickly assess the participation of users on the social network. Figures 5.2, 5.3 and 5.4 show the probability density functions (PDF) for number of tweets, number of followers and number of friends, respectively. All three distributions are intrinsically associated with the time the data was collected (June 2012). We observe that the three properties roughly follow a power law distribution, resulting in approximate straight lines in the log scale plots. The distribution for number of followers in figure 5.3 is not strictly factual since we limited our data collection to users with at most 100,000 followers. Therefore, it applies to our dataset only and not to Twitter in general. Furthermore, looking at figure 5.4, we can see a glitch in the probability at 2,000 friends, which is caused by an artificial friend limit imposed by Twitter. There is a small number of users with more than 2,000 friends, though, since Twitter makes some exceptions to this rule based on the user's follower to friend ratio.

[1] igraph.sourceforge.net

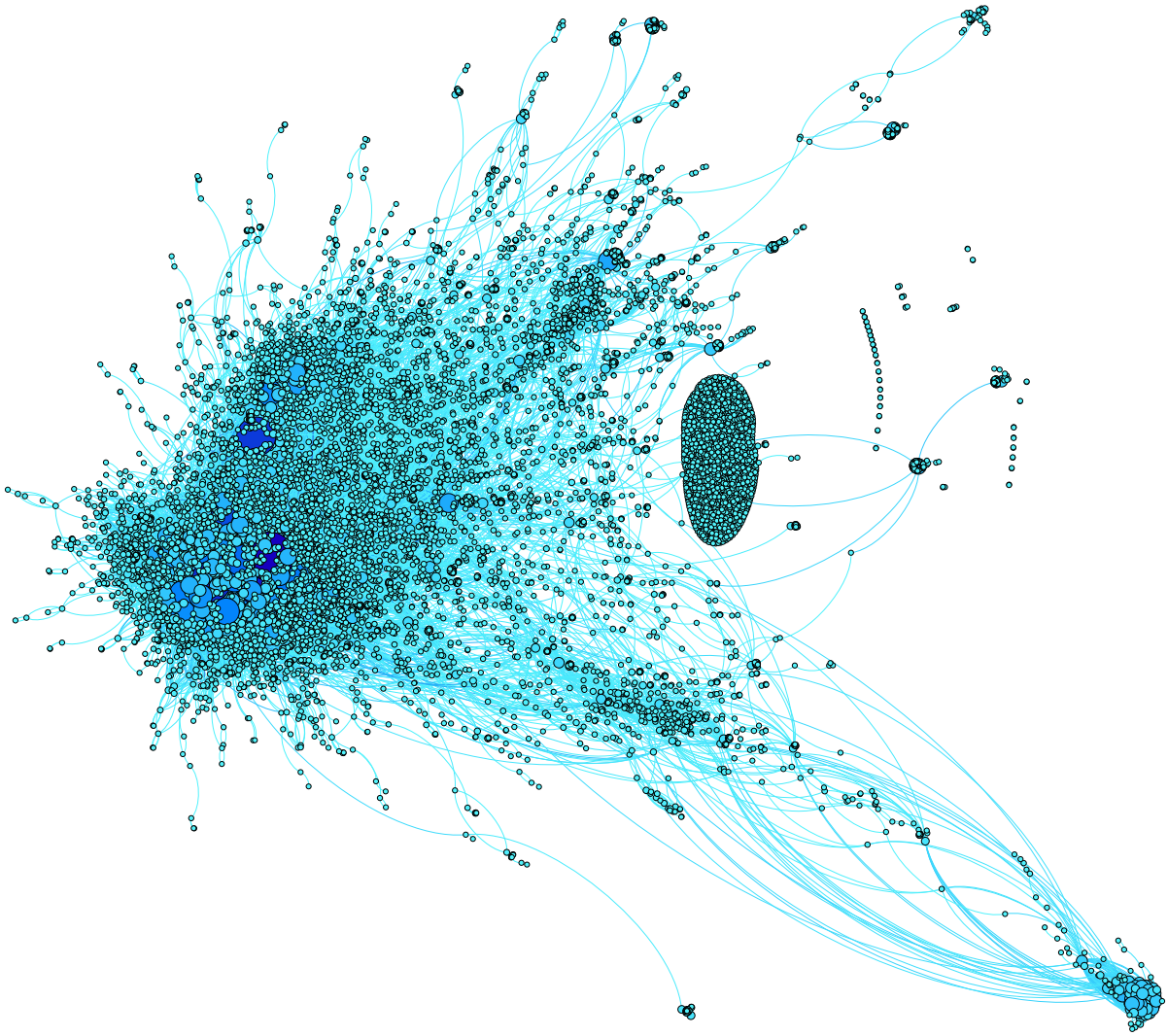


Figure 5.1: Graph of the Twitter network in the uncategorised dataset. Only the edges connecting two users that exist in our dataset are shown. Node size and colour are proportional to node degree. This plot was generated in the Gephi open-source platform, and the layout was obtained through the use of the Force Atlas algorithm.

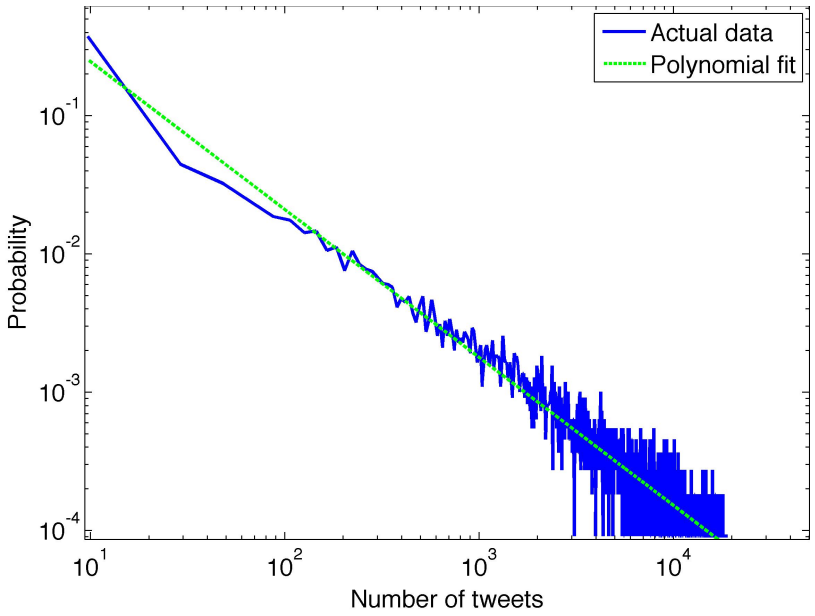


Figure 5.2: Probability density function for number of tweets in the uncategorised dataset. The distribution follows a power law with exponent approximately -1.07 (slope of the polynomial fit). The coefficient of determination obtained for the polynomial fit of degree 1 was $R^2 = 0.88$.

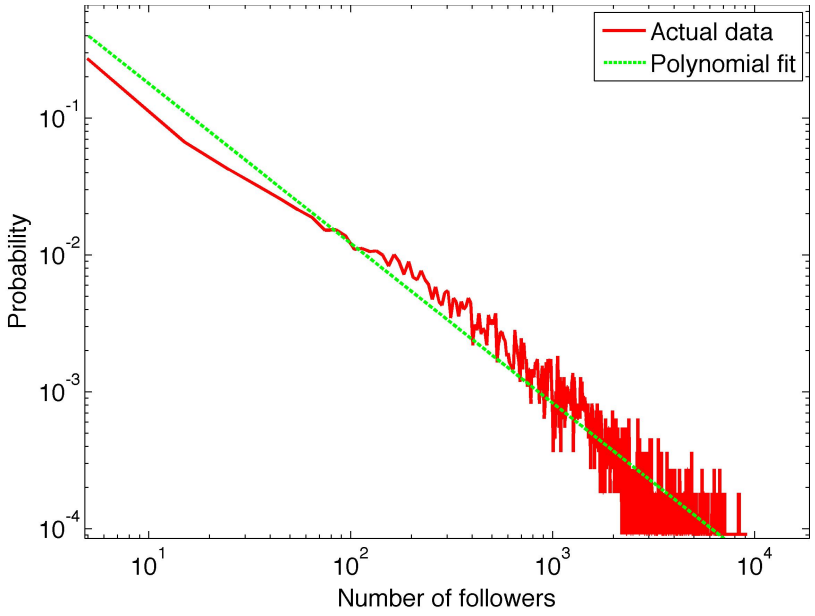


Figure 5.3: Probability density function for number of followers in the uncategorised dataset. The distribution follows a power law with exponent approximately -1.17 (slope of the polynomial fit). The coefficient of determination obtained for the polynomial fit of degree 1 was $R^2 = 0.89$. This distribution applies to our dataset only and does not correspond to the real Twitter network, since we limited data collection to users with at most 100,000 followers.

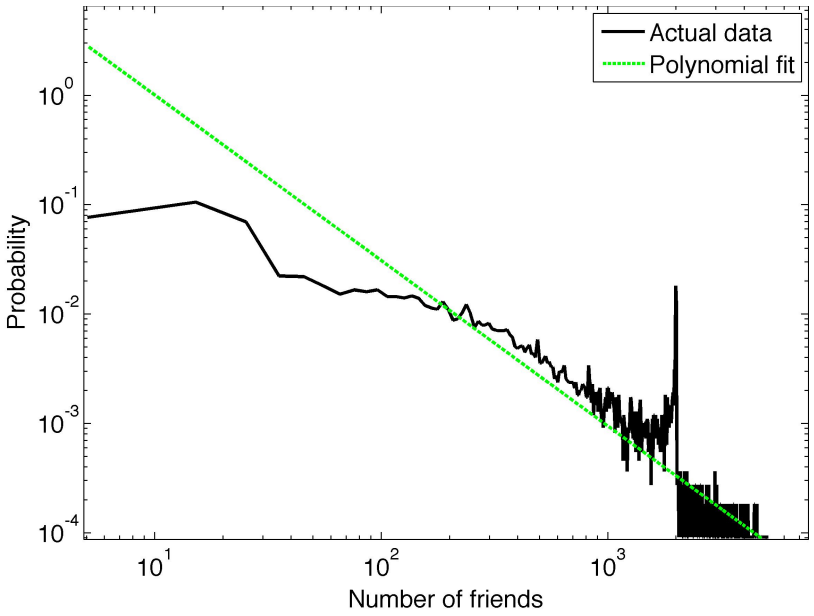


Figure 5.4: Probability density function for number of friends in the uncategorised dataset. The distribution follows a power law with exponent approximately -1.51 (slope of the polynomial fit). The coefficient of determination obtained for the polynomial fit of degree 1 was $R^2 = 0.84$. The glitch in the distribution at 2,000 friends corresponds to the maximum friend limit imposed by Twitter.

Figures 5.5, 5.6 and 5.7 show the complementary cumulative density functions (CCDF) of the fraction of users in the dataset who have number of tweets, friends and followers above a certain number. Again, the glitch at 2000 friends in the number of friends CCDF is due to the artificial friend limit imposed by Twitter.

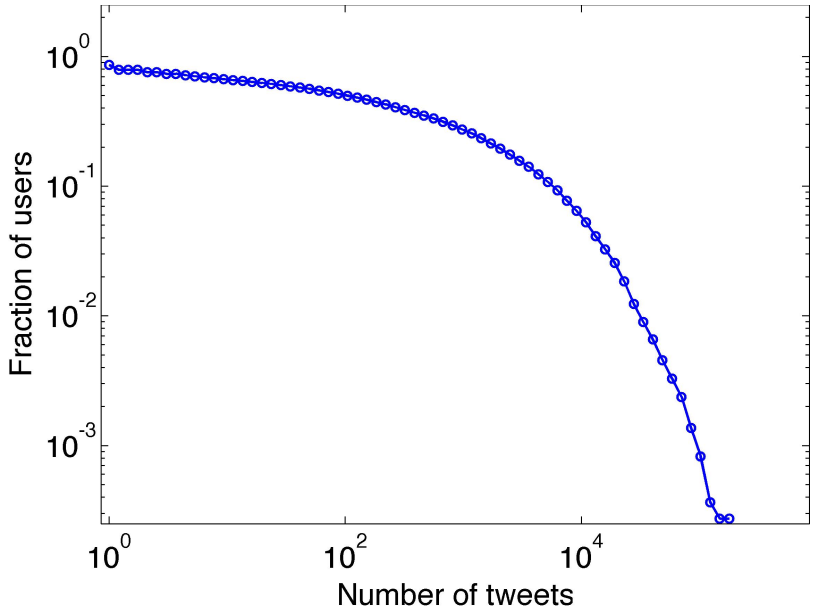


Figure 5.5: Complementary cumulative density function for number of tweets in the uncategorised dataset. This distribution shows the fraction of users in the dataset that have at least a given number of tweets.

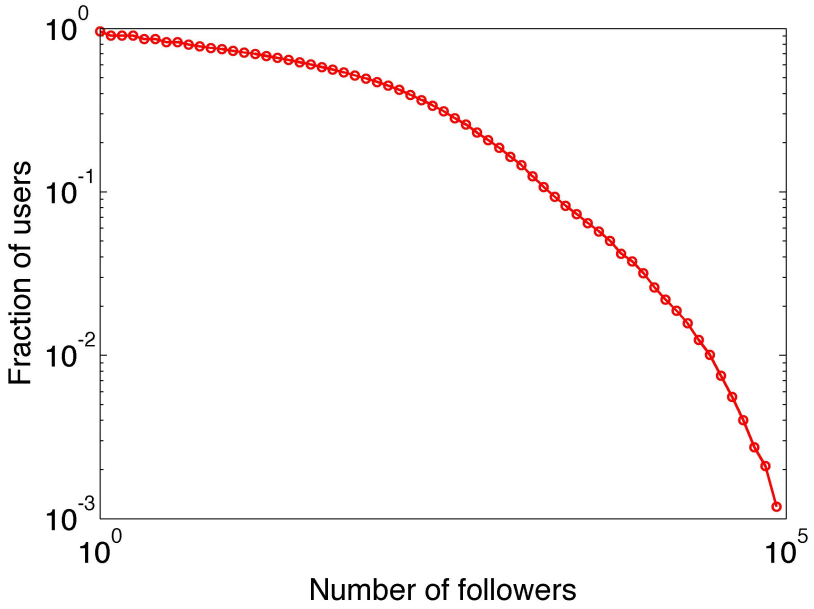


Figure 5.6: Complementary cumulative density function for number of followers in the uncategorised dataset. This distribution shows the fraction of users in the dataset that have at least a given number of followers.

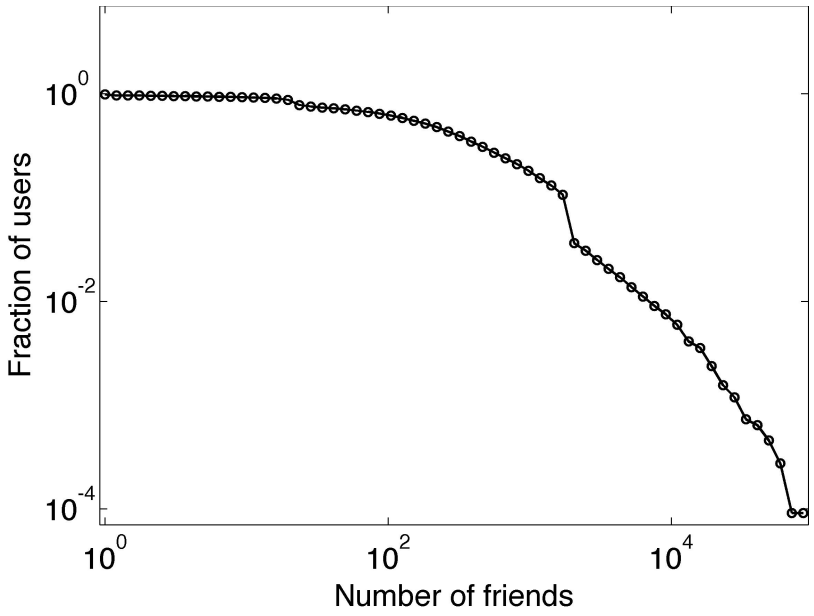


Figure 5.7: Complementary cumulative density function for number of friends in the uncategorised dataset. This distribution shows the fraction of users in the dataset that have at least a given number of friends. The glitch at 2,000 friends is due to the maximum friend limit imposed by Twitter.

The tweet frequency measures how actively users are posting on Twitter. We define this measure as the number of tweets posted by a user in one day, including retweets of other users. We can calculate a user's tweet frequency by dividing their number of tweets by the time elapsed between their first and their last tweet in the dataset. Figure 5.8 shows the histogram for tweet frequencies in the uncategorised dataset. Since we collected at most 200 tweets per user, the maximum value for the tweet frequency is 200.

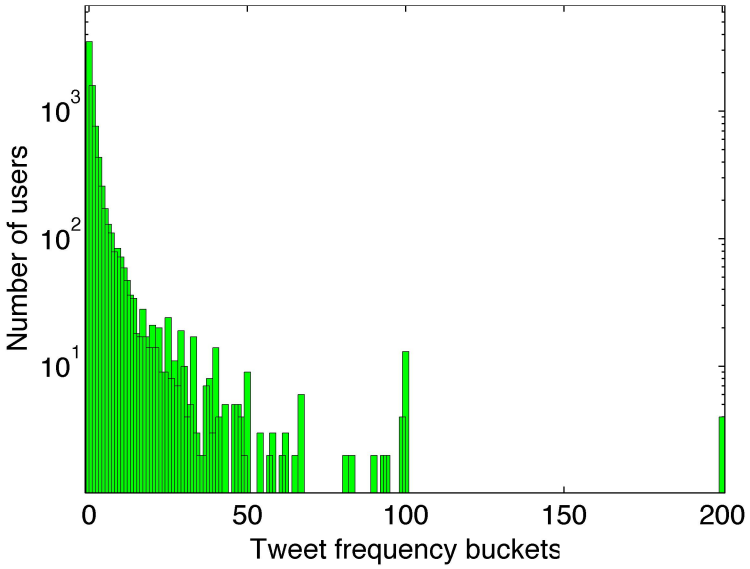


Figure 5.8: Tweet frequency for the uncategorised dataset. The maximum value is 200 because we collected at most 200 tweets per user.

All probability distributions obtained for the uncategorised dataset are consistent with those found in previous research [2, 23, 12]. This indicates that our datasets are valid and unbiased.

5.2 Analysis of the Categorised Dataset

The categorised dataset was used for analysing and comparing the behaviour of users in each account class, namely, Personal, Managed, and Bot-controlled. The two main properties studied were the tweet times, determined by the timestamp of each tweet, and the inter-tweet delay, i.e., the amount of time elapsed between two consecutive tweets by the same user. The timestamps of tweets, provided by the Twitter API, are regulated by the UTC standard. In order to adjust these timestamps to the time zone of each user, we added to each timestamp the UTC offset, an attribute also provided by the Twitter API for users who have specified their time zones. Users who did not specify their time zone were hence discarded from this analysis. Consequently, our dataset, which originally had 100 accounts in each category, was reduced to 86 personal accounts, 91 managed accounts and 67 bot-controlled accounts.

We begin by studying the inter-tweet delay distributions in each class. Figures 5.9, 5.10 and 5.11 show the PDF's for the inter-tweet delay in classes Personal, Managed and Bot-controlled, respectively. For all three classes, we obtain an approximate straight line in the log scale plot, which indicates that the inter-tweet delay follows a power law distribution. In particular, for the personal accounts, which are controlled by only one individual, the coefficient of determination obtained between the distribution and the polynomial fit was the highest at 0.91. This finding is in accordance with many previous studies in Computational Social Science [3, 30, 14], some of which were described in chapter 3. According to Barabási [3], the power law distribution is indicative of a bursty, non-Poisson character: during a single session, an individual will post several tweets, then follow with a long period of inactivity on Twitter, a behaviour that is characteristic of the timing of many human actions.

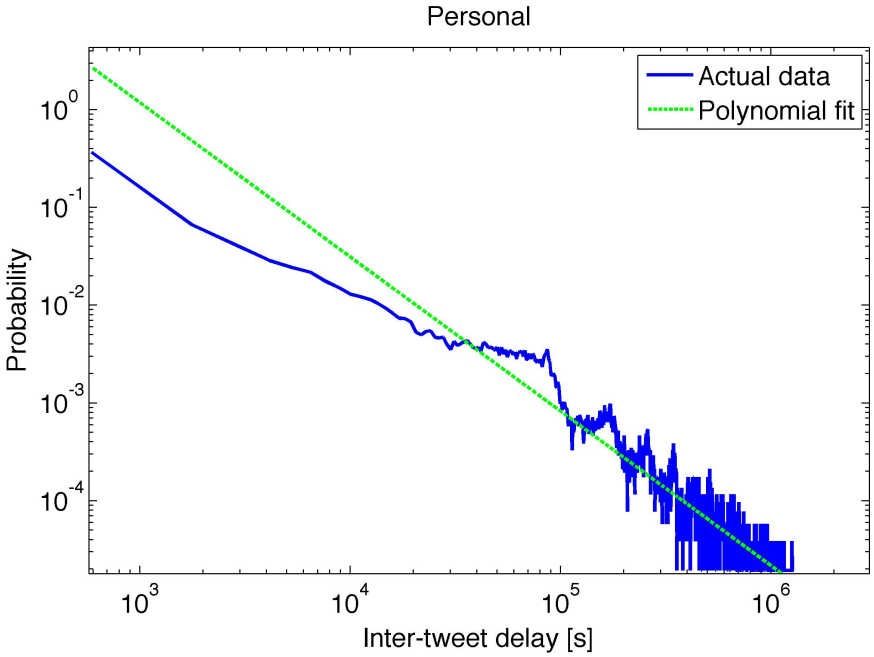


Figure 5.9: Probability density function for the inter-tweet delay of personal accounts. The distribution follows a power law with exponent approximately -1.58 (slope of the polynomial fit). The coefficient of determination obtained for the polynomial fit of degree 1 was $R^2 = 0.91$. A total of 86 accounts was used in this analysis.

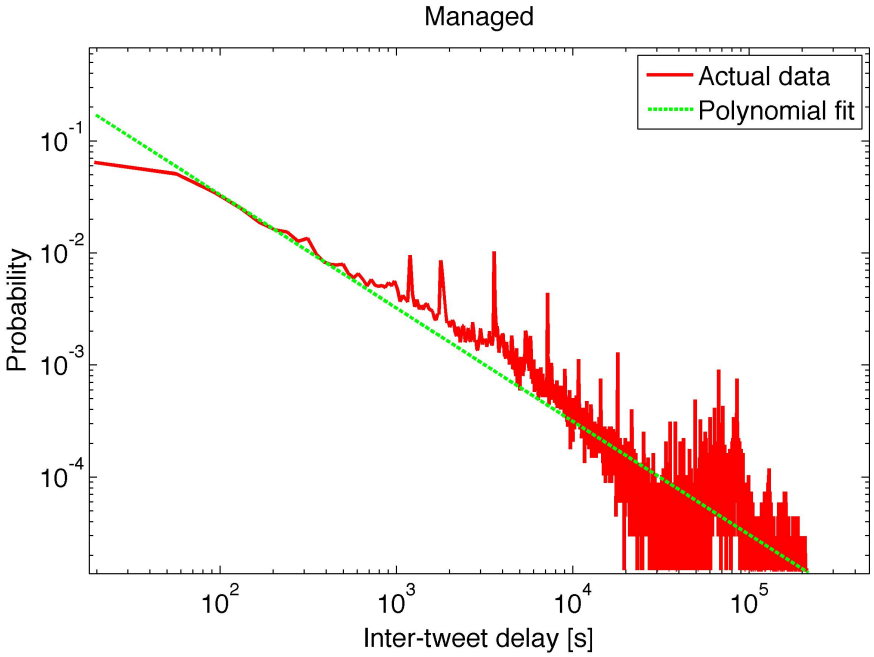


Figure 5.10: Probability density function for the inter-tweet delay of managed accounts. The distribution follows a power law with exponent approximately -1.01 (slope of the polynomial fit). The coefficient of determination obtained for the polynomial fit of degree 1 was $R^2 = 0.73$. A total of 91 accounts was used in this analysis.

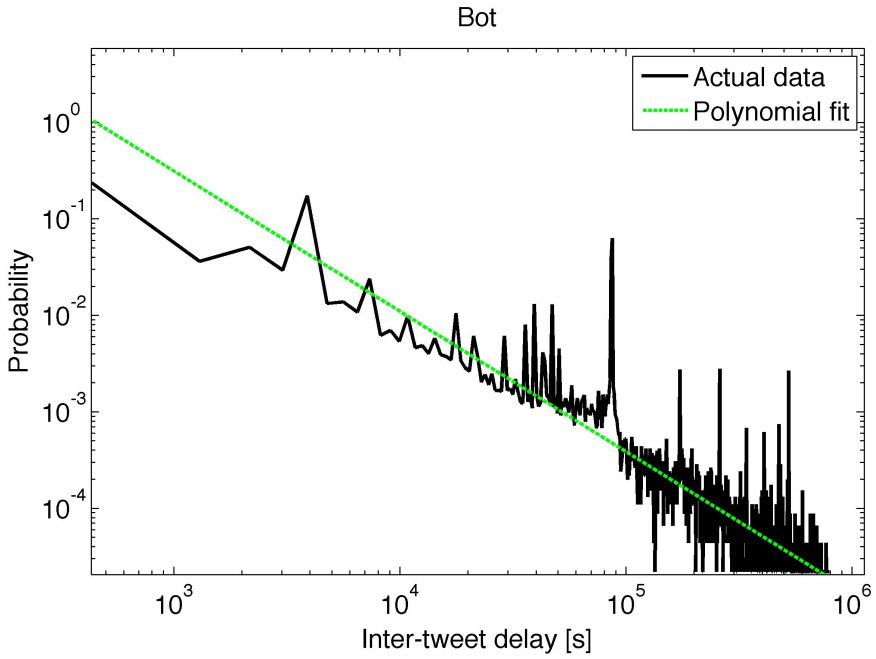
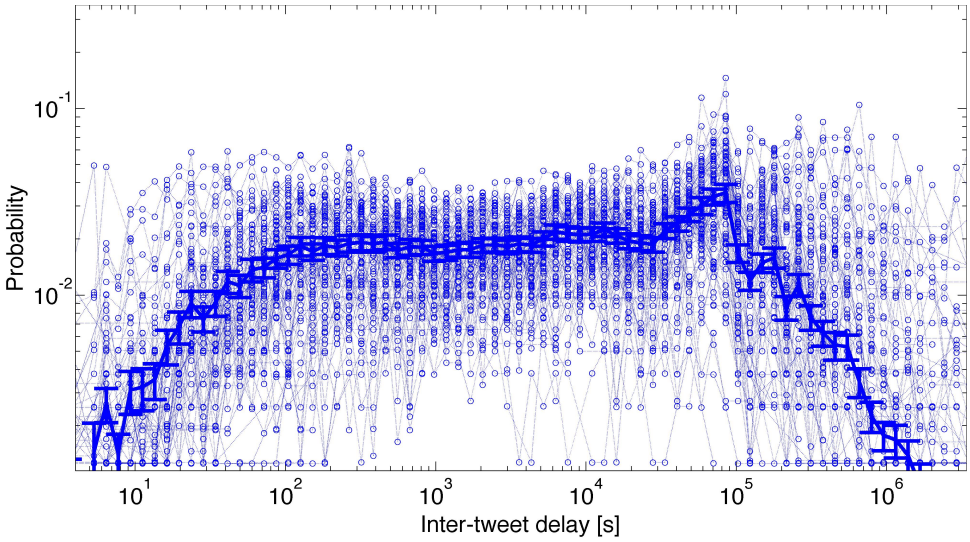
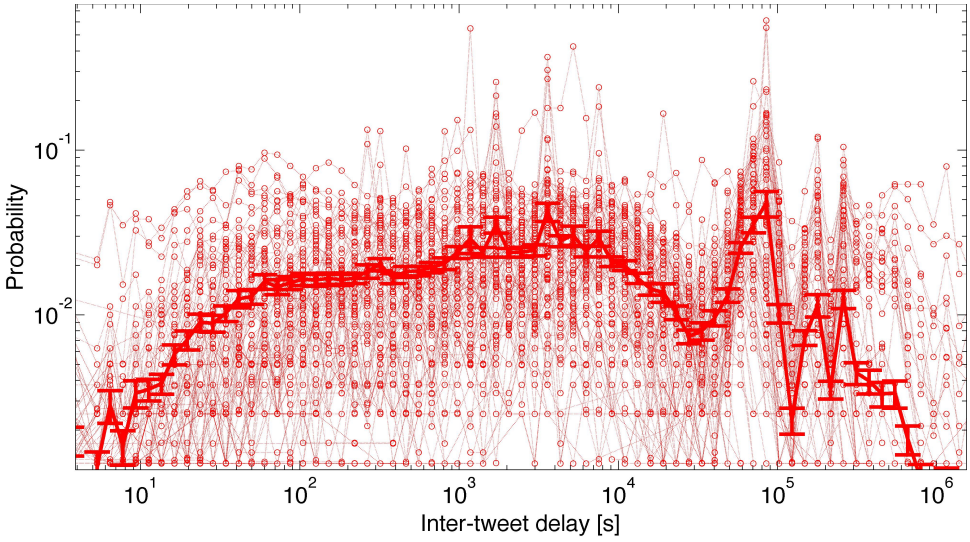


Figure 5.11: Probability density function for the inter-tweet delay of bot-controlled accounts. The distribution follows a power law with exponent approximately -1.45 (slope of the polynomial fit). The coefficient of determination obtained for the polynomial fit of degree 1 was $R^2 = 0.83$. A total of 67 accounts was used in this analysis.

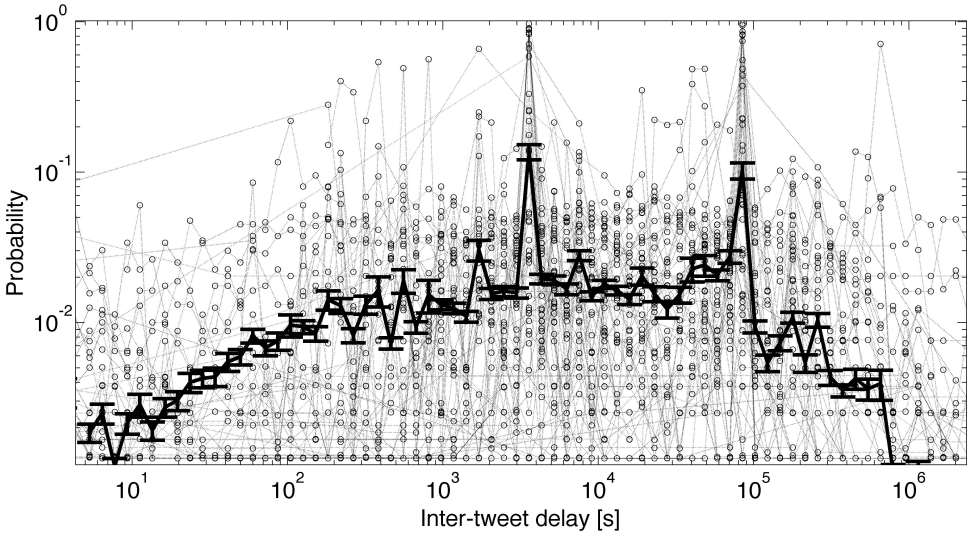
Figure 5.12 shows error bar plots for the inter-tweet delay distribution of each account class. In these plots, the dotted lines depict the probabilities for individual accounts, while the solid line depicts the mean for the whole class. The error bars represent the standard deviations. Since the distributions obtained for the three classes are very distinct, we were able to use these distributions in a naive Bayes classifier that differentiates between classes, as explained in chapter 6.



(a) Error bar plot for the inter-tweet delay of personal accounts (86 samples).



(b) Error bar plot for the inter-tweet delay of managed accounts (91 samples).



(c) Error bar plot for the inter-tweet delay of bot-controlled accounts (67 samples).

Figure 5.12: Error bar plots for the inter-tweet delay of personal (86 samples), managed (91 samples) and bot-controlled (67 samples) accounts. The dotted lines depict the probabilities for each individual account, while the solid line is the mean for the whole class. The error bars represent the standard deviations across the accounts analysed.

As explained in the beginning of this chapter, all tweet timestamps were adjusted in order to correspond to the local time zone of each user. Figure 5.13 shows the PDF's for the hour of tweets in classes Personal, Managed and Bot-controlled, respectively. We can see that personal accounts increase their tweeting level as the day progresses, peaking at 9pm. Managed accounts tend to tweet more during work hours, between 9am and 6pm. The dip in the distribution at 12pm can probably be explained by lunch hour breaks. From these findings we can conclude that the distribution of tweets throughout the day is related to a user's daily routine. Finally, the distribution for bot-controlled accounts exhibits a variety of peaks, which is probably because their behaviour is not associated with a structured daily routine.

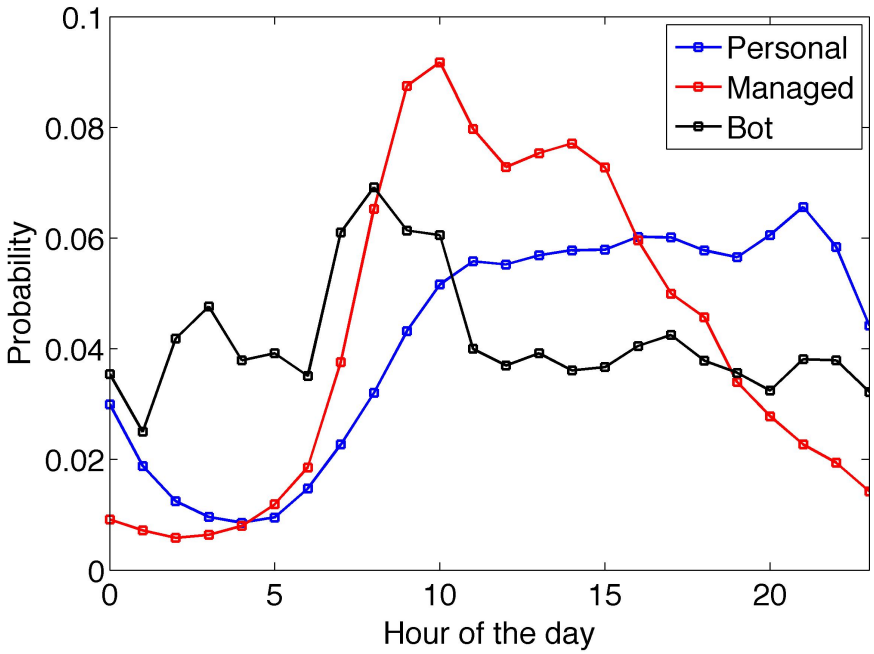
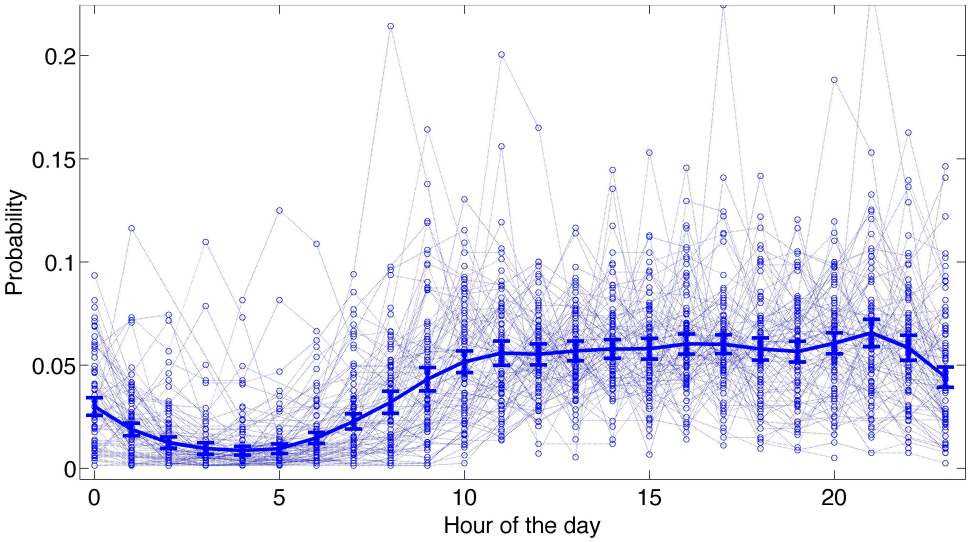
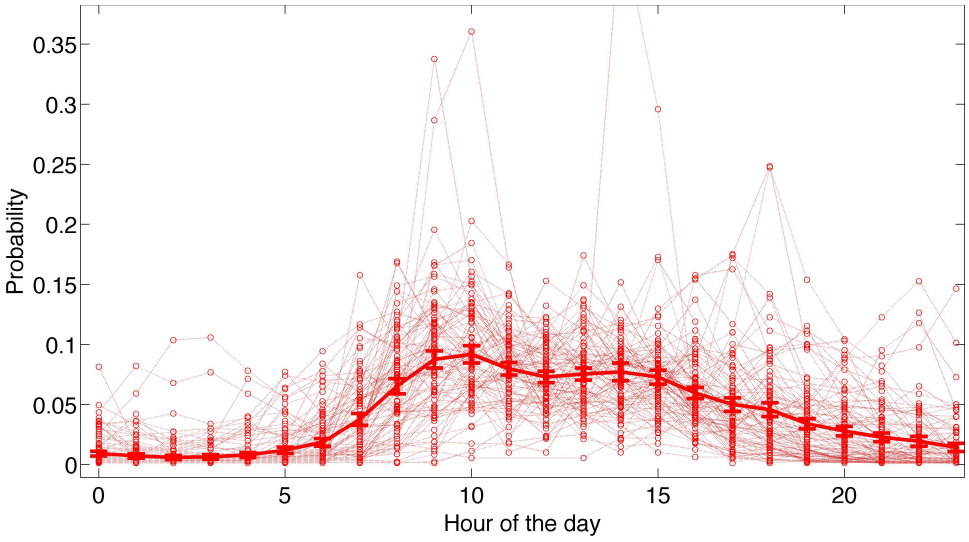


Figure 5.13: Probability density functions for the tweeting times of personal (86 samples), managed (91 samples) and bot-controlled (67 samples) accounts. The horizontal axis corresponds to the hours of the day, from 0 (midnight) to 23 (11pm). All timestamps were adjusted in order to correspond to the local time zone of each user.

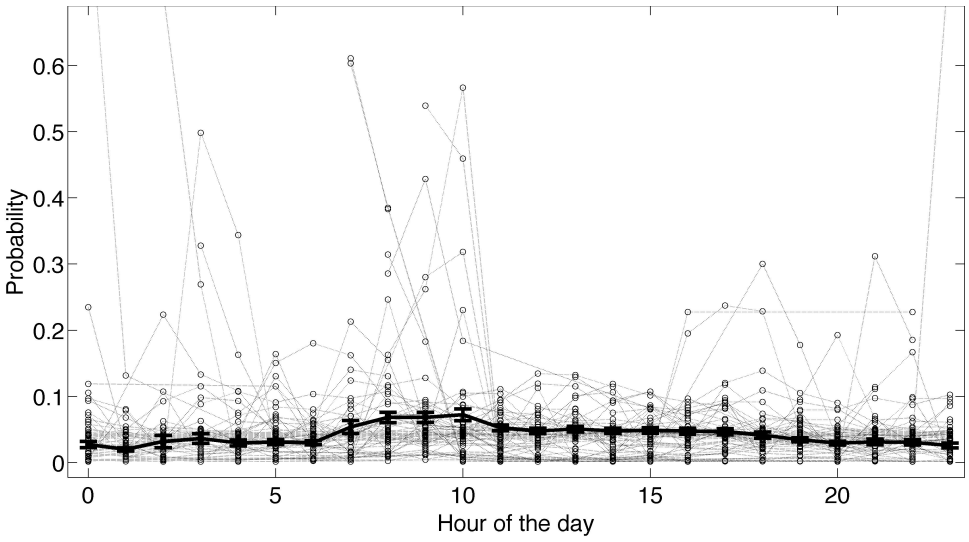
Figures 5.14 shows error bar plots for the tweeting times of each account class. In these plots, the dotted lines depict the probabilities for individual accounts, while the solid line depicts the mean for the whole class. The error bars represent the standard deviations.



(a) Error bar plot for thetweeting times of personal accounts (86 samples).



(b) Error bar plot for the tweeting times of managed accounts (91 samples).



(c) Error bar plot for the tweeting times of bot-controlled accounts (67 samples).

Figure 5.14: Error bars plot for the tweeting times of personal (86 samples), managed (91 samples) and bot-controlled (67 samples) accounts. The dotted lines depict the probabilities for each individual account, while the solid line is the mean for the whole class. The error bars represent the standard deviations across the accounts analysed.

We now study the individual behaviour of users in each account class. The hourly tweeting patterns for the 65 most active users of each class are shown in figure 5.15. In these plots, each tile is associated with a user and an hour of the day, and the tile's colour intensity is proportional to the amount of tweets posted by that user at that hour. We can clearly observe the differences in behaviour between the three classes: personal accounts tend to tweet more in the afternoons and evenings; managed accounts tweet more during work hours, from 8am to 6pm; and bot-controlled accounts either have a regular behaviour, tweeting at an approximately constant rate throughout the day, or display a low tweet rate with a very high peak at one or a few specific hours. Among the bot-controlled accounts, it is possible to detect behaviour that is characteristically programmed or scheduled: account number 40, for instance, exhibits a clear alternating pattern, tweeting every two hours.

These behavioural plots show that the tweeting patterns for both personal and managed accounts are intrinsically related to a real life daily routine. Bot-controlled accounts, on the other hand, exhibit an artificially designed behaviour. The very distinct patterns obtained for the three account classes allow us to use tweeting behaviour as a classification criterion for Twitter accounts, which is explained in chapter 6.

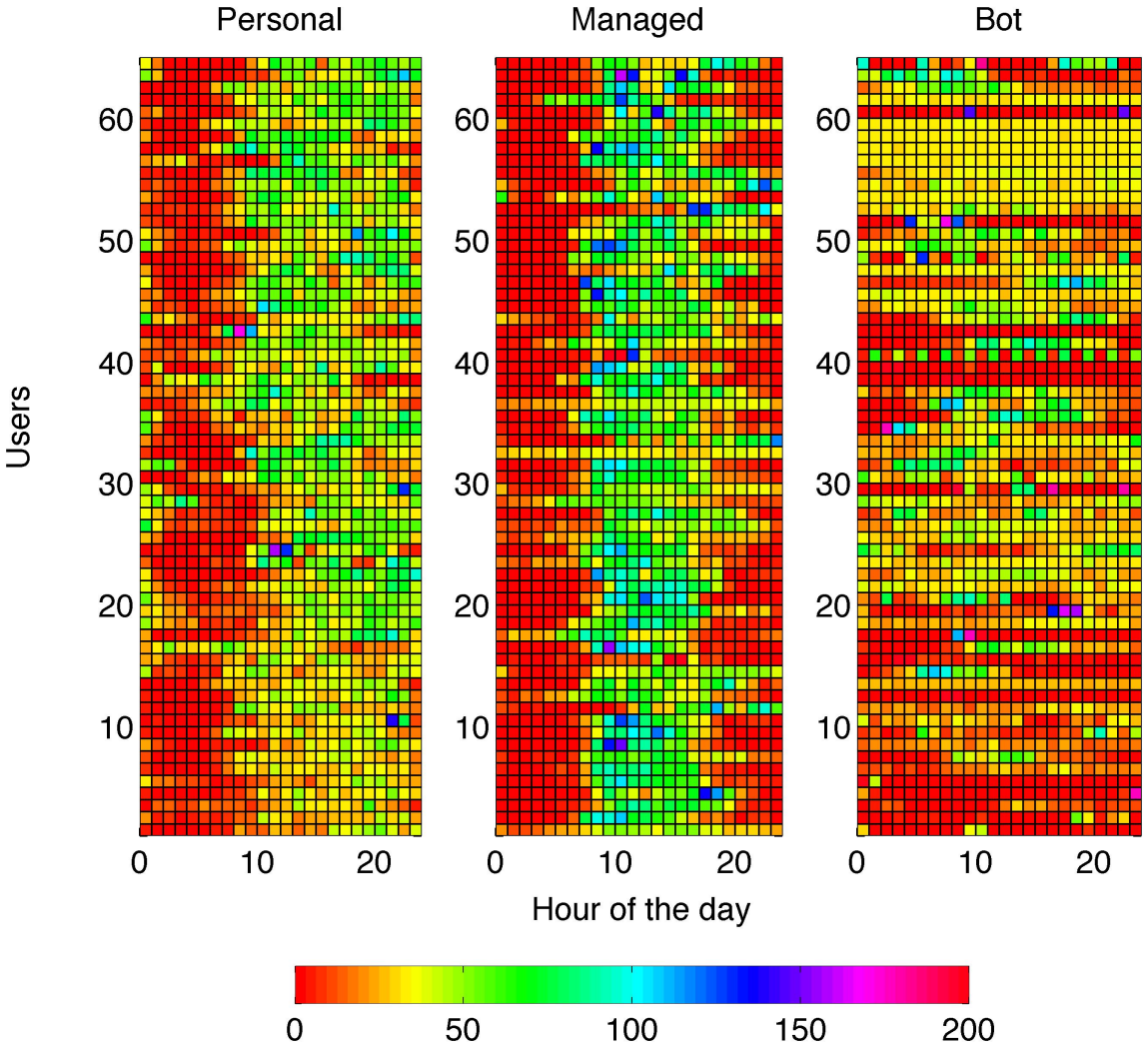


Figure 5.15: Number of tweets at each hour for each account class. Each row corresponds to a user and each column corresponds to an hour of the day. Users are sorted by increasing total number of tweets collected.

The weekly tweeting patterns for the same 65 users are shown in figure 5.16. In these plots, each tile is associated with a user and a day of the week, and the tile's colour intensity is proportional to

the amount of tweets posted by that user on that day. We can observe that personal accounts tend to maintain approximately the same amount of tweeting throughout the whole week, although the tweeting amount slightly drops in the weekends. This pattern could be indicative that tweeting is an activity used as pastime during work or school days. Managed accounts tweet significantly more during the work week (Monday to Friday) than in weekends, which is expected since tweeting in this case is a work-related activity. Finally, bot-controlled accounts present a diverse behaviour, with some accounts maintaining a uniform rate while others tweet more on one or a few specific days.

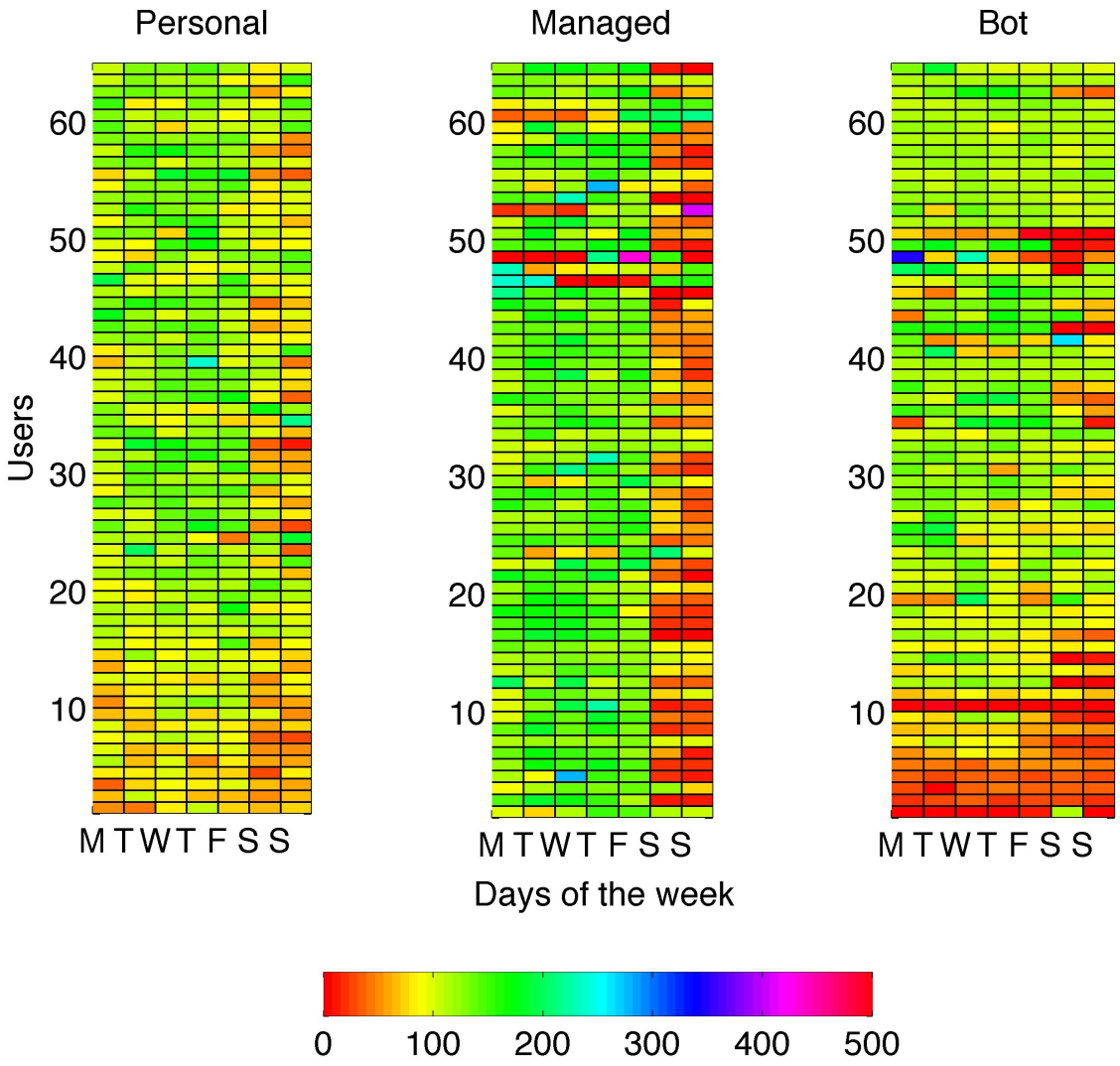


Figure 5.16: Number of tweets throughout the week for each account class. Each row corresponds to a user and each column corresponds to a day of the week. Users are sorted by increasing total number of tweets collected.

In this chapter, we have discussed the statistical analysis of both our uncategorised and our categorised Twitter datasets. We obtained probability distributions for number of tweets, number of followers and number of friends that are consistent with previous work. Furthermore, we have found that the inter-tweet delay follows a power law distribution, and that the distribution of tweets throughout the day and throughout the week is associated with users' daily routines.

Chapter 6

Methods

In this chapter, we describe the Machine Learning algorithms created for classification of Twitter accounts and prediction of tweeting behaviour. The theoretical concepts presented in chapter 2 are the building blocks for these algorithms. The main tool used for implementation was the MATLAB environment due to its aptitude for dealing with probability distributions and processing large amounts of data.

6.1 Naive Bayes Classifiers for Twitter Accounts

We begin by describing the two naive Bayes classifiers created. Our classification system is based on equation 2.2.7 from section 2.2, in which we discussed the naive Bayes classifier. Based on that equation, we obtain the following maximum a posteriori (MAP) decision rules for our naive Bayes classifiers:

$$\text{class}(d) = \arg\max_c \{p(\text{delay} = d|C = c)\}\,,\ c \in \{P, M, B\} \tag{6.1.1}$$

$$\text{class}(t) = \arg\max_c \{p(\text{time} = t|C = c)\}\,,\ c \in \{P, M, B\} \tag{6.1.2}$$

$$\text{class}(d, t) = \arg\max_c \{p(\text{delay} = d|C = c) \times p(\text{time} = t|C = c)\}\,,\ c \in \{P, M, B\} \tag{6.1.3}$$

where P, M and B correspond to our three account classes, personal, managed and bot, respectively. The first equation uses only the marginal distribution for inter-tweet delay, the second equation uses only the marginal distribution for tweeting time, and the third equation uses the joint distribution of both variables assuming independence between them.

The 2-Classifier

The 2-Classifier distinguishes between account classes Personal and Managed. Four attempts of classification were made: using only inter-tweet delay distributions; using only tweeting time distributions; using both properties as independent variables; and using both properties as non-independent variables. Due to the small number of samples obtained for each class, we applied leave-one-out cross validation instead of using separate training and testing datasets. In leave-one-out cross validation, with N samples in the dataset, the classification is performed as follows: the model is constructed N times, and at each time one of the samples is left out while the other $N-1$ samples are used for construction of the model. The sample that was not used in the model is then classified based on the probability distributions given by the model samples.

In each cross-validation loop, the $N-1$ model samples are grouped into their respective classes, and then four different probability density functions for each of the two classes is computed, namely, the marginal distributions for inter-tweet delay (ITD), the marginal distribution for tweeting times

(TT), the joint distribution assuming independent variables (JI), and the joint distribution assuming non-independent variables (JNI). The marginal distributions are created in the same way as shown in the data analysis perfumed in chapter 5. The joint distribution assuming independent variables is obtained by simply multiplying the values of the two marginal distributions. Finally, the joint distribution assuming non-independent variables is obtained as a three-dimensional surface, where the first two dimensions are given by the values of inter-tweet delay and tweet time, and the third dimension corresponds to the joint probability.

To classify the left-out sample, the values (inter-tweet delay and timestamp hour) of each one of that sample's tweets are interpolated into the distributions of both classes. For each of the four attempts, the classification score $S^c(i)$ of class c for sample i is computed as:

$$S^c(i) = \sum_{t \text{ in } T(i)} \log(\text{interpolate}(t, \text{pdf}(c))) \tag{6.1.4}$$

where $T(i)$ is the set of tweets for sample account i, and $\text{interpolate}(t, \text{pdf}(c))$ is the interpolation of the value of t into the probability density function of class c. Once both class scores have been computed, sample i is classified into the class with highest classification score.

Since scores are computed separately for each classification attempt, a different outcome is obtained for each attempt, resulting in four different classification outcomes for each sample. The pseudocode showing the outline of this algorithm is presented below.

```
for subj = 1 to size(Samples)
     testSample = Samples[subj]
     sumPersonal[subj] = 0
     sumManaged[subj] = 0
     pdfPersonal = getPDF(Samples[i] where i!=subj and Samples[i] in Personal)
     pdfManaged = getPDF(Samples[i] where i!=subj and Samples[i] in Managed)
     for tweet = 1 to size(testSample)
         probPersonal[subj,tweet] = interpolate(testSample[tweet],pdfPersonal)
         sumPersonal[subj] += log(probPersonal[subj,tweet])
         probManaged[subj,tweet] = interpolate(testSample[tweet],pdfManaged)
         sumManaged[subj] += log(probManaged[subj,tweet])
     end for
     if sumPersonal[subj] > sumManaged[subj]
         testSample classified as Personal
     else
         testSample classified as Managed
end for
```

The 3-Classifier

The implementation of the 3-Classifier is analogous to that of the 2-Classifier, the only difference being the number of classes used. The increase in the number of classes from 2 to 3 caused classification results to be slightly worse, as explained in the next chapter. In order to evaluate the robustness of the classification algorithms regarding the amount of training data available, we tested both classifiers with varying dataset sizes. The plots showing the percentage of correct classification as a function of the number of samples used in the can be seen in figures 6.1 and 6.2. These plots indicate that the use of larger datasets might improve the classification results, especially when using joint probability distributions. The results obtained with the different classification attempts for both classifiers are presented in chapter 7.

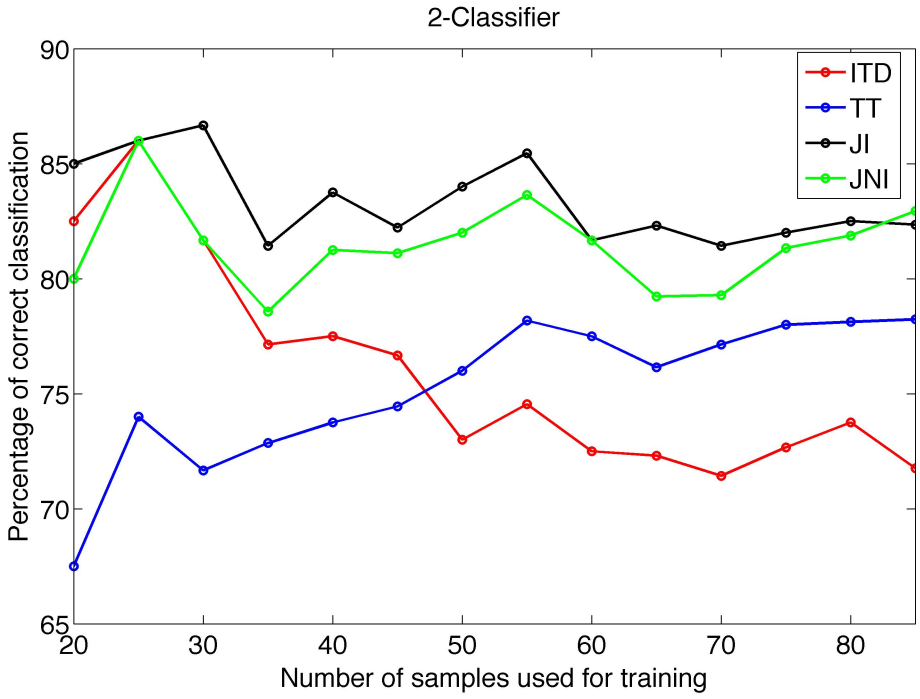


Figure 6.1: Correct classification percentage vs. number of samples for the 2-Classifier.

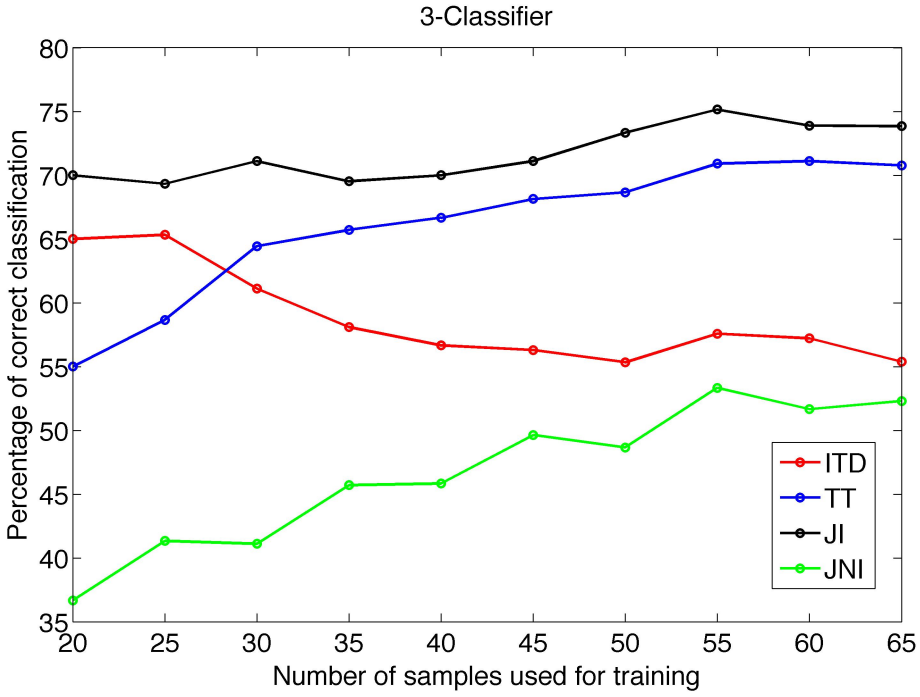


Figure 6.2: Correct classification percentage vs. number of samples for the 3-Classifier.

6.2 Predictive Model for Tweeting Time

Our following step was to try to predict when a user's next tweet would be posted, based on the inter-tweet delay and tweeting time distributions of that user's account class. We perform probabilistic prediction by computing a cumulative distribution function of the probability that a tweet would happen t seconds after the previous tweet. As in the classifier algorithms, we used leave-one-out cross validation due to the small size of the categorised dataset.

In our first attempt for prediction, we simply used the inter-tweet delay distribution of one particular class in order to generate a corresponding cumulative distribution function. The CDF in terms of the inter-tweet delay t describes the probability that a tweet will occur given that t seconds have passed since the last tweet. As before, at each cross validation loop, all samples are used in the computation of the probability distribution, with the exception of one left-out sample. We then take the actual inter-tweet delay we want to predict (among the left-out sample's tweets) and use that delay to compute a step function as follows:

$$\text{step}(t) = \begin{cases} 0 & \text{if } t < \tau \\ 1 & \text{if } t \geq \tau \end{cases} \tag{6.2.1}$$

where τ is the actual inter-tweet delay of the left-out sample, which we aim to predict. This step function represents the observed cumulative probability of a tweet occurring τ seconds after the last tweet: because the tweet occurred exactly after τ seconds, this probability is 0 before τ, and 1 after τ. For each tweet of the sample user account, a different step function is computed and then compared to the CDF generated from the class probability distribution. As an illustrative example, the comparison between the class CDF and ten different step functions for the same user in shown in figure 6.3. Finally, we compute the coefficient of determination in order to evaluate the predictive model: for each account, we take all step functions generated and check how well the probability distribution of the class fits to each of them. The pseudocode for the single distribution prediction algorithm, for an arbitrary class c, is presented below.

```
for subj = 1 to size(Samples_c)
     testSample = Samples_c[subj]
     pdf = getPDF(Samples_c[i] where i!=subj)
     for tweet = 1 to size(testSample)
         delay = testSample[tweet].delay
         prob[subj,tweet] = interpolate(delay, pdf)
         add point (delay, prob[subj,tweet]) to pdf
         cdf = getCDF(pdf)
         step(subj,tweet,t) = 0 if t < delay; 1 otherwise
         compute R_squared for cdf and step(subj,tweet,t)
     end for
end for
```

In a slightly more elaborated version of the predictor, we used the same predictive model but with separate inter-tweet delay distributions for each hour of the day. Each inter-tweet delay is associated with an hour based on the timestamp of the tweet that occurred before that delay, resulting in 24 different probability distributions for the inter-tweet delay. This is reasonable since the length of the time delay between consecutive tweets is very likely related to the time of the day when those tweets occur. After computing the 24 distributions, we select which distribution to use according to the timestamp of the sample user's last tweet. Again, we compute a step function to represent the observed probability of the inter-tweet delay, then use the coefficient of determination

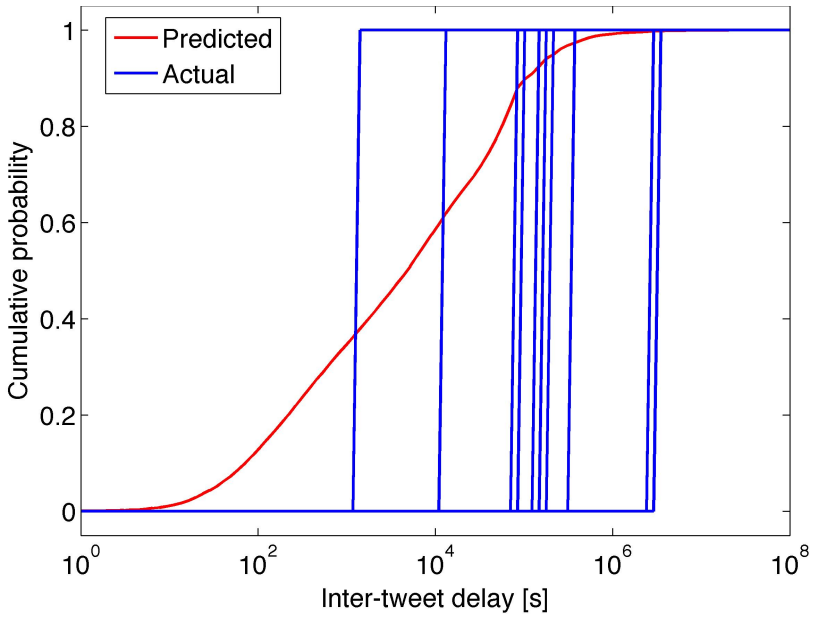


Figure 6.3: Comparison between the predicted cumulative probability function for inter-tweet delay (in red) and ten different step functions for the same user account (in blue), which correspond to the actual cumulative probability functions for the inter-tweet delay of ten different tweets.

to evaluate the prediction. The pseudocode for the multiple distribution predictor, for an arbitrary class c, is presented below.

```
for subj = 1 to size(Samples_c)
     testSample = Samples_c[subj]
     for h = 0 to 23
          pdf[h] = getPDF(Samples_c[i] where i!=subj and Samples_c[i].timestamp==h)
     end for
     for tweet = 1 to size(testSample)
          delay = testSample[tweet].delay
          hour = testSample[tweet].timestamp
          prob[subj,tweet] = interpolate(delay, pdf[hour])
          add point (delay, prob[subj,tweet]) to pdf[hour]
          cdf[hour] = getCDF(pdf[hour])
          step(subj,tweet,t) = 0 if t < delay; 1 otherwise
          compute R_squared for cdf[hour] and step(subj,tweet,t)
     end for
end for
```

We also created a third version of the predictive model, this time using a separate probability distribution for each hour of the day and for each day of the week, resulting in a total of $24 \times 7 = 168$ distributions. Each inter-tweet delay was associated with the hour and the day of the week of the last tweet occurring before the delay. We then used this information to select which distribution to use in the prediction of each particular tweet. The results obtained for the three different predictive algorithms implemented are described in chapter 7.

Chapter 7

Results

In this chapter we demonstrate the results obtained with the Machine Learning algorithms that were described in chapter 6: the two classifier algorithms for Twitter accounts and the three predictive models for next tweet time.

7.1 Automatic Recognition of User Account Types

We begin by examining the confusion matrices generated by the 2-Classifier and by the 3-Classifier. As explained in chapter 6, four attempts of classification were made: using only inter-tweet delay distributions (ITD); using only tweeting time distributions (TT); using the joint distribution of both properties as independent variables (JI); and using the joint distribution of both properties as non-independent variables (JNI). Each of these attempts has its own confusion matrix. In each confusion matrix, the columns correspond to the predicted classes, while the rows correspond to the actual classes of the samples. Therefore, the diagonals of the confusion matrices display the number of samples in each class that were classified correctly.

Tables 7.1 to 7.4 show the confusion matrices of the 2-Classifier, while table 7.5 displays the percentage of correct classifications in each of the four trials. Comparing tables 7.1 and 7.2, we can see that using the marginal distribution for tweeting time yielded better results than using the marginal distribution for inter-tweet delay. This is reasonable since the tweeting time distributions for each class, presented in figure 5.13, exhibit particularly distinct shapes. From tables 7.3 and 7.4, we can conclude that the naive Bayes classifier using the joint distribution of the two variables generated better results than the classifier with the non-independence assumption. We believe that this is due to the small number of samples used for training the model.

		Predicted class:		
		Personal	Managed	Total
Actual class:	Personal	68	18	86
	Managed	31	55	86

Table 7.1: Confusion matrix obtained with the 2-Classifier, using inter-tweet delay marginal probability distribution.

		Predicted class:		
		Personal	Managed	Total
Actual class:	Personal	70	16	86
	Managed	21	65	86

Table 7.2: Confusion matrix obtained with the 2-Classifier, using tweeting time marginal probability distribution.

		Predicted class:		
		Personal	Managed	Total
Actual class:	Personal	73	13	86
	Managed	16	70	86

Table 7.3: Confusion matrix obtained with the 2-Classifier, using the joint probability distribution of inter-tweet delay and tweeting time, assuming independent variables.

		Predicted class:		
		Personal	Managed	Total
Actual class:	Personal	72	14	86
	Managed	15	70	85

Table 7.4: Confusion matrix obtained with the 2-Classifier, using the joint probability distribution of inter-tweet delay and tweeting time, assuming non-independent variables.

2-Classifier	Correctness
ITD	71.5%
TT	78.5%
JI	83.1%
JNI	82.6%

Table 7.5: Correct classification percentage for the 2-Classifier.

The total of samples used in the 2-Classifier was 86, since we discarded accounts for which the time zone information was not available. In the last confusion matrix (table 7.4), the total number of classified samples is shown to be less than the actual total. This is because, in some cases, the classifier was not be able to make a decision for one or more samples. As one would expect, using both inter-tweet delay and tweeting time properties as classification criteria yielded better results than using only one of them. Even though the assumption that these properties were independent variables led to better results than the non-independence assumption, it is likely that the deterioration in correctness caused by the non-independence assumption is due to subsampling: with a small amount of samples, the interpolation of sample values into the three-dimensional joint distribution is very poor. This argument is corroborated by the plot in figure 6.1, which shows that the correct classification percentage for the JNI distribution is still increasing just before the final test, which used 85 samples. Therefore, a larger dataset could potentially improve the classification results obtained with the joint distribution.

Tables 7.6 to 7.9 show the confusion matrices of the 3-Classifier, while table 7.10 displays the percentage of correct classifications in each of the four trials. As expected, the 3-Classifier performed slightly worse than the 2-Classifier, due to its larger number of classes. From tables 7.6 and 7.7, we see that again the tweeting time marginal distribution led to better classification results than the inter-tweet delay distribution, and that in the 3-Classifier this difference was even more pronounced than in the 2-Classifier. Similarly, from tables 7.8 and 7.9, we see that once again the variable independence assumption yielded better results than the non-independence assumption. In both cases, JI and JNI, 3 samples could not be classified due to poor interpolation into one of the class joint distributions.

The total of samples used in the 3-Classifier was 67. For this algorithm, results obtained when assuming that inter-tweet delay and tweeting time were non-independent variables were considerably worse than the results obtained with the independence assumption. As in the 2-Classifier, this is probably largely due to subsampling, which causes poor interpolation of the test samples into the three-dimensional probability distribution. The classification correctness percentage we have obtained with the JI distribution is only slightly worse than those obtained by previous researchers [10, 2, 12], with one important advantage: unlike previous work, our classification algorithm is based solely on tweeting behaviour and does use any other account feature or require parsing of tweet contents.

		Predicted class:			
		Personal	Managed	Bot	Total
Actual class:	Personal	38	22	7	67
	Managed	17	40	10	67
	Bot	18	18	31	67

Table 7.6: Confusion matrix obtained with the 3-Classifier, using inter-tweet delay marginal probability distribution.

		Predicted class:			
		Personal	Managed	Bot	Total
Actual class:	Personal	52	11	4	67
	Managed	13	45	9	67
	Bot	11	11	45	67

Table 7.7: Confusion matrix obtained with the 3-Classifier, using tweeting time marginal probability distribution.

		Predicted class:			
		Personal	Managed	Bot	Total
Actual class:	Personal	56	7	3	66
	Managed	14	47	6	67
	Bot	13	8	44	65

Table 7.8: Confusion matrix obtained with the 3-Classifier, using the joint probability distribution of inter-tweet delay and tweeting time, assuming independent variables.

		Predicted class:			
		Personal	Managed	Bot	Total
Actual class:	Personal	37	26	3	66
	Managed	17	41	9	67
	Bot	15	22	28	65

Table 7.9: Confusion matrix obtained with the 3-Classifier, using the joint probability distribution of inter-tweet delay and tweeting time, assuming non-independent variables.

3-Classifier Correctness	
ITD	54.2%
TT	70.6%
JI	73.1%
JNI	52.7%

Table 7.10: Correct classification percentage for the 3-Classifier.

7.2 Prediction of Next Tweet Time

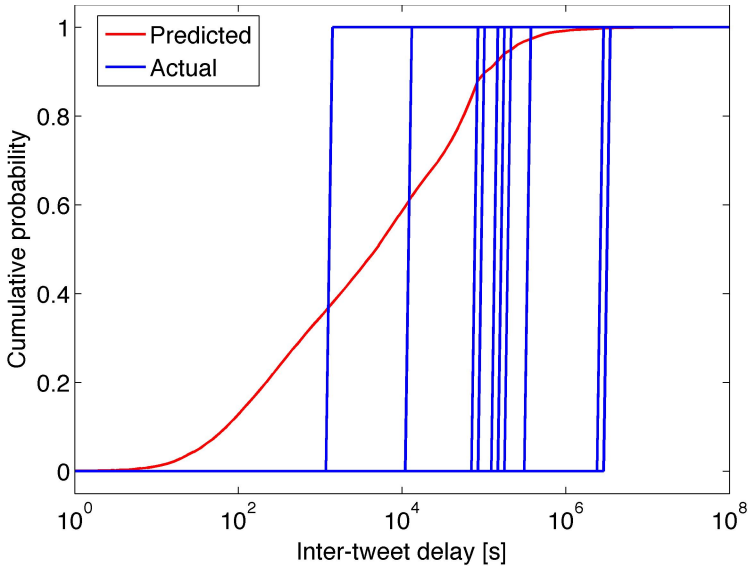
For the prediction algorithms, we used the coefficient of determination, R^2, as a goodness of fit measure of the predictive models created. As explained in the previous chapter, the actual data is represented by a step function, while the model data is represented by a cumulative distribution function, both describing the cumulative probability of a tweet being posted t seconds after the previous tweet occurred. Figure 7.1(a) shows a comparison between the cumulative distribution function (the model) and the step functions generated for 10 sample tweets. Figure 7.1(b) shows a scatter plot for all points of the same 10 sample tweets, with the horizontal axis corresponding to the value of the CDF (predicted value) and the vertical axis corresponding to the value of the step function (actual value). This example scatter plot shows the points for which the coefficient of determination was computed in order to evaluate the model.

A total of 60 samples from each class was used in the prediction algorithms. Table 7.11 shows the average R^2 obtained for each account class by the three predictive models constructed: the first one using a single probability distribution, the second one using a separate probability distribution for each hour of the day (H), and the third one using a separate probability distribution for each hour of the day and each day of the week (HW). The results obtained for the coefficient of determination are in a good range for human generated data. From table 7.11 we can see that the use of separate distributions for each hour and each day of the week was especially beneficial for the prediction of tweets by bot-controlled accounts. This is because bot-controlled accounts have a programmed behaviour which is typically hour or day specific.

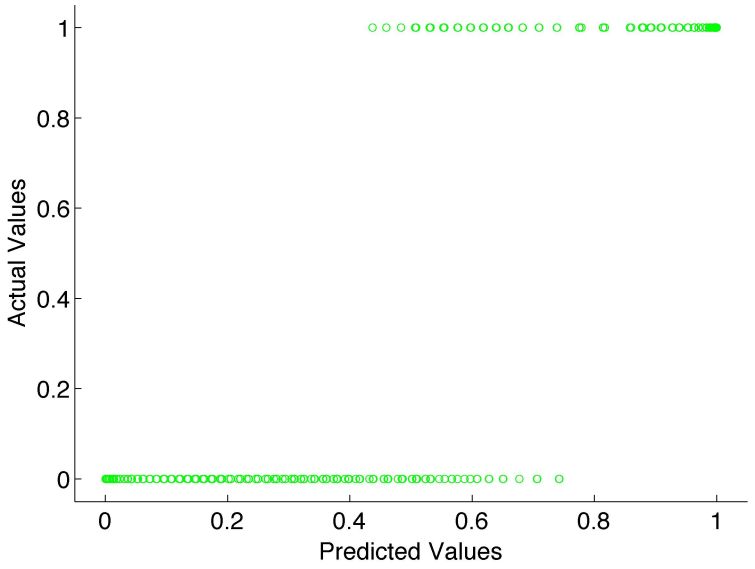
In order to evaluate the statistical significance of our results, we applied the same predictive models to randomly generated data. We used a pseudo-random number generator, drawing numbers from a uniform distribution over range 1 to 1,000,000. In these tests, the average R^2 obtained was 0.33, which is much lower than any of the values obtained for the real data. We can conclude that our results are statistically significant, but can potentially be improved by the use of additional information about the tweeting patterns observed.

	Average R^2		
	Personal	Managed	Bot
Single Distribution	0.661	0.716	0.518
Multiple Distribution (H)	0.662	0.723	0.574
Multiple Distribution (HW)	0.667	0.725	0.676

Table 7.11: Average coefficient of determination obtained for each class by the three probabilistic prediction models.



(a) Comparison between the predicted cumulative probability function for inter-tweet delay (in red) and 10 different step functions for the same user account (in blue), which correspond to the actual cumulative probability functions for the inter-tweet delay of 10 different tweets.



(b) Scatter plot for all points of the same 10 sample tweets, with the horizontal axis corresponding to the value of the CDF (predicted value) and the vertical axis corresponding to the value of the step function (actual value).

Figure 7.1: Plots used in the computation of the coefficient of determination for the predictive algorithms.

Chapter 8

Discussion and Conclusion

In this project, we have used data collected from the online social network Twitter in order to study the behaviour of different types of user accounts. Three different classes of Twitter accounts were studied: personal accounts, belonging to a single individual; managed accounts, belonging to a corporation; and bot-controlled accounts, which are administered by a computer program and therefore receive no human input. In order to collect the data, we created Creepy Crawly, a Twitter crawler application which retrieves account information and tweet feeds from either a pre-specified set of accounts or from the social network surrounding a seed user. The dataset collected through Creepy Crawly was used in two different types of Machine Learning algorithms: account classifiers, with the aim of distinguishing between the three account classes in the dataset, and predictive models, with the aim of determining when the next tweet of a user would be posted.

In the analysis of our uncategorised dataset, consisting of over 10,000 users, we studied the Twitter graph, tweet frequency and the probability distributions for three account features, namely, number of tweets, number of followers and number of friends. We obtained through this analysis results consistent with those reported in previous work.

For our categorised dataset, we examined the inter-tweet delay distributions and the tweet frequency variation throughout different hours of the day and different days of the week, for each of the account classes studied. We were able to observe that personal, managed and bot-controlled accounts present very distinct tweeting patterns, and as a result these patterns could be used to distinguish between the classes in an automated manner. When studying the inter-tweet delay, i.e., the time interval between two consecutive tweets by the same user, we found that this measure follows a power-law distribution. This finding is in accordance with the findings of many other studies in Computational Social Science, and reinforces the idea that a bursty, fat-tailed behaviour is characteristic of the time of many human actions. Furthermore, we found that the distribution of a user's tweets throughout the day and throughout the week is closely related to that user's daily routine. In the tweeting patterns studies, we were able to detect, for instance, evidence for work schedules, weekends and lunch hour breaks.

Using the categorised dataset, we created two classification algorithms based on probability distributions, the first algorithm to classify only personal and managed accounts, and the second one to classify all three types of accounts studied. Both classifiers performed well, with the best results being generated by the use of joint probability distributions of inter-tweet delays and tweet times. When compared to previous research, our classification results were only slightly worse, but with two important advantages: first, we do not determine a priori what behaviours or features are characteristic of each class; and second, we not use any profile attribute or tweet content in order to perform classification.

Additionally, we implemented three different predictive models in order to attempt predicting when the next tweet of a user would be posted. In our first attempt at probabilistic prediction, we used only the inter-tweet delay distribution of a given class in order to predict the next delay for a user of the same class. We then tried to improve our results by using separate distributions for each hour of the day and for each day of the week. Our predictive models produced statistically significant results, with all coefficients of determination highly surpassing those of randomly gener-

ated data. To the best of our knowledge, no previous work has been performed in studying human behaviour through the prediction of tweeting activity.

Achievements

From a software development perspective, in this project we have successfully implemented a Twitter crawler application and a total of five Machine Learning algorithms that aim to study the behaviour of Twitter users through their tweeting patterns. With our classifier algorithms we were able to distinguish between personal, managed and bot-controlled accounts, which allows users to know with whom they are interacting on Twitter and can potentially aid spam detection. Furthermore, we believe that this work is a small but significant contribution to the field of Computational Social Science, in that it uses the digital traces of Twitter accounts in order to study the behaviour of its users. We have obtained results that extend the findings of many previous studies on the bursty, heavy-tailed character of distributions pertaining to human actions.

Future Work

This project leaves room for many future advances in the use of Twitter data for studying human behaviour. The Creepy Crawly application can be easily adapted to collect any kind of dataset from Twitter. Ultimately, our goal is to be able to predict even more information about Twitter users based on their tweeting behaviour. This information could include, for instance, a user's gender, nationality and age group. If it were possible to collect real-life information from users, such as their personality traits and their real friendship networks, many additional studies could be conducted in order to find the correlation between behaviour on Twitter and aspects of the users' real lives. Moreover, by collecting data from specific user classes, in the same way as we did for personal, managed and bot-controlled accounts, one could create classifier algorithms for distinguishing between different genders, nationalities, age groups, and many other user categories.

Bibliography

[1] Y. Bachrach, M. Kosinski, T. Graepel, P. Kohli, and D. Stillwell. Personality and patterns of Facebook usage. In *ACM Web Science*, 2012.

[2] V. A. Balasubramaniyan, A. Maheswaran, V. Mahalingam, M. Ahamad, and H. Venkateswaran. A crow or a blackbird?: Using true social network and tweeting behavior to detect malicious entities in Twitter. 2010.

[3] A. L. Barabasi. The origin of bursts and heavy tails in human dynamics. *Nature*, 435(7039):207–211, 2005.

[4] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[5] N. Boccara. *Modeling Complex Systems*. Springer, 2010.

[6] J. Bollen, H. Mao, and X. Zeng. Twitter mood predicts the stock market. *Journal of Computational Science*, 2:1–8, 2011.

[7] J. Bollen, A. Pepe, and H. Mao. Modeling public mood and emotion: Twitter sentiment and socio-economic phenomena. *arXiv:0911.1583*, 2009.

[8] J. Bosari. The developing role of social media in the modern business world. Forbes, www.forbes.com/sites/moneywisewomen/2012/08/08/the-developing-role-of-social-media-in-the-modern-business-world/, August 2012.

[9] L. Brousell. How Citibank uses Twitter to improve customer service. CIO, www.cio.com/article/708709/How_Citibank_Uses_Twitter_to_Improve_Customer_Service, June 2012.

[10] M. C. Calzolari. Analysis of Twitter followers of leading international companies: Quantitative and qualitative study of behaviours demonstrated by humans (users which are presumably real) or by bots (users which are presumably fake). Technical report, IULM University of Milan, 2012.

[11] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *Fourth International AAAI Conference on Weblogs and Social Media (ICWSM)*, pages 10–17, 2010.

[12] Z. Chu, S. Gianvecchio, H. Wang, and S. Jajodia. Who is tweeting on Twitter: human, bot, or cyborg? In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 21–30. ACM, 2010.

[13] M. H. DeGroot. *Probability and Statistics*. Addison-Wesley Pub. Co., 1975.

[14] Z. Dezsö, E. Almaas, A. Lukács, B. Rácz, I. Szakadát, and A.L. Barabási. Dynamics of information access on the web. *Physical Review E*, 73(6):066132, 2006.

[15] N. Eagle and A. S. Pentland. Eigenbehaviors: Identifying structure in routine. *Behavioral Ecology and Sociobiology*, 63(7):1057–1066, 2009.

[16] N. Eagle, A. S. Pentland, and D. Lazer. Inferring friendship network structure by using mobile phone data. *Proceedings of the National Academy of Sciences*, 106(36):15274–15278, 2009.

[17] N.B. Ellison et al. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication*, 13(1):210–230, 2007.

[18] A. Faisal. *Modelling in Biology II: Biological Networks and Stochastic Processes in Biology.* 2011.

[19] J. Giles. Computational social science: making the links. Nature, www.nature.com/news/computational-social-science-making-the-links-1.11243#/b3, August 2012.

[20] K.I. Goh and A.L. Barabási. Burstiness and memory in complex systems. *EPL (Europhysics Letters)*, 81:48002, 2008.

[21] J. Haushofer, A. Biletzki, and N. Kanwisher. Both sides retaliate in the Israeli-Palestinian conflict. *Proc Natl Acad Sci USA*, 107:17927–17932, 2010.

[22] B. Huberman, D. Romero, and F. Wu. Social networks that matter: Twitter under the microscope. 2008.

[23] A. Java, X. Song, T. Finin, and B. Tseng. Why we Twitter: understanding microblogging usage and communities. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pages 56–65. ACM, 2007.

[24] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th international conference on World Wide Web*, pages 591–600. ACM, 2010.

[25] D. Lazer, A. S. Pentland, L. Adamic, S. Aral, A. L. Barabasi, D. Brewer, N. Christakis, N. Contractor, J. Fowler, M. Gutmann, et al. Life in the network: the coming age of Computational Social Science. *Science (New York, NY)*, 323(5915):721, 2009.

[26] C. Lumezanu, N. Feamster, and H. Klein. # bias: Measuring the tweeting behavior of propagandists. In *Sixth International AAAI Conference on Weblogs and Social Media (ICWSM)*, 2012.

[27] D. J. C. MacKay. *Information Theory, Inference & Learning Algorithms.* Cambridge University Press, 2002.

[28] T. Neiman and Y. Loewenstein. Reinforcement learning in professional basketball players. *Nat. Commun.*, 2:569, 2011.

[29] B. O'Connor, R. Balasubramanyan, B. R. Routledge, and N. A. Smith. From tweets to polls: Linking text sentiment to public opinion time series. *Proceeding of the Fourth International AAAI Conference on Weblogs and Social Media (ICWSM)*, 2010.

[30] J. G. Oliveira and A. L. Barabási. Human dynamics: Darwin and Einstein correspondence patterns. *Nature*, 437(7063):1251–1251, 2005.

[31] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[32] W. Pan, M. Cebrian, W. Dong, T. Kim, J. Fowler, and A. Pentland. Modeling dynamical influence in human interaction patterns. *arXiv:1009.0240v5*, 2012.

[33] A. Parker. In nonstop whirlwind of campaigns, Twitter is a critical tool. The New York Times, www.nytimes.com/2012/01/29/us/politics/twitter-is-a-critical-tool-in-republican-campaigns.html?pagewanted=all, January 2012.

[34] M. J. Paul and M. Dredze. You are what you tweet: Analyzing Twitter for public health. *Proceedings of the Fifth International AAAI Conference on Weblogs and Social Media (ICWSM)*, 2011.

[35] A. Resulaj, R. Kiani, D. M. Wolpert, and M. N. Shadlen. Changes of mind in decision-making. *Nature*, 461:263–266, 2009.

[36] C. Song, T. Koren, P. Wang, and A. L. Barabási. Modelling the scaling properties of human mobility. *Nature Physics*, 6(10):818–823, 2010.

[37] E. Stepanova. The role of information communication technologies in the Arab Spring. *Implications beyond the Region. Washington, DC: George Washington University (PONARS) Eurasia Policy Memo no. 159*, 2011.

[38] J. Weng, E. P. Lim, J. Jiang, and Q. He. Twitterrank: finding topic-sensitive influential twitterers. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, pages 261–270. ACM, 2010.

Appendix A

Creepy Crawly (Unrestricted)

```
import twitter
import databaseAccess_unrestricted
import rateLimiter_unrestricted
import errorReport_unrestricted

QUIET=0
STDOUTPUT=1
VERBOSE=2
DEBUG=3


def main(seeduser='gabioptavares',recentusers=1000,**kwargs):

    print 'Main arguments: ', kwargs

    verbosity = kwargs.get('verbosity')

    # Create log file
    log = errorReport_unrestricted.getErrorReport()
    log.startLog()

    # Connect to database and create tables
    dbAccess = databaseAccess_unrestricted.getDatabaseAccess()
    dbargs = dict([ (k,kwargs[k]) for k in ['host','db','user','passwd'] ])
    dbAccess.makeConnection(**dbargs)
    dbAccess.createTables()

    rlapi = rateLimiter_unrestricted.getRateLimiter(**kwargs)

    profilefields = ['uid',
                     'name',
                     'screen_name',
                     'location',
                     'protected',
                     'utc_offset',
                     'time_zone',
                     'statuses_count',
                     'followers_count',
                     'friends_count',
                     'geo_enabled',
                     'lang',
                     'created_at']

    # Initialize the queue
    profile = rlapi.freqLimitGetUser(seeduser)
    startid = profile.id
    samelanguage = profile.lang
    queue = set([])
    queue.add(startid)
    recent = []

    while len(queue) > 0 :
        queue = queue - set(recent)
        if len(recent) > recentusers:
            recent = recent[-recentusers:]
        newuserids = [ queue.pop() for _ in range(min(100,len(queue))) ]
        if verbosity >= DEBUG:
            print '[DEBUG] newuserids:', newuserids

        # Get all new user profiles in a single call
        try:
            profiles = rlapi.freqLimitUsersLookup(newuserids)
            for profile in profiles:
                if verbosity >= VERBOSE:
                    print 'Processing profile ', str(profile.screen_name), ', ', str(profile.id)
                    log.writeMessage('Processing profile ' + str(profile.screen_name) + ', ' + str(profile
                        .id))
                if profile.lang == samelanguage and profile.followers_count < 100000:
                    try:
                    # Try to insert into profiles table
                        dbAccess.profileToSQL(profile, profilefields, **kwargs)
                    except databaseAccess.ConnectionClosed, e:
                        log.writeError()
                        print e.message
                        log.writeMessage(e.message)
                        dbAccess.makeConnection(**dbargs)
                        print 'Connection reestablished.'
                        log.writeMessage('Connection reestablished.')
```

```
78                      except databaseAccess.CursorClosed, e:
79                          log.writeError()
80                          print e.message
81                          log.writeMessage(e.message)
82                          dbAccess.getCursor()
83                          print 'Cursor updated.'
84                          log.writeMessage('Cursor updated.')
85                      except Exception, e:
86                          log.writeError()
87                          print 'Database exception: ', e
88                          log.writeMessage('Database exception: ' + e.message)
89                          # Note that insert failed (probably because entry already exists)
90                          newuserids.remove(profile.id)
91                  else:
92                      # Note that language is not the same as the seed user or user has too many followers
93                      newuserids.remove(profile.id)
94                      print 'User ' + str(profile.id) + ' ignored due to language or large number of
                            followers.'
95                      log.writeMessage('User ' + str(profile.id) + ' ignored due to language or large number
                            of followers.')
96              except Exception, e:
97                  log.writeError()
98                  print 'Twitter API exception: ', e
99                  log.writeMessage('Twitter API exception: ' + e.message)
100
101             # Now get followers, friends and tweets for each user
102             if verbosity >= DEBUG:
103                 print '[DEBUG] newuserids:', newuserids
104             for usrid in newuserids:
105                 log.writeMessage('Fully processing id: ' + str(usrid))
106                 if verbosity >= VERBOSE:
107                     print 'Fully processing id: ', usrid
108
109                 cursor = -1
110                 while cursor != 0:
111                     res = []
112                     try:
113                         res = rlapi.freqLimitGetFollowerIDs(usrid, cursor)
114                         folids = res[u'ids']
115                         cursor = res[u'next_cursor']
116
117                         # Write followers to the database
118                         for folid in folids:
119                             try:
120                                 dbAccess.followToSQL(folid, usrid)
121                                 queue.add(folid)
122                             except databaseAccess.ConnectionClosed, e:
123                                 log.writeError()
124                                 print e.message
125                                 log.writeMessage(e.message)
126                                 dbAccess.makeConnection(**dbargs)
127                                 print 'Connection reestablished.'
128                                 log.writeMessage('Connection reestablished.')
129                             except databaseAccess.CursorClosed, e:
130                                 log.writeError()
131                                 print e.message
132                                 log.writeMessage(e.message)
133                                 dbAccess.getCursor()
134                                 print 'Cursor updated.'
135                                 log.writeMessage('Cursor updated.')
136                             except Exception, e:
137                                 log.writeError()
138                                 print 'Database exception: ', e
139                                 log.writeMessage('Database exception: ' + e.message)
140                     except Exception, e:
141                         log.writeError()
142                         print 'Twitter API exception: ', e
143                         print 'id: ', usrid
144                         log.writeMessage('Twitter API exception: ' + e.message + '\nid: ' + str(usrid))
145                         cursor = 0
146
147                 cursor = -1
148                 while cursor != 0:
149                     res = []
150                     try:
151                         res = rlapi.freqLimitGetFriendIDs(usrid,cursor)
152                         friendids = res['ids']
153                         cursor = res['next_cursor']
154
155                         # Write friends to the database
156                         for friendid in friendids:
157                             try:
158                                 dbAccess.followToSQL(usrid, friendid)
159                                 queue.add(friendid)
160                             except databaseAccess.ConnectionClosed, e:
161                                 log.writeError()
162                                 print e.message
163                                 log.writeMessage(e.message)
164                                 dbAccess.makeConnection(**dbargs)
165                                 print 'Connection reestablished.'
166                                 log.writeMessage('Connection reestablished.')
167                             except databaseAccess.CursorClosed, e:
168                                 log.writeError()
169                                 print e.message
170                                 log.writeMessage(e.message)
171                                 dbAccess.getCursor()
172                                 print 'Cursor updated.'
173                                 log.writeMessage('Cursor updated.')
174                             except Exception, e:
175                                 log.writeError()
176                                 print 'Database exception: ', e
177                                 log.writeMessage('Database exception: ' + e.message)
178                     except Exception, e:
```

```
                    log.writeError()
                    print 'Twitter API exception: ', e
                    print 'id: ', usrid
                    log.writeMessage('Twitter API exception: ' + e.message + '\nid: ' + str(usrid))
                    cursor = 0

            # Get timeline
            log.writeMessage('Getting user timeline')
            if verbosity >= VERBOSE:
                print 'Getting user timeline'
            timeline = []
            try:
                timeline = rlapi.freqLimitGetUserTimeline(user_id=usrid, count=200)

                # Write tweets to the database
                for status in timeline:
                    try:
                        dbAccess.tweetToSQL(status, **kwargs)
                    except databaseAccess.ConnectionClosed, e:
                        log.writeError()
                        print e.message
                        log.writeMessage(e.message)
                        dbAccess.makeConnection(**dbargs)
                        print 'Connection reestablished.'
                        log.writeMessage('Connection reestablished.')
                    except databaseAccess.CursorClosed, e:
                        log.writeError()
                        print e.message
                        log.writeMessage(e.message)
                        dbAccess.getCursor()
                        print 'Cursor updated.'
                        log.writeMessage('Cursor updated.')
                    except Exception, e:
                        log.writeError()
                        print 'Database exception: ', e
                        log.writeMessage('Database exception: ' + e.message)
                        log.writeMessage('Tweet: ' + status.text)
                except Exception, e:
                    log.writeError()
                    print 'Twitter API exception: ', e
                    print 'id: ', usrid
                    log.writeMessage('Twitter API exception: ' + e.message + '\nid: ' + str(usrid))

            # Finally, mark the user as recent
            recent.append(usrid)
            log.writeMessage('Appended user to the recent list.')
            if verbosity >= VERBOSE:
                print 'Appended user to the recent list. Now is: ', recent

    # Close database connection
    dbAccess.closeConnection()

    # End log file
    log.endLog()


if __name__=='__main__':

    from optparse import OptionParser
    parser = OptionParser()
    parser.add_option('-v',"--verbosity",
                      type=int,
                      default=VERBOSE,
                      help="Set the verbosity")
    parser.add_option("--host",
                      type=str,
                      default='localhost',
                      help="Host machine for database access")
    parser.add_option("--db",
                      type=str,
                      default='XXXXX',
                      help="Database name")
    parser.add_option("--user",
                      type=str,
                      default='XXXXX',
                      help="User name for database access")
    parser.add_option("--passwd",
                      type=str,
                      default='XXXXX',
                      help="Password for database access")


    (options, args) = parser.parse_args()
    kwargs = dict([[k,v] for k,v in options.__dict__.iteritems() if not v is None])
    main(*args,**kwargs)
```

crawler_unrestricted.py

```
import time
import twitter

QUIET=0
STDOUTPUT=1
VERBOSE=2
DEBUG=3


class SuppressedCallException(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self ):
```

```
14          return repr(self.value)
15
16
17  class RateLimiter(object):
18
19      readmethods = [ 'FilterPublicTimeline',
20                      'GetUser',
21                      'GetDirectMessages',
22                      'GetFavorites',
23                      'GetFeatured',
24                      'GetFollowerIDs',
25                      'GetFollowers',
26                      'GetFriendIDs',
27                      'GetFriends',
28                      'GetFriendsTimeline',
29                      'GetLists',
30                      'GetMentions',
31                      'GetPublicTimeline',
32                      'GetReplies',
33                      'GetRetweets',
34                      'GetSearch',
35                      'GetStatus',
36                      'GetSubscriptions',
37                      'GetTrendsCurrent',
38                      'GetTrendsDaily',
39                      'GetTrendsWeekly',
40                      'GetUser',
41                      'GetUserByEmail',
42                      'GetUserRetweets',
43                      'GetUserTimeline',
44                      'MaximumHitFrequency',
45                      'UsersLookup'
46                      ]
47
48      writemethods = []
49
50      # Initializing rate limiting variable
51      def __init__(self, api, verbosity):
52          self.max_calls = 0  # number of allowed calls in one hour (the quota)
53          self.calls_left = 0  # what is left in the quota
54          self.t_end = 0 # end time of an one-hour period
55          self.api = api
56          for methodname in self.readmethods+self.writemethods:
57              self.WrapMethod(methodname)
58              self.WaitToMethod(methodname)
59              self.FreqLimitMethod(methodname)
60          self.GetTwitterRateLimit(api=self.api)
61          self.timefrom = None
62          self.verbosity = verbosity
63
64
65      def getTwitterRateLimit(self, api=None):
66
67          # Refresh the number of calls left, max calls and refresh time.
68          if api is None:
69              api = self.api
70          rl = api.GetRateLimitStatus()
71          self.calls_left = rl['remaining_hits']
72          self.max_calls = rl['hourly_limit']
73          self.t_end = rl['reset_time_in_seconds']
74
75
76      def freqLimitMethod(self, methodname):
77
78          # Waits the appropriate length before calling.
79          # Twitter api may throttle us if we try calling repeatedly but within our limit.
80          method = getattr(self.api,methodname)
81          def freqlimitmethod(*args,**kwargs ):
82              if self.verbosity >= DEBUG:
83                  print "[DEBUG] In FreqLimit"+methodname
84              if self.timefrom == None:
85                  self.timefrom = time.time()
86              mhf = self.MaximumHitFrequency()
87              if mhf == None:
88                  mhf = 30
89              waittill = self.timefrom + mhf
90              now = time.time()
91              if now > waittill:
92                  pass
93              else:
94                  if self.verbosity >= STDOUTPUT:
95                      print "Sleeping for "+ str(waittill-now)+ " seconds"
96                  time.sleep(waittill-now)
97                  if self.verbosity >= STDOUTPUT:
98                      print "Waking up"
99              res = method(*args,**kwargs)
100             self.timefrom = time.time()
101             return res
102         setattr(self, 'freqLimit'+methodname, freqlimitmethod)
103
104
105 def getRateLimiter(consumer_key = 'XXXXX',
106                    consumer_secret = 'XXXXX',
107                    access_token_key = 'XXXXX',
108                    access_token_secret = 'XXXXX',
109                    **kwargs):
110
111     api = twitter.Api(consumer_key=consumer_key,
112                       consumer_secret=consumer_secret,
113                       access_token_key=access_token_key,
114                       access_token_secret=access_token_secret,
115                       cache=None)
116
```

```
    rlapi = RateLimiter(api,verbosity=kwargs.get('verbosity'))
    return rlapi
```

rateLimiter_unrestricted.py

```
import psycopg2
from datetime import datetime

DATETIME_STRING_FORMAT = '%a %b %d %H:%M:%S +0000 %Y'
QUIET=0
STDOUTPUT=1
VERBOSE=2
DEBUG=3

profiletable = 'profiles'
socialgraphtable = 'socialgraph'
tweettable = 'tweets'


class ConnectionClosed(Exception):
    def __init__(self):
        self.message = 'ConnectionClosed Exception!'
        return


class CursorClosed(Exception):
    def __init__(self):
        self.message = 'CursorClosed Exception!'
        return


def getDatabaseAccess():
    dbAccess = DatabaseAccess()
    return dbAccess


class DatabaseAccess():

    def __init__(self):
        self.dbconn = None
        self.dbcursor = None


    def makeConnection(self, **kwargs):
        host = kwargs.get('host')
        database = kwargs.get('db')
        user = kwargs.get('user')
        password = kwargs.get('passwd')
        self.dbconn = psycopg2.connect(database=database,user=user,password=password)
        self.getCursor()


    def getCursor(self):
        self.dbcursor = self.dbconn.cursor()


    def closeConnection(self):
        self.dbconn.close()


    def createTables(self):

        # Create table 'profiles' if it doesn't already exist
        tableCheck = self.dbcursor.execute('SELECT count(table_name)::int FROM information_schema.tables
            WHERE table_name = \'' + profiletable + '\'')
        tableCheck = self.dbcursor.fetchone()
        count = tableCheck[0]
        if count == 0:
            try:
                self.dbcursor.execute('CREATE TABLE ' \
                                      + profiletable + '(' \
                                      + 'uid BIGINT PRIMARY KEY,' \
                                      + 'name VARCHAR(40),' \
                                      + 'screen_name VARCHAR(40),' \
                                      + 'location VARCHAR(40),' \
                                      + 'protected BOOL,' \
                                      + 'utc_offset INTEGER,' \
                                      + 'time_zone VARCHAR(40),' \
                                      + 'statuses_count INTEGER,' \
                                      + 'followers_count INTEGER,' \
                                      + 'friends_count INTEGER,' \
                                      + 'geo_enabled BOOL,' \
                                      + 'lang VARCHAR(2),' \
                                      + 'created_at TIMESTAMP)')
            except Exception, e:
                print 'DatabaseAccess exception: ', e
                print 'Connection status: ', self.dbconn.closed
                print 'Cursor status: ', self.dbcursor.closed
                if self.dbconn.closed:
                    raise ConnectionClosed()
                elif self.dbcursor.closed:
                    raise CursorClosed()
                else:
                    raise e

        # Create table 'tweets' if it doesn't already exist
        tableCheck = self.dbcursor.execute('SELECT count(table_name)::int FROM information_schema.tables
            WHERE table_name = \'' + tweettable + '\'')
        tableCheck = self.dbcursor.fetchone()
        count = tableCheck[0]
        if count == 0:
            try:
```

```
96                  self.dbcursor.execute('CREATE TABLE ' + tweettable + '(' \
97                                        + 'tid BIGINT PRIMARY KEY,' \
98                                        + 'uid BIGINT NOT NULL,' \
99                                        + 'text VARCHAR(160),' \
100                                       + 'created_at TIMESTAMP,' \
101                                       + 'truncated BOOL,' \
102                                       + 'retweeted BOOL)')
103             except Exception, e:
104                 print 'DatabaseAccess exception: ', e
105                 print 'Connection status: ', self.dbconn.closed
106                 print 'Cursor status: ', self.dbcursor.closed
107                 if self.dbconn.closed:
108                     raise ConnectionClosed()
109                 elif self.dbcursor.closed:
110                     raise CursorClosed()
111                 else:
112                     raise e
113
114         # Create table 'socialgraph' if it doesn't already exist
115         checkTable = self.dbcursor.execute('SELECT count(table_name)::int FROM information_schema.tables
                WHERE table_name = \'' + socialgraphtable + '\'')
116         checkTable = self.dbcursor.fetchone()
117         count = checkTable[0]
118         if count == 0:
119             try:
120                 self.dbcursor.execute('CREATE TABLE ' + socialgraphtable + '(' \
121                                       + 'parent BIGINT NOT NULL,' \
122                                       + 'child BIGINT NOT NULL,' \
123                                       + 'UNIQUE (parent, child))')
124             except Exception, e:
125                 print 'DatabaseAccess exception: ', e
126                 print 'Connection status: ', self.dbconn.closed
127                 print 'Cursor status: ', self.dbcursor.closed
128                 if self.dbconn.closed:
129                     raise ConnectionClosed()
130                 elif self.dbcursor.closed:
131                     raise CursorClosed()
132                 else:
133                     raise e
134
135         # Commit changes to the database
136         try:
137             self.dbconn.commit()
138         except Exception, e:
139             print 'DatabaseAccess exception: ', e
140             print 'Connection status: ', self.dbconn.closed
141             print 'Cursor status: ', self.dbcursor.closed
142             if self.dbconn.closed:
143                 raise ConnectionClosed()
144             elif self.dbcursor.closed:
145                 self.dbconn.rollback()
146                 raise CursorClosed()
147             else:
148                 self.dbconn.rollback()
149                 raise e
150
151
152     def tableInsert(self, tablename=None, entrydict=None, **kwargs):
153         verbosity = kwargs.get('verbosity')
154         fields = ""
155         valuetemplate = ""
156         values = []
157
158         for k,v in entrydict.iteritems():
159             fields += str(k) + ","
160             valuetemplate += " %s,"
161             values.append(v)
162         fields = fields[:-1]
163         valuetemplate = valuetemplate[:-1]
164         values = tuple(values)
165
166         query = "INSERT INTO "
167         query += tablename + " (" + fields + ") VALUES(" + valuetemplate + ")"
168
169         if verbosity >= DEBUG:
170             print "[debug] query: ", query
171             print "[debug] values: ", values
172         try:
173             self.dbcursor.execute(query,values)
174         except Exception, e:
175             print 'DatabaseAccess exception: failed to insert into table ', tablename, ' with message: ',
                    e
176             print 'Connection status: ', self.dbconn.closed
177             print 'Cursor status: ', self.dbcursor.closed
178             if self.dbconn.closed:
179                 raise ConnectionClosed()
180             elif self.dbcursor.closed:
181                 raise CursorClosed()
182             else:
183                 raise e
184         finally:
185             try:
186                 self.dbconn.commit()
187             except Exception, e:
188                 print 'DatabaseAccess exception: failed to commit changes to table ', tablename, ' with
                        message:', e
189                 print 'Connection status: ', self.dbconn.closed
190                 print 'Cursor status: ', self.dbcursor.closed
191                 if self.dbconn.closed:
192                     raise ConnectionClosed()
193                 elif self.dbcursor.closed:
194                     self.dbconn.rollback()
195                     raise CursorClosed()
```

```
                else:
                    self.dbconn.rollback()
                    raise e


    def profileToSQL(self, profile, fields, **kwargs):
        dprofile = profile.AsDict()
        dprofile['uid'] = dprofile.pop('id')
        entrydict = dict([ (key,dprofile[key]) for key in fields if key in dprofile ])
        try:
            self.tableInsert(tablename=profiletable, entrydict=entrydict, **kwargs)
        except Exception, e:
            raise


    def tweetToSQL(self, status, **kwargs):
        created_at = datetime.strptime(status.created_at, DATETIME_STRING_FORMAT)
        entrydict = dict(uid=status.user.id,
                         tid=status.id,
                         text=status.text,
                         created_at=created_at,
                         truncated=status.truncated,
                         retweeted=status.retweeted)
        try:
            self.tableInsert(tablename=tweettable, entrydict=entrydict, **kwargs)
        except Exception, e:
            raise


    def followToSQL(self, follower, followed, followerfield='parent', followedfield='child', **kwargs):
        entrydict = dict( [(followerfield,follower), (followedfield,followed)])
        try:
            self.tableInsert(tablename=socialgraphtable, entrydict=entrydict)
        except Exception, e:
            raise
```

databaseAccess_unrestricted.py

```
import datetime
import traceback
import sys
import os


def getErrorReport():
    errorReport = ErrorReport()
    return errorReport


class ErrorReport():

    def __init__(self):
        return

    def startLog(self):
        timestamp = str(datetime.datetime.now())
        fileName = 'Log_'+timestamp+'.txt'
        self.logFile = open(fileName,'w')

    def endLog(self):
        self.logFile.close()

    def writeError(self):
        traceback.print_exc(file=self.logFile)
        self.logFile.write('\n')
        self.logFile.flush()
        os.fsync(self.logFile)

    def writeMessage(self, message=''):
        self.logFile.write(message)
        self.logFile.write('\n\n')
        self.logFile.flush()
        os.fsync(self.logFile)
```

errorReport_unrestricted.py

Appendix B

Creepy Crawly (Restricted)

```
import twitter
import rateLimiter_restricted
import errorReport_restricted
import databaseAccess_restricted
from pyparsing import Word, alphas, alphanums, CaselessLiteral, empty, printables, Keyword,
     CaselessKeyword

QUIET=0
STDOUTPUT=1
VERBOSE=2
DEBUG=3


def main(**kwargs):

    print 'Main arguments: ', kwargs

    verbosity = kwargs.get('verbosity')

    # Create log file
    log = errorReport_restricted.getErrorReport()
    log.startLog()

    # Grammar for parsing retweet
    retweeted = Word( alphanums + "_" + "-" )
    grammar = Keyword("RT") + "@" + retweeted.setResultsName("name") + ":"

    # Connect to database and create tables
    dbAccess = databaseAccess_restricted.getDatabaseAccess()
    dbargs = dict([ (k,kwargs[k]) for k in ['host','db','user','passwd'] ])
    dbAccess.makeConnection(**dbargs)
    dbAccess.createTables()

    rlapi = rateLimiter_restricted.getRateLimiter(**kwargs)

    # List of fields in the profile table
    profilefields = ['uid',
                     'name',
                     'screen_name',
                     'location',
                     'protected',
                     'utc_offset',
                     'time_zone',
                     'statuses_count',
                     'followers_count',
                     'friends_count',
                     'favourites_count',
                     'geo_enabled',
                     'lang',
                     'created_at']

    # List of users we want to track
    users = ['gabioptavares']

    for user in users:

        log.writeMessage('Started user: ' + user)

        # Get user profile
        try:
            profile = rlapi.freqLimitGetUser(user)
            usrid = profile.id
        except Exception, e:
             log.writeError()
             print 'Twitter API exception: ', e
             log.writeMessage('Twitter API exception: ' + e.message)
             continue

        # Insert profile into profile table
        try:
            dbAccess.profileToSQL(profile, profilefields, **kwargs)
        except databaseAccess_restricted.ConnectionClosed, e:
            log.writeError()
            print e.message
            log.writeMessage(e.message)
            dbAccess.makeConnection(**dbargs)
            print 'Connection reestablished.'
            log.writeMessage('Connection reestablished.')
```

```
78          except databaseAccess_restricted.CursorClosed, e:
79              log.writeError()
80              print e.message
81              log.writeMessage(e.message)
82              dbAccess.getCursor()
83              print 'Cursor updated.'
84              log.writeMessage('Cursor updated.')
85          except Exception, e:
86              log.writeError()
87              print 'Database exception: ', e
88              log.writeMessage('Database exception: ' + e.message)
89
90          # Get user's followers
91          cursor = -1
92          while cursor != 0:
93              res = []
94              try:
95                  res = rlapi.freqLimitGetFollowerIDs(usrid, cursor)
96                  folids = res[u'ids']
97                  cursor = res[u'next_cursor']
98
99                  # Insert followers into follower table
100                 for folid in folids:
101                     try:
102                         dbAccess.followerToSQL(usrid, folid)
103                     except databaseAccess_restricted.ConnectionClosed, e:
104                         log.writeError()
105                         print e.message
106                         log.writeMessage(e.message)
107                         dbAccess.makeConnection(**dbargs)
108                         print 'Connection reestablished.'
109                         log.writeMessage('Connection reestablished.')
110                     except databaseAccess_restricted.CursorClosed, e:
111                         log.writeError()
112                         print e.message
113                         log.writeMessage(e.message)
114                         dbAccess.getCursor()
115                         print 'Cursor updated.'
116                         log.writeMessage('Cursor updated.')
117                     except Exception, e:
118                         log.writeError()
119                         print 'Database exception: ', e
120                         log.writeMessage('Database exception: ' + e.message)
121             except Exception, e:
122                 log.writeError()
123                 print 'Twitter API exception: ', e
124                 print 'id: ', usrid
125                 log.writeMessage('Twitter API exception: ' + e.message + '\nid: ' + str(usrid))
126                 cursor = 0
127
128         # Get user's friends
129         cursor = -1
130         while cursor != 0:
131             res = []
132             try:
133                 res = rlapi.freqLimitGetFriendIDs(usrid, cursor)
134                 friendids = res[u'ids']
135                 cursor = res[u'next_cursor']
136
137                 # Insert friends into friend table
138                 for friendid in friendids:
139                     try:
140                         dbAccess.friendToSQL(usrid, friendid)
141                     except databaseAccess_restricted.ConnectionClosed, e:
142                         log.writeError()
143                         print e.message
144                         log.writeMessage(e.message)
145                         dbAccess.makeConnection(**dbargs)
146                         print 'Connection reestablished.'
147                         log.writeMessage('Connection reestablished.')
148                     except databaseAccess_restricted.CursorClosed, e:
149                         log.writeError()
150                         print e.message
151                         log.writeMessage(e.message)
152                         dbAccess.getCursor()
153                         print 'Cursor updated.'
154                         log.writeMessage('Cursor updated.')
155                     except Exception, e:
156                         log.writeError()
157                         print 'Database exception: ', e
158                         log.writeMessage('Database exception: ' + e.message)
159             except Exception, e:
160                 log.writeError()
161                 print 'Twitter API exception: ', e
162                 print 'id: ', usrid
163                 log.writeMessage('Twitter API exception: ' + e.message + '\nid: ' + str(usrid))
164                 cursor = 0
165
166         # Get user's timeline
167         timeline = []
168
169         tl = rlapi.freqLimitGetUserTimeline(user_id=usrid, count=200, include_rts=True, include_entities=
                True)
170         while (tl != None and tl != [] and len(timeline) <= 600):
171             log.writeMessage('TIMELINE LENGTH: '+str(len(timeline)))
172             timeline = timeline + tl
173             maxId = (tl[-1]).id -1
174             try:
175                 tl = rlapi.freqLimitGetUserTimeline(user_id=usrid, count=200, max_id=maxId, include_rts=
                        True, include_entities=True)
176             except Exception, e:
177                 log.writeError()
178                 print 'Twitter API exception: ', e
```

```
                print 'id: ', usrid
                log.writeMessage('Twitter API exception: ' + e.message + '\nid: ' + str(usrid))

        for status in timeline:

            # Check if it is a retweet
            isRT = False
            res = []
            try:
                res = grammar.parseString(status.text)
            except Exception, e:
                pass
            finally:
                if len(res) > 0:
                    isRT = True
                    retweeted_user = res.name

            # If it is a retweet, insert into retweet table
            if isRT:
                try:
                    dbAccess.retweetToSQL(status, retweeted_user, **kwargs)
                except databaseAccess_restricted.ConnectionClosed, e:
                    log.writeError()
                    print e.message
                    log.writeMessage(e.message)
                    dbAccess.makeConnection(**dbargs)
                    print 'Connection reestablished.'
                    log.writeMessage('Connection reestablished.')
                except databaseAccess_restricted.CursorClosed, e:
                    log.writeError()
                    print e.message
                    log.writeMessage(e.message)
                    dbAccess.getCursor()
                    print 'Cursor updated.'
                    log.writeMessage('Cursor updated.')
                except Exception, e:
                    log.writeError()
                    print 'Database exception: ', e
                    log.writeMessage('Database exception: ' + e.message)

            # Else, get retweets, check for entities and insert into tweet table
            else:
                # Get retweets
                try:
                    retweets = rlapi.freqLimitGetRetweets(status.id)
                except Exception, e:
                    log.writeError()
                    print 'Twitter API exception: ', e
                    print 'id: ', usrid
                    log.writeMessage('Twitter API exception: ' + e.message + '\nid: ' + str(usrid))
                retweeter_ids = []
                for retweet in retweets:
                    retweeter_ids.append(retweet.user.id)
                retweet_count = len(retweeter_ids)

                # Check for reply and entities (mentions, hashtags, media and urls)
                is_reply = False
                is_mention = False
                mentions = []
                mention_ids = []
                is_hashtag = False
                hashtags = []
                is_media = False
                is_url = False
                if status.in_reply_to_status_id != None:
                    is_reply = True
                if len(status.user_mentions) > 0:
                    is_mention = True
                    for mention in status.user_mentions:
                        mentions.append(mention.screen_name)
                        mention_ids.append(mention.id)
                if len(status.hashtags) > 0:
                    is_hastag = True
                    for hashtag in status.hashtags:
                        hashtags.append(hashtag.text)
                if status.media != None:
                    is_media = True
                if len(status.urls) > 0:
                    is_url = True

                # Insert into tweet table
                try:
                    dbAccess.tweetToSQL(status, retweet_count, retweeter_ids, is_reply, is_mention,
                        mentions, mention_ids, is_hashtag, hashtags, is_media, is_url, **kwargs)
                except databaseAccess_restricted.ConnectionClosed, e:
                    log.writeError()
                    print e.message
                    log.writeMessage(e.message)
                    dbAccess.makeConnection(**dbargs)
                    print 'Connection reestablished.'
                    log.writeMessage('Connection reestablished.')
                except databaseAccess_restricted.CursorClosed, e:
                    log.writeError()
                    print e.message
                    log.writeMessage(e.message)
                    dbAccess.getCursor()
                    print 'Cursor updated.'
                    log.writeMessage('Cursor updated.')
                except Exception, e:
                    log.writeError()
                    print 'Database exception: ', e
                    log.writeMessage('Database exception: ' + e.message)
```

```
        # Get user's favorites

        try:
            favorites = rlapi.freqLimitGetFavorites(usrid)
        except Exception, e:
            log.writeError()
            print 'Twitter API exception: ', e
            print 'id: ', usrid
            log.writeMessage('Twitter API exception: ' + e.message + '\nid: ' + str(usrid))

        # Insert favorites into favorite table
        for favorite in favorites:
            try:
                dbAccess.favoriteToSQL(usrid, favorite, **kwargs)
            except databaseAccess_restricted.ConnectionClosed, e:
                log.writeError()
                print e.message
                log.writeMessage(e.message)
                dbAccess.makeConnection(**dbargs)
                print 'Connection reestablished.'
                log.writeMessage('Connection reestablished.')
            except databaseAccess_restricted.CursorClosed, e:
                log.writeError()
                print e.message
                log.writeMessage(e.message)
                dbAccess.getCursor()
                print 'Cursor updated.'
                log.writeMessage('Cursor updated.')
            except Exception, e:
                log.writeError()
                print 'Database exception: ', e
                log.writeMessage('Database exception: ' + e.message)

        log.writeMessage('Finished user: ' + user)


if __name__=='__main__':

    from optparse import OptionParser
    parser = OptionParser()
    parser.add_option('-v',"--verbosity",
                      type=int,
                      default=VERBOSE,

                      help="Set the verbosity")
    parser.add_option("--host",
                      type=str,
                      default='localhost',
                      help="Host machine for database access")
    parser.add_option("--db",
                      type=str,
                      default='XXXXX',
                      help="Database name")
    parser.add_option("--user",
                      type=str,
                      default='XXXXX',
                      help="User name for database access")
    parser.add_option("--passwd",
                      type=str,
                      default='XXXXX',
                      help="Password for database access")


    (options, args) = parser.parse_args()
    kwargs = dict([[k,v] for k,v in options.__dict__.iteritems() if not v is None])
    main(*args,**kwargs)
```

crawler_restricted.py

```
import time
import twitter

QUIET=0
STDOUTPUT=1
VERBOSE=2
DEBUG=3


class SuppressedCallException(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self ):
        return repr(self.value)

class RateLimiter(object):

    readmethods = [ 'FilterPublicTimeline',
                    'GetUser',
                    'GetDirectMessages',
                    'GetFavorites',
                    'GetFeatured',
                    'GetFollowerIDs',
                    'GetFollowers',
                    'GetFriendIDs',
                    'GetFriends',
                    'GetFriendsTimeline',
                    'GetLists',
                    'GetMentions',
                    'GetPublicTimeline',
                    'GetReplies',
                    'GetRetweets',
                    'GetSearch',
```

```
                    'GetStatus',
                    'GetSubscriptions',
                    'GetTrendsCurrent',
                    'GetTrendsDaily',
                    'GetTrendsWeekly',
                    'GetUser',
                    'GetUserByEmail',
                    'GetUserRetweets',
                    'GetUserTimeline',
                    'MaximumHitFrequency',
                    'UsersLookup'
                    ]

    writemethods = []

    # Initializing rate limiting variable
    def __init__(self, api, verbosity):
        self.max_calls = 0  # number of allowed calls in one hour : the quota
        self.calls_left = 0  # what is left in the quota
        self.t_end = 0 # end time of an one-hour period
        self.api = api
        for methodname in self.readmethods+self.writemethods:
            self.WrapMethod(methodname)
            self.WaitToMethod(methodname)
            self.FreqLimitMethod(methodname)
        self.GetTwitterRateLimit(api=self.api)
        self.timefrom = None
        self.verbosity = verbosity


    def GetTwitterRateLimit(self, api=None):
        # Refresh the number of calls left, max calls and refresh time.
        if api is None:
            api = self.api
        rl = api.GetRateLimitStatus()
        self.calls_left = rl['remaining_hits']
        self.max_calls = rl['hourly_limit']
        self.t_end = rl['reset_time_in_seconds']


    def freqLimitMethod(self, methodname):
        # Waits the appropriate length before calling.
        # Twitter api may throttle us if we try calling repeatedly but within our limit.
        method = getattr(self.api,methodname)
        def freqlimitmethod(*args,**kwargs ):
            if self.verbosity >= DEBUG:
                print "[DEBUG] In FreqLimit"+methodname
            if self.timefrom == None:
                self.timefrom = time.time()
            mhf = self.MaximumHitFrequency()
            if mhf == None:
                mhf = 30
            waittill = self.timefrom + mhf
            now = time.time()
            if now > waittill:
                pass
            else:
                if self.verbosity >= STDOUTPUT:
                    print "Sleeping for "+ str(waittill-now)+ " seconds"
                time.sleep(waittill-now)
                if self.verbosity >= STDOUTPUT:
                    print "Waking up"
            res = method(*args,**kwargs)
            self.timefrom = time.time()
            return res
        setattr(self, 'freqLimit'+methodname, freqlimitmethod)


def getRateLimiter(consumer_key = 'XXXXX',
                   consumer_secret = 'XXXXX',
                   access_token_key = 'XXXXX',
                   access_token_secret = 'XXXXX',
                   **kwargs):

    api = twitter.Api(consumer_key=consumer_key,
                      consumer_secret=consumer_secret,
                      access_token_key=access_token_key,
                      access_token_secret=access_token_secret,
                      cache=None)

    rlapi = RateLimiter(api,verbosity=kwargs.get('verbosity'))
    return rlapi
```

rateLimiter_restricted.py

```
import psycopg2
from datetime import datetime, date

DATETIME_STRING_FORMAT = '%a %b %d %H:%M:%S +0000 %Y'
QUIET=0
STDOUTPUT=1
VERBOSE=2
DEBUG=3

profiletable = 'profiles'
followertable = 'followers'
friendtable = 'friends'
tweettable = 'tweets'
retweettable = 'retweets'
favoritetable = 'favorites'


```

```
18 class ConnectionClosed(Exception):
19     def __init__(self):
20         self.message = 'ConnectionClosed Exception!'
21         return
22 
23 
24 class CursorClosed(Exception):
25     def __init__(self):
26         self.message = 'CursorClosed Exception!'
27         return
28 
29 
30 def getDatabaseAccess():
31     dbAccess = DatabaseAccess()
32     return dbAccess
33 
34 
35 class DatabaseAccess():
36 
37     def __init__(self):
38         self.dbconn = None
39         self.dbcursor = None
40 
41 
42     def makeConnection(self, **kwargs):
43         host = kwargs.get('host')
44         database = kwargs.get('db')
45         user = kwargs.get('user')
46         password = kwargs.get('passwd')
47         self.dbconn = psycopg2.connect(database=database,user=user,password=password)
48         self.getCursor()
49 
50 
51     def getCursor(self):
52         self.dbcursor = self.dbconn.cursor()
53 
54 
55     def closeConnection(self):
56         self.dbconn.close()
57 
58 
59     def createTables(self):
60 
61         # Create table 'profiles' if it doesn't already exist
62         tableCheck = self.dbcursor.execute('SELECT count(table_name)::int FROM information_schema.tables
                WHERE table_name = \'' + profiletable + '\'')
63         tableCheck = self.dbcursor.fetchone()
64         count = tableCheck[0]
65         if count == 0:
66             try:
67                 self.dbcursor.execute('CREATE TABLE ' \
68                                       + profiletable + '(' \
69                                       + 'uid BIGINT PRIMARY KEY,' \
70                                       + 'name VARCHAR(40),' \
71                                       + 'screen_name VARCHAR(40),' \
72                                       + 'location VARCHAR(40),' \
73                                       + 'protected BOOL,' \
74                                       + 'utc_offset INTEGER,' \
75                                       + 'time_zone VARCHAR(40),' \
76                                       + 'statuses_count INTEGER,' \
77                                       + 'followers_count INTEGER,' \
78                                       + 'friends_count INTEGER,' \
79                                       + 'favourites_count INTEGER,' \
80                                       + 'geo_enabled BOOL,' \
81                                       + 'lang VARCHAR(2),' \
82                                       + 'created_at TIMESTAMP)')
83             except Exception, e:
84                 print 'DatabaseAccess exception: ', e
85                 print 'Connection status: ', self.dbconn.closed
86                 print 'Cursor status: ', self.dbcursor.closed
87                 if self.dbconn.closed:
88                     raise ConnectionClosed()
89                 elif self.dbcursor.closed:
90                     raise CursorClosed()
91                 else:
92                     raise e
93 
94         # Create table 'followers' if it doesn't already exist
95         checkTable = self.dbcursor.execute('SELECT count(table_name)::int FROM information_schema.tables
                WHERE table_name = \'' + followertable + '\'')
96         checkTable = self.dbcursor.fetchone()
97         count = checkTable[0]
98         if count == 0:
99             try:
100                self.dbcursor.execute('CREATE TABLE ' + followertable + '(' \
101                                      + 'uid BIGINT NOT NULL,' \
102                                      + 'follower_id BIGINT NOT NULL,' \
103                                      + 'UNIQUE (uid, follower_id))')
104            except Exception, e:
105                print 'DatabaseAccess exception: ', e
106                print 'Connection status: ', self.dbconn.closed
107                print 'Cursor status: ', self.dbcursor.closed
108                if self.dbconn.closed:
109                    raise ConnectionClosed()
110                elif self.dbcursor.closed:
111                    raise CursorClosed()
112                else:
113                    raise e
114 
115        # Create table 'friends' if it doesn't already exist
116        checkTable = self.dbcursor.execute('SELECT count(table_name)::int FROM information_schema.tables
                WHERE table_name = \'' + friendtable + '\'')
117        checkTable = self.dbcursor.fetchone()
```

```
118         count = checkTable[0]
119         if count == 0:
120             try:
121                 self.dbcursor.execute('CREATE TABLE ' + friendtable + '(' \
122                                       + 'uid BIGINT NOT NULL,' \
123                                       + 'friend_id BIGINT NOT NULL,' \
124                                       + 'UNIQUE (uid, friend_id))')
125             except Exception, e:
126                 print 'DatabaseAccess exception: ', e
127                 print 'Connection status: ', self.dbconn.closed
128                 print 'Cursor status: ', self.dbcursor.closed
129                 if self.dbconn.closed:
130                     raise ConnectionClosed()
131                 elif self.dbcursor.closed:
132                     raise CursorClosed()
133                 else:
134                     raise e
135 
136         # Create table 'tweets' if it doesn't already exist
137         tableCheck = self.dbcursor.execute('SELECT count(table_name)::int FROM information_schema.tables
                WHERE table_name = \'' + tweettable + '\'')
138         tableCheck = self.dbcursor.fetchone()
139         count = tableCheck[0]
140         if count == 0:
141             try:
142                 self.dbcursor.execute('CREATE TABLE ' + tweettable + '(' \
143                                       + 'uid BIGINT NOT NULL,' \
144                                       + 'tid BIGINT PRIMARY KEY,' \
145                                       + 'text VARCHAR(160),' \
146                                       + 'created_at TIMESTAMP,' \
147                                       + 'is_reply BOOL,' \
148                                       + 'in_reply_to_user_id BIGINT,' \
149                                       + 'in_reply_to_status_id BIGINT,' \
150                                       + 'is_mention BOOL,' \
151                                       + 'mentions VARCHAR(40)[],' \
152                                       + 'mention_ids BIGINT[],' \
153                                       + 'is_hashtag BOOL,' \
154                                       + 'hashtags VARCHAR(100)[],' \
155                                       + 'is_media BOOL,' \
156                                       + 'is_url BOOL,' \
157                                       + 'retweet_count INTEGER,' \
158                                       + 'retweeter_ids BIGINT[])')
159 
160             except Exception, e:
161                 print 'DatabaseAccess exception: ', e
162                 print 'Connection status: ', self.dbconn.closed
163                 print 'Cursor status: ', self.dbcursor.closed
164                 if self.dbconn.closed:
165                     raise ConnectionClosed()
166                 elif self.dbcursor.closed:
167                     raise CursorClosed()
168                 else:
169                     raise e
170 
171         # Create table 'retweets' if it doesn't already exist
172         tableCheck = self.dbcursor.execute('SELECT count(table_name)::int FROM information_schema.tables
                WHERE table_name = \'' + retweettable + '\'')
173         tableCheck = self.dbcursor.fetchone()
174         count = tableCheck[0]
175         if count == 0:
176             try:
177                 self.dbcursor.execute('CREATE TABLE ' + retweettable + '(' \
178                                       + 'uid BIGINT NOT NULL,' \
179                                       + 'retweeted_user VARCHAR(30),' \
180                                       + 'tid BIGINT NOT NULL,' \
181                                       + 'text VARCHAR(160),' \
182                                       + 'created_at TIMESTAMP,' \
183                                       + 'UNIQUE (uid, tid))')
184             except Exception, e:
185                 print 'DatabaseAccess exception: ', e
186                 print 'Connection status: ', self.dbconn.closed
187                 print 'Cursor status: ', self.dbcursor.closed
188                 if self.dbconn.closed:
189                     raise ConnectionClosed()
190                 elif self.dbcursor.closed:
191                     raise CursorClosed()
192                 else:
193                     raise e
194 
195         # Create table 'favorites' if it doesn't already exist
196         tableCheck = self.dbcursor.execute('SELECT count(table_name)::int FROM information_schema.tables
                WHERE table_name = \'' + favoritetable + '\'')
197         tableCheck = self.dbcursor.fetchone()
198         count = tableCheck[0]
199         if count == 0:
200             try:
201                 self.dbcursor.execute('CREATE TABLE ' + favoritetable + '(' \
202                                       + 'uid BIGINT NOT NULL,' \
203                                       + 'tid BIGINT NOT NULL,' \
204                                       + 'favorited_user_id BIGINT NOT NULL,' \
205                                       + 'text VARCHAR(160),' \
206                                       + 'created_at TIMESTAMP,' \
207                                       + 'UNIQUE (uid, tid))')
208             except Exception, e:
209                 print 'DatabaseAccess exception: ', e
210                 print 'Connection status: ', self.dbconn.closed
211                 print 'Cursor status: ', self.dbcursor.closed
212                 if self.dbconn.closed:
213                     raise ConnectionClosed()
214                 elif self.dbcursor.closed:
215                     raise CursorClosed()
216                 else:
217                     raise e
```

```
218
219
220         # Commit changes to the database
221         try:
222             self.dbconn.commit()
223         except Exception, e:
224             print 'DatabaseAccess exception: ', e
225             print 'Connection status: ', self.dbconn.closed
226             print 'Cursor status: ', self.dbcursor.closed
227             if self.dbconn.closed:
228                 raise ConnectionClosed()
229             elif self.dbcursor.closed:
230                 self.dbconn.rollback()
231                 raise CursorClosed()
232             else:
233                 self.dbconn.rollback()
234                 raise e
235
236
237     def tableInsert(self, tablename=None, entrydict=None, **kwargs):
238         verbosity = kwargs.get('verbosity')
239         fields = ""
240         valuetemplate = ""
241         values = []
242
243         for k,v in entrydict.iteritems():
244             fields += str(k) + ","
245             valuetemplate += " %s,"
246             values.append(v)
247         fields = fields[:-1]
248         valuetemplate = valuetemplate[:-1]
249         values = tuple(values)
250
251         query = "INSERT INTO "
252         query += tablename + " (" + fields + ") VALUES(" + valuetemplate + ")"
253
254         if verbosity >= DEBUG:
255             print "[debug] query: ", query
256             print "[debug] values: ", values
257         try:
258             self.dbcursor.execute(query,values)
259         except Exception, e:
260             print 'DatabaseAccess exception: failed to insert into table ', tablename, ' with message: ',
                  e
261             print 'Connection status: ', self.dbconn.closed
262             print 'Cursor status: ', self.dbcursor.closed
263             if self.dbconn.closed:
264                 raise ConnectionClosed()
265             elif self.dbcursor.closed:
266                 raise CursorClosed()
267             else:
268                 raise e
269         finally:
270             try:
271                 self.dbconn.commit()
272             except Exception, e:
273                 print 'DatabaseAccess exception: failed to commit changes to table ', tablename, ' with
                      message:', e
274                 print 'Connection status: ', self.dbconn.closed
275                 print 'Cursor status: ', self.dbcursor.closed
276                 if self.dbconn.closed:
277                     raise ConnectionClosed()
278                 elif self.dbcursor.closed:
279                     self.dbconn.rollback()
280                     raise CursorClosed()
281                 else:
282                     self.dbconn.rollback()
283                     raise e
284
285
286     def profileToSQL(self, profile, fields, **kwargs):
287         dprofile = profile.AsDict()
288         dprofile['uid'] = dprofile.pop('id')
289         entrydict = dict([ (key,dprofile[key]) for key in fields if key in dprofile ])
290         try:
291             self.tableInsert(tablename=profiletable, entrydict=entrydict, **kwargs)
292         except Exception, e:
293             raise
294
295
296     def followerToSQL(self, usrid, follower_id, **kwargs):
297         entrydict = dict(uid=usrid,
298                          follower_id=follower_id)
299         try:
300             self.tableInsert(tablename=followertable, entrydict=entrydict, **kwargs)
301         except Exception, e:
302             raise
303
304
305     def friendToSQL(self, usrid, friend_id, **kwargs):
306         entrydict = dict(uid=usrid,
307                          friend_id=friend_id)
308         try:
309             self.tableInsert(tablename=friendtable, entrydict=entrydict, **kwargs)
310         except Exception, e:
311             raise
312
313
314     def tweetToSQL(self, status, retweet_count, retweeter_ids, is_reply, is_mention, mentions, mention_ids
          , is_hashtag, hashtags, is_media, is_url, **kwargs):
315         entrydict = dict(uid=status.user.id,
316                          tid=status.id,
317                          text=status.text,
```

```
                            created_at=status.created_at,
                            is_reply=is_reply,
                            in_reply_to_user_id=status.in_reply_to_user_id,
                            in_reply_to_status_id=status.in_reply_to_status_id,
                            is_mention=is_mention,
                            mentions=mentions,
                            mention_ids=mention_ids,
                            is_hashtag=is_hashtag,
                            hashtags=hashtags,
                            is_media=is_media,
                            is_url=is_url,
                            retweet_count=retweet_count,
                            retweeter_ids=retweeter_ids)
        try:
            self.tableInsert(tablename=tweettable, entrydict=entrydict, **kwargs)
        except Exception, e:
            raise


    def retweetToSQL(self, status, retweeted_user, **kwargs):
        created_at = datetime.strptime(status.created_at, DATETIME_STRING_FORMAT)
        entrydict = dict(uid=status.user.id,
                            retweeted_user=retweeted_user,
                            tid=status.id,
                            text=status.text,
                            created_at=created_at)
        try:
            self.tableInsert(tablename=retweettable, entrydict=entrydict, **kwargs)
        except Exception, e:
            raise


    def favoriteToSQL(self, usrid, status, **kwargs):
        created_at = datetime.strptime(status.created_at, DATETIME_STRING_FORMAT)
        entrydict = dict(uid=usrid,
                            tid=status.id,
                            favorited_user_id=status.user.id,
                            text=status.text,
                            created_at=created_at)
        try:
            self.tableInsert(tablename=favoritetable, entrydict=entrydict, **kwargs)
        except Exception, e:
            raise
```

databaseAccess_restricted.py

```
import datetime
import traceback
import sys
import os


def getErrorReport():
    errorReport = ErrorReport()
    return errorReport


class ErrorReport():

    def __init__(self):
        return

    def startLog(self):
        timestamp = str(datetime.datetime.now())
        fileName = 'Log_'+timestamp+'.txt.'
        self.logFile = open(fileName,'w')

    def endLog(self):
        self.logFile.close()

    def writeError(self):
        traceback.print_exc(file=self.logFile)
        self.logFile.write('\n')
        self.logFile.flush()
        os.fsync(self.logFile)

    def writeMessage(self, message=''):
        self.logFile.write(message)
        self.logFile.write('\n\n')
        self.logFile.flush()
        os.fsync(self.logFile)
```

errorReport_restricted.py

Appendix C

2-Classifier

```
% This function classifies users in one of two classes: personal accounts
% (P) and managed accounts (M). It takes as input the size of the dataset
% used in the classification and performs leave-one-out cross validation.
% Four different classifications are used: using only inter-tweet delay
% intervals (ITD), using only tweeting times (TT), using both as
% independent variables (JI) and using both as non-independent variables
% (JNI).

function [numSamples, percCorrect_JNI, percCorrect_JI, percCorrect_ITD, percCorrect_TT] = Classifier2(
    varargin)

load companies.mat
J{1} = T;

load people.mat
J{2} = T;

% Class M - ITD
T_M_ITD = J{1};
g_M_ITD = grp2idx(T_M_ITD(:,1));

% Class P - ITD
T_P_ITD = J{2};
g_P_ITD = grp2idx(T_P_ITD(:,1));

% Class M - TT
T_M_TT = J{1};
g_M_TT = grp2idx(T_M_TT(:,1));

% Class P - TT
T_P_TT = J{2};
g_P_TT = grp2idx(T_P_TT(:,1));

% Class M - J
T_M_J = J{1};
g_M_J = grp2idx(T_M_J(:,1));

% Class P - J
T_P_J = J{2};
g_P_J = grp2idx(T_P_J(:,1));

% Confusion matrix counts
countPP_ITD = 0;
countPM_ITD = 0;
countMP_ITD = 0;
countMM_ITD = 0;

countPP_TT = 0;
countPM_TT = 0;
countMP_TT = 0;
countMM_TT = 0;

countPP_JI = 0;
countPM_JI = 0;
countMP_JI = 0;
countMM_JI = 0;

countPP_JNI = 0;
countPM_JNI = 0;
countMP_JNI = 0;
countMM_JNI = 0;

if nargin == 0
    for i=1:2
        T = J{i};
        g = grp2idx(T(:,1));
        n(i) = length(unique(g));
    end
    for i=1:2
        T = J{i};
        g = grp2idx(T(:,1));
        n(i) = length(unique(g));
    end
    numSamples = min(n);
else
    numSamples = varargin{1};
end

```

```
%% ITD

% Leave-one-out cross validation
for i=1:numSamples

    % Probability distribution for class M
    [n_M,xout_M] = hist(T_M_ITD(g_M_ITD~=i & g_M_ITD<=numSamples,2),logspace(0,8,30));
    n_M = n_M/sum(n_M);
    xout_M_lin = log(xout_M);

    % Probability distribution for class P
    [n_P,xout_P] = hist(T_P_ITD(g_P_ITD~=i & g_P_ITD<=numSamples,2),logspace(0,8,30));
    n_P = n_P/sum(n_P);
    xout_P_lin = log(xout_P);

    % Get subject SubjM from class M, get probability for classes M and P
    SubjM = [];
    SubjM = T_M_ITD(g_M_ITD==i,2);
    Sum_M_M = 0;
    Sum_M_P = 0;
    for j=1:size(SubjM,1)
        Sum_M_M = Sum_M_M + log(interp1(xout_M_lin,n_M,log(SubjM(j,1)),'spline'));
    end
    for j=1:size(SubjM,1)
        Sum_M_P = Sum_M_P + log(interp1(xout_P_lin,n_P,log(SubjM(j,1)),'spline'));
    end

    % Get subject SubjP from class P, get probability for classes M and P
    SubjP = [];
    SubjP = T_P_ITD(g_P_ITD==i,2);
    Sum_P_M = 0;
    Sum_P_P = 0;
    for j=1:size(SubjP,1)
        Sum_P_M = Sum_P_M + log(interp1(xout_M_lin,n_M,log(SubjP(j,1)),'spline'));
    end
    for j=1:size(SubjP,1)
        Sum_P_P = Sum_P_P + log(interp1(xout_P_lin,n_P,log(SubjP(j,1)),'spline'));
    end

    resM = Sum_M_P - Sum_M_M;
    if (resM > 0)
        countMP_ITD = countMP_ITD + 1;
    elseif (resM < 0)
        countMM_ITD = countMM_ITD + 1;
    end

    resP = Sum_P_P - Sum_P_M;
    if (resP > 0)
        countPP_ITD = countPP_ITD + 1;
    elseif (resP < 0)
        countPM_ITD = countPM_ITD + 1;
    end
end


%% TT

% Leave-one-out cross validation
for i=1:numSamples

    % Probability distribution for class M
    [n_M,xout_M] = hist(T_M_TT(g_M_TT~=i & g_M_TT<=numSamples,3),0:1:23);
    n_M = n_M/sum(n_M);

    % Probability distribution for class I
    [n_P,xout_P] = hist(T_P_TT(g_P_TT~=i & g_P_TT<=numSamples,3),0:1:23);
    n_P = n_P/sum(n_P);

    % Get subject SubjM from class M, get probability for classes M and P
    SubjM = [];
    SubjM = T_M_TT(g_M_TT==i,3);
    Sum_M_M = 0;
    Sum_M_P = 0;
    for j=1:size(SubjM,1)
        Sum_M_M = Sum_M_M + log(interp1(xout_M,n_M,SubjM(j,1),'spline'));
    end
    for j=1:size(SubjM,1)
        Sum_M_P = Sum_M_P + log(interp1(xout_P,n_P,SubjM(j,1),'spline'));
    end

    % Get subject SubjP from class P, get probability for classes M and P
    SubjP = [];
    SubjP = T_P_TT(g_P_TT==i,3);
    Sum_P_M = 0;
    Sum_P_P = 0;
    for j=1:size(SubjP,1)
        Sum_P_M = Sum_P_M + log(interp1(xout_M,n_M,SubjP(j,1),'spline'));
    end
    for j=1:size(SubjP,1)
        Sum_P_P = Sum_P_P + log(interp1(xout_P,n_P,SubjP(j,1),'spline'));
    end

    resM = Sum_M_P - Sum_M_M;
    if (resM > 0)
        countMP_TT = countMP_TT + 1;
    elseif (resM < 0)
        countMM_TT = countMM_TT + 1;
    end

    resP = Sum_P_P - Sum_P_M;
    if (resP > 0)
        countPP_TT = countPP_TT + 1;
    elseif (resP < 0)
```

```
        countPM_TT = countPM_TT + 1;
    end
end


%% JI

% Leave-one-out cross validation
for i=1:numSamples

    % Probability distribution for class M (Intertweet delay)
    [n_M_ITD,xout_M_ITD] = hist(T_M_ITD(g_M_ITD~=i & g_M_ITD<=numSamples,2),logspace(0,8,30));
    n_M_ITD = n_M_ITD/sum(n_M_ITD);
    xout_M_ITD_lin = log(xout_M_ITD);

    % Probability distribution for class P (Intertweet delay)
    [n_P_ITD,xout_P_ITD] = hist(T_P_ITD(g_P_ITD~=i & g_P_ITD<=numSamples,2),logspace(0,8,30));
    n_P_ITD = n_P_ITD/sum(n_P_ITD);
    xout_P_ITD_lin = log(xout_P_ITD);

    % Probability distribution for class M (Tweeting time)
    [n_M_TT,xout_M_TT] = hist(T_M_TT(g_M_TT~=i & g_M_TT<=numSamples,3),0:1:23);
    n_M_TT = n_M_TT/sum(n_M_TT);

    % Probability distribution for class P (Tweeting time)
    [n_P_TT,xout_P_TT] = hist(T_P_TT(g_P_TT~=i & g_P_TT<=numSamples,3),0:1:23);
    n_P_TT = n_P_TT/sum(n_P_TT);

    % Get subject SubjM from class M, get probability for classes M and P
    SubjM_ITD = T_M_ITD(g_M_ITD==i,2);
    SubjM_TT = T_M_TT(g_M_TT==i,3);
    Sum_M_M = 0;
    Sum_M_P = 0;
    for j=1:size(SubjM_ITD,1)
        Sum_M_M = Sum_M_M + log(interp1(xout_M_ITD_lin,n_M_ITD,log(SubjM_ITD(j,1)),'spline'));
        Sum_M_P = Sum_M_P + log(interp1(xout_P_ITD_lin,n_P_ITD,log(SubjM_ITD(j,1)),'spline'));
    end
    for j=1:size(SubjM_TT,1)
        Sum_M_M = Sum_M_M + log(interp1(xout_M_TT,n_M_TT,SubjM_TT(j,1),'spline'));
        Sum_M_P = Sum_M_P + log(interp1(xout_P_TT,n_P_TT,SubjM_TT(j,1),'spline'));
    end

    % Get subject SubjP from class P, get probability for classes M and P
    SubjP_ITD = T_P_ITD(g_P_ITD==i,2);
    SubjP_TT = T_P_TT(g_P_TT==i,3);
    Sum_P_M = 0;
    Sum_P_P = 0;
    for j=1:size(SubjP_ITD,1)
        Sum_P_M = Sum_P_M + log(interp1(xout_M_ITD_lin,n_M_ITD,log(SubjP_ITD(j,1)),'spline'));
        Sum_P_P = Sum_P_P + log(interp1(xout_P_ITD_lin,n_P_ITD,log(SubjP_ITD(j,1)),'spline'));
    end
    for j=1:size(SubjP_TT,1)
        Sum_P_M = Sum_P_M + log(interp1(xout_M_TT,n_M_TT,SubjP_TT(j,1),'spline'));
        Sum_P_P = Sum_P_P + log(interp1(xout_P_TT,n_P_TT,SubjP_TT(j,1),'spline'));
    end


    resM = Sum_M_P - Sum_M_M;
    if (resM > 0)
        countMP_JI = countMP_JI + 1;
    elseif (resM < 0)
        countMM_JI = countMM_JI + 1;
    end

    resP = Sum_P_P - Sum_P_M;
    if (resP > 0)
        countPP_JI = countPP_JI + 1;
    elseif (resP < 0)
        countPM_JI = countPM_JI + 1;
    end
end


%% JNI

ctrs{1} = logspace(0,8,60);
ctrs{2} = 0:1:23;

% Leave-one-out cross validation
for i=1:numSamples

    % Joint probability distribution for class M
    [n_M_J,xout_M_J] = hist3(T_M_J(g_M_J~=i & g_M_J<=numSamples,2:3),ctrs);
    n_M_J = n_M_J/sum(sum(n_M_J));
    xout_M_J_lin = log(xout_M_J{1,1});

    % Joint probability distribution for class P
    [n_P_J,xout_P_J] = hist3(T_P_J(g_P_J~=i & g_P_J<=numSamples,2:3),ctrs);
    n_P_J = n_P_J/sum(sum(n_P_J));
    xout_P_J_lin = log(xout_P_J{1,1});

    % Get subject SubjM from class M, get join probability for classes M and P
    SubjM = [];
    SubjM(:,1) = T_M_J(g_M_J==i,2);
    SubjM(:,2) = T_M_J(g_M_J==i,3);
    Sum_M_M = 0;
    Sum_M_P = 0;
    for j=1:size(SubjM,1)
        [X,Y] = meshgrid(xout_M_J{1,2},xout_M_J_lin);
        aa = interp2(X,Y,n_M_J,SubjM(j,2),log(SubjM(j,1)),'spline');
        Sum_M_M = Sum_M_M + log(aa);
    end
    for j=1:size(SubjM,1)
```

```
284             [X,Y] = meshgrid(xout_P_J{1,2},xout_P_J_lin);
285             aa = interp2(X,Y,n_P_J,SubjM(j,2),log(SubjM(j,1)),'spline');
286             Sum_M_P = Sum_M_P + log(aa);
287         end
288
289         % Get subject SubjP from class P, get join probability for classes M and P
290         SubjP = [];
291         SubjP(:,1) = T_P_J(g_P_J==i,2);
292         SubjP(:,2) = T_P_J(g_P_J==i,3);
293         Sum_P_M = 0;
294         Sum_P_P = 0;
295         for j=1:size(SubjP,1)
296             [X,Y] = meshgrid(xout_M_J{1,2},xout_M_J_lin);
297             aa = interp2(X,Y,n_M_J,SubjP(j,2),log(SubjP(j,1)),'spline');
298             Sum_P_M = Sum_P_M + log(aa);
299         end
300         for j=1:size(SubjP,1)
301             [X,Y] = meshgrid(xout_P_J{1,2},xout_P_J_lin);
302             aa = interp2(X,Y,n_P_J,SubjP(j,2),log(SubjP(j,1)),'spline');
303             Sum_P_P = Sum_P_P + log(aa);
304         end
305
306         resM = Sum_M_P - Sum_M_M;
307         if (resM > 0)
308             countMP_JNI = countMP_JNI + 1;
309         elseif (resM < 0)
310             countMM_JNI = countMM_JNI + 1;
311         end
312
313         resP = Sum_P_P - Sum_P_M;
314         if (resP > 0)
315             countPP_JNI = countPP_JNI + 1;
316         elseif (resP < 0)
317             countPM_JNI = countPM_JNI + 1;
318         end
319     end
320
321
322     %% Obtain final results
323
324     numCorrect_ITD = countMM_ITD + countPP_ITD;
325     numCorrect_TT = countMM_TT + countPP_TT;
326     numCorrect_JI = countMM_JI + countPP_JI;
327     numCorrect_JNI = countMM_JNI + countPP_JNI;
328
329     percCorrect_ITD = numCorrect_ITD/(2*numSamples) * 100;
330     percCorrect_TT = numCorrect_TT/(2*numSamples) * 100;
331     percCorrect_JI = numCorrect_JI/(2*numSamples) * 100;
332     percCorrect_JNI = numCorrect_JNI/(2*numSamples) * 100;
333
334
335     end
```

Classifier2.m

Appendix D

3-Classifier

```
% This function classifies users in one of three classes: personal
% accounts (P), managed accounts (M), and bots (B). It takes as input the
% size of the dataset used in the classification and performs leave-one-out
% cross validation. Four different classifications are used: using only
% inter-tweet delay intervals (ITD), using only tweeting times (TT), using
% both as independent variables (JI), and using both as non-independent
% variables (JNI).

function [numSamples, percCorrect_JNI, percCorrect_JI, percCorrect_ITD, percCorrect_TT] = Classifier3(
    varargin)

load companies.mat
J{1} = T;

load people.mat
J{2} = T;

load bots.mat
J{3} = T;

% Class M - ITD
T_M_ITD = J{1};
g_M_ITD = grp2idx(T_M_ITD(:,1));

% Class P - ITD
T_P_ITD = J{2};
g_P_ITD = grp2idx(T_P_ITD(:,1));

% Class B - ITD
T_B_ITD = J{3};
g_B_ITD = grp2idx(T_B_ITD(:,1));

% Class M - TT
T_M_TT = J{1};
g_M_TT = grp2idx(T_M_TT(:,1));

% Class P - TT
T_P_TT = J{2};
g_P_TT = grp2idx(T_P_TT(:,1));

% Class B - TT
T_B_TT = J{3};
g_B_TT = grp2idx(T_B_TT(:,1));

% Class M - J
T_M_J = J{1};
g_M_J = grp2idx(T_M_J(:,1));

% Class P - J
T_P_J = J{2};
g_P_J = grp2idx(T_P_J(:,1));

% Class B - J
T_B_J = J{3};
g_B_J = grp2idx(T_B_J(:,1));

% Confusion matrix counts
countPP_ITD = 0;
countPM_ITD = 0;
countPB_ITD = 0;
countMP_ITD = 0;
countMM_ITD = 0;
countMB_ITD = 0;
countBP_ITD = 0;
countBM_ITD = 0;
countBB_ITD = 0;

countPP_TT = 0;
countPM_TT = 0;
countPB_TT = 0;
countMP_TT = 0;
countMM_TT = 0;
countMB_TT = 0;
countBP_TT = 0;
countBM_TT = 0;
countBB_TT = 0;

countPP_JI = 0;
```

```
countPM_JI = 0;
countPB_JI = 0;
countMP_JI = 0;
countMM_JI = 0;
countMB_JI = 0;
countBP_JI = 0;
countBM_JI = 0;
countBB_JI = 0;

countPP_JNI = 0;
countPM_JNI = 0;
countPB_JNI = 0;
countMP_JNI = 0;
countMM_JNI = 0;
countMB_JNI = 0;
countBP_JNI = 0;
countBM_JNI = 0;
countBB_JNI = 0;

if nargin == 0
    for i=1:3
        T = J{i};
        g = grp2idx(T(:,1));
        n(i) = length(unique(g));
    end
    numSamples = min(n);
else
    numSamples = varargin{1};
end

%% ITD

% Leave-one-out cross validation
for i=1:numSamples

    % Probability distribution for class M
    [n_M,xout_M] = hist(T_M_ITD(g_M_ITD~=i & g_M_ITD<=numSamples,2),logspace(0,8,30));
    n_M = n_M/sum(n_M);
    xout_M_lin = log(xout_M);

    % Probability distribution for class P
    [n_P,xout_P] = hist(T_P_ITD(g_P_ITD~=i & g_P_ITD<=numSamples,2),logspace(0,8,30));
    n_P = n_P/sum(n_P);
    xout_P_lin = log(xout_P);

    % Probability distribution for class B
    [n_B,xout_B] = hist(T_B_ITD(g_B_ITD~=i & g_B_ITD<=numSamples,2),logspace(0,8,30));
    n_B = n_B/sum(n_B);
    xout_B_lin = log(xout_B);

    % Get subject Subj_M from class M, get probability for each class
    SubjM = [];
    SubjM = T_M_ITD(g_M_ITD==i,2);
    Sum_M_M = 0;
    Sum_M_P = 0;
    Sum_M_B = 0;
    for j=1:size(SubjM,1)
        Sum_M_M = Sum_M_M + log(interp1(xout_M_lin,n_M,log(SubjM(j,1)),'spline'));
    end
    for j=1:size(SubjM,1)
        Sum_M_P = Sum_M_P + log(interp1(xout_P_lin,n_P,log(SubjM(j,1)),'spline'));
    end
    for j=1:size(SubjM,1)
        Sum_M_B = Sum_M_B + log(interp1(xout_B_lin,n_B,log(SubjM(j,1)),'spline'));
    end

    % Get subject Subj_P from class P, get probability for each class
    SubjP = [];
    SubjP = T_P_ITD(g_P_ITD==i,2);
    Sum_P_M = 0;
    Sum_P_P = 0;
    Sum_P_B = 0;
    for j=1:size(SubjP,1)
        Sum_P_M = Sum_P_M + log(interp1(xout_M_lin,n_M,log(SubjP(j,1)),'spline'));
    end
    for j=1:size(SubjP,1)
        Sum_P_P = Sum_P_P + log(interp1(xout_P_lin,n_P,log(SubjP(j,1)),'spline'));
    end
    for j=1:size(SubjP,1)
        Sum_P_B = Sum_P_B + log(interp1(xout_B_lin,n_B,log(SubjP(j,1)),'spline'));
    end

    % Get subject Subj_B from class B, get probability for each class
    SubjB = [];
    SubjB = T_B_ITD(g_B_ITD==i,2);
    Sum_B_M = 0;
    Sum_B_P = 0;
    Sum_B_B = 0;
    for j=1:size(SubjB,1)
        Sum_B_M = Sum_B_M + log(interp1(xout_M_lin,n_M,log(SubjB(j,1)),'spline'));
    end
    for j=1:size(SubjB,1)
        Sum_B_P = Sum_B_P + log(interp1(xout_P_lin,n_P,log(SubjB(j,1)),'spline'));
    end
    for j=1:size(SubjB,1)
        Sum_B_B = Sum_B_B + log(interp1(xout_B_lin,n_B,log(SubjB(j,1)),'spline'));
    end

    if max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_M
        countMM_ITD = countMM_ITD + 1;
    elseif max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_P
        countMP_ITD = countMP_ITD + 1;
    elseif max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_B
```

```
        countMB_ITD = countMB_ITD + 1;
    end

    if max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_M
        countPM_ITD = countPM_ITD + 1;
    elseif max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_P
        countPP_ITD = countPP_ITD + 1;
    elseif max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_B
        countPB_ITD = countPB_ITD + 1;
    end

    if max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_M
        countBM_ITD = countBM_ITD + 1;
    elseif max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_P
        countBP_ITD = countBP_ITD + 1;
    elseif max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_B
        countBB_ITD = countBB_ITD + 1;
    end
end


%% TT

% Leave-one-out cross validation
for i=1:numSamples

    % Probability distribution for class M
    [n_M,xout_M] = hist(T_M_TT(g_M_TT~=i & g_M_TT<=numSamples,3),0:1:23);
    n_M = n_M/sum(n_M);

    % Probability distribution for class I
    [n_P,xout_P] = hist(T_P_TT(g_P_TT~=i & g_P_TT<=numSamples,3),0:1:23);
    n_P = n_P/sum(n_P);

    % Probability distribution for class B
    [n_B,xout_B] = hist(T_B_TT(g_B_TT~=i & g_B_TT<=numSamples,3),0:1:23);
    n_B = n_B/sum(n_B);

    % Get subject Subj_M from class M, get probability for each class
    SubjM = [];
    SubjM = T_M_TT(g_M_TT==i,3);
    Sum_M_M = 0;
    Sum_M_P = 0;
    Sum_M_B = 0;
    for j=1:size(SubjM,1)
        Sum_M_M = Sum_M_M + log(interp1(xout_M,n_M,SubjM(j,1),'spline'));
    end
    for j=1:size(SubjM,1)
        Sum_M_P = Sum_M_P + log(interp1(xout_P,n_P,SubjM(j,1),'spline'));
    end
    for j=1:size(SubjM,1)
        Sum_M_B = Sum_M_B + log(interp1(xout_B,n_B,SubjM(j,1),'spline'));
    end

    % Get subject Subj_P from class P, get probability for each class
    SubjP = [];
    SubjP = T_P_TT(g_P_TT==i,3);
    Sum_P_M = 0;
    Sum_P_P = 0;
    Sum_P_B = 0;
    for j=1:size(SubjP,1)
        Sum_P_M = Sum_P_M + log(interp1(xout_M,n_M,SubjP(j,1),'spline'));
    end
    for j=1:size(SubjP,1)
        Sum_P_P = Sum_P_P + log(interp1(xout_P,n_P,SubjP(j,1),'spline'));
    end
    for j=1:size(SubjP,1)
        Sum_P_B = Sum_P_B + log(interp1(xout_B,n_B,SubjP(j,1),'spline'));
    end

    % Get subject Subj_B from class B, get probability for each class
    SubjB = [];
    SubjB = T_B_TT(g_B_TT==i,3);
    Sum_B_M = 0;
    Sum_B_P = 0;
    Sum_B_B = 0;
    for j=1:size(SubjB,1)
        Sum_B_M = Sum_B_M + log(interp1(xout_M,n_M,SubjB(j,1),'spline'));
    end
    for j=1:size(SubjB,1)
        Sum_B_P = Sum_B_P + log(interp1(xout_P,n_P,SubjB(j,1),'spline'));
    end
    for j=1:size(SubjB,1)
        Sum_B_B = Sum_B_B + log(interp1(xout_B,n_B,SubjB(j,1),'spline'));
    end

    if max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_M
        countMM_TT = countMM_TT + 1;
    elseif max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_P
        countMP_TT = countMP_TT + 1;
    elseif max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_B
        countMB_TT = countMB_TT + 1;
    end

    if max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_M
        countPM_TT = countPM_TT + 1;
    elseif max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_P
        countPP_TT = countPP_TT + 1;
    elseif max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_B
        countPB_TT = countPB_TT + 1;
    end

    if max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_M
```

```
        countBM_TT = countBM_TT + 1;
    elseif max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_P
        countBP_TT = countBP_TT + 1;
    elseif max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_B
        countBB_TT = countBB_TT + 1;
    end
end


%% JI

% Leave-one-out cross validation
for i=1:numSamples

    % Probability distribution for class M (Intertweet delay)
    [n_M_ITD,xout_M_ITD] = hist(T_M_ITD(g_M_ITD~=i & g_M_ITD<=numSamples,2),logspace(0,8,30));
    n_M_ITD = n_M_ITD/sum(n_M_ITD);
    xout_M_ITD_lin = log(xout_M_ITD);

    % Probability distribution for class I (Intertweet delay)
    [n_P_ITD,xout_P_ITD] = hist(T_P_ITD(g_P_ITD~=i & g_P_ITD<=numSamples,2),logspace(0,8,30));
    n_P_ITD = n_P_ITD/sum(n_P_ITD);
    xout_P_ITD_lin = log(xout_P_ITD);

    % Probability distribution for class B (Intertweet delay)
    [n_B_ITD,xout_B_ITD] = hist(T_B_ITD(g_B_ITD~=i & g_B_ITD<=numSamples,2),logspace(0,8,30));
    n_B_ITD = n_B_ITD/sum(n_B_ITD);
    xout_B_ITD_lin = log(xout_B_ITD);

    % Probability distribution for class M (Tweeting time)
    [n_M_TT,xout_M_TT] = hist(T_M_TT(g_M_TT~=i & g_M_TT<=numSamples,3),0:1:23);
    n_M_TT = n_M_TT/sum(n_M_TT);

    % Probability distribution for class P (Tweeting time)
    [n_P_TT,xout_P_TT] = hist(T_P_TT(g_P_TT~=i & g_P_TT<=numSamples,3),0:1:23);
    n_P_TT = n_P_TT/sum(n_P_TT);

    % Probability distribution for class B (Tweeting time)
    [n_B_TT,xout_B_TT] = hist(T_B_TT(g_B_TT~=i & g_B_TT<=numSamples,3),0:1:23);
    n_B_TT = n_B_TT/sum(n_B_TT);

    % Get subject SubjM from class M, get probability for each class
    SubjM_ITD = T_M_ITD(g_M_ITD==i,2);
    SubjM_TT = T_M_TT(g_M_TT==i,3);
    Sum_M_M = 0;
    Sum_M_P = 0;
    Sum_M_B = 0;
    for j=1:size(SubjM_ITD,1)
        Sum_M_M = Sum_M_M + log(interp1(xout_M_ITD_lin,n_M_ITD,log(SubjM_ITD(j,1)),'linear'));
        Sum_M_P = Sum_M_P + log(interp1(xout_P_ITD_lin,n_P_ITD,log(SubjM_ITD(j,1)),'linear'));
        Sum_M_B = Sum_M_B + log(interp1(xout_B_ITD_lin,n_B_ITD,log(SubjM_ITD(j,1)),'linear'));
    end
    for j=1:size(SubjM_TT,1)
        Sum_M_M = Sum_M_M + log(interp1(xout_M_TT,n_M_TT,SubjM_TT(j,1),'linear'));
        Sum_M_P = Sum_M_P + log(interp1(xout_P_TT,n_P_TT,SubjM_TT(j,1),'linear'));
        Sum_M_B = Sum_M_B + log(interp1(xout_B_TT,n_B_TT,SubjM_TT(j,1),'linear'));
    end

    % Get subject SubjP from class P, get probability for each class
    SubjP_ITD = T_P_ITD(g_P_ITD==i,2);
    SubjP_TT = T_P_TT(g_P_TT==i,3);
    Sum_P_M = 0;
    Sum_P_P = 0;
    Sum_P_B = 0;
    for j=1:size(SubjP_ITD,1)
        Sum_P_M = Sum_P_M + log(interp1(xout_M_ITD_lin,n_M_ITD,log(SubjP_ITD(j,1)),'linear'));
        Sum_P_P = Sum_P_P + log(interp1(xout_P_ITD_lin,n_P_ITD,log(SubjP_ITD(j,1)),'linear'));
        Sum_P_B = Sum_P_B + log(interp1(xout_B_ITD_lin,n_B_ITD,log(SubjP_ITD(j,1)),'linear'));
    end
    for j=1:size(SubjP_TT,1)
        Sum_P_M = Sum_P_M + log(interp1(xout_M_TT,n_M_TT,SubjP_TT(j,1),'linear'));
        Sum_P_P = Sum_P_P + log(interp1(xout_P_TT,n_P_TT,SubjP_TT(j,1),'linear'));
        Sum_P_B = Sum_P_B + log(interp1(xout_B_TT,n_B_TT,SubjP_TT(j,1),'linear'));
    end

    % Get subject SubjB from class B, get probability for each class
    SubjB_ITD = T_B_ITD(g_B_ITD==i,2);
    SubjB_TT = T_B_TT(g_B_TT==i,3);
    Sum_B_M = 0;
    Sum_B_P = 0;
    Sum_B_B = 0;
    for j=1:size(SubjB_ITD,1)
        Sum_B_M = Sum_B_M + log(interp1(xout_M_ITD_lin,n_M_ITD,log(SubjB_ITD(j,1)),'linear'));
        Sum_B_P = Sum_B_P + log(interp1(xout_P_ITD_lin,n_P_ITD,log(SubjB_ITD(j,1)),'linear'));
        Sum_B_B = Sum_B_B + log(interp1(xout_B_ITD_lin,n_B_ITD,log(SubjB_ITD(j,1)),'linear'));
    end
    for j=1:size(SubjB_TT,1)
        Sum_B_M = Sum_B_M + log(interp1(xout_M_TT,n_M_TT,SubjB_TT(j,1),'linear'));
        Sum_B_P = Sum_B_P + log(interp1(xout_P_TT,n_P_TT,SubjB_TT(j,1),'linear'));
        Sum_B_B = Sum_B_B + log(interp1(xout_B_TT,n_B_TT,SubjB_TT(j,1),'linear'));
    end

    if max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_M
        countMM_JI = countMM_JI + 1;
    elseif max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_P
        countMP_JI = countMP_JI + 1;
    elseif max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_B
        countMB_JI = countMB_JI + 1;
    end

    if max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_M
        countPM_JI = countPM_JI + 1;
    elseif max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_P
```

```
        countPP_JI = countPP_JI + 1;
    elseif max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_B
        countPB_JI = countPB_JI + 1;
    end

    if max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_M
        countBM_JI = countBM_JI + 1;
    elseif max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_P
        countBP_JI = countBP_JI + 1;
    elseif max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_B
        countBB_JI = countBB_JI + 1;
    end
end


%% JNI

ctrs{1} = logspace(0,8,60);
ctrs{2} = 0:1:23;

% Leave-one-out cross validation
for i=1:numSamples

    % Joint probability distribution for class M
    [n_M_J,xout_M_J] = hist3(T_M_J(g_M_J~=i & g_M_J<=numSamples,2:3),ctrs);
    n_M_J = n_M_J/sum(sum(n_M_J));
    xout_M_J_lin = log(xout_M_J{1,1});

    % Joint probability distribution for class P
    [n_P_J,xout_P_J] = hist3(T_P_J(g_P_J~=i & g_P_J<=numSamples,2:3),ctrs);
    n_P_J = n_P_J/sum(sum(n_P_J));
    xout_P_J_lin = log(xout_P_J{1,1});

    % Joint probability distribution for class B
    [n_B_J,xout_B_J] = hist3(T_B_J(g_B_J~=i & g_B_J<=numSamples,2:3),ctrs);
    n_B_J = n_B_J/sum(sum(n_B_J));
    xout_B_J_lin = log(xout_B_J{1,1});

    % Get subject SubjM from class M, get joint probability for each class
    SubjM = [];
    SubjM(:,1) = T_M_J(g_M_J==i,2);
    SubjM(:,2) = T_M_J(g_M_J==i,3);
    Sum_M_M = 0;
    Sum_M_P = 0;
    Sum_M_B = 0;
    for j=1:size(SubjM,1)
        [X,Y] = meshgrid(xout_M_J{1,2},xout_M_J_lin);
        aa = interp2(X,Y,n_M_J,SubjM(j,2),log(SubjM(j,1)),'linear');
        Sum_M_M = Sum_M_M + log(aa);
    end
    for j=1:size(SubjM,1)
        [X,Y] = meshgrid(xout_P_J{1,2},xout_P_J_lin);
        aa = interp2(X,Y,n_P_J,SubjM(j,2),log(SubjM(j,1)),'linear');
        Sum_M_P = Sum_M_P + log(aa);
    end
    for j=1:size(SubjM,1)
        [X,Y] = meshgrid(xout_B_J{1,2},xout_B_J_lin);
        aa = interp2(X,Y,n_B_J,SubjM(j,2),log(SubjM(j,1)),'linear');
        Sum_M_B = Sum_M_B + log(aa);
    end

    % Get subject SubjP from class P, get join probability for each class
    SubjP = [];
    SubjP(:,1) = T_P_J(g_P_J==i,2);
    SubjP(:,2) = T_P_J(g_P_J==i,3);
    Sum_P_M = 0;
    Sum_P_P = 0;
    Sum_P_B = 0;
    for j=1:size(SubjP,1)
        [X,Y] = meshgrid(xout_M_J{1,2},xout_M_J_lin);
        aa = interp2(X,Y,n_M_J,SubjP(j,2),log(SubjP(j,1)),'linear');
        bb = log(aa);
        Sum_P_M = Sum_P_M + bb;
    end
    for j=1:size(SubjP,1)
        [X,Y] = meshgrid(xout_P_J{1,2},xout_P_J_lin);
        aa = interp2(X,Y,n_P_J,SubjP(j,2),log(SubjP(j,1)),'linear');
        bb = log(aa);
        Sum_P_P = Sum_P_P + bb;
    end
    for j=1:size(SubjP,1)
        [X,Y] = meshgrid(xout_B_J{1,2},xout_B_J_lin);
        aa = interp2(X,Y,n_B_J,SubjP(j,2),log(SubjP(j,1)),'linear');
        bb = log(aa);
        Sum_P_B = Sum_P_B + bb;
    end

    % Get subject SubjB from class B, get join probability for each class
    SubjB = [];
    SubjB(:,1) = T_B_J(g_B_J==i,2);
    SubjB(:,2) = T_B_J(g_B_J==i,3);
    Sum_B_M = 0;
    Sum_B_P = 0;
    Sum_B_B = 0;
    for j=1:size(SubjB,1)
        [X,Y] = meshgrid(xout_M_J{1,2},xout_M_J_lin);
        aa = interp2(X,Y,n_M_J,SubjB(j,2),log(SubjB(j,1)),'linear');
        bb = log(aa);
        Sum_B_M = Sum_B_M + bb;
    end
    for j=1:size(SubjB,1)
        [X,Y] = meshgrid(xout_P_J{1,2},xout_P_J_lin);
        aa = interp2(X,Y,n_P_J,SubjB(j,2),log(SubjB(j,1)),'linear');
```

```
490         bb = log(aa);
491         Sum_B_P = Sum_B_P + bb;
492     end
493     for j=1:size(SubjB,1)
494         [X,Y] = meshgrid(xout_B_J{1,2},xout_B_J_lin);
495         aa = interp2(X,Y,n_B_J,SubjB(j,2),log(SubjB(j,1)),'linear');
496         bb = log(aa);
497         Sum_B_B = Sum_B_B + bb;
498     end
499
500     if max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_M
501         countMM_JNI = countMM_JNI + 1;
502     elseif max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_P
503         countMP_JNI = countMP_JNI + 1;
504     elseif max([Sum_M_M,Sum_M_P,Sum_M_B]) == Sum_M_B
505         countMB_JNI = countMB_JNI + 1;
506     end
507
508     if max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_M
509         countPM_JNI = countPM_JNI + 1;
510     elseif max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_P
511         countPP_JNI = countPP_JNI + 1;
512     elseif max([Sum_P_M,Sum_P_P,Sum_P_B]) == Sum_P_B
513         countPB_JNI = countPB_JNI + 1;
514     end
515
516     if max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_M
517         countBM_JNI = countBM_JNI + 1;
518     elseif max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_P
519         countBP_JNI = countBP_JNI + 1;
520     elseif max([Sum_B_M,Sum_B_P,Sum_B_B]) == Sum_B_B
521         countBB_JNI = countBB_JNI + 1;
522     end
523 end
524
525
526 %% Obtain final results
527
528 numCorrect_ITD = countMM_ITD + countPP_ITD + countBB_ITD;
529 numCorrect_TT = countMM_TT + countPP_TT + countBB_TT;
530 numCorrect_JI = countMM_JI + countPP_JI + countBB_JI;
531 numCorrect_JNI = countMM_JNI + countPP_JNI + countBB_JNI;
532
533 percCorrect_ITD = numCorrect_ITD/(3*numSamples) * 100;
534 percCorrect_TT = numCorrect_TT/(3*numSamples) * 100;
535 percCorrect_JI = numCorrect_JI/(3*numSamples) * 100;
536 percCorrect_JNI = numCorrect_JNI/(3*numSamples) * 100;
537
538
539 end
```

Classifier3.m

Appendix E

Predictive Model (Single Distribution)

```
% This script loads matrices that include both tweeting times (TT) and the
% time intervals after each tweet (intertweet delay, ITD). The predictor
% uses the probability distribution of intertweet delay for each class
% (Personal, Managed, Bot) in order to predict what the next interval will
% be.

clear all
close all
clc

load companies.mat
T_M = T;
g_M = grp2idx(T_M(:,1));

load people.mat
T_P = T;
g_P = grp2idx(T_P(:,1));

load bots.mat
T_B = T;
g_B = grp2idx(T_B(:,1));

numSamples = 60;


%% Leave-one-out cross validation for class P

for i=1:numSamples

    % Probability distribution for class P (Intertweet delay)
    [n_P_ITD,xout_P_ITD] = hist(T_P(g_P~=i & g_P<=numSamples,2),logspace(0,8,100));
    xout_P_ITD_lin = log(xout_P_ITD);

    % Get Subj_P from class P
    Subj_P = T_P(g_P==i,2:3);

    for j=1:(size(Subj_P,1)-1)

        % What we want to predict
        h = Subj_P(j,2);
        delta = Subj_P(j,1); % time interval after h

        % Get cumulative distribution for class I including interpolation
        % for point observed (model data)
        [minDiff, pos] = min(abs(delta - xout_P_ITD));
        if (delta - xout_P_ITD(pos) < 0)
            pos = pos-1;
        end
        xout_P = [xout_P_ITD(1:pos),delta,xout_P_ITD(pos+1:length(xout_P_ITD))];

        p_delta = interp1(xout_P_ITD_lin,n_P_ITD,log(delta),'spline');
        n_P = [n_P_ITD(1:pos),p_delta,n_P_ITD(pos+1:length(n_P_ITD))];
        n_P = n_P/sum(n_P);

        for k=1:length(n_P)
            n_P_cmlt{j}(k) = 0;
            for l=1:k
                n_P_cmlt{j}(k) = n_P_cmlt{j}(k) + n_P(l);
            end
        end

        % Create step function: cumulative prob for the point observed
        % (actual data)
        S{j} = zeros(length(xout_P),1);
        for k=(pos+1):length(S{j})
            S{j}(k)=1;
        end

    end

    % Evaluate predictions using coefficient of determination R_squared
    avg = 0;
    den = 0;
```

```
    for j=1:length(S)
        avg = avg + sum(S{j});
        den = den + length(S{j});
    end
    avg = avg/den;

    SS_tot = 0;
    SS_err = 0;
    for j=1:length(S)
        for k=1:length(S{j})
            SS_tot = SS_tot + ((S{j}(k) - avg)^2);
            SS_err = SS_err + ((S{j}(k) - n_P_cmlt{j}(k))^2);
        end
    end

    R_squared_P(i) = 1 - (SS_err/SS_tot);

end


%% Leave-one-out cross validation for class M

for i=1:numSamples

    % Probability distribution for class M (Intertweet delay)
    [n_M_ITD,xout_M_ITD] = hist(T_M(g_M~=i & g_M<=numSamples,2),logspace(0,8,100));
    xout_M_ITD_lin = log(xout_M_ITD);

    % Get Subj_M from class M
    Subj_M = T_M(g_M==i,2:3);

    for j=1:(size(Subj_M,1)-1)

        % What we want to predict
        h = Subj_M(j,2);
        delta = Subj_M(j,1); % time interval before h

        % Get cumulative distribution for class M including interpolation
        % for point observed (model data)
        [minDiff, pos] = min(abs(delta - xout_M_ITD));
        if (delta - xout_M_ITD(pos) < 0)
            pos = pos-1;
        end
        xout_M = [xout_M_ITD(1:pos),delta,xout_M_ITD(pos+1:length(xout_M_ITD))];

        p_delta = interp1(xout_M_ITD_lin,n_M_ITD,log(delta),'spline');
        n_M = [n_M_ITD(1:pos),p_delta,n_M_ITD(pos+1:length(n_M_ITD))];
        n_M = n_M/sum(n_M);

        for k=1:length(n_M)
            n_M_cmlt{j}(k) = 0;
            for l=1:k
                n_M_cmlt{j}(k) = n_M_cmlt{j}(k) + n_M(l);
            end
        end

        % Create step function: cumulative prob for the point observed
        % (actual data)
        S{j} = zeros(length(xout_M),1);
        for k=(pos+1):length(S{j})
            S{j}(k)=1;
        end

    end

    % Evaluate predictions using coefficient of determination R_squared
    avg = 0;
    den = 0;
    for j=1:length(S)
        avg = avg + sum(S{j});
        den = den + length(S{j});
    end
    avg = avg/den;

    SS_tot = 0;
    SS_err = 0;
    for j=1:length(S)
        for k=1:length(S{j})
            SS_tot = SS_tot + ((S{j}(k) - avg)^2);
            SS_err = SS_err + ((S{j}(k) - n_M_cmlt{j}(k))^2);
        end
    end

    R_squared_M(i) = 1 - (SS_err/SS_tot);

end


%% Leave-one-out cross validation for class B

for i=1:numSamples

    % Probability distribution for class B (Intertweet delay)
    [n_B_ITD,xout_B_ITD] = hist(T_B(g_B~=i & g_B<=numSamples,2),logspace(0,8,100));
    xout_B_ITD_lin = log(xout_B_ITD);

    % Get Subj_B from class B
    Subj_B = T_B(g_B==i,2:3);

    for j=1:(size(Subj_B,1)-1)

        % What we want to predict
        h = Subj_B(j,2);
```

```
        delta = Subj_B(j,1); % time interval before h

        % Get cumulative distribution for class B including interpolation
        % for point observed (model data)
        [minDiff, pos] = min(abs(delta - xout_B_ITD));
        if (delta - xout_B_ITD(pos) < 0)
            pos = pos-1;
        end
        xout_B = [xout_B_ITD(1:pos),delta,xout_B_ITD(pos+1:length(xout_B_ITD))];

        p_delta = interp1(xout_B_ITD_lin,n_B_ITD,log(delta),'spline');
        %n_B_IT = n_B_IT * sum(n_B_IT);
        n_B = [n_B_ITD(1:pos),p_delta,n_B_ITD(pos+1:length(n_B_ITD))];
        n_B = n_B/sum(n_B);

        for k=1:length(n_B)
            n_B_cmlt{j}(k) = 0;
            for l=1:k
                n_B_cmlt{j}(k) = n_B_cmlt{j}(k) + n_B(l);
            end
        end

        % Create step function: cumulative prob for the point observed
        % (actual data)
        S{j} = zeros(length(xout_B),1);
        for k=(pos+1):length(S{j})
            S{j}(k)=1;
        end

    end

    % Evaluate predictions using coefficient of determination R_squared
    avg = 0;
    den = 0;
    for j=1:length(S)
        avg = avg + sum(S{j});
        den = den + length(S{j});
    end
    avg = avg/den;

    SS_tot = 0;
    SS_err = 0;
    for j=1:length(S)
        for k=1:length(S{j})
            SS_tot = SS_tot + ((S{j}(k) - avg)^2);
            SS_err = SS_err + ((S{j}(k) - n_B_cmlt{j}(k))^2);
        end
    end

    R_squared_B(i) = 1 - (SS_err/SS_tot);

end
```

PredictorSingle.m

Appendix F

Predictive Model (Multiple Distribution H)

```
% This script loads matrices that include both tweeting times (TT) and the
% time intervals after each tweet (intertweet delay, ITD). The predictor
% uses 24 probability distributions (each associated with the time 0-23 of
% the last tweet) of intertweet delays for each class (Personal, Managed,
% Bot) in order to predict what the next interval will be.

clear all
close all
clc

load companies.mat
T1 = T;
g1 = grp2idx(T1(:,1));
for i=1:24
    T_M{i} = T(T(:,3)==i-1,:);
    g_M{i} = grp2idx(T_M{i}(:,1));
end

load people.mat
T2 = T;
g2 = grp2idx(T2(:,1));
for i=1:24
    T_P{i} = T(T(:,3)==i-1,:);
    g_P{i} = grp2idx(T_P{i}(:,1));
end

load bots.mat
T3 = T;
g3 = grp2idx(T3(:,1));
for i=1:24
    T_B{i} = T(T(:,3)==i-1,:);
    g_B{i} = grp2idx(T_B{i}(:,1));
end

numSamples = 60;


%% Leave-one-out cross validation for class P

for i=1:numSamples

    % Probability distributions for class I (Intertweet delay)
    for j=1:length(T_P)
       [n_P_ITD{j},xout_P_ITD{j}] = hist(T_P{j}(g_P{j}~=i & g_P{j}<=numSamples,2),logspace(0,8,100));
       xout_P_ITD_lin{j} = log(xout_P_ITD{j});
    end

    % Get Subj_P from class P
    Subj_P = T2(g2==i,2:3);

    for j=1:(size(Subj_P,1)-1)

        % What we want to predict
        h = Subj_P(j,2);
        delta = Subj_P(j,1); % time interval after h

        % Get cumulative distribution for class P including interpolation
        % for point observed (model data)
        [minDiff, pos] = min(abs(delta - xout_P_ITD{h+1}));
        if (delta - xout_P_ITD{h+1}(pos) < 0)
            pos = pos-1;
        end
        xout_P = [xout_P_ITD{h+1}(1:pos),delta,xout_P_ITD{h+1}(pos+1:length(xout_P_ITD{h+1}))];

        p_delta = interp1(xout_P_ITD_lin{h+1},n_P_ITD{h+1},log(delta),'spline');
        n_P = [n_P_ITD{h+1}(1:pos),p_delta,n_P_ITD{h+1}(pos+1:length(n_P_ITD{h+1}))];
        n_P = n_P/sum(n_P);

        for k=1:length(n_P)
            n_P_cmlt{j}(k) = 0;
            for l=1:k
                n_P_cmlt{j}(k) = n_P_cmlt{j}(k) + n_P(l);
            end
```

```
        end

        % Create step function: cumulative prob for the point observed
        % (actual data)
        S{j} = zeros(length(xout_P),1);
        for k=(pos+1):length(S{j})
            S{j}(k)=1;
        end

    end

    % Evaluate predictions using coefficient of determination R_squared
    avg = 0;
    den = 0;
    for j=1:length(S)
        avg = avg + sum(S{j});
        den = den + length(S{j});
    end
    avg = avg/den;

    SS_tot = 0;
    SS_err = 0;
    for j=1:length(S)
        for k=1:length(S{j})
            SS_tot = SS_tot + ((S{j}(k) - avg)^2);
            SS_err = SS_err + ((S{j}(k) - n_P_cmlt{j}(k))^2);
        end
    end

    R_squared_P(i) = 1 - (SS_err/SS_tot);

end

%% Leave-one-out cross validation for class M

for i=1:numSamples

    % Probability distributions for class M (Intertweet delay)
    for j=1:length(T_M)
        [n_M_ITD{j},xout_M_ITD{j}] = hist(T_M{j}(g_M{j}~=i & g_M{j}<=numSamples,2),logspace(0,8,100));
        xout_M_ITD_lin{j} = log(xout_M_ITD{j});
    end

    % Get Subj_M from class M
    Subj_M = T1(g1==i,2:3);

    for j=1:(size(Subj_M,1)-1)

        % What we want to predict
        h = Subj_M(j,2);
        delta = Subj_M(j,1); % time interval before h

        % Get cumulative distribution for class I including interpolation
        % for point observed (model data)
        [minDiff, pos] = min(abs(delta - xout_M_ITD{h+1}));
        if (delta - xout_M_ITD{h+1}(pos) < 0)
            pos = pos-1;
        end
        xout_M = [xout_M_ITD{h+1}(1:pos),delta,xout_M_ITD{h+1}(pos+1:length(xout_M_ITD{h+1}))];

        p_delta = interp1(xout_M_ITD_lin{h+1},n_M_ITD{h+1},log(delta),'spline');
        n_M = [n_M_ITD{h+1}(1:pos),p_delta,n_M_ITD{h+1}(pos+1:length(n_M_ITD{h+1}))];
        n_M = n_M/sum(n_M);

        for k=1:length(n_M)
            n_M_cmlt{j}(k) = 0;
            for l=1:k
                n_M_cmlt{j}(k) = n_M_cmlt{j}(k) + n_M(l);
            end
        end

        % Create step function: cumulative prob for the point observed
        % (actual data)
        S{j} = zeros(length(xout_M),1);
        for k=(pos+1):length(S{j})
            S{j}(k)=1;
        end

    end

    % Evaluate predictions using coefficient of determination R_squared
    avg = 0;
    den = 0;
    for j=1:length(S)
        avg = avg + sum(S{j});
        den = den + length(S{j});
    end
    avg = avg/den;

    SS_tot = 0;
    SS_err = 0;
    for j=1:length(S)
        for k=1:length(S{j})
            SS_tot = SS_tot + ((S{j}(k) - avg)^2);
            SS_err = SS_err + ((S{j}(k) - n_M_cmlt{j}(k))^2);
        end
    end

    R_squared_M(i) = 1 - (SS_err/SS_tot);

end
```

```
%% Leave-one-out cross validation for class B

for i=1:numSamples

    % Probability distributions for class B (Intertweet delay)
    for j=1:length(T_B)
        [n_B_ITD{j},xout_B_ITD{j}] = hist(T_B{j}(g_B{j}~=i & g_B{j}<=numSamples,2),logspace(0,8,100));
        xout_B_ITD_lin{j} = log(xout_B_ITD{j});
    end

    % Get Subj_B from class B
    Subj_B = T3(g3==i,2:3);

    for j=1:(size(Subj_B,1)-1)

        % What we want to predict
        h = Subj_B(j,2);
        delta = Subj_B(j,1); % time interval before h

        % Get cumulative distribution for class B including interpolation
        % for point observed (model data)
        [minDiff, pos] = min(abs(delta - xout_B_ITD{h+1}));
        if (delta - xout_B_ITD{h+1}(pos) < 0)
            pos = pos-1;
        end
        xout_B = [xout_B_ITD{h+1}(1:pos),delta,xout_B_ITD{h+1}(pos+1:length(xout_B_ITD{h+1}))];

        p_delta = interp1(xout_B_ITD_lin{h+1},n_B_ITD{h+1},log(delta),'spline');
        n_B = [n_B_ITD{h+1}(1:pos),p_delta,n_B_ITD{h+1}(pos+1:length(n_B_ITD{h+1}))];
        n_B = n_B/sum(n_B);

        for k=1:length(n_B)
            n_B_cmlt{j}(k) = 0;
            for l=1:k
                n_B_cmlt{j}(k) = n_B_cmlt{j}(k) + n_B(l);
            end
        end

        % Create step function: cumulative prob for the point observed
        % (actual data)
        S{j} = zeros(length(xout_B),1);
        for k=(pos+1):length(S{j})
            S{j}(k)=1;
        end

    end

    % Evaluate predictions using coefficient of determination R_squared
    avg = 0;
    den = 0;
    for j=1:length(S)
        avg = avg + sum(S{j});
        den = den + length(S{j});
    end
    avg = avg/den;

    SS_tot = 0;
    SS_err = 0;
    for j=1:length(S)
        for k=1:length(S{j})
            SS_tot = SS_tot + ((S{j}(k) - avg)^2);
            SS_err = SS_err + ((S{j}(k) - n_B_cmlt{j}(k))^2);
        end
    end

    R_squared_B(i) = 1 - (SS_err/SS_tot);

end
```

PredictorMult_H.m

Appendix G

Predictive Model (Multiple Distribution HW)

```
% This script loads matrices that include both tweeting times (TT) and the
% time intervals after each tweet (intertweet delay, ITD). The predictor
% uses (24*7) probability distributions (each associated with the time 0-23
% and the day of the week of the last tweet) of intertweet time delays for
% each class (Personal, Managed, Bot) in order to predict what the next
% interval will be.

clear all
close all
clc

load companies.mat
T1 = T;
g1 = grp2idx(T1(:,1));
for i=1:24
    for j=1:7
        T_M{i,j} = T(T(:,3)==i-1 & T(:,5)==j,:);
        g_M{i,j} = grp2idx(T_M{i,j}(:,1));
    end
end

load people.mat
T2 = T;
g2 = grp2idx(T2(:,1));
for i=1:24
    for j=1:7
        T_P{i,j} = T(T(:,3)==i-1 & T(:,5)==j,:);
        g_P{i,j} = grp2idx(T_P{i,j}(:,1));
    end
end

load bots.mat
T3 = T;
g3 = grp2idx(T3(:,1));
for i=1:24
    for j=1:7
        T_B{i,j} = T(T(:,3)==i-1 & T(:,5)==j,:);
        g_B{i,j} = grp2idx(T_B{i,j}(:,1));
    end
end

numSamples = 60;


%% Leave-one-out cross validation for class P

for i=1:numSamples

    % Probability distributions for class P (Intertweet delay)
    for j=1:size(T_P,1)
        for k=1:size(T_P,2)
            [n_P_ITD{j,k},xout_P_ITD{j,k}] = hist(T_P{j,k}(g_P{j,k}~=i & g_P{j,k}<=numSamples,2),logspace
                (0,8,100));
            xout_P_ITD_lin{j,k} = log(xout_P_ITD{j,k});
        end
    end

    % Get Subj_P from class P
    Subj_P = T2(g2==i,2:5);

    for j=1:(size(Subj_P,1)-1)

        % What we want to predict
        h = Subj_P(j,2); % hour
        w = Subj_P(j,4); % day of the week
        delta = Subj_P(j,1); % time interval after h

        % Get cumulative distribution for class I including interpolation
        % for point observed (model data)
        [minDiff, pos] = min(abs(delta - xout_P_ITD{h+1,w}));
        if (delta - xout_P_ITD{h+1,w}(pos) < 0)
            pos = pos-1;
        end
```

```
        xout_P = [xout_P_ITD{h+1,w}(1:pos),delta,xout_P_ITD{h+1,w}(pos+1:length(xout_P_ITD{h+1,w}))];

        p_delta = interp1(xout_P_ITD_lin{h+1,w},n_P_ITD{h+1,w},log(delta),'spline');
        n_P = [n_P_ITD{h+1,w}(1:pos),p_delta,n_P_ITD{h+1,w}(pos+1:length(n_P_ITD{h+1,w}))];
        n_P = n_P/sum(n_P);

        for k=1:length(n_P)
            n_P_cmlt{j}(k) = 0;
            for l=1:k
                n_P_cmlt{j}(k) = n_P_cmlt{j}(k) + n_P(l);
            end
        end

        % Create step function: cumulative prob for the point observed
        % (actual data)
        S{j} = zeros(length(xout_P),1);
        for k=(pos+1):length(S{j})
            S{j}(k)=1;
        end

    end

    % Evaluate predictions using coefficient of determination R_squared
    avg = 0;
    den = 0;
    for j=1:length(S)
        avg = avg + sum(S{j});
        den = den + length(S{j});
    end
    avg = avg/den;

    SS_tot = 0;
    SS_err = 0;
    for j=1:length(S)
        for k=1:length(S{j})
            SS_tot = SS_tot + ((S{j}(k) - avg)^2);
            SS_err = SS_err + ((S{j}(k) - n_P_cmlt{j}(k))^2);
        end
    end

    R_squared_P(i) = 1 - (SS_err/SS_tot);

end


%% Leave-one-out cross validation for class M

for i=1:numSamples

    % Probability distributions for class M (Intertweet delay)
    for j=1:size(T_M,1)
        for k=1:size(T_M,2)
            [n_M_ITD{j,k},xout_M_ITD{j,k}] = hist(T_M{j,k}(g_M{j,k}~=i & g_M{j,k}<=numSamples,2),logspace
                (0,8,100));
            xout_M_ITD_lin{j,k} = log(xout_M_ITD{j,k});
        end
    end

    % Get Subj_M from class M
    Subj_M = T1(g1==i,2:5);

    for j=1:(size(Subj_M,1)-1)

        % What we want to predict
        h = Subj_M(j,2); % hour
        w = Subj_M(j,4); % day of the week
        delta = Subj_M(j,1); % time interval after h

        % Get cumulative distribution for class M including interpolation
        % for point observed (model data)
        [minDiff, pos] = min(abs(delta - xout_M_ITD{h+1,w}));
        if (delta - xout_M_ITD{h+1,w}(pos) < 0)
            pos = pos-1;
        end
        xout_M = [xout_M_ITD{h+1,w}(1:pos),delta,xout_M_ITD{h+1,w}(pos+1:length(xout_M_ITD{h+1,w}))];

        %n_M_IT{h+1,w} = n_M_IT{h+1,w}/sum(n_M_IT{h+1,w});
        p_delta = interp1(xout_M_ITD_lin{h+1,w},n_M_ITD{h+1,w},log(delta),'spline');
        n_M = [n_M_ITD{h+1,w}(1:pos),p_delta,n_M_ITD{h+1,w}(pos+1:length(n_M_ITD{h+1,w}))];
        n_M = n_M/sum(n_M);
        %n_M_IT{h+1,w} = n_M_IT{h+1,w} * sum(n_M_IT{h+1,w});

        for k=1:length(n_M)
            n_M_cmlt{j}(k) = 0;
            for l=1:k
                n_M_cmlt{j}(k) = n_M_cmlt{j}(k) + n_M(l);
            end
        end

        % Create step function: cumulative prob for the point observed
        % (actual data)
        S{j} = zeros(length(xout_M),1);
        for k=(pos+1):length(S{j})
            S{j}(k)=1;
        end

    end

    % Evaluate predictions using coefficient of determination R_squared
    avg = 0;
    den = 0;
    for j=1:length(S)
        avg = avg + sum(S{j});
```

```
        den = den + length(S{j});
    end
    avg = avg/den;

    SS_tot = 0;
    SS_err = 0;
    for j=1:length(S)
        for k=1:length(S{j})
            SS_tot = SS_tot + ((S{j}(k) - avg)^2);
            SS_err = SS_err + ((S{j}(k) - n_M_cmlt{j}(k))^2);
        end
    end

    R_squared_M(i) = 1 - (SS_err/SS_tot);

end


%% Leave-one-out cross validation for class B

for i=1:numSamples

    % Probability distributions for class B (Intertweet delay)
    for j=1:size(T_B,1)
        for k=1:size(T_B,2)
            [n_B_ITD{j,k},xout_B_ITD{j,k}] = hist(T_B{j,k}(g_B{j,k}~=i & g_B{j,k}<=numSamples,2),logspace
                (0,8,100));
            xout_B_ITD_lin{j,k} = log(xout_B_ITD{j,k});
        end
    end

    % Get Subj_B from class B
    Subj_B = T1(g1==i,2:5);

    for j=1:(size(Subj_B,1)-1)

        % What we want to predict
        h = Subj_B(j,2); % hour
        w = Subj_B(j,4); % day of the week
        delta = Subj_B(j,1); % time interval after h

        % Get cumulative distribution for class B including interpolation
        % for point observed (model data)
        [minDiff, pos] = min(abs(delta - xout_B_ITD{h+1,w}));
        if (delta - xout_B_ITD{h+1,w}(pos) < 0)
            pos = pos-1;
        end
        xout_B = [xout_B_ITD{h+1,w}(1:pos),delta,xout_B_ITD{h+1,w}(pos+1:length(xout_B_ITD{h+1,w}))];

        p_delta = interp1(xout_B_ITD_lin{h+1,w},n_B_ITD{h+1,w},log(delta),'spline');
        n_B = [n_B_ITD{h+1,w}(1:pos),p_delta,n_B_ITD{h+1,w}(pos+1:length(n_B_ITD{h+1,w}))];
        n_B = n_B/sum(n_B);

        for k=1:length(n_B)
            n_B_cmlt{j}(k) = 0;
            for l=1:k
                n_B_cmlt{j}(k) = n_B_cmlt{j}(k) + n_B(l);
            end
        end

        % Create step function: cumulative prob for the point observed
        % (actual data)
        S{j} = zeros(length(xout_B),1);
        for k=(pos+1):length(S{j})
            S{j}(k)=1;
        end

    end

    % Evaluate predictions using coefficient of determination R_squared
    avg = 0;
    den = 0;
    for j=1:length(S)
        avg = avg + sum(S{j});
        den = den + length(S{j});
    end
    avg = avg/den;

    SS_tot = 0;
    SS_err = 0;
    for j=1:length(S)
        for k=1:length(S{j})
            SS_tot = SS_tot + ((S{j}(k) - avg)^2);
            SS_err = SS_err + ((S{j}(k) - n_B_cmlt{j}(k))^2);
        end
    end

    R_squared_B(i) = 1 - (SS_err/SS_tot);

end
```

PredictorMult_HW.m