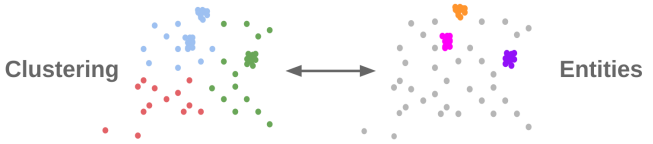
Finding Frequent Entities in Continuous Data

Ferran Alet, Rohan Chitnis, Leslie P. Kaelbling and **Tomás Lozano-Pérez**
MIT Computer Science and Artificial Intelligence Laboratory
{alet, ronuchit, lpk, tlp}@mit.edu

Abstract

In many applications that involve processing high-dimensional data, it is important to identify a small set of entities that account for a significant fraction of detections. Rather than formalize this as a clustering problem, in which all detections must be grouped into hard or soft categories, we formalize it as an instance of the *frequent items* or *heavy hitters* problem, which finds groups of tightly clustered objects that have a high density in the feature space. We show that the *heavy hitters* formulation generates solutions that are more accurate and effective than the clustering formulation. In addition, we present a novel online algorithm for heavy hitters, called HAC, which addresses problems in continuous space, and demonstrate its effectiveness on real video and household domains.



(a) Noisy environment: outliers (red points) greatly influence clustering. Entities, defined locally, are robust to large amounts of noise.

(b) Groups of different sizes: clustering tends to join nearby groups; entities may be close together yet distinct.

Figure 1: In clustering *all/most* points belong to a group, forming *big* clusters which are defined *globally*. In entity finding *some* points belong to a group, forming *small tight* regions defined *locally*.

1 Introduction

Many applications require finding entities in raw data, such as individual objects or people in image streams or particular speakers in audio streams. Often, entity-finding tasks are addressed by applying clustering algorithms such as k-means (for instance in [Niebles *et al.*, 2008]). We argue that instead they should be approached as instances of the *frequent items problem*, also known as the *heavy hitters problem*. The classic frequent items problem assumes discrete data and involves finding the most frequently occurring items in a stream of data. We propose to generalize it to continuous data.

Figure 1 shows examples of the differences between clustering and entity finding. Some clustering algorithms fit a global objective assigning *all/most* points to centers, whereas entities are defined *locally* leading to more robustness to noise (1a). Others, join nearby dense groups while trying to detect sparse groups, whereas entities are still distinct (1b). These scenarios are common because real world data is often noisy and group sizes are often very unbalanced [Newman, 2005].

We characterize entities using two natural properties: **similarity** - the feature vectors should be similar according to some (not necessarily Euclidean) distance measure, such as cosine distance, and **salience** - the region should include a sufficient number of detections over time.

Even though our problem is not well-formulated as a clustering problem, it might be tempting to apply clustering algorithms to it. Clustering algorithms optimize for a related, but different, objective. This makes them less accurate for our problem; moreover, our formulation overcomes typical limitations of some clustering algorithms such as relying on the Euclidean distance metric and performing poorly in high-dimensional spaces. This is important because many natural embeddings, specially those coming from Neural Networks, are in high dimensions and use non-Euclidean metrics.

In this paper we suggest addressing the problem of entity finding as an extension of heavy hitters, instead of clustering, and propose an algorithm called HAC with multiple desirable properties: handles an online stream of data; is guaranteed to place output points near high-density regions in feature space; is guaranteed to *not* place output points near low-density regions (i.e., is robust to noise); works with *any* distance metric; can be time-scaled, weighting recent points more; is easy to implement; and is easily parallelizable.

We begin by outlining a real-world application of tracking important objects in a household setting without any labeled data and discussing related work. We go on to describe the algorithm and its formal guarantees and describe experiments that find the main characters in video of a TV show and that address the household object-finding problem.

1.1 Household Setting

The availability of low-cost, network-connected cameras provides an opportunity to improve the quality of life for people with special needs, such as the elderly or the blind. One application is helping people to find misplaced objects.

More concretely, consider a set of cameras recording video streams from some scene, such as a room, an apartment or a shop. At any time, the system may be queried with an image or a word representing an object, and it has to answer with candidate positions for that object. Typical queries might be: *"Where are my keys?"* or *"Warn me if I leave without my phone."* Note that, in general, the system won't know the query until it is asked and thus cannot know which objects in the scene it has to track. For such an application, it is important for the system to not need specialized training for every new object that might be the focus of a query.

Our premise is that images of interesting objects are such that 1) a neural network embedding [Donahue *et al.*, 2014; Johnson *et al.*, 2016; Mikolov *et al.*, 2013] will place them close together in feature space, and 2) their position stays constant most of the time, but changes occasionally. Therefore objects will form high-density regions in a combined feature×position space. Random noise, such as people moving or false positive object detections, will not form dense regions. Objects that don't move (walls, sofas, etc) will be always dense; interesting objects create dense regions in feature×position space, but eventually change position and form a new dense region somewhere else. We will exploit the fact that our algorithm is easy to scale in time, to detect theses changes over time.

1.2 Related Work

Our algorithm, HAC, addresses the natural generalization of *heavy hitters*, a very well-studied problem, to continuous settings. In heavy hitters we receive a stream of elements from a discrete vocabulary and our goal is to estimate the most frequently occurring elements using a small amount of memory, which does not grow with the size of the input. Optimal algorithms have been found for several classes of heavy hitters, which are a logarithmic factor faster than our algorithm, but they are all restricted to *discrete* elements [Manku and Motwani, 2002]. In our use case (embeddings of real-valued data), elements are not drawn from a discrete set, and thus repetitions have to be defined using regions and distance metrics. Another line of work [Chen and Zhang, 2016] estimates the total number of different elements in the data, in contrast to HAC that finds (not merely counts) different *dense* regions.

Our problem bears some similarity to clustering but the problems are fundamentally different (see figure 1). The closest work to ours within the clustering literature is *density-based (DB) clustering*. In particular, they first find all dense regions in space (as we do) and then join points via paths in those dense regions to find arbitrarily-shaped clusters. In contrast, we only care about whether a point belongs to one of the dense regions. This simplification has two advantages: first, it prevents joining two close-by entities, second, it allows much more efficient, general and simple methods.

The literature on DB clustering is very extensive. Most of the popular algorithms, such as DBScan [Ester *et al.*, 1996] and Level Set Tree Clustering [Chaudhuri and Dasgupta, 2010], as well as more recent algorithms [Rodriguez and Laio, 2014], require simultaneous access to all points and have complexity quadratic in the number of points; this makes them impractical for big datasets and specially streaming data. There are some online DB clustering algorithms [Chen and Tu, 2007], [Wan *et al.*, 2009], [Cao *et al.*, 2006], but they either tessellate the space or assume a small timescale, tending to work poorly for non-Euclidean metrics and high dimensions.

Two pieces of work join ideas from clustering with *heavy hitters*, albeit in very different settings and with different goals. [Larsen *et al.*, 2016] uses graph partitioning to attack the discrete l_p *heavy hitters* problem in the general turnstile model. [Braverman *et al.*, 2017] query a *heavy hitter* algorithm in a tessellation of a high dimensional discrete space, to find a coreset which allows them to compute an approximate k-medians algorithm in polynomial time. Both papers tackle streams with discrete elements and either use clustering as an intermediate step to compute *heavy hitters* or use *heavy hitters* as an intermediate step to do clustering (k-medians). In contrast, we make a connection pointing out that the generalization of *heavy hitters* to continuous spaces allows us to do *entity finding*, previously seen as a clustering problem.

We illustrate our algorithm in some applications that have been addressed using different methods. Clustering faces is a well-studied problem with commercially deployed solutions. However, these applications generally assume we care about most faces in the dataset and that faces occur in natural positions. This is not the case for many real-world applications, where photos are taken in motion from multiple angles and are often blurry. Therefore, algorithms that use clustering in the conventional sense, [Schroff *et al.*, 2015; Otto *et al.*, 2017], do not apply.

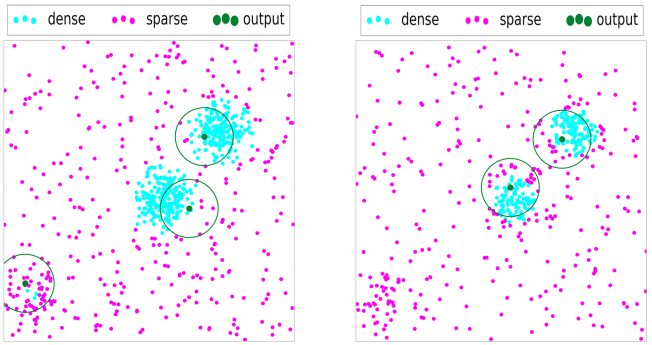
[Rituerto *et al.*, 2016] proposed using *DB*-clustering in a setting similar to our object localization application. However, since our algorithm is online, we allow objects to change position over time. Their method, which uses DBScan, can be used to detect what we will call *stable objects*, but not movable ones (which are generally what we want to find). [Nirjon and Stankovic, 2012] built a system that tracks objects assuming they will only change position when interacting with a human. However, they need an object database, which makes the problem easier and the system much less practical, as the human has to register every object to be tracked.

2 Problem Setting

In this section we argue that random sampling is surprisingly effective (both theoretically and experimentally) at finding entities by detecting *dense* regions in space and describe an algorithm for doing so in an online way. The following definitions are of critical importance.

Definition 2.1. Let $d(\cdot,\cdot)$ be the distance metric. A point p is (r, f)-*dense* with respect to dataset $\mathcal{D}$ if the subset of points in $\mathcal{D}$ within distance r of p represents a fraction of the points that is at least f. If $N = |\mathcal{D}|$; then p must satisfy:

$$|\{x \in \mathcal{D} \mid d(x,p) \leq r\}| \geq fN.$$



(a) $f = 7\%$ (b) $f = 15\%$

Figure 2: Varying fraction f with fixed radius r. Data comes from 3 Gaussians plus uniform random noise. A circle of radius r near the sparsest Gaussian captures more than 7% of the data but less than 15%; thus being dense in (a), but not in (b).

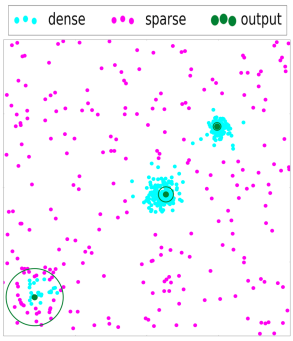


Figure 3: Varying radius r with fixed frequency f. We can detect Gaussians with different variances by customizing r for each output. The goal *isn't* to cover the whole group with the circle but to return the smallest radius that contains a fraction f of the data. Points near an output are guaranteed to need a similar radius to contain the same fraction of data.

Definition 2.2. A point p is (r, f)-*sparse* with respect to dataset $\mathcal{D}$ if and only if it is not (r, f)-*dense*.

The basic version of our problem is the natural generalization of *heavy hitters* to continuous spaces. Given a metric d, a frequency threshold f, a radius r and a stream of points $\mathcal{D}$, after each input point the output is a set of points. Every (r, f)-*dense* point (even those not in the dataset) has to be close to at least one output point and every $(r, f/2)$-*sparse* region has to be far away from all output points.

Our algorithm is based on samples that *hop* between data points and *count* points nearby; we therefore call it *Hop And Count (*HAC*)*.

2.1 Description of the Algorithm

A very simple *non*-online algorithm to detect dense regions is to take a random sample of m elements and output only those samples that satisfy the definition of (r, f)-*dense* with respect to the whole data set. For a large enough m, each dense region in the data will contain at least one of the samples with high probability, so the output will include a sample from this region. For sparse regions, even if they contain a sampled point, this sample will not be in the output since it will not pass the denseness test.

Let us try to make this into an online algorithm. A known way to maintain a uniform distribution in an online fashion is *reservoir sampling* [Vitter, 1985]: we keep m *stored samples*. After the i-th point arrives, each sample changes, independently, with probability $1/i$ to this new point. At each time step, samples are uniformly distributed over all the points in the data. However, once a sample has been drawn we cannot go back and check whether it belongs to a dense or sparse region of space, since we have not kept all points in memory.

The solution is to keep a counter for each sample in memory and update the counters every time a new point arrives. In particular, for any sample x in memory, when a new point p arrives we check whether $d(x, p) \leq r$; if so, we increase x's counter by 1. When the sample hops to a new point x', the counter is no longer meaningful and we set it to 0.

Since we are in the online setting, every sample only sees points that arrived after it and thus only the first point in a region sees all other points in that region. Therefore, if we want to detect a region containing a fraction f of the data, we have to introduce an acceptance threshold lower than f, for example $f/2$, and only output points with frequency above it. The probability of a sample being in the first half of any dense region is at least $f/2$ and thus, for a large enough number of samples m, with high probability every dense region will contain a sample detected as dense. Moreover, since we set the acceptance threshold to $f/2$, regions much sparser than f will not produce any output points. In other words, we will have false positives but they will be *good* false positives, since those points are guaranteed to be in regions almost as dense as the target dense regions we actually care about. In general we can change $f/2$ to $(1 - \epsilon)f$ by ϵ trading memory for performance. Finally, note that this algorithm is easy to parallelize because all samples and their counters are independent.

2.2 Multiple Radii

In the previous section we assumed a specific known threshold r. What if we don't know r, or if every dense region has a different diameter? We can simply have counts for multiple values of r for every sample. In particular, for every x in memory we maintain a count of streamed points within distance r for every $r \in \{r_0 = r_{\min}, r_0\gamma, r_0\gamma^2, \ldots, r_0\gamma^c = r_{\max}\}$. At output time we can output the smallest r_i such that the x is (r_i, f)-*dense*. With this exponential sequence we guarantee a constant-factor error while only losing a logarithmic factor in memory usage. r_0 and c may be user-specified or automatically adjusted at runtime.

Algorithm 1 shows the pseudo-code for HAC with multiple specified radii. Note that the only data-dependent parameters are r_0 and c, which specify the minimum and maximum radii, and f_0 which specifies the minimum fraction that we will be able to query. The other parameters (ϵ, δ, γ) trade off memory vs. probability of satisfying guarantees.

2.3 Guarantees

We make a guarantee for every dense or sparse point in space, even those that are not in the dataset. Our guarantees are probabilistic; they hold with probability $1 - \delta$ where δ is a

Algorithm 1: Hop And Count with multiple radii.

Subroutine *Hop And Count Processing*(f_0, ϵ, δ, r_0, γ, c)
  $m \leftarrow \log(f_0^{-1}\delta^{-1})/f_0\epsilon$ // to satisfy guarantees
  $Mem \leftarrow [\emptyset, \overset{(m)}{\ldots}, \emptyset]$; $Counts \leftarrow Zeros(m, c)$
  $t = 0$
  for p *in stream* **do**
    $t += 1$
    for $0 \le i \le m$ **do**
      if $Bernoulli(1/t)$ **then**
        $Mem[i] \leftarrow p$ // hop
        for $0 \le r \le c$ **do**
          $Counts[i][r] \leftarrow 0$ // reset counters
      $r \leftarrow$ **max** $\left(0, \text{ceil}\left(\log_\gamma\left(d(Mem[i], p)/r_0\right)\right)\right)$
      if $r \le c$ **then**
        $Counts[i][r] += 1$
Subroutine *Hop And Count Query*(f, t, ϵ, Mem, $Counts$)
  for $0 \le i < len(Counts)$ **do** // $0 \le i < m$
    $count \leftarrow 0$
    for $0 \le r < len(Counts[i])$ **do** // $0 \le r < c$
      $count \leftarrow count + Mem[i][r]$
      if $count \ge (1-\epsilon)ft$ **then**
        output $(Mem[i], r)$
        break

parameter of the algorithm that affects the memory usage. We have three types of guarantees, from loose but very certain, to tighter but less certain. For simplicity, we assume here that $r_{\min} = r_{\max} = r$. Here, we state the theorems; proofs are available in the online version with an appendix at http://lis.csail.mit.edu/alet/entities-appendix.html.

Definition 2.3. $r_f(p)$ is the smallest r s.t. p is (r, f)-dense. For each point p we refer to its *circle/ball* as the sphere of radius $r_f(p)$ centered at p.

Theorem 2.1. *For any tuple $(\epsilon < 1, \delta, f)$, with probability $1-\delta$, for any point p s.t. $r_f \le r_{\max}/2\gamma$ our algorithm will give an output point o s.t. $d(o,p) \le 3r_f(p)$.*
Moreover, the algorithm always needs at most $\Theta\left(\frac{\log(f\delta)}{\epsilon f}\log_\gamma\left(\frac{r_{\max}}{r_{\min}}\right)\right)$ memory and $\Theta(\frac{\log(f\delta)}{\epsilon f})$ time per point. Finally, it outputs at most $\Theta\left(\frac{\log(f\delta)}{\epsilon f}\right)$ points.

Lemma 2.2. *Any $(\Delta, (1-\epsilon)f)$-sparse point will not have an output point within $\Delta - 2r_{\max}$.*

Notice that we can use this algorithm as a noise detector with provable guarantees. Any $(r_{\max}, f)$-*dense* point will be within $3r_{\max}$ of an output point and any $(5r_{\max}, (1-\epsilon)f)$-*sparse* point will not.

Theorem 2.3. *For any tuple (ϵ, δ, f), with probability $(1-\delta)$, for any (r, f)-dense point p our algorithm will output a point o s.t. $d(o,p) \le r$ with probability at least $(1-\delta f)$.*

Theorem 2.4. *We can apply a post-processing algorithm that takes parameter γ in time $\Theta\left(\frac{\log(f\delta)}{\epsilon f^2}\right)$ to reduce the number of output points to $(1+2\epsilon)/f$ while guaranteeing that for any point p there is an output within $(4\gamma+3)r_f(p)$. The same algorithm guarantees that for any (r_{max}, f)-dense point there will be an output within $7r_{max}$.*

Note that the number of outputs can be arbitrarily close to the optimal $1/f$.

The post-processing algorithm is simple: iterate through the original outputs in increasing $r_f(p)$. Add p to the final list of outputs O if there is no $o \in O$ s.t. $d(o,p) \le r_f(p) + r_f(o)$. See the linked appendix for a proof of correctness.

In high dimensions many clustering algorithms fail; in contrast, our performance can be shown to be provably good in high dimensions. We prove asymptotically good performance for dimension $d \to \infty$ with a convergence fast enough to be meaningful in real applications.

Theorem 2.5. *With certain technical assumptions on the data distribution, if we run* HAC *in high dimension d, for any $(r, 1.05f)$-dense point there will be an output point within $(1+\alpha)r$, with $\alpha = O(d^{-1/2})$, with probability $(0.95 - \delta f - O(e^{-fn}))$, where n is the total number of datapoints.*
Moreover, the probability that a point p is $(r, 0.98(1-\epsilon)f)$-sparse yet has an output nearby is at most $0.05 + O(e^{-fn})$.

We refer the reader to the linked appendix for a more detailed definition of the theorem and its proof.

The intuition behind the proof is the following: let us model the dataset as a set of high-dimensional Gaussians plus uniform noise. It is well-known that most points drawn from a high dimensional Gaussian lie in a thin spherical shell. This implies that all points drawn from the same Gaussian will be similarly dense (have a similar $r_f(p)$) and will either all be *dense* or all *sparse*. Therefore, if a point is (r, f)-*dense* it is likely that another point from the same Gaussian will be an output and will have a similar radius. Conversely, a point that is $(r, (1-\epsilon)f)$-*sparse* likely belongs to a *sparse* Gaussian and no point in that Gaussian can be detected as *dense*.

Note that, for $d, n \to \infty$ and $\delta \to 0$ the theorem guarantees that any $(r, f)-$*dense* point will have an output within r with probability 95% and any $(r, (1-\epsilon))$-*sparse* point will not, with probability 5%; close to the ideal guarantees. Furthermore, in the appendix we show how these guarantees are non-vacuous for values as small as $n = 5000, d = 128$: the values of the dataset in section 3.

2.4 Time Scaling

We have described a time-independent version of HAC in which all points have equal weight, regardless of when they arrive. However, it is simple and useful to extend this algorithm to make point i have weight proportional to $e^{-(t-t_i)/\tau}$ for any timescale τ, where t is the current time and t_i is the time when point i was inserted.

Trivially, a point inserted right now will still have weight 1. Now, let t' be the time of the last inserted point. We can update all the weights of the previously received points by a factor $e^{-(t-t')/\tau}$. Since all the weights are multiplied by the same factor, sums of weights can also be updated by multiplying by $e^{-(t-t')/\tau}$.

We now only need to worry about hops. We can keep a counter for the total weight of the points received until now. Let us define $w_{j,k}$ as the weight of point p_j at the time point k arrives. Since we want to have a uniform distribution over those weights, when the i-th point arrives we simply assign

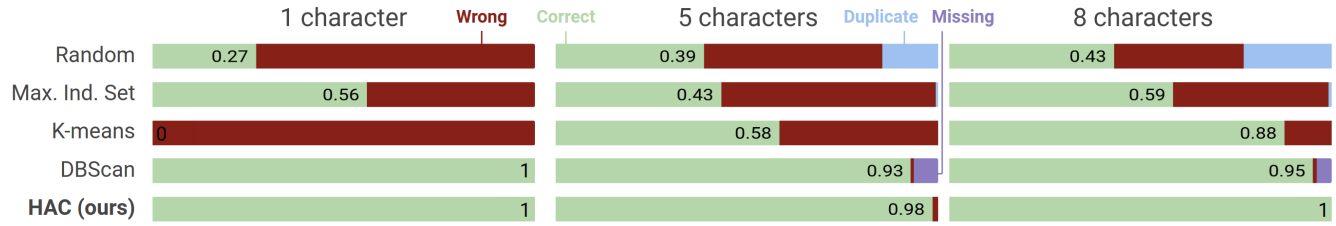


Figure 4: Identifying the main n characters for $n \in \{1, 5, 8\}$. We ask each algorithm to give n outputs and compute the fraction of main n characters found in those n outputs. We report the average of 25 different random seeds sampling the original dataset for 70% of the data. There are 3 ways of missing: Wrong: a noisy image (such as figure 5a) or unpopular character, Duplicate: an extra copy of a popular character, Missing: the algorithm is unable to generate enough outputs. Despite being online, HAC outperforms all baselines.

the probability of hopping to be $1/\sum_{j\leq i} w_{j,i}$. Note that for the previous case of all weights being 1 (i.e. $\tau = \infty$) this reduces to a probability of $1/i$ as before.

We prove in the appendix that by updating the weights and modifying the hopping probability, the time-scaled version has guarantees similar to the original ones.

2.5 Fixing the Number of Outputs

We currently have two ways of querying the system: 1) Fix a single distance r and a frequency threshold f, and get back all regions that are (r, f)-dense; 2) Fix a frequency f, and return a set of points $\{p_i\}$, each with a different radius $\{r_i\}$ s.t. a point p near output point p_i is guaranteed to have $r_f(p) \approx r_i$.

It is sometimes more convenient to fix the number of outputs instead. With HAC we go one step further and return a list of outputs sorted according to density (so, if you want o outputs, you pick the first o elements from the output list). Here are two ways of doing this: 1) Fix radius r. Find a set of outputs p_i each (r, f_i)-dense. Sort $\{p_i\}$ by decreasing f_i, thus returning the densest regions first. 2) Fix frequency f, sort the list of regions from smallest to biggest r. Note, however, that the algorithm is given a fixed memory size which governs the size of the possible outputs and the frequency guarantees.

In general, it is useful to apply duplicate removal. In our experiments we sort all (r, f)-*dense* outputs by decreasing f, and add a point to the final list of outputs if it is not within r_d of any previous point on the list. This is similar to but not exactly the same as the method in theorem 2.4; guarantees for this version can be proved in a similar way.

3 Identifying People

As a test of HAC's ability to find a few key entities in a large, noisy dataset, we analyze a season of the TV series *House M.D.*. We pick 1 frame per second and run a face-detection algorithm (dlib [King, 2009]) that finds faces in images and embeds them in a 128-dimensional space. Manually inspecting the dataset reveals a main character in 27% of the images, a main cast of four characters appearing in 6% each and three secondary characters in 4% each. Other characters account for 22% and poor detections (such as figure 5a) for 25%.

We run HAC with $r = 0.5$ and apply duplicate reduction with $r_d = 0.65$. These parameters were *not* fine-tuned; they were picked based on comments from the paper that created the CNN and on figure 6. We fix $\epsilon = \delta = 0.5$ for all experiments; these large values are sufficient because HAC works better in high dimensions than guaranteed by theorem 2.1.

(a) (b)

Figure 5: Shortcomings of clustering algorithms in *entity finding*. (a) The closest training example to the mean of the dataset (1-output of k-means) is a blurry misdetection. (b) *DBSCAN* merges different characters through paths of similar faces.

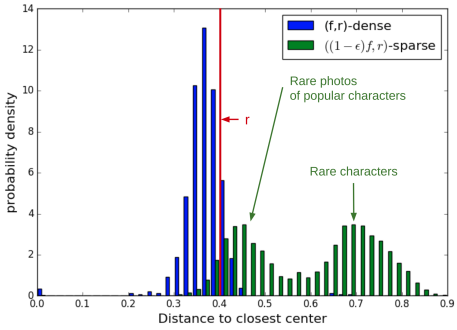


Figure 6: Most (r, f)-*dense* points are within r of an output, most $(r, (1-\epsilon)f)$-*sparse* points are not, as predicted by theorem 2.5. We run HAC with $f = 0.02$, $r = 0.4$. We compare two probability distributions: distance to the closest output for *dense* points and for *sparse* points. Ideally, we would want all the dense points (blue distribution) to be to the left of the threshold r and all the *sparse* points (green) to be to its right; which is almost the case. Moreover, notice the two peaks in the frequency distribution (intra-entity and inter-entity) with most uncertainty between 0.5 and 0.65.

We compare HAC against several baselines to find the most frequently occurring characters. For $n = \{1, 5, 8\}$ we ask each algorithm to return n outputs and check how many of the top n characters it returned. The simplest baseline, *Random*, returns a random sample of the data. *Maximal Independent*

Set starts with an empty list and iteratively picks a random point and adds it to the set iff it is at least $r = 0.65$ apart from all points in the list. We use *sklearn* [Pedregosa *et al.*, 2011] for both k-means and *DBSCAN*. *DBSCAN* has two parameters: we set its parameter r to 0.5, since its role is exactly the same as our r and grid-search to find the best ϵ. For k-means we return the image whose embedding is closer to each center and for *DBSCAN* we return a random image in each cluster.

As seen in figure 4, HAC consistently outperforms all baselines. In particular, k-means suffers from trying to account for most of the data, putting centers near unpopular characters or noisy images such as figure 5a. *DBSCAN*'s problem is more subtle: to detect secondary characters, the threshold frequency for being dense needs to be lowered to 4%. However, this creates a path of *dense regions* between two main characters, joining the two clusters (figure 5b).

While we used *offline* baselines with fine-tuned parameters, HAC is *online* and its parameters do not need to be fine-tuned. Succeeding even when put at a disadvantage, gives strong evidence that HAC is a better approach for the problem.

Finally, with this data we checked the guarantees of theorem 2.5: most (f, r)-dense points have an output within distance r, 95%, whereas few $(r, (1-\epsilon))$-sparse points do: 6%. This is shown in figure 6.

4 Object Localization

In this section we show an application of *entity finding* that cannot be easily achieved using clustering. We will need the flexibility of HAC: working online, with arbitrary metrics and in a time-scaled setting as old observations become irrelevant.

4.1 Identifying Objects

In the introduction we outlined an approach to object localization that does not require prior knowledge of which objects will be queried. To achieve this we exploit many of the characteristics of the HAC algorithm. We assume that: 1) A convolutional neural network embedding will place images of the same object close together and images of different objects far from each other. 2) Objects only change position when a human picks them up and places them somewhere else.

Points in the data stream are derived from images as follows. First, we use SharpMask[Pinheiro *et al.*, 2016] to segment the image into patches containing object candidates (figure 7). Since SharpMask is not trained on our objects, proposals are both unlabeled and very noisy. For every patch, we feed the RGB image into a CNN (Inception-V3 [Szegedy *et al.*, 2016]), obtaining a 2048-dimensional embedding. We then have 3 coordinates for the position (one indicates which camera is used, and then 2 indicate the pixel in that image).

We need a distance for this representation. It is natural to assume that two patches represent the same object if their embedding features are similar *and* they are close in the 3-D world. We can implement this with a metric that is the maximum between the distance in feature space and the distance in position space:

$$d((p_1, f_1), (p_2, f_2)) = \max(d_p(p_1, p_2), d_f(f_1, f_2))$$

We can use cosine distance for d_f and l_2 for d_p; HAC allows for the use of arbitrary metrics. However, for good performance, we need to scale the distances such that close in position space and close in feature space correspond to roughly similar numerical values.

Figure 7: All the candidate objects from a random camera and time. Only a few proposals (first 6) capture objects of actual interest.

We can now apply HAC to the resulting stream of points. In contrast to our previous experiment, time is now very important. In particular, if we run HAC with a large timescale τ_l and a small timescale τ_s, we'll have 3 types of detections:

- Noisy detections (humans passing through, false positive camera detections): not dense in either timescale;
- Detections from stable objects (sofas, walls, floor): dense in both timescales; and
- Detections from objects that move intermittently (keys, mugs): not dense in τ_l, and alternating dense and sparse in τ_s. (When a human picks up an object from a location, that region will become sparse; when the human places it somewhere else, a new region will become dense.)

We are mainly interested in the third type of detections.

4.2 Experiment: Relating Objects to Humans

We created a dataset of 8 humans moving objects around 20 different locations in a room.[1] Locations were spread across 4 tables with 8, 4, 4, 4 on each respectively. Each subject had a bag and followed a script with the following pattern: Move to the table of location A; Pick up the object in your location and put it in your bag; Move to the table of location B; Place the object in your bag at your current location.

The experiment was run in steps of 20 seconds: in the first 10 seconds humans performed actions, and in the last 10 seconds we recorded the scene without any actions happening. Since we're following a script and humans have finished their actions, during the latter 10 seconds we know the position of every object with an accuracy of 10 centimeters. The total recording lasted for 10 minutes and each human picked or placed an object an average of 12 times. In front of every table we used a cell phone camera to record that table (both human faces and objects on the table).

We can issue queries to the system such as: Which human has touched each object? Which objects have not been touched? Where can I find a particular object? Note that if the query had to be answered based on only the current camera image, two major issues would arise: 1) We would not know whether an object is relevant to a human. 2) We would not detect objects that are currently occluded.

[1] You can find it at http://lis.csail.mit.edu/alet/entities.html

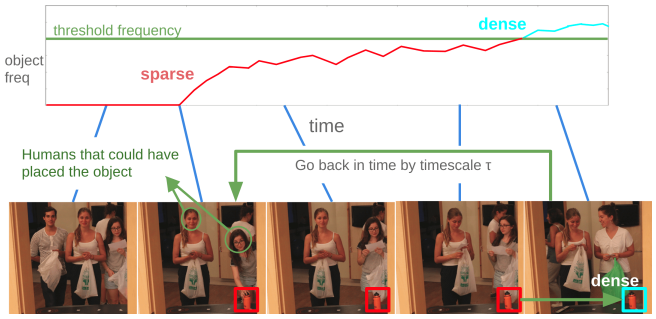


Figure 8: When an object is placed, its frequency starts growing. It takes on the order of the timescale τ to reach its stationary value, surpassing the threshold frequency. When an object becomes dense/sparse we assume a human placed/picked it, go τ back time and mark the pair $(obj, human)$. This system is completely unlabeled; obj and $human$ are both just feature vectors.

This experimental domain is quite challenging for several reasons: 1) The face detector only detects about half the faces. Moreover, false negatives are very correlated, sometimes missing a human for tens of seconds. 2) Two of the 8 subjects are identical twins. We have checked that the face detector can barely tell them apart. 3) The scenes are very cluttered: when an interaction happens, an average of 1.7 other people are present at the same table. 4) Cameras are 2D (no depth map) and the object proposals are very noisy.

We focus on answering the following query: for a given object, which human interacted with it the most? The algorithm doesn't know the queries in advance nor is it provided training data for particular objects or humans. Our approach, shown in figure 8, is as follows:

- Run HAC with $\tau_l = \infty$ (all points have the same weight regardless of their time), $\tau_s = 10$ seconds, $f = 2.5\%$ and a distance function and threshold which link two detections that happen roughly within 30 centimeters and have features that are close in embedding space.
- Every $10s$, query for outputs representing dense regions.
- For every step, look at all outputs from the algorithm and check which ones do not have any other outputs nearby in the previous step. Those are the detections that *appeared*. Similarly, look at the outputs from the previous step that do not have an output nearby in the current step; those are the ones that *disappeared*.
- (Figure 8) For any output point becoming dense/sparse on a given camera, we take its feature vector (and drop the position); call these features v and the current time t. We then retrieve all detected faces for that camera at times $[t - 2\tau_s, t - \tau_s]$, which is when a human should have either picked or placed the object that made the dense region appear/disappear. For any face f_i we add the pair (v, f_i) to a list with a score of $1/|f_i|$, which aims at distributing the responsibility of the action between the humans present.

Now, at query time we want to know how much each human interacted with each object. We pick a representative picture of every object and every human to use as queries. We compute the pair of feature vectors, compare against each object-face pair in the list of interactions and sum its weight if both the objects and the faces are close. This estimates the number of interactions between human and object.

#pick/place top human	#pick/place pred. human	Rank pred. human (of 8)	Explanation
12	12	1	✓
8	8	1	✓
7	7	1	✓
6	6	1	✓
6	6	1	✓
6	6	1	✓
4	2	2	(a)
4	2	2	(b)
4	2	2	(c)
0	-	-	✓ (d)

Table 1: Summary of results. The algorithm works especially well for more interactions, where it is less likely that someone else was also present by accident. (a) Predicted one twin, correct answer was the other. (b) Both twins were present in many interactions by coincidence, one of them was ranked first. (d) Failure due to low signal-to-noise ratio. (d) Untouched object successfully gets no appearances or disappearances matched to a human.

Results are shown in table 1. There is one row per object. For each object, there was a true primary human who interacted with it the most. The columns correspond to: the number of times the top human interacted with the object, the number of times the system predicted the top human interacted with the object, the rank of the true top human in the predictions, and explanations. HAC successfully solves all but the extremely noisy cases, despite being a hard dataset and receiving no labels and no specific training.

5 Conclusion

In many datasets we can find entities, subsets of the data with internal consistency, such as people in a video, popular topics from Twitter feeds, or product properties from sentences in its reviews. Currently, most practitioners wanting to find such entities use clustering.

We have demonstrated that the problem of entity finding is well-modeled as an instance of the heavy hitters problem and provided a new algorithm, HAC, for heavy hitters in continuous non-stationary domains. In this approach, entities are specified by indicating how close data points have to be in order to be considered from the same entity and when a subset of points is big enough to be declared an entity. We proved, both theoretically and experimentally, that random sampling (on which HAC is based), works surprisingly well on this problem. Nevertheless, future work on more complex or specialized algorithms could achieve better results.

We used this approach to demonstrate a home-monitoring system that allows a wide variety of post-hoc queries about the interactions among people and objects in the home.

Acknowledgements

We gratefully acknowledge support from NSF grants 1420316, 1523767 and 1723381 and from AFOSR grant FA9550-17-1-0165. F. Alet is supported by a La Caixa fellowship. R. Chitnis is supported by an NSF GRFP fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

We want to thank Marta Alet, Sílvia Asenjo, Carlota Bozal, Eduardo Delgado, Teresa Franco, Lluís Nel-lo, Marc Nel-lo and Laura Pedemonte for their collaboration in the experiments and Maria Bauza for her comments on initial drafts.

References

[Braverman *et al.*, 2017] Vladimir Braverman, Gereon Frahling, Harry Lang, Christian Sohler, and Lin F Yang. Clustering high dimensional dynamic data streams. *arXiv preprint arXiv:1706.03887*, 2017.

[Cao *et al.*, 2006] Feng Cao, Martin Estert, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *SIAM international conference on data mining*, 2006.

[Chaudhuri and Dasgupta, 2010] Kamalika Chaudhuri and Sanjoy Dasgupta. Rates of convergence for the cluster tree. In *NIPS*, 2010.

[Chen and Tu, 2007] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *ACM SIGKDD International Conference On Knowledge Discovery And Data Mining*, pages 133–142, 2007.

[Chen and Zhang, 2016] Di Chen and Qin Zhang. Streaming algorithms for robust distinct elements. In *International Conference on Management of Data*, 2016.

[Donahue *et al.*, 2014] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *ICML*, 2014.

[Ester *et al.*, 1996] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, volume 96, pages 226–231, 1996.

[Johnson *et al.*, 2016] Melvin Johnson, Mike Schuster, Quoc V Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, et al. Google's multilingual neural machine translation system: enabling zero-shot translation. *arXiv preprint arXiv:1611.04558*, 2016.

[King, 2009] Davis E. King. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009.

[Larsen *et al.*, 2016] Kasper Green Larsen, Jelani Nelson, Huy L Nguyên, and Mikkel Thorup. Heavy hitters via cluster-preserving clustering. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 61–70. IEEE, 2016.

[Manku and Motwani, 2002] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 346–357. Elsevier, 2002.

[Mikolov *et al.*, 2013] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[Newman, 2005] Mark EJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[Niebles *et al.*, 2008] Juan Carlos Niebles, Hongcheng Wang, and Li Fei-Fei. Unsupervised learning of human action categories using spatial-temporal words. *IJCV*, 79(3), 2008.

[Nirjon and Stankovic, 2012] Shahriar Nirjon and John A Stankovic. Kinsight: Localizing and tracking household objects using depth-camera sensors. In *Distributed Computing in Sensor Systems (DCOSS), 2012 IEEE 8th International Conference on*, 2012.

[Otto *et al.*, 2017] Charles Otto, Anil Jain, et al. Clustering millions of faces by identity. *IEEE PAMI*, 2017.

[Pedregosa *et al.*, 2011] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Pinheiro *et al.*, 2016] Pedro O Pinheiro, Tsung-Yi Lin, Ronan Collobert, and Piotr Dollár. Learning to refine object segments. In *European Conference on Computer Vision*, 2016.

[Rituerto *et al.*, 2016] Alejandro Rituerto, Henrik Andreasson, Ana C Murillo, Achim Lilienthal, and José Jesús Guerrero. Building an enhanced vocabulary of the robot environment with a ceiling pointing camera. *Sensors*, 16(4), 2016.

[Rodriguez and Laio, 2014] Alex Rodriguez and Alessandro Laio. Clustering by fast search and find of density peaks. *Science*, 344(6191):1492–1496, 2014.

[Schroff *et al.*, 2015] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *CVPR*, 2015.

[Szegedy *et al.*, 2016] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

[Vitter, 1985] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[Wan *et al.*, 2009] Li Wan, Wee Keong Ng, Xuan Hong Dang, Philip S Yu, and Kuan Zhang. Density-based clustering of data streams at multiple resolutions. *ACM Transactions on Knowledge discovery from Data (TKDD)*, 3(3):14, 2009.