Active Image Clustering: Seeking Constraints from Humans to Complement Algorithms

November 22, 2011

Abstract

We propose a method of clustering images that combines algorithmic and human input. An algorithm provides us with pairwise image similarities. We then actively obtain selected, more accurate pairwise similarities from humans. A novel method is developed to choose the most useful pairs to show a person, obtaining constraints that improve clustering. In a clustering assignment elements in each data pair are either in the same cluster or in different clusters. We simulate inverting these pairwise relations and see how that affects the overall clustering. We choose a pair that maximizes the expected change in the clustering. The proposed algorithm has high time complexity, so we also propose a version of this algorithm that is much faster and exactly replicates our original algorithm. We further improve run-time by adding heuristics, and show that these do not significantly impact the effectiveness of our method. We have run experiments in two different domains, namely leaf images and face images, and show that clustering performance can be improved significantly.

1 Introduction

Clustering, or unsupervised learning, is a critical part of the analysis of data. There has been a huge volume of work on clustering, producing many interesting and effective algorithms. However, all clustering depends on some method of computing a distance between items to be clustered that reflects their similarity. For most tasks, automatically computed distances provide useful information about similarity, but still produce significant errors. This leads even the best clustering algorithms to produce clusters that do not contain objects from the same class.

We therefore propose a new clustering method that brings a human into the loop. In many tasks, experts, or even naive humans, can provide very accurate answers to the question of whether two objects belong in the same cluster. In spite of this accuracy, it is not practical to expect people to cluster thousands of objects into meaningful groups. Our goal, therefore is to meld human and automatic resources by directing valuable human attention to those judgments that are most critical to improving clustering produced by automatic means.
Figure 1: Pipeline of our system: Active-HACC is our proposed active algorithm for selecting data-pairs to get constraints and HACC is a constrained clustering algorithm.

To illustrate the value of this approach, we use the example of clustering in surveillance videos and plant images.

- There are many applications for clustering faces or actions in surveillance videos. This could allow, for example, an analyst to determine whether the same person has visited a number of locations, or find different people who have performed similar actions. Images from videos have variations in pose, illumination and resolution that make automatic analysis extremely challenging, so that automatic clustering will be quite error-prone. But a person can readily look at two face images or actions and tell if they are similar.

- There has been a great deal of interest recently in obtaining large, labeled image sets for plant species identification [6]. Classifiers that can identify species require large sets of leaf images, labeled by species. Accurately labeling such images requires experience and botanical knowledge. One approach that can reduce this effort is to cluster all the images into groups that each come from a single species, and then have botanists label each group. Initial clustering can be performed using generic algorithms that measure the similarity of two leaves, but this clustering will be quite noisy, because such algorithms are still imperfect. At the same time, we observe that even an untrained person can compare two leaf images and provide an accurate assessment of their similarity.

We may then summarize the clustering problem we have solved as having the following characteristics:

- We begin with a collection of objects that can be grouped into a set of disjoint clusters.
- We have an automatic algorithm that can give us some useful information about the similarity of two objects.
- A person can make these judgments with much greater accuracy than existing algorithms.
We also assume that a person always provides correct constraints and that we
know the number of clusters beforehand. In practice, humans are highly accurate at
image comparison. For example, [13] shows that humans achieve 99.2% accuracy
in a face comparison task, even without the option of responding “don’t know”.
Like much work in clustering and all prior work on active clustering, we focus
on the problem of forming clusters. Many approaches have been developed for
determining the number of clusters [19] but this is outside the scope of our current
work.

Given this problem formulation, we have proposed an algorithm that does the
following:

1. Cluster objects into groups, combining automatically computed distances
   with any constraints provided by people.
2. Choose a useful question to ask a person. The person will compare two
   objects and indicate whether they belong to the same group (or answer “don’t
   know”).
3. Repeat, using the information provided by the person as a new constraint.
4. Continue asking questions until a reasonable clustering is achieved or the
   human budget is exhausted.

We show the pipeline of our algorithm in Figure 1 (face images from [16]).

We do experiments to evaluate our algorithm in two different domains: face
images and leaf images. Since we assume that people are highly accurate, in ex-
periments we can simulate their behavior using ground truth.

2 Related Work

Combining active learning [2, 18] with constrained clustering [4] has been a grow-
ing area of interest in machine learning as well as in computer vision. Some active
constraint clustering approaches are also known for image clustering [7, 9]. In [7],
the authors have proposed a heuristic, which works reasonably well but has several
parameters that must be tuned properly for different datasets. In [9], the authors
take a fuzzy clustering based approach to find images near cluster boundaries to
form useful data pairs.

In [3], Basu et al. also proposed an active clustering algorithm. They sug-
gest two major phases in an active learning setting namely “Explore” (cluster cen-
ter initialization) and “Consolidate” (data points are added to cluster centers). In
problems with a large number of clusters (which is very common in the image
clustering domain), the “Explore” stage itself takes a large number of questions to
initialize the distinct cluster centers. Mallapragada et al. [15] have proposed an
approach that uses a min-max criterion to find informative questions. They rely on
the “Explore” stage in the beginning as [3] does. There are also a couple of active
clustering algorithms [22, 23] based on spectral eigenvectors, but they are good for
two-cluster problems only.
In [11], Huang et al. have proposed an active framework for constrained document clustering. This paper is philosophically similar to our approach, i.e. they also try to ask questions to maximize the gain. They begin with a skeleton structure of neighborhoods covering all the clusters. They then search for an informative data pair to match an unlabeled data point to one of the centroids of the existing neighborhoods. Also, they use an “Explore” stage to build an initial skeleton structure, which we already know to be a potential problem when there is a large number of clusters. Another approach by Huang et al. [10] has an active constraint clustering algorithm for documents with language modeling and it is not clear how we could adopt this algorithm for image clustering.

3 Our Approach

We now describe our approach to active clustering. We first motivate this approach with a simple example, and then describe technical details.

3.1 High Level Intuition

We have developed a novel algorithm that determines which questions to ask a person in order to improve clustering. This algorithm is based on the intuitive idea of asking questions that will have the largest expected effect on the clustering. This really has two components. First, if we ask a person to compare two objects, her answer will only affect the clustering immediately if it differs from it; that is, if the person says either that two objects that are not currently in the same cluster should be, or that two objects that are in the same cluster should not be. Any answer that differs from the current clustering must result in our moving at least one object to a different cluster, but some answers will affect many more objects. So the second component of our algorithm is to ask questions whose answers might have a big effect on the way objects are clustered.

To provide better intuition about our approach, we consider the simple toy example of clustering a set of 2D points. Figure 2 shows a collection of such points as black disks. The circles indicate four clusters that might be formed by an automatic clustering algorithm. We have marked five of these points with the letters “A” to “E” for ease of reference. We now imagine that an expert can compare two of these points and tell us whether they truly belong in the same cluster. Considering the following possibilities, we find:

![Figure 2: Toy example to motivate our approach.](image)
• Comparing B and C is not that desirable, since it is likely that we have already correctly placed them in different clusters. A human opinion about these two points is unlikely to change anything.

• Comparing A and B (or A and C) will tell us in which cluster A truly belongs. Since A is between two clusters, it is quite possible that this question will change the cluster to which we assign A. However, the answer to this question will only affect A.

• Comparing D and E will provide the most information. These two clusters are close, and it is somewhat ambiguous whether they should be separated into two clusters, or joined into one. So it is reasonably likely that a person might say D and E belong in the same cluster. If they do, this will lead us not only to treat D and E differently, but in fact to join the two clusters together, affecting the grouping of many points.

Consequently, we select questions for human attention that maximize the product of the probability that the answer will cause a change in our current clustering and the size of this change, should it occur. Finding the best such question can potentially require a large amount of computation. If we are clustering $N$ objects, then there will be $O(N^2)$ same-or-different questions to consider, and for each we must determine its possible effects. For this reason, we adopt a simple, greedy clustering algorithm. Without human assistance, this algorithm does not perform well, but by using a simple clustering algorithm, we can more easily and quickly select good questions to ask a human, and rapidly improve our clustering performance.

In order to further speed up our algorithm, we have also experimented with two additional heuristics.

First, when our estimate of the probable response of a human indicates that it is very likely that the human response will agree with the current clustering, we do not bother to simulate the results of a different response. For all datasets we exclude simulation of pairs which are very unlikely to be in the same cluster. For larger datasets (of size more than 1000), we initially use K-means [14] to group very close points together and represent them using their centroids and then run our algorithm. We refer to this heuristic as H1.

Second, we observe that when we simulate an additional constraint between a data pair, change in clustering assignments is often limited to clusters that contain the points in that pair. Determining those changes is much faster than checking for all possible changes. We perform experiments with this approximation and we find that it makes our algorithm’s performance a little worse but much faster. We refer to this heuristic as H2.

Below we give the high level intuition of our full algorithm.

### 3.1.1 Clustering Algorithm

We use a simple algorithm to perform clustering. As input to this algorithm, we assume that we can compute a distance between every pair of objects (Many clustering algorithms assume that objects are embedded in a vector space. However, in
many applications we have an algorithm able to compare objects, but no obvious such embedding. Since we only make use of a pairwise distance, our algorithm is quite general. In addition, we have constraints on possible clusters. For a given pair of objects, we may know that they must appear in the same cluster, or that they cannot possibly appear in the same cluster. We can represent this using a complete graph, in which each object is a node, and the edge between two nodes represents their distance or a “must link” or “cannot link” constraint. A must link constraint implies a user mentioned that two images are similar and cannot link constraint means two objects are from different classes.

We initialize clustering by forming a cluster (or a single node tree) for each object. Next, we merge all clusters connected by a must link constraint. Then we sort all edges. Beginning with the shortest edge, we progress through all edges. If an edge links two clusters, we merge these clusters unless there is a cannot link constraint connecting them. We halt when we have formed a predefined number of clusters. What we get is a spanning forest and in section 3.2.2, we describe this algorithm in detail.

3.1.2 Estimating the probability of a response

To select the best questions to ask an expert, we must estimate the probability distribution of their answer. While in general, this is quite difficult, we make use of the following, simple heuristic. At the beginning of processing, we run the $k$-means clustering algorithm on our initial data set and distances many times, using many random starting points. Then, for each pair of objects, we record the fraction of times that they are assigned to the same cluster. We use this as an estimate of the probability that a human will assign them to the same cluster. We show experimentally that this simple approach works well in practice.

3.1.3 Simulating an update

The most difficult challenge in our algorithm lies in efficiently determining the effect that a new constraint, provided by a human, will have on how the objects are clustered. Determining this change is made complex by the presence of cannot link constraints. Suppose a human response causes us to add a cannot link constraint between two objects that are currently in the same cluster. This cluster must somehow be broken apart to separate these two objects. To do this, any of our previous decisions to merge clusters might have to be undone. At the same time, undoing one decision to merge clusters might make it feasible to merge other clusters that were previously not allowed to merge. In fact, even the effects of a new must link constraint can cause a domino effect of changes. If human input tells us that we should merge two clusters, but some pairs of objects in these clusters already have don’t-link constraints, we must merge parts of the clusters, while breaking them apart in other ways, to ensure that all constraints are respected. At the same time, it is important to simulate all these effects quickly, since we must calculate these
effects for all \(O(n^2)\) possible questions that could be asked of a human.

To cope with this challenge, we have developed an algorithm that performs these simulations quickly. The algorithm is described in detail in Section 3.4. Intuitively, though, the algorithm simulates the addition of a new constraint by efficiently determining which previous merging decisions need to be undone to maintain consistency with the new constraint, and then performing new merging steps that are made possible when these decisions are undone. The key point here is that our algorithm produces exactly the same result that would be achieved by simply starting the greedy algorithm from the beginning with the addition of this new constraint, but with significantly less computation.

### 3.2 Components of Our Algorithm

We now explain our algorithm and its components more formally. We define the best image pair that we will present to a person for labeling as:

\[
(\hat{d}_i, \hat{d}_j) = \arg \max_{d_i, d_j} E_J(d_i, d_j)
\]

\(E_J(d_i, d_j)\) is the expected change in the clustering if a question is asked about an image pair \(d_i\) and \(d_j\). Since we do not know the ground truth clustering, we cannot choose image pairs that are guaranteed to increase the quality of the clustering. One of the main insights of our work is that by finding pairs that most rapidly change the clustering we quickly converge to a good clustering. Now, we formally describe how we compute the components needed to determine \(E_J(d_i, d_j)\) given we have any distance matrix for a dataset.

#### 3.2.1 Consensus Clustering based Pairwise Probability Distribution

We first need to estimate the probability that each data pair will be in same cluster. We have borrowed ideas from consensus clustering [17] (also referred to as aggregation of clustering) to find those probabilities. In consensus clustering we typically compute multiple clusterings of the same dataset and then produce a single clustering assignment that reflects a consensus. This can be developed as an optimization problem that is known to be NP-complete. Consensus clustering in unsupervised learning is similar to ensemble learning in a supervised framework. However we avoid optimization, and just use multiple clusterings to estimate the probability that the elements of a pair belong to the same cluster.

Specifically, if we have \(N\) data points and \(S\) clusterings, let \(A_s\) be the \(N \times N\) matrix where element \((i, j)\) is 1 if the \(i\)-th and \(j\)-th data points are in the same cluster in the \(s\)-th clustering, and zero otherwise. We use the K-means algorithm to produce clusters, each time beginning with different random initial cluster centroids. Now if \(P\) is the probability matrix for \(N\) data points where again element \((i, j)\) is the pairwise probability of the \(i\)-th and \(j\)-th data points being in the same cluster,
\[ P = \frac{1}{S} \sum_{s=1}^{S} A_s \]  

(2)

If \((d_i, d_j)\) is any pair of points and \(R\) is a random variable corresponding to pair \((d_i, d_j)\) resulting from the above random process then we estimate \(P(d_i, d_j) = \text{prob}(d_i = d_j|R) = R\). \(d_i = d_j\) implies \(d_i\) and \(d_j\) are from same class. We experimentally verify that this method produces reasonable values. In Figure 3 (data from [1]), we plot \(\text{prob}(d_i = d_j|R) = R\) and a histogram generated from the results of those experiments showing the fraction of times that a pair with a given \(R\) is from the same cluster. Our heuristic provides a good estimate of \(\text{prob}(d_i = d_j|R) = R\), which grows more accurate for larger data sets.

![Figure 3: Pairwise probability distribution of real data.](image)

### 3.2.2 Minimum Spanning Forests (MSF)

We perform clustering by creating a Minimum Spanning Forest (MSF). We define an MSF as a collection of trees which we get if we run Kruskal’s algorithm[12] and stop when we have \(K\) spanning trees. We can think of clustering as finding a forest with \(K\) spanning trees from a set of disconnected nodes. We use a constrained clustering algorithm very similar to Kruskal’s algorithm but also respects constraints. In the clustering community a similar approach is well-known as bottom up or hierarchical agglomerative clustering (HAC). We assume we are provided with the distance matrix for a given dataset. We can consider each image of the dataset as an individual node in a complete graph \(G = (V, E)\) and let their mutual distances represent edge weights. We can also have information about a pair of images being in the same cluster (“must-link” constraints) or different clusters (“can’t-link” constraints). Let us assume we create a copy of the graph \(G\) without any edges and call it \(F = (V, \emptyset)\). First, we add edges corresponding to must-link constraints to the forest \(F\). Next we sort the edges in \(E\) in an ascending order of their weights and store them in a queue \(E_p\). We start popping edges from \(E_p\) and add to \(F\). While doing this, we always maintain the tree structure, i.e. do not add an edge between two nodes if the nodes are already connected. We will also not add an edge between two trees if there is any can’t-link constraint between any of the pairs of nodes in
those two trees. We continue doing this until we have K trees in the forest (referred as MSF in future). We refer to this algorithm as HAC with constraints (HACC).

We will discuss later why we build on the MSF rather than constrained K-means [21] or other constrained algorithms [24].

Since we are working with hierarchical clustering with constraints we have to discuss feasibility and dead-end issues [8]. The feasibility problem is defined as, given a set of data and set of constraints, does there exist a partitioning of the data into K clusters? In our problem the answer is obviously yes because all of the constraints are true. However determining whether there is a feasible solution which satisfy all constraints is NP-complete [8]. In HACC, dead-end situations (reaching an iteration with more than K clusters, where no further pair of clusters can be joined without violating any of the constraints) can occur in principle, but in practice we find this is not a problem.

3.3 Our Algorithm (Active-HACC)

In 3.2.1, we estimate a distribution that allows us to determine the probability that a person will provide a constraint that differs from the current clustering. In this section, we determine the magnitude of the change this will have on the current clustering. To do this, we define a measure of similarity between two clustering assignments, which we call Relative Jaccard’s Coefficient, by analogy to the Jaccard’s Coefficient [20]. If \( C_1 \) and \( C_2 \) are two clustering assignments of the same dataset, the Relative Jaccard’s Coefficient of clustering \( C_2 \) with respect to \( C_1 \) is defined as:

\[
JCC_{C_1}(C_2) = \frac{SS}{SS + SD + DS}
\]  

where \( SS \) is the number of pairs that are in the same cluster in both \( C_1 \) and \( C_2 \), \( SD \) is the number of pairs that are in same cluster in \( C_1 \) but in different clusters in \( C_2 \), and \( DS \) is the number of pairs that are in different clusters in \( C_1 \) but are in same cluster in \( C_2 \). This becomes the traditional Jaccard’s coefficient if \( C_1 \) is the ground truth clustering.

Now, we describe our algorithm, assuming that we have asked \( q \) questions and need to determine the \((q + 1)\)-th question to ask. We define:

- \( D \): The dataset, that should be clustered.
- \( K \): Number of clusters.
- \( d_i, d_j \): Any pair of images from \( D \).
- \( C_q \): The set of can’t-link constraints obtained after we have asked \( q \) questions.
- \( M_q \): The set of must-link constraints obtained after we have asked \( q \) questions.

Note \( |C_q| + |M_q| = q \).

\( H_q = HACC(D, C_q, M_q) \) : HACC is the clustering function on a given dataset \( D \), using the must-link constraints \( (M_q) \) and can’t-link constraints \( (C_q) \). \( H_q \) is the clustering that is produced.
\[ J_{q+1}^{d_i, d_j, y} = \text{JCC}_{H_q}(\text{HACC}(D, C_q, M_q \cup d_i = d_j)) : \text{Relative Jaccard's Coefficient of a clustering after } q + 1 \text{ questions with respect to } H_q, \text{ if the } (q + 1)\text{-th constraint is that } d_i \text{ and } d_j \text{ are in same cluster.} \]

\[ J_{q+1}^{d_i, d_j, n} = \text{JCC}_{H_q}(\text{HACC}(D, C_q \cup d_i \neq d_j, M_q)) : \text{Relative Jaccard's Coefficient of a clustering after } q + 1 \text{ questions with respect to } H_q, \text{ if the } (q + 1)\text{-th constraint is that } d_i \text{ and } d_j \text{ are not in same cluster.} \]

Now, we ask the user about the pair:

\[ (d_i, d_j) = \arg \max_{d_i, d_j} E_J(d_i, d_j) \quad (4) \]

Where \( E_J(d_i, d_j) \) is the expected change in the Relative Jaccard’s Coefficient and is defined as follows:

\[ E_J(d_i, d_j) = |\text{JCC}_{H_q}(H_q) - (P(d_i, d_j) J_{q+1}^{d_i, d_j, y} + (1 - P(d_i, d_j)) J_{q+1}^{d_i, d_j, n})| \quad (5) \]

Note that \( \text{JCC}_{H_q}(H_q) = 1 \) and \( P(d_i, d_j) \) is the probability of \( d_i \) and \( d_j \) being in the same cluster.

Now we can see that if points \( d_i \) and \( d_j \) are in the same cluster after \( q \) questions, then \( \text{HACC}(D, C_q, M_q \cup d_i = d_j) = H_q \) and if they are in different clusters then \( \text{HACC}(D, C_q \cup d_i \neq d_j, M_q) = H_q \). So we have:

- \( d_i \) and \( d_j \) are in the same cluster after \( q \) questions:
  \[ E_J(d_i, d_j) = |(1 - P(d_i, d_j))(1 - J_{q+1}^{d_i, d_j, n})| \quad (6) \]

- \( d_i \) and \( d_j \) are in different clusters after \( q \) questions:
  \[ E_J(d_i, d_j) = |P(d_i, d_j)(1 - J_{q+1}^{d_i, d_j, n})| \quad (7) \]

Using this approach, we find the data pair that will produce the greatest expected change in the clustering. After receiving an answer from the user, we update our constraints and continue.

When we plot the clustering performance, we show the Jaccard’s Coefficient relative to ground truth, as the number of questions increases. This curve does not always increase monotonically, but it generally increases.

### 3.3.1 Complexity of the algorithm

We now discuss the rather high complexity of computing a brute-force version of Active-HACC. We then consider some optimizations and heuristics to improve this run time. For each question we have \( O(N^2) \) (\( N \) is the number of points) possible pairs. To simulate what will happen for each pair in a brute force way, we will have to run the constrained clustering algorithm for each pair. We now describe the complexity of Active-HACC.

Kruskal’s algorithm [12] for minimum spanning tree runs in \( O(N^2 \log N) \) time (if \(|E| = O(N^2)\)). HACC also has \( O(N^2 \log N) \) complexity from a very similar analysis to Kruskal’s, except for the issue of keeping track of the can’t-links. To
Algorithm 1 Active-HACC

Given: \( D, \text{Max\_questions}, M_q = \emptyset, C_q = \emptyset, \text{num\_questions}=0 \)

\[
\text{while \ num\_questions} \leq \text{Max\_questions \ do} \\
\quad \text{HACC}(D,M_q,C_q) \\
\quad \text{For all pairs } (d_i,d_j), \text{ evaluate } E_{ij}(d_i,d_j) \\
\quad \text{Find the pair } (d_i,d_j) \text{ with maximum } E_{ij}(d_i,d_j) \\
\quad \text{Ask and Update } M_q \text{ and } C_q \\
\quad \text{num\_questions=}\text{num\_questions}+1 \\
\text{end while} \\
\text{Output: HACC}(D,M_q,C_q)
\]

...do this efficiently, we maintain an \( l \times l \) lower triangular matrix \( A \) in which \( l \) is the current number of clusters. \( A(m,n) = 1 \) if \( m > n \) and there is a can’t-link constraint between clusters \( m \) and \( n \), and \( A(m,n) = 0 \) otherwise. Before merging two clusters \( m \) and \( n \), we check that \( A(m,n) = 0 \) \((m > n)\). In this case, we assign cluster \( n \) to have identity \( m \). We update \( A \) by setting row \( m \) equal to the OR of rows \( m \) and \( n \), and removing row and column \( n \). This update takes \( O(N) \) time, and can occur in \( O(N) \) iterations. So enforcing the can’t-link constraints adds \( O(N^2) \) time to Kruskal’s algorithm, which still runs in \( O(N^2 \log N) \) time.

If we run this variation on Kruskal’s algorithm for \( O(N^2) \) pairs, the complexity of choosing each question will be \( O(N^4\log N) \). Even with moderate \( N \) (say \( N=100 \)) we cannot ask \( O(N) \) questions with this much cost for each question. So we will propose a much less computationally complex version of Active-HACC.

In part, this complexity helps motivate our use of a very simple clustering algorithm such as HACC. Since we must simulate \( O(N^2) \) possibilities for each question, it is important that the clustering algorithm be relatively cheap. Moreover, as we will see, HACC lends itself to incremental clustering, in which simulating the effects of one constraint can be done efficiently. At the same time, HACC is quite interesting because the addition of one constraint can in some cases significantly alter the clustering.

3.4 FAST-Active-HACC

Our previous analysis assumes that we rerun HACC \( O(N^2) \) times to simulate the effects of every question we consider asking. This is wasteful, since most new constraints only have a small effect on the clustering. We save a good deal of effort by incrementally computing these changes starting from the existing clustering. However, these changes can be somewhat complex. When a can’t-link constraint is added to points in the same cluster, we must remove an edge from the current MSF, to disconnect these points. Removing one edge can have a domino effect, since there may be other edges that would have been added if that edge had not been present. Similarly, adding a must-link constraint might require us to merge two clusters that have can’t-link constraints between them, requiring us to remove...
edges to separate any points connected by a can’t-link constraint. We must simulate any effects of these changes.

Our complete algorithm is quite complex, and is described below. Here we provide a couple of key intuitions. First, we save a good deal of time by creating data structures once that can be used in $O(N^2)$ simulations. Whenever we add a can’t-link constraint between two points in a cluster, we must remove the last edge added to the MSF that is on the path between these points. To facilitate this, we keep track of the path on the MSF between all pairs of points in the same cluster. Also, as we perform the initial clustering, whenever an edge is not added to the MSF because of a can’t-link constraint or because it would destroy the tree structure, we keep track of any edge which blocks it. That is, edge B blocks edge A if edge A would be added if not for the presence of edge B. Then, whenever an edge is removed, we can reconsider adding any edge that was blocked by this edge. Of course, as we “go back in time” and make changes to the MSF, we must carefully account for any later decisions that might also change.

In a second optimization, we notice that we do not need to simulate the effects of all Can’t-Link constraints. If a cluster has $N_c$ elements, there are $(N_c^2)$ possible Can’t-Link constraints, but only $N_c$ possible edges that can be removed. This means that many can’t-link constraints will cause us to remove the same edge from the MSF, and have the same effect on the clustering. These can be simulated together.

Since this overall procedure is complex, code will be made publicly available.

3.4.1 Definitions

We are given a set of points and pairwise distances. As we mentioned earlier, we consider a complete graph $G = (V, E)$ where each pair of distinct nodes are connected by a unique edge and each edge has a weight given by the distance matrix. We are computing a minimum spanning forest for $G$. To do this, we sort the edges, $E$ by distance, place them in a priority queue, and pop them off the queue in turn. As we progress, we can divide $E$ into four disjoint groups.

**Active Edges (AE):** This is the set of edges that have been added to the MSF. The number of active edges may vary from $(N - K - M)$ to $(N - K)$ (M is the number of must link constraints).

**Cannot Link Edges (CLE):** CLE is the set of edges that have been popped from the queue but have not been added to the MSF due to cannot link restrictions. For example in figure 3.b we do not add the edge due to the presence of cannot link constraints between AE, CG and DH. We should not confuse a Cannot Link Edge with a Cannot Link Constraint between two points.

**Redundant Edges (RE):** An edge is included in set RE if it was not added to the MSF because the pair of points connected by this edge were already in the
same cluster. Since we want to maintain a forest, we avoid all cycles.

**Other Edges (OE):** We call all the edges in E which are still on the queue, other edges.

We also put a time stamp, which is the index of an element in the sorted list, on each edge. An edge with lower weight will have a lower time stamp than an edge with higher weight. In figure 3, it is explained how we build the sorted list and apply time stamps. We use the abbreviations AE, CLE, RE and OE as both set and as an element of a set.

![Graph with Time Stamps](image)

Figure 4: Complete graph $G=(V,E)$ with edge weights on the edges and a sorted edge list.

### 3.4.2 Data Structures

In this section we discuss some data structures required for the faster version of Active-HACC. The basic intuition is as follows. When we simulate adding a cannot link edge to the same tree in an MSF, we must remove some edge, to break the tree in two and separate the tree into two components that each contain one of the nodes in the cannot link edge. We must then simulate HACC’s behavior from the point at which the removed link had been initially added. Similarly, if we simulate adding a must link edge that connects two trees that contain cannot link edges, we must join these two trees, and then remove edges to break this new tree apart, so that again no tree contains a cannot link edge. In either event, we must undo some
(a) When we add an edge between two clusters we update Store Simple path with paths between all nodes in the two trees.

(b) Simulation of SCC

(c) An example that shows how CLEs are dependent on active edges (see text).

(d) An example that shows how REs are dependent on active edges (see text).

(e) Simulation of DCM.

Figure 5: Simple examples for explaining the faster version of our algorithm.
edge or edges that were added to the MSF, and determine which subsequent decisions might be altered by this change. To make this efficient, we maintain data structures that keep track of the dependencies between various decisions we have made. It is important to note that these data structures need to be constructed only once, for all $O(N^2)$ questions whose answers we simulate. For each question, we only need to simulate changes to the data structures that can occur in simulating the changes to the cluster that this question induces; these changes are typically not large.

**Store simple path between all pairs of points**: Store-simple-path is a structure that stores the simple path between any pair of points in the same tree in the forest. A simple path in a graph is a path containing edges, where no vertices repeat. In a tree a simple path between any two points is unique. We update Store-simple-path whenever we add an edge while running HACC. For example in figure 3(a) when we add an edge between trees 1 and 2 (i.e., add an edge between C and F), we can store the simple paths between any pair of points in $T_1 \times T_2$, where $T_1 = \{A, B, C, D\}$ and $T_2 = \{E, F, G, H, J, L\}$. For example, the simple path between A and L is $\text{Store} - \text{simple} - \text{path}(A, L) = \text{Store} - \text{simple} - \text{path}(A, C) \cup \text{Store} - \text{simple} - \text{path}(F, L) \cup (C, F)$. Store-simple-path should already have stored the simple paths between A and C, and F and L.

**Active edge and Cannot Link edge dependency**: AEs and CLEs are mutually dependent. While we build a MSF, a CLE is not added because some set of active edges, with lower time stamps, were already present in the forest. If one or more of those active edges should be removed there is a possibility that a CLE can be added back to the MSF, without violating any constraints. In FAST-Active-HACC, when we simulate a new constraint, we look at which AEs must be removed to incorporate a simulated constraint and as a result which CLEs can be added to the forest. But these additions and removals of edges should be consistent with the given set of constraints including the one we are simulating.

We consider an example scenario in figure 4(c) to understand the modeling of dependency between AEs and CLEs. Let us assume we are building a MSF with edges from a sorted edge list E. We have added some edges to the forest. The next edge to be considered from the list for addition to MSF is an edge between B and F. B and F are part of trees TREE-1 and TREE-2 respectively. Although there is no cannot link constraint between B and F there are other cannot links between AE, CG and DH. So we do not add edge BF to MSF and include BF in set CLE. Say BF has a time stamp $t$. Now, BF was not added because a subset of active edges $\{AB, BC, CD, EF, FG, GH\}$ were added before time $t$. We create lists that will contain those active edges. Since there are three cannot links between trees TREE-1 and TREE-2, we have three lists for BF corresponding to each of the cannot links; we call them $LIST_{BF}^{AE}$, $LIST_{BF}^{CG}$, $LIST_{BF}^{DH}$. In $LIST_{AE}^{BF}$, we have $\{BC, CD, FG, GH\}$ in the list. We also have to keep track of the multiplicity of dependency, e.g., edges BC and FG are present in both $LIST_{CG}^{BF}$ and $LIST_{DH}^{BF}$.
So BC and FG have multiplicity of two. If we remove any one of BC or FG, we can delete two lists together. That implies that if either of BC or FG was not added when we built the MSF, we would not have had the cannot link restrictions due to cannot links between CG or DH. Now let us assume we remove active edge BC. So we can get rid of two lists. Still there is no chance we can add BF and make it active unless either AB or EF is inactive, because the cannot link constraint AE will prevent addition of BF. If either of those edges become inactive we can also delete the list $LIST_{AE}^{BF}$ and can add BF to the MSF.

So for all CLEs we keep lists of active edges and their multiplicity. Whenever we delete one active edge we see which of the lists can be deleted. If a cannot link edge is free from all lists it can be considered for addition to the MSF. However if a CLE itself is also a cannot link constraint we can not ever add that. Above subsection describes how removal of an active edge makes way for other CLEs to be added.

**Redundant edge and Cannot Link edge dependency:** A redundant edge is not added to the MSF if the points connected by the RE is already connected by a set of active edges. So we also note which of the active edges are responsible for not adding an RE.

Let us take an example in figure 4(d), where we have a set of active edges already added to the MSF. Now if AF is the next edge from the sorted list E to be considered, we cannot add that edge as it will create a cycle. So AF is included in RE. Now when we simulate constraints if one of the active edges from \{AB, BC, CD, DE, EF\} is removed, can we add AF to MSF? No, if all other edges between A and F in TREE-1 remain active. Let us assume active edge CD is removed to satisfy some cannot link constraint. Now we have two subtrees TREE-11 and TREE-12. If redundant edge EF is added in this scenario, the constraint for which CD was removed would be violated again. But if any other active edges from \{AB, BC, DE, EF\} is removed AF comes under consideration for possible addition to MSF. Now we may wonder what happens if any other edge from \{BG, BH, DI, EJ\} is removed. If that gets rid of the constraint for which CD was removed, we would add CD back and will not add AF, because AF has a higher time stamp than CD. So we don’t worry about any other edge other than the active edges on the simple path between A and F. If two or more active edges from that path are removed we start considering AF. So we keep a separate list of active edges for all redundant edges and see if two or more edges are removed from the list.

Now we have discussed how removal of active edges bring CLEs and REs into consideration. But when we add an edge to the MSF from CLE or RE, how does this change active edges? At a high level, whenever we are about to add a CLE or RE, we remove all active edges, with a later time stamp than that of the CLE/RE, from the two trees connected by that CLE/RE. However we reconsider them again for addition to the MSF. This is pretty fast in practice. The main reason is, addition or removal of edges is very local, so in reality we do not need to remove or consider adding huge number of edges.
**Priority Queue (PQ):** This is the most important structure we use while we simulate a SCC or a DCM. In this queue we store possible edges to be added in the MSF but sorted based on their time stamps. But there are some additional properties of the elements in this queue and we describe them below:

- Edges in the priority queue must not be present in the MSF.
- The MSF should always be consistent. That means it can have more or less than K clusters but all the must link and cannot link constraints must be satisfied.
- Once we start popping elements from the priority queue, if we have considered adding an edge at time $t$, we never look at an edge with time stamp $T \leq t$. That implies any other edge with $T \leq t$ can not be added to the priority queue from now on.

In our framework we keep a priority queue with a set of edges, that can possibly be added to the MSF. We want to minimize the total number of elements (say $|PQ|$) added to PQ. The minimum set is the set of edges, which we can definitely add to MSF. Although it is easier to find which of the CLE/REs are affected when we remove an active edge, tracking which of active edges could be affected while we add a CLE/RE is complicated. It is hard to find a polynomial time precomputation process for this part, which can be computed once and used for all possible question simulations. The precomputation in that case involves taking care of all possible cycles of a complete graph, which could take exponential time. So we have a simple work around, that works well in practice. We try to minimize $|PQ|$ as much as possible but we do not try to get the exact minimum of $|PQ|$ using our algorithm. We make sure the algorithm is correct, but we may do some extra computation.

### 3.4.3 Primitive Operations

These are some set of basic operations which we perform while we simulate the SCC or DCM.

- **Insert with priority:** Add an edge to the priority queue with the priority based on time.
- **Pop highest priority element:** Pop out an edge from the priority queue which has the lowest time stamp.
- **Add an edge to the MSF:** Consider an edge to add to the MSF such that all constraints are satisfied and if added, relabel necessary nodes.
- **Remove an edge from MSF:** Remove an edge and relabel nodes.
3.4.4 Same Cluster Cannot Links (SCC)

In this section, we consider an additional optimization that speeds up the simulation of adding a cannot link edge between two edges from the same cluster. In fact, we do not need to consider possible pairs within the same cluster separately. It turns out that many possible cannot link edges, if added, will have exactly the same effect. If we have \( n \) points within a cluster we should simulate the results of cannot link edges between \( \binom{n}{2} \) possible pairs. But in reality we have to simulate only \( (n-1) \) pairs. Let us take an example in figure 4(b) to explain this. The edge weights are shown on the edges. Let us assume we want to simulate the effects of a cannot link constraint between D and E. Now if there is a CL between D and E, all the active edges would have been added except CF, because CF is the highest weighted edge. When we would have tried to add CF, the cannot link constraint between D and E would stop addition of CF. Now let us assume we want to simulate a cannot link constraint between A and L. Again we would not have added the edge CF because it has the maximum weight on a path connecting A and L. So in both of these cases we have to remove CF to simulate a cannot link constraint. Now we can observe that if we want to simulate a cannot link constraint between any pair \( \{A, B, C, D\} \times \{E, F, G, H, J, L\} \), we have to remove CF, because this is the maximum weighted edge on any simple path between a pair of points from \( T_1 \) and \( T_2 \). For all of those pairs we would ultimately not add edge CF. So we can proceed like this: We find the maximum weighted edge (say \( E_M \) in a tree, find the subtrees (say \( T_1 \) and \( T_2 \)) connected by this edge. For all possible pairs in \( T_1 \times T_2 \), we would remove edge \( E_M \). Now we can proceed to the next maximum weighted edge and do the same thing, but we don’t update simulation of cannot link constraints for pairs that we have already considered. We continue doing this till we reach the least weighted edge. In this way, by simulating the effects of removing \( n \) possible edges, we determine the effects of every possible cannot link edge that could be added to this cluster.

While we simulate a SCC or removal of an edge, we first remove that edge (say time stamp of this edge is \( t_M \)) and then run a module QUICK-SIMULATE to complete the simulation of SCC. We will provide a brief high level idea and detailed pseudo code of this part later. We will use the same module also in DCM simulation.

3.4.5 Different Cluster Must Links (DCM)

When we simulate the must links between different clusters finding the set of active edges to be removed is harder. However we have precomputed some data before the start of the simulation to make things easier. We will refer to figure 4(e) to explain which of the active edges should be removed to add a DCM. Let us assume we would like to add an ML between B and F. There are three CLs between TREE-1 and TREE-2. So if the cannot link constraint AE and simulated must link constraint...
BF both have to be satisfied one of the active edges on each simple path (in this particular example each simple path is just an edge) AB or EF must be removed. The higher weight edge will be removed for the same reasons we mentioned in our discussion of SCC. So while we simulate a DCM, we find the maximum weighted edges on the union of simple paths between the cannot link ends and simulated must link ends. So if there are three cannot links between TREE-1 and TREE-2 we will have three such paths and we have to find the maximum weighted edge in each path. Since we already have the simple paths calculated between any pair of points in a tree, this process is fast. In our example three paths are, \{AB, EF\}, \{BC, FG\} and \{BC, CD, EG, GH\}. The set of edges to be removed are \(RS = \{BC, CD, EF\}\). Now we start removing edges with the minimum time stamped edge being removed first. So we remove BC first. We also see the fact that BC has multiplicity two and if we remove BC we do not need to remove CD. Then we can remove EF. Also assume the minimum time stamp of RS is \(t_M\). Unlike SCC, here we will get a set of edges to be removed that can contain more than one element. Now we would run the module **QUICK-SIMULATE** to complete the simulation of DCM.

### 3.4.6 QUICK-SIMULATE

This is a module which is used both for SCC and DCM. Once we remove the set of required edges, we can run this module. Now whatever happened with HACC before time stamp \(t_M\) will remain the same in the simulation. We know if an active edge is removed, which CLE/REs come under consideration. We insert them into the priority queue. Now we pop the first element (say time stamp \(t\)) from the priority queue and try to add that into the forest. We should remember again any edge with time stamp \(T \leq t\), will not be included in priority queue in future. Now, if the popped edge is a CLE or RE with time stamp \(t\), we remove all the active edges (with time stamp \(T \geq t\)) in the two subtrees, which are going to be connected by that CLE/RE and put them into the priority queue. We also see if we can add any other CLE/RE due to removal of these edges. One of our observations is that if we don’t remove the active edges with \(T \geq t\), sometimes we may end up with a different clustering than HACC would obtain. As we mentioned earlier this does not take lot of time in real experiments because we remove a few edges very locally and the total number of edges to be removed in this process is very small. We continue doing this until we do not have any element in the priority queue.

Then, if we have more clusters than required, we pull the minimum time stamped element from OE and add that to the forest. If we have less than the required number of clusters we remove the maximum weighted edge from the forest.

We get the clustering but we are not done yet. To complete the simulation, we need to compute the change in the Relative Jaccard’s coefficient. Calculation of Jaccard’s coefficient is complex, i.e, we have to do \(O(N^2)\) work. But we can easily keep track of which of the points change assignments or move from one cluster
to another. We use that information to calculate Jaccard’s coefficient and we have empirically seen on an average it does not take more than $O(N)$ time.

**Algorithm 2 QUICK-SIMULATE**

Given: Priority-Queue with edges to be removed

while Priority-Queue $\neq \phi$ do

Pop the first element $e$ from Priority-Queue

$e$ connects clusters $C_1$ and $C_2$ and has time stamp $t$

if $e$ is in CLE or RE then

Remove all active edges with time stamp $T \geq t$ from $C_1$ and $C_2$

Insert them into Priority-Queue

end if

end while

If required remove the last added edge or add more edges from OE to maintain K clusters

Once we have found the effect of simulation of all possible constraints, we select the pair with maximum expected change in the Relative Jaccard’s Coefficient. We present that pair to some user, get feedback and continue doing this until our human budget is exhausted or we get reasonable clustering. Since FAST-Active-HACC is complex, code for this will be made publicly available for use.

4 Experimental Results

We have experimented in two different domains, leaf images and face images. For leaves, we create three subsets from a huge corpora of leaf images used for Leafsnap [1]. All leaf images are iPhone images of leaves on a white background. The leaf subsets are called Leaf-100 (containing 100 images from 10 different species), Leaf-250 (250 images from 25 different species) and Leaf-1042 (1042 images from 62 species). Leaf-100 and Leaf-250 have same number of leaves from all the species. But in Leaf-1042, the number of leaves in each species vary from 4 to 31. Similarly for faces, we have extracted three subsets of images from a face dataset called PubFig [13]. The PubFig database is a large, real-world face dataset consisting of 58,797 images of 200 people collected from the Internet. Unlike most other existing face datasets, these images are taken in completely uncontrolled situations with non-cooperative subjects. Thus there is large variation in pose, lighting, expression, scene, camera, and imaging conditions, etc. The subsets of images are called Face-100 (100 images from 10 different people), Face-250 (250 images from 25 different people) and Face-500 (500 images from 50 different people). All of these face datasets have same number of images in each class.

The distance matrix for face images and leaf images are calculated based on
**Table 1: Summary of the algorithms that we compare.**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active-HACC</td>
<td>Proposed active learning version for Image Clustering</td>
</tr>
<tr>
<td>FAST-Active-HACC</td>
<td>Faster version of our proposed algorithm without any heuristic</td>
</tr>
<tr>
<td>FAST-Active-HACC-H1</td>
<td>Faster version of our proposed algorithm with H1 only</td>
</tr>
<tr>
<td>FAST-Active-HACC-H1_H2</td>
<td>Faster version of our proposed algorithm with H1 and H2 both</td>
</tr>
<tr>
<td>Random Constraints</td>
<td>Seek pairwise constraints randomly and use HACC</td>
</tr>
<tr>
<td>CACTI [7]</td>
<td>A heuristic to find the best pair</td>
</tr>
</tbody>
</table>

We have run our algorithm on all of the datasets described above. All the algorithms that we compare are described in Table 1. Figures 6a-6f (where Jaccard’s Coefficient corresponding to one misclassification per cluster is displayed using green squares) show performance evaluations of all the algorithms on Leaf-100, Face-100, Leaf-250, Face-250, Leaf-1042 and Face-500. We use S=100 to find the pairwise probability distributions. We see how Jaccard’s Coefficient changes with the number of questions. Since we have the ground truth for these datasets we were able to calculate the Jaccard’s Coefficient with respect to the ground truth. In real world situations we will not have the ground truth and will have to decide when we have reached a good clustering. One possible way to stop asking questions is when clustering assignments do not change even with additional constraints. Also,
one of the advantages of FAST-Active-HACC is that we do not need to set any parameters. One of our main interests is in problems that grow big because the number of clusters becomes large. We make the following observations from the experiments:

- In all of these experiments our algorithm significantly outperforms all other algorithms. We were able to greatly reduce the number of constraints required for a reasonable clustering.
- We run both Active-HACC and FAST-Active-HACC for the Leaf-100 and Face-100 datasets. We see that FAST-Active-HACC is 25-30 times faster than Active-HACC for a dataset of size 100. Overall we expect FAST-Active-HACC to be $O(N)$ faster than Active-HACC. Since Active-HACC is slow, we could not experiment with it in larger datasets. We also observe that FAST-Active-HACC-H1 produces the exact same results as FAST-Active-HACC for Leaf-100/Face-100. For a dataset of size 1042, FAST-Active-HACC-H1,H2 takes around a minute per question. We believe this could be further sped up with more optimized code (our current implementation is in MATLAB) or parallel processing as our algorithm is highly parallelizable.
- In Figure 6c, we compare the results for different algorithms for the Leaf-1042 dataset. For this dataset only we use K-means initially as part of $H_1$, to reduce the number of points to 700. Even with that, we get Jaccard's Coefficient of 0.8152 (one misclassification per cluster on average) by asking just 3782 questions.
- We wanted to compare our algorithm with [9] and [11], but a direct comparison on our image datasets is not possible due to the complexity of their algorithm and the lack of publicly available code. However we also run experiments using the iris dataset to compare with [9], on which they have reported results. This is a relatively easy dataset with 150 points from 3 clusters in 4 dimensions. They have used the ratio of well categorized points to the total number of points as an evaluation metric (let us call it RWCP). Our algorithm reaches RWCP of 0.97 within three questions, where they take ten questions to reach the same RWCP.
- One of the major differences in these domains is the distance matrix. In leaf images the distance matrix is more accurate than in face images. So even if we have two similar datasets from two different domains, we need more questions for face images than leaf images. For smaller datasets in which the distance matrix is not very accurate, FAST-Active-HACC-H1,H2 becomes comparable to, though still better than [3].

5 Conclusions

We have presented an approach for image clustering that incorporates pairwise constraints from humans. An algorithm is developed for choosing data pairs to
Figure 6: Performance plot showing how the Jaccard’s Coefficient increase with the number of questions. Our proposed algorithm Active-HACC significantly outperforms all other algorithms we compare.
use when querying a person for additional constraints. Since a brute force version of our algorithm is time consuming we also formulate a more complex but faster version in this paper. Our algorithm outperforms all state-of-the-art results in image clustering. Although this paper was focused on solving image clustering, this idea could be extended to any clustering domain in general.

References


