LEARNING INFLUENCE PROBABILITIES IN SOCIAL NETWORKS

Gheorghiţă Cătălin Bordianu

Master of Science

School of Computer Science

McGill University

Montreal, Quebec

March 2012

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

DEDICATION

This thesis is dedicated to my friends and family. Their trust in me never faded.

ACKNOWLEDGEMENTS

My gratitude goes to those who shaped my life in the past two years. Foremost, I thank my supervisor, professor Doina Precup. Her kind heart and good advice kept me on the right path throughout my studies at McGill. Thank you for your support, for always smiling, and for knowing how to get my spirit up when I felt I was not moving forward. You were the best supervisor and mentor a student could have asked for.

Many thanks to my colleagues in the Reasoning and Learning lab, who made me walk to my desk each morning with a smile on my face. Special thanks to Gheorghe Comănici and Cosmin Păduraru, our conversations and late nights in the lab both helped me in my research and gave me new perspectives on important things in life. I will sure miss taking turns with you guys to sleep on the office couch when tired. My friends and fellow grad students Jarryd, Tim, Ian, Arash, Philip, Danesh, Pierre-Luc, Radu and many others also deserve gratitude for helping me reach this point in my life.

Many more thanks go to my family. Without their support I would not be here to type these words. I thank my parents Vasile and Maria Bordianu from the bottom of my soul, you have been and will always be the pillars of support in my life. Many thanks to my two sisters Alina and Diana, my brother-in-law Lucian and my little niece Ella. As I am finishing an exciting journey of 17 years of study, she is starting her own in elementary school. I hope one day to read my name in her thesis acknowledgments.

Thanks to Laurent Féral-Pierssens for encouraging me to pursue graduate studies. Finally, thanks to James McGill for making this place happen, may you rest in peace in heavens.

ABSTRACT

Social network analysis is an important cross-disciplinary area of research, with applications in fields such as biology, epidemiology, marketing and even politics. Influence maximization is the problem of finding the set of seed nodes in an information diffusion process that guarantees maximum spread of influence in a social network, given its structure. Most approaches to this problem make two assumptions. First, the global structure of the network is known. Second, influence probabilities between any two nodes are known beforehand, which is rarely the case in practical settings. In this thesis we propose a different approach to the problem of learning those influence probabilities from past data, using only the local structure of the social network. The method is grounded in unsupervised machine learning techniques and is based on a form of hierarchical clustering, allowing us to distinguish between influential and the influenceable nodes. Finally, we provide empirical results using real data extracted from Facebook.

ABRÉGÉ

L'analyse des réseaux sociaux est un domaine d'études interdisciplinaires qui comprend des applications en biologie, épidémiologie, marketing et même politique. La maximisation de l'influence représente un problème où l'on doit trouver l'ensemble des noeuds de semence dans un processus de diffusion de l'information qui en même temps garantit le maximum de propagation de son influence dans un réseau social avec une structure connue. La plupart des approches à ce genre de problème font appel à deux hypothèses. Premièrement, la structure générale du réseau social est connue. Deuxièmement, les probabilités des influences entre deux noeuds sont connues à l'avance, fait qui n'est d'ailleurs pas valide dans des circonstances pratiques. Dans cette thèse, on propose un procédé différent visant la problème de l'apprentissage de ces probabilités d'influence à partir des données passées, en utilisant seulement la structure locale du réseau social. Le procédé se base sur l'apprentissage automatique sans surveillance et il est relié à une forme de regroupement hiérarchique, ce qui nous permet de faire la distinction entre les noeuds influenceurs et les noeuds influencés. Finalement, on fournit des résultats empiriques en utilisant des données réelles extraites du réseau social Facebook.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Algorithms

CHAPTER 1
Introduction

1.1 Introduction

Large scale social network analysis has become a subject of particular interest in research, given the rise and incredible growth of online social networks such as Facebook, Twitter and LinkedIn. Early network analysis research focused on explaining statical properties of the networks, as most of their dynamics and formation process were impossible to observe and track at that time. Paul Erdős and Alfred Rényi's impressive work on random graphs [15] was an important step forward as it formalized the sufficient conditions that need to be met, in order to prove that some property holds for "almost all" graphs. Important questions about the properties and formation of large scale networks have been addressed as early as 1999 by pioneers Albert-László Barabási and Réka Albert, who proposed the preferential attachment model [5] aimed at explaining the self-similar aspect of a lot of real life networks viewed at different scales. The difficulty in obtaining relevant datasets for study has hindered research in this field for a long time, but the emergence of online social networks that expose most of their data through public APIs alleviated this problem to a great extent.

Besides studying general properties of the network, one of the important directions of research in social network analysis is the study of properties of individual nodes [14]. Quantifying these properties can be used to distinguish between different categories of nodes based on their level of activity and behavior tendencies. This is an open research

question of great importance. Although no precise definition of influence has been formulated yet, the general consensus is that it describes the ability of a particular node to persuade its neighbors to adopt a particular type of behavior [10]. This idea gave rise to a number of fundamental diffusion models aimed at explaining how a particular behavior spreads through a social network, such as the Linear Threshold model [22, 25] and its generalization for multiple cascades [40], the Independent Cascade model [25], the Decreasing Cascade model [26] and topical factor graphs studied by Tang et al. [46]. These models provide a natural framework for measuring the network reach achieved by using a particular method of selecting seed nodes for the diffusion process. The likelihood of a node becoming activated and further spreading a piece of information — seen as a function of the neighbors of that node which are already activated — has nice properties, such as being submodular and incremental in most of these models. This makes these approaches computationally efficient.

The ease of obtaining very large datasets for analysis enables researchers for the very first time to empirically test and prove or disprove a lot of their models regarding the formation and dynamics of networks. This abundance of data also provides a great opportunity to identify new patterns and formulate novel hypotheses regarding the way information spreads from one node to another. One particular problem of interest — closely related to diffusion of information models — that has been the object of study in social network analysis is the problem of influence maximization posed by Domingos and Richardson [13, 42] and further researched by Kempe, Kleinberg and Tardos [25] and others [29]. The problem of influence maximization with parameter k has the following informal description: given a network modeled as a directed graph $G = (V, E)$ and influence probabilities

on each edge denoted by $w(v, u)$ (where v and u are any two nodes), select a set of k nodes that will be used as seeds in an information diffusion process, so that a maximum number of nodes is reached. The number of nodes reached is called the influence spread through the network of that specific seed set.

Influence maximization is an NP-hard discrete optimization problem in both the linear threshold model as well as the independent cascade model [25]. Practical approaches focus on finding approximate solutions. For this purpose, heuristics derived from measures of structural properties of the network or of individual nodes from the field of social network analysis are employed. Such measures include (in or out) degree centrality, betweenness and closeness centrality, or more complex ones such as PageRank [30]. A recent empirical comparison of some seeding strategies has been performed by Hinz et al. [23].

Most of these algorithms are successful in the sense that the number of nodes reached in a diffusion process increases by a factor of up to 30 in some cases [30], when compared to the baseline of using random nodes as seeds. However, none of these heuristics take into account the chronological interactions observed between nodes. Accounting for these interactions shows some promise in improving the capacity to learn influence probabilities and thus distinguish between influencers and influenceable nodes, as shown by Saito et al. [45] and more recently Goyal et al. [21]. While significant and enticing, these findings still leave room for research, as in some settings we might not have access to a detailed timestamped log of interactions between nodes. A second assumption often made is complete knowledge of the social network structure, which is rarely encountered in practical settings. Relaxing this constraint has a lot of practical value and promises to yield good

results, if the local structure of the social graph can be used as a good proxy for the global structure of the network [42, 4].

The goal of this thesis is to find a computationally efficient approach for learning influence probabilities in the local neighborhood of a given node. Ideally, the algorithm should also allow us to discriminate between the neighbors of a particular node, namely to distinguish between the influential and the influenceable nodes. Formally, for a given node v and the set of his neighbors $N(v)$, we want to learn a function $f_v : N(v) \to [0, 1]$, with $f_v(u) = w_{v,u}$; the function f_v assigns to every edge $(v, u), u \in N(v)$ a weight describing the degree to which v influences u. We will focus on applying techniques from the discipline of machine learning such as hierarchical clustering, which seem promising according to some recent work [45].

1.2 Outline

The rest of this thesis is split into five chapters. The second chapter provides the required background in various machine learning algorithms that are relevant to the topic. Chapter 3 introduces the method we propose for learning influence probabilities in the local neighborhood of a node. Chapter 4 presents a high-level overview of the software we wrote for extracting data from Facebook. All the major required details needed for replicating our setup and performing the same experiments are given. Chapter 5 presents findings on real data extracted from Facebook. Chapter 6 discusses how the new knowledge we gained from this work fits in the current research landscape and presents new possible directions of research.

CHAPTER 2
Background and Related Work

This chapter provides the required machine learning background for understanding the problem at hand, and presents related research work and results.

Section 2.1 introduces the problem and formalisms of unsupervised learning (one of the core areas of machine learning research), some important applications and clustering techniques.

Section 2.2 presents some important diffusion of information models, delves into the problem of influence maximization and formalizes the problem of learning influence probabilities. Different methods and related approaches that have already been explored in the literature are introduced when needed.

2.1 Unsupervised Learning

Machine learning concerns itself with the development of methods and techniques for learning from past data or experience. This enables the creation of programs that achieve complex tasks in areas where there are no human experts (such as DNA analysis), or in which humans performing them cannot explain how do they do it (e.g. understanding speech, recognizing handwriting). More recently, machine learning has significantly expanded due to its ability to analyze very large, heterogeneous datasets (including text, images, etc.) made available on the web.

In this thesis, we focus on methods from unsupervised learning, in which we are provided with a dataset consisting of vectors containing the values for different attributes of interest. The goal is to find interesting patterns or associations in the data. Generally, this is achieved by employing some measure of similarity between different input points. To some extent, unsupervised learning can be thought of as the problem of recovering the labels for a given dataset, in the absence of an expert. Some major applications for this type of learning are anomaly detection, noise reduction, finding groups of similar points (called clustering), learning new important features, and data visualization. Unsupervised learning techniques are often used for preprocessing and better understanding the data, before classifiers or regression functions are to be trained.

In unsupervised learning we are given a dataset of m points $\mathbf{x_1}, \mathbf{x_2} \dots \mathbf{x_m}$. Each of these points or inputs is a n-dimensional vector $\mathbf{x_i} = (x_{i,1}, x_{i,2} \dots x_{i,n})^\mathsf{T}$, where by $x_{i,j}$ we denote the value attribute j takes for the i-th input. These attributes or features are problem specific and can be continuous or discrete. In matrix form, the entire dataset can be written as

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix}$$

The task of an unsupervised learner is to find interesting patterns in the data beyond what would be considered noise. Assuming that the inputs $\mathbf{x_i}$ are independent and identically distributed according to some distribution $P(\mathbf{x})$, almost all unsupervised learning algorithms are equivalent to fitting a model $Q(\mathbf{x})$ to the data, to approximate $P(\mathbf{x})$.

Typical examples of unsupervised learning tasks are finding groups of similar points called clusters, or performing dimensionality reduction for compression or visualization purposes. Modeling structured data such as time series using State-Space Models (SSMs) or Hidden Markov Models (HMMs), inference in graphical models and approximate Bayesian inference algorithms such as Laplace approximation or Expectation Propagation can also be regarded as unsupervised learning tasks. The rest of this section will focus on clustering techniques. A high-level overview of the algorithms from a statistical modeling perspective, is given by Zoubin Grahramani [18].

2.1.1 Clustering

Given input data in the form presented above, the problem of clustering is to find interesting patterns by grouping similar points together. This is achieved by employing a dissimilarity function. A group of points that belong together is called a cluster and usually the mean of the points in the cluster is called the centroid.

A dissimilarity function over the input space $\mathbf{X}$ is defined as a function $D : \mathbf{X} \times \mathbf{X} \rightarrow \Re$ with the following properties:

1. Symmetry: $D(x, y) = D(y, x), \forall x, y \in \mathbf{X}$
2. Positivity: $D(x, y) \geq 0, \forall x, y \in \mathbf{X}$

If the triangle inequality $D(x, y) \leq D(x, z) + D(z, y)$ and reflexivity $D(x, y) = 0 \iff x = y, \forall x, y, z \in \mathbf{X}$ also hold, D is called a metric.

After deciding on a function with these properties, the next step in elaborating a clustering algorithm is to define a criterion or loss function to optimize. A commonly used and well studied optimization criterion is minimizing the sum of squared distances

between each point and its corresponding centroid. The output of a clustering algorithm is a set of centroids and an assignment of each input point to a particular centroid. This effectively achieves compression of the dataset, as all points in a particular cluster can now be encoded by their centroid.

Clustering has a large number of practical applications in outlier and anomaly detection, where it is used to determine when a system or part of it is functioning outside its normal parameters, by using a past history of normal activity. Efficient communication in a network, as well as data compression are also important applications of clustering techniques.

Clustering algorithms can be classified using a variety of criteria, but a widely agreed taxonomy is based on the nature of the clusters generated [16]. Partition methods return a single partition of the data into clusters, while hierarchical methods return an entire sequence of partitions, ranging from every point in its own cluster, to all the points in one cluster. An expert then chooses the best partition, or a numerical criterion is used to determine which partition best captures the true structure of the data [38].

2.1.2 Partitional clustering

This type of clustering methods attempt to directly decompose the dataset into a partition consisting of a pre-specified number of clusters. The most straightforward solution would be to choose an optimization criterion and to evaluate all possible partitions of the dataset into k clusters. However, this is an intractable combinatorial optimization problem, even for small datasets. Approximate solutions are generated by starting with an initial (usually random) partition, reassigning points to clusters and recomputing centroids

in such a way that the value of the optimization criterion improves. This type of approach is prone to converge to local minima instead of the global one, so techniques such as random restarts need to be employed to obtain useful solutions.

One of the simplest examples of partitional clustering is the K-means algorithm [35]. K-means is based on the idea of optimizing the sum of squared distances between every point and its corresponding centroid (also called within-cluster sum of squares), through an iterative process. Let the cluster centroids be denoted by $\mu_1, \mu_2 \dots \mu_k$ and the actual partition by $C = \{C_1, C_2 \dots C_k\}$. The optimization problem can be formulated as:

$$\min_{C} \sum_{i=1}^{k} \sum_{x_j \in C_i} \|x_j - \mu_i\|^2 \tag{2.1}$$

After randomly initializing k centroids, K-means performs the following steps repeatedly until no more points change cluster assignment:

1. Assign each point x_i to the cluster with the closest centroid
2. Re-compute centroids $\mu_i, i = 1 \rightarrow k$ as the means of the points $x_j \in S_i$

Different variations of this idea have been studied, giving rise to the K-medoids and fuzzy C-means algorithms [47]. Although quite computationally efficient, all approaches suffer from local minima issues and need to have k pre-specified. Good values for this parameter can be determined using cross-validation or the silhouette method [44].

2.1.3 Hierarchical clustering methods

Hierarchical clustering methods return a succession of partitions of the data into clusters, according to a proximity matrix. The partitions range from every point in its own cluster, to all the points in one cluster. No pre-specified number of clusters is needed. If

the partitioning is done bottom-up — starting with each individual point being a cluster — the method is called agglomerative clustering. Starting with all points in a single cluster gives rise to divisive methods. Divisive methods are rarely used in practice. For a cluster of n elements, $2^{n-1} - 1$ divisions into two subsets are possible, and considering them is computationally intractable for large values of n [16].

Agglomerative methods work by starting with every point in its own cluster and joining two clusters at each iteration based on some measure of similarity, until all the points are in one cluster. A common way to represent the arrangement of clusters produced by such as method is through a dendogram (see figure 2–1), a binary tree in which the length of an edge is proportional to the distance between the adjacent clusters. Different simi-

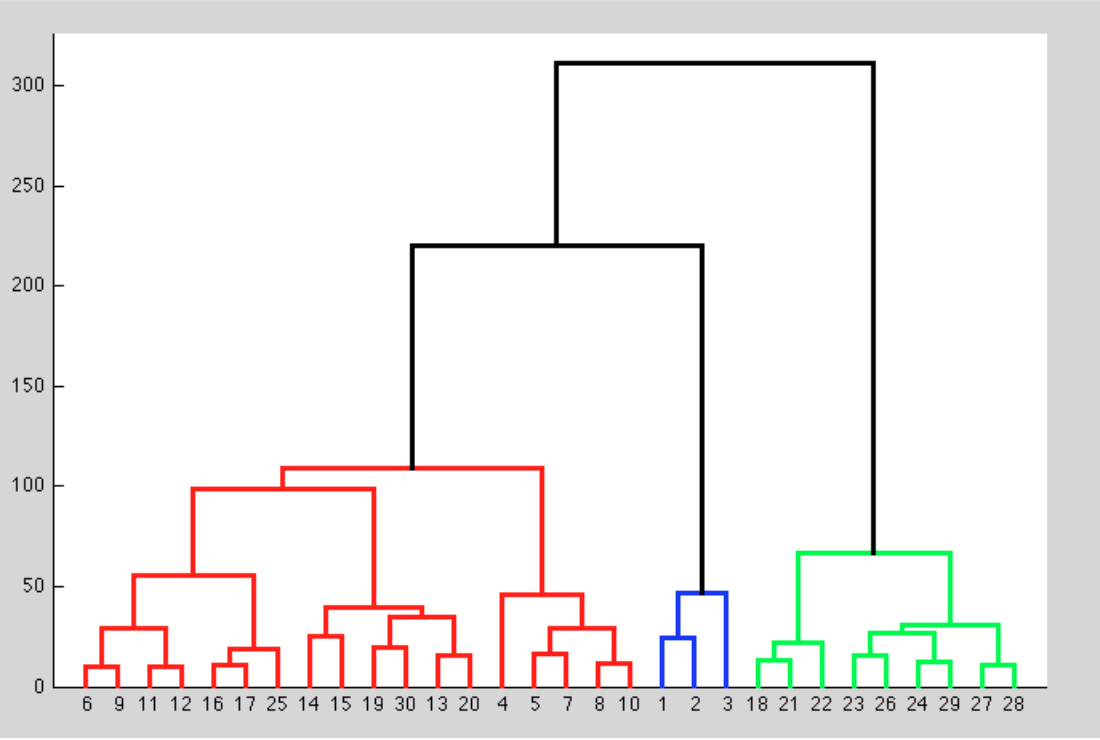


Figure 2–1: Example of a dendogram

larity measures between clusters — also called linkages — generate different hierarchies. The following recurrence formula [31] describes the distance between a cluster C_k and the cluster resulting by merging clusters C_i and C_j:

$$\begin{aligned} D(C_k,(C_i,C_j)) = \alpha_i D(C_k,C_i) + \alpha_j D(C_k,C_j) + \\ \beta D(C_i,C_j) + \gamma |D(C_k,C_i) - D(C_k,C_j)| \end{aligned} \tag{2.2}$$

Tweaking the values for the parameters $\alpha_i, \alpha_j, \beta$ and γ in (2.2) yields different linkages. For example, setting $\alpha_i = \alpha_j = 1/2$, $\beta = 0$ and $\gamma = -1/2$ yields the formula for single linkage, which is just the distance between the two closest points in the two clusters. Jain and Dubes [24, Table 3.1] present in detail the values of parameters for obtaining various linkages.

While it is true that hierarchical clustering methods are suitable for detailed data analysis and give more information than partitional methods, at least m^2 similarity coefficients need to be computed and updated during the process. In terms of computational efficiency, this class of algorithms might not be suitable for very large datasets.

Another drawback of hierarchical clustering algorithms is that results can be easily skewed by outlying data points, subject to the kind of linkage used. For example, Ward's linkage is significantly more sensitive to outliers than centroid linkage. Complete linkage can too be severely distorted even by moderate outliers. A preprocessing step of detecting and removing outlying observations from the dataset is a must when using hierarchical clustering algorithms in practice. Some studies have shown that clustering techniques themselves can be successfully used as a tool for achieving this task [48, 36].

2.1.4 Expectation Maximization

One widely used unsupervised learning method which deals with missing data is called Expectation Maximization (EM). EM is used to compute maximum likelihood (ML) or maximum a posteriori (MAP) estimates for the parameters of a model, when part of the data is unobserved. Published by Dempster et al. in 1977 [12], this idea generated over time an entire class of algorithms suitable in different problem settings [37].

The basic algorithm makes the assumption that observations depend on latent random variables or are incomplete (e.g. due to sensor noise). We introduce the following notation, and for the rest of the discussion — for ease of explanation — we assume observations are discrete:

- $Y = \{y_1, y_2 \dots y_m\}$ is the observed data, consisting of m points.
- Z is the set of unobserved latent variable.
- θ is a vector of unknown parameters.

The likelihood of θ given the observed data is:

$$L(\theta; Y, Z) = \prod_{i=1}^{m} \sum_{z} P(y_i, z|\theta) \tag{2.3}$$

However, the quantity in (2.3) is computationally intractable to maximize because of the marginal likelihood of the observed data (the sum over all possible values of Z) inside a product. EM works around this issue by filling in the missing values for Z using the current estimate of the parameters θ, and then using those values to find a maximum likelihood re-estimate of the parameters. Applying these two steps iteratively leads to convergence, as a consequence of Jensen's inequality. After each iteration, the estimate of θ has a greater or same log-likelihood as the estimate from the previous iteration.

Formalized, the basic EM algorithm is the following:

Algorithm 1 Expectation Maximization ($Y, L(\theta; Y, Z), \epsilon$)

Input:

Y - the observed (incomplete) data

$L(\theta; Y, Z)$ - the likelihood of θ, given the complete data

ϵ - small real value, used for determining algorithm convergence

Output:

$\theta^{(t+1)}$ - a maximum likelihood estimate of the parameter vector θ

1: **initialize** $\theta^{(0)}$ to random values
2: **repeat** for $t = 0, 1 \ldots$
3: $\quad Q(\theta|\theta^{(t)}) = E_{Z|Y,\theta^{(t)}}[logL(\theta; Y, Z)]$
4: $\quad \theta^{(t+1)} = \arg\max_{\theta} Q(\theta|\theta^{(t)})$
5: **until** $\|\theta^{(t+1)} - \theta^{(t)}\| < \epsilon$
6: **return** $\theta^{(t+1)}$

Step 3 is called the expectation step, since it involves computing the expected value of the log-likelihood function. Note that the expectation is taken with respect to the conditional distribution $Z|Y, \theta^{(t)}$, where $\theta^{(t)}$ is the current estimate for the parameters. In step 4 — called the maximization step — new ML estimates for the parameters of the model are computed, keeping the values for the realizations of the latent variables fixed.

To summarize, EM is an iterative method for finding the parameters of a model and provides a natural framework for dealing with missing data. It takes advantage of the fact that the log-likelihood of the complete dataset is easy to maximize. EM suffers from the same local minima issues as other unsupervised learning algorithms, so multiple random restarts need to be applied to get a good estimate for the parameters.

2.2 Information diffusion models and influence maximization

The way ideas and technology spread has been a subject of great importance in research. While the origins of this field span across disciplines, the first detailed synthesis was published by Everett Rogers in 1962 in "Diffusion of Innovations" [43], which is now considered a standard textbook.

Studying properties of diffusion processes has a large number of practical applications. In epidemiology, they are used to understand the spread of pandemics, in the military, to dismantle terrorist cells, and in marketing, to conduct efficient campaigns using limited budgets. We are interested in a subset of diffusion models which can be used to describe the way information spreads through a social network of individuals.

In most practical settings, we can directly observe the times when nodes get "infected" or "activated" and adopt a certain behavior. However, it is impossible to get accurate information about who infects or influences whom, especially in large-scale online networks. Even survey-based methods are unreliable, as the individuals surveyed are subjective or may not even be aware that their behavior was influenced by some of their peers. Several researchers conducted studies on inferring the underlying influence network by observing multiple instances of information or behavior propagating between nodes, called cascades [45, 20, 21]. Leskovec et al. [33] studied the topology of cascades that arise frequently and their relationship to properties of the underlying network. Tang et al. [46] worked on inferring subnetworks based on the intuition that if some node u consistently and repeatedly adopts a certain behavior shortly after some node v, then v influences u to some extent.

The models presented in subsections 2.2.1 and 2.2.2 are both progressive, meaning that once activated, a node cannot become inactive, although Kempe et al. [25] show that this assumption can be easily lifted.

2.2.1 The Linear Threshold model

One of the simplest models — often used in viral or "word of mouth" marketing — is called the Linear Threshold (LT) model and was proposed by Mark Granovetter in 1978 [22] to explain social processes such as riots or segregation. The LT model is based on the idea that each node in the network has an individual threshold, defined as a weighted fraction of his neighbors that need to activate to make it become active. If the threshold value is reached for a particular individual, the benefits gained exceed the cost of activating and the node will adopt the behavior, assuming that it is a rational agent.

Let v be a node in the network, θ_v its corresponding threshold value chosen uniformly at random in the interval $[0, 1]$ and $N(v)$ its set of neighbors. Each neighbor $u \in N(v)$ influences v according to a weight $b_{u,v}$ such that:

$$\sum_{u \in N(v)} b_{u,v} \leq 1 \tag{2.4}$$

Initially a set of seed nodes S_0 are activated and the model works in discrete time steps $i = 1, 2 \ldots$ until at some time step τ no more new nodes activate (i.e. $S_\tau = \emptyset$).

Associated with each node v is a monotone threshold function, $f_v : 2^{N(v)} \rightarrow [0, 1]$,

$$f_v(S) = \sum_{u \in S} b_{u,v} \tag{2.5}$$

with $f_v(\emptyset) = 0$. The function maps subsets of v's neighbors to real numbers in the interval $[0, 1]$.

Let us denote the set of active neighbors of v at time step t by $AN^t(v)$. A node v activates at time step $t + 1$ if:

$$f_v(AN^t(v)) \geq \theta_v \tag{2.6}$$

The model assumes that the weights $b_{u,v}$ are known beforehand, while the thresholds θ_v are chosen uniformly at random in $[0, 1]$. This reflects the lack of knowledge of the true thresholds, as underlined in the seminal work published by Kempe et. al [25]. There have been studies, such as [41], which use a fixed value (namely, $1/2$) for all thresholds.

The model has been adapted recently to work with multiple cascades of information by Pathak et al. [40]. Let us consider a network of nodes, and a world which starts randomly in one of two possible states. The state of the world is hidden from the nodes, which are trying to figure out which of the two it is. Furthermore, assume a sequential process, in which every node decides to accept or reject some option, based on a combination between the decision of the previous nodes and some private information. Every decision has an associated payoff. Rejecting the option has a payoff of 0, while accepting the option has a negative or positive payoff, depending on the hidden random state of the world. An information cascade occurs when the optimal strategy for a node for maximizing its payoff is to make the same decision as the majority of the preceding nodes, disregarding his own private signal [14, Chapter 16].

In Pathak's adapted model [40], if k cascades of information are in effect simultaneously, a node can be in $k + 1$ states: active in any of the k cascades, or inactive. Pathak et al. employ a stochastic graph coloring process (which they show to be equivalent to a

rapidly mixing Markov chain) to learn the most likely state of the entire network, out of $(k+1)^{|V|}$ possible states.

2.2.2 The Independent Cascade model

Another widely used probabilistic information diffusion model is called the Independent Cascade (IC) model, motivated by work in interactive particle systems [34].

The social network is modeled as a digraph $G = (V, E)$, where on each edge $(u, v) \in E$ there is an influence probability $p_{u,v}$. Time unfolds in discrete steps $i = 1, 2 \ldots$ and the initial seed set S_0 is considered to become active at $t = 1$. The process stops when at some time step no more activations occur. The set of inactive neighbors of v at time step t is denoted by $IN^t(v)$.

When some node v becomes active at time step t, it is given one chance to activate every other node $u \in IN^t(v)$ with probability $p_{u,v}$. If the activation succeeds, u will become active at time step $t+1$; if not, v will never be given the chance again to activate it. If for some node u there is more than one node who attempts to activate it at some t, they will do so in random order.

2.2.3 A generalized framework for diffusion models

Kempe et al. [25] propose a broader framework for diffusion models. The Linear Threshold and the Independent Cascade model are generalized simultaneously and are shown to have equivalent formulations.

In the General Threshold model, the functions f_v can be any monotone set functions with $f_v(\emptyset) = 0$, which take values in $[0, 1]$. The threshold functions in the LT model are

just a special case, taking the form in (2.5) subject to the constraint (2.4), for every node in the network.

In the General Cascade model, the probability of a node u infecting one of his neighbors v is an incremental function of the set $S \subset N(v)$ of v's neighbors which have already tried and failed, denoted $p_v(u, S)$. The IC model is a special case of this more general model, in which the incremental functions $p_v(u, S)$ are the constants $p_{u,v}$, independent of S.

Let us consider an instance of the Linear Threshold model. If a set of nodes S have already tried and failed in activating a node v, then the IC probabilities of any other neighbor $u \in N(v) \setminus S$ activating it are given by:

$$p_{u,v} = \frac{b_{u,v}}{1 - \sum_{w \in S} b_{w,v}} \tag{2.7}$$

Conversely, if we consider the same scenario for the Independent Cascade model, the threshold functions f_v are given by:

$$f_v(S) = 1 - \prod_{u \in S} (1 - p_{u,v}) \tag{2.8}$$

Note that in this case normalization needs to be performed to enforce constraint (2.4).

2.2.4 Influence maximization

We can now formally express the influence maximization problem, first studied by Domingos and Richardson [13, 42] from an algorithmic point of view, and later on by Kempe et al [25] as a discrete optimization problem.

Given either of the models presented in 2.2.1 and 2.2.2, or any generalized version of them, as time unfolds, at each step there is a set of newly activated nodes S_t. Note that an influence cascade cannot last more than n steps (by convention, the activation of the seed set is considered one step). We need to start with at least one seed node and at each step at least one other node needs to activate for the cascade to continue. The influence spread of a seed set S is defined as:

$$\sigma(S) = \Big| \bigcup_{i=0}^{n-1} S_i \Big| \tag{2.9}$$

Intuitively, $\sigma(S)$ is the number of nodes that will be infected at the end of the diffusion process, given S as the seed set. The problem of influence maximization with parameter k is choosing a seed set of size k over all possible seed sets of this size that maximizes the quantity in (2.9).

As a function of S, σ has a property called submodularity which can be interpreted as a law of diminishing returns. The probability of a node activating saturates, as more of its neighbors become activated.

A set function f is submodular if for any $S \subseteq T \subseteq V$ and any $w \in V \setminus T$:

$$f(S \cup \{w\}) - f(S) \geq f(T \cup \{w\}) - f(T) \tag{2.10}$$

The proofs that σ is submodular in both the LT and the IC models will be omitted for brevity [25, Theorem 2.2, Theorem 2.5].

Kempe et al. [25, Theorem 2.4, Theorem 2.7] proved that influence maximization is an NP-hard problem in both the LT and IC models. However, given that the influence spread is submodular, non-decreasing, and $\sigma(\emptyset) = 0$, a greedy hill-climbing algorithm can

provide an approximation to the optimal solution within a factor of $1 - 1/e$ [11, 39]. The algorithm is given in Algorithm 2.

Algorithm 2 Greedy Influence Maximization (f, k)

Input:

f - a submodular, non-decreasing influence spread function with $f(\emptyset) = 0$

k - the size of the desired seed set

Output:

S - a seed set of size k

1: $S = \emptyset$
2: **for** $i = 1 \rightarrow k$ **do**
3: $\quad u = \arg\max_{w \in V \setminus S} f(S \cup \{w\}) - f(S)$
4: $\quad S = S \cup \{u\}$
5: **end for**
6: **return** S

Selecting a seed set using the greedy algorithm guarantees that at least 63% of the nodes that would be activated by the optimal seed set, will be active at the end of the diffusion process.

Algorithm 2 does not scale to large datasets. At every iteration, it needs to determine the node with the biggest marginal increase in influence spread. This is a #P-hard problem in both models [25, 9]. Kempe et al. circumvent this issue through Monte-Carlo simulations of cascades, estimating the actual spread of a seed set within some ϵ. Unfortunately, the algorithm still takes days to select 50 seeds in a network of $30K$ nodes, as pointed out by Chen et al. [9]. Recent research focuses on improving the greedy algorithm [32, 8], finding new heuristics for the LT and IC models [28, 8, 7] or rooting directed acyclic graphs (DAGs) at each node, to compute approximations to the influence spread in linear time [9].

2.2.5 Learning influence probabilities

One assumption made by both diffusion models presented above is that influence probabilities (or influence weights in the case of the LT model) on each edge are known beforehand. This, however, is not the case in most practical settings. One largely open research question is how and from where can these influence probabilities be computed, as pointed out by Goyal et al. [21].

Independently and concurrently, Goyal et al. studied this question for the General Threshold model (presented in subsection 2.2.3) [21], while Saito et al. tried answering it for the Independent Cascade model [45]. Tang et al. [46] worked on efficiently inferring subnetworks of influence and their appropriate weights induced by different topics of interest in a social network. Using the network structure and topic distributions on all nodes, they developed a probabilistic graphical model based on factor graphs called Topical Affinity Propagation (TAP). While omitting TAP for brevity, we present below the ideas developed by Goyal et al. [21] and Saito et al. [45].

General Threshold model

Goyal et al. learned influence probabilities using a so called "action log" which is a history of past information cascades. Let us consider a specific cascade, an action in that cascade denoted a and two nodes v and u which are neighbors in the social network. If v performs a at time $t_a(v)$ and u performs the same action at a later time $t_a(u) > t_a(v)$, we conclude that v infected node u for action a. Let A_v denote the total number of cascades in which v became active, and A_{v2u} the number of times v infected u (summed up over all actions in all cascades in which v was active). Following the same notation, $A_{v|u}$ denotes

the number of times either nodes activated and $A_{v\&u}$ the number of times both nodes were active in the same cascade.

The first model developed in [21] is called the Bernoulli model:

$$p_{v,u} = \frac{A_{v2u}}{A_v} \tag{2.11}$$

Intuitively, equation (2.11) states that the probability of v infecting u is the ratio between the number of times u got infected from v in some cascade, and the total number of cascades in which v was active node.

The second model is based on the Jaccard index, usually used to compare how similar two samples are:

$$p_{v,u} = \frac{A_{v2u}}{A_{v|u}} \tag{2.12}$$

Goyal et al. also introduced the notion of partial credit. Let $AN^t(u)$ be the set of active neighbors of a node u at time t (with $|AN^t(u)| = d$), and c be some cascade. Denote by $t_w(c)$ the time at which node w activated in cascade c. Then the credit an active node v gets for contributing to the activation of u is given by

$$credit_{v,u}(c) = \frac{1}{\sum_{w \in AN^t(u)} I(t_w(c) < t_u(c))} \tag{2.13}$$

where I is an indicator function. If a node u gets activated in some cascade, each of its d activated neighbors (which might have influenced its decision) receive $1/d$ credit.

Mixing the notion of partial credit into the models above yields two more. The third model is called the Bernoulli model with partial credit. Let $\mathbf{A}$ be the set of actions in the training data. The probability of a node v infecting some node u is estimated as the ratio between the total credit v receives from influencing u over all cascades, and the total

number of cascades in which v was active.

$$p_{v,u} = \frac{\sum_{c \in \mathbf{A}} credit_{v,u}(c)}{A_v} \tag{2.14}$$

Similarly, the Jaccard model with partial credit is computed using a modified version of equation (2.14), where the denominator is $A_{v|u}$ instead of A_v. All these four models are static, in the sense that $p_{v,u}$ does not change over time.

Goyal et al. [21] devised learning and testing algorithms for these four models, and extended them into continuous time, by multiplying the probabilities $p_{v,u}$ with a time-dependent exponential decay term. Furthermore, they tested both the static models and their extensions over Flickr data. The results show the continuous time models to perform best in terms of precision and recall. From the four static models, the Bernoulli ones win over the ones based on the Jaccard index. Out of the two Bernoulli models, the one that mixes in the notion of partial credit is slightly better.

An interesting consequence of extending the models into the continuos domain is that they can also predict the expected time at which some node will activate, with an impressively small error margin. However, for computational efficiency, Goyal et al. produced a discretized time model, in which a node influences its neighbors only within some time interval after it actives. After the time window passes, the node stops being contagious. We omit explaining in detail the continuous and discretized time models for brevity.

Independent Cascade model

Saito et al. [45] studied the same problem of inferring influence probabilities from past cascades, but for the IC model. They derived the likelihood of a sequence of cascades

as a function of the diffusion probabilities and applied the Expectation Maximization algorithm to find maximum likelihood estimates for them.

Let $C_k : k = 1, 2 \dots K$ be a set of K independent cascades with corresponding durations T_{C_k} and $\theta = \{p_{v,u} | (v, u) \in E\}$ be the parameter vector. We denote by $S_k(t)$ the set of newly activated nodes in cascade C_k at time t, and by $P_u^k(t+1)$ the probability that node u will become active at time $t+1$ in cascade C_k:

$$P_u^k(t+1) = 1 - \prod_{v \in AN^t(u)} (1 - p_{v,u}) \tag{2.15}$$

Equation (2.15) can be interpreted as 1 minus the probability that all of u's active neighbors in cascade k at time t ($AN^t(u)$) fail to activate it. As a function of u, t and k, $P_u^k(t)$ is monotone and submodular [21, Theorem 1].

The log-likelihood of some θ in the series of K cascades is:

$$l(\theta) = \sum_{k=1}^{K} \sum_{t=0}^{T_{C_k}-1} \left(\sum_{w \in S_k(t+1)} \log P_w^k(t+1) + \sum_{v \in S_k(t)} \sum_{w \in IN^{(t+1)}(v)} \log(1 - p_{v,w}) \right) \tag{2.16}$$

In equation (2.16) first we sum over all cascades and for each cascade, over every time step of that cascade. For the nodes that were successfully activated in the next time step ($w \in S_k(t+1)$), we add the log of the probabilities that they will be activated. For the nodes that were not successfully activated ($w \in IN^{(t+1)}(v)$) even though they had at least one contagious neighbor v, we add the log of the probabilities that each of their contagious neighbors at time t ($v \in S_k(t)$) failed to activate them.

Saito et al. applied Expectation Maximization for maximizing (2.16) with respect to θ. First, they initialized each $p_{u,v}$ to random values in some interval $[\alpha, \beta]$. The interval

was determined by using a bond percolation process equivalent to the IC model; we omit the details for brevity.

A $Q(\theta|\hat{\theta})$ function for K episodes is provided in [45, Equation 7]. Solving $\partial Q/\partial p_{u,v} = 0$, yields the following update rule for the influence probabilities:

$$p_{u,v} = \frac{1}{|K^+_{u,v}| + |K^-_{u,v}|} \sum_{k \in K^+_{u,v}} \frac{\hat{p}_{u,v}}{\hat{P}^k_v} \tag{2.17}$$

In equation (2.17), $|K^+_{u,v}|$ is the number of cascades in which u successfully infected v, while $|K^-_{u,v}|$ is the number of cascades in which u was contagious, but failed to infect v. Also, $\hat{p}_{u,v}$ is the current estimate for the influence probability and $\hat{P}^k_v$ is computed using (2.15).

Saito et al. [45] tested this approach on a medium-sized network of approximately 80.000, nodes consisting of trackbacks between blogs. On average — when using at least 60 past cascades — the difference between the learned and the true probabilities is roughly 0.14. The standard deviation is also big. In some runs the probabilities are learned almost perfectly, while in others the error is 0.25 or more. However, performance always improves as the number of past cascades used in the learning process increases.

One of the assumptions that both Goyal et al. [21] and Saito et al. [45] do in their studies is access to a very rich set of past data, which consists of information cascades whose propagation was tracked over the entire network. However, in most practical settings such a dataset is difficult, if not impossible to obtain. Recording the activation times of every node in an information cascade is most of the time impractical due to missing information about the global structure of the network. A new approach is needed for dealing

with settings where we only have partial information about the network structure and can effectively track just the local interactions between a set of nodes and their neighbors.

The proposed method presented in chapter 3 works under the same assumption about influence adopted by Tang et al. [46]. Intuitively, if there is any influence between a pair of two nodes, the influenced node adopts to some degree the behavior of the influencer node. Recent studies over large-scale datasets from online social networks such as the one performed by Bisgin et al. [6] demonstrate that this is a realistic assumption, as homophily does not dictate the way connections are initially formed in any significant way.

CHAPTER 3
Proposed method

In this chapter we present a novel approach for learning influence probabilities in the local neighborhood of a node, grounded in unsupervised learning techniques. Let v be a node and $N(v)$ the set of its neighbors. Somewhat similar to the action log assumed by Goyal et al. [21], we assume to have access to a timestamped history of interactions between v and every $u \in N(v)$, denoted by $\mathbf{A_v}$. This history is composed of different types of interactions, denoted by $A_v^1, A_v^2 \dots A_v^k$ ($k \geq 1$). The k subsets of interactions form a partition over $\mathbf{A_v}$. Our task is to learn a function $f_v : N(v) \rightarrow [0, 1]$, with $f_v(u) = w_{v,u}$ that assigns to every edge (v, u) a weight quantifying the extent to which v influences u.

In contrast to the approaches in subsection 2.2.5 which use a log of past cascades over the entire network, this method is local and uses only the history of interactions between a node and its neighbors. This makes it computationally efficient due to inherent parallelism. The same algorithm can be run concurrently and independently on every node of interest in the network to learn the desired probabilities.

Section 3.1 explains how agglomerative clustering is applied to learn influence probabilities in this context. The proposed algorithm learns a probability mass function on v's outgoing edges. Running the algorithm for all of v's neighbors allows us to combine those mass functions and obtain the weights $b_{u,v}$ assumed by the Linear Threshold model, in a manner we explain towards the end of the section. Note that in the generalized framework presented in subsection 2.2.3, our proposed method can be used to learn the parameters

for the entire class of diffusion models introduced in section 2.2.

3.1 Learning influence probabilities using Agglomerative Clustering

Let $\mathbf{A_v}(u) = (|A_v^1(u)|, |A_v^2(u)| \dots |A_v^n(u)|)^\mathsf{T}$ be a vector containing aggregated counts of actions performed by some neighbor $u \in N(v)$. Any social interaction between u and v that we can observe and record (e.g. retweeting one of v's statuses on Twitter, tagging v in a picture on Facebook) can be regarded as an action. Let $|A_v^i(u)|$ be the integer value denoting the number of times node u has interacted with node v by performing action A_v^i. In practical settings not all actions are equally important. Let $\mathbf{S_{A_v}} = (s_{A_v^1}, s_{A_v^2} \dots s_{A_v^n})^\mathsf{T}$ be a scoring vector, with $s_{A_v^i}$ being the score associated with action A_v^i. For example, commenting on someone's picture on Facebook involves more commitment than just liking it. In this case we might decide to give more importance to the first action, by assigning a higher score. The score vectors act as heuristics which allow us to encode prior knowledge about the problem domain in our framework. Whenever it is clear that we are referring to the local neighborhood of v, we drop the subscript from $\mathbf{A_v}$.

Let n be the total number of distinct actions and m be the number of neighbors for node v. The input dataset for agglomerative clustering is obtained by performing the Hadamard product between $\mathbf{A}(u)$ and $\mathbf{S_A}$, $\forall u \in N(v)$:

$$\mathbf{X} = \begin{pmatrix} |A^1(u_1)|s_{A^1} & |A^2(u_1)|s_{A^2} & \cdots & |A^n(u_1)|s_{A^n} \\ |A^1(u_2)|s_{A^1} & |A^2(u_2)|s_{A^2} & \cdots & |A^n(u_2)|s_{A^n} \\ \vdots & \vdots & \ddots & \vdots \\ |A^1(u_m)|s_{A^1} & |A^2(u_m)|s_{A^2} & \cdots & |A^n(u_m)|s_{A^n} \end{pmatrix} \tag{3.1}$$

Hence, $x_{i,j}$ is the product between the number of times node u_i performed action A^j and the score associated with that action (s_{A^j}).

The next step in learning influence probabilities is to cluster the dataset $\mathbf{X}$ which consists of m input vectors. We first compute a $m \times m$ matrix D of all pairwise Euclidean distances between the points (assuming the scoring vectors allow such a distance to be justified). We then apply an iterative process for performing the clustering. At every iteration, we find for the two closest clusters C_i and C_j, based on the dissimilarity function given by equation (2.2). Let C_k be the result of merging C_i and C_j. We compute the distances from every other cluster to C_k and update the distance matrix accordingly, by removing two rows and columns (i-th and j-th ones) and adding a new row and column corresponding to C_k. Storing the sequence of merges allows us to find the corresponding partition associated with a particular iteration. The process stops when the matrix D is reduced to a single element, which corresponds to the case when all inputs are clustered together.

Finally, we assess the quality of every partition returned by the iterative process using a measure called the Calinsky-Harabasz index. According to a comprehensive analysis on both real and synthetic data performed by Milligan and Cooper [38], this index performs best in assessing how well a given partition captures the true structure of the data. The bigger the value for the index, the better the partition. The Calinsky-Harabasz index for a partition P consisting of k clusters is computed as follows:

$$CH(P) = \frac{(m-k)BCSS(P)}{(k-1)WCSS(P)} \tag{3.2}$$

In equation (3.2), the term $WCSS(P)$ is the sum of squared distances between every point and its corresponding cluster centroid formally defined as:

$$WCSS(P) = \sum_{C_i \in P} \sum_{x_j \in C_i} \|x_j - \mu_i\|^2 \tag{3.3}$$

The term $BCSS(P)$ is the sum of squared distances between every point and all other centroids.

$$BCSS(P) = \sum_{C_i \in P} \sum_{x_j \notin C_i} \|x_j - \mu_i\|^2 \tag{3.4}$$

Intuitively, the closer the points are inside the clusters and the farther away the clusters are from one another, the bigger the score. The term $\frac{m-k}{k-1}$ accounts for the number of parameters in the model. Partitions with small number of clusters are preferred, while those with large number of clusters are penalized.

3.1.1 Outlier detection and removal

The input dataset needs to be curated of any potential outliers before attempting to cluster it. Certain nodes which exhibit a level of activity considerably higher than the rest can skew the clustering results significantly. Due to the large distance between them and every other input point, the resulting partition will contain most of the data clustered together while the outliers will share their own cluster.

The following preprocessing steps are performed to prevent this issue. Agglomerative clustering is performed over the original dataset. If some of the resulting clusters contain less than a small fraction ϵ of the total number of input points, we take them out of the dataset and repeat the process. The outliers removed in this iterative process are put all

together into one cluster and added to the final partition.

3.1.2 Proposed Algorithm

After performing outlier detection and removal, we run agglomerative clustering on the curated dataset and then choose the partition with the highest Calinsky-Harabasz score. Algorithm 3 presents our approach for performing agglomerative clustering on a dataset $\mathbf{X}$. The distance function between two clusters, denoted d, is given by equation (2.2).

Algorithm 3 Agglomerative Clustering ($\mathbf{X}$)

Input:

$\mathbf{X}$ - input data consisting of m vectors of size n

Output:

P_{fin} - the partition with the highest Calinsky-Harabasz score

$P = \{C_1, C_2 \dots C_m\}$, where $C_i = \{x_i\}$
$P_{fin} = P$
$D = []$ // list for storing the distances between merged clusters
$AP = []$ // list for storing all the partitions
repeat
    $(a, b) = \arg\min_{(i \neq j)} d(C_i, C_j)$
    append $d(C_a, C_b)$ to D
    $C_{ab} = C_a \cup C_b$
    $P = (P \setminus \{C_a, C_b\}) \cup \{C_{ab}\}$
    if $CH(P) \leq CH(P_{fin})$ **then** // $CH(P)$ is computed using eq. (3.2)
        $P_{fin} = P$
    end if
until $|P| = 1$
return P_{fin}

Next we want to assess the quality of every individual cluster and quantify our confidence that the points in a particular cluster really belong there. We start by defining a measure of dissimilarity between a point x_i to a cluster C_j as the average distance between

x_i and the points $x_j \in C_j$.

$$dissim(x_i, C_j) = \frac{1}{|C_j|} \sum_{x_j \in C_j} \|x_i - x_j\| \tag{3.5}$$

Let C_{x_i} be the cluster to which x_i belongs. We denote $dissim(x_i, C_{x_i})$ by $a(x_i)$. Let $b(x_i)$ be the lowest dissimilarity of a point x_i to one of the rest of the clusters it does not belong to:

$$b(x_i) = \min_{C_j, C_j \neq C_{x_i}} sim(x_i, C_j) \tag{3.6}$$

Intuitively, $b(x_i)$ is the average distance to the cluster in which aside from C_{x_i}, x_i fits best. The following equation called the silhouette of a point [44] allows us to quantify the confidence in the fact that the point x_i belongs to the cluster C_{x_i}:

$$s(x_i) = \frac{b(x_i) - a(x_i)}{max\{(a(x_i), b(x_i))\}} \tag{3.7}$$

It can be easily seen that $-1 \leq s(x_i) \leq 1$. As $s(x_i)$ approaches 1, the confidence that the point is appropriately clustered increases. Averaging this measure over all the points in a cluster gives the cluster silhouette:

$$s(C_i) = \frac{1}{|C_i|} \sum_{x_i \in C_i} s(x_i) \tag{3.8}$$

A high value for the cluster silhouette indicates a high cohesion cluster.

The last step is to define the importance of a point in a cluster as the sum of the fraction of scores for every activity:

$$J(x_i) = \sum_{j=1}^{n} \frac{x_{i,j}}{\sum_{x_k \in C_{x_i}} x_{k,j}} \tag{3.9}$$

Equation (3.9) gives the relative importance of node u_i compared to the rest of the nodes in the cluster, based on its level of activity. Intuitively, if u_i interacts more with v than with the rest of the nodes in the cluster, its importance grows.

Using all the measure from above, we now assign the following score to every point $x_i \in \mathbf{X}$:

$$\omega(x_i) = \frac{J(x_i)(s(x_i) + 1 + \epsilon)(s(C_{x_i}) + 1 + \epsilon)m}{|C_{x_i}|} \tag{3.10}$$

The score given by equation (3.10) accounts for the quality of the cluster in which the point is (C_{x_i}), the importance of the point in the cluster ($J(x_i)$), our confidence in the cluster assignment ($s(x_i)$) and penalizes big clusters by multiplying with $\frac{m}{|C_{x_i}|}$. Since we aim for the scores $w(x_i)$ to be positive and rank the corresponding nodes u_i in terms of their influenceability, we add $1 + \epsilon$ to $s(C_{x_i})$ and $s(x_i)$ to make the quantities strictly positive. For outliers and their associated cluster, we set $s(C_{x_i})$ and $s(x_i)$ to -1 in (3.10), to denote the fact that we consider the quality of the cluster to be bad.

Let x_i be the data point in $\mathbf{X}$ associated with neighbor u_i. The function $f_v : N(v) \rightarrow [0, 1]$ is defined as follows:

$$f_v(u_i) = \frac{\omega(x_i)}{\sum_{u_j \in N(v)} \omega(x_j)} \tag{3.11}$$

In equation (3.11) $f_v(u_i)$ is the probability of node v influencing node u_i. This probability is obtained by normalizing the score $\omega(x_i)$ (dividing by the total sum of scores of all nodes in $N(v)$). Putting everything together, our proposed approach for learning influence probabilities in the local neighborhood of a node v using agglomerative clustering is presented in Algorithm 4.

To learn the required weights $b_{u,v}$ for the Linear Threshold model, we run the proposed algorithm on all of v's neighbors. We then multiply each influence probability on

the incoming edges of v by:

$$J_{N(v)}(u) = \sum_{j=1}^{n} \frac{x_{u,j}}{\sum_{w \in N(v)} x_{w,j}} \tag{3.12}$$

This ensures we take into account the relative importance of v's neighbors to one another. Let θ_v be the unknown threshold value (chosen uniformly at random in $[0, 1]$) associated with node v, as assumed by the Linear Threshold model described in subsection 2.2.1. The weights required as input by the model can be computed with the following formula:

$$b_{u,v} = \frac{J_{N(v)}(u) f_u(v) \theta_v}{\sum_{w \in N(v)} \left(J_{N(v)}(w) f_w(v) \theta_v \right)} \tag{3.13}$$

We can easily reason about the worst case time complexity of Algorithm 4, by looking at the individual steps involved in the computation. Agglomerative clustering (see Algorithm 3) has a worst case time complexity of $O(m^2)$, where m is the size of the input. Performing outlier removal and clustering the dataset implies running Algorithm 3 for some small number of times denoted by c. Since computing S and J can be done in time linear to the size of the input, the worst case complexity of Algorithm 4 is $O(cm^2)$. As c is significantly smaller than m, this reduces to $O(m^2)$.

Algorithm 4 Learning Influence Probabilities: $(N(v), A_v, S_{A_v}, \epsilon)$

Input:

$N(v)$ - v's set of neighbors

A_v - log of interactions between v and its neighbors

S_{A_v} - scores associated with each action for v

ϵ - the minimum size of a cluster in order to not be considered as consisting of outliers

Output:

f_v - a probability mass function on the outgoing edges of v, quantifying the degree to which it influences its neighbors

compute $\mathbf{X}$ using eq. (3.1)
$C_{out} = \emptyset$
repeat
    $C_{out'} = \emptyset$
    $P = Agglomerative\ Clustering(\mathbf{X})$
    for all $C_i \in P$, $\frac{|C_i|}{m} \leq \epsilon$ **do**
        $C_{out'} = C_{out'} \cup C_i$
    end for
    $C_{out} = C_{out} \cup C_{out'}$
until $C_{out'} = \emptyset$
$\mathbf{X} = \mathbf{X} \setminus C_{out}$ // outlier removal step
$P' = Agglomerative\ Clustering(\mathbf{X})$
$P' = P' \cup C_{out}$ // adding back the cluster of outliers
$S = \{s(C_i), \forall C_i \in P'\}$ using eq. (3.8)
$J = \{J(x_i), \forall x_i \in \mathbf{X}\}$ using eq. (3.9)
$\omega = \{\omega(x_i), \forall x_i \in \mathbf{X}\}$ using eq. (3.10)
$f_v = \{f_v(u), \forall u \in N(v)\}$ using eq. (3.11)
return f_v

CHAPTER 4
Experimental framework and setup

For the purpose of obtaining Facebook data to test the performance of our proposed algorithm, we created a web application. The application is written in the Ruby programming language and follows the 3-tier architectural pattern. The first tier is composed of the database management systems in charge of data storage, as well as the appropriate data abstractions in the code. The middle tier contains the code responsible for crawling the data from Facebook, as well as the implementations of our proposed algorithm. The last tier offers a basic user interface, required for authorization. All layers are as decoupled as possible, but reside on the same physical machine, as this provides sufficient computational power for our needs.

This setup enables us to crawl, store and analyze all the data associated with a particular Facebook account, provided the owner grants us permissions in the form of an OAuth 2.0 access token. The use of the open standard OAuth 2.0 for the purpose of authorization and access to a user's private resources avoids the need to obtain and store login credentials (usernames and passwords). The user has the ability to revoke the access token and stop the application from accessing his data at any time. After a straightforward authorization process, the account data is extracted from Facebook using their Graph API and stored for future reference and access. The proposed algorithm can thus be run entirely offline.

Section 4.1 defines the vocabulary of the problem domain. Sections 4.2, 4.3 and 4.4 introduce and explain each of the three layers of the application.

4.1 Terminology

For eliminating confusion, we define below the terms commonly used throughout the rest of this chapter.

Application: the software system we designed and implemented for the purpose of extracting, storing and analyzing Facebook data. It follows the inversion of control design principle and it exposes an application public interface. The API can be further extended and built upon.

Connector: a Facebook account that is being tracked, associated with a particular user. Multiple connectors can be associated with one user, since he can own multiple accounts or Facebook Pages simultaneously. A connector consists of a set of snapshots.

Snapshot: contains the full set of crawled data associated with a Facebook account (e.g. posts, statuses, pictures, likes), as available via the Graph API at a particular moment in time. Each snapshot is associated with a timestamp.

Sub: a part of a snapshot pertaining to a specific type of data (e.g. all status updates). Subs can be classified as main subs (status updates, links etc.) and collection subs (or simply collections). A sub consists of a set of items.

Item: the most basic unit of data retrieved and stored from Facebook. It can be a comment, like, picture, status update etc. Items contain the raw data in JSON format

obtained during the crawling process. An item can be associated with zero or more collections.

Collection: a specific type of sub, containing items associated with another particular item called the collection parent. Some examples include but are not limited to: all likes associated with a comment, all comments associated with a picture, all photos in a photo album.

Crawl: the process of requesting data from Facebook's Graph API and storing the responses in a snapshot composed of items.

Contributor: a person who has interacted with a Facebook user at least once on the social platform. A contributor is part of the user's social graph, but does not necessarily need to be a Facebook friend.

Activity: denotes any unidirectional interaction between two users on Facebook, such as liking someone's post, "friending", or tagging someone in a picture.

Report: a JSON document stored in MongoDB containing the aggregated results of running some algorithm for analyzing the data. A report is associated with either a connector, or a specific snapshot.

Job: a Ruby class which contains a class method called *#perform*. A job can be serialized to MongoDB and run at a future point in time in a separate operating system process.

4.2 The database layer

This layer of the application stores three different types of information. First, there is user profile and login information including the OAuth access tokens. Second, we have

raw data in JSON format crawled from Facebook. The last type of data we need to store is filtered data containing the exact pieces of information needed for running the experiments.

Storing and accessing these different categories of data employ different use cases and enforce certain non-functional requirements for the system. We deploy three different database management systems (DBMSs) to serve our needs. One of them falls under the relational paradigm, while the other two have different data models and fall under the so called NoSQL paradigm. These three DBMSs and other critical details needed for replicating the experimental setup are the subject of the following subsections.

4.2.1 MySQL and memcached

For the purpose of storing user profile information and access tokens we deploy a standard relational open source database management system called MySQL. Choosing Ruby on Rails as a web framework enables us to add a higher level of abstraction (in the form of Ruby classes called ActiveRecord models) to our database tables. The ActiveRecord models provide built in create, read, update and delete capabilities and can also be extended as needed. This object-relational mapper provides ease of access, as well as the option of switching to a different relational DBMS at any point in future, without having to update any code.

To avoid too many disk operations when accessing data, a distributed open source memory object caching system called memcached is deployed between the MySQL database and the application layer. Data is fetched inside the main memory allocated to the memcached process upon first being accessed. When the memory gets full, data is evicted according to a least-recently used policy. Disk writes only happen in two scenarios: when

a row is evicted from the cache, and when the entire cache is being flushed to the disk (operation that happens periodically).

4.2.2 Raw data and the Riak cluster

The Graph API returns raw JSON data that does not follow a fixed schema (i.e. is semi-structured). Storing this data without performing any preprocessing is useful for two purposes. First, we want to mine the structure and the interactions of the local social graph associated with that particular account without having to constantly query Facebook. Second, we want to archive the data for backup purposes. Fault tolerance and high availability are also a concern. A failure in a write could result in an entire failed snapshot and the result of several hours of crawling being lost. For these reasons, we employ a NoSQL DBMS called Riak which uses a very simple key-value storage and access model for the data.

While Riak can generate unique keys for the stored values, we have put in place a couple of conventions for their format. This way we can easily retrieve data related to a snapshot or a specific sub. A snapshot's key follows the pattern:

connector_id:timestamp

The timestamp is of the form *yyyyddmmxx* (e.g. 2012300100). The last two digits (denoted by xx) enable the creation of multiple snapshots in the same day. They incrementally increase by 1, starting with the value 00 associated with the first snapshot taken in a specific day.

The key for a sub follows the pattern:

connector_id:timestamp:kind:[remote_id:collection]

The *kind* identifies what type of items are contained in the sub (e.g. “statuses”, “links”). The optional pair of parameters *remote_id:collection* identify the collection the item belongs to, and the collection parent. Each item key is generated by applying a cryptographic hash function on the minimal set of fields (dependent on the item type) which uniquely identify the item content. This saves a lot of processing time and storage space. A particular item – even though part of multiple snapshots – is stored only once.

A cluster of three Riak nodes has been deployed for handling the load, each of them with 1023 threads in its asynchronous thread pool. In a Riak cluster there are no special or master nodes. Every node is treated the same, facilitating horizontal scaling. No dedicated node for performing query routing is needed. Every instance knows where to redirect a read request, through consistent hashing. This architecture does not contain any single points of failure.

At the Principles of Distributed Computing conference in 2000 [3], professor Eric Brewer from University of California, Berkeley stated the conjecture that a distributed system can only guarantee two of the following three properties:

Partition tolerance: a distributed system should be fault-tolerant and respond correctly when faced with any set of failures, besides a total network crash.

Consistency: all connected clients should have the same view of the data at any given time. Multiple conflicting values for the same piece of data of different network nodes are prohibited.

Availability: every incoming request should receive a response.

The conjecture was formally proven by Gilbert and Lynch [19] and is now known as the CAP theorem. The theorem implies a taxonomy for distributed DBMSs, depending on the tradeoffs made by their architects. Based on this classification, Riak sacrifices consistency for increased availability, while providing partition tolerance. Moreover, the number of Riak nodes who need to read or write a value successfully in the cluster before a success is returned to the client can be configured on a per bucket (a bucket is the logical equivalent of a table in a RDBMS) or even per query basis. This kind of flexibility makes Riak a perfect solution for storing semi-structured raw data crawled from Facebook.

4.2.3 Storing filtered data in MongoDB

In the kind of analysis we are performing, computational speed is an important factor. The raw data obtained from Facebook is not on the appropriate abstraction level for running the experiments. Recreating the data structures we need each time we run the algorithm would be wasteful in terms of time and resources. In effect, we preprocess the raw data by mining it for contributors and activities, effectively modeling the underlying social network. The preprocessed dataset is stored in another NoSQL DBMS called MongoDB.

MongoDB is a open-source document-oriented database that stores JSON documents in a binary-encoded format called BSON [2]. This DBMS provides high throughput and complex query capabilities (e.g. logical operators, queries based on regular expressions) as well as powerful indexing features.

Consistency is one of the main focuses of MongoDB and is achieved by employing the master/slave paradigm for distributing load. To ensure each client has the same view of the data, all writes are performed on a particular running instance called “the master”. The

master records the write in an append-only replication log that is subsequently replayed by the rest of the nodes in the cluster called “slaves.”.

We deploy three MongoDB nodes (one master, two slaves) for both read-load distribution, as well as replication purposes. The schema-less storage paradigm, the ability to only retrieve certain fields from documents in a result set, and the high performance in terms of throughput justify the use of MongoDB as a solution for storing datasets needed for analysis.

4.3 The application layer

The code in this layer concerned with authorization and the filtered data in MongoDB is packaged as a standalone gem. A Ruby gem is the standard format for distributing Ruby programs and libraries. Most of the code in the cloud backend is aimed at providing an abstract framework for authorizing the application to any service supporting the OAuth 2.0 protocol. Access tokens are obtained from the service API as dictated by the protocol.

A separate gem containing specializations of some key classes in the cloud backend enables us to obtain the required permissions from Facebook users, permissions needed to extract the required data for the experiments. Two extensible APIs are exposed by the cloud backend gem, for handling filtered data.

4.3.1 The Models API

This API offers six simple methods for handling the filtered MongoDB data associated with a particular connector. Three of them allow for mining data for contributors and

activities, while the other two allow for the destruction and re-creation of the entire filtered dataset. The method signatures and short descriptions about their behavior follow:

Analytics#parse_item_for_data(item, options) - uses an item along with information identifying the snapshot to which it belongs and uses a JSON parser to scan the item content for Facebook user ids. The appropriate MongoDB models are then created or updated accordingly.

Analytics#parse_sub_for_data(sub, options) - mines an entire sub for contributors and activities, by calling the previous method on every composing item. Before parsing the item for the data, the associated collection subs for that item (if any) are stored in a list. The method recursively calls itself on each of those subs.

Analytics#parse_snapshot_for_data(snapshot, options) - parses a entire snapshot for contributors and activities in two steps. First, the main subs for that particular snapshot are fetched from Riak. For each such sub, a new operating system process is forked and the sub is parsed in it.

Analytics#create_models_for_connector(connector) - processes all the data available for a given connector across all the snapshots taken and creates the filtered MongoDB models. This method can be safely called multiple times, as no duplicate records will be stored.

Analytics#drop_models_for_connector(connector) - the counterpart for the previous method. Deletes anything in MongoDB related to the connector passed as a parameter.

Analytics#models_status_for_connector(connector) - writes a simple report on the standard output about the status of the connector and its associated models.

4.3.2 The Reports API

The second API exposed by the cloud backend controls the creation of reports containing aggregated results of running experiments on the preprocessed dataset stored in MongoDB. Three methods are made available, with the following signatures and behavior:

Analytics#create_report_for_connector(connector, options) - creates a particular type of report for a connector, based on the information passed in the *options* hash. The report is computed, associated with a particular snapshot (as indicated by the *:report_name* and *:timestamp* options) and stored in MongoDB. A human readable version of the report is cached in Riak and sent to the standard output.

Analytics#drop_report_for_connector(report) - destroys the report passed as a parameter, as well as any cached data associated with it.

Analytics#drop_all_reports_for_connector(connector) - destroys all the reports belonging to a given connector, as well as any associated data structures.

4.3.3 Performing and monitoring jobs with resque-mongo

Manually dispatching and keeping track of jobs is a tedious task, especially across multiple snapshots. Moreover, the Facebook Graph API has limitations regarding the number of queries allowed per ten minutes. This forces us to schedule jobs at future points in time. Sometimes, for various reasons such as bad network connectivity or malformed responses from Facebook, some jobs fail. The appropriate corrections have to be made and the jobs need to be retried. For these reasons we have integrated in the application a

framework for creating background jobs and placing them on multiple execution queues called resque-mongo.

The framework is composed of three separate parts. The first part allows for the creation of background jobs that will be executed in separate operating system processes. Any Ruby class or module that responds to the class method *#perform* is a valid resque-mongo job. When enqueued, jobs are serialized to MongoDB as simple documents containing the job class and the arguments that will be sent to the *#perform* method upon execution.

The second part of the framework is a small application for launching multiple worker operating system processes. The workers will constantly poll MongoDB, reserve jobs and execute them by instantiating the appropriate class and calling *#perform* with the serialized arguments. Workers can be configured to only poll certain queues or prioritize certain groups of jobs.

The last part consists of a web interface for monitoring queues, jobs, and workers, as well as retrying them in case of failure. The user interface also contains a search feature and provides the ability to drop entire queues. An utility for monitoring workers in realtime is also present.

To be able to queue jobs at a specific future time or based on a schedule and keep track of groups of jobs, we have extended the framework through two plugins, called resque-mongo-scheduler and resque-mongo-groups. The latter was implemented by us and is available as open-source software on Github.

4.3.4 Crawling Facebook accounts

The process of extracting data associated with a Facebook account consists of performing and keeping track of the execution of multiple jobs simultaneously. These jobs request data from the Graph API and create the appropriate items in Riak for storing the results. There are three types of jobs involved in successfully crawling a new snapshot. An initial job called the "dispatcher" verifies the validity of the access token, creates a new empty snapshot and dispatches a job for each main sub. Each of these jobs, called "crawlers" make the actual API calls, requesting data from Facebook. Any crawler can dispatch other crawlers, for getting data such as comments or likes associated with a particular item. Each crawler operates on multiple threads and requests data in batches, writing 1500 items at a time to Riak. The last type of job called the "post_process" is enqueued and performed periodically. Its responsibility is to check the status of the snapshots, mark the ones that finished as complete, and delete any temporary or intermediate data stored.

During the crawling process we cannot know ahead of time how many jobs will be dispatched. Instead, what we can do is track how many have been dispatched so far, how many are scheduled to perform at a future point in time and how many have been completed. When the number of total dispatched jobs equals the number of completed jobs and there are no more delayed ones, a post_process job will mark the snapshot as complete.

4.4 The presentation layer

The last layer in the architecture is in charge of displaying a basic UI to the users, allowing them to authorize connectors. It also includes the resque-mongo monitoring interface, for keeping track of jobs, queues, and workers.

This frontend layer is built on top of the Ruby on Rails web framework. Rails follows the Model-View-Controller architectural pattern, separating presentational concerns from the actual models and the rest of the application logic. The controllers themselves do nothing more than just call certain methods from the cloud backend gem for retrieving data needed to build the views. The views themselves are written in eRuby, a templating system that allows the insertion of Ruby code in HTML documents.

To decouple the backend gem from the Rails application we have implemented an internal REST-ful [17] API that provides information related to the crawling process via HTTP calls. The Rails application listens to incoming requests on a specific port and builds and returns information stored in the JSON format. The response contains information such as the list of connectors for which there are snapshots in progress, the list of connectors to trigger snapshots for and so on.

The web application is efficient in terms of both storing the data, and time taken to create a snapshot of an account. Whenever an API call is made to Facebook, the response is matched against what is already stored in the Riak cluster, by computing a hash function over the content received. In effect, after performing the first snapshot, the subsequent ones store just a delta of the data. Applying deltas over the data stored during the initial snapshot allows us to access the state of the account at any specific point in time. Since the crawling process is parallelized using background and delayed jobs handled by resque-mongo, the major factor in determining the speed of taking a snapshot is the rate limit policy of Facebook's Graph API.

For a better understanding, figure 4–1 shows a deployment diagram of the entire system. Note that the only kind of jobs we show are the crawlers, since they are the ones

interfacing with the Facebook API. Other jobs such as the post process task or the code implementing algorithm 4 are spawned by workers processes, run, save their results back to Riak or MongoDB and then quit.

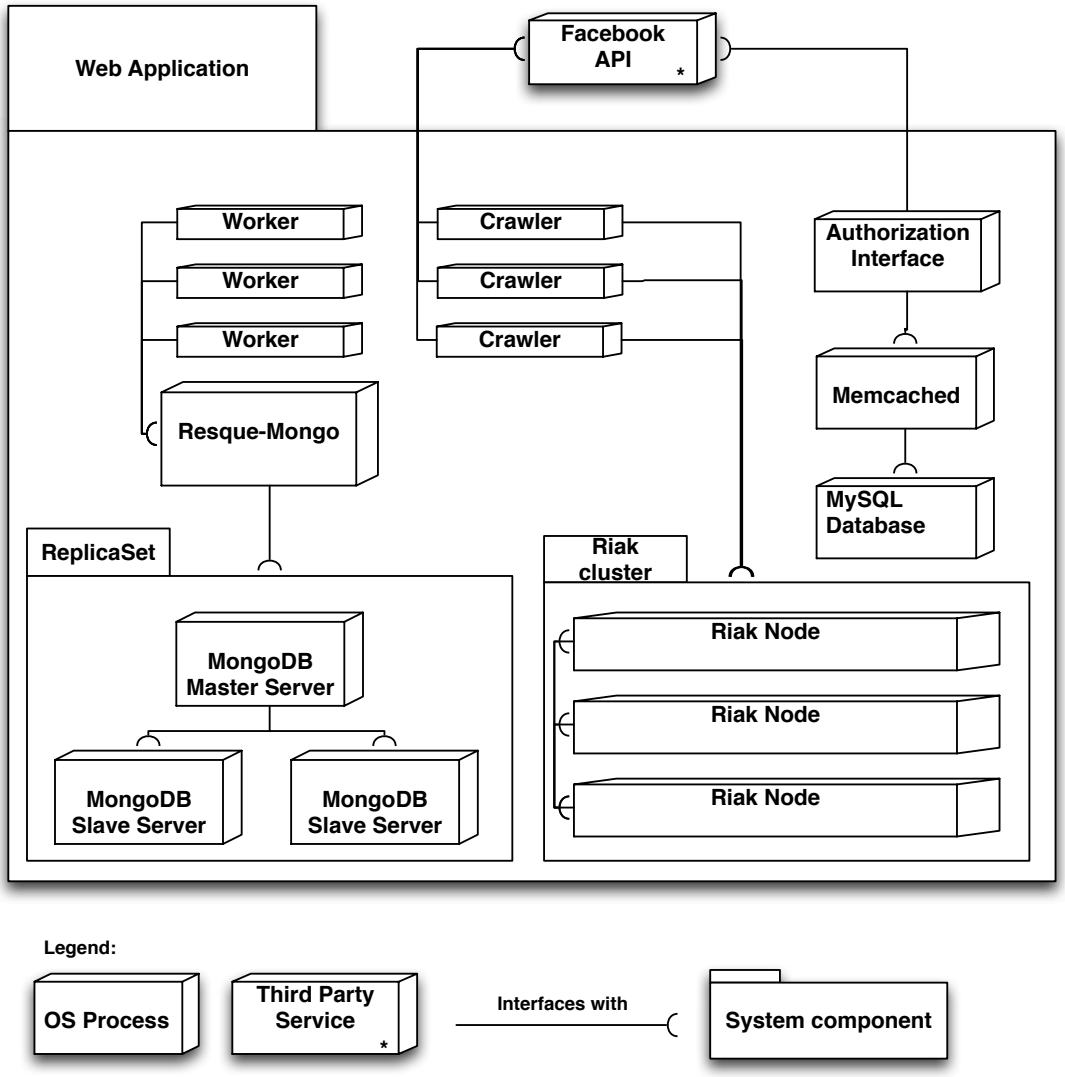


Figure 4–1: Deployment diagram of the web application

CHAPTER 5
Experimental results

In this chapter we present and analyze results obtained from performing three experiments on real data extracted from Facebook. The first two ones aimed at assessing whether the approach detailed in chapter 3 can correctly identify influenceable nodes in the neighborhood of some node or not. The last experiment assesses the quality of the influence probabilities learned by looking at the rankings of the nodes returned by the algorithm.

Section 5.1 introduces the datasets and the methodology used for performing the experiments and assessing the results. In section 5.2 we present the details needed to replicate them, and give some preliminary statistics regarding the output of Algorithm 4 on the datasets. Section 5.3 presents and discusses the results of the experiments.

5.1 Methodology

For the purpose of running the experiments, we enrolled the help of 16 volunteers. Each of them authorized our web application to access their private Facebook data and crawl their accounts. Furthermore, the volunteers helped assess the results, since the experiments presented in section 5.3 involve obtaining ground truth data from them. The data is obtained by asking the volunteers to answer some survey questions, and the output of the algorithm is compared against their responses.

Let $v_1, v_2 \ldots v_{16}$ be the nodes in the social network corresponding to the volunteers. Where there is no confusion, we use v_i interchangeably to refer either to the volunteer, or the node associated with him/her. For each volunteer v_i we crawl the Facebook data and mine the set of contributors denoted by $N(v_i)$, as well as the timestamped history of interactions between v_i and every $u \in N(v_i)$, denoted by A_{v_i}. We detected and mined 20 distinct types of activities that determine the type of social ties between a contributor and some volunteer v_i. The list of types of activities and their explanations can be found in table 5–1.

Activity number	Activity name	Description
1	employer	u is the employer of v
2	significant_other	u is v's significant other
3	family_member	u is part of v's family
4	wall_post_author	u posted on v's wall
5	tagged_in_video	u is tagged in one of v's videos
6	tagged_in_photo	u is tagged in one of v's photos
7	checkin_author	u checking in the same place as v
8	post_comment	u commented on one of v's posts
9	status_comment	u commented on one of v's status updates
10	link_comment	u commented on one of v's posted links
11	checkin_comment	u commented on one of v's checkins
12	video_comment	u commented on one of v's videos
13	photo_comment	u commented on one of v's photos
14	post_like	u liked one of v's posts
15	status_like	u liked one of v's status updates
16	link_like	u liked one of v's posted links
17	checkin_like	u liked one of v's checkins
18	video_like	u liked one of v's videos
19	photo_like	u liked one of v's photo
20	comment_like	u liked one of v's comments

Table 5–1: Types of activities detected

However, the aggregated number of comments and likes (rows 8-20) account for more than 98.118% of the total number of activities. Hence, in the analysis we only take into account these two types of activity since the others would not provide significant data. Table 5–2 contains the total number of contributors and a detailed breakdown of the activities mined from the account data of each of the volunteers.

Volunteer	Contributors	Comments	Likes	Other	Total
v_1	583	508	771	56	1335
v_2	993	802	951	79	1832
v_3	514	528	436	47	1011
v_4	459	65	475	13	553
v_5	2164	2919	5584	390	8263
v_6	261	1679	752	59	2490
v_7	390	78	190	7	275
v_8	384	554	1085	63	1702
v_9	452	546	259	25	830
v_{10}	273	204	510	27	741
v_{11}	515	1162	960	102	2224
v_{12}	361	295	311	28	634
v_{13}	613	1173	1441	106	2720
v_{14}	610	69	116	8	193
v_{15}	720	446	828	39	1313
v_{16}	344	1672	2039	145	3856

Table 5–2: Mined activities and contributors

To be able to run Algorithm 4 on the datasets we crawl, we need to define a scoring vector $\mathbf{S_{A_v}} = (s_{A_v^1}, s_{A_v^2} \dots s_{A_v^n})^\mathsf{T}$ that tells us how important each type of actions is, compared to the others. We follow the simple intuition that a comment involves much more commitment from a contributor than a like, claim that is supported by click-through data [1]. We choose the scoring vector $\mathbf{S_{A_v}} = (1, 4)^\mathsf{T}$ for every node v_i, which means a comment is considered 4 times more valuable than a like.

We then run Algorithm 4 on each of the 16 datasets $\mathbf{X^i}, i = 1 \rightarrow 16$ and rank the contributors for each volunteer in $N(v_i)$ in decreasing order of their associated weights, as computed by the algorithm. For every v_i we interpret the output as being a permutation from the set $N(v_i)$ to the set $[|N(v_i)|] = \{1, 2 \ldots |N(v_i)|\}$, denoted $\sigma(v_i)$.

5.2 Determining influenceable nodes using Agglomerative Clustering

We ran Algorithm 4 on the 16 datasets crawled, using values between 0.5% and 1% for the parameter ϵ. This parameter dictates the minimum fraction of input points a cluster must contain to not be considered as consisting of outliers. Multiple linkages were tested against each dataset to determine the best clustering possible: single, complete, average, weighted average and Ward's linkage. Because the single linkage criterion is local (the similarity of two clusters is the similarity of their most similar members and in effect we are only paying attention to the area where the two clusters come closest together), we experienced the phenomenon of chaining. Clusters were merged because of single elements being close to each other, even though most elements in the merged clusters were relatively far away from one another. The chaining effect encountered when using single linkage is illustrated in figure 5–1, which shows the resulting dendogram for the dataset associated with volunteer v_{16}.

Centroid and median linkages sometimes produced non-monotonic cluster trees. Merging two clusters can move the centroid in such a way that the next cluster to be merged in is actually closer to the centroid than the previous one. For this reason we avoided them. Good results were obtained using complete and average linkages, but the most consistent results were obtained using Ward's linkage and weighted average linkage. The formula for

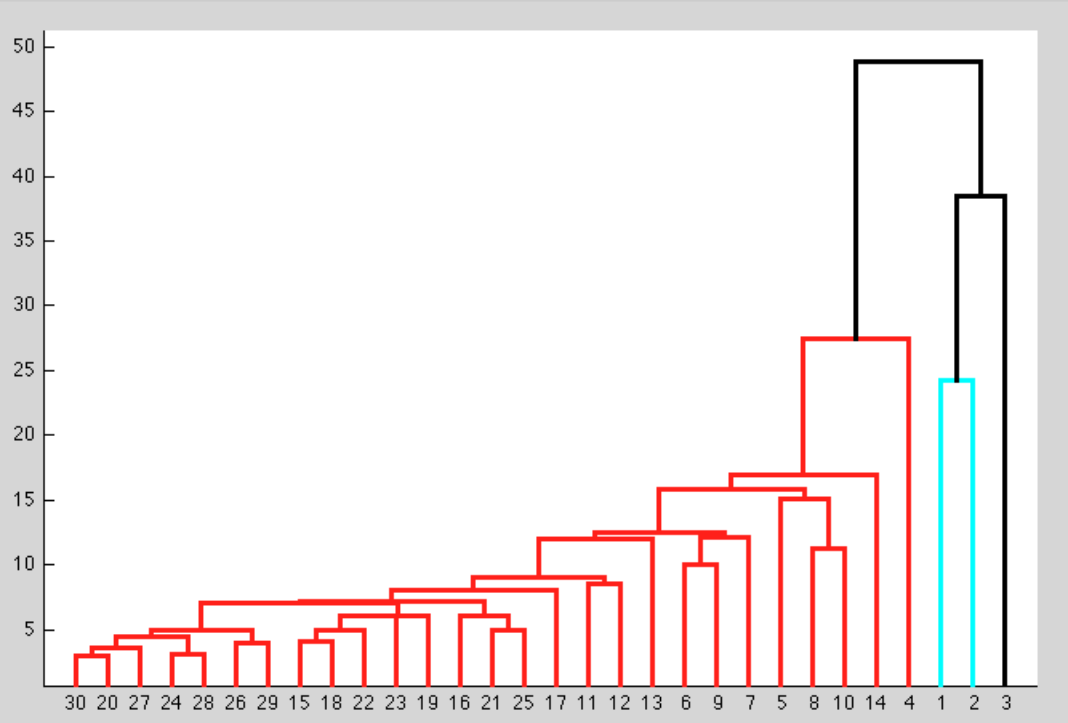


Figure 5–1: Chaining effect when using single linkage

the distance between two clusters according to Ward's linkage is shown in equation (5.1).

$$d(C_i, C_j) = \sqrt{\frac{|C_i||C_j|}{|C_i| + |C_j|}} \|\mu_i - \mu_j\| \tag{5.1}$$

The formula for weighted average linkage can be obtained by setting $\alpha_i = \alpha_j = 1/2$ and $\beta = \gamma = 0$ in equation (2.2). If cluster C_k was created by merging clusters C_i and C_j, the distance from it to some other cluster C_t is just the mean of the distances between C_i and C_j respectively to C_t.

After running Algorithm 4 on the datasets, we took into account only the neighbors who are being influenced with a probability of more than 0.001. These restricted datasets are denoted by $\mathbf{X}^i_{res}, i = 1 \to 16$. In table 5–3 we present basic statistics about the output

of the algorithm. The first two columns contain the original dataset size and the percentage of contributors which are influenced with probability of more than 0.001. Note that the dataset sizes are smaller than the corresponding number of crawled contributors, since a lot of them just added the volunteer as a friend and then did not interact with them anymore. The rest of the columns contain the maximum values for the influence probabilities as well as their the mean, mode, median and standard deviation.

v_i	$\lvert\mathbf{X}^i\rvert$	$\frac{\lvert\mathbf{X}^i_{res}\rvert}{\lvert\mathbf{X}^i\rvert}$	Max	Mean	Mode	Median	Stdev
v_1	271	0.140	0.081	0.026	0.002	0.017	0.027
v_2	284	0.151	0.129	0.022	0.002	0.006	0.032
v_3	139	0.237	0.127	0.030	0.001	0.018	0.035
v_4	195	0.328	0.104	0.015	0.002	0.003	0.027
v_5	1190	0.048	0.100	0.017	0.004	0.005	0.027
v_6	116	0.284	0.136	0.029	0.001	0.016	0.038
v_7	96	0.344	0.565	0.030	0.005	0.005	0.103
v_8	207	0.266	0.111	0.018	0.001	0.005	0.028
v_9	139	0.252	0.146	0.028	0.001	0.005	0.044
v_{10}	118	0.314	0.160	0.027	0.001	0.007	0.037
v_{11}	228	0.145	0.141	0.030	0.141	0.012	0.044
v_{12}	107	0.178	0.268	0.514	0.002	0.004	0.083
v_{13}	204	0.304	0.211	0.016	0.001	0.003	0.033
v_{14}	89	0.483	0.240	0.023	0.002	0.002	0.049
v_{15}	253	0.146	0.207	0.027	0.006	0.007	0.047
v_{16}	224	0.134	0.162	0.032	0.003	0.011	0.047

Table 5–3: Basic statistics for Algorithm 4

Table 5–3 reveals a number of interesting facts about the data. First, the mean fraction of neighbors which are influenced with probability of more than 0.001 is 0.235. Thus, each node v_i engages and influences on average almost a quarter of their neighbors, keeping the online interactions with the rest of them casual. The standard deviation from this mean however is quite big, 0.109, meaning that it is not unusual to see nodes which manage to

influence almost half of their neighbors. Even in those close-knit communities ($\mathbf{X}^i_{res}, i = 1 \rightarrow 16$), the influence the nodes v_i exhibit varies from one neighbor to the other (mean standard deviation of 0.044, which is big considering the average mode of 0.011).

Appendix A contains plots offering a visual interpretation of the results obtained from running Algorithm 4 for each dataset $\mathbf{X}^i, i = 1 \rightarrow 16$. In each case, first plot shows the variation of the Calinsky-Harabasz index over the topmost 25 partitions in the sequence generated by the algorithm, the second one is a dendogram plot, and the last two are scatter and silhouette plots for the final clustering of each dataset. Note that in each scatter plot, the brown points correspond to outliers. We observe two emerging patterns across the clustering for datasets. Part of the volunteers such as v_8 and v_{13} have most of the data in clusters close to the origin which tend to be vertically shaped. Some of the datasets such as the ones for v_{10} and v_{11} present spherically shaped clusters at various distances from the origin. The diversity we see in the latter clustering pattern suggest the existence of small communities of neighbors which are quite different in the behavior they exhibit from one another, due to the way the volunteer engages them.

5.3 Experiments and analysis of results

First, we want to see if the proposed method correctly identifies the influenceable nodes in the neighborhood of v_i. The first step in answering this question is estimating the true fraction of influenceable nodes in $N(v_i)$, which we denote by $\hat{f}_{v_i}$. For every node v_i, we sample uniformly, at random, a list of n of their neighbors. Let γ_{v_i} denote that list. Since we are trying to find an estimate for $\hat{f}_{v_i}$ through means of survey, we choose a moderately small value for the sample size n, of 50. The following survey task was posed to each of the volunteers:

"You are given a random list of 50 of your friends. For each of them, answer by "yes" or "no" the following question: "Is this friend highly likely to act (e.g. like, comment, re-share) on a piece of information I post on Facebook?""

Each survey gives us a sample statistic we can use to approximate $\hat{f}_{v_i}$, which is the fraction of nodes in γ_{v_i} for which the question was answered with *"yes"*. Let that fraction be $\bar{f}_{v_i}$. As a consequence of the fact that the sample size n is greater than 40, the Central Limit theorem states that the distribution of the sample statistic $\bar{f}_{v_i}$ is normal or nearly normal. We can now choose a confidence level and build a confidence interval for the population statistic $\hat{f}_{v_i}$, centered around $\bar{f}_{v_i}$:

$$\hat{f}_{v_i} = \bar{f}_(v_i) \pm z_{\alpha/2} * \sqrt{\frac{\bar{f}_{v_i}(1-\bar{f}_{v_i})}{n}} * \sqrt{\frac{m-n}{m-1}} \tag{5.2}$$

For a level of confidence of $1-\alpha$, equation (5.2) computes a margin of error around the estimate of the true fraction of influenceable nodes. This margin of error is computed as a critical value $z_{\alpha/2}$, multiplied by the standard error of the sample statistic, which is an unbiased estimate of the standard deviation of the population statistic $\hat{f}_{v_i}$. The critical value $z_{\alpha/2}$ represents the value in a standard normal distribution (mean 0 and variance 1) beyond which the area in each tail of the distribution is $\alpha/2$. For this analysis, we choose a confidence level of 95% ($\alpha = 5\%$), which gives us the critical value of $z_{\alpha/2} = 1.96$. The results of the survey, as well as confidence intervals for the true fractions of influenceable nodes in $N(v_i)$ are presented in table 5–4.

The last column in table 5–4 is the fraction of influenceable nodes that Algorithm 4 produced (i.e. percentage of nodes influenced with probability of more than 0.001). Note that aside from nodes v_6, v_{10} and v_{14}, the fractions produced by the algorithm fall in the

v_i	$\bar{f}_{v_i}$	$\hat{f}_{v_i}, \alpha = 5\%$	$\frac{\lvert X^i_{res}\rvert}{\lvert X^i\rvert}$
v_1	0.10	0.10 ± 0.075	0.140
v_2	0.14	0.14 ± 0.087	0.151
v_3	0.24	0.24 ± 0.095	0.237
v_4	0.28	0.28 ± 0.108	0.328
v_5	0.08	0.08 ± 0.074	0.048
v_6	0.16	0.16 ± 0.092	0.284
v_7	0.26	0.26 ± 0.085	0.344
v_8	0.30	0.30 ± 0.111	0.266
v_9	0.22	0.22 ± 0.092	0.252
v_{10}	0.20	0.20 ± 0.085	0.314
v_{11}	0.10	0.10 ± 0.074	0.145
v_{12}	0.14	0.14 ± 0.071	0.178
v_{13}	0.24	0.24 ± 0.103	0.304
v_{14}	0.32	0.32 ± 0.086	0.483
v_{15}	0.12	0.12 ± 0.081	0.146
v_{16}	0.12	0.12 ± 0.080	0.134

Table 5–4: Confidence intervals for the true fraction of influenceable neighbors

confidence intervals learned by sampling. In the three cases above, the fractions produced by the algorithm are bigger than the upper limits of the corresponding confidence intervals, but within two margins of error from $\bar{f}_{v_i}$. This means that either Algorithm 4 tends to incorrectly mark nodes as being influenceable, or the volunteers were conservative in their assessment of certain nodes as being influenceable.

In the next step of the analysis, we will look at a list of k neighbors influenced with probability of more than 0.001 according to the algorithm. Specifically, we are interested to know whether such a list contains more influenceable nodes than a list of the same size, sampled uniformly at random from $N(v_i)$. This experiment is aimed at determining whether algorithm 4 can successfully identify the truly influenceable nodes. In a random

sample of size k, we would expect to find the same fraction of truly influenceable nodes $\hat{f}_{v_i}$ as in the entire population. We next investigate whether in a random sample of size k chosen from the set of nodes marked by the algorithm as being influenced with probability of more than 0.001, the number of truly influenceable nodes is statistically significant higher than $k\hat{f}_{v_i}$. Let $\tau^k(v_i)$ be such a list of size k, where $\forall u_j \in \tau^k(v_i), x_j \in \mathbf{X}^i_{res}$. We formulate the following null hypothesis:

"For any value of k less than $|\mathbf{X}^i_{res}|$, the number of influenceable neighbors in $\tau^k(v_i)$ is not significantly bigger than the number of influenceable neighbors in a list of k of them, picked at random."

To test this hypothesis we perform an experiment consisting of the following steps. First, we sample $\tau^k(v_i)$ uniformly at random from the neighbors influenced with probability of more than 0.001. Note that since the experiment consists of a survey, we choose a reasonably small value for k, of 10. The following survey task is posed to the volunteers:

"You are given a list of 10 of your friends, in random order. For each of them, answer by "yes" or "no" the following question: "Is this friend highly likely to act (e.g. like, comment, re-share) on a piece of information I post on Facebook?""

Let r be the number of nodes marked as being influenceable by the volunteers in $\tau^{10}(v_i)$. Since we are sampling without replacement, r is the realization of a random variable R which is hyper-geometrically distributed, with parameters $|\mathbf{X}^i_{res}|$, $\hat{f}_{v_i}|\mathbf{X}^i_{res}|$ and k (population size, number of successes and sample size respectively). Now that we know the population size and the true number of influenceable nodes in the population, we can determine whether the number of successes r present in our sample of size k could have occurred by chance or not. Table 5–5 presents the results of this experiment.

v_i	$\lvert\mathbf{X}^i_{res}\rvert$	$\hat{f}_{v_i}\lvert\mathbf{X}^i_{res}\rvert$	r	$P(R \geq r)$
v_1	38	7	6	4.75×10^{-4}
v_2	43	10	5	3.60×10^{-2}
v_3	33	11	6	4.24×10^{-2}
v_4	64	25	7	3.48×10^{-2}
v_5	210	32	6	1.07×10^{-3}
v_6	33	8	7	2.02×10^{-4}
v_7	33	11	7	5.91×10^{-3}
v_8	55	23	9	9.33×10^{-4}
v_9	35	11	6	3.06×10^{-2}
v_{10}	37	11	8	1.58×10^{-4}
v_{11}	39	7	5	7.05×10^{-3}
v_{12}	19	4	6	0.00
v_{13}	62	21	8	1.66×10^{-3}
v_{14}	43	17	8	4.46×10^{-3}
v_{15}	99	20	6	4.15×10^{-3}
v_{16}	30	6	5	8.84×10^{-3}

Table 5–5: Results for the second survey task

The first column in the table is the number of nodes influenced with probability of more than 0.001 (i.e. the size of the population from which we are sampling). The second column contains the number of influenceable nodes we would expect to see in a dataset of size $|\mathbf{X}^i_{res}|$, according to the estimate of $\hat{f}_{v_i}$. For the purpose of this study, the upper margin of the corresponding confidence interval is taken into account and the number is rounded to the nearest integer. The third column represents the number of nodes the volunteers marked as being influenceable in the sample $\tau^{10}(v_i)$. The last column is the probability of seeing at least that many successes in the sample (i.e. the p value for determining statistical significance).

The values in the last column of table 5–5 are all below a statistical significance level of $p = 0.05$. This means we can safely reject the null hypothesis we are testing, and state that Algorithm 4 marks a significantly higher proportion of truly influenceable nodes as being influenced with a probability of more than 0.001, than an algorithm who marks them at random. Note that the p value for node v_{12} is 0.00, since the expected number of successes in the entire population is actually smaller than the number of successes in the sampl. We estimated there are 4 truly influenceable nodes in the dataset of size 19 and the volunteer marked 6 of them as such in the sample we presented to him.

We have so far shown two things about Algorithm 4. First, the fraction of nodes identified as influenceable is close to the true fraction of such nodes. Second, in lists sampled uniformly at random from $\mathbf{X}^i_{res}$ (the set of neighbors influenced with probability of more than 0.001) there are more truly influenceable nodes than we would expect to see by pure chance. The last thing of interest is whether a significant number of the truly influenceable nodes (as marked by the volunteers in the first survey) are properly identified by the algorithm (i.e. they are in $\mathbf{X}^i_{res}$).

Let $\hat{\gamma_{v_i}}$ denote the number of nodes in γ_{v_i} which were marked as influenceable by the volunteer in the survey process. We now formulate a second null hypothesis to test:

"There is no significant overlap between $\hat{\gamma_{v_i}}$ and $\mathbf{X}^i_{res}$."

To test this hypothesis, we need to make a critical observation. Let $\delta_{v_i} = |\hat{\gamma}_{v_i} \cap \mathbf{X}^i_{res}|$, be the size of the overlap between the nodes marked by the volunteers as influenceable, and the ones marked by the algorithm as influenceable. Note that both $\hat{\gamma}_{v_i}$ and $\mathbf{X}^i_{res}$ are subsets of $\mathbf{X}^i$. We now imagine a sampling process in which we sample without replacement $|\mathbf{X}^i_{res}|$ nodes from $\mathbf{X}^i$ and are interested in what fraction of them are in $\hat{\gamma}_{v_i}$. This

means the size of the overlap δ_{v_i} is the realization of a random variable H, which is hypergeometrically distributed with parameters $|\mathbf{X}^i|$, $\hat{\gamma}_{v_i}$ and $|\mathbf{X}^i_{res}|$ (population size, number of successes, sample size).

With this observation made, we can now determine whether the size of the overlap observed could have arisen by chance or not. Table 5–6 contains, for each volunteer, the following information: the population size ($|\mathbf{X}^i|$), the number of contributors marked as influenceable by Algorithm 4, the number of contributors marked as influenceable during the survey, the size of the overlap between them and $\mathbf{X}^i_{res}$ and the probability of observing that overlap.

v_i	$\lvert\mathbf{X}^i\rvert$	$\mathbf{X}^i_{res}$	$\lvert\hat{\gamma}_{v_i}\rvert$	$\lvert\delta_{v_i}\rvert$	$P(H \geq \lvert\delta_{v_i}\rvert)$
v_1	271	38	5	3	2.09×10^{-2}
v_2	284	43	7	4	1.14×10^{-2}
v_3	139	33	12	6	3.60×10^{-2}
v_4	195	64	14	8	4.63×10^{-2}
v_5	1190	210	4	0	5.40×10^{-1}
v_6	116	33	8	5	4.08×10^{-2}
v_7	96	33	13	8	3.08×10^{-2}
v_8	207	55	15	9	4.80×10^{-3}
v_9	139	35	11	6	2.95×10^{-2}
v_{10}	118	37	10	6	5.00×10^{-2}
v_{11}	228	39	5	3	3.63×10^{-2}
v_{12}	107	19	7	4	1.81×10^{-2}
v_{13}	204	62	12	8	8.40×10^{-3}
v_{14}	89	43	16	12	1.78×10^{-2}
v_{15}	253	99	6	5	3.53×10^{-2}
v_{16}	224	30	6	3	3.29×10^{-2}

Table 5–6: Results for the second survey task

Except the one corresponding to v_5, the values in the last column of table 5–6 are below a statistical significance level of $p = 0.05$. The fact that during the first survey we selected a sample size of 50 which is significantly lower than the population size of 1190 for v_5 might explain why out of the 4 nodes which he marked as being influenceable, we do not see even one in the list the algorithm produces. There is a high chance that the sample we sent to the volunteer did not even contain any truly influenceable nodes (Algorithm 4 detects only 57 such nodes). The volunteer might have marked the 4 nodes out of subjectivity. With this observation made, we reject the null-hypothesis postulated earlier and conclude that there is indeed significant overlap between the nodes that the algorithm marks as influenceable and the ones that the volunteers do.

The last part of the analysis focuses on assessing the quality of the rankings produced by the algorithm. We want to know if the relative order of the nodes the algorithm marks as highly influenceable is accurate or not. Let $\sigma^k(v_i)$ denote the ordered list of the first k nodes, as returned by Algorithm 4, and $\hat{\sigma}^k(v_i)$ the true ranking of those nodes, in terms of their influenceability. We formulate the following null hypothesis:

"For any value of k less than $|\mathbf{X}^i_{res}|$, there is no significant correlation between $\sigma^k(v_i)$ and $\hat{\sigma}^k(v_i)$."

To test this hypothesis, we give the following survey task to the volunteers, choosing a small value for k, of 10.

"You are given a list of 10 of your friends, in random order. Rank them in decreasing order of who is more prone to act (e.g. like, comment, re-share) on any piece of information you post on Facebook."

Note that both the rankings $\sigma^k(v_i)$ and the true rankings returned by the volunteers $\hat{\sigma}^k(v_i)$ are permutations of the same set of elements denoted D. Let $n = |D|$ be the number of elements in the set. To measure the correlation between the two permutations, we use a statistic called Kendall's Tau rank correlation coefficient [27], which has the following definition. Let D be a set of elements, $P_D = \{\{i, j\} | i \neq j$, and $i, j \in D\}$ the set of all unordered pairs of distinct elements from D, and $i, j \in P_D$ such a pair. The pair i, j is said to be concordant if the order of the two elements is the same in both permutations $\sigma^k(v_i)$ and $\hat{\sigma}^k(v_i)$. Conversely, the pair is said to be discordant if the permutations disagree on the order of the i and j. Let NC be the number of concordant pairs, ND the number of discordant pairs, and $NP = |P_D| = n(n-1)/2$ the total number of pairs. Kendall's Tau rank correlation coefficient is defined as:

$$KT(\sigma^k(v_i), \hat{\sigma}^k(v_i)) = \frac{NC - ND}{NP} \tag{5.3}$$

In equation (5.3), the number of concordant minus the number of discordant pairs is divided by the total number of pairs, which brings the coefficient in the range $[-1, 1]$. If the two rankings agree perfectly (are the same), the coefficient takes the value 1. If they disagree completely (one raking is the reverse of the other) the coefficient value is -1. When there is no correlation between the rankings, we expect the value of the coefficient to be 0.

Since there are no ties in the rankings, we can apply the Tau-a statistical test to determine if the correlation (if any) is statistically significant. If the rankings would be realizations of two independent random variables, the distribution of the following statistic z_A

would be nearly normal:

$$z_A = \frac{3(NC - ND)}{\sqrt{(n(n-1)(2n-5)/2)}} \tag{5.4}$$

All that is left is to compute the cumulative probability for the standard normal distribution at $-|z_A|$ and multiply it by two, since we are interested in both significant positive or negative correlation. This gives us a p value — the probability of seeing that correlation by chance — which we can use to accept or reject the null hypothesis at a specific significance level. Table 5–7 contains the correlation coefficient value, as well as the p value for each of the datasets.

v_i	$KT(\sigma^{10}(v_i), \hat{\sigma}^{10}(v_i))$	p value
v_1	0.511	4.90×10^{-2}
v_2	0.378	1.52×10^{-1}
v_3	0.467	7.36×10^{-2}
v_4	0.378	1.52×10^{-1}
v_5	0.556	3.18×10^{-2}
v_6	0.244	3.71×10^{-1}
v_7	0.511	4.90×10^{-2}
v_8	0.422	1.07×10^{-1}
v_9	0.600	2.00×10^{-2}
v_{10}	0.556	3.18×10^{-2}
v_{11}	0.689	7.29×10^{-3}
v_{12}	0.333	2.10×10^{-1}
v_{13}	0.289	2.83×10^{-1}
v_{14}	0.644	1.22×10^{-2}
v_{15}	0.422	1.07×10^{-1}
v_{16}	0.556	3.18×10^{-2}

Table 5–7: Correlation between top 10 rankings

There is definitely a positive correlation between the rankings we got from the volunteers and the ones Algorithm 4 computed (mean 0.472 and standard deviation of 0.128). However, at a significance level of $p = 0.05$ we can reject the null hypothesis only in 8 out of 16 cases. For the rest of them, the only statement we can make is that the rankings $\sigma^{10}(v_i)$ weakly agree with the true rankings $\hat{\sigma}^{10}(v_i)$. To conclude this chapter, we note that although we regard the rankings we got from the volunteers as ground truth, we currently have no way of measuring or accounting for subjectivity in their assessment. The fact that in half of the cases our rankings only weakly agree with them does not imply the fact that the quality of the results is bad. It is possible that Algorithm 4 provides insight into who is more influenceable, that cannot be observed by eye. Also, the heuristic used to score the nodes was not designed to give an exact ranking, but to quantify the magnitude of the influence the node v_i exerts on its neighbors. Although convenient, survey methods might not be the best way of obtaining ground truth data.

CHAPTER 6
Conclusion

In this thesis, we tackled the problem of learning influence probabilities between nodes in social networks using past data. Influence probabilities are assumed as inputs in information diffusion models such as the Linear Threshold models and the Independent Cascade model. Finding efficient and accurate solutions to this problem enables us to find more accurate solutions to other ones, such as the problem of influence maximization.

After presenting the necessary background in understanding this work, we delved into the theory behind the two diffusion models mentioned above, as well as the influence maximization problem. We then presented previous research in the field, which attempts to find solutions to the problem, for the two models. Most of this research assumes knowledge of the global structure of the network, as well as a rich dataset of past information cascades. The influence probabilities are learned either as empirical estimates from the data, or by solving maximum likelihood problems using techniques such as Expectation Maximization.

In chapter 3 we proposed a new method for learning influence probabilities between nodes. The method is grounded in agglomerative clustering techniques from the field of machine learning and uses only the local structure of a network. Specifically, the algorithm we propose learns a probability mass function on the outgoing edges of a chosen node using the past history of interactions between that specific node and its neighbors.

Besides relaxing the constraint of knowing the entire structure of the network, a significant advantage of this approach is computational efficiency. The algorithm can be run in parallel for multiple nodes of interest at once, and the results aggregated. We concluded the chapter by showing how the output of the algorithm can be transformed into the input for the Linear Threshold model. Implicitly, this means the output of Algorithm 4 can be used as the input for an entire class of diffusion algorithms, in the generalized framework presented in subsection (2.2.3).

In chapter 4 we presented a high-level overview of the architecture of a 3-tier web application we built to extract and store real data from Facebook, needed for performing experiments. The results of the algorithm are assessed by comparing them to ground truth data obtained by performing various experiments, detailed in chapter 5. First, we estimated the true fraction of influenceable neighbors in the network with the help of 16 volunteers which agreed to participate in multiple surveys. We then showed that most of the time, the fraction predicted by the algorithm falls in the confidence interval for the estimate. Second, we showed that the algorithm correctly marks significantly more nodes as truly influenceable in lists of fixed size, compared to an algorithm who would mark them at random. To further support this claim we showed the overlap between the nodes marked by the algorithm as being influenceable and the ones the volunteers mark, to be statistically significant (significance level of $p = 0.05$). For the last part of the analysis, we looked at the top 10 rankings of nodes (as returned by algorithm 4 and the volunteers) and concluded there is a positive correlation between them. The correlation is sometimes statistically significant, at a significance level of $p = 0.05$.

One of the core hypothesis we have made throughout our analysis is that common behavior of nodes is explained by influence, and not by homophily. Although we understand the limitations of this approach, we do note that studies such as the one made by Bisgin et al. [6] show this to be a valid assumption, at least in the context of large-scale online social networks. Another limitation is the fact that the Linear Threshold and the Independent Cascade model are not flexible enough to incorporate the notion of external influences in a meaningful way. Future work could explore other diffusion models, as well as exploit richer multi-dimensional datasets and the effect that tweaking the scores for different types of activity would have on the influence probabilities learned.

Another important direction for future work is to simulate information cascades in one of the diffusion models presented above, using the influence probabilities learned. We could then choose some specific seeding strategy and determine the influence spread of that seed set. Comparing the results of the simulation with the observed propagation would give us a good assessment of the quality of the influence probabilities learned through the proposed method. Testing different heuristics for scoring the nodes after the clustering is also something worth looking into, as it might give us better rankings or even more insight into the data. One last direction of future work would be to study how the influence probabilities we learned evolve in time, from one snapshot to another. Using our algorithm in tandem with one for performing community detection in social networks will allow us to determine whether there is significant correlation between the influence a community exerts on a specific node, and its decision to join or not. This would effectively enable us to predict how much influence a community must exert on a node on its fringe, before it decides to join it.

References

[1] Comments 4x more valuable than likes. http://edgerankchecker.com/blog/2011/11/comments-4x-more-valuable-than-likes/.

[2] MongoDB's official website. http://www.mongodb.org/.

[3] Nineteenth ACM Symposium on Principles of Distributed Computing. http://www.podc.org/podc2000/.

[4] Eytan Bakshy, Jake M. Hofman, Winter A. Mason, and Duncan J. Watts. Everyone's an influencer: quantifying influence on twitter. In *Proceedings of the fourth ACM international conference on Web search and data mining*, WSDM '11, pages 65–74, New York, NY, USA, 2011. ACM.

[5] A Barabási. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[6] Halil Bisgin, Nitin Agarwal, and Xiaowei Xu. Investigating Homophily in Online Social Networks. *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pages 533–536, August 2010.

[7] Wei Chen, Chi Wang, and Yajun Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, number January, pages 1029–1038. ACM, 2010.

[8] Wei Chen, Yajun Wang, and Siyu Yang. Efficient influence maximization in social networks. *Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining (2009)*, 67(1):199, 2009.

[9] Wei Chen, Yifei Yuan, and Li Zhang. Scalable influence maximization in social networks under the linear threshold model. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 88–97, Washington, DC, USA, 2010. IEEE Computer Society.

[10] Robert B Cialdini. *Influence: Science and Practice*, volume 4. Allyn and Bacon, 2001.

[11] Gerard Cornuejols, Marshall L. Fisher, and George L. Nemhauser. Location of bank accounts to optimize float: An analytic study of exact and approximate algorithms. *Management Science*, 23(8):pp. 789–810, 1977.

[12] A P Dempster, N M Laird, and D B Rubin. Maximum likelihood from incomplete data via em algorithm. *Journal of the Royal Statistical Society Series B Methodological*, 39(1):1–38, 1977.

[13] Pedro Domingos and Matt Richardson. Mining the network value of customers. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '01, pages 57–66, New York, NY, USA, 2001. ACM.

[14] David Easley and Jon Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*, volume 1. Cambridge University Press, 2010.

[15] Paul Erdős and Alfred Rényi. On the evolution of random graphs. *Evolution*, 5(1):17–61, 1960.

[16] B S Everitt, S Landau, and M Leese. *Cluster Analysis*, volume 33. Arnold, 2001.

[17] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[18] Zoubin Ghahramani. Unsupervised learning. In *Advanced Lectures on Machine Learning*, pages 72–112. Springer-Verlag, 2004.

[19] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[20] Manuel Gomez Rodriguez, Jure Leskovec, and Andreas Krause. Inferring networks of diffusion and influence. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 1019–1028, New York, NY, USA, 2010. ACM.

[21] Amit Goyal, Francesco Bonchi, and Laks V.S. Lakshmanan. Learning influence probabilities in social networks. In *Proceedings of the third ACM international conference on Web search and data mining*, WSDM '10, pages 241–250, New York, NY, USA, 2010. ACM.

[22] Mark Granovetter. Threshold models of collective behavior. *American Journal of Sociology*, 83(6):pp. 1420–1443, 1978.

[23] Oliver Hinz, Bernd Skiera, Christian Barrot, and J.U. Becker. Seeding Strategies for Viral Marketing: An Empirical Comparison. *Journal of Marketing*, 75(November):55–71, 2011.

[24] Anil K Jain and Richard C Dubes. *Algorithms for Clustering Data*, volume 355. Prentice Hall, 1988.

[25] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 137–146, New York, NY, USA, 2003. ACM.

[26] David Kempe, Jon Kleinberg, and Éva Tardos. Influential nodes in a diffusion model for social networks. In *IN ICALP*, pages 1127–1138. Springer Verlag, 2005.

[27] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):pp. 81–93, 1938.

[28] Masahiro Kimura and Kazumi Saito. Tractable models for information diffusion in social networks. *Knowledge Discovery in Databases PKDD 2006*, 4213:259–271, 2006.

[29] Masahiro Kimura, Kazumi Saito, Ryohei Nakano, and Hiroshi Motoda. Extracting influential nodes on a social network for information diffusion. *Data Min. Knowl. Discov.*, 20:70–97, January 2010.

[30] C Kiss and M Bichler. Identification of influencers Measuring influence in customer networks. *Decision Support Systems*, 46(1):233–253, December 2008.

[31] G N Lance and W T Williams. A general theory of classificatory sorting strategies. *The computer journal*, 10(3):271–277, 1967.

[32] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective outbreak detection in networks. *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining KDD 07*, 124(June):420, 2007.

[33] Jure Leskovec, Ajit Singh, and Jon Kleinberg. Patterns of influence in a recommendation network. In *In Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD*, pages 380–389. Springer-Verlag, 2005.

[34] Thomas M Liggett. *Interacting Particle Systems*, volume 276. Springer-Verlag, 1985.

[35] S Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[36] Antonio Loureiro, Luis Torgo, and Carlos Soares. Outlier detection using clustering methods: a data cleaning application. In *Proceedings of the Data Mining for Business Workshop*, 2004.

[37] Geoffrey J McLachlan and Thriyambakam Krishnan. *The EM Algorithm and Extensions*, volume 382. Wiley-Interscience, 2008.

[38] G W Milligan and M C Cooper. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50(2):159–179, 1985.

[39] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functionsi. *Mathematical Programming*, 14:265–294, 1978. 10.1007/BF01588971.

[40] Nishith Pathak, Arindam Banerjee, and Jaideep Srivastava. A generalized linear threshold model for multiple cascades. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 965–970, Washington, DC, USA, 2010. IEEE Computer Society.

[41] D Peleg. Local majority voting , small coalitions and controlling monopolies in graphs : A review 1 introduction. *Science*, (June):0–28, 1996.

[42] Matthew Richardson and Pedro Domingos. Mining knowledge-sharing sites for viral marketing. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '02, pages 61–70, New York, NY, USA, 2002. ACM.

[43] E.M. Rogers. *Diffusion of innovations*. Free Press of Glencoe, 1962.

[44] P Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20(1):53–65, 1987.

[45] Kazumi Saito, Ryohei Nakano, and Masahiro Kimura. Prediction of information diffusion probabilities for independent cascade model. In *Proceedings of the 12th international conference on Knowledge-Based Intelligent Information and Engineering Systems, Part III*, KES '08, pages 67–75, Berlin, Heidelberg, 2008. Springer-Verlag.

[46] Jie Tang, Jimeng Sun, Chi Wang, and Zi Yang. Social influence analysis in large-scale networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 807–816, New York, NY, USA, 2009. ACM.

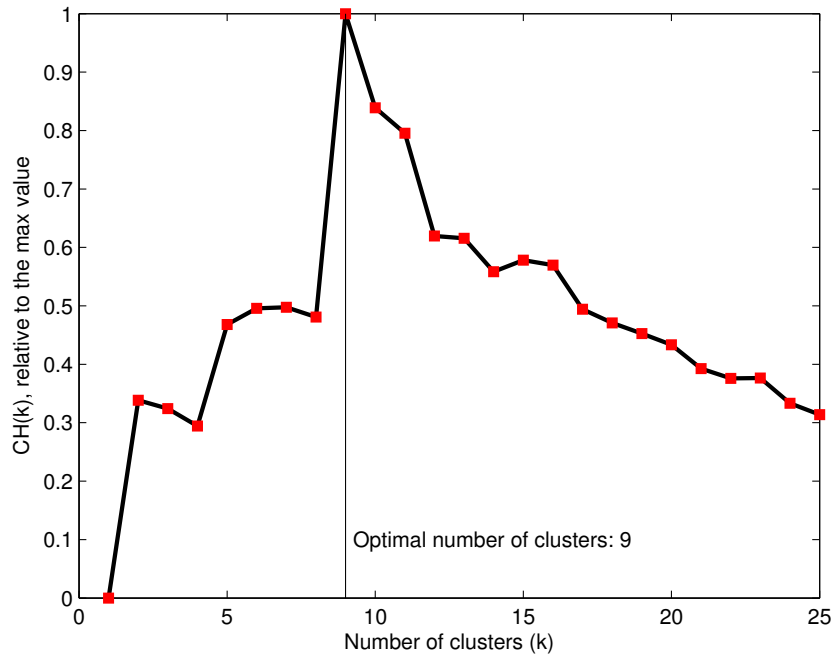
[47] T Velmurugan. A survey of partition based clustering algorithms in data mining: An experimental approach. *Information Technologies Journal*, 2011.

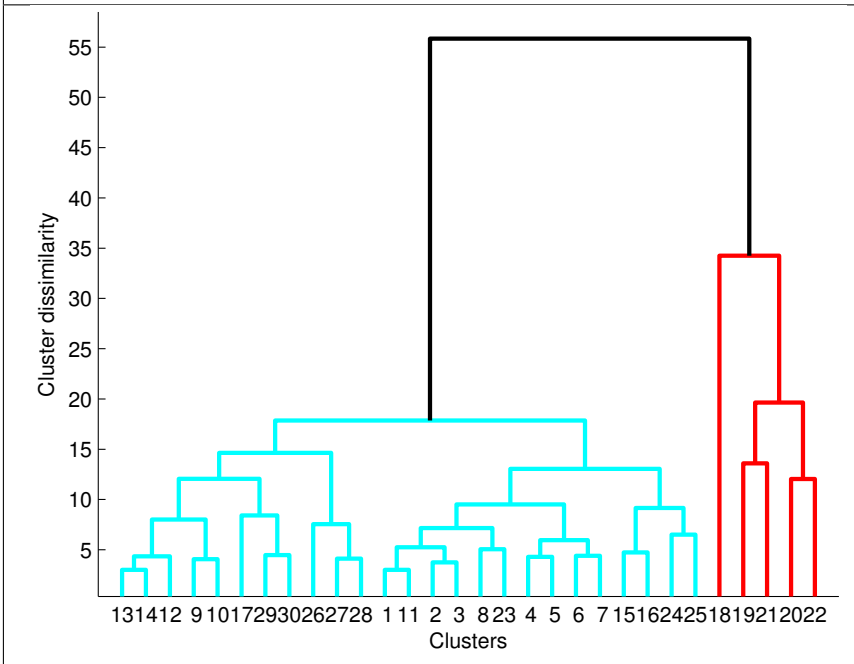
[48] Belal Al Zoubi. An Effective Clustering-Based Approach for Outlier Detection. *European Journal of Scientific Research*, 28(2):310–316, 2009.

Appendix A

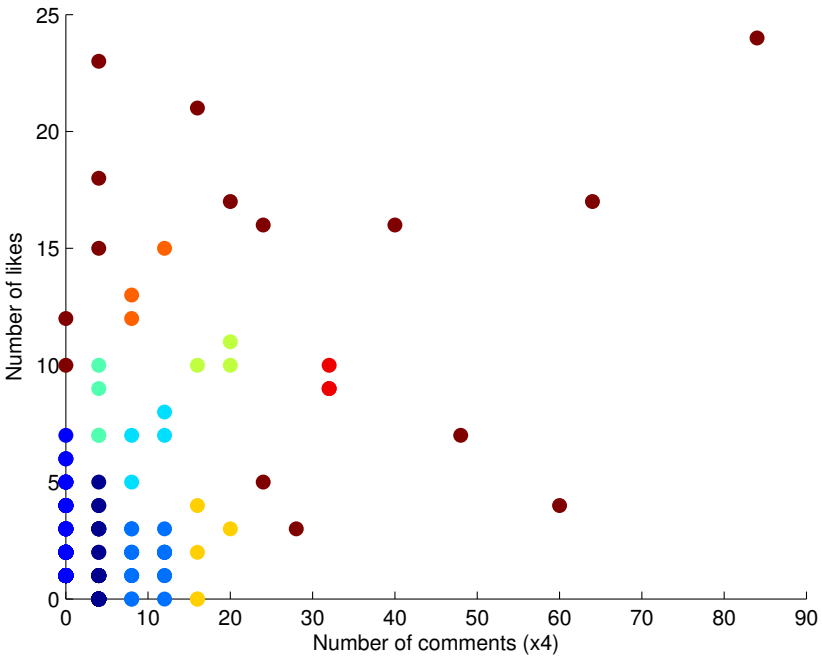
Below are plots describing the output of Algorithm 4 on the 16 datasets obtained from the volunteers. In each case, first plot shows the variation of the Calinsky-Harabasz index over the topmost 25 partitions in the sequence generated by the algorithm, the second one is a dendogram plot, and the last two are scatter and silhouette plots for the final clustering of each dataset. Note that in each scatter plot, the brown points correspond to outliers.

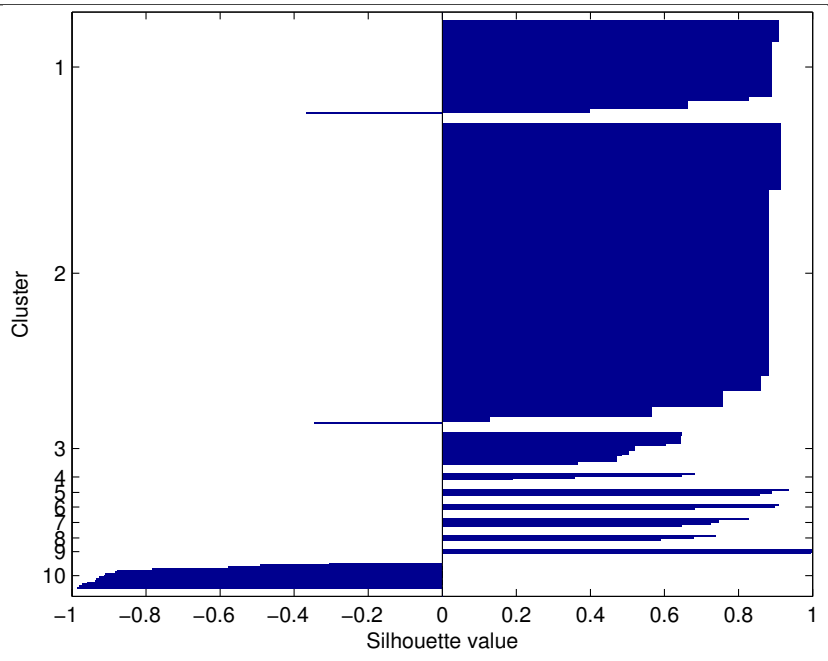
Plots for volunteer v_1, weighted linkage (1/2)



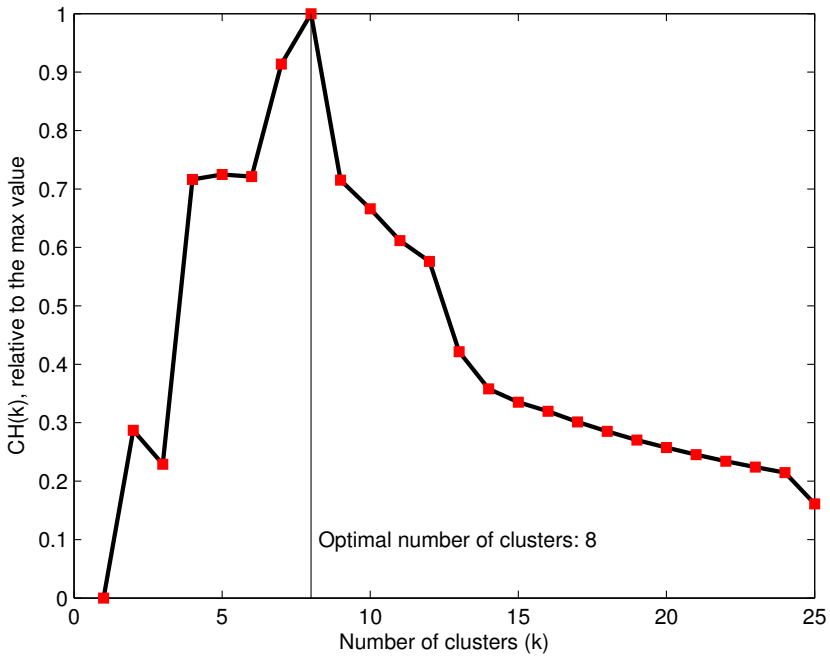


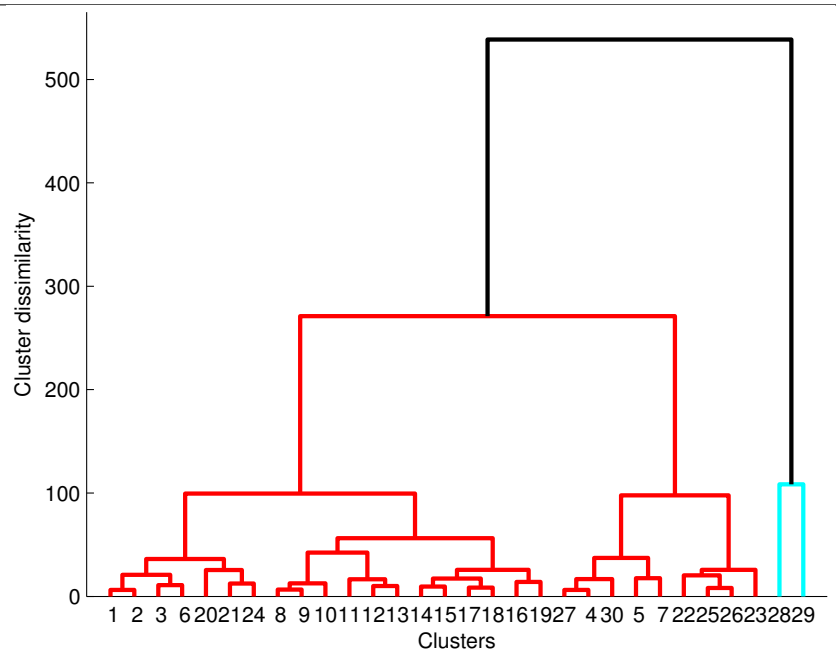
Plots for volunteer v_1, weighted linkage (2/2)



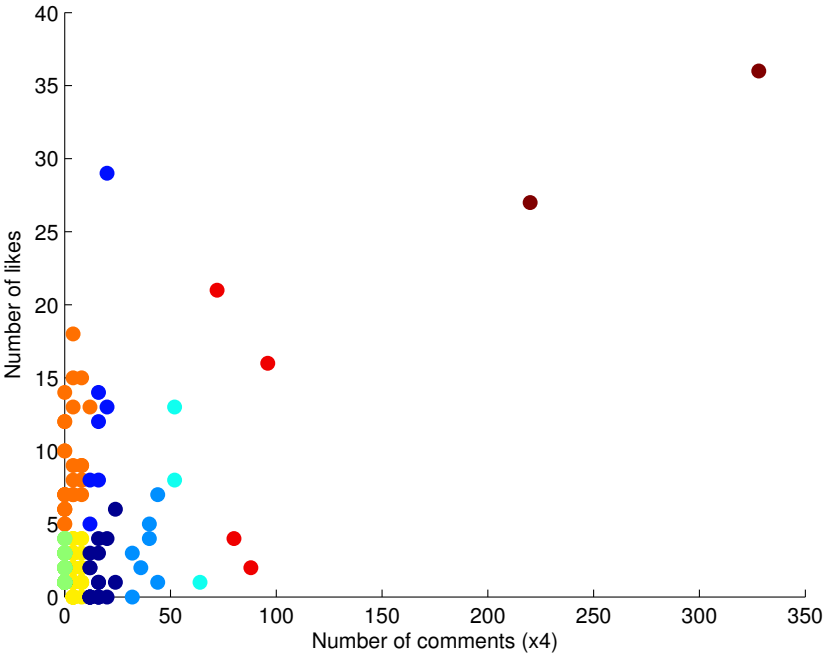


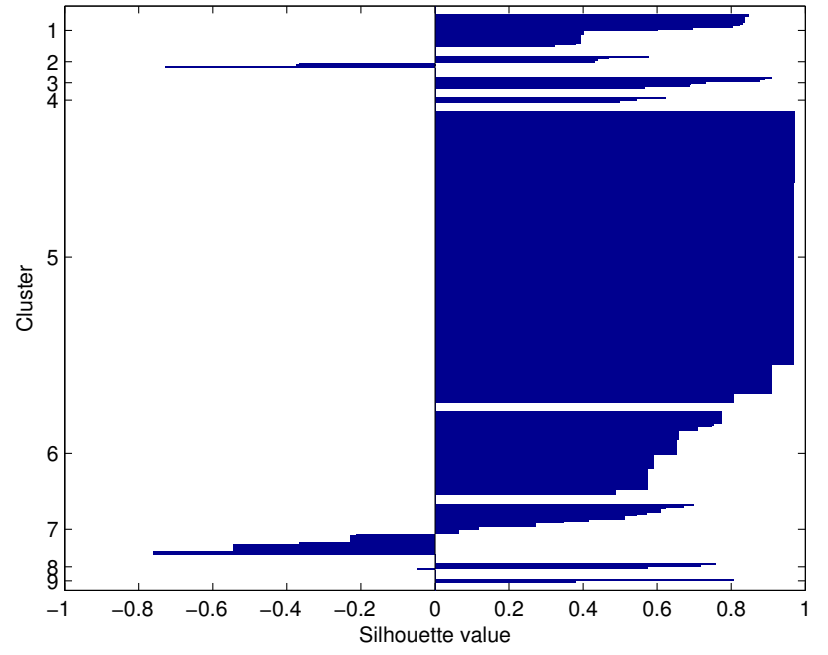
Plots for volunteer v_2, ward linkage (1/2)



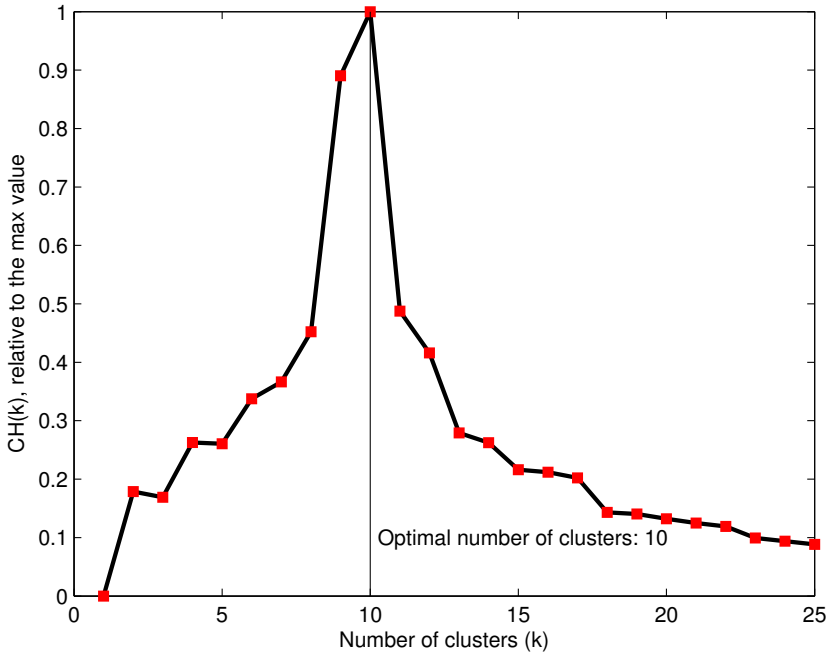


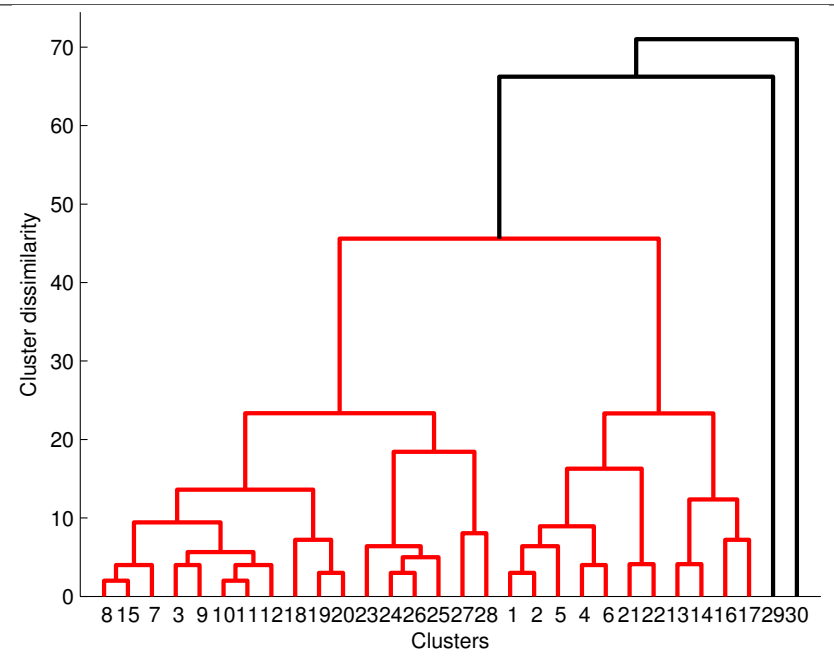
Plots for volunteer v_2, ward linkage (2/2)



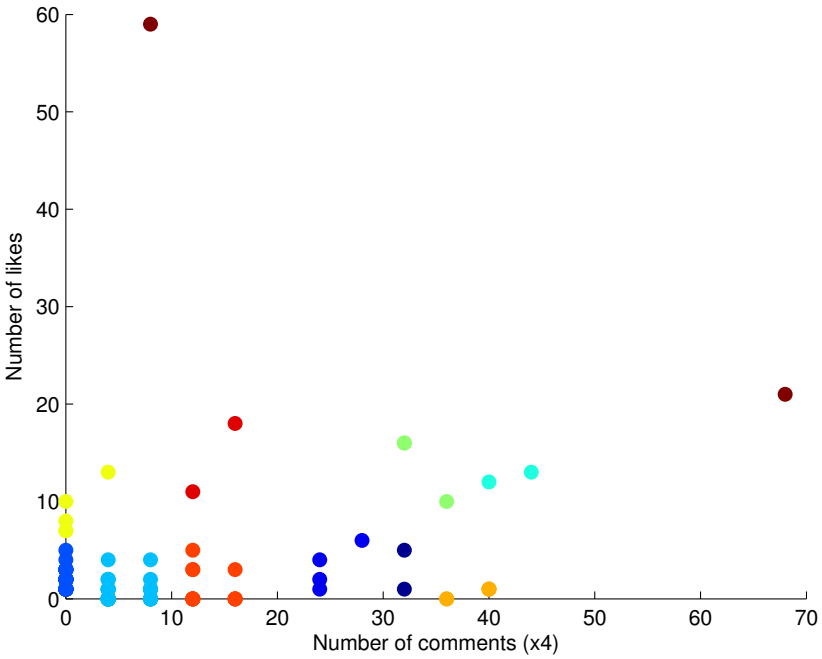


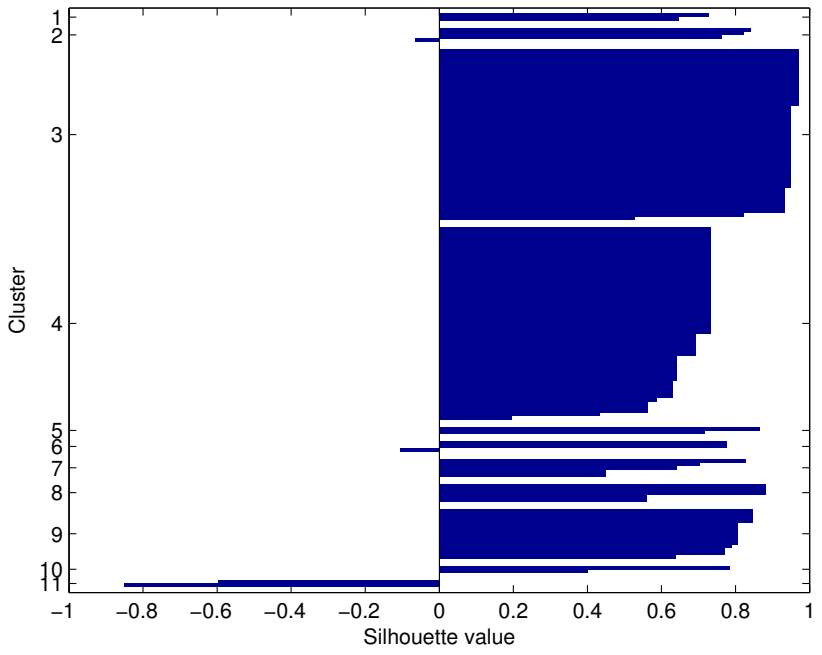
Plots for volunteer v_3, complete linkage (1/2)



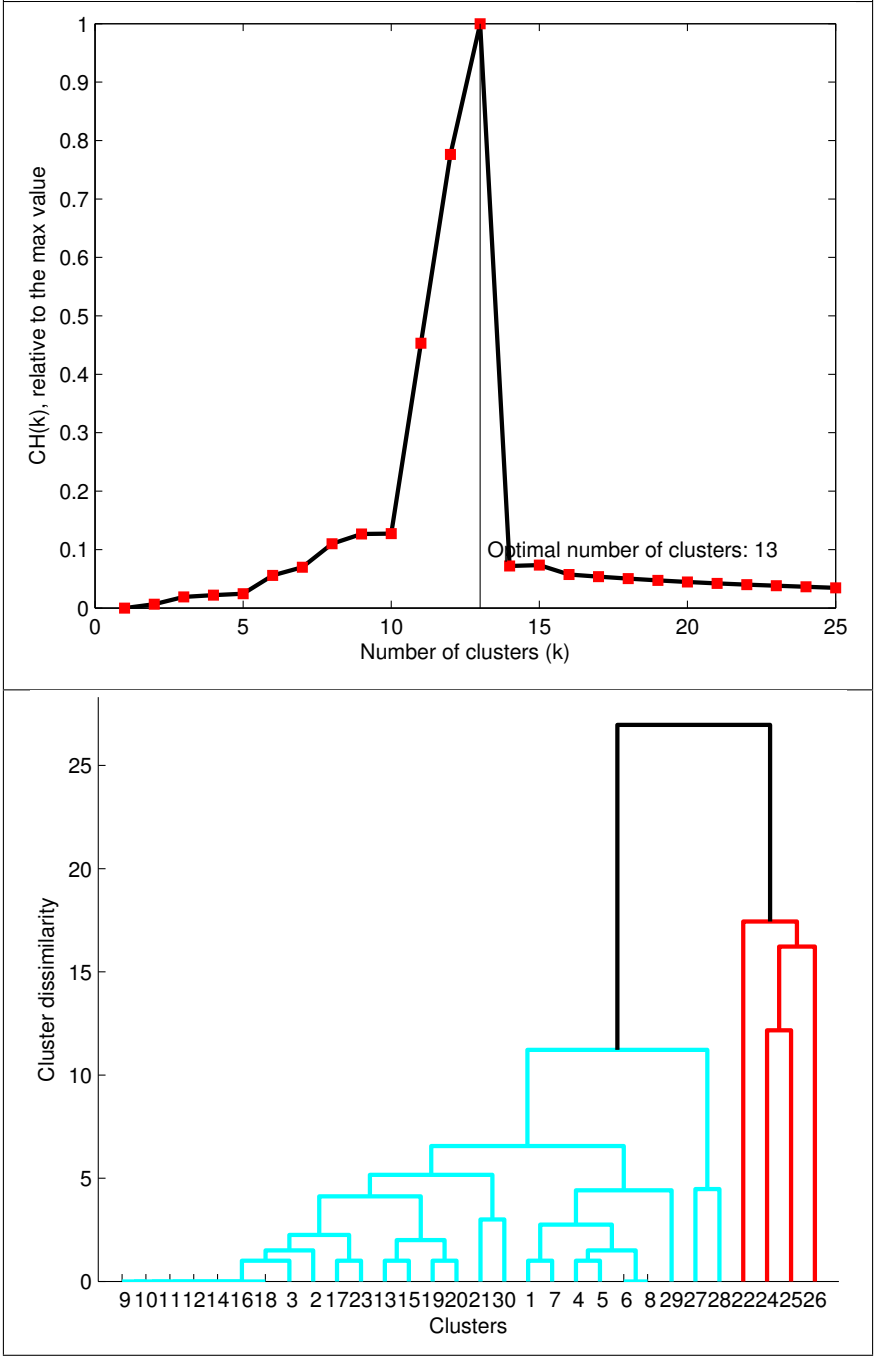


Plots for volunteer v_3, complete linkage (2/2)

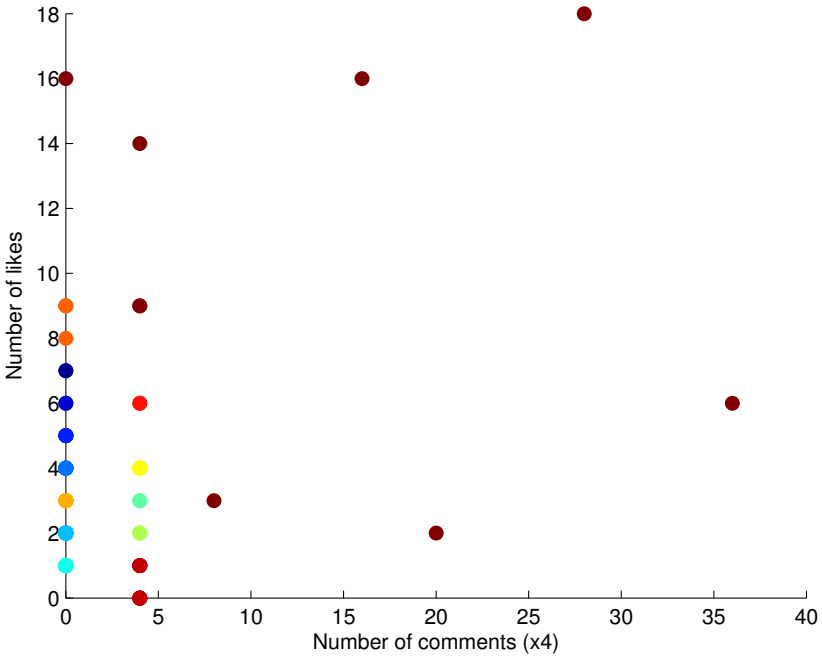


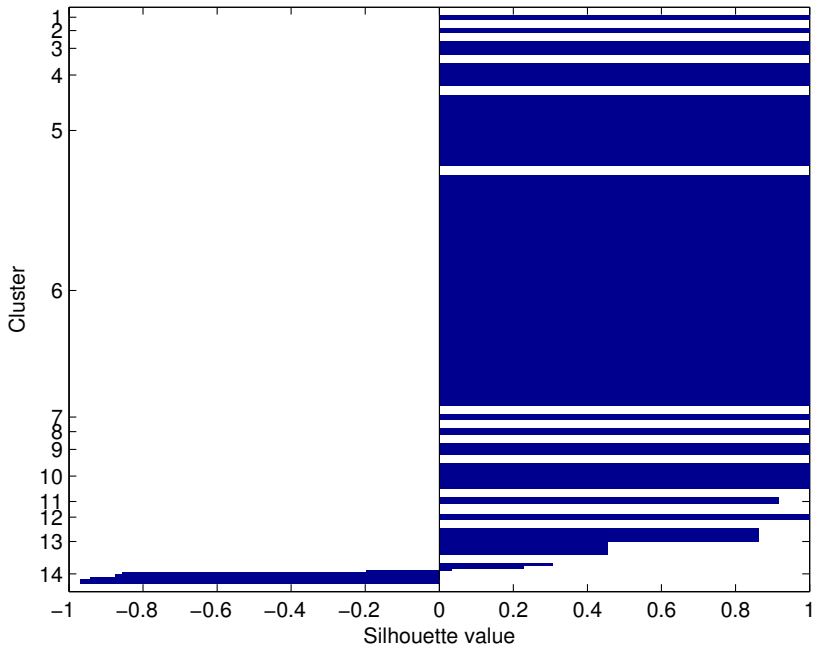


Plots for volunteer v_4, weighted linkage (1/2)

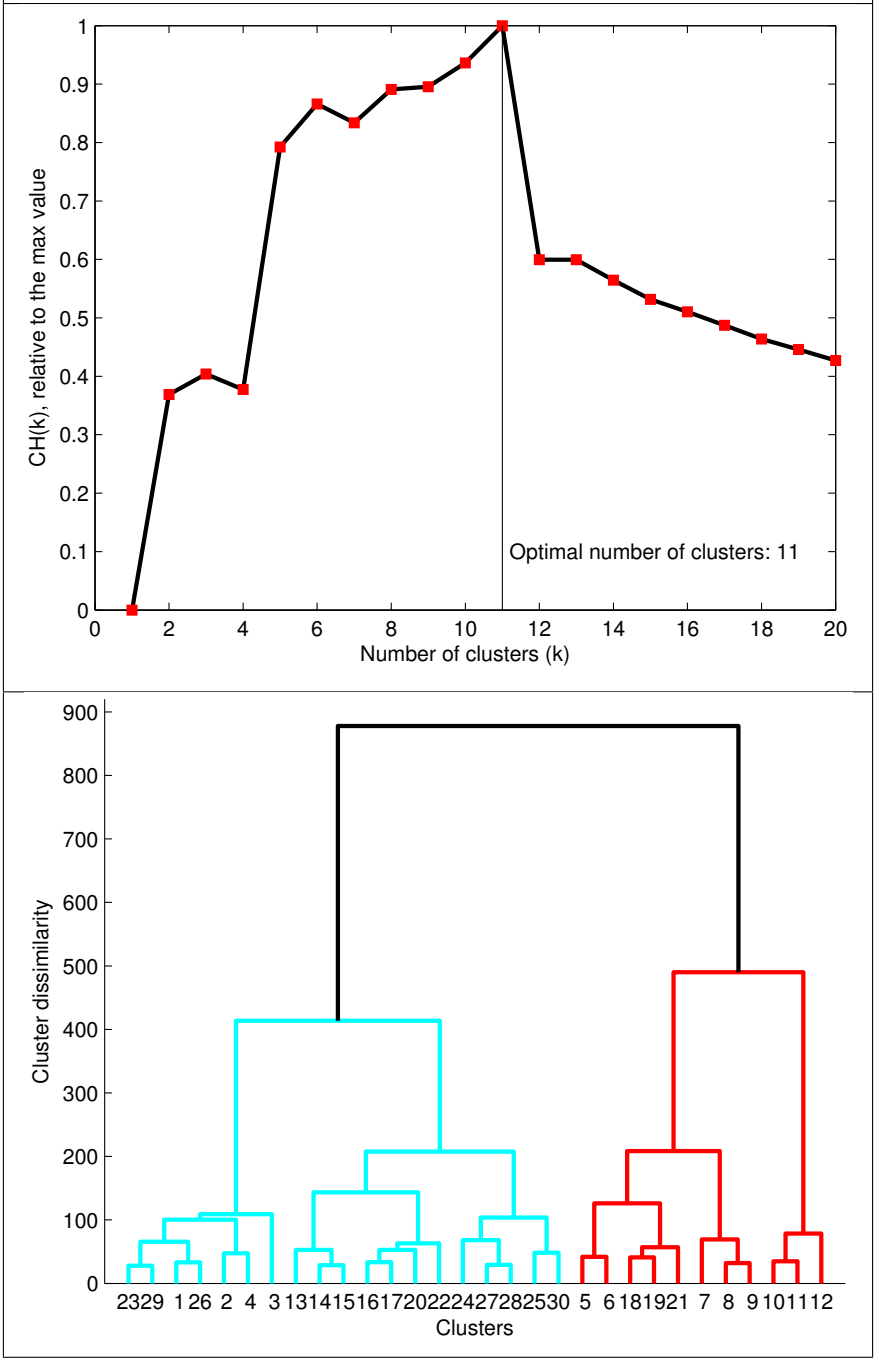


Plots for volunteer v_4, weighted linkage (2/2)

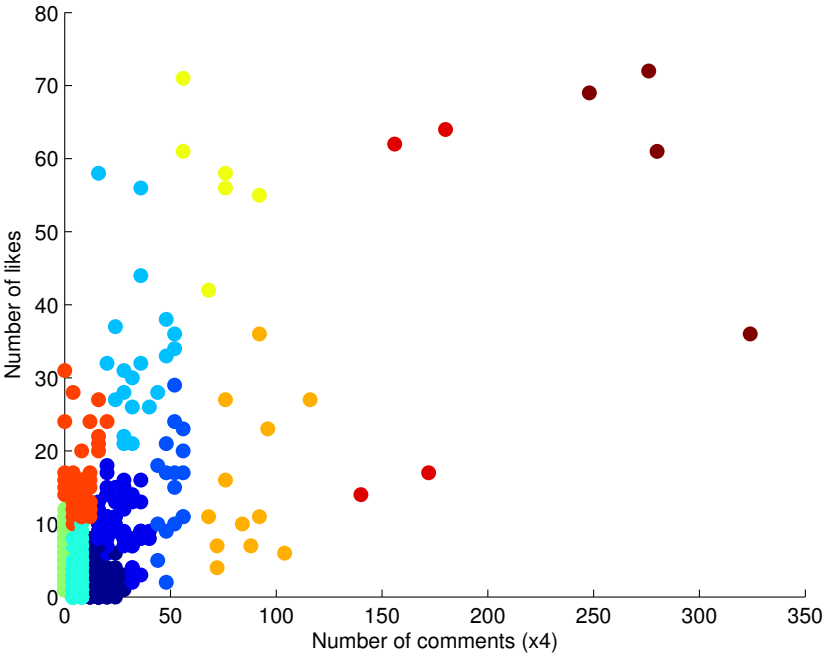


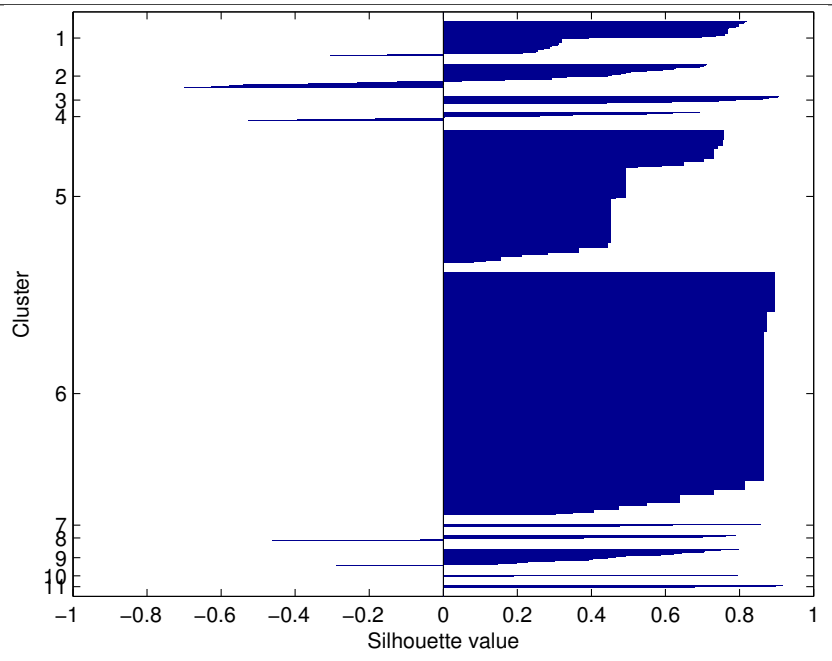


Plots for volunteer v_5, ward linkage (1/2)

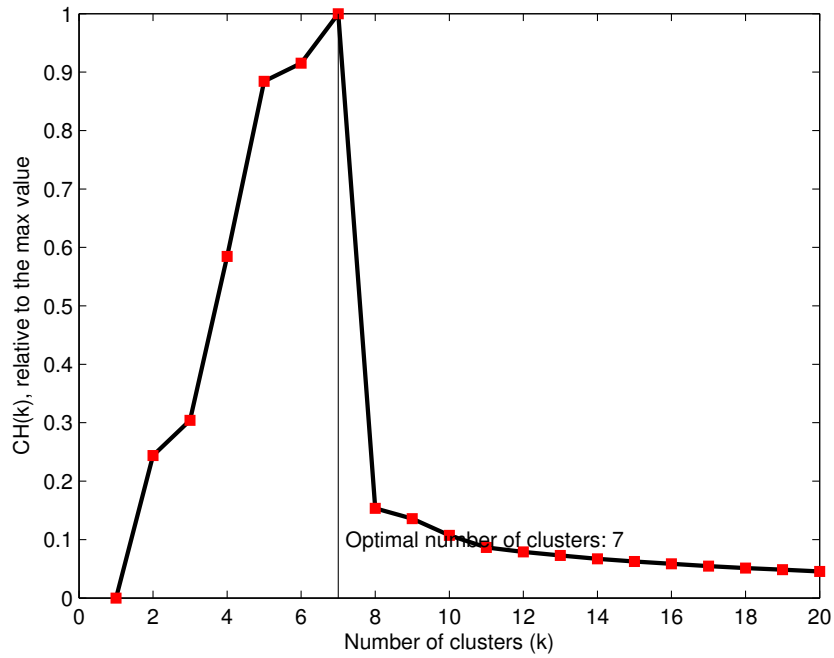


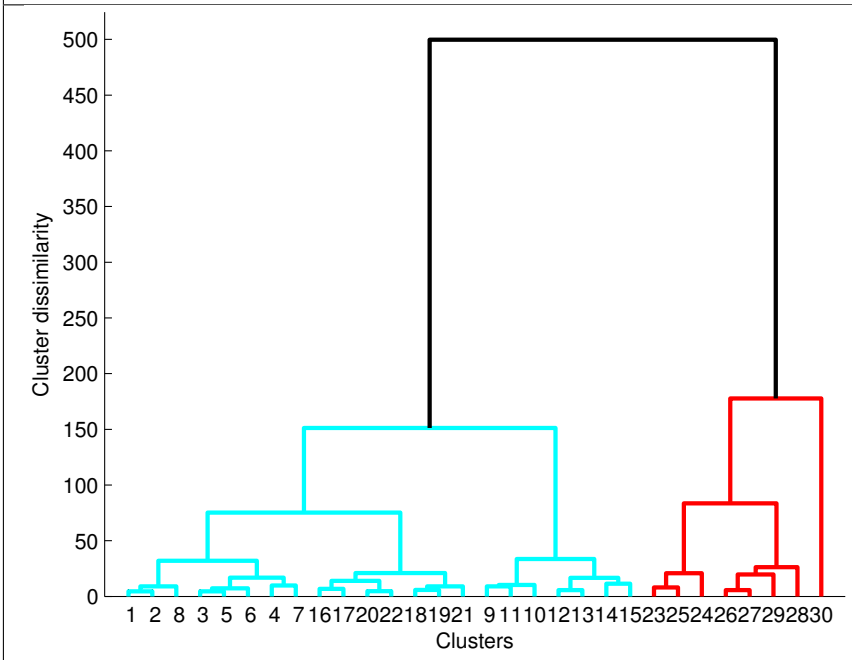
Plots for volunteer v_5, ward linkage (2/2)



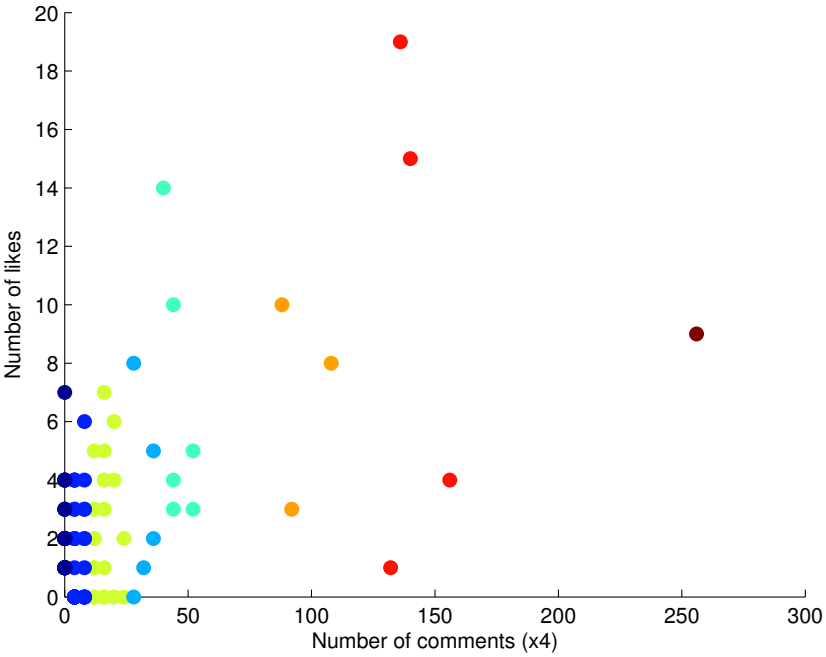


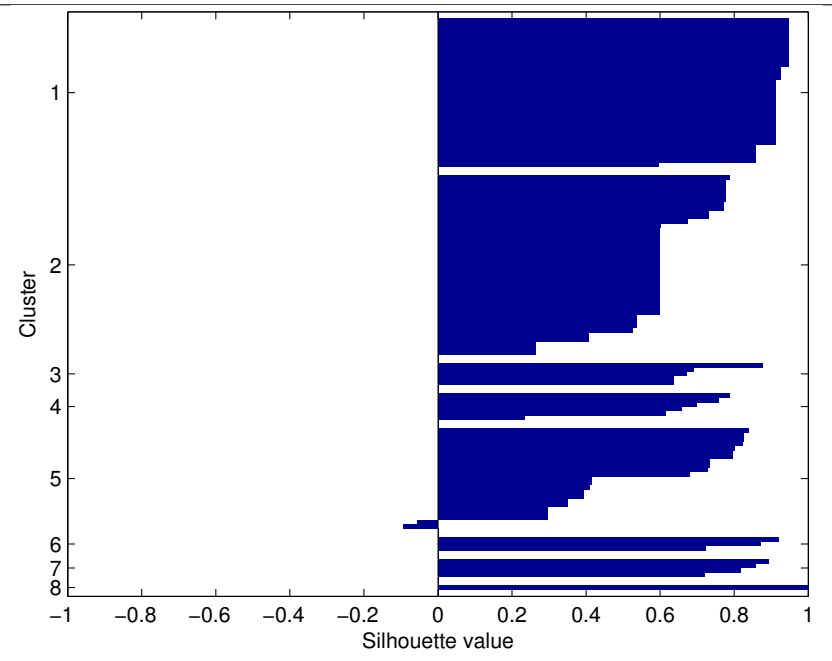
Plots for volunteer v_6, ward linkage (1/2)



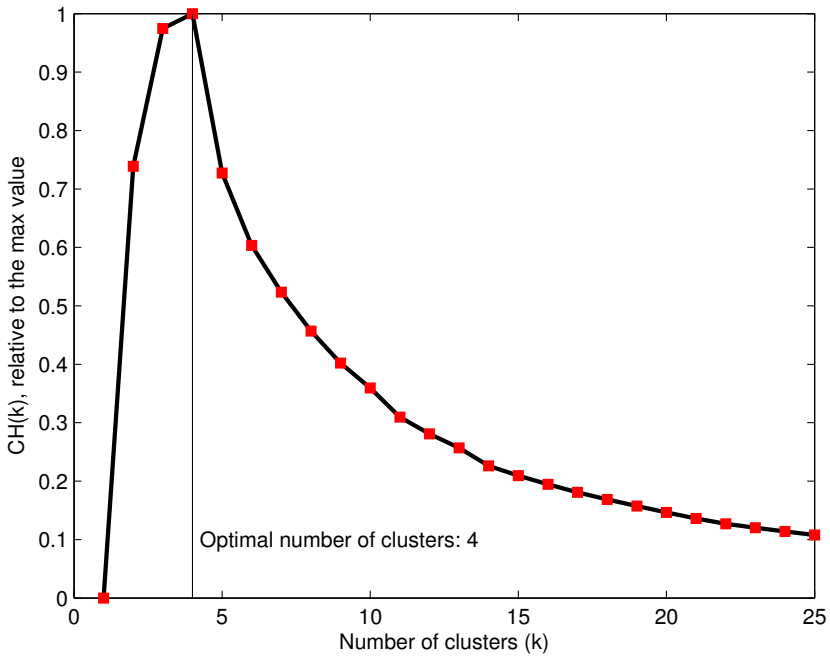


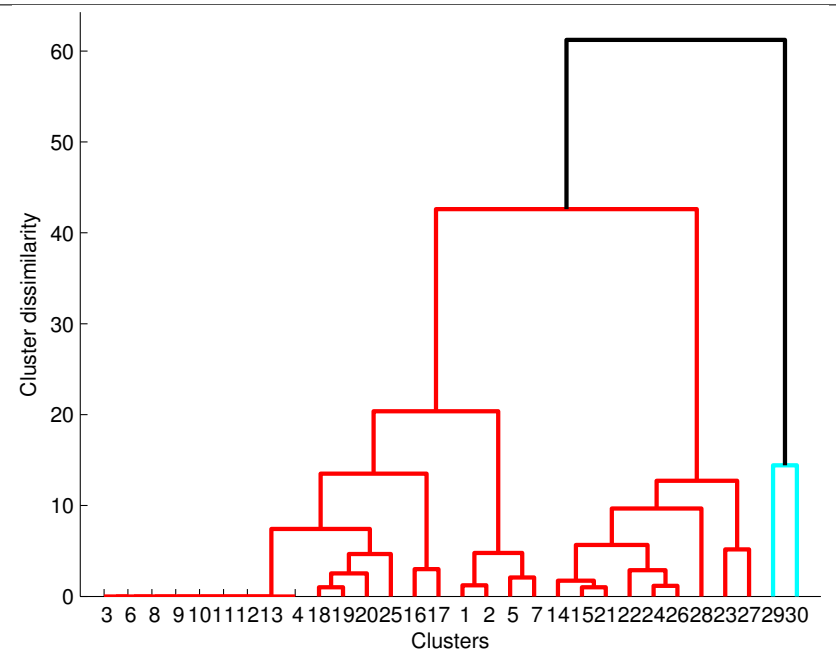
Plots for volunteer v_6, ward linkage (2/2)



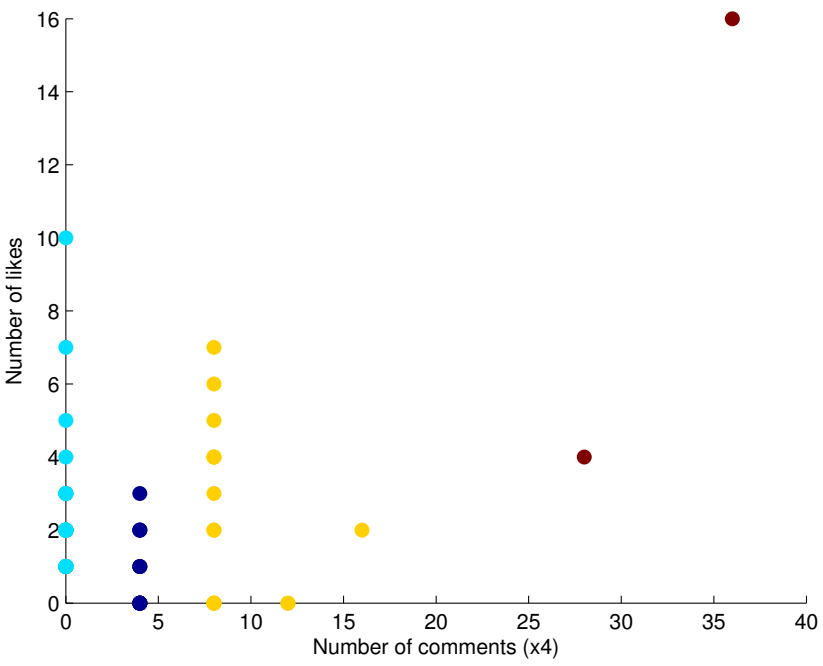


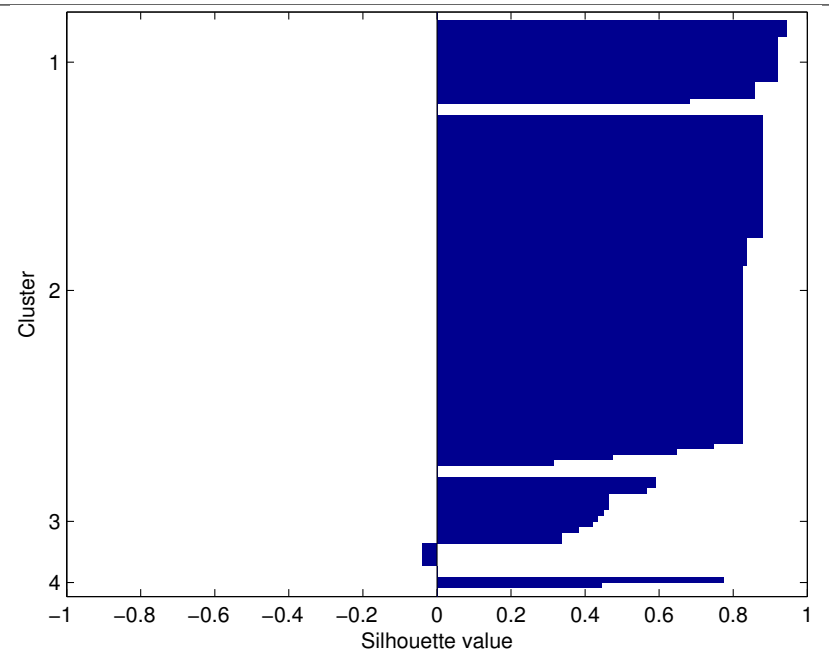
Plots for volunteer v_7, ward linkage (1/2)



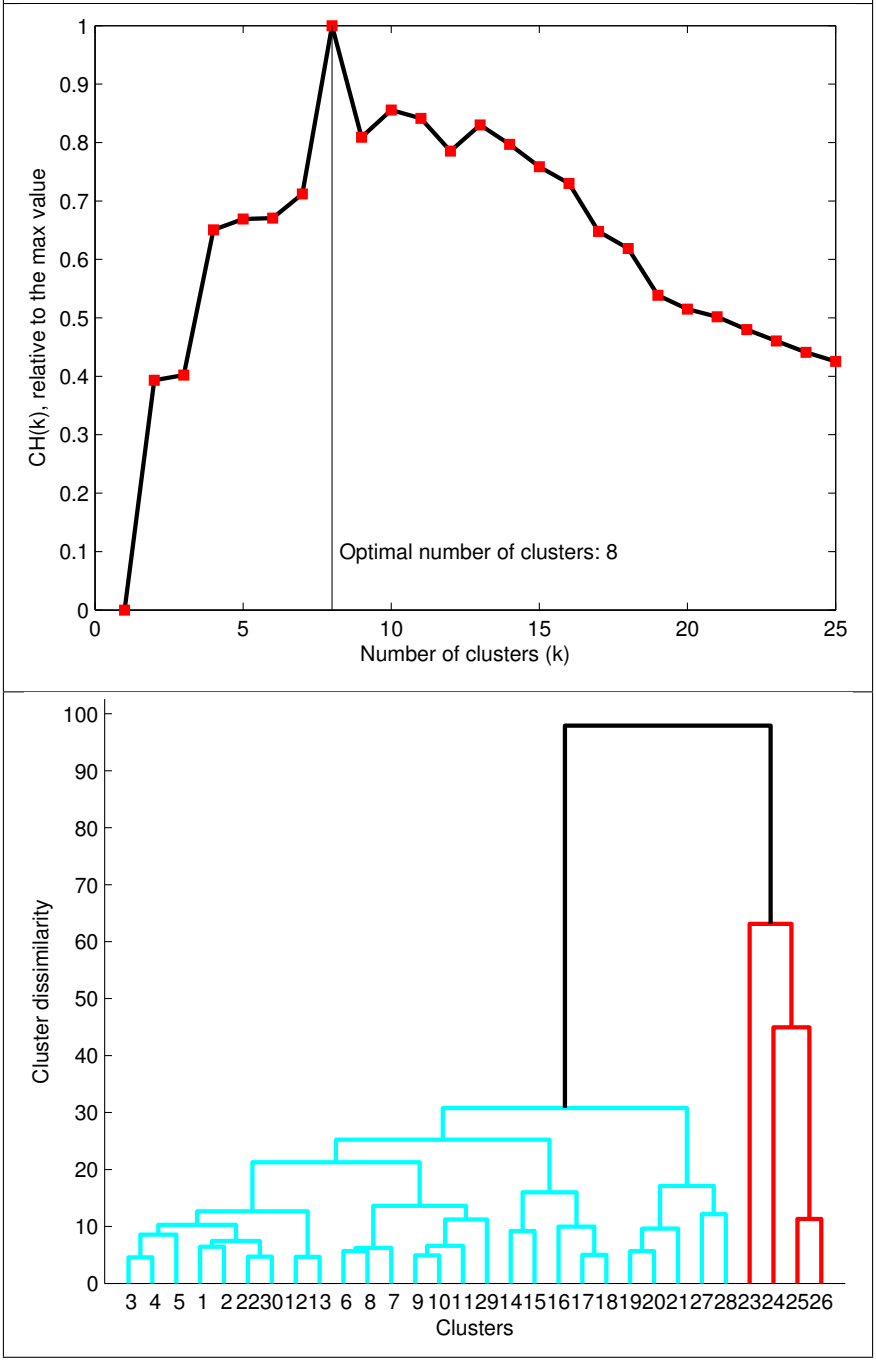


Plots for volunteer v_7, ward linkage (2/2)

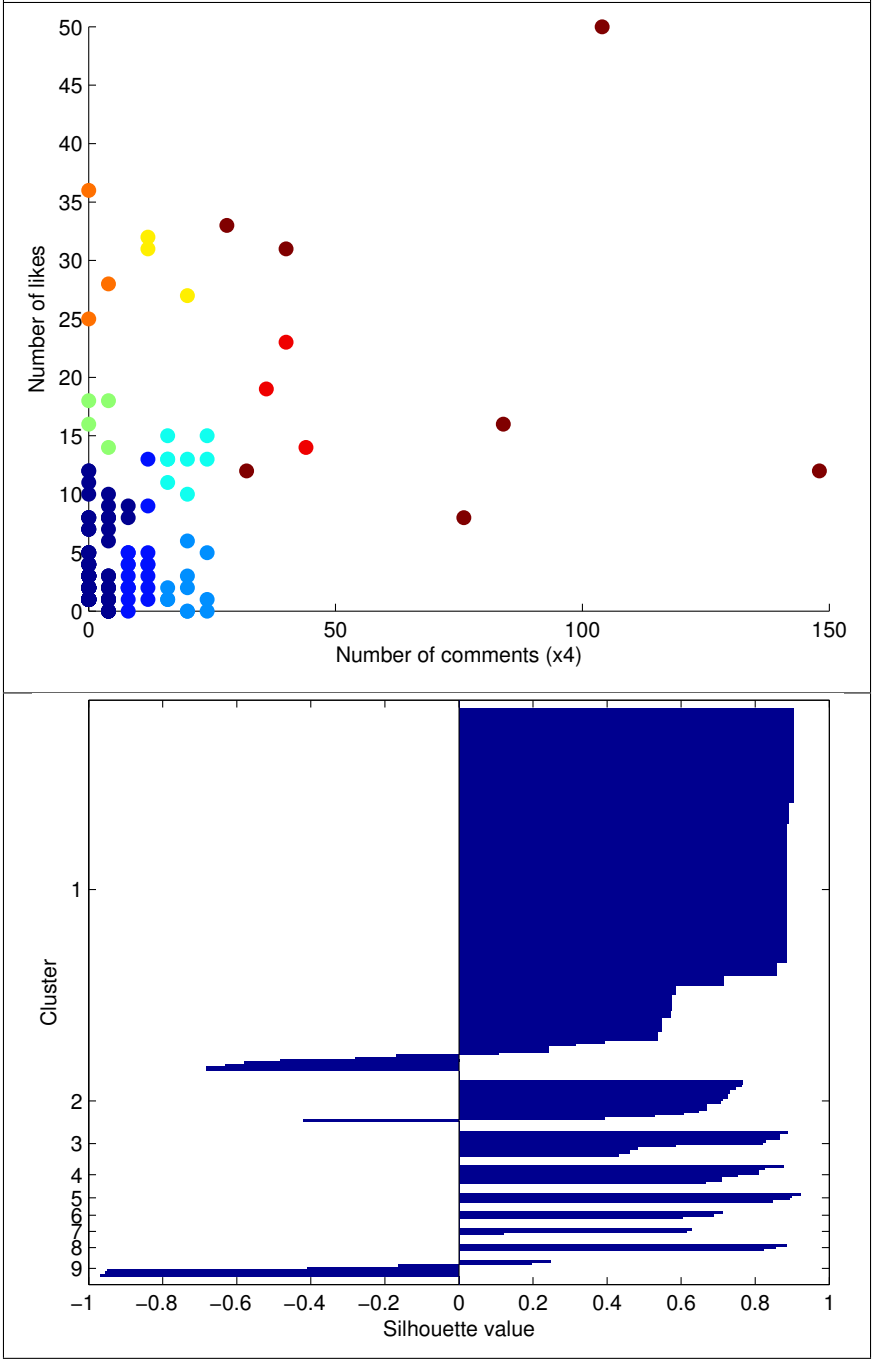




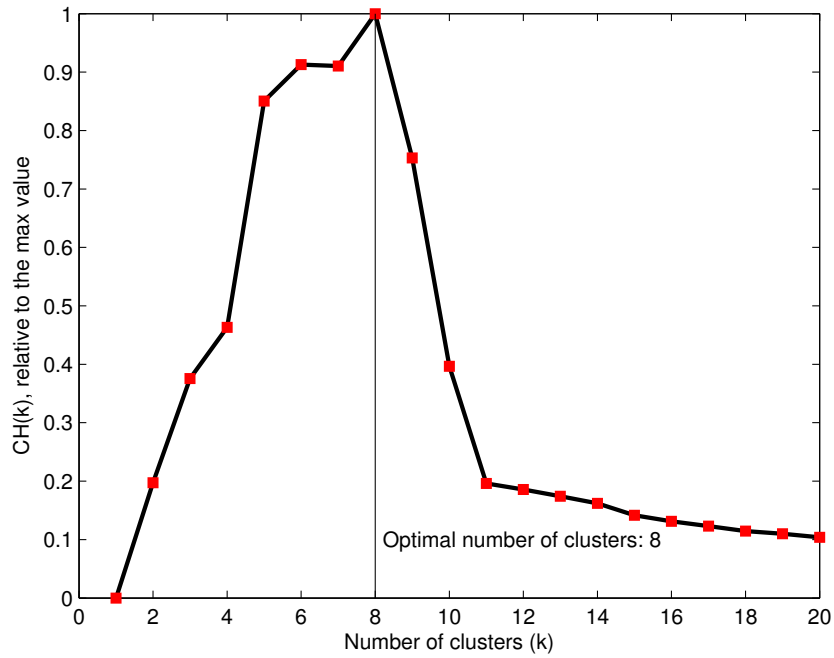
Plots for volunteer v_8, weighted linkage (1/2)

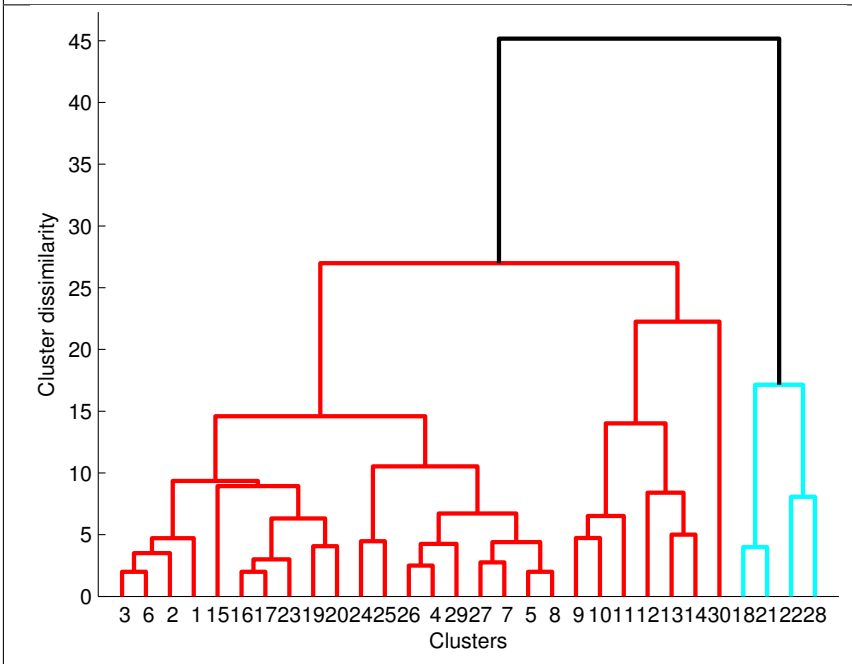


Plots for volunteer v_8, weighted linkage (2/2)

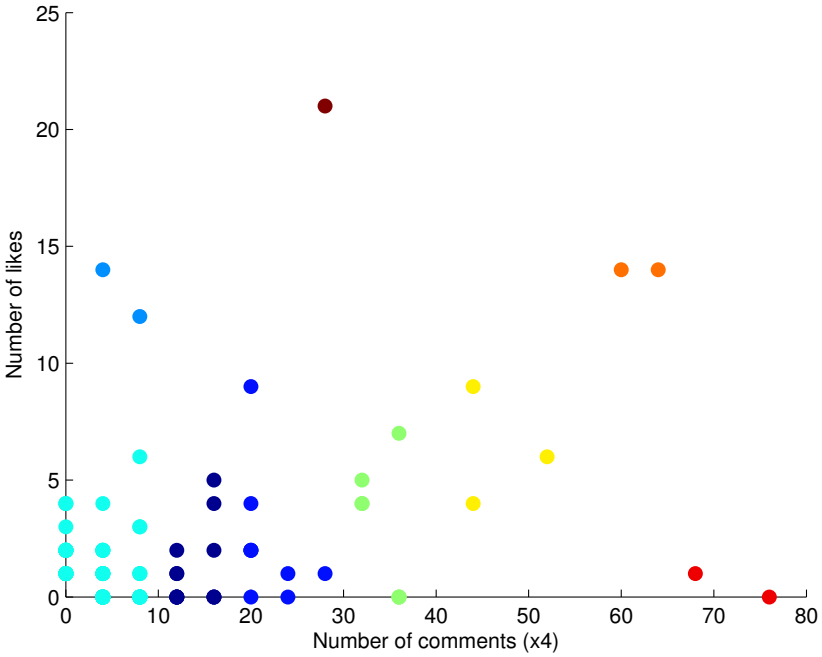


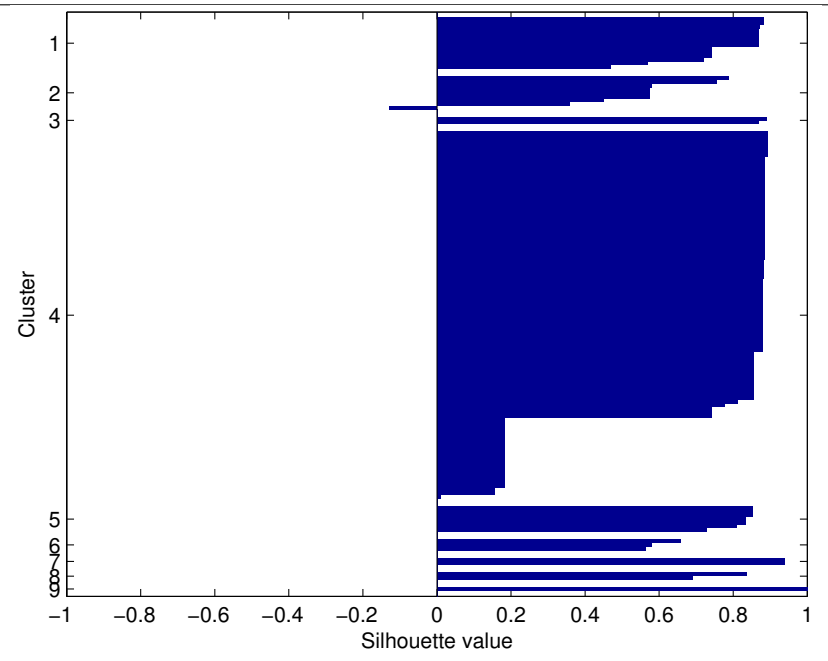
Plots for volunteer v_9, weighted linkage (1/2)



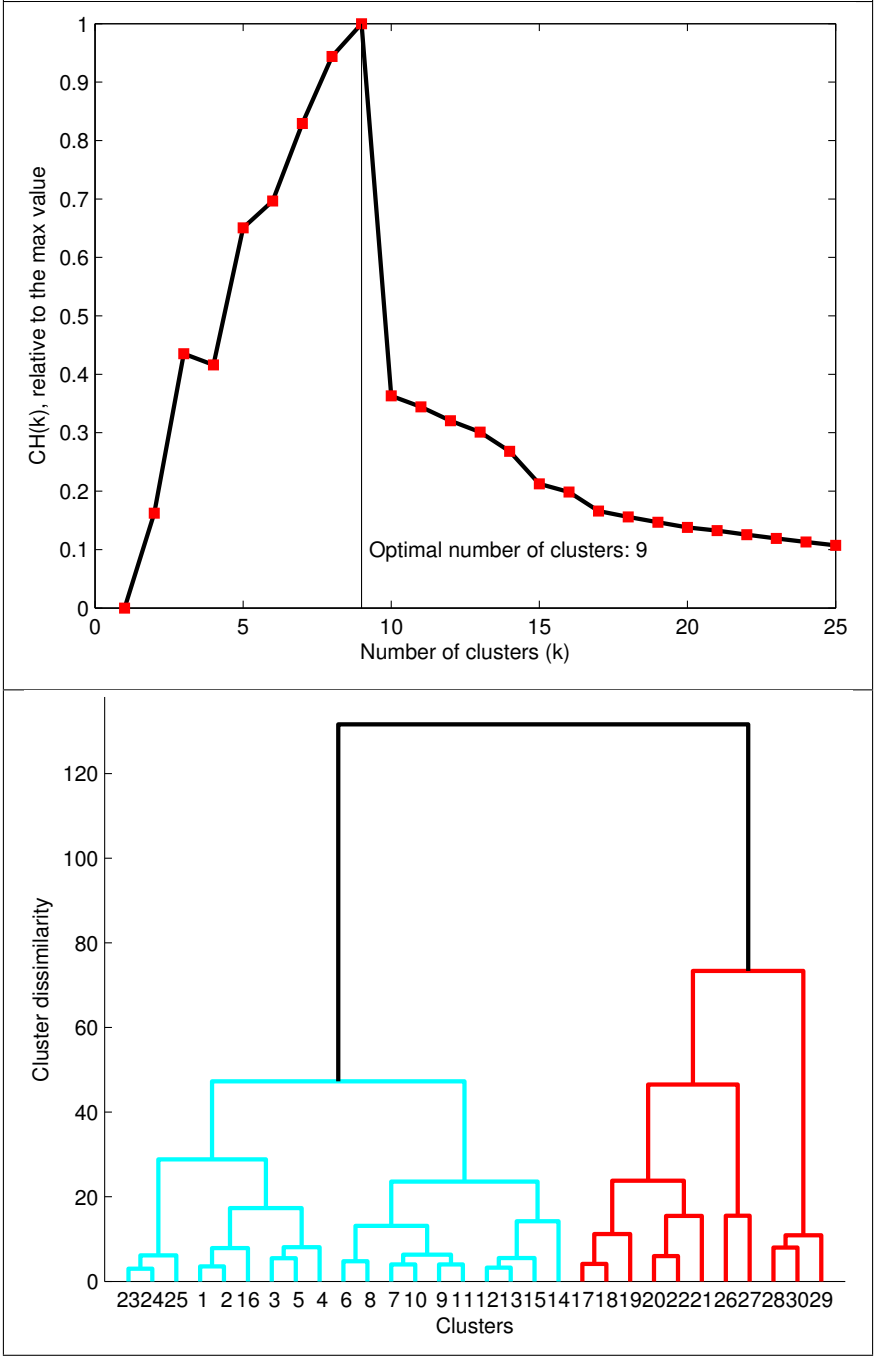


Plots for volunteer v_9, weighted linkage (2/2)

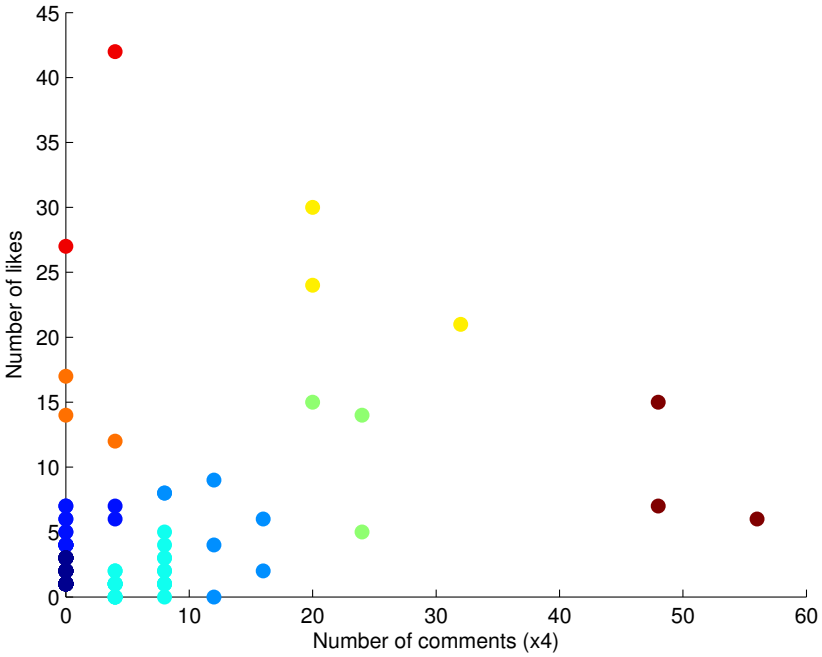


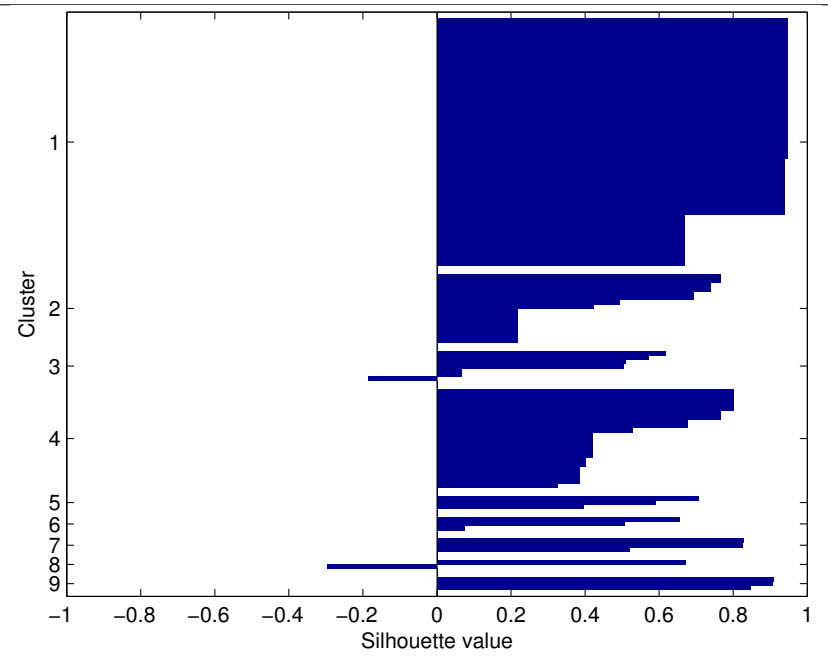


Plots for volunteer v_{10}, ward linkage (1/2)

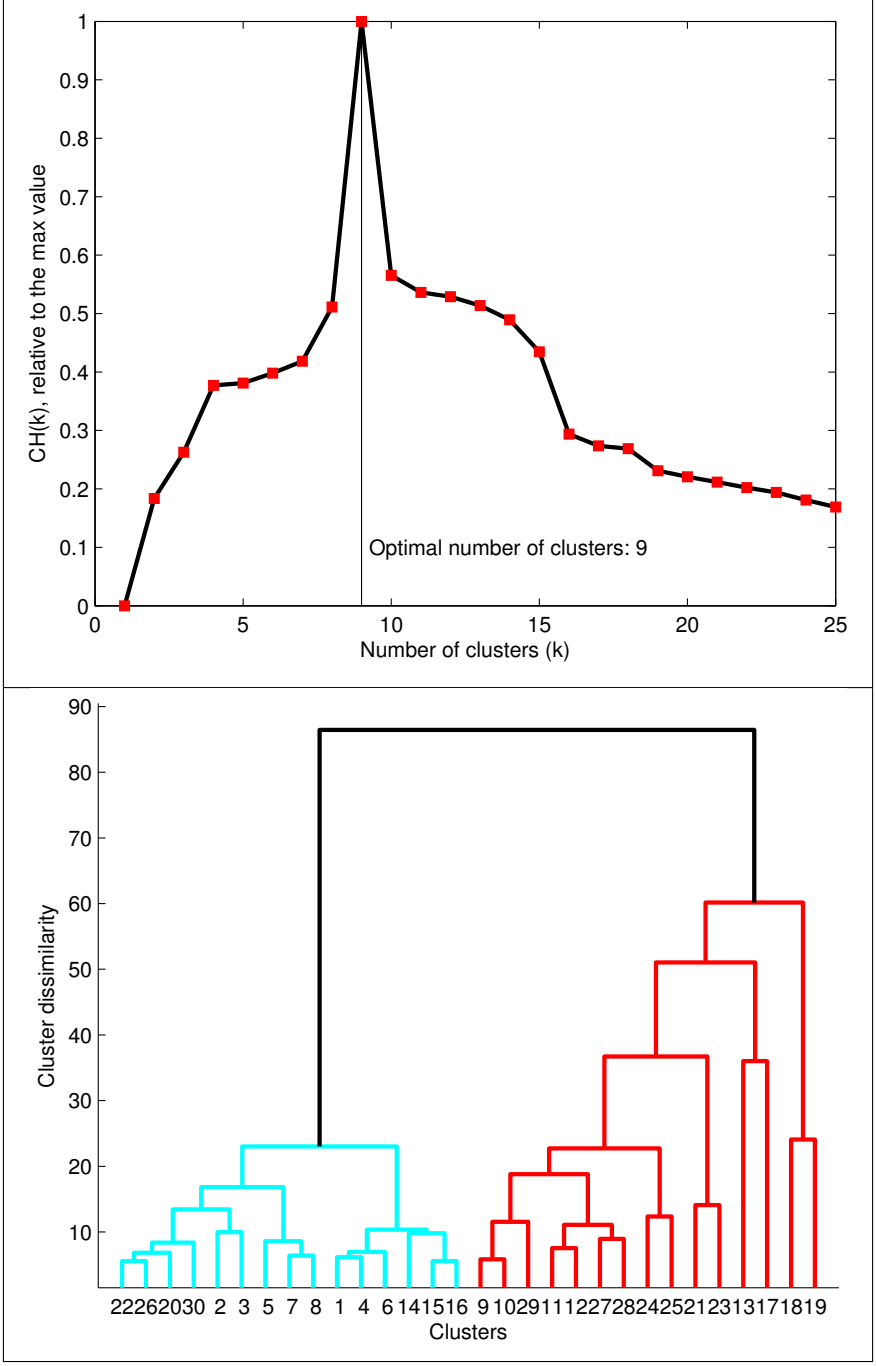


Plots for volunteer v_{10}, ward linkage (2/2)

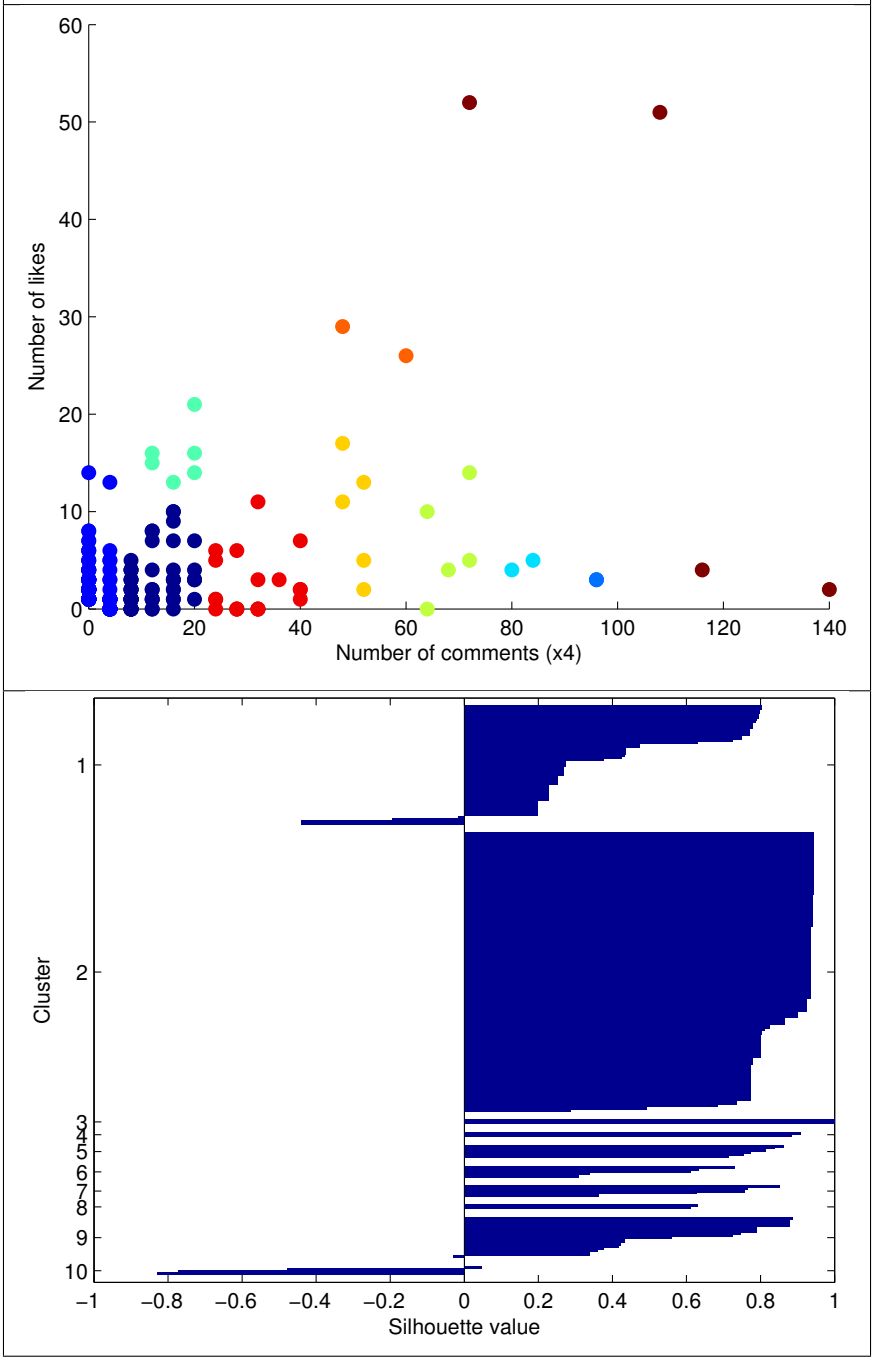




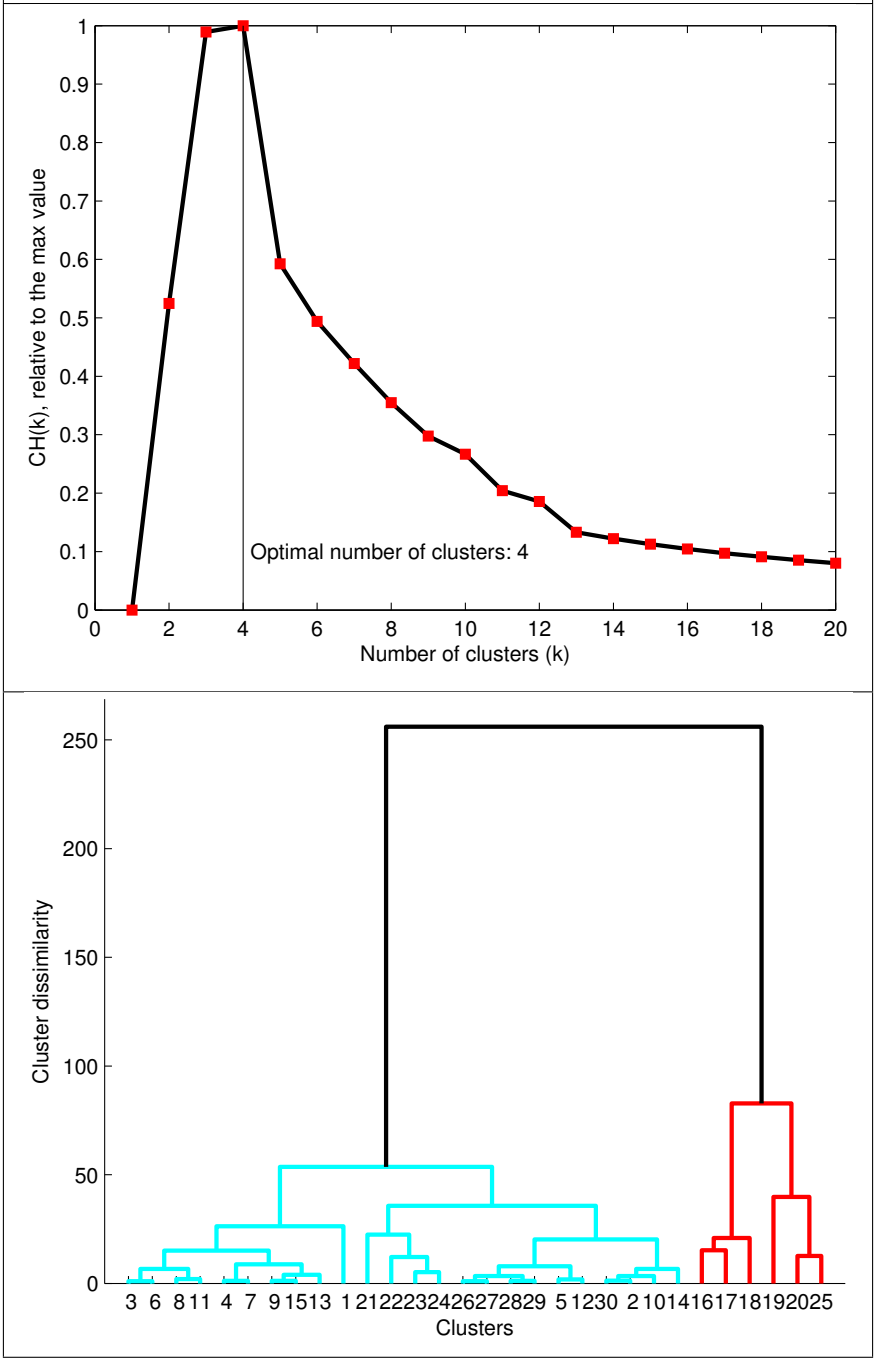
Plots for volunteer v_{11}, weighted linkage (1/2)



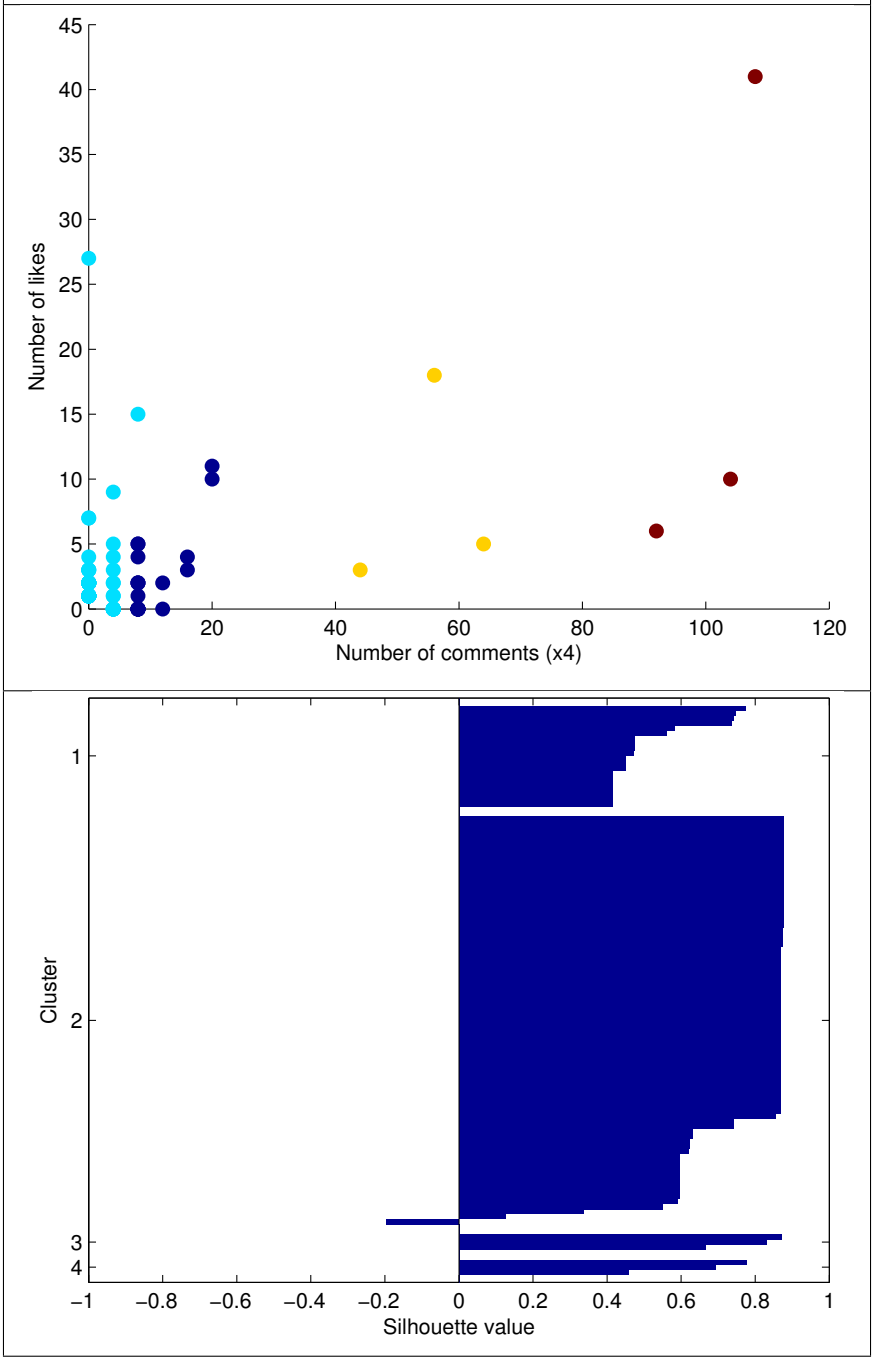
Plots for volunteer v_{11}, weighted linkage (2/2)



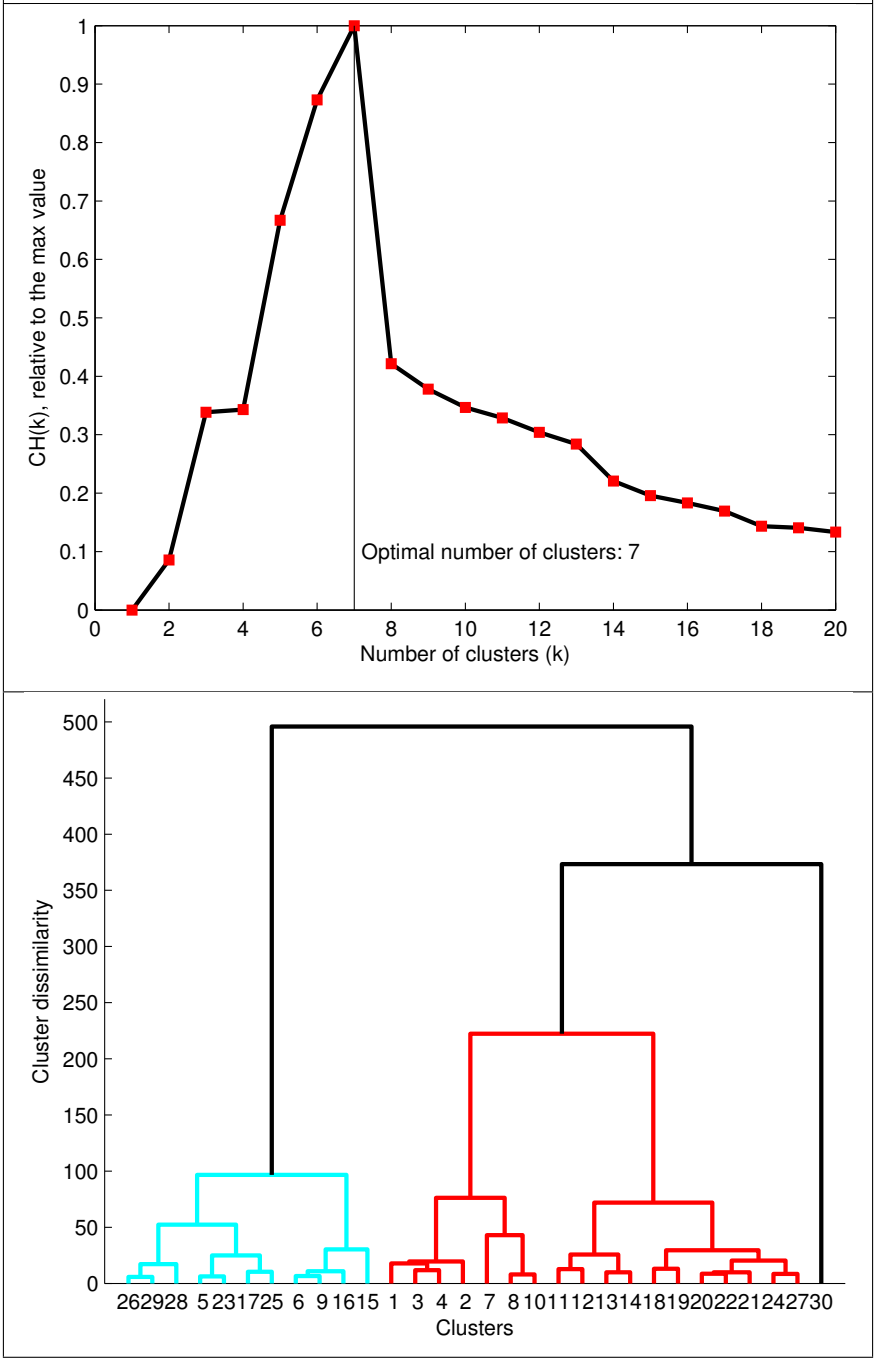
Plots for volunteer v_{12}, ward linkage (1/2)



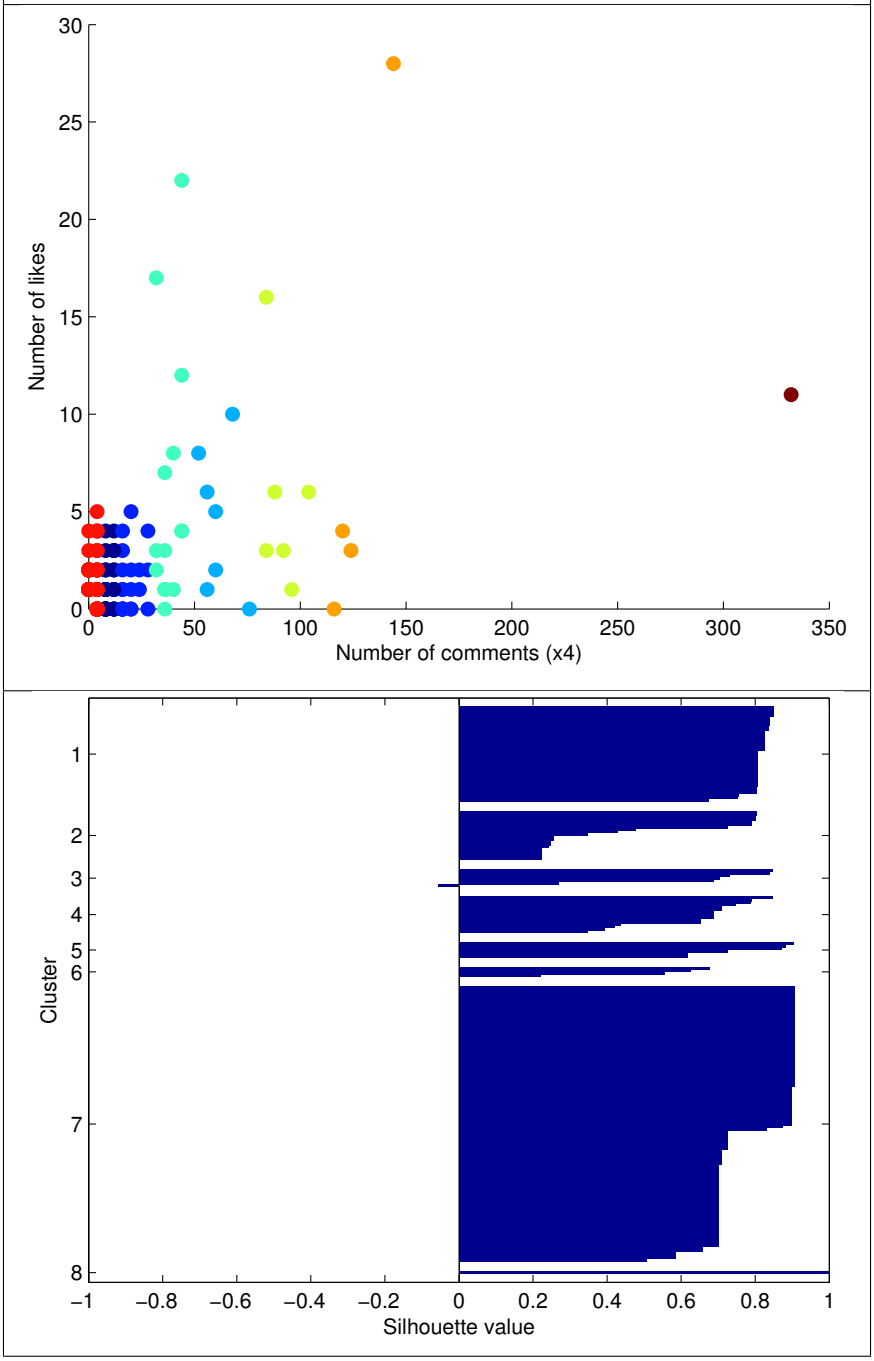
Plots for volunteer v_{12}, ward linkage (2/2)



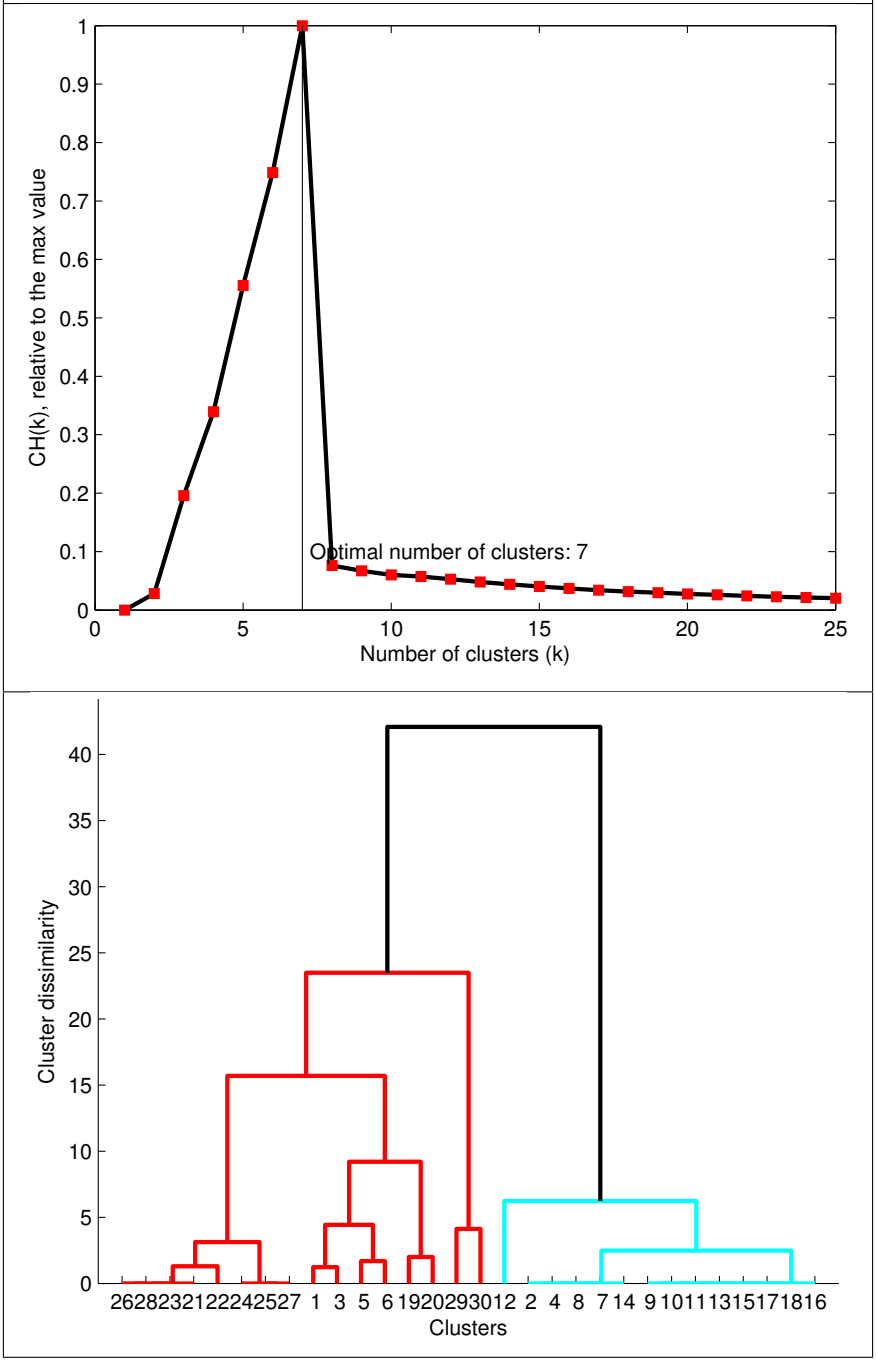
Plots for volunteer v_{13}, ward linkage (1/2)



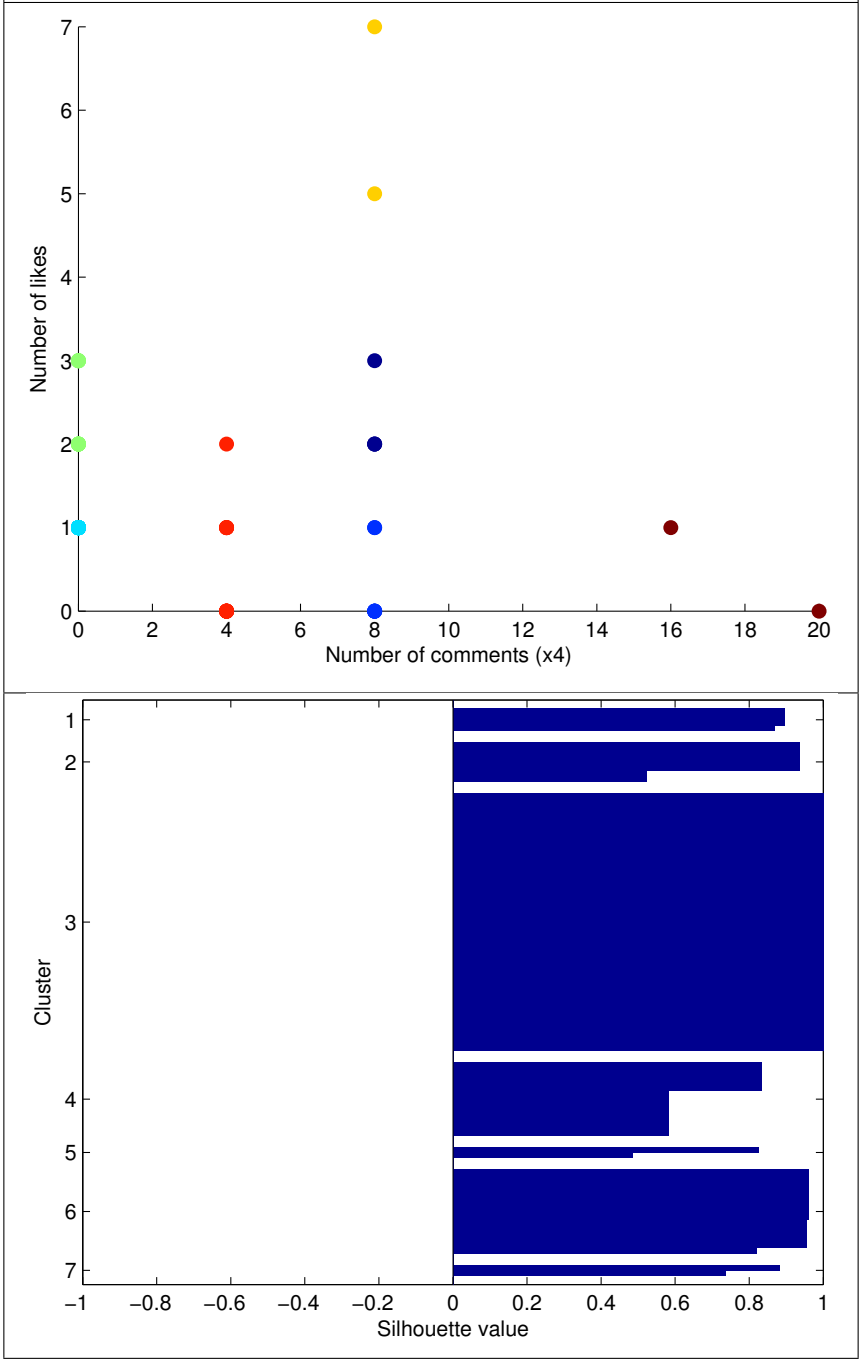
Plots for volunteer v_{13}, ward linkage (2/2)



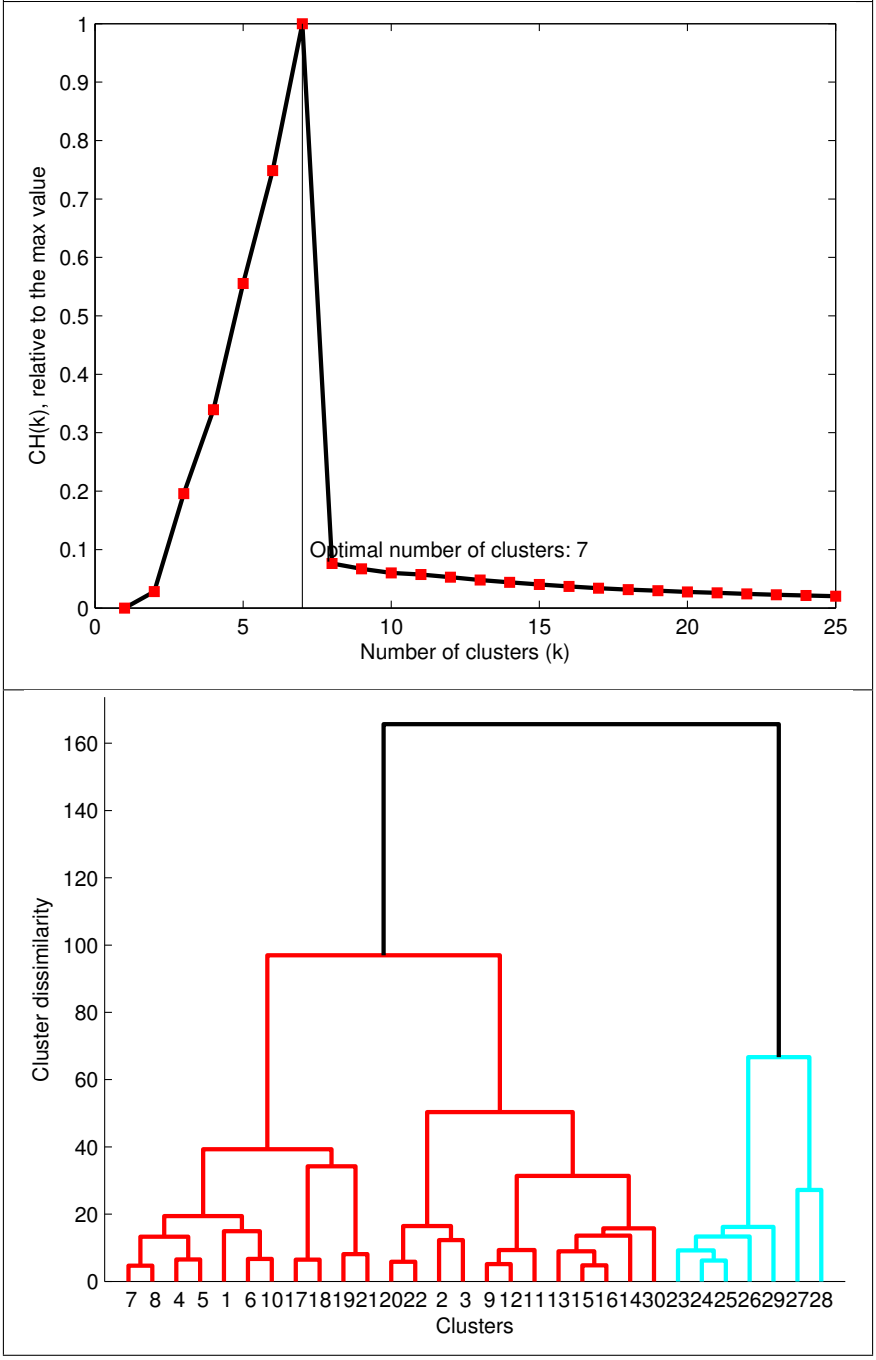
Plots for volunteer v_{14}, ward linkage (1/2)



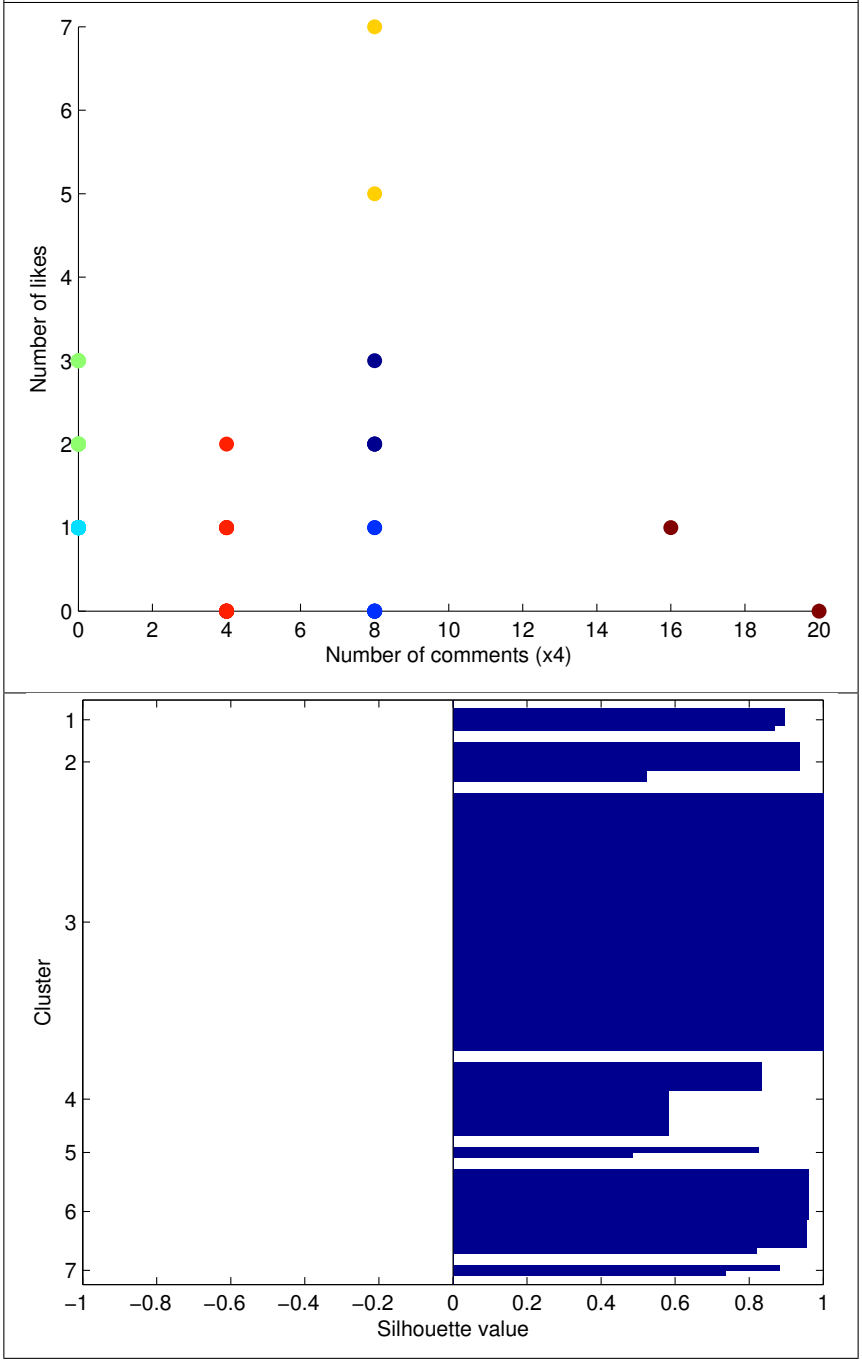
Plots for volunteer v_{14}, ward linkage (2/2)



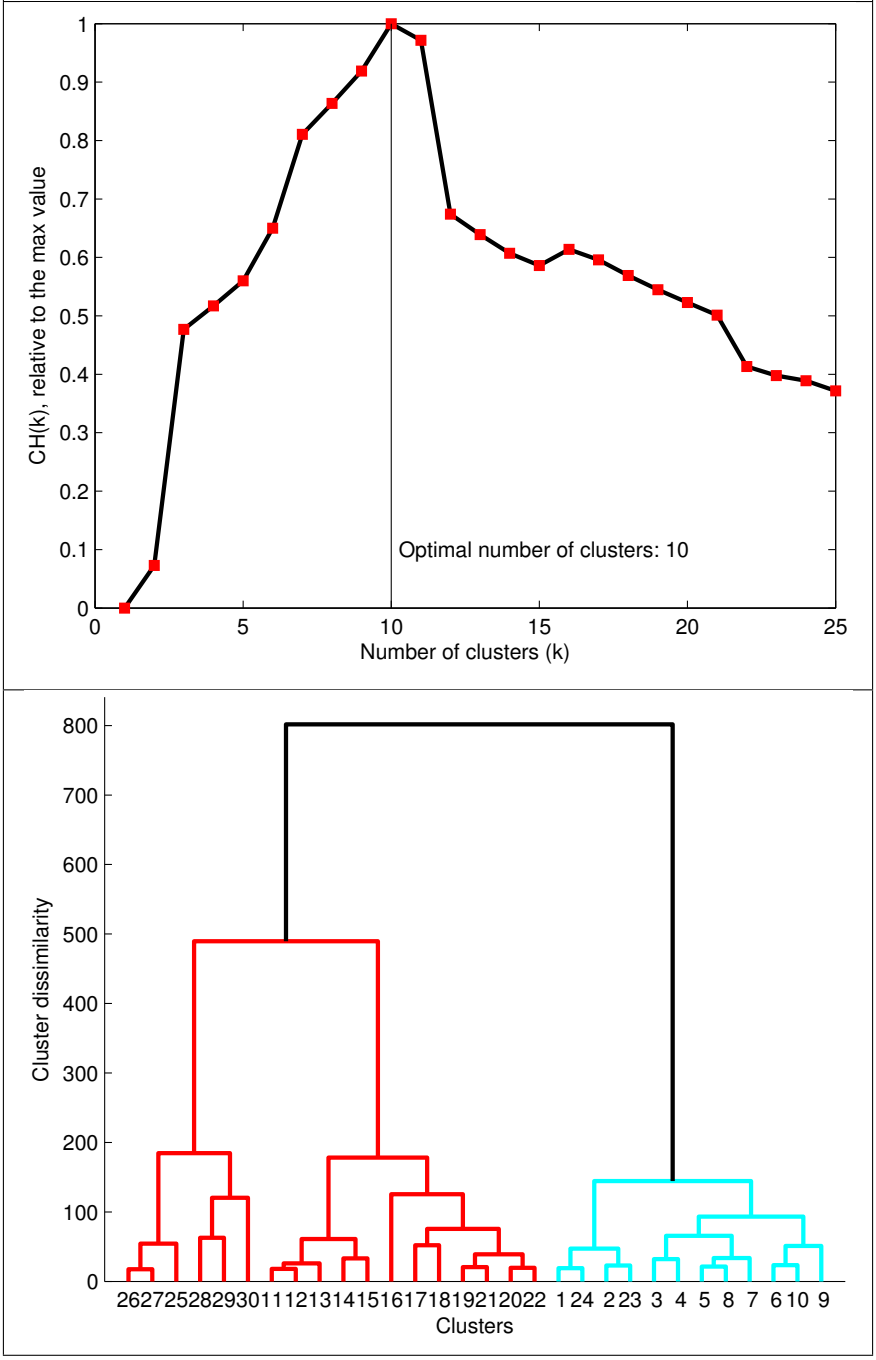
Plots for volunteer v_{15}, ward linkage (1/2)



Plots for volunteer v_{15}, ward linkage (2/2)



Plots for volunteer v_{16}, ward linkage (1/2)



Plots for volunteer v_{16}, ward linkage (2/2)