Efficient Hierarchical Clustering of Large High Dimensional Datasets

Sean Gilpin
University of California, Davis
sagilpin@ucdavis.edu

Buyue Qian
IBM T. J. Watson
bqian@us.ibm.com

Ian Davidson
University of California, Davis
davidson@cs.ucdavis.edu

ABSTRACT

Hierarchical clustering is extensively used to organize high dimensional objects such as documents and images into a structure which can then be used in a multitude of ways. However, existing algorithms are limited in their application since the time complexity of agglomerative style algorithms can be as much as $O(n^2 \log n)$ where n is the number of objects. Furthermore the computation of similarity between such objects is itself time consuming given they are high dimension and even optimized built in functions found in MATLAB take the best part of a day to handle collections of just 10,000 objects on typical machines. In this paper we explore using angular hashing to hash objects with similar angular distance to the same hash bucket. This allows us to create hierarchies of objects within each hash bucket and to hierarchically cluster the hash buckets themselves. With our formal guarantees on the similarity of objects in the same bucket this leads to an elegant agglomerative algorithm with strong performance bounds. Our experimental results show that not only is our approach thousands of times faster than regular agglomerative algorithms but surprisingly the accuracy of our results is typically as good and can sometimes be substantially better.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data mining

General Terms

Algorithms, Experimentation

Keywords

Binary Codes, Hierarchical Clustering

1. INTRODUCTION

Hierarchical clustering is used extensively to organize documents [18, 17, 5], images [2] and even graphs [13] into a hierarchy. The hierarchical organization of these objects has many advantages. The resultant hierarchy can be "cut" at any level to achieve a $k = 1 \ldots n-1$ partitional clustering or used to navigate the collection. Consider a collection of images organized into a hierarchy. We can easily return as few or as many related images to a given query image by navigating the tree. In particular hierarchies of complex high dimensional objects such as images and documents are common since they are most in need of organizing.

The flexibility of hierarchical agglomerative clustering (HAC) comes with a computational cost. Just to calculate the distance function used by all agglomerative methods requires $O(n^2)$ distance computations and each join requires additional computation. The well optimized CLINK algorithm [3] that uses complete linkage distance function has time complexity $O(n^2 \log n)$. Even using the simpler to calculate single linkage distance yields a quadratic time complexity of $O(n^2)$ [14]. To scale hierarchical clustering to large datasets, a linear time complexity is required. Furthermore for high dimensional data not only is the number of pairwise distance calculations great, but just a single distance calculation can be time consuming. For high dimensional objects[1] with features that are physical recordings and lie in a positive space, it is well known that the cosine distance function produces preferable results, [15, 1] but it requires an inner-product calculation in the high dimensional space. Therefore efficient and clever computation of hierarchies of high dimensional objects offers a considerable promise of immense speedup.

In this paper we explore the novel direction of using angular hashing to compute a collection of hash buckets to reduce the number of computations. A hash function simply takes an object in high dimensional space and maps to a hash code (a binary string) and corresponding hash bucket. Angular hashing involves creating a hash bucket so that the distance (in terms of angle) between any two objects in the same bucket is small. This means the closest point to any given point typically lies in its hash bucket. Hence, after angular hashing occurs we can perform agglomerative clustering on the instances inside each hash bucket and then perform agglomerative clustering on the hash buckets themselves using their binary codes to compute distances. This is only possible since our results show that the Hamming distance between the hash codes are representative of the angular distances between the instance at the center of each bucket.

[1] In this paper high dimensional data refers to objects with thousands+ of dimensions.

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2505515.2505527.

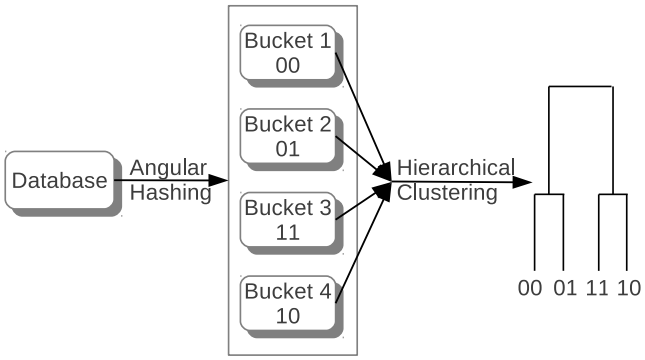

Figure 1: A high level overview of our two step approach. After the instances are hashed to buckets we hierarchically cluster the buckets (using their hash codes) and then hierarchically cluster the points within each bucket. Lemma 1 and Theorem 1 shows why this is principled including that the Hamming distance between hash codes approximates cosine distance.

Our work has several advantages and contribute the following to the field:

- Our algorithm runs in linear time and space (see Section 2, and Figure 5). Hence for large datasets the performance improvement can be 10,000 fold faster (see Figures 11 and 12) than traditional algorithms and can create hierarchies of 60,000 images (see Figure 7) in less than a minute.
- Our algorithm produces comparable results and for some distance functions even better results than standard HAC algorithms (see Figures 8, 9).
- The relationship between Hamming and cosine distance makes our algorithm applicable to many types of datasets of high dimension. [1, 15].
- We produce several interesting bounds (see Lemma 1, Theorem 1) that can be used to guarantee the composition of the hierarchy and intelligently navigate the hierarchy.

The rest of the paper is organized as follows. In Section 1.1 we explain the relevant previous work, and most importantly the results from angular based quantization which are utilized in our work (see Equations 1 and 2). In Section 2 we give the details of our algorithm. Section 2.1 first introduces the algorithm and shows how it can be used to build a hierarchy of hash buckets, while in Section 2.2 we extend the algorithm so that it can be used to efficiently create hierarchies of greater detail in a recursive manner. In Section 2.3 we present our novel bounds that relate to the effectiveness of the binary code approximations we use with respect to our clustering algorithm and finally in Section 3 we empirically show that our algorithm not only scales well, but that it can have as good as or better clustering results for both image and document datasets.

1.1 Literature Review

Hierarchical clustering has its roots in numerical and mathematical taxonomy [7, 4] with a primary purpose being to organize flora and fauna into a taxonomy. Therefore the traditional agglomerative and divisive algorithms were not designed to scale. Consider agglomerative algorithms, naively, at each level $O(k^2)$ distance calculations must be performed to find the closest clusters to join leading to the series $n^2 + (n-1)^2 + (n-2)^2 \ldots 2^2 \approx n^3$ calculations. Two popular distance functions are single linkage (the distance between two clusters is the closest pair's distance such that one point is in each cluster) and complete linkage (the distance being the furthermost pair's distance). By reusing prior calculations and properties of the single linkage and complete linkage distance functions the SLINK and CLINK algorithms time complexities of $O(n^2)$ and $O(n^2 \log n)$ are achieved. More recent larger applications has pushed the need for even more efficient algorithms at the cost of finding the closest pair of clusters at each level. Since the agglomerative approach is in itself greedy and not guaranteed to converge to any global optima this strategy is worth pursuing. We now overview the general directions of scaling hierarchical clustering.

Hashing and Agglomerative Clustering. There has been some previous work on speeding up hierarchical clustering using hashing. One previous work that is only specific to single linkage [8] used locality sensitive hashing nearest neighbor search to speed up single linkage clustering. Although the purpose of their work was to speed up hierarchical clustering it has several limitations: 1) The authors do not show analytically under what conditions the method will scale and the largest data-set they evaluated empirically had only 6,000 points (which takes a reasonable 90 minutes using MATLAB's standard functions on the 20-Newsgroup dataset), 2) It is built around just one distance function (single linkage). In our work we show that our algorithm has provable worst case linear time and space use (See Section 2.2) and in the experimental section we empirically show that our runtime scales to allow our method to run on very large datasets (See Figures 8 and 9). Furthermore, our experiments show our method can be applied to single, complete linkage and other distance functions.

Using Cheap Secondary Distance Functions. The canopy method [11] cleverly exploits a fast secondary distance function, that can be used to prune the number of nearest cluster computations, and is applicable to any clustering algorithm that uses distances as inputs. The canopy method is highly dependent on a cheap similarity measurement used to create the canopies and the angular based codes in this paper could be used to create a cheap distance measurement. However, although very useful, the canopy method is limited in that no theoretical guarantees are given and performance is highly dependent on the canopy parameters for which it is not always clear how to set. Our work does not have these limitations. Section 2 shows that our algorithm uses provable worst case linear time and space and the only parameter needed for our method, the length of binary codes, is shown in Figure 10 to improve the performance in terms of Rand-Index for increasing lengths.

Angular Quantization based Binary Codes for Fast Similarity Search. Previous work on angular quantization hashing was limited to similarity search [6] and not applied to hierarchical clustering. The former work shows that for positive data-sets, binary codes can be constructed such that their Hamming distance can approximate cosine distance. These codes can be constructed for an entire data-set in linear time and in our experiments for even 60,000 image

datasets take no more than 30 seconds. Our work builds upon the idea of angular hashing and our novel contributions are: i) The application of angular hashing to our algorithm for hierarchical clustering (see Figure 2 for an overview of our algorithm), ii) The performance guarantee (see Theorem 1) is novel and describes the effectiveness of the binary code approximation for our problem.

The main results from [6] that we use are shown in Equations 1 and 2. The integer program for finding the binary approximation is shown in Equation 1 and is used to find the angularly closest binary landmark to a point $\mathbf{x}$. In general integer programs can be intractable, but this problem can be solved exactly using a simple greedy algorithm which takes $O(d \log d)$ time and space (where d is the number of dimensions). Therefore these binary code approximations can be calculated for an entire dataset in linear time and space with respect to the number of points n.

$$\hat{\mathbf{b}} = \underset{\mathbf{b}}{arg\ max} \frac{\mathbf{b}^T \mathbf{x}}{||\mathbf{b}||_2} \ s.t.\ \mathbf{b} \in \{0,1\}^d \tag{1}$$

One limitation of the integer program in Equation 1 is that it can only find binary codes of length d (i.e. the number of dimensions in the binary codes must be the same as the number of dimensions in the original points). By introducing a projection matrix $\mathbf{R} \in \mathbb{R}^{d \times c}$ we can simultaneously control the size of our binary codes through the number of columns c, but we can also do a better job of mapping our points to binary codes of higher quality (see Lemma 1 to see how binary codes with greater Hamming length can produce higher quality results). Equation 2 shows the optimization problem that allows us to control the length of our binary codes and simultaneously rotates the original points so that binary codes of larger Hamming length are found.

$$\begin{aligned} \mathcal{Q}(\mathbf{B}, \mathbf{R}) &= \sum_{i=1}^{n} \underset{\mathbf{B},\mathbf{R}}{arg\ max} \frac{\mathbf{b}_i^T}{||\mathbf{b}_i||_2} \mathbf{R}^T \mathbf{x}_i \\ &\textit{subject to:} \\ &\mathbf{b} \in \{0,1\}^c \\ &\mathbf{R}^T \mathbf{R} = \mathbf{I}_c \end{aligned} \tag{2}$$

Equation 2 can be approximately solved using alternating optimization by finding the best rotation, and then the binary code approximation of those rotated points. The additional step of calculating the best projection matrix $\mathbf{R}$ can be done in linear time and space, because it only requires finding the first c singular vectors the $d \times c$ matrix $\mathbf{XB^T}$ (see [6] for details of optimization problem).

2. OUR ALGORITHM

We begin this section by overviewing our algorithm and in later subsections provide more detail. Our algorithm gains its efficiency in two ways: 1) using Hamming distance of binary codes is faster than calculating cosine distance between floating point vectors, and 2) we use the binary codes to dynamically coarsen the original problem into a collection of subproblems that can be handled efficiently. At a high level, our algorithm works by creating hash buckets corresponding to each unique binary code, and hierarchically clustering first the hash buckets and then the set of hash points mapped to each hash bucket. Section 2.1 describes how to choose the length of binary codes so that clustering hash buckets / unique binary codes are guaranteed to be efficient without sacrificing the quality of the hierarchy. In Section 2.2 we describe how the points within each hash bucket can be efficiently hierarchically clustered as well. Figure 2 gives a high level overview of our algorithm. In steps 1 and 6 of the figure, algorithmic choices are provided that are discussed in Sections 2.1 and 2.2 respectively.

The technique we describe can be used with any agglomerative hierarchical clustering algorithm (e.g. single / complete / average linkage), however it is designed to be only used with cosine distance, for which we use Hamming distance to more efficiently approximate angular distances between points.

Define BCA-HAC
Input $\mathbf{X} \in \mathbb{R}^{n \times d}$ (points)
Output $\mathbf{T}$ (dendrogram)
Begin

1. Choose number of bits c for binary codes.
   (a) Set $c = min(\lfloor \log_2(\sqrt{n}) \rfloor, d)$ number of bits (see Section 2.1.1).
   OR
   (b) *Adaptively* choose c (see Section 2.1.2).
2. Calculate binary codes $\mathbf{B}$ by solving optimization problem in Equation 2.
3. Find the unique set of binary codes $\mathbf{B}^*$ which will be used as hash buckets.
4. Hierarchically cluster $\mathbf{B}^*$, resulting in dendrogram whose leaf nodes are $\mathbf{T}_i$.
5. Set $\mathcal{B}_i$ to be the set of instances that have binary code equal to the i^{th} binary code in $\mathbf{B}^*$ (i.e. the points in the i^{th} hash bucket).
6. Attach points from sets $\mathcal{B}_i$ to associated leaf nodes $\mathbf{T}_i$ of hierarchy created in previous step:
   (a) Attach points in $\mathcal{B}_i$ flatly (See Section 2.1).
   OR
   (b) Recursively apply BCA-HAC to points $\mathcal{B}_i$ using unused portions of binary codes (See Section 2.2).

End

Figure 2: The Binary Code Approximation - Hierarchical Agglomerative Clustering (BCA-HAC) algorithm. Section 2.1 discusses the the different methods for choosing the length of binary codes used in Step 1. Section 2.2 discusses Step (6b).

2.1 Base Method

In this section we show how to efficiently calculate a hierarchy of $\sqrt{n}$ hash buckets in linear time and space with respect to the total number of points. In terms of Figure 2, **in this section we will ignore option (6b)** which can used to increase the detail of our hierarchy. We will however discuss the differences between choices (1a) and (1b), which

are alternative methods to choose the number of bits for the binary codes. As the algorithm in 2 shows, we can create a hierarchy out of a set of hash buckets. This can be done efficiently if the number of of hash codes is small enough which can be controlled by adjusting the length of the binary codes created. In section 2.1.1 we discuss how to limit the number of hash buckets, and in section 2.1.2 we discuss a method of creating binary codes so that the number of hash buckets is upper and lower bounded.

2.1.1 Static Methods of Creating Hash Codes

Here we show the simplest variant of our algorithm to implement where the length of the binary codes are computed apriori. If we can create $\sqrt{n}$ or less unique binary codes / hash buckets, then standard HAC algorithms such as SLINK [14] are capable of hierarchically clustering them in linear time with respect to the original number of points n, because $O(\sqrt{n}^2) = O(n)$. In this section we do not recursively apply the algorithm to cluster each hash bucket but the problem of doing that in linear time and space is also dicussed in Section 2.2. **It should be noted that in all our experimental results we use the later adaptive method in section 2.2 to build a complete tree.** We can easily find a binary codes approximations with less than or equal to $\sqrt{n}$ unique codes by setting the number of bits to $c = min(\lfloor \log_2(\sqrt{n}) \rfloor, d)$, for which the total possible number of unique binary codes is less than $\sqrt{n}$. Formally:

$$\begin{aligned} 2^c &= 2^{min(\lfloor \log_2(\sqrt{n}) \rfloor, d)} \\ &\leq 2^{\lfloor \log_2(\sqrt{n}) \rfloor} \leq 2^{\log_2(\sqrt{n})} = \sqrt{n} \end{aligned} \tag{3}$$

As discussed in the previous work [6], calculating the binary codes for a set of points can be done in linear time and space. Also we can efficiently associate the original points to the appropriate hash buckets (i.e. Step (6a)) in linear time with respect to the number of points. For example, this can be done by creating a binary tree of the binary codes associated with each hash bucket. Such a binary tree which would have maximum depth c, the length of the binary codes, so that the total time to create sets of points belonging to each hash bucket is $O(nc)$ time and space.

The disadvantage of choosing the number of bits using this closed form equation is that although it provides an upper bound on the number of unique binary codes corresponding to hash buckets, it does not provide a lower bound. A larger number of binary codes creates a larger degree of freedom for the hierarchical clustering algorithm to choose the hierarchical structure of the points, and so can be desirable from that point of view. The following section describes a method for creating binary codes in a way that will produce a number of hash buckets with an upper and lower.

2.1.2 Adaptive Methods of Creating Hash Codes

In this section we describe a technique for choosing the binary code length so that there is not only an upper bound on the number of unique binary codes but also a lower bound (i.e. $|\mathbf{B}^*| = \Theta(\sqrt{n})$. The trick to finding such a set of codes is to search for it by iteratively expanding the length of codes and efficiently checking the number of unique codes/bins. Step (2d) allows us to skip searching unnecessary code lengths by finding the minimum number of additional bits required to meet the lower bound.

Define Adaptive-BC-Length
Input $\mathbf{X} \in \mathbb{R}^{n \times d}$ points
Output $c, \mathbf{B} \in \mathbb{B}^{\mathbf{c}}$ such that $|\mathbf{B}^*| = \Theta(\sqrt{n})$
Begin

1. Set $c = \lceil \log_2(\sqrt{n}) \rceil$
2. **do**
   (a) Solve optimization problem to find $\mathbf{B}$
   (b) Calculate number of unique codes $\mathbf{B}^*$
   (c) $\Delta_c = \log_2(2\sqrt{n} - |\mathbf{B}^*|)$
   (d) **if** $\Delta_c \geq 1$ **then** set $c = c + \Delta_c$
3. **while**$(\Delta_c \geq 1)$

End

Figure 3: Description of algorithm to find number of bits in binary codes that will yield unique number of binary codes $|\mathbf{B}^*| = \Theta(\sqrt{n})$.

A simple approach is shown in Figure 3. Because checking the number of hash buckets can be done using only linear time and space (see Section 2.1.1), and because there are only a constant number of possible code lengths to check (with respect to number of points), this method is also guaranteed to run using linear time and space.

2.2 Efficiently Computing Subtrees

The above described method works to create a hierarchy of $\Theta(\sqrt{n})$ hash buckets with all of the points in the same hash bucket connected to the tree in a flat manner (i.e. they are all merged to the leaf corresponding to the hash bucket they belong to at the same time). If we are interested in further developing additional detail in the hierarchy by creating subtrees for each hash bucket, there can easily be a large amount of work needed. For example, if the sets of points in the same hash buckets $\mathcal{B}_i$ are all of equal size then there will be $O(\frac{n}{\sqrt{n}})$ points in each and performing standard HAC will take at minimum $O((\frac{n}{\sqrt{n}})^2) = O(n)$ time. That complexity may be acceptable if only a few of the hash buckets need to be expanded into subtrees on a per need basis, but if all hash buckets need to be expanded then it will take $O(n\sqrt{n})$ time in the best case (uniform set sizes) which can be unacceptable for large n.

In this section we will describe an efficient method that allows us to recursively apply our algorithm to create subtrees, without calculating new sets of binary codes. Instead we will create one set of binary codes with more bits than previously described, and use different parts of the binary codes on different levels of the recursive call tree. Figure 4 shows an example of how our method can be applied recursively on a small example.

In order to apply our method we need longer binary codes. If for example we only need c_1 bits to create binary codes with $O(\sqrt{n})$ unique values, we will need an additional c_2 bits so that each set $\mathcal{B}_i$ can be hierarchically organized into a tree of $\sqrt{\sqrt{n}} = \sqrt[4]{n}$ hash buckets / leaf nodes. In the case of

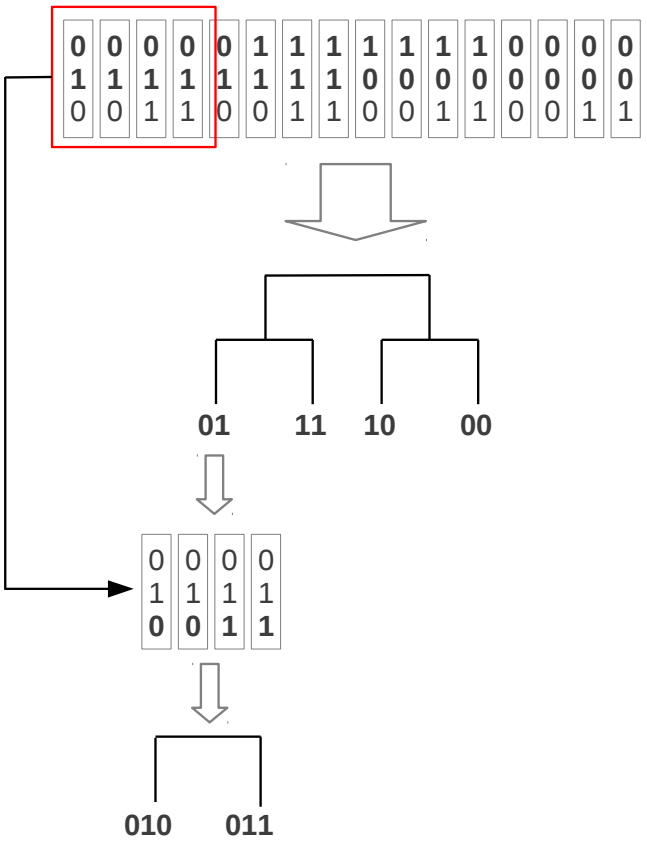

Figure 4: Visual example of our recursive method. The original 16 points can be hashed to $\sqrt{16} = 4$ buckets which requires binary codes of length 2. Our recursive method uses these as prefixes and uses the additional bits to further cluster points in each of the buckets.

binary codes that produce balanced hierarchies, recursively applying our method r times will produce a tree with the following number of leaf nodes:

$$\sqrt{n} * \sqrt[4]{n} * \ldots * \sqrt[2^r]{n} = n^{1-\frac{1}{2^r}} \quad (4)$$

In the more general case the number of leaf nodes will simply be the number of unique binary codes created $|\mathbf{B}^*|$. We will also generally expect that the number of bits needed to extend will decrease by roughly half at each recursive level (i.e. $c_i < c_{i+1}$ and $\frac{1}{2}c_i \approx c_{i+1}$) because $\log_2(n^{1/2^r}) = \frac{1}{2}\log_2(n^{1/2^{r-1}})$. Figure 5 shows the algorithm that formalizes this intuition.

The worst case time and space for the recursive algorithm is linear with respect to the total number of points. We have already shown that we can adaptively calculate binary codes with desired number of unique codes using only linear time and space. The recursive call tree has depth at most d and the the sum of points in the sets $\mathcal{B}_i$ passed into the function BCA-HAC at a specific level in the recursive hierarchy is at most n. The amount of time and space needed for each call to BCA-HAC is linear with respect to $|\mathcal{B}|$ (see Section 2.1), and therefore each level of the recursive tree takes linear time, and given the constant number of recursive levels

---

Input ℓ (desired number of leaf nodes) X (points)
Output T (tree structure where leaf nodes are binary codes).
Begin

1. Adaptively create binary codes $\mathbf{B}$ such that the number of bits c produces $\ell \leq |\mathbf{B}^*| \leq 2\ell$
2. Create set of all points indices $\mathcal{B}$.
3. Call `BAC-HAC`$(1, \mathcal{B})$
4. **define function** `BAC-HAC`(s (first bit), $\mathcal{B}$ (point indices))
   (a) Set $m = |\mathcal{B}|$
   (b) Find c so that $\sqrt{m} \leq |\mathbf{B}^*_{\mathcal{B},s:c}| \leq \sqrt{2m}$
   (c) Create hierarchy T from $\mathbf{B}^*_{\mathcal{B},s:c}$ using standard HAC and create sets. of point $\mathcal{B}_i$ associated with each leaf node T_i.
   (d) **if (** $c > d$ **) return** T //base case
   (e) **foreach** $\mathcal{B}_i$
      i. $replace(T_i, \texttt{BAC-HAC}(c+1, \mathcal{B}_i))$

End

Figure 5: Algorithm for recursively applying BCA-HAC using one set of binary codes. The method is defined recursively to simplify the presentation, but it is very efficient because the same binary codes are used in each recursive call. When moving down the recursive call tree, prefixes of increasing size are ignored until the entire binary code has been consumed (base case). The notation $\mathbf{B}^*_{\mathcal{B},s:c}$ describes the unique codes when considering bits s through c, and only the codes with indices from $\mathcal{B}$.

only linear time and space is needed for the entire recursive method.

2.3 Some Useful Bounds

Though our work uses the idea of angular hashing [6] all of the following results are novel. These results are important because our method relies on hierarchically clustering hash buckets and these results show show how well the hash buckets will approximate the original points. Lemma 1 is necessary for proving Theorem 1, and it describes an upper bound on the angle between two points in the same bucket. This is a desirable property because it guarantees that two points in the same hash bucket cannot be too dissimilar. Figure 6 plots this bound for increasing hamming length (number of non-zero entries or $||\mathbf{b}||_1$). As the plot shows, hash buckets with larger hamming length have better theoretical guarantees with respect to how angularly close points in the same hash bucket will be.

Theorem 1 is an important result because it explains how the sizes of the different hash buckets relate to each other. The angular size of the cone corresponding to a hash bucket can vary depending on the hamming length of its binary code. A cause for concern is that the sizes of the hash

buckets could be completely disproportionate to each other, which would lead to irregularities in how points are mapped to hash buckets. Specifically what we would like to avoid is the scenario where some areas of the instance space map many points to a single hash bucket even though they are not a dense cluster of points, while in other areas of the instance space an equally dense set of points would be mapped to multiple (greater than two) hash buckets. Theorem 1 guarantees that this scenario cannot occur and can be roughly interpreted to guarantee that the largest hash bucket is less than twice the size of the smallest hash bucket.

Lemma 1. ***The intra hash bucket maximum angular distance.*** *Given two instances* $\mathbf{x}$ *and* $\mathbf{x}'$ *that are assigned to the same hash bucket with binary code b and Hamming distance greater than or equals to 2, i.e.,* $h(\mathbf{x}) = h(\mathbf{x}') = \mathbf{b}$ *and* $||\mathbf{b}||_1 \geq 2$. *Then the following bound on the angle between* $\mathbf{x}$ *and* $\mathbf{x}'$ *applies:*

$$\theta(\mathbf{x}, \mathbf{x}') < \arccos\left(\sqrt{\frac{||\mathbf{b}||_1 - 1}{||\mathbf{b}||_1}}\right)$$

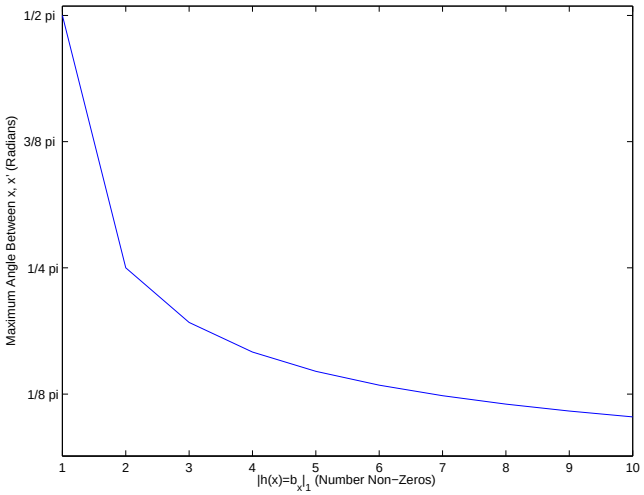

Figure 6: Plot of bound in Lemma 1 for increasing hamming length (number of non-zero entries or $||\mathbf{b}||_1$). As the hamming length increases (x-axis) the maximum angle between two points mapped to the same hash code decreases (y-axis).

Proof. According to Lemma 2 of [6], the cosine of the angle between an arbitrary binary code $\mathbf{b}$ and one of its adjacent binary codes (i.e their Hamming distance is one) is bounded by $\left[\sqrt{\frac{||\mathbf{b}||_1-1}{||\mathbf{b}||_1}}, \sqrt{\frac{||\mathbf{b}||_1}{||\mathbf{b}||_1+1}}\right]$. From the definition of angular quantization based hashing, we can infer that the separating boundary of two adjacent binary codes equally divides the space between them (the two binary codes). Therefore, we can conclude that since both $\mathbf{x}$ and $\mathbf{x}'$ are assigned to the hash bucket $\mathbf{b}$, the angle between either $\mathbf{x}$ or $\mathbf{x}'$ to $\mathbf{b}$ is strictly smaller than $\frac{1}{2}\arccos\left(\sqrt{\frac{||\mathbf{b}||_1-1}{||\mathbf{b}||_1}}\right)$. Formally, we have $\theta(\mathbf{x}, \mathbf{b}) < \frac{1}{2}\arccos\left(\sqrt{\frac{||\mathbf{b}||_1-1}{||\mathbf{b}||_1}}\right)$ and $\theta(\mathbf{x}', \mathbf{b}) < \frac{1}{2}\arccos\left(\sqrt{\frac{||\mathbf{b}||_1-1}{||\mathbf{b}||_1}}\right)$. Then, it must be the case that the angle between $\mathbf{x}$ and $\mathbf{x}'$ is strictly smaller than two times the angle between either of them to $\mathbf{b}$, formally, $\theta(\mathbf{x}, \mathbf{x}') < \arccos\left(\sqrt{\frac{||\mathbf{b}||_1-1}{||\mathbf{b}||_1}}\right)$, otherwise either $\mathbf{x}$ and $\mathbf{x}'$ would be assigned to an adjacent hash bucket of $\mathbf{b}$. □

Theorem 1. ***The inter hash bucket minimum angular distance guarantee.*** *Given two arbitrary instances* $\mathbf{x}$ *and* $\mathbf{x}'$ *that are assigned to the same hash bucket with binary code* $\mathbf{b}$ *with Hamming length greater than or equals to 2, i.e.,* $h(\mathbf{x}) = h(\mathbf{x}') = \mathbf{b}$ *and* $||\mathbf{b}||_1 \geq 2$. *Let* $\mathbf{b}'$ *denote the binary code of an arbitrary hash bucket, if the Hamming distance between b and* b' *is greater than or equals to 2, i.e.,* $||\mathbf{b} - \mathbf{b}'||_1 \geq 2$. *Then the angle between* $\mathbf{x}$ *and* $\mathbf{x}'$ *is guaranteed to be smaller than the angle between* $\mathbf{b}$ *and* $\mathbf{b}'$. *Formally, under the above conditions, the following bound applies:*

$$\theta(\mathbf{x}, \mathbf{x}') < \theta(\mathbf{b}, \mathbf{b}')$$

Proof. According to the Lemma 2 of [6], when the Hamming distance between two binary codes $\mathbf{b}$ and $\mathbf{b}'$ is greater than or equals to 2, $\cos\theta(\mathbf{b}, \mathbf{b}') \leq \sqrt{\frac{||\mathbf{b}||_1}{||\mathbf{b}||_1+2}}$. According to the Lemma 1 when both $\mathbf{x}$ and $\mathbf{x}'$ are assigned to the hash bucket $\mathbf{b}$ then $\cos\theta(\mathbf{x}, \mathbf{x}') > \sqrt{\frac{||\mathbf{b}||_1-1}{||\mathbf{b}||_1}}$. Since $\sqrt{\frac{||\mathbf{b}||_1-1}{||\mathbf{b}||_1}} \geq \sqrt{\frac{||\mathbf{b}||_1}{||\mathbf{b}||_1+2}} \implies ||\mathbf{b}||_1^2 + ||\mathbf{b}||_1 - 2 > ||\mathbf{b}||_1^2 \implies ||\mathbf{b}||_1 \geq 2$, the condition that satisfies $\sqrt{\frac{||\mathbf{b}||_1-1}{||\mathbf{b}||_1}} \geq \sqrt{\frac{||\mathbf{b}||_1}{||\mathbf{b}||_1+2}}$ is $||\mathbf{b}||_1 \geq 2$, which is already satisfied as a prerequisite. This yields the guarantee that $\cos\theta(\mathbf{x}, \mathbf{x}') > \sqrt{\frac{||\mathbf{b}||_1-1}{||\mathbf{b}||_1}} \geq \sqrt{\frac{||\mathbf{b}||_1}{||\mathbf{b}||_1+2}} \geq \cos\theta(\mathbf{b}, \mathbf{b}')$. After applying arccos to both sides of this inequality, the result is the bound shown in Theorem 1. □

3. EMPIRICAL ANALYSIS

The previous sections described our algorithm and here we describe its performance. The algorithm we use in these experiments is described in Figure 5 and all code and data to reproduce these experimental results will be made available on our website. We wish to answer the following questions:

- How efficient is our method (both building the hash buckets and creating the dendrogram) compared to the optimized code found in the built-in MATLAB functions?
- Is the hashing method suitable for typical high dimensional data such as images and documents?
- How do the quality of the dendrograms built using our method compare to standard hierarchical agglomerative clustering using a variety of linkage functions?
- How big a collection of objects can we hierarchically cluster in less than a minute on a typical desktop machine?

To answer these questions we perform experiments on common types of high dimensional data, images and documents, which we now describe.

3.1 Datasets

The two datasets we use in our experiments are the `20-newsgroup dataset` [10] and `cifar-10` [9]. These datasets were chosen because they are both large in terms of number of instances and number of dimensions, and also contain ground truth labels for each instance. While our hierarchical

clustering method is capable of learning from much larger datasets than these, standard HAC algorithms are not capable, and thus not comparable.

The 20-newsgroup data is a collection 20,000 documents corresponding to online newsgroup posts. The documents are preprocessed using Rainbow[12] to create a bag of words representation with roughly 50,000 dimensions. Though this may seem excessive, the cosine distance function measures directional distance and is suited for such datasets. The ground truth labels for this dataset used are the actual newsgroup that a document was originally posted in of which there are 20. This document set is of particular interest for hierarchical clustering because the labels have a natural hierarchical structure corresponding to the hierarchical nature in which the newsgroups are organized (see Figure 1).

- **Computers**
  - Hardware
    - comp.os.mswindows.misc
    - comp.windows.x
    - comp.graphics
  - Software
    - comp.sys.ibm.pc.hardware
    - comp.sys.mac.hardware
- **Recreation**
  - Automobiles
    - rec.autos
    - rec.motorcycles
  - Sports
    - rec.sport.baseball
    - rec.sport.hockey
- **Sale**
  - misc.forsale
- **Science**
  - sci.med
  - Technology
    - sci.crypt
    - sci.electronics
    - sci.space
- **Politics**
  - talk.politics.guns
  - International
    - talk.politics.mideast
    - talk.politics.misc
- **Philosophy/Religion**
  - alt.atheism
  - Theism
    - talk.religion.misc
    - soc.religion.christian

Table 1: Hierarchical structure of the 20 Newsgroup data.

The cifar-10 dataset is a subset of the 80 million tiny images dataset [16]. This subset is a collection of 60,000 32×32 images that have each been labeled with one of 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. These classes are balanced (6000 images for each class) and each 3072 dimensions corresponding to 3×32×32 pixel color combinations.

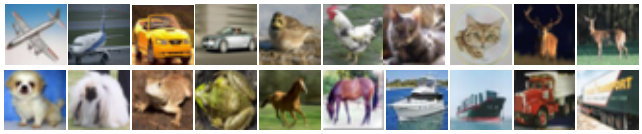

Figure 7: Example images from the cifar-10 dataset. Every image has 32×32 color pixels.

3.2 Measuring Clustering Performance

To evaluate the ability of our algorithms to learn the hierarchical structure of data, we use a common practice of evaluating the quality of the flat clusterings produced by cutting the hierarchy at a given point and comparing to a ground truth labeling/partition using Rand index. The Rand index is the proportion of instance pairs that are either in (or not in) the same clustering in both our set partition and the set partition induced by the ground truth. It is naturally interpreted as indicative of the chance that two instances chosen at random will be in together or apart in both our results and the ground truth set partition.

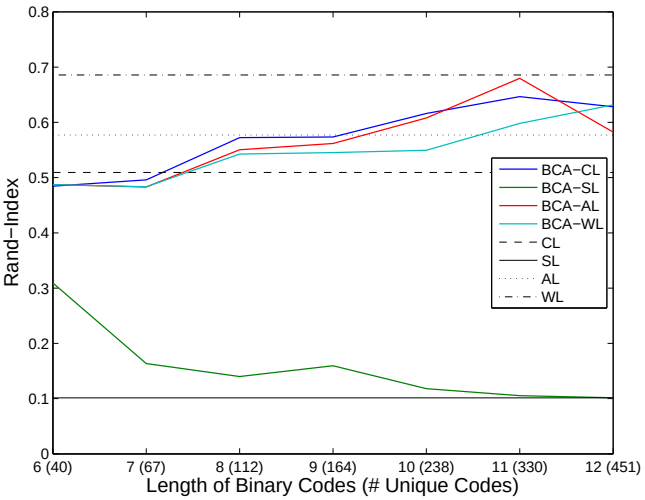

Figure 8: Performance of standard HAC vs BCA-HAC (our method) for increasing binary codes size on the cifar-10 dataset. Plot labels CL,SL,AL,WL correspond to complete, single, average, and weighted average linkages respectively. Each of these standard HAC algorithms is compared to our algorithm while used in conjunction with the same set of linkages. 10 samples of 10000 points were used to evaluate both our method and standard HAC. The results shown are the average over all 10 samples.

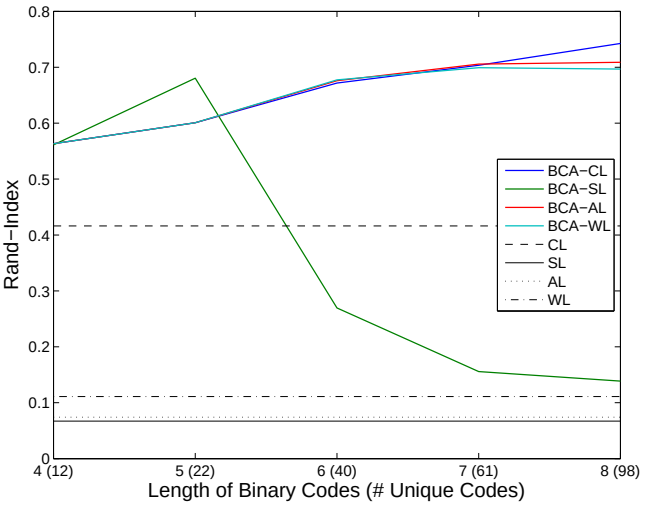

Figure 9: Performance of standard HAC vs BCA-HAC for increasing binary codes size on the 20-Newsgroups dataset. 10 samples of 1000 points were used to evaluate both our method and standard HAC. The results shown are the average over all 10 samples.

Figures 8 and 9 both compare our method with standard HAC using the Rand index. In these figures the horizontal lines represent the standard HAC results, and the other

lines represent the results of our methods for varying number length binary codes. Note the longer the binary code the more hash buckets (shown in parentheses in the figure's x-axis labels) and the more complex the algorithm's behavior. On the far left hand side of the x-axis is the binary code length dictated by choosing the length statically with the results for increasing binary code lengths shown for the remainder of the axis. The plots represent the averages of repeating the experiments 10 times with random samples of the original dataset (samples of size 10,000 from the population of 60,000 for cifar-10 and 5,000 from the population of 20,000 for 20-newsgroups). Though our method can handle much larger datasets, our purpose here is to compare against existing methods which can take 10+ hours for the later dataset.

These results show that single linkage, the fastest standard HAC algorithm ($O(n^2)$ versus $O(n^2 \log n)$ for complete and average linkage algorithms), performs poorly on both of these datasets. However, when our method is combined with single-linkage clustering, our results are consistently as good as or better than the standard single linkage results. An interesting trend in this result is that as the number of unique codes increases, our algorithm decreases performance, until it converges with the performance of standard single linkage. We believe this is the because as the number of unique codes increases, the more the results converge to the single linkage algorithm's results since there are fewer points in each hash bucket. Contrast this with the results for complete linkage where performance increases with the number of unique codes. Clearly complete linkage is a much better fit for both of these datasets but it is interesting to note why our method is so much better. We note that our method is comparable to the average and weighted average linkage methods which effectively measure the distance between clusters as the distances between the middle of the clusters. Similarly the hash code falls approximately at the center of the hyper-cone that encompasses all instances in the hash bucket. Therefore we can reason that when our approach merges together hash buckets it is doing a similar calculation to the average and weighted average linkage methods. Since the focus of our work is efficient clustering, we will leave it to future work to better understand when these performance improvements occur.

The experiment whose results in Figure 10 was designed to test if the design decisions used in our recursive formulation of our algorithm could hurt performance. Specifically for our algorithm to work we need to create binary codes that are longer than would otherwise be needed. The question we try to answer then, is if we create long binary codes, does it negatively effect the quality of appropriately sized prefixes. Figure 10 shows the results of increasing binary code lengths on the performance of the nonrecursive formulation of our algorithm. The non-recursive formulation uses only the first 10 bits (chosen adaptively) of the binary code to create a low detail hierarchy, and the remaining bits are completely ignored. What the plot shows is that increasing the number of bits on the scale that would be used in our recursive algorithm does not decrease the quality of the prefixes. To our surprise the quality of the prefixes actually increases, which we believe is a side effect of the alternating optimization used to solve Equation 2 being better suited to this setting. To support that claim we show on the x-axis in parenthesis the number of unique binary codes found for the prefix set increases for longer total binary code lengths.

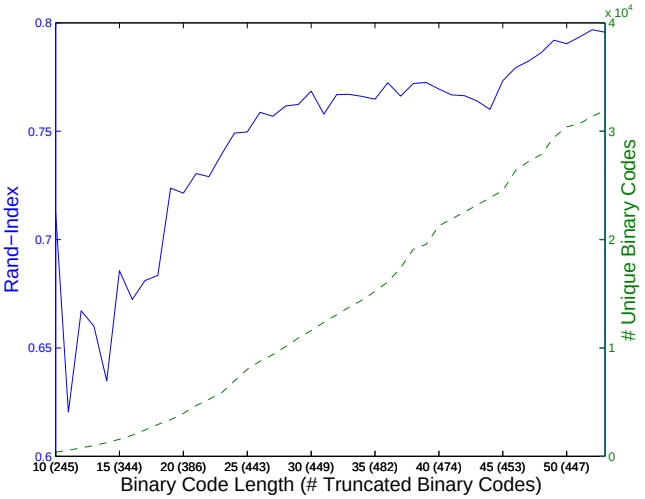

Figure 10: This figure demonstrates that using binary codes of longer length in the recursive process of BCA-HAC does not hurt performance. This experiment used all 60,000 points from cifar-10 dataset. The left hand y-axis shows Rand-Index performance for increasing code lengths, and the right hand y-axis show the number of unique binary codes in the lengthened binary codes (whereas values in parenthesis of x-axis only show number of unique codes in prefix of length 10).

3.3 Runtime Performance

Figures 11 and 12 show the run time results of our algorithms for increasing sized samples of the cifar10 and 20-newsgroups datasets respectively. Note that the former dataset has approximately 3,000 dimensions and the later 50,000 dimension. Our speedup for the image dataset (Figure 11) is approximately 100 fold and in the later document dataset (Figure 12) is 10,000 fold faster (note the y-axis is the square root).

Finally, we wish to determine how big of a real dataset we can completely hierarchically cluster in less than a minute. We were able to cluster the 60,000 images of the cifar-10 dataset (see Figure 7) in less than one minute. The total computation time can be broken up into three parts: binary code computation, data structures for indexing sets of points in each bin, and time spent hierarchically clustering hash buckets. The time to calculate the binary codes was 6 seconds The data structures used to efficiently find sets of points in hash buckets were produced using the Matlab `unique` function which only took less than 1 second for the set of 60,000 binary codes produced. Finally it took 1 second to perform single-linkage HAC on the collections of hash buckets. Note that since we recursively apply our algorithm (option 6b) in Figure 2) we build a complete dendrogram. This gives a total of approximately 8 seconds.

Our comparison is more than fair to standard HAC clustering methods because we use the built-in optimized matlab standard libraries for the methods (`pdist, linkage` etc.), while our methods are written in using user created Matlab functions and script. The core computational functions of

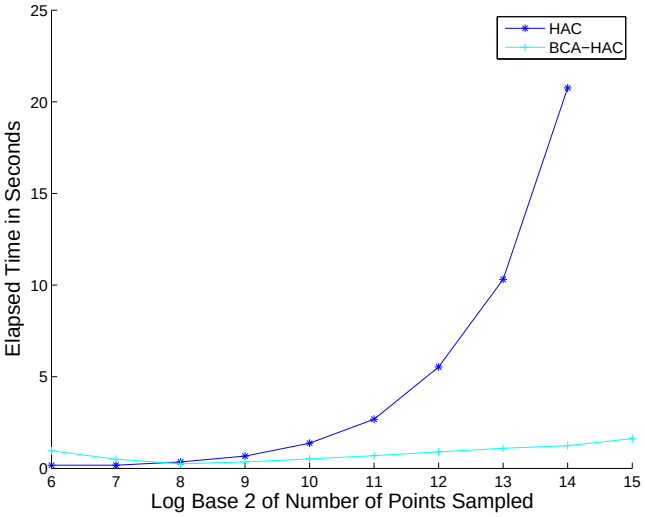

Figure 11: Time of standard HAC compared to our method for the cifar10 dataset (3,072 dimensions). For standard HAC we stopped after sample sizes of $2^{13} = 8192$ because for the next largest sample size the algorithm did not converge in a reasonable amount of time (terminated after 12 hours). These figures were produced using single linkage for both standard HAC and for our method BC-HAC.

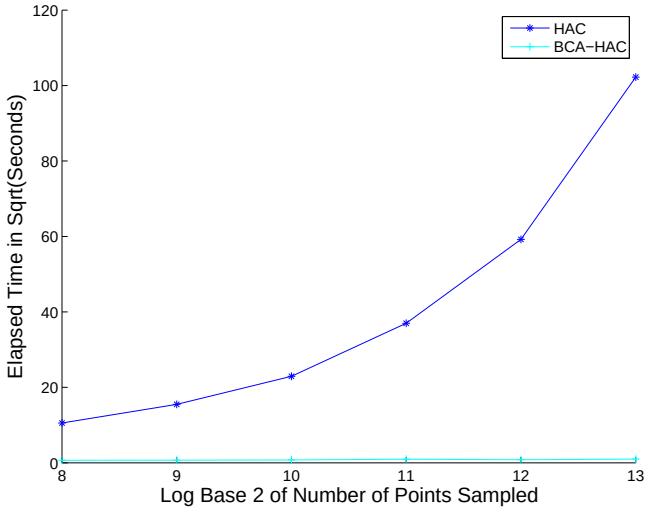

Figure 12: Time of standard HAC compared to our method for the 20-newsgroups data-set (50,000 dimensions). This figure uses a sqrt transformed y-axis and therefore for sample sizes of $2^{13} = 8192$ our algorithm is roughly 10,000 times faster than standard HAC.

Matlab libraries are implemented in machine compiled languages such as Fortran, whereas our functions and scripts are not, however our functions take advantage of same parts of those libraries, such as matrix multiplication and sorting.

4. CONCLUSIONS

Hierarchical clustering is used extensively to organize documents and images. Agglomerative methods are particularly popular but suffer from two main problems that limit their application to typically no more than 10,000 instances: i) The time complexity requires at least quadratic distance calculations, and ii) The commonly used cosine distance function requires a time consuming inner-product. Minimizing the number of these calculations offers promise to scale agglomerative methods to handle much larger datasets.

We propose the first angular hashing based method for efficient hierarchical clustering. Angular hash functions create a series of hash buckets, each of which has its own hash code and can be used to allocate instances to the hash buckets. Our approach (see Figures 2 and 5) builds a dendrogram by both hierarchically clustering the points in each hash bucket and then clustering the hash buckets hierarchically based on their hash codes. This is possible since we show that the angle between the instances at the approximate center of the hash bucket is related to the Hamming distance of the hash codes.

We tried our method on a collection of images (3,000 dimensions) and documents (50,000 dimensions). We found that our method achieves a speed-up of 100 times for the former dataset and 10,000 times for the latter higher dimensional dataset. Unusually we found that our method produced results as good as or better (when measured compared to the ground truth) than the commonly used single linkage, complete linkage, average linkage and weighted average distance functions. We postulated that this is so since our method approximates the more suitable average cluster distance function. In practice our methods can easily scale to hierarchically cluster 60,000 images in under a minute of computation and it can scale to data sets of 100,000's of points which was not previously possible with agglomerative clustering.

5. REFERENCES

[1] S. Basu, I. Davidson, and K. L. Wagstaff, editors. *Constrained Clustering: Advances in Algorithms, Theory, and Applications*. Chapman and Hall/CRC, 1 edition, Aug. 2008.

[2] D. Cai, X. He, Z. Li, W.-Y. Ma, and J.-R. Wen. Hierarchical clustering of www image search results using visual, textual and link information. In *Proceedings of the 12th annual ACM international conference on Multimedia*, pages 952–959. ACM, 2004.

[3] D. Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 1977.

[4] B. S. Everitt, S. Landau, and M. Leese. *Cluster Analysis*. Wiley Publishing, 4th edition, 2009.

[5] S. Gilpin and I. Davidson. Incorporating sat solvers into hierarchical clustering algorithms: an efficient and flexible approach. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1136–1144. ACM, 2011.

[6] Y. Gong, S. Kumar, V. Verma, and S. Lazebnik. Angular quantization-based binary codes for fast similarity search. In P. Bartlett, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1205–1213. 2012.

[7] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 99th edition, 1975.

[8] H. Koga, T. Ishibashi, and T. Watanabe. Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. *Knowledge and Information Systems*, 12(1):25–53, 2007.

[9] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[10] K. Lang. Newsweeder: Learning to filter netnews. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 331–339, 1995.

[11] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 169–178, New York, NY, USA, 2000. ACM.

[12] A. K. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. http://www.cs.cmu.edu/ mccallum/bow, 1996.

[13] S. E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27 – 64, 2007.

[14] R. Sibson. Slink: an optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[15] A. Singhal. Modern Information Retrieval: A Brief Overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24(4):35–42, 2001.

[16] A. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970, 2008.

[17] P. Willett. Recent trends in hierarchic document clustering: A critical review. *Information Processing & Management*, 24(5):577 – 597, 1988.

[18] Y. Zhao, G. Karypis, and U. Fayyad. Hierarchical clustering algorithms for document datasets. *Data Mining and Knowledge Discovery*, 10(2):141–168, 2005.