Tweets on a tree: Index-based clustering of tweets

Mert Kemal Erpam[a,1]

[a]*Sabanci University, Istanbul*

Abstract

Computer-mediated communication, CMC, is a type of communication that occurs through use of two or more electronic devices. With the advancement of technology, CMC has started to become a more preferred type of communication between humans. Through computer-mediated technologies, news portals, search engines and social media platforms such as Facebook, Twitter, Reddit and many other platforms are created. In social media platforms, a user can post and discuss his/her own opinion and also read and share other users' opinions. This generates a significant amount of data which, if filtered and analyzed, can give researchers important insights about public opinion and culture.

Twitter is a social networking service founded in 2006 and became widespread throughout the world in a very short time frame. The service has more than 310 million monthly active users and throughout these users more than 500 million tweets are generated daily as of 2016. Due the volume, velocity and variety of Twitter data, it cannot be analyzed by using conventional methods. A clustering or sampling method is necessary to reduce the amount of data for analysis.

To cluster documents, in a very broad sense two similarity measures can be used: Lexical similarity and semantic similarity. Lexical similarity looks for syntactic similarity between documents. It is usually computationally light to compute lexical similarity, however for clustering purposes it may not be very accurate as it disregards the semantic value of words. On the other hand, semantic similarity looks for semantic value and relations between words to calculate the similarity and while it is generally more accurate than lexical similarity, it is computationally difficult to calculate semantic similarity.

In our work we aim to create computationally light and accurate clustering of short documents which have the characteristics of big data. We propose a hybrid approach of clustering where lexical and semantic similarity is combined together. In our approach, we use string similarity to create clusters and semantic vector representations of words to interactively merge clusters.

Keywords: clustering, twitter, summarization, suffix tree, semantic relatedness, data mining

[☆]This technical report is based on the thesis "Tweets on a tree: Index-based clustering of tweets".

Email address: merterpam@sabanciuniv.edu (Mert Kemal Erpam)

1. Introduction

Data mining is the subject of extracting useful information from a set of data by using filtering, clustering and classification methods. In recent years, big data analysis has become a popular research area in the field of data analysis. A set of data can be classified as big data if it has enormous volume, a continuous influx of data and contains varied content. Due to its volume and velocity, big data cannot be analyzed by using conventional data processing methods as these methods usually have high time complexities.

Twitter is a social networking service founded with the intention of sharing news and opinions of people across the world in a summarized way. A registered user can send a text message with a maximum length of 140 characters and this message is called a tweet. Depending on the user's preferences, a tweet can be read globally by anyone or by users which are approved by sender to read his/her tweets.

Twitter has more than 310 million monthly active users [1] and generates 500 million daily tweets [2]. By looking at the volume of tweets and daily influx, Twitter is one of the big data sources. The data Twitter contains varies greatly from news, political debates, popular culture to daily conversations, personal complaints, spam and advertising messages. Although the voluminous and varied data can contain significant insights about society which may be very beneficial to the research of social scientists and other related fields, the same characteristics make it impossible to accurately analyze the data manually or even automatically with conventional data processing methods. The reason for this situation is that; aside from the volume, the velocity that tweets come is simply too fast and conventional data processing methods do not scale well with this volume and velocity.

The data Twitter contains is user-generated and has varied content. The data contains distinct tweets, but it can also contain duplicate tweets which do not contribute much to analysis. One way of eliminating duplicate, similar tweets and reducing the number of tweets necessary for analysis is clustering. Clustering is the task of grouping a set of data together based on a similarity metric. By clustering tweets, it is possible to obtain a more refined and distinct data and reduce the volume which in return reduces the time consumed by data processing methods. However, not all document clustering algorithms can be applied to tweets. Tweets have two distinct characteristics which differ them from documents. First, they are very short and many document clustering algorithms which use word-based similarity metrics will not work well with tweets since a tweet has a very small number of words. Secondly, Twitter has no writing format, users can use informal language, emoticons and abbreviations in their tweets and it makes semantic-based similarity metrics behave poorly as they cannot find the word and therefore the semantic. In addition to that, some document clustering algorithms cannot be used on Twitter data due to having high computational complexity.

In our work, we propose a new tweet clustering algorithm which takes note of the characteristics of Twitter data and uses them to obtain efficient and accurate clusters. Our algorithm has two steps: In the first step we use lexical clustering based on string similarity to cluster duplicate and similar tweets. The clustering technique is based on generalized suffix trees and has a low time and space complexity. The first step eliminates excess data

and creates representatives for clusters which we use in the second step with the combination of word embedding to determine semantic relatedness between clusters. The second step has a high time complexity; however, it can capture relations missed in the first step. Due to informal language and abbreviations, semantic relatedness may not be accurate, therefore we also present an interactive system based on semantic relatedness for users to improve the clustering.

1.1. Motivation

Due to its enormous volume, it is difficult to obtain any useful information from Twitter data without any processing or analysis. However, data processing algorithms, more specifically clustering algorithms, generally have high time complexities which do not scale well with big data.

Twitter is a hot topic in data analysis communities; the research in Twitter is mostly focused on classification and topic detection. These fields have numerous publications, however there is little research on summarization and representation of data on Twitter [3] [4] [5]. The publications about tweet clustering either miss semantic relations between tweets or are not suitable for big data.

Despite the limited attention, summarization and representation of data is a topic equally important to classification and topic detection as they provide significant insights about distribution and significance of topics. Another implicit advantage of a data summarization algorithm is its ability to act as a preprocessing step for more complex data analysis algorithms. Thanks to the reduction of data, data summarization algorithms allow complex data analysis algorithms to run faster.

The motivation of this work is to develop a clustering methodology for short documents which is able to create summarization and representation of Twitter data in a fast and efficient manner. For this purpose, we propose a lexical clustering approach based on suffix trees and complement it with word embedding, a semantic relatedness approach. The lexical clustering part of our algorithm has linear time and space complexity and is able to create clusters with representative labels, while the semantic relatedness part of the algorithms merges clusters which are related but are not caught by lexical clustering.

1.2. Contribution

In this work, we provide a new hybrid algorithm for clustering tweets. The clustering algorithm leads to reduction of data by creating clusters and representative labels for each cluster. This reduction gives a summarization and general overview of data. Using this overview and cluster distribution, topics mentioned in the data and the popularity of these topics can be inferred. Another advantage of this reduction is that it can be used for preprocessing step for more complex data analysis algorithms.

In the literature, the corporation of suffix trees into Twitter is a rarely studied topic. To the best our knowledge, there are only two publications which incorporate suffix trees into Twitter [6] [7]. Authors in [7] uses a different approach compared to our work and focus on hot topic detection by using temporal and regional features, while authors in [6] propose an adaptation of Suffix Tree Clustering algorithm [8] in Thai language. However, the direct

adaptations of STC such as [6] is not suitable for English language as tweets are not suitable for clustering word-by-word. In this work, our main contribution is to provide an adaptation of a character-based suffix tree algorithm with a new heuristic function for optimization and merging of clusters for Twitter domain. Our suffix tree clustering algorithm provides the most common phrase for each cluster and these phrases generally consist of short and have few words. We also use recently developed methodologies used in deep learning which are called word embedding to improve the clustering results. Basic word embedding methods adapt well to short phrases and this makes word embedding compatible with our suffix tree clustering algorithm. Therefore, our work uses the benefits of semantics provided by word embedding as well as syntactics provided by suffix trees.

2. Preliminaries and Background Information

In this section, we formally introduce required background knowledge which we use to solve the problem of tweet clustering. Firstly, we introduce the concept of big data and its characteristics. Secondly, we introduce suffix trees as we make use of this data structure to create a lexical clustering method with a linear time and space complexity. We discuss the construction, space and time-complexity of suffix trees and introduce Ukkonen's Algorithm, a suffix tree construction algorithm for constant-sized alphabets. Then, we introduce word embedding which we use to find semantic relatedness between two clusters. Lastly, we introduce the problem of Longest Common Subsequence which we use for evaluation purposes.

2.1. Big Data

With the advance of technology, the ability to collect data from various sources has increased tremendously. The size of the collected data has started to pass beyond the capabilities of conventional data processing algorithms and a need for a new field where suitable methods for data with huge volume and velocity is studied has arisen. The term big data is pronounced firstly in 1998 [9] and the term became coined quickly for the research of processing methods which handles large amount of data.

Big data generally represents a large volume of data with real-time flow and variety. For this reason, the characteristics of big data is represented as 3Vs: Volume, velocity and variety. The data of many social media platforms such as Facebook, Twitter are examples of big data.

Due to its characteristics, the processing methods for big data require to have low time complexities. The algorithms for big data can be classified in three sections: One-pass algorithms, sampling algorithms and distributed clustering algorithms [10]. Our algorithm, having linear time and space complexity, is a sub-member of one-pass algorithms

One-pass algorithms read input only once and process it immediately. They have $O(n)$ time complexity and generally require $O(1)$ space. For example, CURE [11] is a one-pass hierarchical clustering algorithm which uses random sampling for large databases.

Sampling algorithms use statistical methods to retrieve and shrink the data. The purpose of sampling algorithms is to summarize the data by selecting a subset of points from the data set. An analysis of sampling algorithms in Twitter can be found in [12].

Distributed clustering algorithms use distributed systems and parallelization for computation. The high complexity of clustering algorithms is compensated by high processing power.

2.2. Suffix Tree

Suffix tree is a data structure which represents the suffixes of a given string. Suffix trees provide linear time and space complexity for many string operations such as pattern and regular expression matching, longest common, repeated and palindromic substring finding. They are also used in the field of biology which requires pattern searching in sequences [13] [14].

A suffix tree, as the name indicates, is a tree data structure. It has a root and each node is connected to the root with a unique path using edges. Given a suffix tree constructed by string S, each edge contains a non-empty substring of S. Given that $|S| = n$, the suffix tree has n leaves and each path from root to a leaf represents a suffix of S.

Description of suffix trees can be made best alongside with suffix tries. A suffix trie is also a tree data structure which stores all suffixes of a given string. A suffix trie and a suffix tree has similar structures, however in comparison to a suffix tree, the edge of a suffix trie contains a character of the string S, while an edge of a suffix tree can contain multiple characters. A suffix tree is a compressed version of a suffix trie where only root, leaves and internal nodes of the suffix trie which have branching (nodes which have more than one child) are displayed.

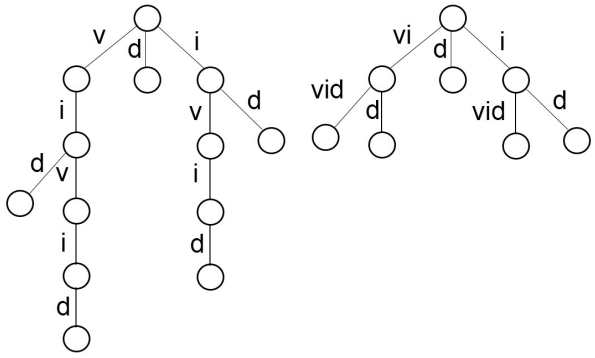


Figure 1: A suffix trie and tree representation using string "vivid"

Figure 1 gives an illustration of a suffix trie and suffix tree with the same string *vivid*. In both data structures paths from root to leaves form the suffixes of *vivid*. The branching nodes in suffix trie are nodes which represent the prefixes *vi* and *i*. Suffix tree also has these nodes, however other internal nodes of the suffix trie, the nodes with no branches, are merged with the branching nodes or leaves of the suffix tree.

Other than the root and the leaves, each node on a suffix tree needs to have at least two children. In a suffix tree there are a total of n leaves for $|S| = n$. This means there can be at most $n - 1$ internal nodes and a suffix tree can have a maximum of $2n$ nodes. Therefore, suffix trees have linear space complexity.

The concept of suffix trees is firstly introduced in 1973 by Weiner [15]. Weiner proposed a tree structure with linear-time and linear-space complexity for pattern matching. Later on, various linear-time algorithms are proposed for suffix trees, [16], [17] and [18] being some of the more important and prominent algorithms. McCreight's algorithm [16] provides a simplified construction algorithm, while Ukkonen's algorithm [17] has an on-line property and Farach's algorithm [18] is the first suffix tree algorithm which is linear time for integer alphabets.

In our work we use generalized suffix trees, a variety of suffix trees which accepts a set of strings. Our dataset contains Twitter messages. We use Twitter messages which have a fixed-sized alphabet and length. We use Ukkonen's algorithm, because the algorithm can be easily modified to construct generalized suffix trees and is optimal for fixed-sized alphabets.

2.2.1. Ukkonen's Algorithm

Ukkonen's Algorithm is a left-to-right suffix tree construction algorithm. The algorithm reads the input from left-to-right and inserts the characters once at a time. In Ukkonen's algorithm, the suffix tree is constructed by adding on top of the suffix tree of prefixes: Let $S = t_1t_2...t_n$ and $S_i = t_1t_2...t_i$ where $0 \leq i \leq n$ and S_i be a prefix of S. In the initial state the algorithm has the suffix tree of S_0 where only root exists. At each step, the algorithm inserts t_j to the suffix tree of S_{j-1} to construct the suffix tree of S_j. The algorithm finishes when symbol t_n is inserted to the suffix tree of S_{n-1} and when suffix tree of S_n, in other words S, is constructed.

To describe Ukkonen's algorithm, it is first necessary to introduce some notions and explain the terminology in the paper. We will stick to terminology described in the original paper [17], which explains the construction of suffix tree using a state-machine, to avoid any confusion with other sources.

A suffix tree is a compressed suffix trie. All of the nodes in a suffix trie can also be found in a suffix tree. The nodes in suffix trie, called states in original paper, can be present in the suffix tree implicitly or explicitly. Root, leaves and internal nodes with more than one child are presented explicitly in the suffix tree and these are called explicit states. Other internal nodes are not present in the suffix tree, but they can be reached by using explicit states and the information provided by edges. These nodes are called implicit states. An edge or edges that are necessary to go from one state to another are called transitions. Boundary path of a suffix tree is defined as the set of states which represent suffixes of the current tree such as $s_1 = t_1t_2...t_i$, $s_2 = t_2t_3...t_i$, s_3 ... $s_{i+1} = root$ for the suffix tree of S_i.

In the light of given terminology, we define two functions: Transition function g and suffix function f. Given state x, the transition function g is defined as $g(x, a) = y$ for all $y = xa$ and the suffix function f is defined as $f(x) = y$ for all $x = ay$ where y and x are states and a is a sequence of symbols of the alphabet.

Ukkonen's algorithm is a left-to-right algorithm, it creates the suffix tree of S_j by adding a new symbol to S_{j-1}. A naive approach is to find all states in the boundary path and update them. However, Ukkonen's algorithm defines two special states in the boundary path and uses them for update: The smallest index in the boundary path which has a transition is called active point and the smallest index in the boundary path which has the necessary

transition for the newly added symbol is called end point.

The states until the active point are leaves in the suffix tree. Because leaves always represent the suffixes of the string, the algorithm makes the transition to leaves an open transition: A transition which grows automatically as a new symbol is added. This can be done because Ukkonen's algorithm uses pointers to represent strings and it gives the option to assign the end pointer to an open value.

The states between active point and end point are the states in which a transition needs to be added. These states can be explicit or implicit states. If a state is explicit, then a new transition for the added symbol is created and a new state connected to the transition is created. This newly created state is a leaf; therefore, suffix links are updated accordingly. If a state is implicit, then it is first made explicit by creating a new state and then the same steps are applied.

The states after end point already have a transition for the new symbol, therefore there is no need for an action. These states are taken care of when a new symbol with no transition is added and the end point is pushed back. To handle these states correctly, a symbol outside of the alphabet is usually appended at the end of the input string.

The algorithm uses active points and suffix links to achieve linear time complexity. Active points are represented by a reference pair which consists of an explicit state and the string of the transition which leads to active point. To avoid unnecessary traversals and achieve liner time, active points are updated using a method called canonize which makes the explicit state in the reference pair as closest as possible to the active point.

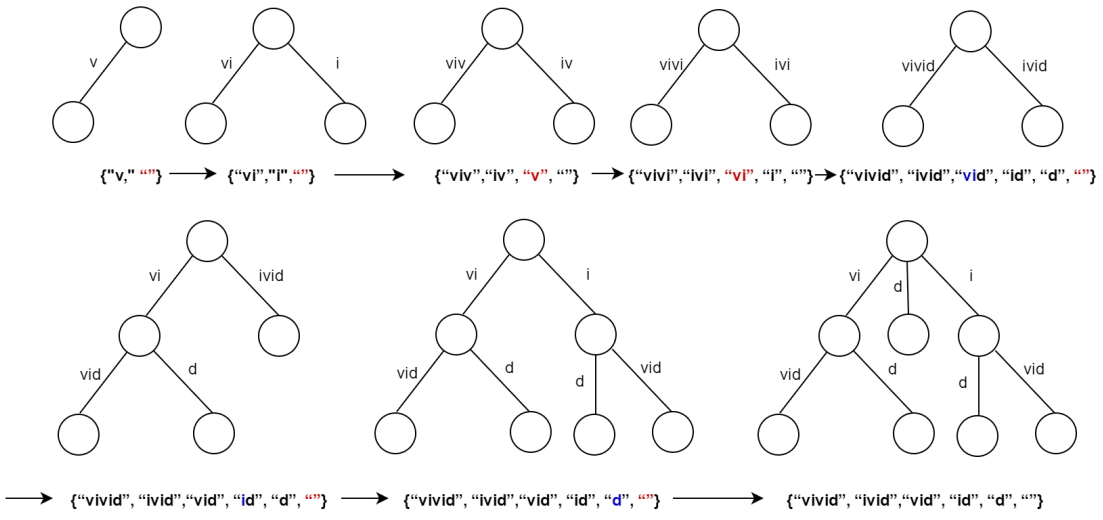


Figure 2: Suffix tree construction using Ukkonen's Algorithm for string "vivid"

Figure 2 gives an illustration of Ukkonen's algorithm using string "vivid". In the boundary path, end points are represented by red and active points are represented by blue whenever they do not overlap with end points. The pseudo-code of Ukkonen's algorithm can be found in Appendix Appendix A.

2.2.2. Generalized Suffix Tree

Generalized suffix tree is a suffix tree which is able to process multiple documents. In general suffix trees, a unique and identifying symbol outside of the alphabet is devoted to each document and these symbols are used as end symbols for the documents. In our work we use an open-source generalized suffix tree implementation based on Ukkonen's algorithm [19]. In this implementation, each document has a unique id. Because a path from root to a leaf represents a suffix, each leaf stores a set of ids. These ids belong to the documents which contain the suffix that the path represents.

2.3. Word Embeddings

Word embedding is a technique used in natural language processing where words or phrases are represented as a vector of real numbers. The technique of using vector representation of words has been used since 2000s [20], however it has gained popularity 10 years later, when word2vec [21], a toolkit for training and using word embedding is created.

Word embedding is an unsupervised learning technique over a large corpus of text. The main idea is to create similar vector representations for words which are in a similar context. Before 2010, it was achieved by using neural network architectures which had an expensive computational cost of training over a large corpus. Later on, two new techniques which have lower computational costs compared to neural network architectures are proposed. In recent years, these two techniques became more predominant and popular in the field of word embedding.

The first proposed technique is word2vec method. In this method, the vector representation of each word is learned by looking at the neighbor words. The algorithm updates the vector representations of a word by looking at neighbor words in a window, making the vector representations of words within the window more similar and of words outside the window more distant. After many iterations, words with similar context start to have similar vectors.

The second proposed method is Glove method [22]. The end result of Glove is similar to word2vec in the sense that both create similar vector representations of words with similar context. However, instead of learning, Glove uses dimensionality reduction to create vector representations. It creates a co-occurrence matrix of word counts in each window and selects features which best represent these co-occurrences in a lower dimension.

Word embedding represents the words in a lower dimension and also retain the semantic relatedness between words. The arithmetic operations between vector representations give semantic information about words. A famous example is that the arithmetic operation **king - man + woman** gives a vector which is very similar to the vector representation of **queen**.

2.4. Longest Common Subsequence Problem

The longest common subsequence (LCS) is a long studied computer science problem. Given a set of sequences, LCS is the longest sequence which exists in all sequences. It is used as a similarity measure[23][24] and is used to determine and display differences between documents.

Finding LCS of arbitrary number of sequences is a NP-hard problem. However, LCS of two sequences can be found in $O(m * n)$ time and space using dynamic programming where m and n are the length of the sequences.

In our work we use LCS for evaluation purposes. Our work uses string similarity to cluster the dataset and sequence similarity is an important criterion for determining the quality of clusters for us. For this purpose, we design a similarity function between two documents which uses the length of LCS.

2.4.1. Computing the length of LCS for two sequences

The problem of finding the length of LCS can be divided in overlapping subproblems: Let X and Y be two strings and $X_i = x_1x_2...x_i$ be the prefix of X until ith character and $Y_j = y_1y_2...y_j$ be the prefix of Y until jth character, then the length of the LCS for X and Y can be found with the following function:

$$LCS(X_i, Y_j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) + 1 & \text{if } x_i = j_i \\ max(LCS(X_{i-1}, Y_j), LCS(X_i, Y_{j-1})) & \text{if } x_i \neq j_i \end{cases}$$

In order to find LCS for X and Y, it is necessary to compute LCS of the prefixes of X and Y which breaks the original problem down to subproblems. From the LCS function it can be seen evidently that the solution of subproblems is used more than once. All of these properties makes the LCS problem a perfect candidate for dynamic programming approach:

Algorithm 1 Longest Common Subsequence Length Algorithm (X[1..m], Y[1..n])

```
1:  C ← array(0..m, 0..n)
2:  for i := 0 to m do
3:      C[i, 0] ← 0
4:  for j := 0 to n do
5:      C[0, j] ← 0
6:  for i := 1 to m do
7:      for j := 1 to n do
8:          if X[i] = Y[j] then
9:              C[i, j] := C[i − 1, j − 1] + 1
10:         else
11:             C[i, j] = max(C[i, j − 1], C[i − 1, j])
```

Algorithm 1 computes the LCS for prefixes of X and Y and then stores the solution in a table which can access later on when computing the LCS for (other prefixes of) X and Y. The complexity of the algorithm is $O(m \times n)$ in both space and time domain which makes it impractical while working on big data.

3. Related Work

Twitter is a social networking service which became the focus of many researchers since its launch. With its launch, Twitter provided a new form of communication by microblogging, gaining popularity quickly. Initially, the rising popularity and the new form of communication caught the attention of researchers, leading them to make publications about the definition of Twitter in social media and its use cases [25] [26]. However, the real focus of research circles in Twitter has quickly become the data Twitter possesses. With the rise of its popularity, Twitter data contains a variety of information, making it a good source for data and opinion mining. The research on Twitter has reached excessive amount and even though we are unable to show all of them, we make an extensive literature review on topics that are related to our work by showing prominent and state-of-the-art works.

The aim of our work is to create a representation of Twitter data by clustering. Our work is directly connected to clustering, but it also has indirect connections to event detection, classification and spam detection algorithms. With statistical analysis, results of our algorithm can be used for event detection. In classification algorithms, tweets are classified depending on their polarity or pre-set labels and although our algorithm cannot make classification, the intra-cluster purity of clusters our algorithm generates is high in respect to pre-set labels. Our algorithm is also able to create high purity spam clusters as a side-effect.

Event detection is the task of detecting and at times predicting events or topics using Twitter. The research on this subject [27] [28] [29] [30] [31] [32] [33], differ from each other in terms of the topics and methods they choose. As an example, the authors in [28], [29] and [32] use topology between users for detection of disasters and political predisposition in community, while the authors in [30] [31] use cluster-based methods to find event-based information. On the other hand, the publication [33] is aimed to detect communities and trend topics in Twitter.

Spam detection is the common research area amongst many social media platforms such as Twitter, Reddit, Facebook and message exchange services such as forums and e-mails. With its enormous volume, Twitter also contains messages from automated agents, whose purpose is to advertise, promote or manage perception about specific topics. In academics, spam detection in Twitter is generally seen as a binary classification problem where a tweet is either a spam or not a spam and the spam detection algorithms originate generally from classification methods [34] [35] [36] [37] [38]. Spam detection aims to either identify automated users by using features such as user creation date, username selection and posting patterns or it aims to detect spam tweet by alienating spams based on content [39]. Our work is able to collect most spam messages by using string similarity as an unintended consequence, however it is not comparable to the state-of-the-art spam detection algorithms.

Classification is a very active topic in Twitter. Twitter has no classification system which means that users do not select under which category their tweet falls to. The class labels and classification tasks are defined by researches. A popular classification approach is sentiment classification where a tweet is labelled as positive, neutral or negative. Sentiment classification is a popular research topic in Twitter, there has been a lot of research on it; [40], [41], [42], [43] and [44] being more known works. In addition to sentiment classification,

there also works which pre-define labels and make classification accordingly [45].

Clustering is a task of grouping similar documents in a document set. Traditional clustering algorithms such as K-means, DBScan or hierarchical clustering do not work very well in large datasets which contain undetermined number of clusters such as Twitter. However, in Twitter, users may talk about similar topics, give similar responses or retweet each other, making Twitter data a good platform for clustering. Because of that, clustering algorithms specialized for Twitter are developed. Like classification, clustering is also a popular research area in Twitter. Twitter clustering algorithms can be categorized in many aspects such as methodology, complexity or cluster definition. In our work, we differentiate clustering algorithms by using their similarity metrics and roughly divide in two parts: Lexical similarity and semantic similarity.

Although there are slight variations on the scope of the definition of lexical similarity, we define lexical similarity as a degree which measures how syntactically similar the words between two documents are. Lexical similarity between documents in Twitter is usually examined by using Named Entity Recognition (NER) [46] [47] [48]. There are other clustering approaches which use genetic algorithms [49] or word occurrences as a similarity measure [6]. The purpose of lexical similarity is to find similarities between documents without using the semantic context of words.

Semantic similarity is a measure of degree which calculates the similarity of documents by looking at the context of each word and their contextual relations to each other. In semantic similarity, words which can be interchanged with each other are close to each other. Semantic similarity is usually used in short-document clustering along with lexical similarity [50] [51]. Semantic similarity is also used in Twitter in conjunction with lexical similarity at both classification [52] [53] and clustering tasks [7].

For tweet clustering, NER-based approaches require training in a domain to correctly recognize entities, however because of informality and abbreviations, general NER models have problems recognizing tweets using informal language. They require domain-based semi-supervised training in Twitter, however Twitter is a changing and evolving platform where everyday different topics are discussed, making domain-based NER models substandard.

Calculation of semantic similarity between documents is usually a non-linear operation, which makes semantic clustering unsuitable for large amounts of data. In Twitter, semantic similarity and semantic relatedness is usually used in conjunction with other processing methods. In the field of clustering and topic retrieval, [7] is one of the hybrid approaches in Twitter domain. [7] also uses suffix tree as a basis for clustering. The difference between our work is that they are interested in the first k popular topics generated in Twitter and use other features such as temporal and tag data to enhance their clustering, while we are interested in all clusters in data as we aim to obtain a summarized representation of data and use semantic relatedness to enhance our clustering.

A part of our algorithm, lexical clustering, is based on suffix trees. There has been a lot of research on suffix trees and clustering, Zamir's Suffix Tree Cluster algorithm [8] being the center of them. Zamir's STC algorithm is a linear time clustering algorithm which is based on phrases that are common in a group of documents. There are many variations of STC algorithm [54] [55] [56] [57], including semantic variations [58] [59]. These clustering

algorithms are generally used for clustering web documents on search engines. They do not work very well with Twitter data, because they use phrase similarity for clustering and Twitter data is too short for word-by-word clustering analysis. There is currently only one study in Twitter using suffix trees and it uses Zamir's algorithm as basis and employs a merging algorithm for Thai tweets [6].

One of our main contributions in this work is our adaptation of suffix tree clustering for Twitter, therefore it is significant to stress out the differences between state-of-the-art suffix tree algorithms and ours. Current suffix tree clustering algorithms use Zamir's STC and constructs the suffix tree using words as the smallest unit, treats nodes as a cluster, rank the nodes, take the top k of nodes and merge with other nodes to obtain top clusters. The constraint of retrieving k clusters is to keep the algorithm in linear time. In our algorithm, we aim to retrieve every cluster which suffix tree can generate. We can do it in linear time thanks to the property of our document-set having fixed size. Although the base of our algorithms show similarity because both algorithms see nodes as cluster representatives, we construct our suffix tree using characters to capture word variations and employ specific heuristics for character-based analysis. Due to this, our constraints for cluster membership and our overlapping algorithms differ compared to Zamir's algorithm, creating a new suffix tree clustering algorithm specifically designed for Twitter.

With its rising popularity, word embeddings is also a method which is used in Twitter tasks. In Twitter, word embeddings are generally used for classification tasks which focuses on sentiment classification such as [60], [61] and also other classification tasks like [62], [63]. Among the research which uses word embeddings, [64] has the most similar approach to our work, as it uses a hybrid approach using tf-idf and word embeddings. [64] is evaluated with Wikipedia and Twitter data. It performs well on Wikipedia, however the error rate on Twitter is very high due to insufficient number of words in each tweet necessary for tf-idf.

4. Methodology and Problem Definition

In this section we explain and discuss our method for tweet clustering. Our algorithm is a hybrid approach which clusters large data sets of tweets using string similarity and merges them using semantic relatedness. We divide the section in three subsections: First, we make a formal problem definition for our work. Then, we explain lexical clustering and lastly, we talk about interactive merging part of our algorithm which is based on semantic relatedness. We discuss optimizations, space and time complexity of each part in their respective subsections.

4.1. Preliminaries and Problem Definition

It is impossible to manually analyze Twitter data due to its enormous volume and velocity. Twitter contains significant amount of information about multitude of topics. In order to reveal these insights, there is a need for reduction and summarization of data for representation and further processing. Clustering is a suitable method for this task.

In this work, we are interested in the textual representations of tweets. We define a tweet, t, as a sequence of characters that a user sends where $|t|$ is the length of the sequence

of the characters. Given two tweets t_i and t_j, we define common substrings of t_i and t_j as a string which occurs in both tweets. We use common substrings in our similarity calculations for cluster creation. We find common substring of tweets with the help of a suffix tree. The path from root to each node in a suffix tree represents a unique substring and we define this substring as the pathString of a node where $|pathString|$ is the length of the substring.

Formally, given a set of tweets $T = \{t_1, t_2, ..., t_n\}$, we would like to create a set of clusters $C = \{c_1, c_2, ..., c_k\}$ such that the length of the common substring of tweets inside a cluster is above a threshold. We define the common substring of all tweets in the cluster as the cluster label and this label represents the cluster. With the condition that $k << n$, we aim to obtain a summarization of data which allows us to gain insights easier about Twitter data.

4.2. Lexical clustering

In lexical clustering, we use a string-based similarity to cluster duplicate or similar entries. We propose an algorithm with linear space and time complexity which determines the similarity based on common substrings between tweets. In our algorithm, initially we do preprocessing to reduce the noise in tweets. Then, we create a generalized suffix tree and create clusters using the nodes of the suffix tree. We eliminate clusters with high correlations and eliminate overlapping. At the end, we create representative labels for clusters and finish lexical clustering.

4.2.1. Preprocessing

Twitter data is a data generated by users all around the world without any specifications. The structure of the data varies greatly and is depended on user: It could be written in perfect English, all in lower cases or upper cases, different punctuation marks or different spacings could be used in tweets. We use string similarity for lexical clustering and these variations may cause the algorithm to fail to recognize similar tweets with different variations; therefore, there is a need to standardize the Twitter data as much as possible to obtain best results.

Tweets often contain many words which have no clear semantic context. Links, usernames, retweet tags and hashtags are some of the examples of such words. Our method relies on string clustering and these words, when present in large quantities, may skew the clustering to center around them as tweets have a short length. These words are removed during the preprocessing phase. In addition to them, words with high document frequencies in the data set we are clustering are also removed from documents, because they skew the clustering to from big clusters, overshadowing more distinct and meaningful clusters.

The preprocessing phase for Twitter data has the following steps:

- We find the document frequency of each word and remove words which are above a certain threshold from tweets
- We remove usernames, hashtags, retweet tags, punctuation marks and links from tweets as they have no semantic value and influence cluster selection process negatively.

- We transform all tweets to lower case and adjust white spaces
- We remove tweets which have less than 5 characters after removing words with no semantic

4.2.2. Suffix tree Construction

After preprocessing, the next step is to create a structure which represents all common substrings between tweets. For this purpose, we use a generalized suffix tree. Each node in the suffix tree represents a substring of a tweet or multiple tweets and each leaf contains the ids of tweets which contain the string the leaf represents. The set of tweet ids of each node can be found by aggregating the tweet ids from descendant leaves.

For efficient clustering, we need to store two more variables in suffix tree nodes. Firstly, we need to use the string each node represents, however node strings are not stored in nodes for space efficiency. In order find a node string, we can try every possible path from root to every node until the node we are looking for is found, or we can traverse in a bottom-up manner starting from node until root. The first option is costly, while second one is not possible in a one-directional suffix tree. Therefore, during construction process, we add a link from each node to its parent and make the suffix-tree bi-directional. Secondly, we will also need to access the length of node strings frequently. It is not efficient to recreate node string each time the length is required, therefore we create a variable to store the length of the string and calculate it during the suffix tree construction phase. The calculation is trivial, as a node string is a combination of the parent node string and the string of the edge which connects the parent to the node.

4.2.3. Cluster Creation

We define a tweet to be similar to another tweet, if the ratio of their common substring to their length is over a certain threshold. Each node in the suffix tree represents a string and a set of tweets contains this string which makes it a common substring between the tweets in the set. Therefore, each node is actually a candidate for a cluster. Given the node n, let the cluster created by n be c_n, then a tweet is a member of the cluster c_n if:

- Tweet contains the pathString of node n
- |n.pathString| / |tweet| > **thrCluster**, where **thrCluster** is a user-defined threshold.

We create clusters by traversing each node, finding their id sets and checking their ratios against **thrCluster**.

4.2.4. Overlapping Cluster Elimination and Merging

In the suffix tree, the id of a tweet can exist in multiple nodes. This situation implies that a tweet can exist in multiple clusters. In clustering, if an object belongs to multiple clusters, it is called overlapping. The clusters we obtain after last step are overlapping clusters.

Overlapping clusters are not inherently bad, however clusters created by nodes of suffix tree have high correlation and this causes many clusters with similar contents. In order to reduce the amount of clusters and achieve data compression, we eliminate overlapping.

To remove overlapping, we initially sort clusters by their size. We flag the tweets starting from the biggest clusters. Then, we proceed to the smaller clusters and if a tweet is flagged, then we remove the tweet from the smaller cluster.

During the overlapping elimination process, there are two special cases. Our observations show that if the majority of tweets in a cluster is flagged and removed by a single cluster, then the remaining tweets in the first cluster is most likely similar to the tweets in the second cluster and can be merged into the second cluster. For this reason, if a cluster contains more than 80% of the tweets of the other cluster, we merge these two clusters together.

The second case is more elementary. If a cluster is not merged with another cluster and has only one tweet as an element, then this cluster is removed, since a cluster with only one element in not a cluster anymore.

Algorithm 2 Overlapping Elimination and Merging (clusters[0 ... m]))

Require: *clusters* is the list of clusters and are sorted based on their size
Require: n is the total number of tweet ids

```
1:  flagMask ← array(0...n)
2:  for i := 0 to n do
3:      flagMask[i] ← −1
4:  for i := 0 to m do
5:      indexMap ← array(0...n)
6:      clusterSize ← |clusters[i]|
7:      for index in clusters[i] do
8:          if flagMask[index] = −1 then
9:              flagMask[index] ← i
10:         else
11:             cluster[i] ← cluster[i] \ index
12:             cIndex ← flagMask[index]
13:             indexMap[cIndex] ← indexMap[cIndex] + 1
14:     (cIndex, count) ← Retrieve the index with the most occurrence from indexMap
15:     if count > clusterSize ∗ 0.8 then
16:         clusters[cIndex] ← clusters[cIndex] ∪ clusters[i]
17:     else if |cluster| < 2 then
18:         clusters ← clusters \ clusters[i]
19:         Mark indices left in cluster as -1 in flagMask
```

4.2.5. Cluster Labeling

At the end of lexical clustering process, we create representatives for clusters. Originally, clusters do not have labels, however our clustering method groups tweets with a common substring together, making the substring a perfect representative for cluster. We use bi-directional suffix tree to traverse from the node of the cluster to root and assign the node text as the label of cluster.

In the next phase of the clustering, we use word embeddings to find semantic relatedness

between clusters. For this purpose, we use cluster labels to represent clusters, however the start and end of the cluster labels may contain incomplete words and word embeddings may not recognize incomplete words. To complete incomplete words, we take a sample tweet from cluster which contains the label and complete the beginning and end of the label.

4.2.6. Complexity Analysis

Our algorithm is proposed with the intention of clustering tweet sets. Therefore, when analyzing the space and time complexity, we assume that the maximum length of documents which are being clustered is fixed and can have a maximum of 140 characters. Our complexity analysis is based on the total number of characters in the dataset.

We first start with the space complexity. Our algorithm is based on a generalized suffix tree. Suffix trees have linear space complexity. Given a string with length n, a suffix tree created by this string can at most have $2n$ nodes and $2n-1$ edges. In case of the generalized suffix tree, given a set of documents $S = \{s_1, s_2, ..., s_n\}$, the tree can have at most $2*\sum_{i=0}^{n} |s_i|$ nodes.

In comparison with suffix trees, generalized suffix tree stores the indices of it documents in its leaves. A document can have multiple suffixes and accordingly the index of a document can be in multiple leaves. This situation may create an overhead for the tree. In our input set, length of documents is limited to 140 characters; therefore, a document can have a maximum of 140 suffixes. This implies an index can be present at most in 140 leaves, making the overhead of storing indices at most $140n$, given that n is the size of the document set. Thus, the complexity of storing indices is linear.

Aside from generalized suffix tree, we also create clusters and store indices of documents inside clusters. Indices in clusters come from nodes and at worst-case, each node is going to be represented by one cluster and each index in a node will be stored in a cluster. Therefore, in order to compute the space complexity of clusters, we have to find the total number of indices of documents in the suffix tree. For this purpose, we will dissect the tree level by level: At the level of leaves, there can be at most $140n$ indices. As we go up, the overlapping of indices will increase and each level will contain less indices. At the top level, root, there are exactly n indices. The total number of indices in each level can be formulated as: $140n + \sum_{i=1}^{h-1} m_i n + n$, where $1 \leq m_1 \; ... \; m_{h-1} \leq 140$, m_i is the overlapping constant in level i and h is the height of the tree. The height of a suffix tree is determined by the longest length of a suffix and with the length constraint, the height can be at most 140. By relaxing our summation formula, we obtain: $\sum_{i=1}^{140} 140n + n = 19461n$ which is linear, albeit with a high coefficient.

We analyze the time complexity of the algorithm step by step. In the first phase, preprocessing, two iterations over the set of documents are made: In the first iteration we remove words with no semantic significance, make documents lower case, arrange white-spaces and create a term frequency list. In the second pass, terms with high frequencies are removed from tweets. Operations in both iterations are constant time operations and the preprocessing phase has linear time complexity.

In the second phase, generalized suffix tree is constructed based on Ukkonen's algorithm which has linear time complexity.

In cluster creation phase, we retrieve the index set of each node and make a threshold check for each document in the index set for cluster membership. The thresholding check is a constant time operation and the number of times the check is done is equal to the total number of indices in each node. As the total number of indices in each node has linear space complexity, the total number of checks is done in linear time. The retrieval of indices of a node is done by traversing from the node to its descendant leaves. At root, the retrieval is done by traversing all nodes, while at leaves the retrieval is done instantly. To calculate the complexity, we analyze the tree again level by level. Let k_i be the number of edges going from height i to $i-1$, then the recurrence function for retrieval of nodes at height h is:

$T(h) = k_h * T(h-1)$ and $T(1) = 1$

Telescoping the recurrence function, we obtain: $T(h) = \prod_{j=2}^{h} k_j$ which represents all of the nodes in the tree when h is equal to height of the tree.

We can calculate the number of nodes at each level by using edges. Let h be the height of the tree, then at height $h-1$ there are k_h nodes, at height $h-2$ there are $k_h * k_{h-1}$ nodes. Using edge information, we define a function to calculate the number of nodes at each level: $n(i) = \prod_{j=i+1}^{h} k_j$

Using $n(i)$ and $T(h)$, the complexity of retrieval of indexes of each node is:

$$
\begin{aligned}
R(n) &= \sum_{i=1}^{h} n(i) * T(i) \\
&= \sum_{i=1}^{h} \prod_{j=i+1}^{h} k_j * \prod_{j=2}^{i} k_j \\
&= \sum_{i=1}^{h} \prod_{j=2}^{h} k_j, \text{ where } \prod_{j=2}^{h} k_j \text{ is at-worst case 2n, which leads to:} \\
&= \sum_{i=1}^{h} 2n \\
&= 2hn \text{ and because } h \leq 140 \rightarrow R(n) \subset O(n)
\end{aligned}
$$

The overlapping elimination phase makes one iteration over clusters. In each iteration, indices inside a cluster is taken and checked whether it is flagged. The flagging operation is a constant time operation and the amount of times it takes is equal to total number of indices. As discussed before in space complexity analysis, at the worst-case, the total number of indices in the set of clusters $19461n$, which is linear.

Cluster labeling is an operation where the text a node represents is constructed from suffix tree. The operation is string-length based and because length is limited to 140, it is a constant-time operation. At the worst-case, the operation is done for each node, which makes this phase linear.

4.2.7. Optimizations

The complexity analysis subsection demonstrates us that with limited document-length, the algorithm has both linear space and time complexity. The linearity, however, comes with very high coefficients. This makes the algorithm slow and non-interactive in practice. We make two observations about the algorithm: Firstly, our experimental evaluations show that critical steps of the algorithm are the cluster creation and overlapping elimination phases which depend on the size of the processing nodes and secondly, we create many redundant clusters in the cluster creation phase which are removed during overlapping elimination phase. To reduce the redundancy, we use a heuristic to detect overlapping clusters and reduce the number of nodes which are processed during cluster creation and overlapping elimination phase, making the algorithm run faster. To achieve this, we offer a new phase, elimination of redundant nodes which operates after preprocessing phase and a couple of optimizations which lead to the decrease of the number processing nodes and make both cluster creation and overlapping elimination phase run faster.

4.2.8. Duplicate Elimination

Twitter data, even after preprocessing, is a data with many variations for string similarity detection. These variations come from users who post the same content with slight differences or users who retweet other users. The variations cause different substrings and all of these substrings are represented at a node in a suffix tree. From these nodes similar clusters are created and these clusters are eliminated and merged in the overlapping phase. In our algorithm, we define these nodes as duplicate nodes. Our observations show that, most of the duplicate nodes have similar patterns which can be used for early detection and elimination. Elimination of duplicate nodes leads to a faster clustering and overlapping process.

Duplicate nodes can be observed mostly between nodes with ancestry relations. An example would be the node which has the string "palin asks why muslims hate peanuts" and its parent node which has the string "palin asks why muslims hate pea". Both nodes target the tweets with similar content: the questioning of palin about muslims and peanuts. The branching from parent node occurs due to retweeting, the sibling node of the node has the string "palin asks why muslims hate pea..." which comes from a retweet and is the truncated version of the content due to character limitation.

At occurrences where branching occurs due to retweet or small variations, mostly one of the child nodes contains the majority of the tweets and other nodes represent the variations which are the minority. We use this pattern to determine duplicate nodes: We define a threshold **thrSize** and check a node and its ancestors. If there is a burst on the size of index sets between a node and its ancestor over the threshold, then they are likely not duplicates. If the burst is under the threshold, then they are labelled as duplicates. It has to be noted that on ancestors where tweets with two different contents are merged, the burst is usually large and differentiable.

In the duplication elimination phase, we traverse the suffix tree in a reverse breadth-first manner starting from leaves. We use prefix and suffix ancestry to label a node as duplicate. Using **thrSize**, we check the ancestors of the node until a satisfying burst is observed. When

we observe the burst and find the non-duplicate ancestor, we label all nodes between the node and that ancestor as duplicate nodes as the burst is not enough for these nodes. To determine whether the node is duplicate, we check string length between the node and its non-duplicate ancestor, if the ratio is below a threshold, then the node is also labeled as duplicate.

Each ancestor node contains the tweets contained in its descendants. Therefore, when a node is labeled as duplicate, as long as it has a non-duplicate ancestor, the tweets in the duplicate node can be represented in its ancestor. However, as we visit ancestors of a node, the length of the string of ancestor nodes decreases which may make the threshold check in cluster creation phase fail for tweets in the duplicate nodes. Therefore, we create new variable and store the maximum string length of the descendant node in the non-duplicate node and use this variable for threshold check.

Algorithm 3 Duplicate Elimination based on suffix ancestry(nodes[0 ... m]))

Require: *nodes* is the list of nodes in suffix tree traversed in breadth-first

```
1:  for i := m to 0 do
2:      suffix ← nodes[i].suffix
3:      indexThreshold ← nodes[i].indexSize * thrSize
4:      while suffix.indexSize < indexThreshold do
5:          suffix.suffixDuplicate = true
6:          suffix ← next suffix ancestor
7:      if nodes[i].suffixLength is not initialized then
8:          nodes[i].suffixLength ← |nodes[i]|
9:      suffix ← first suffix ancestor which is not suffixDuplicate
10:     stringThreshold ← nodes[i].suffixLength * thrString
11:     if |suffix.text| > stringThreshold then
12:         nodes[i].suffixDuplicate = true
13:         suffix.suffixLength ← max(nodes[i].suffixLength, suffix.suffixLength)
```

Algorithm 3 is the base algorithm for duplicate elimination based on suffix ancestry. The same algorithm with different variables is also used for prefix ancestry duplicate elimination.

Experimentation results show that a threshold value around 1.2 is a good choice for **thrSize** and a threshold value around 0.8 is a good choice for **thrString**.

4.2.9. Optimized Cluster Creation

With the addition of duplicate elimination phase, we update the process of selecting nodes for each cluster. Instead of looking at each node, we look at nodes which are not marked duplicate by their prefix or suffix ancestry. In addition to that, we also make small optimizations in this phase: If the index set of a node is completely added to a cluster, then the descendant nodes of this node does not need to be investigated, as the clusters created by these nodes will be eliminated in the overlapping elimination phase. Because of that, we traverse the suffix tree in a breadth-first manner, check for such nodes and mark their descendants.

Algorithm 4 Cluster Creation(nodes[0 ... m], tweets[0...n]))

Require: *nodes* is the list of nodes in suffix tree traversed in breadth-first
Require: *tweets* is the list of preprocessed tweets

```
clusters ← ()
for i := 0 to m do
    if parent or suffix is inACluster then
        nodes[i].inACluster ← true
    else
        if !(nodes[i].prefixDuplicate && nodes[i].suffixDuplicate) then
            cluster ← ()
            nodeLength = max(nodes[i].suffixLength, nodes[i].prefixLength)
            for index in nodes[i] do
                if nodeLength > |tweets[index]| * thrCluster then
                    cluster ← cluster ∪ index
            if |cluster| > 1 then
                clusters ← clusters ∪ cluster
                if |cluster| ≥ |nodes[i]|-1 then
                    nodes[i].inACluster ← true
```

4.3. Interactive Merging

After lexical clustering we obtain clusters of tweets which have similar contents, however there may be tweets which can convey the same content with different words. These tweets may be assigned in different clusters, because in lexical clustering, tweets are clustered based on their string similarity, disregarding their semantic meanings. If these clusters share similar contents, they have to be merged together and lexical clustering is not sufficient for this task. Therefore, we introduce a new step after lexical clustering where users can merge clusters based on semantic relatedness.

In interactive merging, we use semantic relatedness to determine clusters which can be possible candidates for merging. For this purpose, we use cluster labels. We calculate the pairwise semantic relatedness of clusters by using average word embeddings of each cluster label and display the score to user for reference. Depending on the content of cluster labels and relatedness score, the user makes as decision for merging of clusters. The process of displaying clusters continues until the user stops or no new cluster is found for merging.

4.3.1. Semantic Relatedness Calculation

Semantic relatedness is a degree which measures how related the meaning of two or more words are. It is generally calculated by investigating how often words are used together in a large corpus. An example for semantic relatedness would be **"bus"** and **"road"** or **"driver"**.

Due to its window training method, the vector representations obtained by word embeddings retain their semantic relatedness. To compute semantic relatedness between labels,

we use word embeddings. To obtain a normalized score, we take the average of word embeddings of each label and use cosine similarity to obtain a relatedness score. Given two clusters c_i and c_j, the relatedness score is:

$$rScore(c_i, c_j) = cosinesimilarity(\frac{l_i}{|l_i|}, \frac{l_j}{|l_j|}) = \frac{1}{|l_i||l_j|} * \frac{\sum_{k=0}^{d} l_{ik} l_{jk}}{\sqrt{\sum_{k=0}^{d} l_{ik}^2} \sqrt{\sum_{k=0}^{d} l_{jk}^2}}$$

For word embeddings model, we use Google's pre-trained model [65]. The model is trained over for 3 million words and phrases using the data obtained by Google News. The dimensionality of each vector in the pre-trained model is 300.

4.3.2. Complexity Analysis

In the interactive design phase, we select k clusters from all clusters and calculate pairwise relatedness score. The pairwise calculation and storage of the relatedness score require quadratic time and space complexity, making the complexity $O(k^2)$. On default, we set k to 100 which is the average number of clusters which can be shown to user without having visual crowdedness. This makes the time and space complexity negligible, despite being quadratic.

5. Interactive System Design

For the purpose of interactive merging, we design a graphical user interface using JavaScript and Java Servlets. Our implementation is divided in two sections: Client and server side. In the client side, the user can upload the data, adjust threshold parameters, display and download cluster results, while in the server side the algorithm is run using Java and the communication is done using Servlets.

5.1. Client Side Implementation

The interface in the client side has two pages: Lexical clustering and interactive merging pages. In the lexical clustering page, the result of the lexical clustering is shown along with a bubble and bar histogram. In addition to histograms, the user can adjust parameters for lexical clustering send into server for re-clustering.

In the interactive merging phase, the semantic relatedness of clusters is shown on a wheel. It is not possible to display all clusters on the wheel; therefore, we select k clusters which has a certain amount of words for meaningful relatedness calculation.

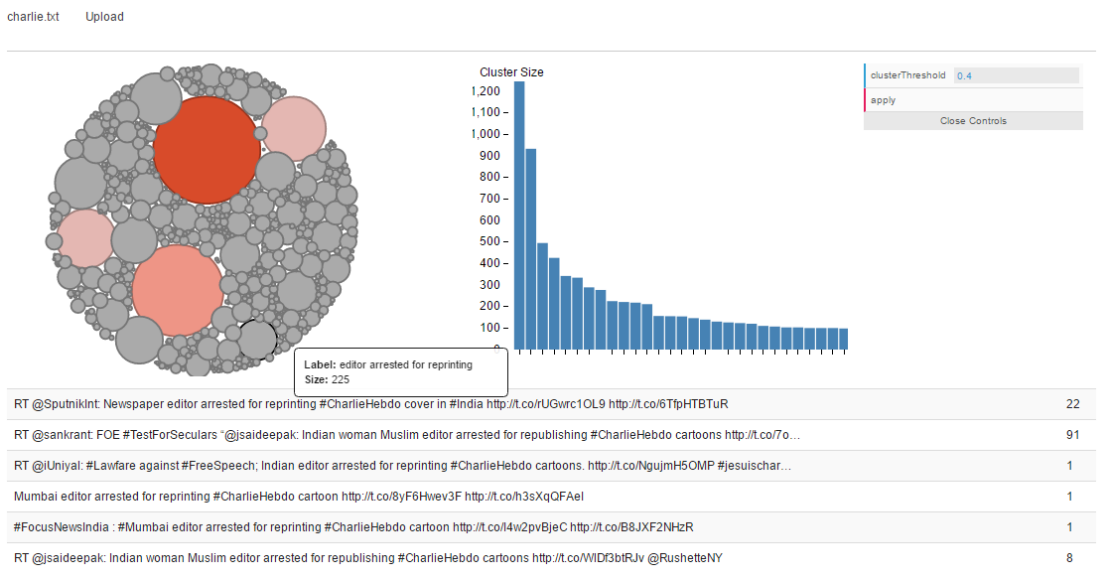


Figure 3: Lexical clustering GUI when Charlie Hebdo data is used

Each cluster is represented on the wheel by its label and the connection between clusters is presented as a path between the labels around the wheel. It is possible to adjust the threshold for displaying connections which is a masking operation on the semantic relatedness matrix. It is also possible to change parameters for cluster display which results to the recalculation of semantic matrix in server.

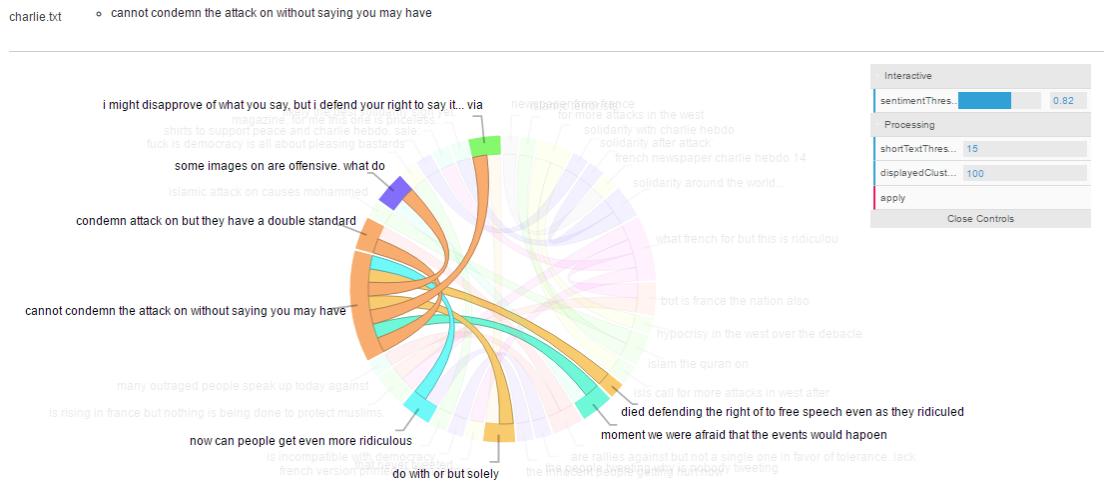


Figure 4: Interactive merging GUI when merging a cluster, using Charlie Hebdo data

5.2. Server Side Implementation

We make the main computations of the algorithm such as lexical clustering and semantic relatedness calculation in the server side. As we have a client-server system where multiple

clients can connect to server, we assign each client a session id which is valid during the duration of his/her usage. When a client uploads data to the server for clustering, the server processes the data on memory and sends back the cluster results. However, the data may be necessary for re-clustering or parameter adjustment, for further processing it needs to stay in a persistent state. In file system, we open a temporary folder associated with the clients session id and serialize the suffix tree inside the folder. When a further processing is required, the suffix tree is deserialized and the lexical clustering process is rerun starting at cluster creation phase.

The time required for the calculation of semantic relatedness matrix is too long for interactive systems. Therefore, we calculate semantic relatedness matrix using an asynchronous thread and serialize the matrix. Another thread waits for this serialization when it is done, it deserializes the matrix and sends it to client side. The mutual exclusion between threads is done by mutexes.

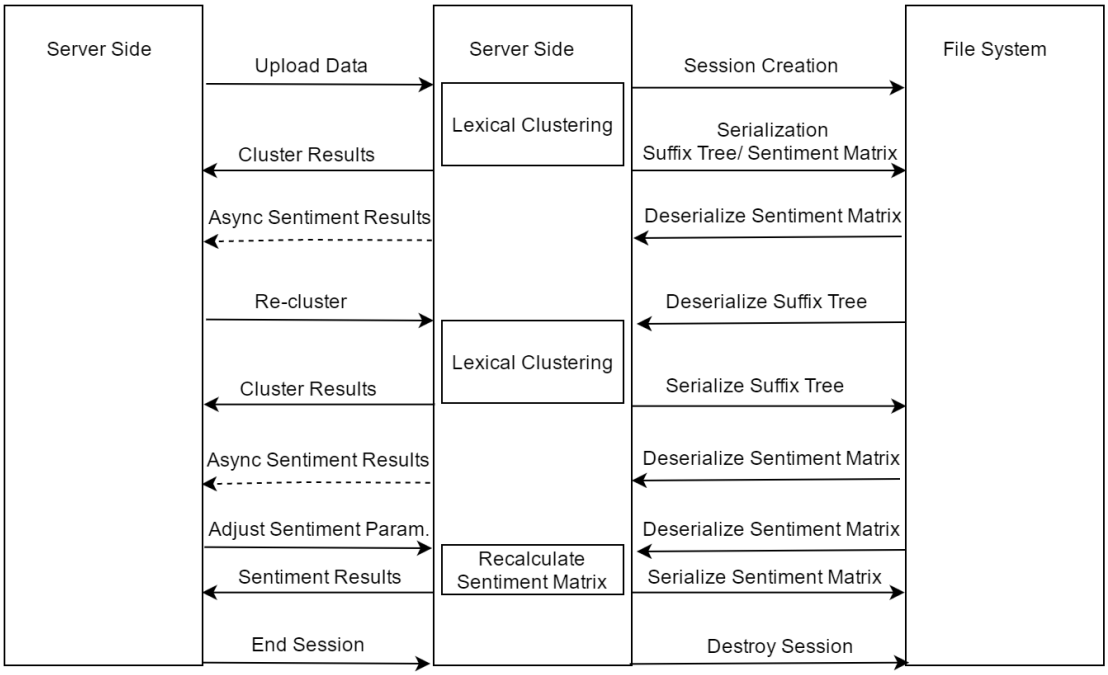


Figure 5: Design of server side implementation

6. Experimental Evaluation

In this section we will talk about experimental evaluations done on our algorithm. We evaluate our algorithm using four different datasets taken from Twitter Streaming API which are about Charlie Hebdo, Christmas, NBA and Trump. Each dataset contains more than 15.000 tweets.

Given a dataset $D = \{t_1, t_2, ..., t_k\}$, we create a set of clusters $C = \{c_1, c_2, ...c_n\}$ such that every cluster c_i contains at least two tweets. During the lexical clustering, we set **thrSize**

to 1.2 and **thrString** to 0.8 use these values for evaluation. In the experimental evaluation, we measure the quality of clusters by using intra-cluster similarity, class validation and compression ratio. We use intra-cluster similarity for lexical clustering, class validation for interactive merging and compression ratio to measure the summarization rate of the data. We perform several experiments to find the optimum intra-cluster similarity, class validation and compression ratio values.

6.1. Intra-cluster Similarity

Intra-cluster similarity is a degree which measures how similar tweets inside a cluster are. We use Longest Common Subsequence (LCS) as the basis for intra-cluster similarity and calculate pairwise similarity between cluster elements using the following equation:

$$pairwiseSim(t_i, t_j) = \frac{2 * |LCS(t_i, t_j)|}{|t_i| + |t_j|} \tag{1}$$

$$intraCSim(c) = \frac{2}{|c| * (|c| - 1)} * \sum_{i=0}^{|c|} \sum_{j=i+1}^{|c|} pairwiseSim(t_i, t_j) \tag{2}$$

We calculate a pairwise similarity for each item in the cluster using equation 1 and use equation 2 to find the intra-cluster similarity. Each cluster c_i has an intra-cluster similarity. To find the average intra-cluster similarity of all clusters in set C, we use the following equations:

$$avgISim(C) = \frac{1}{n} * \sum_{i=0}^{n} intraCSim(c_i) \tag{3}$$

$$wAvgIntraSim(C) = \frac{1}{\sum_{i=0}^{n} |c_i|} * \sum_{i=0}^{n} |c_i| * intraCSim(c_i) \tag{4}$$

Equation 3 calculates the average intra-cluster similarity, while equation 4 uses weighted average intra-cluster similarity where the size of clusters are used as weights. Equation 4 is a significant measure, because it shows the distribution of intra-cluster scores with the combination of equation 3.

6.2. Compression Ratio

One of the purposes of lexical clustering is to create a summarized data by creating clusters and cluster representatives. We use compression ratio to measure to compression of data obtained by lexical clustering:

$$compR(C) = \frac{n}{\sum_{i=0}^{n} |c_i|} \tag{5}$$

$$uCompR(C) = \frac{n + |\text{\# of unclustered tweets}|}{\sum_{i=0}^{n} |c_i| + |\text{\# of unclustered tweets}|} \tag{6}$$

For compression ratio, we have two equations: Equation 5 looks at the compression on clusters and leaves unclustered tweets out of the calculation. Equation 6 treats the unclustered tweets as single-element clusters and includes to the calculation. Both equations are significant in their own aspects: Equation 5 displays the compression ratio of clusters and it is a logical measure as not all tweets can possibly belong to a cluster and the unclustered tweets are negligible compared to clusters. On the other hand, in practice the unclustered tweets may contain a significant amount of data and they may have to be represented. Equation 6 and the difference between it and equation 5 displays this representation.

6.3. Class Validation

In order to measure the effectiveness of interactive merging, we use class validation. For interactive merging, we use a combination of 4 datasets; Each dataset has its own topic and we define these topics as the class label of the tweets in the dataset. Ideally, each cluster should consist of only one class label, we use the following equation the measure the class purity of each cluster:

$$classValS(c_i) = \frac{|max_{class}|}{|c_i|} \tag{7}$$

Equation 7 finds the most occurring class label in the elements of a cluster and takes the ratio of it with the cluster size.

6.4. Lexical Clustering Evaluation

In this subsection, we will evaluate lexical clustering by using 4 datasets. We will use intra-cluster similarity and compression ratio for evaluation. In our experiments we try to maximize the intra-cluster similarity and minimize the compression ratio. For this reason, we make experiments with the different thresholds to find the optimum threshold. Our experiments start with the threshold 0.1 until 0.8 with incremental steps of 0.1.

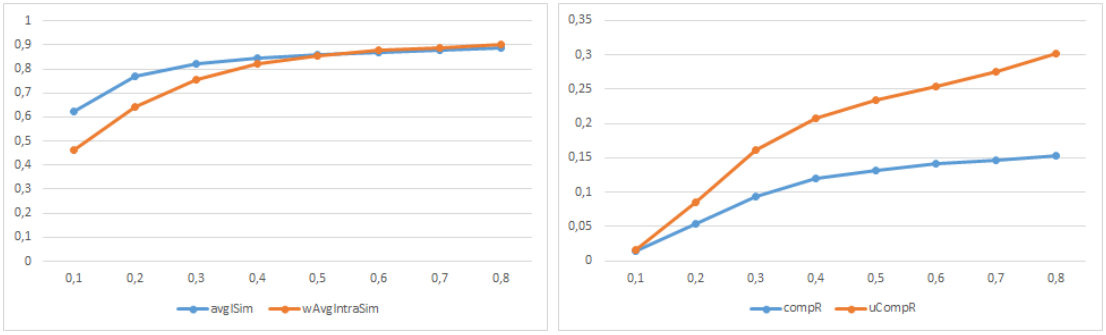
For our experiments, we use a computer which has an i3 3.7 GHz processor and 16 GB RAM. In all threshold and dataset combinations, the experiment takes less than 1 minute. The table 1 demonstrates that suffix tree creation and cluster creation phases take most of the time in the lexical clustering phase.

Table 1: Timing of experimental evaluations with threshold 0.3 (in secs)

	Charlie Hebdo	Christmas	NBA	Trump	Combined
Preprocessing	0.76	1.42	0.19	0.61	0.61
Suffix tree construction	3.24	16.73	0.96	5.24	7.51
Index size population	1.12	3.58	0.57	2.24	2.09
Duplicate node elimination	0.29	0.67	0.15	0.34	0.39
Cluster creation	3.82	17.76	2.49	8.53	8.63
Overlapping Elimination	0.53	3.02	0.09	1.01	1.01

6.4.1. Charlie Hebdo

Charlie Hebdo dataset is a set of tweets collected by Twitter Streaming API during Charlie Hebdo events in 2015. The dataset is collected using hashtags #CharlieHebdo and #JeSuisCharlie and contains 32.883 tweets. In the dataset, after processing, 15 tweets are found which are too short or have no content.



(a) Intra-Cluster Similarities (b) Compression Ratios

Figure 6: Experimental evaluations on Charlie Hebdo dataset

Figure 6(a) shows a natural increase in intra-cluster similarity as the threshold increases. Starting with threshold value 0.3, the intra-cluster similarity is above 0.7, which shows that the elements in the clusters are highly similar and pure. After the threshold 0.5, weighted average intra-cluster similarity passes over average intra-cluster similarity which indicates that the similarity is higher on larger clusters compared to small clusters.

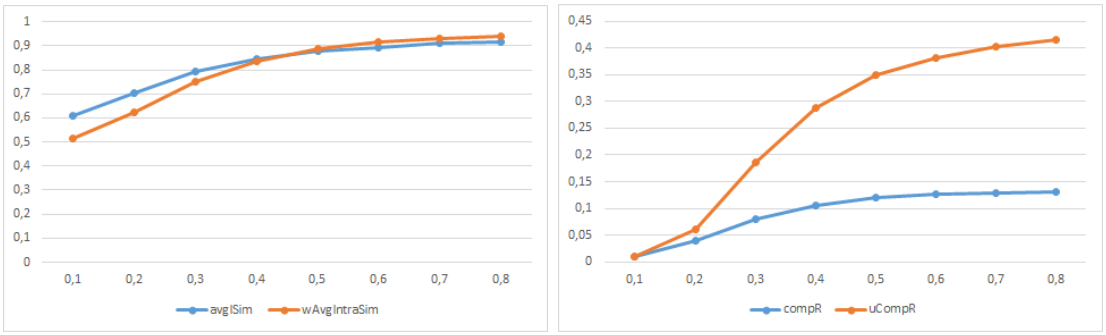
Figure 6(b) shows a natural increase in compression ratio as the threshold increases. Compression ratios rise sharply until 0.3, then the rise slows down. The difference between compR and uCompR shows the ratio of unclustered tweets and it becomes over 10% of the whole dataset after 0.4.

We aim to obtain a high intra-cluster similarity with a low compression and unclustered tweets ratio and for this purpose, threshold values between 0.3-0.4 applies well to the Charlie Hebdo dataset. This interval gives clusters with a high intra-cluster similarity and 10-15% compression ratio with 5-10% unclustered tweets.

6.4.2. Christmas

Christmas dataset is a set of tweets collected by Twitter Streaming API before Christmas. The dataset is collected using hashtag #Christmas and contains 120.864 tweets. In the dataset, 3269 tweets have little or no content and removed from clustering process.

Figure 7(a) shows similarity to the intra-cluster similarity graphs in Charlie Hebdo, having a natural increase. Starting with threshold value 0.3, the above intra-cluster similarity between clusters becomes above 0.7 and at the threshold value 0.5, weighted average intra-cluster similarity passes over average intra-cluster similarity. On the other hand, figure 7(b) demonstrates that unclustered tweet ratio increases sharply after the threshold value 0.3. The optimal thresholding value for intra-cluster similarity is above 0.3, while it is below 0.3



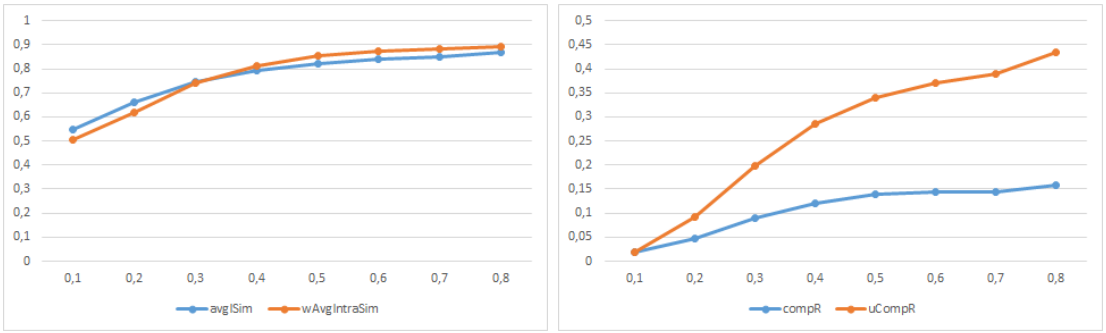
(a) Intra-Cluster Similarities (b) Compression Ratios

Figure 7: Experimental evaluations on Christmas dataset

for compression ratios, making the optimal threshold value for Christmas dataset around 0.3.

6.4.3. NBA

NBA dataset is a set of tweets collected by Twitter API. The dataset is collected using hashtag #NBA and contains 17.554 tweets. In the dataset, 5 tweets are removed due to not having content.



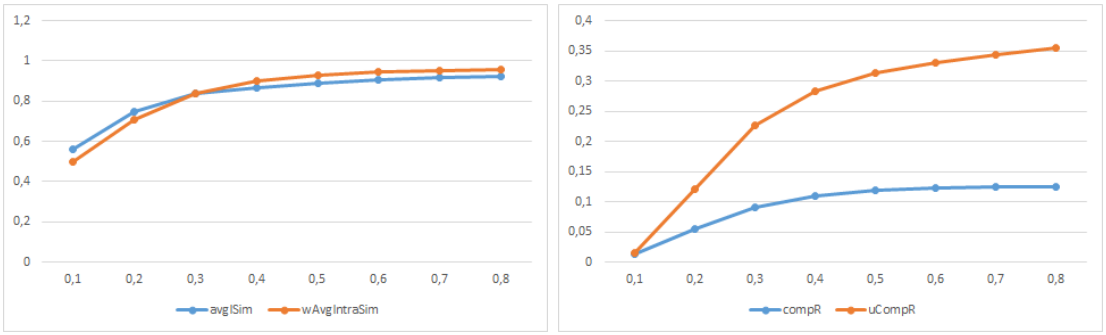
(a) Intra-Cluster Similarities (b) Compression Ratios

Figure 8: Experimental evaluations on NBA dataset

Figure 8(a) shows high intra-cluster similarity starting from 0.2 and at 0.3 the weighted average intra-cluster similarity passes over average intra-cluster similarity around 0.75. Figure 8(b) shows that compression ratio is below 15% at almost every threshold and the unclustered tweet ratio rises after 0.3, making the optimal threshold for the dataset around 0.3.

6.4.4. Trump

Trump dataset is a set of tweets collected by Twitter API. The dataset is collected using hashtag #Trump and contains 55.223 tweets. There are 135 tweets in the dataset which have little or no content.



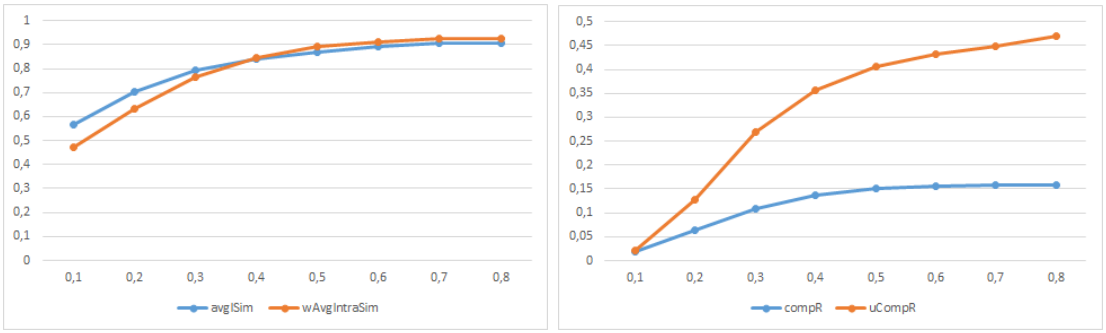
(a) Intra-Cluster Similarities (b) Compression Ratios

Figure 9: Experimental evaluations on Trump dataset

The Trump dataset has a high intra-cluster similarity from low thresholds, evident in figure 9(a). Starting from 0.2, the intra-cluster similarity rises from 0.7 and weighted intra-cluster similarity passes over intra-cluster similarity at 0.3. Figure 9(b) demonstrates similar compression ratio characteristics as the other datasets. The unclustered tweet ratio rises sharply starting from 0.3, making the optimum threshold for Trump dataset between 0.2 and 0.3.

6.5. Interactive Merging

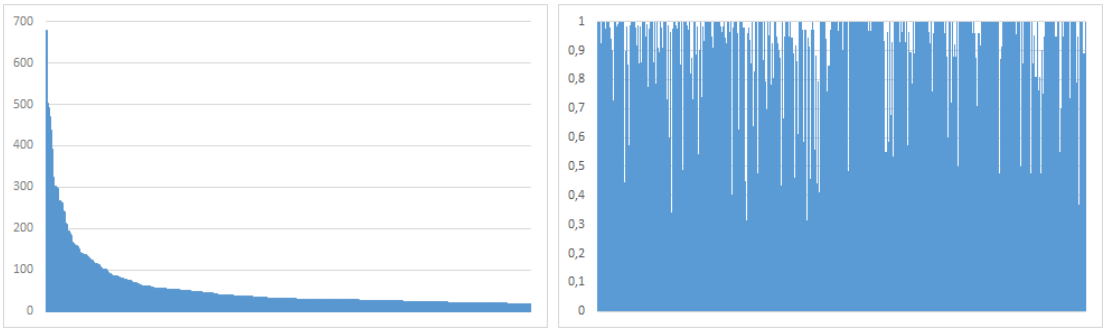
In this subsection we will make evaluations for interactive merging. In order to evaluate merging, we will use a combined dataset which consists of Charlie Hebdo, Christmas, NBA and Trump dataset. From each dataset, 15000 tweets are retrieved.



(a) Intra-Cluster Similarities (b) Compression Ratios

Figure 10: Experimental evaluations on the combined dataset

In order to evaluate interactive merging, we first need to select a suitable threshold for lexical clustering. For this purpose, we use figures 10(a) and 10(b).The combined dataset reaches a high intra-cluster similarity at threshold 0.3. At this threshold the compression ratio is around 10% which is a good ratio. As the other datasets also have their optimal thresholds at 0.3, we select 0.3 for interactive merging.



(a) Histogram based on cluster sizes (b) Class Validation scores

Figure 11: Histogram and class val. scores on the top 500 of the combined dataset

Figure 11(a) demonstrates the histogram at threshold 0.3. 47% of the combined dataset is represented by top 500 clusters, becoming a good representation for the whole data.

Figure 11(b) shows the class validation scores for first 500 clusters. Lexical clustering has a very high class validation score, at the threshold 0.3 the weighted class validation score is 0.92 and average class validation score is 0.94. As long as merging is done between the same classes, we expect class validation scores to remain constant or rise, while the average compression ratio decreases for clusters.

Interactive merging process demonstrates that there are content-wise similar clusters which are constructed by different words. These clusters could not be merged by lexical clustering, however they can be detected and merged with semantic relatedness. By using our user interface, human agents were able to merge 47 clusters which have the exact same context with different wordings.

6.6. Results and Discussion

In this work, we use 4 different datasets to perform experimentation and evaluate our algorithm. Each dataset contains at least 15.000 tweets and collected using Twitter API. We perform thresholding experiments on each dataset and evaluate the lexical clustering part of our algorithm and we perform an experiment on a dataset combined from these 4 datasets to measure the effectiveness of interactive merging.

Our evaluations show that the lexical clustering part of our algorithm works well in Twitter datasets. We observe that there is a trade-off between cluster similarity and compression of data alongside with the size of unclustered tweets. As the threshold increases, the cluster similarity and the size of unclustered tweets increase, while the compression of data decreases.

Our experiments show that the optimum threshold for the 4 datasets is between 0.3 and 0.4. At this interval we obtain an average intra-cluster similarity higher than 0.7, while the compression ratio stays around 10% and the ratio of uncompressed tweets is around 10-15%. Considering linear time and space complexity of the algorithm, lexical clustering gives outstanding results as a data representation and summarization tool. The results can further be utilized by using other data processing algorithms with high time complexities:

With the reduction of data these data processing algorithms will run faster by the order of their time complexities.

We use a combined dataset of 4 different topics for evaluation of interactive merging. The interactive merging phase demonstrates that there are clusters with similar contexts and different wordings. In this phase, the human agent is the key for successful merging. Our experiment shows that a further reduction and compression in data is possible with interactive merging.

7. Conclusion and Future Work

With the advancement of technology and change of communication forms towards digital medium, social media platforms become a valuable source of data for analysis. Twitter is such a social media platform which grows continuously and offers public data for researchers. In contrary to the widely researched fields such as tweet classification and topic detection, summarization and representation of Twitter data is also a less researched field which require recognition of equal size.

In this work, we propose a hybrid tweet clustering algorithm for summarization of Twitter data. As part of our clustering algorithm, we propose a new character-based suffix tree clustering algorithm tailored for short-document sets. The new suffix tree clustering algorithm is linear in space and time for document sets with fixed maximum length. The algorithm matches well with Twitter data and obtains average and weighted average intra-cluster similarity over 0.7 with 10% compression rate. For further refinement of clusters, we propose an interactive merging method based on semantic relatedness. The quality of the interactive merging is dependent on the human agent; however, it is able to find clusters with similar contexts but different phrases.

The main use case of the algorithm is summarization of representation of Twitter data using clusters. Knowledge and the distribution of topics about a firm or a politician in Twitter is very valuable for information management departments and we can obtain this knowledge by using our algorithm. Another use case for our algorithm lies in its compression rate and linear time: It can swiftly compress the data and the resulting clusters can be used for other data processing algorithms.

For future work, our algorithm has many aspects which can be improved and evaluated. Our algorithm is a variant one-pass algorithm with linear time and space, however with its decoupled structure of clustering and with the current research on the parallel construction of suffix trees, it can be tuned to work in distributed systems. The suffix tree construction of our algorithm is based on Ukkonen's algorithm which has an on-line property; therefore, another possible future work for our algorithm is to convert it to an on-line algorithm. As for interactive merging part, a research can be done to find better heuristics for semantic relatedness between words and a semi-automated or automated system can be designed to reduce the dependency on human agents.

Appendix A. Ukkonen's Algorithm: Pseudocode

Below is the complete pseudo-code of the Ukkonen's algorithm. To keep consistency at the root state, an invisible state $\perp$ before root is created. This state is connected to the root with all possible transitions from alphabet.

Algorithm 5 Construction of STree(T) for string $T = t_1 t_2 ... \#$ in alphabet $\sum = \{t_1, ..., t_m\}$

create states *root* and $\perp$
for $j \leftarrow 1, ..., m$ **do** create transition $g'(\perp, (-j, -j)) = root$
create suffix link $f'(root) = \perp$;
$s \leftarrow root$; $k \leftarrow 1$; $i \leftarrow 0$;
while $t_{i+1} \neq \#$ **do**
    $i \leftarrow i + 1$;
    $(s, k) \leftarrow$ update$(s, (k, i))$;
    $(s, k) \leftarrow$ canonize$(s, (k, i))$;

The functions of the algorithm update, test-and-split and canonize are:

function UPDATE(s,(k,i))
    $oldr \leftarrow root$; (end-point, r) $\leftarrow$ test-and-split$(s, (k, i-1), t_i)$;
    while not end-point **do**
        create new transition $g'(r, (i, \infty)) = r'$ where r' is a new state;
        if $oldr \neq root$ **then** create new suffix link $f'(oldr) = r$;
        $oldr \leftarrow r$;
        $(s, k) \leftarrow$ canonize$(f'(s), (k, i-1))$;
        (end-point, r) $\leftarrow$ test-and-split$(s, (k, i-1), t_i)$;
    if $oldr \neq root$ **then** create new suffix link $f'(oldr) = s$;
    return (s,k);

```
1: function TEST-AND-SPLIT(s,(k,i),t)
2:     if k ≤ p then
3:         let g'(s,(k',p')) = s' be the t_k-transition from s;
4:         if t = t_{k'+p-k+1} then return (true, s);
5:         else
6:             replace the t_k-transition above by transitions g'(s,(k',k'+p-k)) = r
7:             and g'(r,(k'+p-k+1,p')) = s' where r is a new state;
8:             return (false,r);
9:     else
10:        if there is no t-transition from s then return (false,s);
11:        else  return (true,s);
```

```
1: function CANONIZE((s,(k,p))
2:     if p < k then return (s,k);
3:     else
4:         find the t_k-transition g'(s,(k',p')) = s' from s;
5:         while p' - k' ≤ p - k do
6:             k ← k + p' - k' + 1;
7:             s ← s';
8:             if k ≤ p then find the t_k-transition g'(s,(k',p')) = s' from s;
9:         return (s,k);
```

Appendix B. Tabular results of evaluations on each dataset

Table B.2: Evaluations of Charlie Hebdo dataset

Threshold	avgISim	wAvgIntraSim	compR	uCompR	Cluster Size	Unclustered Tweets	Total Size
0,1	0,625	0,463	0,015	0,015	497	7	32883
0,2	0,770	0,640	0,054	0,086	1715	1115	32883
0,3	0,823	0,755	0,094	0,161	2850	2443	32883
0,4	0,847	0,824	0,120	0,208	3535	3311	32883
0,5	0,861	0,856	0,132	0,234	3831	3858	32883
0,6	0,871	0,876	0,141	0,254	4029	4327	32883
0,7	0,879	0,888	0,147	0,275	4105	4928	32883
0,8	0,890	0,903	0,152	0,302	4117	5864	32883

Table B.3: Evaluations of Christmas dataset

	avgISim	wAvgIntraSim	compR	uCompR	Cluster Size	Unclustered Tweets	Total Size
0,1	0,610	0,515	0,010	0,010	1136	13	120846
0,2	0,702	0,622	0,039	0,060	4537	2526	120846
0,3	0,795	0,748	0,080	0,186	8293	13624	120846
0,4	0,845	0,834	0,106	0,287	9943	23854	120846
0,5	0,876	0,886	0,120	0,349	10474	30555	120846
0,6	0,894	0,914	0,127	0,382	10574	34374	120846
0,7	0,909	0,929	0,129	0,402	10450	36848	120846
0,8	0,916	0,938	0,131	0,414	10361	38372	120846

Table B.4: Evaluations of NBA dataset

	avgISim	wAvgIntraSim	compR	uCompR	Cluster Size	Unclustered Tweets	Total Size
0,1	0,547	0,507	0,019	0,020	337	17	17554
0,2	0,662	0,620	0,048	0,091	798	802	17554
0,3	0,745	0,741	0,089	0,198	1372	2109	17554
0,4	0,792	0,814	0,121	0,285	1721	3276	17554
0,5	0,822	0,857	0,139	0,339	1878	4077	17554
0,6	0,839	0,873	0,143	0,369	1846	4640	17554
0,7	0,849	0,882	0,144	0,390	1795	5057	17554
0,8	0,867	0,894	0,159	0,434	1872	5742	17554

Table B.5: Evaluations of Trump dataset

	avgISim	wAvgIntraSim	compR	uCompR	Cluster Size	Unclustered Tweets	Total Size
0,1	0,561	0,500	0,014	0,014	764	30	55223
0,2	0,747	0,708	0,055	0,120	2797	3836	55223
0,3	0,837	0,840	0,091	0,228	4245	8320	55223
0,4	0,869	0,899	0,109	0,283	4844	10795	55223
0,5	0,892	0,930	0,119	0,314	5095	12268	55223
0,6	0,906	0,944	0,122	0,331	5131	13163	55223
0,7	0,916	0,953	0,124	0,343	5116	13846	55223
0,8	0,923	0,959	0,125	0,354	5084	14488	55223

Table B.6: Evaluations of combined dataset

	avgISim	wAvgIntraSim	compR	uCompR	Cluster Size	Unclustered Tweets	Total Size
0,1	0,566	0,472	0,020	0,020	1195	26	60000
0,2	0,705	0,633	0,064	0,128	3561	4132	60000
0,3	0,793	0,763	0,108	0,268	5255	10844	60000
0,4	0,838	0,843	0,136	0,356	6031	15316	60000
0,5	0,870	0,891	0,151	0,406	6241	18130	60000
0,6	0,890	0,911	0,155	0,431	6185	19689	60000
0,7	0,904	0,924	0,157	0,449	6080	20851	60000
0,8	0,904	0,924	0,157	0,470	6091	22103	60000

References

[1] Company — about - twitter about, https://about.twitter.com/company, accessed: 2016-11-22.

[2] Twitter usage statistics - internet live stats, http://www.internetlivestats.com/twitter-statistics/, accessed: 2016-11-22.

[3] K. S. Gaikwad, M. S. Patwardhan, Tweets clustering: Adaptive pso, in: 2014 Annual IEEE India Conference (INDICON), IEEE, 2014, pp. 1–6.

[4] J. Yin, Clustering microtext streams for event identification., in: IJCNLP, 2013, pp. 719–725.

[5] N. Kaur, C. M. Gelowitz, A tweet grouping methodology utilizing inter and intra cosine similarity, in: Electrical and Computer Engineering (CCECE), 2015 IEEE 28th Canadian Conference on, IEEE, 2015, pp. 756–759.

[6] S. Thaiprayoon, A. Kongthon, P. Palingoon, C. Haruechaiyasak, Search result clustering for thai twitter based on suffix tree clustering, in: Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2012 9th International Conference on, IEEE, 2012, pp. 1–4.

[7] Y. Fang, H. Zhang, Y. Ye, X. Li, Detecting hot topics from twitter: A multiview approach, Journal of Information Science (2014) 0165551514541614.

[8] O. E. Zamir, Clustering web documents: a phrase-based method for grouping search engine results, Ph.D. thesis, University of Washington (1999).

[9] F. X. Diebold, On the origin (s) and development of the term'big data'.

[10] C. C. Aggarwal, C. K. Reddy, Data clustering: algorithms and applications, CRC Press, 2013.

[11] S. Guha, R. Rastogi, K. Shim, Cure: an efficient clustering algorithm for large databases, in: ACM SIGMOD Record, Vol. 27, ACM, 1998, pp. 73–84.

[12] D. Palguna, V. Joshi, V. Chakaravarthy, R. Kothari, L. Subramaniam, analysis of sampling algorithms for twitter, in: Proc. of the 24th Int. Joint Conf. on Artificial Intelligence (IJCAI 2015), 2015, pp. 967–973.

[13] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, S. L. Salzberg, Versatile and open software for comparing large genomes, Genome biology 5 (2) (2004) 1.

[14] M. Höhl, S. Kurtz, E. Ohlebusch, Efficient multiple genome alignment, Bioinformatics 18 (suppl 1) (2002) S312–S320.

[15] P. Weiner, Linear pattern matching algorithms, in: Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on, IEEE, 1973, pp. 1–11.

[16] E. M. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM (JACM) 23 (2) (1976) 262–272.

[17] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.

[18] M. Farach, Optimal suffix tree construction with large alphabets, in: Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on, IEEE, 1997, pp. 137–143.

[19] Github - abahgat/suffixtree: A java implementation of a generalized suffix tree using ukkonen's algorithm, https://github.com/abahgat/suffixtree/, accessed: 2016-11-22.

[20] Y. Bengio, R. Ducharme, P. Vincent, C. Jauvin, A neural probabilistic language model, journal of machine learning research 3 (Feb) (2003) 1137–1155.

[21] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Advances in neural information processing systems, 2013, pp. 3111–3119.

[22] J. Pennington, R. Socher, C. D. Manning, Glove: Global vectors for word representation, in: Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1532–1543.
URL http://www.aclweb.org/anthology/D14-1162

[23] A. Islam, D. Inkpen, Semantic text similarity using corpus-based word similarity and string similarity, ACM Transactions on Knowledge Discovery from Data (TKDD) 2 (2) (2008) 10.

[24] A. Banerjee, J. Ghosh, Clickstream clustering using weighted longest common subsequences, in: Proceedings of the web mining workshop at the 1st SIAM conference on data mining, Vol. 143, Citeseer, 2001, p. 144.

[25] H. Kwak, C. Lee, H. Park, S. Moon, What is twitter, a social network or a news media?, in: Proceedings of the 19th international conference on World wide web, ACM, 2010, pp. 591–600.
[26] A. Java, X. Song, T. Finin, B. Tseng, Why we twitter: understanding microblogging usage and communities, in: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis, ACM, 2007, pp. 56–65.
[27] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, J. Sperling, Twitterstand: news in tweets, in: Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems, ACM, 2009, pp. 42–51.
[28] J. Golbeck, D. Hansen, A method for computing political preference among twitter followers, Social Networks 36 (2014) 177–184.
[29] F. Toriumi, S. Baba, Real-time tweet classification in disaster situation, in: Proceedings of the 25th International Conference Companion on World Wide Web, International World Wide Web Conferences Steering Committee, 2016, pp. 117–118.
[30] S. B. Kaleel, A. Abhari, Cluster-discovery of twitter messages for event detection and trending, Journal of Computational Science 6 (2015) 47–57.
[31] G. Ifrim, B. Shi, I. Brigadir, Event detection in twitter using aggressive filtering and hierarchical tweet clustering, in: Second Workshop on Social News on the Web (SNOW), Seoul, Korea, 8 April 2014, ACM, 2014.
[32] M.-C. Yang, H.-C. Rim, Identifying interesting twitter contents using topical analysis, Expert Systems with Applications 41 (9) (2014) 4330–4336.
[33] Y.-H. Kim, S. Seo, Y.-H. Ha, S. Lim, Y. Yoon, Two applications of clustering techniques to twitter: Community detection and issue extraction, Discrete dynamics in nature and society 2013.
[34] F. Ahmed, M. Abulaish, A generic statistical approach for spam detection in online social networks, Computer Communications 36 (10) (2013) 1120–1129.
[35] X. Zheng, Z. Zeng, Z. Chen, Y. Yu, C. Rong, Detecting spammers on social networks, Neurocomputing 159 (2015) 27–34.
[36] Z. Miller, B. Dickinson, W. Deitrick, W. Hu, A. H. Wang, Twitter spammer detection using data stream clustering, Information Sciences 260 (2014) 64–73.
[37] S. Lee, J. Kim, Early filtering of ephemeral malicious accounts on twitter, Computer Communications 54 (2014) 48–57.
[38] G. Yan, Peri-watchdog: Hunting for hidden botnets in the periphery of online social networks, Computer Networks 57 (2) (2013) 540–555.
[39] J. Martinez-Romo, L. Araujo, Detecting malicious tweets in trending topics using a statistical analysis of language, Expert Systems with Applications 40 (8) (2013) 2992–3000.
[40] A. Go, R. Bhayani, L. Huang, Twitter sentiment classification using distant supervision, CS224N Project Report, Stanford 1 (2009) 12.
[41] A. Montejo-Ráez, E. Martínez-Cámara, M. T. Martín-Valdivia, L. A. Ureña-López, Ranked wordnet graph for sentiment polarity classification in twitter, Computer Speech & Language 28 (1) (2014) 93–107.
[42] L. Barbosa, J. Feng, Robust sentiment detection on twitter from biased and noisy data, in: Proceedings of the 23rd International Conference on Computational Linguistics: Posters, Association for Computational Linguistics, 2010, pp. 36–44.
[43] L. Jiang, M. Yu, M. Zhou, X. Liu, T. Zhao, Target-dependent twitter sentiment classification, in: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1, Association for Computational Linguistics, 2011, pp. 151–160.
[44] A. Agarwal, B. Xie, I. Vovsha, O. Rambow, R. Passonneau, Sentiment analysis of twitter data, in: Proceedings of the workshop on languages in social media, Association for Computational Linguistics, 2011, pp. 30–38.
[45] B. Sriram, D. Fuhry, E. Demir, H. Ferhatosmanoglu, M. Demirbas, Short text classification in twitter to improve information filtering, in: Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval, ACM, 2010, pp. 841–842.

[46] L. Derczynski, D. Maynard, G. Rizzo, M. van Erp, G. Gorrell, R. Troncy, J. Petrak, K. Bontcheva, Analysis of named entity recognition and linking for tweets, Information Processing & Management 51 (2) (2015) 32–49.
[47] J. J. Jung, Online named entity recognition method for microtexts in social networking services: A case study of twitter, Expert Systems with Applications 39 (9) (2012) 8066–8070.
[48] X. Liu, M. Zhou, Two-stage ner for tweets with clustering, Information Processing & Management 49 (1) (2013) 264–273.
[49] A. Adel, E. Elfakharany, A. Badr, Clustering tweets using cellular genetic algorithm.
[50] J. Oliva, J. I. Serrano, M. D. del Castillo, Á. Iglesias, Symss: A syntax-based measure for short-text semantic similarity, Data & Knowledge Engineering 70 (4) (2011) 390–405.
[51] L. Wenyin, X. Quan, M. Feng, B. Qiu, A short text modeling method combining semantic and statistical information, Information Sciences 180 (20) (2010) 4031–4041.
[52] F. H. Khan, S. Bashir, U. Qamar, Tom: Twitter opinion mining framework using hybrid classification scheme, Decision Support Systems 57 (2014) 245–257.
[53] A. Z. Khan, M. Atique, V. Thakare, Combining lexicon-based and learning-based methods for twitter sentiment analysis, International Journal of Electronics, Communication and Soft Computing Science & Engineering (IJECSCSE) (2015) 89.
[54] H. Chim, X. Deng, A new suffix tree similarity measure for document clustering, in: Proceedings of the 16th international conference on World Wide Web, ACM, 2007, pp. 121–130.
[55] Y. Zhuang, Y. Chen, Improving suffix tree clustering algorithm for web documents.
[56] C. Huang, J. Yin, F. Hou, Text clustering using a suffix tree similarity measure, Journal of computers 6 (10) (2011) 2180–2186.
[57] A. Annadurai, A. Annadurai, Architecture of personalized web search engine using suffix tree clustering, in: Signal Processing, Communication, Computing and Networking Technologies (ICSCCN), 2011 International Conference on, IEEE, 2011, pp. 604–608.
[58] J. Janruang, S. Guha, Applying semantic suffix net to suffix tree clustering, in: Data Mining and Optimization (DMO), 2011 3rd Conference on, IEEE, 2011, pp. 146–152.
[59] J. Janruang, S. Guha, Semantic suffix tree clustering, in: First IRAST International Conference on Data Engineering and Internet Technology, DEIT, Citeseer, 2011.
[60] D. Tang, F. Wei, B. Qin, T. Liu, M. Zhou, Coooolll: A deep learning system for twitter sentiment classification, in: Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014), 2014, pp. 208–212.
[61] S. Xiong, Improving twitter sentiment classification via multi-level sentiment-enriched word embeddings, arXiv preprint arXiv:1611.00126.
[62] S. Wijeratne, L. Balasuriya, D. Doran, A. Sheth, Word embeddings to enhance twitter gang member profile identification, arXiv preprint arXiv:1610.08597.
[63] X. Yang, C. Macdonald, I. Ounis, Using word embeddings in twitter election classification, arXiv preprint arXiv:1606.07006.
[64] C. De Boom, S. Van Canneyt, T. Demeester, B. Dhoedt, Representation learning for very short texts using weighted word embedding aggregation, Pattern Recognition Letters 80 (2016) 150–156.
[65] Google — word2vec, `https://code.google.com/archive/p/word2vec/`, accessed: 2016-11-22.