Interactive hierarchical clustering tool
for text segments

Karl-Oskar Masing
University of Tartu
Institute of Computer Science
Liivi 2
Tartu, Estonia
kom@ut.ee

Timo Petmanson
University of Tartu
Institute of Computer Science
Liivi 2
Tartu, Estonia
tpetmanson@gmail.com

Sven Laur
University of Tartu
Institute of Computer Science
Liivi 2
Tartu, Estonia
sven.laur@ut.ee

ABSTRACT
This paper introduces workflow and methods, which were used in creating a web tool for clustering arbitrary natural language text segments hierarchically and interactively. The purpose of the tool is to give an overview of a collection of texts based on their similarity.

For clustering, we used bag-of-words model along with tf-idf statistical measure for weighting and any preferred method and metric applicable to Euclidean space for calculating distances between clusters.

The paper also discusses generating word-level regular expressions from a set of strings and enhancement of dendrograms with interactivity.

General Terms
Algorithms.

Keywords
Hierarchical clustering, text mining, regular expression generation, sequence alignment, interactive tool, dendrogram.

1. INTRODUCTION
We have feedback from students in natural language. Comments, regarding some specific aspect of an event, are often very similar both in structure and meaning. As we’re interested in extracting relevant, informative and varying sample data, for example to describe an event to future participants, and the data set is large, we would like to be able to automatically select the comments that are different from one another to give the best overview.

Another case would be to match comments to given grades in different categories. For example, let’s say that participants were asked to assess the entertainment, educational and relevance part of the event from 1 to 5. If we could detect the common characteristics of the comments, we could find out which language structures describe positive and negative aspects of an event.

In order to carry out such an analysis, we need a tool which would help us to group similar text segments (tegments from now on) and describe, how the elements are related to one another. Moreover, as humans have insight into the language that computers lack, interaction possibility is necessary to adjust the clusters.

Section 2 covers the basics needed to comprehend the solution, in section 3 we describe the methods, workflow and reasoning behind the solution, in section 4 we cover how to use the tool, in section 5 we discuss the solution and in section 6 ideas for future work.

2. BACKGROUND
2.1 Hierarchical clustering
Clustering is the process of grouping similar objects together. In our context we are talking about tegments. Whenever we talk about clustering, we really want to group tegments based on some similarity measure – often based on common words or meaning. Therefore, to automatically cluster, we need a way to estimate the distance of two arbitrary tegments and to determine from the latter, which tegments should belong to one cluster.

When normal clustering’s goal is to find final groups of tegments, hierarchical (agglomerative) clustering is responsible for showing the evolvement of groups, as the allowed distance between tegments to belong to the same cluster is gradually increased. For that purpose, each tegment is initially in its own cluster. At each step, two of the nearest clusters are merged into one, until just one cluster remains. Such evolvement can be displayed, using a special kind of binary tree – a dendrogram (figure 1).

Result of hierarchical clustering depends largely on three aspects:

1) choice and representation of tegment characteristics;
2) metric for assigning distance score between two nodes;
3) distance calculation method for assigning distance score between two clusters.
2.2 Bag-of-words and tf-idf

Bag-of-words model and tf-idf numerical statistic are two fundamental terms in text mining, which are often used together.

2.2.1 Bag-of-words

Bag-of-words is a way of representing a text by just specifying the words that the text contains along with the words’ multiplicity. It is independent of word order and grammar and therefore has limited use, but sometimes the words carry enough information.

For example, when we are dealing with a tegment that conveys positive emotion, it is likely to contain many positive words, such as “good”, “great”, “happy” etc. However, as informative words are much less frequent than common articles, conjunctions and alike, we face a problem when we want to cluster based on the word frequencies alone. Namely, the most common words contribute the most and therefore bias the clustering process the most. This, however, is unfavourable, because such clustering conveys no meaning. For that reason, weighting is used.

2.2.2 Tf-idf

Tf-idf (term frequency-inverse document frequency) is a composite weight, which indicates, how well a term describes the given document that contains it, by increasing, as the number of occurrences of the term increases within the document, and decreasing, as the number of other documents containing the term increases. The formula for the simplest case is as follows:

\[
\text{tf-idf}_{t,d} = tf_{t,d} \times idf_t,
\]

where \( t \in d \) is the term, \( d \in D \) is the document, \( D \) is the collection of documents, \( tf_{t,d} \) is the term frequency from bag-of-words and \( idf_t \) is often calculated as:

\[
idf_t = \log \frac{|D|}{|[d \in D : t \in d]|}
\]

2.3 Sequence alignment

Let’s suppose we have two character sequences \( s_1 \) and \( s_2 \). Task of sequence alignment is usually to generate two new sequences \( s'_1 \) and \( s'_2 \), such that the number of indices \( i \), where \( s'_{1,i} = s'_{2,i} \), would be as large as possible and maximization can be performed by only adding gaps to the initial sequences.

It is often used in bioinformatics to align genome data or proteins in order to discover relationships between different sequences. The sequences can be from different species, individuals, cells etc. When we have two small pieces of DNA sequences, exact sequence alignment would look something like the following:

\[
\begin{align*}
s_1 &= ACGTATTG \\
s_2 &= AATG \\
\downarrow \\
s'_1 &= ACGTATTG \\
s'_2 &= A---AT---G
\end{align*}
\]

There are different approaches to the problem: correct but rather slow dynamic programming, heuristic algorithms and probabilistic methods.

Sequence alignment can also be used in natural language processing by using general methods that are not necessarily working on character sequences, but rather on arbitrary sequences, which can be encoded if necessary. In our case we might want to align words in tegments.

3. SOLUTION

We introduce a web tool written in HTML5, JavaScript and Python. It allows users to discover and manage similar tegments by visualising the tegment similarities with a dendrogram. It also provides features, such as “cutting” the dendrogram at some internode distance and therefore creating a forest of smaller dendrograms, each of which contained tegments form a cluster.

We can collapse nodes, in which case we get a description of child tegments in the form of a word-level regular expression. It allows to extract common words and their order. It is also possible to add new tegments to the dendrogram in such a fashion that the initial structure of the dendrogram is preserved, along with several minor features and parameter choices we introduce in section 4.

3.1 Clustering text segments

As we recall, when it comes to hierarchical clustering, we need to consider 3 aspects listed in subsection 2.3.

3.1.1 Tegment characteristics

Firstly, we need to agree on the features that we’re interested in, when measuring similarity. In our case we consider two tegments the more similar, the more they share words or lemmas. Lemmas are gained from words in the tegment by lemmatization process, which decides which group of words are mapped to an “original word” or lemma. For example, if we consider a lemma “cluster (verb)” and have a correct lemmatization, it corresponds to words like “cluster”, “clustering”, “clusters” and “clustered”.

Figure 1: Rendered dendrogram from the tool.
We represented our tegments as vectors of term scores, where each term score was equivalent to its corresponding tf-idf weight. We used Python library called scikit-learn\cite{1} for text extraction and tf-idf calculation.

3.1.2 Metric and distance calculation methods
We’ve let users to choose from a list of the most common metrics and distance calculation methods that work in Euclidean space. To name a few, we provide Euclidean and Cosine as metrics and complete, UPGMA and WPGMA as distance calculation methods. The clustering process is powered by SciPy\cite{2} library.

3.2 Visualising
We use dendrogram to display tegments’ similarity. It provides a convenient way for a user to “zoom” to the appropriate level of generalisation. Collapsing a node will zoom out, providing a more general word-level regular expression that matches all the child tegments.

![Dendrogram Example]

**Figure 2:** No generalisation.

![Dendrogram Example]

**Figure 3:** Some generalisation.

![Dendrogram Example]

**Figure 4:** Huge generalisation.

When user wants to display clusters after fixing a “cutting” threshold, clusters are highlighted, using background colour.

Visualisation is done, using embedded SVG in HTML5 and jQuery\cite{3}. SVG provides an option to save it as an image for future reference. Rendering a horizontal tree is straightforward, approaching with recursion and first drawing and positioning the leaves, then connecting them and drawing and positioning their parents etc, until the root is reached. For futher information, `draw_dendrogram()` JavaScript function can be consulted.

3.3 Regular expression generation
For generating a word-level regular expression from a collection of tegments, we use sequence alignment. More precisely, because we may have more than two child tegments for a collapsed node, we have to use multiple sequence alignment, which is said to be NP-complete problem\cite{4}.

Because we may need to handle large amount of tegments, we use a heuristic search known as progressive alignment. It uses pairwise alignment to align two most similar tegments at each step, much like hierarchical clustering, until convergence. We didn’t choose it just for being common, but also, because we already have the subdendrogram representing the very same order according to which the pairwise alignments should be performed.

For pairwise alignment, we used Needleman-Wunsch algorithm, which is based on dynamic programming — much like edit distance — and of which objective function is to maximize the similarity of the sequences.

We’ve managed to reduce the problem to $O(nm^2)$, where $n$ is the number of tegments we want to generalise and $m$ is the largest word count of those tegments. That’s because for $n$-leaf binary tree there are $n-1$ nodes of degree 2, which denotes the number of pairwise alignments, and each alignment is quadratic with respect to the length of the longer sequence.

As we recall, each sequence alignment provides two sequences. However, we expect that at each alignment step we receive just one sequence from both of the children in the dendrogram. To solve that, we have to aggregate the two alignments output by pairwise sequence alignment into one sequence. It can be easily done by retaining word at index $i$, if the two alignments have the same word at $i$, and dash otherwise. We can then remove all but one consecutive dash, as dash is mapped to a wildcard in the regular expression.

3.4 Interaction issues
The most notable issues are caused by adding tegments to an existing dendrogram and rerendering dendrogram when collapsing or expanding the nodes for visual integrity.

3.4.1 Adding tegments
Adding tegments would not be a problem if we could just reclustering. However, as user may have put great effort into naming and determining the appropriate level of generalisation for clusters, we need to retain the structure of the dendrogram.

In case we haven’t defined clusters, we need to link each new tegment to the closest existing node $N$, link the new node $N_{new}$ to $N$’s parent and let $N_{new}$ have $N$’s cluster name. In case we have clusters, we have to do the same thing, but this time we have to consider also, whether we allow the new tegments to be added to the clusters. If we allow, there will be no differences. However, when we don’t, we must not proceed with the search beyond cluster’s border.

In both cases we have to do $O(nm)$ comparisons, where $n$ is the number of existing nodes to which we can link the new nodes and $m$ is the number of new tegments, since for each new tegment we have to find its closest existing node.

Whenever we add a tegment, we need to update all its ancestors’ regular expressions and distances between the two subclusters in the dendrogram to keep the dendrogram consistent.
4. USAGE
The general workflow is as follows:

1) loading a dendrogram;
2) working with the dendrogram;
3) saving the dendrogram.

A snapshot of the tool in production can be downloaded from git repository at:

https://bitbucket.org/komasing/datamining-seminar

In order to run the tool, Python 2.7 must be used, *mongodb* needs to be installed and running, *SciPy*, *scikit-learn* and *CherryPy Python* libraries need to be installed and server.py must be started with the port number, on which the *mongodb* runs, as argument.

4.1 Loading a dendrogram
Dendrogram can be loaded from an uploaded file or file URL. Each text segment has to be on a separate row, rows separated by newline character. Another option is to load from MongoDB database by selecting the corresponding document name from the drop-down list.

If dendrogram is loaded from a file, algorithm, metric and distance calculation methods must be specified.

4.2 Working with the dendrogram
Dendrogram nodes can be selected by either mouse or by first selecting with mouse and then moving, using keys. Left arrow moves to the parent node, up arrow moves to the right child, down arrow moves to the left child. Shift+up/down move on the same height of the tree. Enter is used to collapse/expand nodes and tab is used to switch the children. In addition, when clustering is enabled, Shift+enter allows to change the current cluster’s label.

Tool’s settings’ bar can be seen on figure 5.

By checking "Cluster", dendrogram will show clusters of which subclusters are closer than cluster threshold. Cluster’s name can be changed by clicking on it. It is also possible to choose where to place the added segments – either into clusters or next to them.

Dendrogram can be reloaded with a new structure and distances by providing algorithm, metric and linkage method and pressing "Apply changes". Note that when reloading dendrogram, previous dendrogram will be lost if not saved.

New segments can be added by inserting segments into text area or loading segments from a file. Every segment must be on a different line and is added according to the settings.

When more work space is needed for the dendrogram, the settings’ bar can be minimized.

4.3 Saving the dendrogram
Current dendrogram can be saved to MongoDB NoSQL database by either overwriting the previous entry with "Save" or creating a new entry with "Save as". It is also possible to save the dendrogram as an image. In addition to that, each time some interaction occurs, the dendrogram is automatically saved to browser’s web storage. This allows for the application to survive browser crashes or accidental terminations.

4.4 Performance
The tool has been successfully tested on small data sets (up to 50 segments), where it responded instantly. At 150 segments, rendering each interaction took less than 2 seconds.

5. CONCLUSION
We’ve developed a tool for clustering text segments interactively. It has been proved to work well on small data sets but needs some optimization for larger ones. We’ve also proposed and implemented a solution to generate exact regular expressions from a cluster of segments and enhanced dendrogram with interactivity.

6. FUTURE WORK
During the work, several issues have arose. Currently the generation of regular expressions using sequence alignment is limited to exact matches, meaning that when, for example, 10 segments contain a word in some position, then 11th segment has to contain the word as well in the given position or the word won’t be displayed in the regular expression. This is obviously not the perfect solution – we should instead use a threshold $0 \leq p \leq 1$ which would state, that if a word is present in some position more than $p$ percent of the time, then we consider the word to be frequent enough to be displayed in the regular expression and hence represent the cluster.

In case we have a large existing dendrogram and $n \gg m^2$, $O(nm)$ search complexity might be prohibitive. For that reason a more clever approach might be needed – for example, we could cluster the added segments and link the clusters to the existing dendrogram.

Current dendrogram rendering is relatively slow. Considering the fact that almost every interaction causes dendrogram...
to be redrawn, user has to wait for nearly 2 seconds with just 140 tegments. This could be improved by just recalculating the necessary parts.

Visualising the tegments as a dendrogram is infeasible for huge datasets and sometimes even unnecessary. Sometimes just a simple clustering and prediction method based on the clusters might suffice – for that reason an API, which could manipulate the underlying data structure without the need to constantly redraw the dendrogram, would be useful.

7. ACKNOWLEDGMENTS
We wish to thank everyone who cites our work or gives us kind feedback.

APPENDIX
A. REFERENCES