Interactive Clustering
Kiran Vodrahalli, COMS 6998-4: 10/16/17

Contents

1 Interaction for Unsupervised Problems

Thus far, we have seen interactive algorithms for supervised learning problems, where each point had a specific label that we could query if we needed to. However, there exist setting where such information is not readily available. It may be that before seeing all the data points, one does not know what the labels may be, and must see several examples to even decide what the labels are. This setup is essentially the problem of clustering, where we get a set of documents for instance and are told to partition the documents into **clusters** which unite documents under underlying themes present in the documents.

Clustering has clear ties to the equivalence query model of learning, in which after presenting a candidate cluster, the algorithm receives a counter-example: A point which is in the symmetric difference of the candidate and the true cluster. However, in the clustering setting, we imagine that it might be difficult to come up with such a counter-example, as it would involve identifying a specific point which might be the problem in the candidate cluster. Thus, we restrict the kind of feedback the user/oracle may provide to only be of yes/no variety. We assume the user will "know a correct cluster when they see it", but are unable to provide more details about what exactly they are looking for. Since the kind of feedback allowed is restricted, the problem becomes more difficult than equivalence queries. In some sense, we are parametrizing the ambiguity of the problem entirely with the user.

In contrast to most prior work, the notion of clustering we take up in this presentation does not assume a particular generative model of the data given cluster labels. Instead, the assumptions are entirely on the user: Namely, that the user has a hard-to-explicitly-specify notion in their head of what a clustering should be, and also that any of the feedback that the user gives is accurate.

2 Interactive Split-Merge Clustering

In their original work, Balcan & Blum (2008) introduced the theoretical study of the interactive clustering problem via split-merge feedback. The central idea is as follows: We can

denote a "clustering" as a set of hypotheses $\{c_1, \cdots, c_n\}$ from hypothesis class $\mathcal{C}$ such that when we apply the set of maps to some dataset $S = \{x_i\}_{i=1}^n$, the hypotheses partition S into disjoint sets. The goal is to identify a set of hypotheses such that we can achieve this goal by making a minimal number of split-merge queries to a user. In particular, one thing which differentiates this setting from other clustering settings is that we are looking for a kind of worst-case guarantee: We wish to make no assumptions about the frequencies of the various types of queries we will receive. This approach is distinct from Bayesian setting where typically strong (and Gaussian) assumptions are made on the data, as is typical in Gaussian mixture modeling. We instead specify two kinds of feedback which the algorithm receives after outputting a candidate clustering (a **query**). A **split** feedback specifies a hypothesis cluster in the algorithm's outputted clustering which needs to be split into one or more clusters, but does not specify how to split the offending cluster. A **merge** feedback specifies two hypothesis clusters in the clustering which should be merged together into a single cluster. Note that by definition, merging is a pure operation in the sense that the two hypotheses selected to merge must be in the same target cluster. Typically, algorithms in this setting involve specifying a way to initialize clusters and how to react when receiving a split request between two hypotheses.

Since it is trivial to cluster in this model using only m queries (start with each point in its own cluster and receive merge requests), we are only interested in algorithms which depend sublinearly on m (hopefully, logarithmically). In general, we hope to find algorithms which cluster with a number of queries of order $\mathcal{O}(\text{poly}(k, \log|\mathcal{C}|, \log m))$. Using this model, it is possible to design clustering algorithms for specific hypothesis classes (e.g., intervals, disjunctions) which take advantage of the specific structure of the hypothesis class to get good algorithms.

2.1 Example 1: Learning Interval Clusters

In this setting, let the hypothesis class $\mathcal{C}$ be intervals on the line of the form $[p_1, p_2]$.

Theorem 2.1. *Clustering Intervals.*
We only require $\mathcal{O}(k \log m)$ split-merge queries to cluster using intervals on the line, where k is the number of clusters, and m is the number of points in the dataset.

Proof. First, we specify how to initialize the set of clusters. We simply begin with a single cluster containing all m points of the dataset. Next, we specify how to split clusters: Upon a split request to cluster $c := [p_1, p_2]$, we partition c into two clusters of equal cardinality by defining $\lfloor \frac{p_1+p_2}{2} \rfloor$ as the new boundary.

Now, we see how this algorithm results in the desired query complexity. Consider that the true clustering consists of k disjoint intervals which partition the line. Then, let the boundaries of the clustering be the $k-1$ points $a_1, \cdots, a_{k-1}$, where these values do NOT coincide with actual point values. Instead, a_i is an arbitrary point between the largest point in interval i and the smallest point in interval $i+1$. Let $\textbf{size}(a_i)$ denote the number of points in the cluster a_i is contained in, and be 0 if it is not contained in any clusters. At the start, $\textbf{size}(a_i) = m$ for all i.

We first bound the number of split queries. Note that split queries are only ever given to clusters which contain some a_i. Thus, since in the true clustering we must have $\textbf{size}(a_i) \geq 0$, and there always exists at least one a_i for which a split feedback reduces $\textbf{size}(a_i)$ by half, we know that each $\textbf{size}(a_i)$ experiences at most $\log m$ splits. Since there are $\mathcal{O}(k)$ a_i, there are at most $\mathcal{O}(k \log m)$ splits in total.

To bound the number of merges, note that merges only happen between clusters that do not contain any a_i. Since the number of splits bounds the total possible number of individual clusters which do not contain any a_i, we can only select merge operations from this list of $\mathcal{O}(k \log m)$ clusters. Thus, there are $\mathcal{O}(k \log m)$ possible merges that can be requested (each merge reduces the number of clusters by 1), and the total query complexity is $\mathcal{O}(k \log m)$. $\square$

2.2 Example 2: Learning Rectangle Clusters

We can generalize the result from the previous example to higher dimensions of intervals. In general, a d-dimensional interval is a concept defined by the product of 1-dimensional intervals $[a_1^{(1)}, a_2^{(1)}] \times \cdots \times [a_1^{(d)}, a_2^{(d)}]$. In general, it takes $\mathcal{O}((kd \log m)^d)$ queries to cluster d-dimensional intervals, as shown in Awasthi & Zadeh (2010). We will show the result for 2-dimensional rectangles.

Theorem 2.2. *Clustering 2-dimensional Rectangles.*
We can cluster the class of 2-dimensional rectangles using $\mathcal{O}((k \log m)^2)$ queries.

Proof. We use the following algorithm due to Awasthi & Zadeh (2010). We maintain a graph G over the m points in the dataset, initially with no edges. We also maintain a list of x-borders $[x_1, \cdots, x_T]$ and y-borders $[y_1, \cdots, y_T]$, where T is the number of split requests. We will essentially proceed in a manner analogous to the interval case, except on both the x and y dimensions. We will thereby maintain a 2-dimensional grid of rectangular clusters, starting with an initial rectangle which contains the whole space. The algorithm is given by:

(a) Start with points (a_1', a_2') and (b_1', b_2') such that all points in the dataset are contained in the rectangle defined by these two points as its corners. Originally, the x-borders are just $[a_1', b_1']$ and the y-borders are just $[a_2', b_2']$. The graph G initially has no vertices.

(b) Define the candidate clusters according to the rectangular grid (at the first step, only one rectangle exists and all m points are in the same cluster). If the points in two candidate clusters form a clique in G, merge the regions. Do this for all possible merge-able regions, and present the final clustering to the user.

(c) If the user replies with merge(c_1, c_2) feedback, create a clique in G according to the points in the two clusters c_1, c_2.

(d) If the user replies with split(c) feedback, do the following. For reference, c refers to a rectangle which is completely defined by its two corner points $(x_1, y_1), (x_2, y_2)$. Introduce a point x_{new} to the x-borders such that $x_1 < x_{new} < x_2$ and such that if we

project all points in c onto the x-axis, x_{new} divides them into two intervals of equal size. Define y_{new} analogously and add it to y-borders.

We proceed to analyze this algorithm in a similar manner to the 1-dimensional interval case. First, we bound the number of split queries. Note that every time we get a split request, we split in both the x and y axes, even though in reality, we may have only needed to split in one axis. That means that there may be at most twice the number of necessary intervals on each axis, in other words, $2k$, since there are at most k necessary intervals on a single axis (suppose all the points were on the x-axis). Therefore, since the x-axis and y-axis each have at most $2k$ intervals, there are at most $2k \log m$ split requests along the x-axis and $2k \log m$ split requests along the y-axis for a total of $4k \log m$ split requests, which is justified by the arguments for the 1-dimensional interval case. Now note that we have at most $(2k \log m)^2$ total regions. Since each merge request reduces the total number of regions by 1, the number of merge requests is upper bounded by the number of possible regions. Thus, we have $\mathcal{O}(4k \log m + (2k \log m)^2)$ total queries, which is $\mathcal{O}((k \log m)^2)$.

□

2.3 Example 3: Learning DNF and CNFs

We now look to a different kind of concept, DNF formulas.

Definition 2.3. DNF Formula.
A DNF formula is a Boolean formula composed entirely of disjunctions (OR operators). We begin with some set of variables $z_1, \cdots, z_n$ which can take on values 0 or 1. The OR of some subset of these variables constitutes a DNF formula.

We can use DNF formulas to cluster points $x \in S$, where each $x \in S$ lives in $\{0,1\}^n$. We can vizualize x as a string of 0s and 1s, where the i^{th} position corresponds to the value of variable z_i. The clusters are then DNF formulas: A cluster contains all the points $x \in S$ such that the associated DNF formula evaluates to true on them. Say for now that $n = 6$. Then, for instance, an example DNF formula would be (z_1 OR z_5 OR z_6). In this case, the point 011100 would not be in the cluster associated to that formula. However, each of the points $100000, 000010, 000001$ would be, in addition to several others like 110010.

It is possible to cluster with efficient query complexity using the following simple algorithm. First, begin with a cluster for each variable z_i such that it contains all points $x \in S$ such that $x_i = 1$. Respond to split requests by deleting the cluster entirely, and to merge requests by taking the OR of the two DNF formula to construct a new cluster. However, note that this approach may result in intermediate hypothesis clusters which are not disjoint (this can be seen even in the case of the initial clusters, the point 100100 is in both the initial cluster for z_1 and the initial cluster for z_4.)

Theorem 2.4. *Non-Disjoint Query Complexity of DNF Formulas.*
The algorithm described above requires at most $n - k$ requests to cluster disjunctions over $\{0,1\}^n$ when allowed to have hypothesis clusters which are not disjoint as intermediate steps.

Proof. First, note that when clusters are merged, they are never split afterwards because the merge operation contains points only from a single cluster. Thus, split requests can only be made to clusters associated with single variables. Recall that splitting a cluster means you simply delete it. Then, consider that if z_i is a relevant variable (in that one of the valid cluster DNF formulas contains it), it will never be split. Essentially, if we actually need z_i, it can only be merged. Neither split or merge will ever create a situation where there exists a point $x \in S$ such that no cluster evaluates it to true: Merging two clusters just makes it easier for a point x to belong to the merged cluster, and we only split singleton clusters which are irrelevant. In other words, we can either throw out a variable which we don't need, or add a variable to a set if we don't have the clustering we want. Each merge or split reduces the number of clusters by 1. Since there are k clusters in truth, and n clusters at the beginning, there are at most $n - k$ split merge operations before reaching the goal. □

It is possible to fix the disjointness issue at the cost of increased query complexity.

Theorem 2.5. *Disjoint Query Complexity of DNF Formulas.*
A modified algorithm exists which requires at most $\mathcal{O}(n^2)$ queries to cluster disjunctions over $\{0,1\}^n$ while only maintaining hypothesis clusters which are completely disjoint.

Proof. We fix the disjointness issue by using as our initial clusters a single cluster for each variable z_i, such that it contains all points $x \in S$ such that $x_i = 1$ and $x_j = 0$ for all $j < i$. This ensures that there will be no overlap between the cluster z_i and z_j for $j < i$, since z_j will have x such that $x_j = 1$, while the cluster z_i necessarily has x such that $x_i = 1$ and $x_j = 0$. We modify the split request step to cluster z_i by deleting the variable x_i from all points in S, essentially changing the problem from a problem over $\{0,1\}^n$ to a problem over $\{0,1\}^{n-1}$. Then, instead of just continuing on, we treat it as a new problem to solve entirely. Merging stays the same.

These changes allow us to avoid ever using hypotheses which overlap, since (a) merging never causes overlapping hypotheses, (b) the initial hypotheses are disjoint, and (c) splitting never causes overlapping hypotheses since every split means a re-initialization of the hypotheses.

After ℓ splits, there are at most $n - \ell - k$ merges which can be performed before we hit k clusters. Thus, there are a total of $\sum_{\ell=0}^{n-k}(n - \ell - k) = \mathcal{O}((n-k)^2)$ possible merges and $n - k$ splits, which is $\mathcal{O}(n^2)$ total operations. □

2.4 Generalization to Conjunctions

Thus far we have only talked about DNFs. The algorithms described above readily generalize to CNFs, which are formulas of conjunctions (assume they have no negations), in the following way. Since each true cluster is disjoint from the other clusters, we can write $c_i = \text{NOT}(c_1)$ AND $\cdots$ AND $\text{NOT}(c_{i-1})$ AND $\text{NOT}(c_{i+1})$ AND $\cdots$ AND $\text{NOT}(c_k)$. Since each c is a conjunction, and $\text{NOT}(c)$ is a disjunction of negated variables, we can write c_i as a $(k-1)$-conjunction of (disjunctions of negated variables). There are at most n negated variables. The formula is equivalent to an OR over all possible acceptance states. The formula

can be satisfied if there is at least one negated variable in each disjunction which evaluates to 1. Thus, we define new variables $y_{i_1 \cdots i_{k-1}} = \bar{x}_{i_1} \cdots \bar{x}_{i_{k-1}}$, and re-write the expression for c_i as an OR over all these n^{k-1} variables. Here, $i_1, \cdots, i_{k-1}$ each loop over $[n]$.

Thus, we have a transformation from CNFs to DNFs which have n^{k-1} variables, and we can use our algorithms for disjunctions to solve conjunctions with $\mathcal{O}(n^{k-1})$ queries and $\mathcal{O}(n^{2(k-1)})$ queries respectively.

3 Two Generic Inefficient Interactive Clustering Algorithms

However, we would like to design more general algorithms for interactive clustering oblivious to the specific concept class we are dealing with. One approach is to work explicitly over the version space of clusterings, and with each feedback response to a query, reduce the size of the version space by some fraction. In particular, this approach is intimately related to the halving algorithm and is common in interactive learning theory (for instance, the splitting index approach by Dasgupta (2005) is similar in general spirit). In particular, if we can guarantee that we can reduce the version space by a fixed fraction each iteration, *no matter what the result of the query was*, we will be able to obtain a query-efficient algorithm, providing an upper bound to the query complexity of interactive clustering for any hypothesis class for clusters. The first algorithm following this approach was given in Balcan & Blum (2008). Before giving the algorithm, we define an important notion:

Definition 3.1. α-consistent.
A set S of points is α-consistent for some $\alpha \in (0, 1)$ with respect to a concept class $\mathcal{C}$ and a dataset of points P if for an α-fraction of all clusterings of concepts $(c_1, \cdots, c_k) \in \mathcal{C}^{\text{VS}}$ in the version space, it is true that $S \subseteq c_i(P)$ for some $i \in [k]$.

This notion is critical to defining to ensuring we make progress as use feedback to prune the version space. Algorithm 1 implements this strategy and yields a query complexity $\mathcal{O}(k^3 \log |\mathcal{C}|)$, where k is the number of clusters. We can see this fact since to reduce to the case where $|V| = 1$, we need T iterations of the while loop where we are only guaranteed to remove a $\frac{1}{k^2}$ fraction of the version space each round. Solving $\left(1 - \frac{1}{k^2}\right)^T |\mathcal{C}^{VS}| = 1$ yields $T = \frac{k \log |\mathcal{C}|}{\log \frac{k^2}{k^2-1}}$, noting that $|\mathcal{C}^{VS}| \leq |\mathcal{C}|^{k-1}$. Seeing that $\log^{-1} \frac{k^2}{k^2-1} = \mathcal{O}(k^2)$, we see the query complexity is $\mathcal{O}(k^3 \log |\mathcal{C}|)$.

However, we can improve this bound to $\mathcal{O}(k \log |\mathcal{C}|)$ using a modified, simpler algorithm due to Awasthi & Zadeh (2010), given in Algorithm 2. To succinctly describe the algorithm, we need the following definition:

Definition 3.2. Consistent Cluster Set.
For some set of points $s \subset S$, the **consistent cluster set**

$$\text{CCS}(s) := \left\{ \{c_1, \cdots, c_k\} \in \mathcal{C}^{VS} | \{c_1, \cdots, c_k\} \text{ is consistent with } s \right\}$$

Algorithm 1 Generic Clustering (Inefficient)

procedure CLUSTER(S) ▷ S := input dataset of m points
  version space $V := \mathcal{C}^{VS}$ ▷ $\mathcal{C}^{VS}$:= the set of all k-clusterings on S
  while $|V| > 1$ **do**
    Initialize buckets $B_1, \cdots, B_k := \{\}$.
    Initialize output cluster list $L = []$.
    for each point $x \in S$ in arbitrary order **do**
      for i in $[1, \cdots, k]$ **do**
        if $B_i \cup \{x\}$ is $\frac{1}{k^2}$-consistent **then** $B_i := B_i \cup \{x\}$
          if B_i is $\left(1 - \frac{1}{k^2}\right)$-consistent **then**
            Append B_i to L.
            Delete B_i from the list of buckets.
            Append $\{\}$ to the end of the list of buckets.
          end if
          break
        end if
      end for
    end for
    Output cluster list L.
    Receive feedback F from user.
    if $F = \text{merge}(c_i, c_j)$ **then**
      Remove from V all clusterings inconsistent with $c_i \cup c_j$.
    else if $F = \text{split}(c_i)$ **then**
      Remove from V all clusterings consistent with c_i.
    end if
  end while
  return V as the clustering.
end procedure

Algorithm 2 Generic Clustering (Also Inefficient)

procedure CLUSTER(S) ▷ S := input dataset of m points
  version space $V := \mathcal{C}^{VS}$ ▷ $\mathcal{C}^{VS}$:= the set of all k-clusterings on S
  while $|V| > 1$ **do**
    Initialize output cluster list $L = []$.
    Initialize $i = 1$.
    while clusters in L do not cover S **do**
      $c_i = \underset{\substack{s \subset S \setminus L \\ |\text{CCS}(s)| \geq \frac{1}{2}|V|}}{\text{argmax}} |s|$.
      Append c_i to L.
      Update $i = i + 1$.
    end while
    Output cluster list L.
    Receive feedback F from user.
    if $F = \text{merge}(c_i, c_j)$ **then**
      Remove from V all clusterings inconsistent with $c_i \cup c_j$.
    else if $F = \text{split}(c_i)$ **then**
      Remove from V all clusterings consistent with c_i.
    end if
  end while
  return V as the clustering.
end procedure

In the case of Algorithm 2, we are guaranteed to halve the version space each time no matter what the feedback is. Thus, we effectively are performing binary search over the version space, and the query complexity is $\mathcal{O}(\log |\mathcal{C}^{VS}|) = \mathcal{O}(k \log |\mathcal{C}|)$.

Why does Algorithm 2 work? Let us be more specific.

Theorem 3.3. *Generic Algorithm 2 can cluster finite concept classes with $\mathcal{O}(k \log |C|)$ queries.*

Proof. The key trick is to find a **largest** set of points c_i such that the number of clusterings consistent with it is more than half the size of the version space. That way, if we call split on c_i, we remove more than half the version space. If we call merge on c_i with some c', we know that $c_i \cup c'$ was not consistent with more than half the version space (otherwise, we could have added c' to c_i and made a larger set of points that way). Thus, the set of points inconsistent with $c_i \cup c'$ is also larger than half the size of the version space, and we again get to remove half the version space. Thus, no matter what, we can remove half the version space and make quick progress through the space via halving algorithm, yielding a $\mathcal{O}(\log |\mathcal{C}^{VS}|) = \mathcal{O}(k \log |\mathcal{C}|)$ query complexity, since $|\mathcal{C}^{VS}| \leq |\mathcal{C}|^{k-1}$.

One last point: Does a set exist which is consistent with more than half the clusterings in the version space? We merely ask for the largest such set. For instance, we could always find a set of size 1 which is consistent with more than half the clusterings. □

Remark 3.4. Using VC dimension instead.
We can replace the results of the above theorem with query complexity $\mathcal{O}(kd \log m)$, where d is the VC dimension of the hypothesis class since Sauer-Shelah gives that the number of different ways to split m points with members of the hypothesis class is less than m^d. Thus, the size of the version space is bounded by $(m^d)^{k-1} \leq m^{kd}$ and taking log gives the result.

References

Pranjal Awasthi and Reza Bosagh Zadeh. Supervised clustering. *Advances in Neural Information Processing Systems*, 2010.

Maria Florina Balcan and Avrim Blum. Clustering with interactive feedback. *ALT*, 2008.

Sanjoy Dasgupta. Coarse sample complexity bounds for active learning. *Advances in Neural Information Processing Systems*, pp. 235–242, 2005.