Point clouds: subsampling and neighborhood

Abdessamad EL KABID, Zakaria BABA

Abstract

Ce rapport examine les méthodes de sous-échantillonnage et d'analyse de voisinage au sein des nuages de points, essentielles dans des domaines tels que la vision par ordinateur. Nous détaillons les implémentations algorithmiques, analysons leur complexité et discutons de leurs applications pratiques.

1 Implémentations Algorithmiques

1.1 Algorithmes de Sous-échantillonnage

Nous introduisons une approche naïve pour le sous-échantillonnage, visant à calculer une séquence de centres à partir d'un ensemble de points, avec une complexité de $O(n^3)$. Cette méthode s'appuie sur le calcul des distances Euclidiennes entre les points pour déterminer les centres.

Pour tout $x \in R^d$, nous notons $d(x, S) = \inf_{s \in S} d(x, s)$ (où le dernier $d(\cdot, \cdot)$ représente la distance euclidienne entre les points) et nous appelons le s qui atteint le minimum le **centre** de x . p_i est alors défini comme le point $p \in P \setminus S_{i-1}$ qui maximise $d(p, S_{i-1})$.

1.1.1 Task 1

Un algorithme naïf qui prend en entrée P, $p_1 \in P$ et k et calcule la séquence des centres $p_1, \ldots, p_k$, avec une complexité $O(n^3)$.

Algorithm 1 Algorithme Naive_Algorithm

Require: P : Ensemble de points, i : Indice du premier centre, k : Nombre total de centres à sélectionner.
Ensure: Liste des indices des points sélectionnés comme centres.
1: Initialiser $centers$ avec $[i]$.
2: **for** $i = 1$ **to** $k - 1$ **do**
3: Trouver le point le plus éloigné de l'ensemble actuel $centers$.
4: Ajouter ce point à $centers$.
5: **end for**
6: **return** $centers$.

Analyse :

Complexité : $O(nk^2)$

À chaque itération, on parcourt tous les points pour trouver le plus éloigné de S. Cela prend $O(i(n-i))$ où i est le nombre de centres.

D'où:

$$\begin{aligned}
\sum_{i=1}^{k-1} i(n-i) &= \sum_{i=1}^{k-1} (in - i^2) \\
&= \sum_{i=1}^{k-1} in - \sum_{i=1}^{k-1} i^2 \\
&= n \sum_{i=1}^{k-1} i - \sum_{i=1}^{k-1} i^2 \\
&= n \left(\frac{(k-1)k}{2} \right) - \frac{(k-1)k(2k-1)}{6} \\
&= \frac{n(k-1)k}{2} - \frac{(k-1)k(2k-1)}{6} \\
&= \frac{(k-1)k}{6}(3n - (2k-1)) \\
&= O(nk^2)
\end{aligned}$$

1.1.2 Task 2

Il est possible de calculer cette séquence de manière plus efficace. L'idée clé est de se souvenir, pour chaque point dans $P \setminus S$, de sa distance à S. Il est alors possible de *trouver en temps linéaire le maximum de cette liste.* Une fois le nouveau point p_i déterminé, la mise à jour de la liste des distances à $S \cup p_i$ prend également un temps linéaire.

Un algorithme qui calcule la séquence $p_1, \ldots, p_k$, avec une complexité $O(nk)$:

Algorithm 2 Algorithme Efficient_Algorithm

Require: P - Ensemble de points, i - Indice du premier centre, k - Nombre total de centres à sélectionner.

Ensure: Liste des indices des points sélectionnés comme centres.

1: Initialiser *centers* avec $[i]$.
2: Initialiser *distances* pour stocker la distance de chaque point de $P \setminus S$ à S.
3: **for** _ **to** $k-1$ **do**
4: Sélectionner le point le plus éloigné de l'ensemble actuel *centers*.
5: Ajouter ce point à *centers*.
6: Mettre à jour les distances de $P \setminus S$. (en $O(n)$)
7: **end for**
8: **return** *centers*.

Complexité : $O(nk)$

À chaque itération, on parcourt tous les points pour trouver le plus éloigné du S en utilisant les distances de chacun à S.

Parcourir "distances" prend $O(n-i)$ où i est l'itération courante, donc c'est au plus linéaire.

D'où la complexité $O(nk)$.

1.1.3 Task 3 :

Pour trouver p_i, au lieu de parcourir tous les points de P (ou $P \setminus S$) pour trouver celui avec la distance maximale à S, nous pourrions maintenir pour chaque centre $s \in S$ la liste des points de $P \setminus S$ qui ont s comme leur centre (nous appelons cette liste la **région** de s), en s'assurant que celui le plus éloigné de s peut être rapidement accédé (nous appelons rayon de s la distance maximale de s à un point de sa région). Ainsi, nous avons seulement besoin d'itérer sur S pour trouver le centre avec le rayon maximal.

Mise à jour : d est la distance euclidienne et en particulier une métrique, donc elle satisfait l'inégalité triangulaire $d(a,c) \leq d(a,b) + d(b,c)$. Après avoir sélectionné p_i comme nouveau centre, puisque la distance de p_i à s_j est très grande par rapport au rayon de s_j, tous les points qui avaient s_j comme leur centre garderont le même centre, nous n'avons pas besoin de les vérifier un par un. Nous pouvons tester pour chaque ancien centre si le nouveau centre est assez proche pour qu'il puisse voler des points pour sa région, et vérifier uniquement les points de ces régions en détail.

Une condition nécéssaire pour cela est que $\mathbf{d(p,s)} \leq \mathbf{2Rayon(s)}$, où p est le nouveau centre, s est un centre.

Un algorithme qui calcule la séquence $p_1, \ldots, p_k$ de manière efficace tant que k n'est pas trop grand :

Algorithm 3 Efficient Algorithm small k

Require: P - Ensemble de points, i - Indice du premier centre, k - Nombre total de centres à sélectionner.

Ensure: Liste des indices des points sélectionnés comme centres.

Initialisation

$T \leftarrow \{p \mid p \in P, p \neq p_1\}$

$m_1 \leftarrow \text{argmax}_{p \in T}(\text{normy}(p, p_1))$

$s \leftarrow s \setminus \{m_1\}$

$centers \leftarrow \{p_1 : [m_1, s]\}$ {m_1 est un point sur la frontière de s, s est la région de p_1}

Boucle

for _ **to** $k - 1$ **do**

  $centre \leftarrow \text{argmax}_{p \in \text{centers}}(\text{distance}(\text{centers}[p][0], p))$ {Trouver le centre de rayon maximal en temps linéaire }

  $farthest \leftarrow \text{centers}[centre][0]$ {Le nouveau centre}

  $steal \leftarrow \{\}$ {La région du nouveau centre}

  $m \leftarrow 0$

  $boy \leftarrow$ farthest

  for $s, value$ in $centers$ **do**

    $sf \leftarrow \text{distance}(s, \text{farthest})$
    {Condition nécéssaire}

    if $sf < 2 \times \text{distance}(s, value[0])$ **then**

      Chercher si l'un des points de la région peut être volé et mettre steal à jour en calculant les nouveaux rayons actuels

    end if

    Mettre à jour le rayon de l'ancien centre

  end for

  Mettre à jour le rayon du nouveau centre

end for

return Clés de $centers$

Complexité:
Notons $R(j)$ le rayon de la région de j.
On a $0 \leq R(j) \leq n$.
La complexité est :

$$\begin{aligned} C &= \sum_{i=1}^{k-1} i + k \cdot R(j) \quad \text{Si la condition d'entrée est vérifiée} \\ &= O(k^2) + O(nk) \quad \text{car } R(j) \leq n \\ &= O(nk) \quad \text{Au pire des cas} \end{aligned}$$

1.1.4 Task 4

Lorsque k devient plus grand, itérer sur tous les éléments de S devient trop coûteux.

Trouver p_i. Au lieu de chercher répétitivement l'élément de S avec le rayon maximal en temps linéaire, il peut être plus efficace de stocker les éléments de S dans une structure de données qui permet un accès rapide à celui avec le rayon maximal.
Pour cet algorithme , on a implementé un tas (***maxheap***) afin d'extraire le max plus rapidement et aussi de pouvoir changer les priorite des elements en mettant à jour le tas à chaque fois .

Mise à jour. Nous pouvons stocker, pour chaque centre s, une liste d'*amis*, c'est-à-dire d'autres centres $t_1, \ldots, t_{js}$ dans S qui sont assez proches de s pour que si l'algorithme choisit un nouveau centre p_i qui avait s comme son centre au temps $i-1$ (nous appelons s le parent de p_i), alors tous les points qui ont p_i comme leur nouveau centre au temps i avaient soit s soit l'un des t_j comme leur centre au temps $i-1$, donc nous avons seulement besoin de vérifier si ces points changent de région. Cela fonctionne toujours si le graphe d'amis est conservativement plus grand que strictement nécessaire (dans le cas extrême le graphe complet), bien que nous voulions le garder raisonnablement petit pour l'efficacité.

Un algorithme qui calcule la séquence $p_1, \ldots, p_k$ de manière efficace.

La logique d'amitié entre les centres est la suivante: Elle n'est pas commutative , et S_2 est ami de S_1 si $\text{distance}(s_1, s_2) - \text{rayon}(s_1) \leq 2 \times \text{rayon}(s_2)$. Cette condition est **nécessaire**.

Complexité:
Notons C la complexité de l'algorithme.
On a $C = 2n + \sum_{i=1}^{k-1} F(i) \cdot (\log(i) + O(1)) = O(n + \sum_{i=1}^{k-1} F(i) \log(i))$ où $F(i)$ est le nombre d'amis de l'ancien centre au temps i.
Si on suppose que le nombre d'amis est borné par f, c'est-à-dire $F(i) \leq f$, alors
$C \leq n + f \sum_{i=1}^{k-1} \log(i) \leq O(n + fk \log(k))$

Algorithm 4 Algorithme Efficace pour un Grand k

Require: P (ensemble de points), i (indice du premier centre), k (nombre de centres)

Ensure: Centres (séquence sélectionnée de centres)

Initialiser les centres avec le premier centre i
Créer un tas maximal *radii* pour gérer les centres selon leurs distances
Pour chaque centre dans S, calculer son rayon et l'insérer dans *radii*
Initialiser *amis* et *régions* pour le suivi des centres proches et de leurs points associés
for k fois **do**
    Extraire le centre avec le rayon maximal de *radii*
    Identifier le point le plus éloigné du centre actuel et mettre à jour les centres
    Pour chaque *ami* du centre actuel, évaluer les points potentiels à voler selon les critères de distance
    Insérer le nouveau centre dans *radii* avec son rayon calculé
    Mettre à jour les *régions*, *radii*, et les listes d'*amis* selon le nouveau centre et les points volés {selon la condition citée :S_2 est ami de S_1 si distance$(s_1, s_2) -$ rayon$(s_1) \leq 2 \times$ rayon(s_2)}
end for
return La séquence des centres sélectionnés

2 Applications Pratiques

Les algorithmes développés trouvent des applications dans la segmentation de nuages de points, la réduction de bruit, et la reconstruction de surfaces en vision par ordinateur, où l'efficacité et la précision sont primordiales.

Application 1: Retrouvons la cible

La capacité de se disperser rapidement dans un ensemble de points de cet algorithme nous a fait penser à la question suivante : est-ce mieux de chercher une cible dans un ensemble de points aléatoirement ou par un échantillonnage progressif ?

Si on considère un graphique avec des points bleus formant un rectangle et un cercle localisé quelque part, coloré en jaune et considéré comme cible, on cherche la probabilité de trouver cette cible pour des valeurs initiales de l'algorithme d'échantillonnage prises au hasard.

Etonnement , la forte delocalisation de l'algo n'est finalement pas un point fort pour lui permettre de retrouver une cible lorsqu'on compare ses performances avec un modele de recherche aleatoire notre algorithme fait defaut

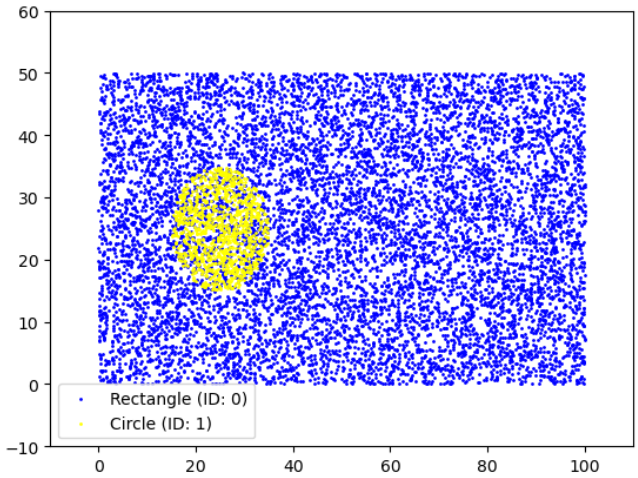

Figure 1: Localisation de région

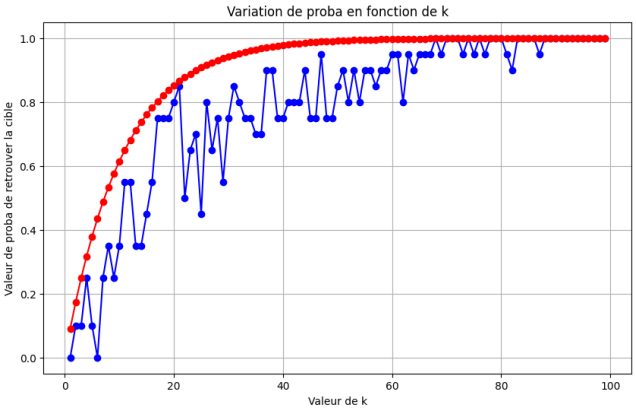

Figure 2: Localisation de région

Application 2:

L'algorithme permet de reduire efficacement un ensemble de data a un sous ensemble qui garde les propietés de l'original.

Exemples:

1. Chat:

Figure 3: Image originale (48738 points)

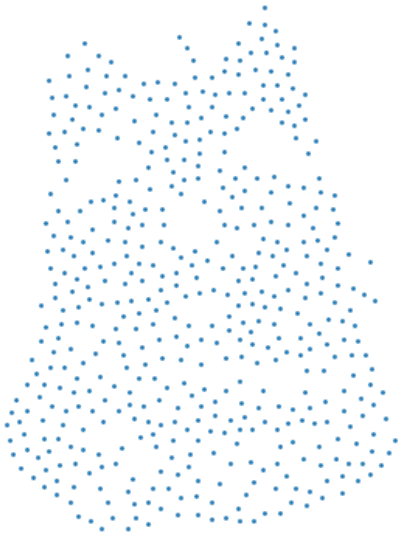

Figure 4: 500 points

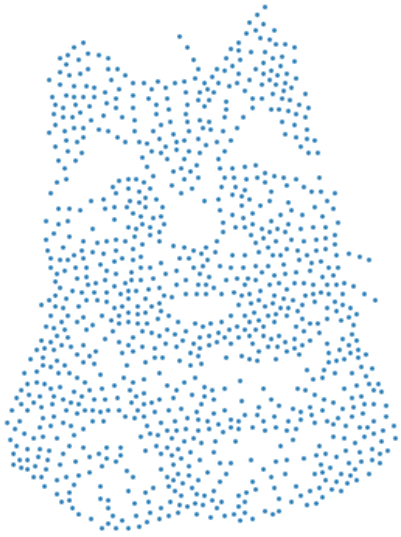

Figure 5: 1000 points

2. Tour Eiffel

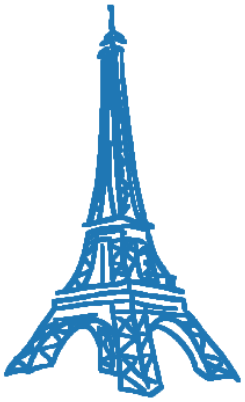

Figure 6: Image originale (76711 points)

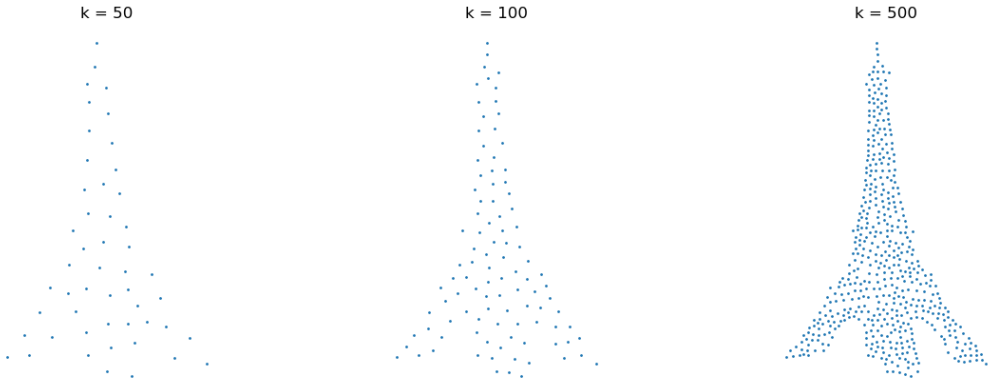

Figure 7: subsampling

Application 3 :

Réduction de bruit Ici on trace un sinus additionné à un bruit gaussien , on effectue un sous-échantillonnage de la courbe et on obtient une approximation de la courbe initiale:

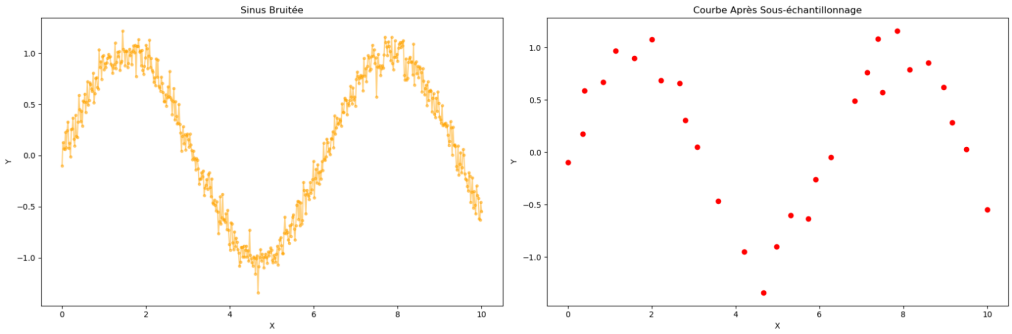

Figure 8: subsampling

2.1 Neighborhood

2.1.1 Task 5

Formellement, dans un espace de dimension d, pour un ensemble P de n points, un graphe G est construit. Chaque point dans P correspond à un sommet dans G, et deux sommets p_i et p_j sont connectés si aucun autre point $p_k \in P$ n'empêche la connexion directe entre p_i et p_j, selon le critère $\max(d(p_i, p_k), d(p_j, p_k)) < d(p_i, p_j)$.

Nous allons d'abord proposer une version naïve de l'algorithme, puis nous essaierons de l'améliorer, pour finalement proposer un lien avec *le sampling* de la partie 1.

Implémentation naive Pour l'implementation naïve la complexité est cubique car on procede à la verification de la propriété pour tous les trois paires de tuples. Nou avons utilisé la programmation orientée objet pour faciliter l'organisation du code .

Algorithm 5 Fonction Relier

Require: $point1, point2$ - Les deux points entre lesquels vérifier la connexion, *sommet* - Ensemble de tous les points
Ensure: Booléen - Vrai si un lien doit être créé, Faux sinon

```
Calculer la distance d_12 entre point1 et point2
for chaque point dans sommet do
    if point ≠ point1 and point ≠ point2 then
        Calculer la distance entre le point et point1, et entre le point et point2
        if la plus grande de ces deux distances est inférieure à d_12 then
            return Faux
        end if
    end if
end for
return Vrai
```

2.1.2 Task 6 :

Plus efficace: Nous décrivons ici une manière de calculer tous les voisins d'un point p en temps $O(n)$. En partant de $P \setminus p$, nous trouvons d'abord le point q le plus proche de p. q est clairement un voisin de p. Nous remarquons alors que pour un point $r \in P \setminus p, q$, q empêche r d'être un voisin de p si et seulement si r est plus proche de q que de p. Nous supprimons donc tous ces points r plus proches de q, et recommençons avec les restants : trouver celui le plus proche de p, etc. Nous nous arrêtons quand il ne reste plus de points. Cependant, bien que cette liste contienne tous les voisins, elle peut contenir des points supplémentaires. En effet, nous avons retiré des points lors des premiers tours parce qu'ils ne pouvaient pas être voisins, mais ils pourraient encore jouer un rôle dans l'empêchement des "points les plus proches" ultérieurs d'être voisins. Nous avons donc besoin d'un passage supplémentaire pour vérifier quels candidats parmi ces $O(1)$ sont réellement des voisins, ce que nous pouvons faire par force brute, en vérifiant si un autre point de P l'empêche (ou nous pouvons fusionner les 2 passages, tant que le comportement est le même).

Un algorithme qui prend en entrée P et calcule le graphe G, avec une complexité quadratique.

Algorithm 6 Fonction Voisin

Require: x - Le point pour lequel trouver les voisins, *sommet* - Ensemble de tous les points
Ensure: L - Liste des voisins de x
1: Copier *sommet* dans P et retirer x de P
2: Trier P par distance croissante par rapport à x, stocker dans *liste_triee*
3: Initialiser L comme liste vide
4: **while** *liste_triee* n'est pas vide **do**
5: Sélectionner *proche* comme le point le plus proche de x dans *liste_triee*
6: Ajouter *proche* à L
7: Nettoyer *liste_triee* en retirant les points plus proches de *proche* que de x
8: **end while**
9: **for** chaque voisin dans L **do**
10: Vérifier si aucun élément de P ne l'empêche d'être voisin
11: **end for**
12: **return** L

2.1.3 Task 8

Pour combiner les tâches 7 et 3, il est pertinent de noter que l'utilisation de la structure de graphe, définie dans la deuxième partie du projet, permet d'éliminer plus efficacement les éléments de $P \setminus S$ à examiner.

En ajoutant un point q à S, et en utilisant la condition suivante :

$$\text{distance}(P[\text{farthest\_point}], P[\text{cen}]) < 2 \times \text{rayon}[\text{cen}]$$

on identifie que les éléments rattachés au centre *cen* peuvent devenir des cibles potentielles pour le nouveau centre.

Selon la définition du graphe dans la partie 2, on établit l'équivalence suivante : Un point x est considéré comme voisin de p si et seulement si il ne peut pas être dissocié de p par l'introduction d'un nouveau centre.

Ainsi, nos recherches se limiteront aux points qui ne sont pas voisins de p, afin de rationaliser l'identification des nouveaux centres et de simplifier la mise à jour des régions associées à chaque centre.

2.2 Benchmarking

On génére des données artificielles :nuage uniforme, gaussien,une grille, des clusters , un rectangle bruité:

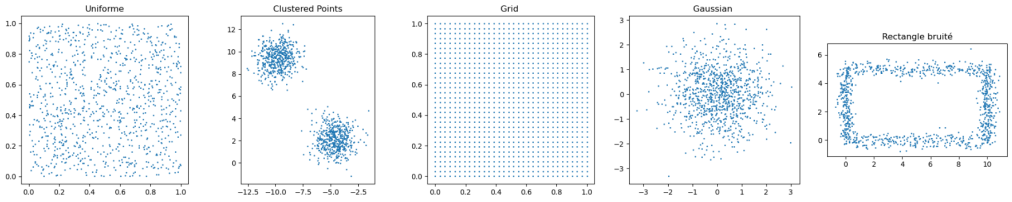

Figure 9: subsampling

Dataset	Naive	Efficient	Efficient_smallk	Efficient_bigk
Uniform	0.256280	0.002732	0.004053	0.004323
Clustered	0.040235	0.002497	0.003518	0.004694
Grid	0.040258	0.002374	0.003700	0.007565
Gaussian	0.046099	0.002873	0.005422	0.005969
Noisy_rectangular	0.041345	0.003704	0.003649	0.004290

Table 1: Benchmarking results for $n = 100$, $k = 50$ (in seconds)

Dataset	Naive	Efficient	Efficient_smallk	Efficient_bigk
Uniform	0.445550	0.072819	0.056783	0.088298
Clustered	0.444384	0.072214	0.063364	0.107601
Grid	0.451936	0.069149	0.050311	0.099347
Gaussian	0.665619	0.077426	0.059325	0.115149
Noisy_rectangular	0.808913	0.071464	0.049792	0.066383

Table 2: Benchmarking results for $n = 1000$, $k = 50$ (in seconds)

Dataset	Naive	Efficient	Efficient_smallk	Efficient_bigk
Uniform	40.333086	0.403157	0.417747	0.228504
Clustered	34.041219	0.439012	0.446190	0.253888
Grid	29.256016	0.470698	0.490613	0.230541
Gaussian	30.640978	0.472149	0.482225	0.294409
Noisy_rectangular	32.415050	0.428818	0.405749	0.159050

Table 3: Benchmarking results for $n = 1000$, $k = 300$ (in seconds)

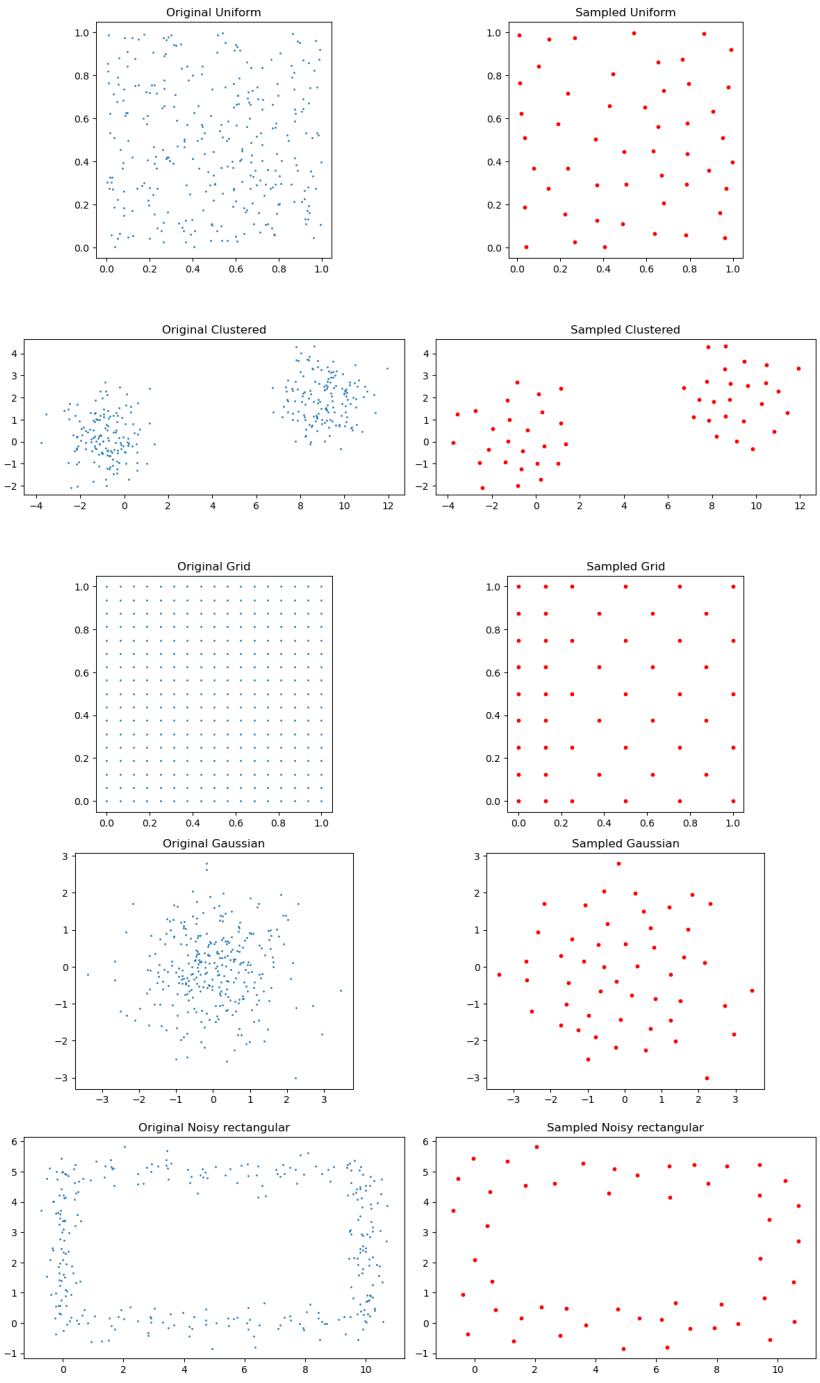

Figure 10: n=300 , k=50