Chapter 8

Approximation algorithms III

By Sariel Har-Peled, November 1, 2015[①] Version: 0.21

8.1. Clustering

Consider the problem of ***unsupervised learning***. We are given a set of examples, and we would like to partition them into classes of similar examples. For example, given a webpage X about "The reality dysfunction", one would like to find all webpages on this topic (or closely related topics). Similarly, a webpage about "All quiet on the western front" should be in the same group as webpage as "Storm of steel" (since both are about soldier experiences in World War I).

The hope is that all such webpages of interest would be in the same cluster as X, if the clustering is good.

More formally, the input is a set of examples, usually interpreted as points in high dimensions. For example, given a webpage W, we represent it as a point in high dimensions, by setting the ith coordinate to 1 if the word w_i appears somewhere in the document, where we have a prespecified list of $10,000$ words that we care about. Thus, the webpage W can be interpreted as a point of the $\{0,1\}^{10,000}$ hypercube; namely, a point in $10,000$ dimensions.

Let X be the resulting set of n points in d dimensions.

To be able to partition points into similar clusters, we need to define a notion of similarity. Such a similarity measure can be any distance function between points. For example, consider the "regular" Euclidean distance between points, where

$$\|p-q\| = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2},$$

where $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$.

As another motivating example, consider the ***facility location problem***. We are given a set X of n cities and distances between them, and we would like to build k hospitals, so that the maximum distance of a city from its closest hospital is minimized. (So that the maximum time it would take a patient to get to the its closest hospital is bounded.)

Intuitively, what we are interested in is selecting good representatives for the input point-set X. Namely, we would like to find k points in X such that they represent X "well".

Formally, consider a subset S of k points of X, and a p a point of X. The ***distance of p from the set S*** is

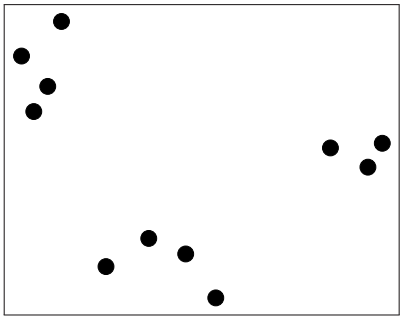
$$\mathbf{d}(p, S) = \min_{q \in S} \|p - q\|;$$

namely, $\mathbf{d}(p, S)$ is the minimum distance of a point of S to p. If we interpret S as a set of centers then $\mathbf{d}(p, S)$ is the distance of p to its closest center.

Now, the ***price of clustering*** X by the set S is

$$\nu(X, S) = \max_{p \in X} \mathbf{d}(p, S).,$$

This is the maximum distance of a point of X from its closest center in S.



It is somewhat illuminating to consider the problem in the plane. We have a set P of n points in the plane, we would like to find k smallest discs centered at input points, such that they cover all the points of P. Consider the example on the right.

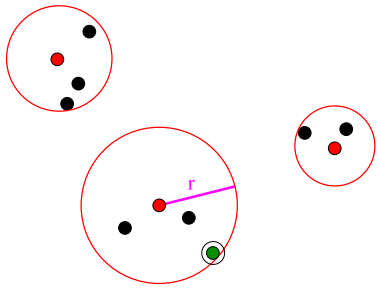


Figure 8.1: The marked point is the bottleneck point.

In this example, assume that we would like to cover it by 3 disks. One possible solution is being shown in Figure 8.1. The quality of the solution is the radius r of the largest disk. As such, the clustering problem here can be interpreted as the problem of computing an optimal cover of the input point set by k discs/balls of minimum radius. This is known as the ***k-center*** problem.

It is known that k-center clustering is NP-Hard, even to approximate within a factor of (roughly) 1.8. Interestingly, there is a simple and elegant 2-approximation algorithm. Namely, one can compute in polynomial time, k centers, such that they induce balls of radius at most twice the optimal radius.

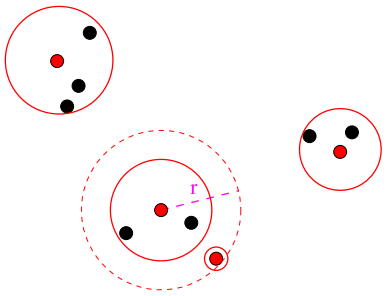
Here is the formal definition of the k-center clustering problem.

k-center clustering

Instance: A set P a of n points, a distance function $\mathbf{d}(p, q)$, for $p, q \in P$, with triangle inequality holding for $\mathbf{d}(\cdot, \cdot)$, and a parameter k.
Question: A subset S that realizes $r_{opt}(P, k) = \min_{S \subseteq P, |S|=k} \mathsf{D}_S(P)$, where $\mathsf{D}_S(P) = \max_{x \in X} \mathbf{d}(x, S)$ and $\mathbf{d}(x, S) = \min_{s \in S} \mathbf{d}(s, x)$.

8.1.1. The approximation algorithm for k-center clustering

To come up with the idea behind the algorithm, imagine that we already have a solution with $m = 3$ centers. We would like to pick the next $m + 1$ center. Inspecting the examples above, one realizes that the solution is being determined by a bottleneck point; see Figure 8.1. That is, there is a single point which determine the quality of the clustering, which is the point furthest away from the set of centers. As such, the natural step is to find a new center that would better serve this bottleneck point. And, what can be a better service for this point, than make it the next center? (The resulting clustering using the new center for the example is depicted on the right.)



Namely, we always pick the bottleneck point, which is furthest away for the current set of centers, as the next center to be added to the solution.

The resulting approximation algorithm is depicted on the right. Observe, that the quantity r_{i+1} denotes the (minimum) radius of the i balls centered at $u_1, \ldots, u_i$ such that they cover P (where all these balls have the same radius). (Namely, there is a point $p \in P$ such that $\mathbf{d}(p, \{u_1, \ldots, u_i\}) = r_{i+1}$.

It would be convenient, for the sake analysis, to imagine that we run **AprxKCenter** one additional iteration, so that the quantity r_{k+1} is well defined.

Observe, that the running time of the algorithm **AprxKCenter** is $O(nk)$ as can be easily verified.

```
AprxKCenter(P, k)
  P = {p_1, ..., p_n}
  S = {p_1}, u_1 ← p_1
  while |S| < k do
    i ← |S|
    for j = 1 ... n do
      d_j ← min(d_j, d(p_j, u_i))
    r_{i+1} ← max(d_1, ..., d_n)
    u_{i+1} ←point of P realizing r_i
    S ← S ∪ {u_{i+1}}

  return S
```

Lemma 8.1.1. *We have that $r_2 \geq \ldots \geq r_k \geq r_{k+1}$.*

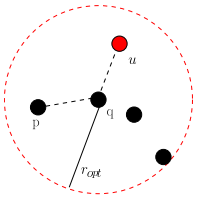
Proof: At each iteration the algorithm adds one new center, and as such the distance of a point to the closest center can not increase. In particular, the distance of the furthest point to the centers does not increase. ■

Observation 8.1.2. *The radius of the clustering generated by* **AprxKCenter** *is r_{k+1}.*

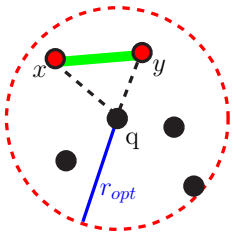
Lemma 8.1.3. *We have that $r_{k+1} \leq 2r_{opt}(P,k)$, where $r_{opt}(P,k)$ is the radius of the optimal solution using k balls.*

Proof: Consider the k balls forming the optimal solution: $D_1, \ldots, D_k$ and consider the k center points contained in the solution S computed by **AprxKCenter**.

If every disk D_i contain at least one point of S, then we are done, since every point of P is in distance at most $2r_{opt}(P,k)$ from one of the points of S. Indeed, if the ball D_i, centered at q, contains the point $u \in S$, then for any point $p \in P \cap D_i$, we have that



$$\mathbf{d}(p,u) \leq \mathbf{d}(p,q) + \mathbf{d}(q,u) \leq 2r_{opt}.$$



Otherwise, there must be two points x and y of S contained in the same ball D_i of the optimal solution. Let D_i be centered at a point q.

We claim distance between x and y is at least r_{k+1}. Indeed, imagine that x was added at the αth iteration (that is, $u_\alpha = x$), and y was added in a later βth iteration (that is, $u_\beta = y$), where $\alpha < \beta$. Then,

$$r_\beta = \mathbf{d}\Big(y, \big\{u_1, \ldots, u_{\beta-1}\big\}\Big) \leq \mathbf{d}(x,y),$$

since $x = u_\alpha$ and $y = u_\beta$. But $r_\beta \geq r_{k+1}$, by Lemma 8.1.1. Applying the triangle inequality again, we have that $r_{k+1} \leq r_\beta \leq \mathbf{d}(x,y) \leq \mathbf{d}(x,q) + \mathbf{d}(q,y) \leq 2r_{opt}$, implying the claim. ■

Theorem 8.1.4. *One can approximate the k-center clustering up to a factor of two, in time $O(nk)$.*

Proof: The approximation algorithm is **AprxKCenter**. The approximation quality guarantee follows from Lemma 8.1.3, since the furthest point of P from the k-centers computed is r_{k+1}, which is guaranteed to be at most $2r_{opt}$. ■

8.2. Subset Sum

Subset Sum

Instance: $X = \{x_1, \ldots, x_n\}$ – n integer positive numbers, t - target number
Question: Is there a subset of X such the sum of its elements is t?

Subset Sum is (of course) NPC, as we already proved. It can be solved in polynomial time if the numbers of X are small. In particular, if $x_i \leq M$, for $i = 1, \ldots, n$, then $t \leq Mn$ (otherwise, there is no solution). Its reasonably easy to solve in this case, as the algorithm on the right shows. The running time of the resulting algorithm is $O(Mn^2)$.

```
SolveSubsetSum (X, t, M)
  b[0 ... Mn] - boolean array init to FALSE.
    // b[x] is TRUE if x can be realized by
          a subset of X.
  b[0] ← TRUE.
  for i = 1, ..., n do
    for j = Mn down to x_i do
      b[j] ← B[j − x_i] ∨ B[j]

  return B[t]
```

Note, that M might be prohibitly large, and as such, this algorithm is not polynomial in n. In particular, if $M = 2^n$ then this algorithm is prohibitly slow. Since the relevant decision problem is NPC, it is unlikely that an efficient algorithm exist for this problem. But still, we would like to be able to solve it quickly and efficiently. So, if we want an efficient solution, we would have to change the problem slightly. As a first step, lets turn it into an optimization problem.

Subset Sum Optimization

Instance: (X, t): A set X of n positive integers, and a target number t.
Question: The largest number γ_{opt} one can represent as a subset sum of X which is smaller or equal to t.

Intuitively, we would like to find a subset of X such that it sum is smaller than t but very close to t. Next, we turn problem into an approximation problem.

Subset Sum Approx

Instance: (X, t, ε): A set X of n positive integers, a target number t, and parameter $\varepsilon > 0$.
Question: A number z that one can represent as a subset sum of X, such that $(1 - \varepsilon)\gamma_{\text{opt}} \leq z \leq \gamma_{\text{opt}} \leq t$.

The challenge is to solve this approximation problem efficiently. To demonstrate that there is hope that can be done, consider the following simple approximation algorithm, that achieves a constant factor approximation.

Lemma 8.2.1. *Let (X, t) be an instance of Subset Sum. Let γ_{opt} be optimal solution to given instance. Then one can compute a subset sum that adds up to at least $\gamma_{\text{opt}}/2$ in $O(n \log n)$ time.*

Proof: Add the numbers from largest to smallest, whenever adding a number will make the sum exceed t, we throw it away. We claim that the generated sum s has the property that $\gamma_{\text{opt}}/2 \leq s \leq t$. Clearly, if the total sum of the numbers is smaller than t, then no number is being rejected and $s = \gamma_{\text{opt}}$.

Otherwise, let u be the first number being rejected, and let s' be the partial subset sum, just before u is being rejected. Clearly, $s' > u > 0$, $s' < t$, and $s' + u > t$. This implies $t < s' + u < s' + s' = 2s'$, which implies that $s' \geq t/2$. Namely, the subset sum output is larger than $t/2$. ∎

8.2.1. On the complexity of ε-approximation algorithms

Definition 8.2.2 (PTAS). For a maximization problem PROB, an algorithm $\mathcal{A}(I, \varepsilon)$ (i.e., A receives as input an instance of PROB, and an approximation parameter $\varepsilon > 0$) is a ***polynomial time approximation scheme*** (***PTAS***) if for any instance I we have

$$(1 - \varepsilon)\left|\mathsf{opt}(I)\right| \leq \left|\mathcal{A}(I, \varepsilon)\right| \leq \left|\mathsf{opt}(I)\right|,$$

where $|\mathsf{opt}(I)|$ denote the price of the optimal solution for I, and $|\mathcal{A}(I, \varepsilon)|$ denotes the price of the solution outputted by $\mathcal{A}$. Furthermore, the running time of the algorithm $\mathcal{A}$ is polynomial in n (the input size), when ε is fixed.

For a minimization problem, the condition is that $|\mathsf{opt}(I)| \leq |\mathcal{A}(I, \varepsilon)| \leq (1 + \varepsilon)|\mathsf{opt}(I)|$.

Example 8.2.3. An approximation algorithm with running time $O(n^{1/\varepsilon})$ is a PTAS, while an algorithm with running time $O(1/\varepsilon^n)$ is not.

Definition 8.2.4 (FPTAS.). An approximation algorithm is ***fully polynomial time approximation scheme*** (***FPTAS***) if it is a PTAS, and its running time is polynomial both in n and $1/\varepsilon$.

Example 8.2.5. A PTAS with running time $O(n^{1/\varepsilon})$ is not a FPTAS, while a PTAS with running time $O(n^2/\varepsilon^3)$ is a FPTAS.

8.2.2. Approximating subset-sum

Let $S = \{a_1, \ldots, a_n\}$ be a set of numbers. For a number x, let $x + S$ denote the translation of S by x; namely, $x + S = \{a_1 + x, a_2 + x, \ldots a_n + x\}$. Our first step in deriving an approximation algorithm for Subset Sum is to come up with a slightly different algorithm for solving the problem exactly. The algorithm is depicted on the right.

Note, that while **ExactSubsetSum** performs only n iterations, the lists P_i that it constructs might have exponential size.

ExactSubsetSum(S, t)
  $n \leftarrow |S|$
  $P_0 \leftarrow \{0\}$
  for $i = 1 \ldots n$ **do**
    $P_i \leftarrow P_{i-1} \cup (P_{i-1} + x_i)$
    Remove from P_i all elements $> t$
  return largest element in P_n

Trim(L', δ)
  // L': inc. sorted list of #s
  $L = \langle y_1 \ldots y_m \rangle$
    // $y_i \leq y_{i+1}$, for $i = 1, \ldots, n-1$.
  $curr \leftarrow y_1$
  $L_{out} \leftarrow \{y_1\}$
  for $i = 2 \ldots m$ **do**
    if $y_i > curr \cdot (1 + \delta)$
      Append y_i to L_{out}
      $curr \leftarrow y_i$
  return L_{out}

Thus, if we would like to turn **ExactSubsetSum** into a faster algorithm, we need to somehow to make the lists L_l smaller. This would be done by removing numbers which are very close together.

Definition 8.2.6. For two positive real numbers $z \leq y$, the number y is a δ-approximation to z if $\dfrac{y}{1+\delta} \leq z \leq y$.

The procedure **Trim** that trims a list L' so that it removes close numbers is depicted on the left.

Observation 8.2.7. *If $x \in L'$ then there exists a number $y \in L_{out}$ such that $y \leq x \leq y(1 + \delta)$, where $L_{out} \leftarrow$ **Trim**(L', δ).*

We can now modify **ExactSubsetSum** to use **Trim** to keep the candidate list shorter. The resulting algorithm **ApproxSubsetSum** is depicted on the right. Note, that computing E_i requires merging two sorted lists, which can be done in linear time in the size of the lists (i.e., we can keep all the lists sorted, without sorting the lists repeatedly).

Let E_i be the list generated by the algorithm in the ith iteration, and P_i be the list of numbers without any trimming (i.e., the set generated by **ExactSubsetSum** algorithm) in the ith iteration.

ApproxSubsetSum(S, t)
  //Assume $S = \{x_1, \ldots, x_n\}$, where
  // $x_1 \le x_2 \le \ldots \le x_n$
  $n \leftarrow |S|$, $L_0 \leftarrow \{0\}$, $\delta = \varepsilon/2n$
  for $i = 1 \ldots n$ **do**
    $E_i \leftarrow L_{i-1} \cup (L_{i-1} + x_i)$
    $L_i \leftarrow$ **Trim**(E_i, δ)
    Remove from L_i all elements $> t$.

  return largest element in L_n

Claim 8.2.8. *For any $x \in P_i$ there exists $y \in L_i$ such that $y \le x \le (1+\delta)^i y$.*

Proof: If $x \in P_1$ the claim follows by Observation 8.2.7 above. Otherwise, if $x \in P_{i-1}$, then, by induction, there is $y' \in L_{i-1}$ such that $y' \le x \le (1+\delta)^{i-1} y'$. Observation 8.2.7 implies that there exists $y \in L_i$ such that $y \le y' \le (1+\delta)y$, As such,

$$y \le y' \le x \le (1+\delta)^{i-1} y' \le (1+\delta)^i y$$

as required.

The other possibility is that $x \in P_i \setminus P_{i-1}$. But then $x = \alpha + x_i$, for some $\alpha \in P_{i-1}$. By induction, there exists $\alpha' \in L_{i-1}$ such that

$$\alpha' \le \alpha \le (1+\delta)^{i-1} \alpha'.$$

Thus, $\alpha' + x_i \in E_i$ and by Observation 8.2.7, there is a $x' \in L_i$ such that

$$x' \le \alpha' + x_i \le (1+\delta) x'.$$

Thus,

$$x' \le \alpha' + x_i \le \alpha + x_i = x \le (1+\delta)^{i-1} \alpha' + x_i \le (1+\delta)^{i-1} (\alpha' + x_i) \le (1+\delta)^i x'.$$

Namely, for any $x \in P_i \setminus P_{i-1}$, there exists $x' \in L_i$, such that $x' \le x \le (1+\delta)^i x'$. ■

8.2.2.1. Bounding the running time of ApproxSubsetSum

We need the following two easy technical lemmas. We include their proofs here only for the sake of completeness.

Lemma 8.2.9. *For $x \in [0,1]$, it holds $\exp(x/2) \le (1+x)$.*

Proof: Let $f(x) = \exp(x/2)$ and $g(x) = 1 + x$. We have $f'(x) = \exp(x/2)/2$ and $g'(x) = 1$. As such,

$$f'(x) = \frac{\exp(x/2)}{2} \le \frac{\exp(1/2)}{2} \le 1 = g'(x), \qquad \text{for } x \in [0,1].$$

Now, $f(0) = g(0) = 1$, which immediately implies the claim. ■

Lemma 8.2.10. *For $0 < \delta < 1$, and $x \geq 1$, we have $\log_{1+\delta} x \leq \dfrac{2 \ln x}{\delta} = O\left(\dfrac{\ln x}{\delta}\right)$.*

Proof: We have, by Lemma 8.2.9, that $\log_{1+\delta} x = \dfrac{\ln x}{\ln(1+\delta)} \leq \dfrac{\ln x}{\ln \exp(\delta/2)} = \dfrac{2 \ln x}{\delta}$. ∎

Observation 8.2.11. *In a list generated by* **Trim**, *for any number x, there are no two numbers in the trimmed list between x and $(1+\delta)x$.*

Lemma 8.2.12. *We have $|L_i| = O\left(\dfrac{n^2}{\varepsilon} \log n\right)$, for $i = 1, \ldots, n$.*

Proof: The set $L_{i-1} + x_i$ is a set of numbers between x_i and ix_i, because x_i is larger than $x_1 \ldots x_{i-1}$ and L_{i-1} contains subset sums of at most $i-1$ numbers, each one of them smaller than x_i. As such, the number of different values in this range, stored in the list L_i, after trimming is at most

$$\log_{1+\delta} \frac{ix_i}{x_i} = O\left(\frac{\ln i}{\delta}\right) = O\left(\frac{\ln n}{\delta}\right),$$

by Lemma 8.2.10. Thus, as $\delta = \varepsilon/2n$, we have

$$|L_i| \leq |L_{i-1}| + O\left(\frac{\ln n}{\delta}\right) \leq |L_{i-1}| + O\left(\frac{n \ln n}{\varepsilon}\right) = O\left(\frac{n^2 \log n}{\varepsilon}\right).$$ ∎

Lemma 8.2.13. *The running time of* **ApproxSubsetSum** *is $O\left(\frac{n^3}{\varepsilon} \log^2 n\right)$.*

Proof: Clearly, the running time of **ApproxSubsetSum** is dominated by the total length of the lists $L_1, \ldots, L_n$ it creates. Lemma 8.2.12 implies that $\sum_i |L_i| = O\left(\dfrac{n^3}{\varepsilon} \log n\right)$. The running time of **Trim** is proportional to the size of the lists, implying the claimed running time. ∎

8.2.2.2. The result

Theorem 8.2.14. **ApproxSubsetSum** *returns a number $u \leq t$, such that*

$$\frac{\gamma_{\text{opt}}}{1+\varepsilon} \leq u \leq \gamma_{\text{opt}} \leq t,$$

where γ_{opt} is the optimal solution (i.e., largest realizable subset sum smaller than t).

The running time of **ApproxSubsetSum** *is $O\left((n^3/\varepsilon) \log n\right)$.*

Proof: The running time bound is by Lemma 8.2.13.

As for the other claim, consider the optimal solution $\mathsf{opt} \in P_n$. By Claim 8.2.8, there exists $z \in L_n$ such that $z \leq \mathsf{opt} \leq (1+\delta)^n z$. However,

$$(1+\delta)^n = (1+\varepsilon/2n)^n \leq \exp\left(\frac{\varepsilon}{2}\right) \leq 1+\varepsilon,$$

since $1 + x \leq e^x$ for $x \geq 0$. Thus, $\mathsf{opt}/(1+\varepsilon) \leq z \leq \mathsf{opt} \leq t$, implying that the output of **ApproxSubsetSum** is within the required range. ∎

8.3. Approximate Bin Packing

Consider the following problem.

Min Bin Packing

Instance: $a_1 \ldots a_n - n$ numbers in $[0, 1]$
Question: Q: What is the minimum number of unit bins do you need to use to store all the numbers in S?

Bin Packing is NP-Complete because you can reduce Partition to it. Its natural to ask how one can approximate the optimal solution to Bin Packing.

One such algorithm is ***next fit***. Here, we go over the numbers one by one, and put a number in the current bin if that bin can contain it. Otherwise, we create a new bin and put the number in this bin. Clearly, we need at least

$$\lceil A \rceil \ \textit{bins} \quad \textit{where} \ \ A = \sum_{i=1}^{n} a_i$$

Every two consecutive bins contain numbers that add up to more than 1, since otherwise we would have not created the second bin. As such, the number of bins used is $\leq 2\lceil A \rceil$. As such, the next fit algorithm for bin packing achieves a $\leq 2\lceil A \rceil / \lceil A \rceil = 2$ approximation.

A better strategy, is to sort the numbers from largest to smallest and insert them in this order, where in each stage, we scan all current bins, and see if can insert the current number into one of those bins. If we can not, we create a new bin for this number. This is known as ***first fit***. We state the approximation ratio for this algorithm without proof.

Theorem 8.3.1. *Decreasing first fit is a* 1.5*-approximation to Min Bin Packing.*

8.4. Bibliographical notes

One can do 2-approximation for the k-center clustering in low dimensional Euclidean space can be done in $\Theta(n \log k)$ time [FG88]. In fact, it can be solved in linear time [Har04].

Bibliography

[FG88] T. Feder and D. H. Greene. Optimal algorithms for approximate clustering. In *Proc. 20th Annu. ACM Sympos. Theory Comput.* (STOC), pages 434–444, 1988.

[Har04] S. Har-Peled. Clustering motion. *Discrete Comput. Geom.*, 31(4):545–565, 2004.