Efficient Clustering of Web-Derived Data Sets

Luís Sarmento[1], Alexander Kehlenbeck[2], Eugénio Oliveira[1] and Lyle Ungar[3]

[1] Faculdade de Engenharia da Universidade do Porto - DEI - LIACC
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
las@fe.up.pt, eco@fe.up.pt
[2] Google Inc
New York, NY, USA
apk@google.com
[3] University of Pennsylvania - CS
504 Levine, 200 S. 33rdSt, Philadelphia, PA, USA
ungar@cis.upenn.edu

Abstract. Many data sets derived from the web are large, high-dimensional, sparse and have a Zipfian distribution of both classes and features. On such data sets, current scalable clustering methods such as streaming clustering suffer from fragmentation, where large classes are incorrectly divided into many smaller clusters, and computational efficiency drops significantly. We present a new clustering algorithm based on connected components that addresses these issues and so works well on web-type data.

1 Introduction

Clustering data sets derived from the web - either documents or information extracted from them - provides several challenges. Web-derived data sets are usually very large, easily reaching several million of items to cluster and terabyte sizes. More fundamentally, web-derived data sets have specific data distributions, which are not usually found in other datasets, that impose special requirements on clustering approaches. First, web-derived datasets usually involve *sparse, high-dimensional features spaces* (e.g., words). In such spaces, comparing items is particularly challenging, not only because of problems arising from high-dimensionality [1], but also because most vectors in sparse spaces will have similarities close to zero. Also, class distributions of the web-derived data are usually *highly unbalanced* (often Zipfian), with one or two dominant classes and a long tail of smaller classes. This causes a problem for clustering algorithms, which need to be able to deal with such an unbalanced distribution in web-derived data, and still correctly cluster items of non-dominant classes. Additionally, methods to cluster such large data sets have to deal with the fact that "all-against-all" comparison of items is impossible. In practice, items can only be compared to cluster summaries (e.g., centroids) or to only a few other items. The most widely used methods for clustering extremely large data sets are *streaming clustering* methods [2] that compare items against centroids. Streaming clustering has linear computational complexity and (under ideal conditions) modest RAM requirements. However, as we will show later, standard streaming clustering methods are less than ideal for web-derived data because

of the difficulty in comparing items in high-dimensional, sparse and noisy spaces. As a result, they tend to produce sub-optimal solutions where classes are fragmented in many smaller clusters. Additionally, their computational performance is degraded by this excessive class fragmentation. We propose a clustering algorithm that has performance comparable to that of streaming clustering for well-balanced data sets, but that is much more efficient for the sparse, unevenly sized data sets derived from the web. Our method relies on an efficient strategy for comparing items in high dimensional spaces that ensures that only the minimal sufficient number of comparisons is performed. A partial link-graph of connected components of items is built which takes advantage of the fact that each item in a large cluster only needs be compared with a relatively small number of other items. Our method is robust to variation in the distribution of items across classes; in particular, it efficiently handles Zipfian distributed data sets, reducing fragmentation of the dominant classes and producing clusters whose distributions are similar to the distribution of true classes.

2 Streaming Clustering of Web Data

For the purpose of explaining the limitations of streaming clustering for web-derived data sets, we will consider a single pass of a simplified streaming clustering algorithm. This simplification emphasizes the problems that streaming clustering algorithms face, while not changing the basic philosophy of the algorithm. (Later we will show that this analysis can be extended to realistic streaming-clustering approaches.) The simplified version of the streaming clustering algorithm we will be using is:

1. shuffle all items to be clustered and prepare them for sequential access;
2. while there are unclustered items, do:
   (a) take the next unclustered item and compare it with all existing cluster centroids;
   (b) if the distance to the closest centroid is less that min_{dist}, add the item to the closest cluster and update the corresponding centroid;
   (c) otherwise, create a new cluster containing this item only.

For n items to be clustered and if C_f clusters are found, this algorithm performs in O($n\ C_f$) time, since each item is compared with the centroids of C_f clusters, and in O(C_f) space: we only need to store the description of the centroid for each clusters.

The high dimensionality and sparseness of web-derived the data hurt streaming clustering because when comparing two items with sparse features there is a non negligible probability of those items not sharing any common attribute. This is so even when the items being compared belong to the *same class*. Such *false negatives* have a very damaging effect on streaming clustering. If a false negative is found while performing comparisons between an item to be clustered and existing cluster centroids, the streaming clustering algorithm will assume that the item belongs to an yet unseen class. In such cases a new cluster will be created, leading to an artificial increase in the number of clusters generated for each class, with two direct consequences: (i) during streaming, clustered items will have to be compared with additional clusters, which will degrade computational performance in time and space; and (ii) the final clustering result will be composed of multiple clusters for each class, thus providing a fragmented solution.

Whether this degradation is significant or not depends basically on how probable it is to find a false negative when comparing items with existing clusters. Our claim is that on web generated data the probability is in fact quite large since the dimensionality of the spaces is very high and vector representations are very sparse. To make matters worse, fragmentation starts right at the beginning of the clustering process because most items will have nothing in common with the early clusters.

To make a more rigorous assessment of the impact of false negatives on the performance of streaming clustering, let us consider only the items belonging to one specific arbitrary class, class A. In the beginning no clusters exist for items of class A, so the first item of that class generates a new cluster, Cluster 1. The following elements of class A to be clustered will have a non-zero probability of being a false negatives. i.e, of not being correctly matched with the already existing cluster for class A. (We assume for now that there are no false positives, i.e. that they will not be incorrectly clustered with elements of other classes.) In this case a new cluster, Cluster 2, will be generated. The same rationale applies when the following items of class A are compared with existing clusters for that class. We assume that in any comparison, there is a probability p_{fn} of incorrectly mismatching the item with a cluster. Therefore, one expects new clusters for class A to be generated as more items are processed by streaming clustering. This behavior can be modeled by an infinite Markov Chain as depicted in Figure 1. The probability of having created s clusters after performing streaming clustering for n + 1 items is the probability of being in state s (1, 2, 3, ...) of the chain. Assuming independence, as more clusters are generated the probability of creating a new (false) cluster decreases exponentially because that would require more consecutive false negative comparisons. Despite the regularities of this Markov Chain, deriving general expressions for the probability of a given state after n iterations is relatively hard except for trivial cases (see [3]). However, for the purpose of our analysis, we can perform some simplifications and obtain numeric values for comparison. By truncating the size of a chain to a maximum length (s_{max}) and changing the last state of the chain to become an "absorbing state" that represents all subsequent states, numeric computation of state probabilities becomes straight-forward for any value of p. Table 1 shows the most probable state, s_{mp} and its corresponding probability, p_{mp} after clustering 10,000 and 100,000 items (with $s_{max} = 16$) for various values of p_{fn}. As can be easily seen, even for very low probabilities for false negatives ($p_{fn} \leq 0.3$), the chances of replicating the number of clusters several times is considerable. In a realistic scenario, values of $p_{fn} > 0.5$ can easily occur for dominant classes because item diversity in those clusters can be very significant. Therefore, when performing streaming clustering in such conditions, cluster fragmentation of at least one order of magnitude should be expected.

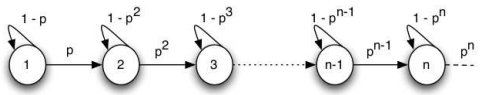

Fig. 1. Markov model for fragmentation in streaming clustering

2.1 Impact on Realistic Streaming Clustering

Actual streaming clustering implementations attempt to solve the fragmentation problems in two ways. The first option is to perform a second pass for clustering the fragmented clusters based on their centroids. The problem with this is that the information that could be used for safely connecting two clusters (i.e., the points in between them) has been lost to centroid descriptions, and these might be too far apart to allow a safe merge since centroids of other clusters may be closer. This situation can more easily occur for *large clusters* in high-dimensional and sparse spaces, where sub-clusters of items might be described by *almost* disjoint sets of features, and thus be actually distant in the hyperspace. Thus, for *web derived data*, re-clustering will not necessarily solve the fragmentation problem, although such an approach is often successful in lower-dimensional and homogeneous datasets. A second variation of streaming clustering algorithms keeps a larger number of clusters than the final target, and alternates between adding more new items to clusters and considering current clusters for merging. However, if each of the items included in the cluster has a sparse representation, and if such "intermediate" clusters have a high level of intra-cluster similarity (as they are supposed to be in order to avoid adding noisy items), then the centroids will probably also have a sparse feature representation. As more items are clustered, each of these many intermediate clusters will tend have only projections in small set of features, i.e. those of the relatively few and *very similar* items it contains. Therefore, feature overlap between clusters will tend to be low, approximately in the same way item feature overlap is low. Such centroids will thus suffer from the same false negative problems as individual items do, and the number of potential clusters to hold in memory may grow large. In practice, unless one reduces the minimum inter-cluster similarity for performing merge operations (which could lead to noisy clusters), this strategy will not lead to as many cluster merging operations as expected, and many fragmented clusters will persist in the final solution. Again, the fragmentation effect should be more visible for larger clusters, in high-dimensional and sparse space.

3 Clustering by Finding Connected Components

It is easy to understand that overcoming the problems generated by false negatives involves changing the way comparisons are made: somehow we need to obtain more information about similarity between items to compensate the effect of false negatives, but that needs to be done without compromising time and space restrictions. Complete

p_{fn}	$s_{mp}(10k)$	$p_{mp}(10k)$	$s_{mp}(100k)$	$p_{mp}(100k)$
0.2	6	0.626	8	0.562
0.3	8	0.588	10	0.580
0.4	10	0.510	13	0.469
0.5	13	0.454	16	0.844
0.6	16	0.941	16	1.000

Table 1. Most probable state of the Markov chain, for 10k and 100k items clustered.

information about item similarity is given by the Link Graph, $\mathcal{G}$, of the items. Two items are linked in $\mathcal{G}$ if their level of pair-wise similarity is larger than a given threshold. The information contained in the Link Graph should allow us to identify the clusters corresponding to the classes. Ideally, items belonging to the same class should exhibit very high levels of similarity and should thus belong to the same connected component of $\mathcal{G}$. On the other hand, items from different classes should almost never have any edges connecting them, implying the they would not be part of the same connected components. In other words, each connected component should be a cluster of items of the same class, and there should be a 1-1 mapping between connected components (i.e. clusters) and classes. Clustering by finding connected-components is robust to the problem of false negatives, because each node in $\mathcal{G}$ is expected to be linked to several other nodes (i.e. for each item we expect to find similarities with several other nodes). The effect of false negatives could be modeled by randomly removing edges from $\mathcal{G}$. For a reasonably connected $\mathcal{G}$, random edge removal should not affect significantly the connectivity within the same connected component, since it is highly unlikely that all critical edges get removed simultaneously. The larger the component, the more unlikely it is that random edge removal will fragment that component because more connectivity options should exist. Thus, for web-derived data sets, where the probability of false negatives is non-negligible, clustering by finding the connected-components of the link graph seems to be an especially appropriate option.

Naive approaches to building $\mathcal{G}$ would attempt an all-against-all comparison strategy. For large data sets that would certainly be infeasible due to time and RAM limitation. However, an all-against-all strategy is not required. If our goal is simply to build the Link Graph for finding the true connected components then we only need to ensure that we make enough comparisons between items to obtain a *sufficiently connected* graph, $\mathcal{G}_{min}$, which has the same set of connected components as the complete Link Graph $\mathcal{G}$. This means that $\mathcal{G}_{min}$ only needs to contain the sufficient number of edges to allow retrieving the same connected components as if a complete all-against-all comparison strategy had been followed. In the most favorable case, $\mathcal{G}_{min}$ can contain only a single edge per node and still allow retrieving the same connected components as in $\mathcal{G}$ (built using an all-against-all comparisons strategy). Since efficient and scalable algorithms exist for finding the connected components of a graph ([4], [5]), the only additional requirement needed for obtaining a scalable clustering algorithm that is robust to the problem of false negatives is a scalable and efficient algorithm for building the link graph. We will start by making the following observation regarding web derived data sets: because the distribution of items among class is usually highly skewed, then for any item that we randomly pick belonging to a dominant class (possibly only one or two) we should be able to rather quickly pick another item that is "similar" enough to allow the creation of an edge in the link graph. This is so even with the finite probability of finding false negatives, although such negatives will force us to test a few more elements. In any case, for items in the dominant classes one can establish connections to other items with vastly fewer comparisons than used in an all-against-all comparison scheme. We only need enough connections (e.g., one) to ensure enough connectivity in order to later retrieve the original complete connected components. For the less frequent items many more comparisons will be needed to find another "similar enough"

item, since such items are, by definition, rare. But since rare items are rare, the total number of comparisons is still much lower than what is required under a complete all-against-all-strategy. We use a simple procedure: for each item keep comparing it with the other items until k_{pos} similar items are found, so as to ensure enough connectivity in the Link Graph. More formally, we will start by shuffling items in set S(n) to obtain $S_{rand}(n)$. Each item in $S_{rand}(n)$ will be given a sequential number i. Then, for all the items starting with i = 0:

1. take item at position i, i_i
2. Set j = 1
3. Repeat until we find k_{pos} positive comparisons (edges)
   (a) Compare item i_i with item i_{i+j}
   (b) Increment j

One can show (Appendix A) that the average computation cost under this "amortized comparison strategy" is:

$$\tilde{O}\left(\frac{n \cdot |C| \cdot k_{pos}}{1 - p_{fn}}\right) \tag{1}$$

with n the number of items in the set, $|C|$ the number of different true classes, p_{fn} is the probability of false negatives and k_{pos} as the number of positive comparisons, corresponding to the number of edges we wish to obtain for each item. This cost is vastly lower than what would be required for a blind all-against-all comparison strategy, without significantly reducing the chances of retrieving the same connected components. Notice that computation cost is rather stable to variation of p_{fn} when $p_{fn} < 0.5$. For $p_{fn} = 0.5$ the cost is just the double of the ideal case ($p_{fn} = 0$), which is comparatively better than values presented in Table 1. One can also show (Appendix A) that the expected value for the maximum number of items that have to be kept in memory during the comparison strategy, n_{RAM} is equal to $E(n_{RAM}) = k_{pos}/(p_{min} \cdot (1 - p_{fn}))$, where p_{min} is the percentage of items of the smallest class. This value depend *solely* on the item distribution for the smallest class and on the probability of false negatives, p_{fn}. If only 0.1% of the elements to be clustered belong to the the smallest class $k_{pos} = 1$, and $p_{fn} = 0.5$ then $E(n_{RAM}) = 2000$. It is perfectly possible to hold information in RAM that many vectors with standard computers. Imposing a hard-limit on this value (for e.g. 500 instead of 2000) will mostly affect the connectivity for less represented classes. Another important property of this strategy is that link graphs produced this way do not depend too much on the order by which items are picked up to be compared. One can easily see that, ideally (i.e., given no false negatives), no matter which item is picked up first, if we were able to correctly identify any pair of items of the same class as similar items, then the link graph produced would contain approximately the same connected components although with different links. In practice, this will not always be the case because false negatives may break certain critical edges of the graph, and thus make the comparison procedure order-dependent. A possible solution for this issue is to increase the number of target positive comparison to create more alternatives to false negative and thus reduce the order dependency.

3.1 Finding Connected Components

Given an undirected graph G with vertices $\{V_i\}_{i=1..N}$ and edges $\{E_i\}_{i=1..K}$, we wish to identify all its connected components; that is, we wish to partition G into disjoint sets of vertices C_j such that there is a path between any two vertices in each C_j, and such that there is no path between any two vertices from different components C_j and C_k. There is a well-known [4] data structure called a disjoint-set forest which naturally solves this problem by maintaining an array R of length N of *representatives*, which is used to identify the connected component to which each vertex belongs. To find the representative of a vertex V_i, we apply the function

```
Find(x) {
 if(R[x] == x) return x;
 else return Find(R[x]);
}
```

starting at i. Initially $R[i] = i$ for all i, reflecting the fact that each vertex belongs to its own component. When an edge connecting V_i and V_j is processed, we update $R[Find(i)] \leftarrow Find(j)$. This naive implementation offers poor performance, but it can be improved by applying both a *rank heuristic*, which determines whether to update via $R[Find(i)] \leftarrow Find(j)$ or $R[Find(j)] \leftarrow Find(i)$ when processing a new edge and *path compression*, under which $Find(i)$ sets each $R[x]$ it ever visits to be the final representative of x. With these improvements, the runtime complexity of a single Find() or update operation can be reduced to $O(\alpha(N))$, where α is the inverse of the (extremely fast-growing) Ackermann function $A(n, n)$ [4]. Since A(4, 4) has on the order of $2^{(10^{19729})}$ digits, the amortized runtime per Find() or or update operation is effectively a small constant.

4 Experimental Setup

We compared the (simplified) streaming clustering (SC) algorithm with our connected component clustering (CCC) approach on artificially generated data-sets. Data-sets were generated with properties comparable to web-derived data, namely: (i) Zipfian distribution of class sizes, with one or two dominant classes; (ii) the number of features associated with each class increases sub-linearly with class size; (iii) the number of non-negative features in each item is Zipfian distributed, and larger for larger classes (items have at least three non-negative features); and (iv) feature distribution inside each class is *lightly Zipfian* (exponent 0.5), meaning that there is a subset of features that occurs more frequently but often enough to make them absolutely discriminant of the class.

Each class has its own set of exclusive features. Therefore, in the absence of noise, items of different classes will never share any feature and thus will always have 0 similarity. Overlap between items of different classes can be achieved by adding noisy features, *shared* by all classes. A given proportion of noise features can be randomly added to each item. To ensure a realistic scenario, we generated a test set with 10,000 items with Zipfian-like item distribution over 10 classes. Noise features were added so that clustering would have to deal with medium level noise. Each item had an additional 30% noise features added, taken from a noise class with 690 dimensions. Noise

features have a moderately decaying Zipfian distribution (exponent 1.0). Table 2 shows some statistics regarding this test set, S_{30}. We show the average number of features per item, avg(#ft), and the average number of noise features per item, avg(#ft$_{noise}$). P_{no} is the probability of not having any overlap between two items randomly picked from a given class (this should be a lower bound for P_{fn}).

4.1 Measures of Clustering Performance

Given a set of $|T|$ test clusters T to be evaluated, and a gold standard, C, containing the true mapping from the items to the $|C|$ classes, we wish to evaluate how well clusters in T, t_1, t_2,...$t_{|T|}$ represent the classes in C, c_1, c_2,... $c_{|c|}$. Ideally, all the items from any given test cluster, t_x, should belong to only one class. Such a t_x cluster would then be considered "pure" because it only contains items of a unique class as defined by the Gold Standard. On the other hand, if items from t_x are found to belong to several gold standard classes, then the clustering algorithm was unable to correctly separate classes. To quantify how elements in test cluster t_x are spread over the true classes, we will measure the *entropy* of the distribution of the elements in t_x over all the true classes, c_y. Let i_{xy} be the number of items from test cluster t_x that belong to class c_y and let $|t_x|$ be the total number of elements of cluster t_x (that can belong to any of the $|C|$ true classes). The *cluster entropy* of the test cluster t_x over all $|C|$ true classes is:

$$e_t(t_x) = \sum_{y=0}^{|C|} -\frac{i_{xy}}{|t_x|} \cdot \ln(\frac{i_{xy}}{|t_x|}) \tag{2}$$

For all test clusters under evaluation we can compute E_t, the *weighted average* of the entropy of each individual test cluster, $e(t_x)$:

$$E_t = \frac{\sum_{x=0}^{|T|} |t_x| \cdot e_t(t_x)}{\sum_{x=0}^{|T|} |t_x|} \tag{3}$$

In the most extreme case, all test clusters would have a single element and be "pure". This, however, would mean that no clustering had been done, so we need to simultaneously measure how elements from the true classes are spread throughout the test clusters. Again, we would like to have all items from a given true class in the fewest test clusters possible, ideally only one. Let $|c_y|$ the the number of items in class c_y.

Class	Items	dim	avg(#ft)	avg(#ft$_{noise}$)	P_{no}	Class	Items	dim	avg(#ft)	avg(#ft$_{noise}$)	P_{no}
1	6432	657	54.14	15.95	0.53	6	187	392	34.70	10.06	0.59
2	1662	556	48.25	14.14	0.56	7	133	366	35.03	10.18	0.58
3	721	493	44.13	12.88	0.568	8	87	334	29.64	8.56	0.58
4	397	448	39.83	11.60	0.589	9	77	325	26.71	7.61	0.61
5	249	413	34.04	9.84	0.57	10	55	300	24.6	7.05	0.61

Table 2. Properties of the test set S_{30}

Then, for each true class, c_y, we can compute the *class entropy*, i.e. the entropy of the distribution of items of such class over the *all* test clusters by:

$$e_c(c_y) = \sum_{x=0}^{|T|} -\frac{i_{xy}}{|c_y|} \cdot \ln(\frac{i_{xy}}{|c_y|}) \tag{4}$$

A global clustering performance figure can be computed as a weighted average over all classes of each individual *class entropy*:

$$E_c = \frac{\sum_{y=0}^{|C|} |c_y| \cdot e_c(c_y)}{\sum_{y=0}^{|C|} |c_y|} \tag{5}$$

Ideally, both E_t and E_c should close to zero as possible, meaning that test clusters are "pure" and that they completely represent the true classes. In the case of a perfect clustering (a 1-to-1 mapping between clusters and classes), both E_t and E_c will be 0.

5 Results

We compared the performance of our connected components clustering (CCC) algorithm with two other algorithms: simplified 1-pass stream clustering (1p-SC) and 2-pass streaming clustering (2p-SC). The simplified 1-pass streaming clustering was described in Section 3 and was included in the comparison for reference purposes only. The 2-pass streaming clustering consists in performing a re-clustering of the clusters obtained in the 1-pass, using information about the centroids of the clusters obtained. The re-clustering is made using the exact same stream-clustering procedure, merging clusters using their centroid information. The 2-pass SC algorithm is thus a closer implementation of the standard streaming clustering algorithm. Each of the algorithms has parameters to be set. For the CCC algorithm we have three parameters that control how the "amortized comparison strategy" is made: (i) *minimum item similarity*, $s_{min_{cc}}$; (ii) *target positive comparisons for each item*, k_{pos}; and (iii) *maximum sequence of comparisons* that can be performed for any item, k_{max} (which is equivalent to the maximum number of items we keep simultaneously in RAM). The k_{pos} and k_{max} parameters was kept constant in all experiments: $k_{pos} = 1$, $k_{max} = 2000$ (see Section 3). The 1-pass SC algorithm has only one parameter, $s_{min_{p1}}$, which is the minimum distance between an item and a cluster centroid to merge it to that cluster. The 2-pass SC algorithm has one additional parameter in relation to the 1-pass SC. $s_{min_{p2}}$ controls the minimum distance between the centroids for the corresponding clusters to be merged together in the second pass. The vector similarity metric used in all algorithms was the Dice metric. Since all algorithms depend on the order of the items being processed, items were shuffled before being clustered. This process (shuffling and clustering) was repeated 5 times for each configuration. All Results shown next report the average over 5 experiments. Figure 2 shows the E_t ("cluster impurity"), E_c ("class dispersion") curves obtained for the three algorithms, using the test set S_{30}. Results were obtained by changing $s_{min_{cc}}$, $s_{min_{p1}}$ and $s_{min_{p2}}$, from relatively high values that ensured almost pure yet fragmented clusters ($E_t \approx 0$ but $E_c >> 0$) to lower values that lead to the generation of less but much

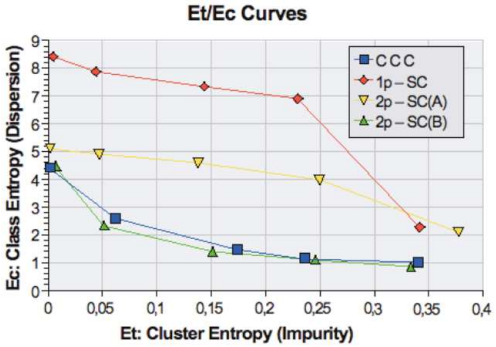

Fig. 2. E_c (y-axis) vs. E_t (x-axis) for four clustering methods. CCC gives better results than most streaming clustering configurations, and is comparable to a carefully tuned streaming method.

noisier clusters ($E_c < 1$ but $E_t >> 0$). We compared the results of the CCC algorithm with results obtained from the 1-pass SC (1p-SC) and two different configuration for the two pass stream-clustering algorithm: 2p-SC(A) and 2p-SC(B). Configuration 2p-SC(A) was obtained by changing $s_{min_{p2}}$ while keeping the value $s_{min_{p1}}$ constant at a level that ensured that the partial results from the first pass would have high purity (yet very high fragmentation). For the configuration 2p-SC(B), we followed for a different strategy for setting parameters: we kept $s_{min_{p2}}$ constant at a medium level, and slowly decreased $s_{min_{p1}}$ to reduce the fragmentation of partial clusters. Configuration 2p-SC(B) was found to the best performing combination among all (several dozens) of configuration tried for the two pass clustering algorithm. We manually verified that, for this test set, values of E_t larger than 0.3 indicate that the clusters produced are mixing items from different classes, so Figure 2 only shows results for $E_t < 0.4$.

We made further comparisons between our CCC algorithm and the *best* performing configuration of the 2p-SC algorithm. Table 3 shows the results of this comparison when aiming at a target value of E_t = 0.15. Relevant criteria for comparing clustering quality are the E_t and E_c values, the number of clusters generated (# clusters) and the number of singleton clusters (# singleton) produced. For comparing computational performance we present the number of comparisons made (# comparisons) and the overall execution time of each algorithm. For 2p-SC we show statistics regarding both the intermediate results (i.e., after pass 1) and the final results (after pass 2), so as to emphasize their relative contributions.

Table 4 shows a typical example of the cluster / true class distribution of the top 10 clusters for the results obtained. (Compare with Table 2). The existence of two or more clusters for Class 1 (and sometimes also for Class 2) was a common result for the 2p-SC algorithm.

6 Analysis of Results

The results plotted in Figure 2 show that the connected components clustering (CCC) algorithm we propose gives clustering qualities very close to those of the best per-

	2p-SC (pass 1)	2p-SC (final)	CCC
E_t	0.08	0.15	0.15
E_c	7.64	1.1	1.53
# clusters	755.4	184	647.6
# singletons	66.4	66.4	478.2
# comparisons	4.2M	74k	2.2M
t (secs.)	142	4	42

Table 3. Comparison between 2p-SC and CCC for target cluster purity $E_t = 0.15$.

forming 2p-streaming clustering approach (2p-SC). Additionally, the CCC algorithm consistently required approximately only *half the number of comparisons* to produce results comparable to the 2p-SC, as the first pass of streaming clustering tends to generate heavy fragmentation (and hence $E_c > 6$). This is especially the case for the relevant part of the E_t / E_c curve ($E_t \leq 0.3$); Thus, we can obtain a significant improvement in computational performance in the regime we most care about. The results in Table 3 suggest that in practice, CCC may have better results than 2p-SC. The E_c (fragmentation) values that the CCC algorithm obtains are worsened by the extremely large tail of singleton or very small clusters that are produced. (These are outliers and items in the end of the buffer that ended up not having the chance to be compared to many others). So, if one were to ignore these smaller clusters in both cases (since filtering is often required in practice), the new corresponding E_c values would become closer.

The question of filtering is, in fact, very important and helps to show another advantage of the CCC for clustering data when processing Zipfian distributed classes on sparse vector spaces. As can be seen from Table 4, 2p-SC failed to generate the single very large cluster for items in Class 1. Instead it generated two medium-size clusters. This type of behavior, which occurred frequently in our experiments for large classes (e.g., 1, 2 and 3), is an expected consequence of the greedy nature of the streaming clustering algorithm. During streaming clustering, if two clusters of the same class happen to have been started by two distant items (imagine, for example, the case of a class defined by "bone-like" hull), greedy aggregation of new items might not help the two corresponding centroids to become closer, and can even make them become more distant (i.e. closer to the two ends of the bone). In high dimensional and sparse spaces, where classes are very large and can have very irregular shapes, such local minima can

	CCC	2p-SC		CCC	2p-SC
Cluster	True Class [#Items]	True Class [#Items]	Cluster	True Class [#Items]	True Class [#Items]
1	1 [6113]	1 [3302]	7	7 [96]	6 [150]
2	2 [1405]	1 [3087]	8	9 [40]	7 [100]
3	3 [582]	2 [1573]	9	4 [38]	8 [68]
4	4 [321]	3 [636]	10	8 [37]	9 [58]
5	5 [170]	4 [323]	11	1 [32]	10 [36]
6	6 [134]	5 [192]	12	10 [30]	2 [18]

Table 4. Typical cluster / true class distribution for target cluster purity $E_t = 0.15$.

easily occur. Thus, if we were to keep only a few of the top clusters produced by 2p-SC (e.g., the top 5), there would be a high probability of ending up only with fragmented clusters corresponding only to the one or two (dominant) classes, and thus loose the other medium-sized, but still important, clusters.

The CCC algorithm we propose, in contrast, is much more robust to this type of problem. CCC tends to *transfer* the distribution of true classes to the clusters, at least for the larger classes, where the chances of finding a link between connected components of the same class is higher. Only smaller classes will be affected by fragmentation. Thus, filtering will mostly exclude only clusters from these smaller classes, keeping the top clusters that should directly match the corresponding top classes. Excluded items might be processed separately later, and since they will be only a small fraction of the initial set of items, more expensive clustering methods can be applied.

7 Related Work

Streaming clustering [2, 6] is one of the most famous classes of algorithms capable of processing very large data sets. Given a stream of items S, classic streaming clustering alternates between linearly scanning the data and adding each observation to the nearest center, and, when the number of clusters formed becomes too large, clustering the resulting clusters. Alternatively, data can be partitioned, each partition clustered in a single pass, and then the resulting clusters can themselves be clustered. BIRCH is another classic method for clustering large data sets. BIRCH performs a linear scan of the data and builds a balanced tree where each node keeps summaries of clusters that best describe the points seen so far. New items to be clustered are moved down the tree until they reach a leaf, taking into account the distance between its features and node summaries. Leafs can be branched when they are over-crowded (have too many items), leading to sharper summaries. BIRCH then applies hierarchical agglomerative clustering over the leaf summaries, treating them as individual data points. The overall complexity is dominated by the tree insertion performed in first stage. A different approach to reducing computational complexity is presented in [7]. In a first stage data is divided into overlapping sets called *canopies* using a very inexpensive distance metric. This can be done, for examples using and inverted index of features. Items under the same inverted index entry (i.e. that share the same feature) fall into the same canopy. In a second stage, an exact - and more expensive - distance metric is used only to compare elements that have been placed in the same canopy. These three last methods process data in two passes, unlike our method which uses only a single pass. None of the other methods deal explicitly with the problem of false negatives, which is crucial in web-derived data. The first two methods also suffer a non-negligible risk of reaching sub-optimal solutions due to their greedy nature. Another line of work aims at finding efficient solutions to the problems arising from high-dimensionality and sparsity, specially those concerned with measuring similarities between items in such spaces [1]. CLIQUE [8] is a density-based subspace clustering algorithm that circumvents problems related to high-dimensionality by first clustering on a 1-dimension axis only and then iteratively adding more dimensions.

In [9], the authors use an approximation to a nearest-neighbor function for very high dimension feature space to recommend news articles, based on user similarity. Instead of directly comparing users, a Locality Sensitive Hashing [10] scheme named Min-Hashing (Min-wise Independent Permutation Hashing) is used. For each item i_j (i.e. user) in the input set $\mathcal{S}$, the hash function $H(i_j)$ returns the index of the first non-null feature from the corresponding the feature vector (corresponding to a click from the user on a given news item). If random permutations of feature positions are performed to $\mathcal{S}$, then it is easy to show ([11], [10]) that the probability of two items hashing to the same value, $H(i_j) = H(i_k)$ is equal to their Jaccard coefficient $J(i_j, i_k)$. Min-hashing can thus be seen as a probabilistic clustering algorithm that clusters together two items with a probability equal to their Jaccard Coefficient. The hash keys for p different permutations can be concatenated so that two item will converge on the same keys with probability $J(i_j, i_k)^p$, leading to high-precision, yet small, clusters. Repeating this process for a new set of p permutations will generate different high-precision clusters, giving increased recall. For any item i_j it is possible to obtain the list of its approximate nearest-neighbors by consulting the set of clusters to which i_j was hashed. Since clusters produced by min-hashing are very small, it will produce extremely fragmented results when directly used for clustering large data sets. It could, however, potentially be used as an alternative technique for building the link graph because it provides a set of nearest neighbors for each item. However, there is no assurance that the link graph thus created would contain the complete connected components. Clusters extracted from that graph could thus be very fragmented.

8 Conclusion and Future Work

We have seen that the Zipfian distribution of features and of feature classes for problems such as web-document clustering can lead to cluster fragmentation when using methods such as streaming clustering, as individual items often fail to share any features with the cluster centroid. (Streaming clustering using medoids, as is often done in the theory literature, would be much worse, as most items would fail to intersect with the medoid.) Connected component clustering does a better job of addressing this problem, as it keeps searching for items close to each target item being clustered until they are found. This is not as expensive as it sounds, since it will be easy to find connected items for the many items that are in large classes. We showed that a reasonably connected link graph can be obtained using an item comparison procedure with cost amortized to $O(n \cdot C)$. We showed that the performance of our algorithm is comparable to best performing configurations of a streaming clustering approach, while consistently reducing the number of comparisons to half. Another important characteristic of our algorithm is that it is very robust to fragmentation and can thus *transfer* the distribution of true classes in the resulting clusters. Basically, this means that the top largest clusters will represent the top largest classes, which is fundamental when filtering is required. The above work has described the clustering as if it were done on a single processor. In practice, web scale clustering requires parallel approaches. Both stages of our algorithm (the amortized comparison procedure and procedure for finding the connected components on the graph) are specially suited for being implemented in the Map-Reduce paradigm

[12]. Future work will focus on parallel implementation of our algorithm using the Map-Reduce platform and studying its scalability and performance.

Acknowledgments

This work was developed while Luís Sarmento was an *engineering intern* and Lyle Ungar was a *visiting researcher* at Google offices in NYC. The authors would like to thank the Google team for all the help and support. Also, special thanks to Paula Milheiro (University of Porto, Portugal) for valuable discussions regarding Markov chains.

A Demonstrations

Consider the set of $\mathcal{I}$ containing $|\mathcal{I}|$ *items* that belong to C classes c_1, c_2, c_3,... c_C. Let p_{ji} be the probability of an item (or element) e_j randomly picked from $\mathcal{I}$ belonging to class c_i: $P(e_j \in c_i) = p_{ji}$ with 1 < i < C. Now consider the problem of sequentially comparing items in $\mathcal{I}$ (previously shuffled) in order to find items similar to the initial (target) item. If we randomly pick one item e_j from $\mathcal{I}$, we wish to estimate the number of additional items that we need to pick (without repetition) from $\mathcal{I}$ before we find another item that belongs to the same class. For a sufficiently large set of items the probabilities $P(e_j \in c_i)$ do not change significantly when we pick elements out of $\mathcal{I}$ without replacement, and we can consider two subsequent draws to be independent. We can thus make $P(e_j \in c_i) = p_i$ and approximate this procedure by a *Bernoulli Process*. Therefore, for a given element of class c_i, the number of comparisons k_i needed for finding a similar item follows a *Geometric Distribution* with parameter, p_i. The expected value for k is $E(k_i) = \frac{1}{p_i}$. For C classes, the average number of comparisons is:

$$E(k) = \sum_{c=1}^{|C|} p_c \cdot E(k_c) = \sum_{c=1}^{|C|} p_c \cdot \frac{1}{p_c} = |C| \tag{6}$$

For sufficiently large $|\mathcal{I}|$, the number of classes will remain constant during almost the entire sampling process. Thus, the total number of comparisons for the $|\mathcal{I}|$ items is: $N_{comp} = |\mathcal{I}| \cdot |C|$. If we extend the previous item comparison procedure to find k_{pos} similar items to the target item,n we can model the process by a *Negative Binomial Distribution* (or *Pascal Distribution*) with parameters p_i and k_{pos}:

$$B_{neg}(k_i, k_{pos}) = \binom{k_i - 1}{k_{pos} - 1} \cdot p_i^{k_{pos}} \cdot (1 - p_i)^{k_i - k_{pos}} \tag{7}$$

In this case, the average number of comparisons made, given by the corresponding Expected Value is: $E_{B_{neg}}(k_i, k_{pos}) = k_{pos}/p_i$. The longest series of comparison wills be made for the class with the lowest p_i, i.e. the small class. However, it lead us to a average number of comparisons when considering all the $|C|$ of classes of:

$$E_{comp}(k) = \sum_{c=1}^{|C|} p_c \cdot E_{B_{neg}}(k_c, k_{pos}) = k_{pos} \cdot |C| \tag{8}$$

For all $|\mathcal{I}|$ items we should thus have $N_{comp} = |\mathcal{I}| \cdot |C| \cdot k_{pos}$. If we now consider that there a probability of p_{fn} of having a false negative when comparing two items, and that p_{fn} is constant and independent of classes, the p_i should be replaced by $p_i \cdot (1 - p_{fn})$, i.e. the probability of a random pick finding another item in class c_i has to be multiplied by the probability of not having a false negative. Then all the above equations will change by a constant factor, giving:

$$N'_{comp} = \frac{|\mathcal{I}| \cdot |C| \cdot k_{pos}}{1 - p_{fn}} \tag{9}$$

Likewise, the expected value for longest series of comparisons will be given by performing the same substitution in Equation 10, and making $p_i = p_{min}$:

$$E_{ls} = \frac{k_{pos}}{p_{min} \cdot (1 - p_{fn})} \tag{10}$$

References

1. Aggarwal, C., Hinneburg, A., Keim, D.: On the Surprising Behavior of Distance Metrics in High Dimensional Spaces. Proceedings of the 8th International Conference on Database Theory (2001) 420–434
2. Guha, S., Meyerson, A., Mishra, N., Motwani, R., O'Callaghan, L.: Clustering Data Streams: Theory and Practice. IEEE Transactions on Knowledge and Data Engineering **15**(3) (2003) 515–528
3. Samuel-Cahn, E., Zamir, S.: Algebraic characterization of infinite markov chains where movement to the right is limited to one step. Journal of Applied Probability **14-14** (December 1977) 740–747
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press and McGraw-Hill Book Company (1990)
5. Hopcroft, J., Tarjan, R.: Algorithm 447: efficient algorithms for graph manipulation. Commun. ACM **16**(6) (1973) 372–378
6. Charikar, M., O'Callaghan, L., Panigrahy, R.: Better streaming algorithms for clustering problems. In: STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing, New York, NY, USA, ACM (2003) 30–39
7. McCallum, A., Nigam, K., Ungar, L.H.: Efficient clustering of high-dimensional data sets with application to reference matching. In: KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, New York, NY, USA, ACM (2000) 169–178
8. Agrawal, R., Gehrke, J., Gunopulos, D., Raghavan, P.: Automatic subspace clustering of high dimensional data for data mining applications. SIGMOD Rec. **27**(2) (1998) 94–105
9. Das, A.S., Datar, M., Garg, A., Rajaram, S.: Google news personalization: scalable online collaborative filtering. In: WWW '07: Proceedings of the 16th international conference on World Wide Web, New York, NY, USA, ACM (2007) 271–280
10. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proc. of 30th STOC. (1998) 604–613
11. Broder, A.Z.: On the resemblance and containment of documents. In: SEQS: Sequences '91. (1998)
12. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI'04: Sixth Symposium on Operating System Design and Implementation. (2004)