Data Mining - Homework 3

Valerio Trenta - matr. 1856471

December 8, 2019

1 Exercise 1

To answer the two questions below, we need to recall what the **k-means** algorithm does and how.
If we assume the cost function of the algorithm to be

$$\mathrm{y} = \sum_{k=1}^{K} \sum_{x_\mathrm{i} \in C_\mathrm{k}} ||x_\mathrm{i} - \mu_\mathrm{k}||_2{}^2$$

then the algorithm is:

- *randomly initialize K cluster centroids;*
- *for each point $v \in V$, compute the* ***distance*** *(in this case, distance is the* ***squared Euclidean distance****) from v to each of the centroids in each cluster, then assign v to the cluster corresponding to its nearest centroid;*
- *for each cluster* ***k****, recompute its centroid to be the* ***mean*** $\mu_{\boldsymbol{k}}$ *of the points contained inside the cluster itself;*
- *repeat the previous two points until the convergence condition is met - that is to say, when a prefixed number of iterations has been reached, or the centroids and the clusters do not change anymore between one iteration and the other.*

By building K clusters through this algorithm, we are minimizing the aforementioned cost function **y**, because we are taking each point belonging to each of the **K** clusters in such a way that the distance between the point itself and the mean of the cluster (μ_k) is minimized, and so the sum of squares of this distance is also minimized, so that the cost function we defined previously cannot increase at each step, meaning we reach at each iteration a value that is going to be at most equal to the one obtained in the previous iteration.

1.1 1

What happens if we change the cost function?
Actually, what happens if, instead of the aforementioned cost function, we exploit the alternative one?
By exploiting the "alternative one", we mean that instead of exploiting the **L2-norm** to compute the cost function, we exploit the **Manhattan distance**, or **L1-norm**.
The main problem with exploiting a different distance measure is that, as we already know by the results obtained and known in literature, there may be *no closed-form solution* for the centroid estimation problem of the **k-means** algorithm by exploiting other distance measures that are not the L2-norm. This means that the algorithm may not converge if we exploit it, but we are lucky in this case: with the given cost function, we know from literature the algorithm will eventually converge.
So what we obtain, is necessarily a different algorithm from **k-means**, since what changes is the way each centroids are computed and recomputed - and this algorithm is the well-known **k-medians**.
The difference between the two algorithms lays in what we are trying to minimize: the **k-means** algorithm, indeed, minimizes the **L2-norm**, while the **k-medians** minimizes obviously the **L1-norm**, and this is because the first, when computing the new centroid of each cluster, computes the **mean** of the points in the cluster itself, while the latter computes the **median**, which results in a point that may also not belong to the set of points **V** at all.[**?**]
So, this is the algorithm we end up with when considering the alternative cost function $\mathrm{y}_2 = \sum_{k=1}^{K} \sum_{x_{\mathbf{i}} \in C_{\mathbf{k}}} ||x_{\mathbf{i}} - \mu_{\mathbf{k}}||_{\mathbf{1}}$:

- *randomly initialize K cluster centroids;*
- *for each point $v \in V$, compute the **distance** from v to each of the centroids in each cluster, then assign v to the cluster corresponding to its nearest centroid;*
- *for each cluster **k**, recompute its centroid to be the **median** μ_k of the points contained inside the cluster itself;*
- *repeat the previous two points until the convergence condition is met - that is to say, when a prefixed number of iterations has been reached, or the centroids and the clusters do not change anymore between one iteration and the other.*

The only change that can be detected between this and the previous algorithm, is the fact that the centroids of each cluster are now computed as the **median** points of each cluster to obtain the updated value of the centroids themselves. This way, the aforementioned cost function $\mathbf{y_2}$ that is computed by considering the distance between each point in the cluster and the value of the updated median for each cluster, can at most be equal to the value obtained at

the previous iteration, so the sum of all the distances for each cluster is always minimized, since it can never grow.
The change in the algorithm is due to the different distance measure we are exploiting. Indeed, the measure distance we exploited in the **k-means** algorithm is the *squared* **L2-norm**, or *squared Euclidean distance*, and is defined as follows (omitting the sums, but the indexes are there):

$$||x_{\mathrm{i}} - \mu_{\mathrm{k}}||_2{}^2 = \sqrt{|(x_{\mathrm{i}} - \mu_{\mathrm{k}})|^2}^2 = |(x_{\mathrm{i}} - \mu_{\mathrm{k}})|^2$$

that is, indeed, the definition of the **squared Euclidean distance** between two given points, in our case the point $\mathrm{x_i}$ belonging to the k-th cluster $\mathbf{C_k}$, and the mean of the cluster itself, μ_{k}.
It is squared because this way the algorithm does not have to compute the square root of each distance, but it's basically giving the same results.
The **L1-norm**, instead, is defined as follows:

$$||x_{\mathrm{i}} - \mu_{\mathrm{k}}||_1 = |(x_{\mathrm{i}} - \mu_{\mathrm{k}})|$$

meaning this is the aforementioned Manhattan distance, and if before minimizing the **squared Euclidean distance** between the points in the cluster and its centroid meant to find the **mean** of the cluster itself, since it is the value which minimizes the distance between all the points in the cluster - so that **k-means** is actually the algorithm which, indeed, minimizes the distance of the points within the cluster they were assigned to - now minimizing the **Manhattan distance** means finding the value of the **median** of each cluster, since the **median** itself is the value that, for each cluster, minimizes the **Manhattan distance** between the points of the cluster itself.
Indeed, given μ_{k} being the value we want to find in our second algorithm - that is, the value of the centroid for cluster k - the previous statement can be proved. Assume we only consider one cluster for now, thus the cost function to minimize is:

$$\mathrm{y_{2,\,k}} = \sum_{x_{\mathbf{i}} \in C_{\mathbf{k}}} ||x_{\mathbf{i}} - \mu_{\mathbf{k}}||_{\mathbf{1}}$$

and we define the median as any number m with $\mathrm{x_{n/2}} \leq \mathrm{m} \leq \mathrm{x_{n/2+1}}$ if the cluster has **n** points in it with **n** being even and as $\mathrm{x}_{\frac{(n+1)}{2}}$ if the cluster has an odd number of points.
So, assume we compute the median on the set of points belonging to our cluster $\mathrm{C_k}$ by sorting the points in increasing order, so that if we have **n** points, we have $\mathrm{x_1 \leq x_2 \leq ... \leq x_n}$. Now, the cost function can be written as:

$$\mathrm{y_{2,\,k}} = \sum_{x_{\mathbf{i}} \in C_{\mathbf{k}}} ||x_{\mathbf{i}} - \mu_{\mathbf{k}}||_{\mathbf{1}} = \sum_{x_{\mathrm{i}} \in C_{\mathrm{k}}} |x_{\mathrm{i}} - \mu_{\mathrm{k}}| = \sum_{x_{\mathrm{i}} \in C_{\mathrm{k}}} |(x_1 + ... + x_{\mathrm{n}}) - n\mu_{\mathrm{k}}|$$

assuming we have **n** points assigned to the cluster. Keep in mind that μ_k the value of the median of the points in the cluster that we defined before, so it's a constant.The median compute on the set of these points splits this set in such a way that we can get rid of the modulus operand, because it will be greater than some points in our ordered set, and less than some other points, so we can write the function as:

$$y_{2,\,k} = \sum_{x_i > \mu_k} (x_i - \mu_k) + \sum_{x_j \leq \mu_k} (\mu_k - x_j)$$

for all $x_i \in [\mu_k, x_n]$ and for all $x_j \in [x_1, \mu_k]$.
The function we obtained is linear in the interval $[x_f, x_g]$ being x_f the point in the cluster with a maximum value $\in [x_1, \mu_k]$ and x_g the point in the cluster with the minimum value $\in [\mu_k, x_n]$, so that $x_f \leq \mu_k \leq x_g$, meaning we can derivate $y_{2,\,k}$ in this interval.
Now, if we take the first derivative with respect to $\boldsymbol{\mu_k}$ of $\mathbf{y_2}$, since μ_k is the value of the centroid which changes from iteration to iteration. So, say that **A** = number of x_i such that $x_i \in [\mu_k, x_n]$ and **B** = number of x_j such that $x_j \in [x_1, \mu_k]$. This means that each of the two summations have a number of **A** and **B** elements, so we get that:

$$\frac{dy_{2,\,k}}{d\mu_k} = \mathbf{1*A\text{-}1*B}$$

and we know that $y_{2,\,k}$ has a minimum if and only if this derivative is equal to zero, which is a condition that happens if and only if **A=B**, meaning when we choose μ_k in such a way that, when we sort the points in the cluster, this value splits the set of points exactly *in half*, that is what the **median** does in either the case the cluster has an even number of points, or an odd number of points - not *exactly* in half to be fair in the latter case, but it is the best approximation we can get.
At this point, we know that for each cluster we define, taking the median as a centroid will minimize the cost function for each cluster (the **L1-norm**) and thus it will minimize the global cost function for all the clusters.
This is why the **k-means** algorithm we were given must be, as aforementioned, modified in such a way that it computes the centroids for each cluster not as their means, but as their medians.

1.2 2

[CSE 291: Unsupervised Learning - Lecture 3: The k-medoid clustering problem]
We stated that by exploiting the following cost function for the **k-means** problem:

$$y = \sum_{k=1}^{K} \sum_{x_i \in C_k} ||x_i - \mu_k||_2{}^2$$

we can compute an optimal solution by computing and recomputing each centroid as the *mean* of the set of points in each cluster so to minimize the *squared L2-norm*, which is in fact the *squared euclidean distance* between the points in the cluster and the centroid.
Now, the point in doing this is that we get the solution with cost **C** given the fact that the value for the mean of each cluster - μ_{k}, can indeed not belong to the set of points in the cluster, but it can even be a point that is not in the set of points **V**, which can also happen in the **k-medians** algorithm when the number of points in the cluster is odd.
Under the costraint that we must take the centroid so as the centroid itself belongs to the set of points **V**, the algorithm for the **k-means** changes a bit, and is the following one:

- *randomly initialize K cluster centroids;*
- *for each point $v \in V$, compute the **distance** (in this case, distance is again the **squared Euclidean distance**) from v to each of the centroids in each cluster, then assign v to the cluster corresponding to its nearest centroid;*
- *for each cluster **k**, recompute its centroid to be **the point c_k** $\in V$ with the value which is the one which minimizes the cost function $y_m = \sum_{k=1}^{K} \sum_{x_i \in C_k} ||x_i - c_k||_2{}^2$;*
- *repeat the previous two points until the convergence condition is met - that is to say, when a prefixed number of iterations has been reached, or the centroids and the clusters do not change anymore between one iteration and the other.*

this algorithm we have just given is one of the many versions that can be found in literature of the so-called **k-medoids** algorithm, which is indeed very similar to the **k-means** algorithm: the only difference is that the centroids are now defined as *medoids*, meaning that they are constrained to be points in the set of the points in the input, unlike the centroids of the **k-means** that are just computed as the mean of each cluster and so can also be points not belonging to the original input. Well, this is exactly what we are supposed to study in this question.
Furthermore, we now have these two cost functions, the first one for the **k-means** and the second one for the **k-medoids** algorithm:

- $y_{means} = \sum_{k=1}^{K} \sum_{x_i \in C_k} ||x_i - \mu_k||_2{}^2$;
- $y_{medoids} = \sum_{k=1}^{K} \sum_{x_i \in C_k} ||x_i - c_k||_2{}^2,\ c_k \in V$;

meaning that both algorithm aim at minimizing the *squared L2-norm* distance between each point in a given cluster and its centroid/medoid. We already know

- it is a very well known result in literature - that this distance is minimized when the centroid is the mean of the cluster itself, and the **k-means** algorithm indeed takes as centroids the means of each cluster, so on one hand we already know it minimizes this cost function.
Now, we can devise an algorithm which exploits the results we get from the **k-means algorithm** and round them in order to get a solution that is **feasible** for the **modified k-means** problem, or **k-medoids problem**. This is an approach that is very similar to the **LP formulation** of an **ILP** problem that is rounded and so approximated to a feasible solution, just like the 4-approximation for the **k-medoids** algorithm that can be found in the cited paper at the beginning of the section.

- solve the **k-means** problem, finding the optimal cost **C**, the clusters C_1, ..., C_k and the centroid (mean) for each optimal cluster **i**, μ_i;
- medoids = $\emptyset$;
- Clusters = $\emptyset$;
- residuals = $\emptyset$;
- for each cluster C^*_i with mean μ_i in the optimal solution:
- pick v $\in C^*_i \in V$ with minimum value of d(v, μ_i) for the i-th cluster in the optimal solution, and make it the "medoid" of the new cluster C_i;
- medoids = medoids $\bigcup$ {v};
- $C_i = C_i \bigcup \{v\} \bigcup V_{\text{to-cluster(i)}}$;
- residuals = residuals $\bigcup C^*_i - C_i$
- end for.
- output medoids, Clusters, residuals

What we do after computing the optimal solution for the **k-means**, is to iterate over the clusters that are provided by the optimal solution and, for each of them, we find the point **v** inside that cluster C^*_i that is the closest to the previously compute mean of that cluster, μ_i.*Notice that the distance we exploit d(i, μ_k) is still obviously the squared L2-norm.* We can then assume that, for each **v** that is chosen, d(v, μ_i) = C_v is the minimum distance between every other point in the optimal cluster and the centroid μ_i of the cluster since **v** was picked.
Now we make the following reasoning for just one cluster - say, C_i - to be made: if we include **v** in the set of our medoids of our solution, we can then make it the medoid of a new cluster which comprehends any other point **j** in the set of to-be-clustered points $\in C^*_i$ whose distance from **v** itself is **at most** $\mathbf{4C_v}$ = 4d(v, μ_i), so we have d(v, j) $\leq 4C_v \leq 4C_j$ with C_j = d(j, μ_i), and this is due

to the fact that, obviously, **v** has been currently chosen as the point with the smallest value of distance $\mathrm{C_v}$ from the centroid.
This may be already enough to prove our point, but there is more: another intuition is that we can exploit **v** so to make it the medoid of a cluster which comprehends any other point **j** $\in$ V such that the intersection between the set of points whose distance from **v** is **at most** $\mathbf{2C_v}$ and the set of points whose distance from **j** is **at most** $\mathbf{2C_j}$, is *not* empty.
In order not to lose focus, remember:

- $\mathrm{C_v} = \mathrm{d}(\mathrm{v}, \mu_\mathrm{i})$
- $\mathrm{C_j} = \mathrm{d}(\mathrm{j}, \mu_\mathrm{i})$

So we define $\mathbf{B(v,\ 2C_v)}$ the first set of points and $\mathbf{B(j,\ 2C_j)}$ the second one, so we can write $\mathrm{B(v,\ 2C_v)} \cap \mathrm{B(j,\ 2C_j)} \neq \emptyset$, meaning we want to include in the cluster every point **j** which has some other point at distance at most $\mathrm{2C_j}$ which is also at distance at most $\mathrm{2C_v}$ from **v**, the new medoid. What we stated before can be proven: for the two subsets of V to intersect and *not be empty*, they need to have some point **p** in common, so by triangle inequality - which holds under the squared L2-norm measure - we get, due also to the fact that $\mathrm{C_v}$ is indeed the minimum current cost:

$$d(v,j) \leq d(v,p) + d(j,\ p) \leq 2C_v + 2C_j \leq 4C_j$$

because obviously $\mathrm{2C_v + 2C_j \leq 2C_j + 2C_j = 4C_j}$ given $\mathrm{C_v \leq C_j}$ by initial assumption. This means that, by including any point **j** in the cluster which verifies this condition, we are including a point that is at distance **at most** $\mathrm{4C_j}$ $\leq 4\mathrm{d}(\mathrm{j}, \mu_\mathrm{i}$.
So, we get that when we select **v** as a medoid, by assigning any point $\mathrm{j} \in \mathrm{C^*{}_i}$ to cluster of **v** ($\mathrm{C_i}$) such that $\mathrm{B(v,\ 2C_v)} \cap \mathrm{B(j,\ 2C_j)} \neq \emptyset$ and so assigning it to medoid **v** if $\mathrm{d(v,j) \leq 4C_j}$, the distance that will be added to the total cost of the cluster by doing this will be at most $\mathrm{4C_j} = 4\mathrm{d}(\mathrm{j}, \mu_\mathrm{i})$, meaning that we are adding a cost to that cluster that is **at most** four times the cost that this point added to the optimal cluster.
We can thus legitimately mark as members of the cluster defined by **v** the set of points $\mathrm{V_{to\text{-}cluster(i)}}$ defined as follows:

$$\mathrm{V_{to\text{-}cluster(i)}} = \{\ \mathrm{j} \in \mathrm{C^*{}_i} \text{ such that } \mathrm{B(v,\ 2C_v)} \cap \mathrm{B(j,\ 2C_j)} \neq \emptyset\ \}$$

so that, at each iteration of the algorithm, we take **v** as a medoid and include inside its cluster $\mathrm{C_i}$ all the points $\mathbf{j} \in \mathrm{V_{to\text{-}cluster(i)}}$.
If we do this, we can prove that, for each cluster $\mathrm{C_i}$ corresponding to the optimal one $\mathrm{C^*{}_i}$, $\mathbf{cost(C_i) \leq 4cost(C^*{}_i)}$.
Indeed, if we pick any point $\mathrm{p} \in \mathrm{C^*{}_i}$, and we pick $\mathbf{v} \in \mathrm{C^*{}_i}$ as the point with minimum value of distance from the mean of the optimal cluster, so $\mathrm{C_v} = \mathrm{d}($v,

μ_i), for which $p \in V_{\text{to-cluster(i)}}$, both of them are the found in C^*_k when **v** is selected, so it follows that $C_v \leq C_p$ being $C_p = d(p, \mu_i)$.
Furthermore, for $p \in V_{\text{to-cluster(i)}}$, $\exists\, q \in C^*_i$ such that, by triangle inequality:

$$d(p, v) \leq d(p, q) + d(q, v) \leq 2C_p + 2C_v \leq 4C_p$$

because **q** is by definition of $p \in V_{\text{to-cluster(i)}}$ a point which is at distance $2C_p$ from p and $2C_v$ from v, and the sum of these two distances will be always **at most** equal to $4C_p$ given $C_v \leq C_p$.
This means that, every time we pick a new medoid **v** in the rounding algorithm proposed, all the other points **p** that are assigned to the cluster of medoid **v**, add a cost **d(p, v)** to the total cost of that cluster, so that d(p, v) is **at most** $\leq 4C_p = 4d(p, \mu_i)$, that is the cost which point **p** adds to the "original" optimal cluster in the optimal solution of **k-means algorithm**.
By repeating this reasoning for each point in each cluster, we get that the total cost that we obtain from this algorithm for all the clusters C_k is such that it is $\leq 4C$, because $d(p, \mu_k)$ would be the cost added for each of those point **p** when μ_i is actually selected as the centroid of the cluster being considered.
So in this way, we find exactly **k** clusters in the proposed solution, one for each optimal cluster of the optimal solution: we have basically "rounded" each cluster making sure that the cost that is compute for each of them is at most equal to four times the cost of the optimal solution.
Problem is, if everything goes right, then we find ourselves with **k** clusters such that, for each of them, $|C_i| = |C^*_i|$. However, this can surely happen, but it's the best case scenario. We are sure that no points from outside the optimal cluster will be added to the cluster of the solution, but we cannot be sure that *every* point in the optimal cluster will be added to the one of our solution. What I'm trying to say, is that $C_i \subseteq C^*_i$.
This means we could exclude, for each cluster in the solution, a certain number of points from the optimal cluster. We can consider these points as **"outliers"** in our solution, and this is what the set of **residuals** deals with in the algorithm aforementioned. We take each point that is excluded from each cluster we compute and store it in this set. Then, we make each of these points we have collected - the residuals, or outliers - the medoids of their own cluster. So, say we have **n** points in the **residuals** set, we add to the solution exactly **n** new clusters, but each of them is empty, it only contains its own medoid and, since by definition the cost function is computed for each cluster as the summation of all the distances between every point in the cluster and its medoid, by having only the medoid in the cluster, all these clusters from the **residuals** will have cost equal to zero - or, they will not add cost to the final cost of the algorithm.
So we can write, being C the value of the cost function for the **k-means problem** and **A** the algorithm proposed:

$$\mathbf{A \leq 4C}$$

that is to say, we have found that the cost of this ”rounded” solution is **at most** four times the cost of the solution given by the corresponding **k-means** problem.
Furthermore, we can also say that, being **n** the number of points in **residuals**, the algorithm will output exactly **k+n** clusters.

2 Exercise 2

In order to reach the goals set at the end of the exercise, apart from writing the code for the *k-means* implementation, we also have to change something in the pseudocode that was given. I am expecting the combination of **PCA** and **k-means** to be more efficient and hopefully successful in retrieving the ground clustering with respect to the implementation where k-means runs alone, especially for different parameters of d (dimensions) and σ, standard deviation, so I run many experiments in order to achieve these results and modified the code that was provided with this intent. I will discuss these changes, then I will discuss the algorithm that was implemented to compare the ground clusters and the ones obtained briefly, then I will discuss the results. I won't comment the whole code, would result in pages and pages of descriptions and such, hopefully the comments I added in the **.py** file are enough. The code is provided in the **exercise2.py** file, but keep in mind most of the results that I will show are obtained through a netbook exploiting a GPU hardware thanks to **colab**.

2.1 Changes in the code provided

One major ”mistake” can be immediately detected in the pseudocode to generate the dataset: when checking whether a point belongs to the j-th cluster ($i\%j == l$, there is some kind of typo, since **l** was never even mentioned before. That has to be replaced with ”j”.
I have been wondering if another line should have been changed, the one where, if the aforementioned condition is true, we assign a value to the j-th component of point i that is *data(i)(j) = 1*. Given what is written in the text of the exercise, that should be *data(i)(j)* $= \frac{1}{\sqrt{n}}$, but also that part of the text is tricky, because what we want to achieve with the identity matrix is just an ”approximatioN” of the theoretically best clustering matrix, and by the way if we don't set the points on the main diagonal to 1, we would lose the definition of identity matrix itself, so I set *data(i)(j) = 1*.
I changed twice this algorithm over the past two weeks, the first one because I found out this was very inefficient, the second one because I found out this was not applying *PCA* as it claimed to do. In order to make this algorithm efficient, I had to set the *full_matrices* parameter to *False*, and to make the algorithm consider only those parts of the matrices that were actually useful to our analysis by applying what is known as *truncated SVD*, meaning the useless parts of the

matrices were cut off (basically, I only kept the rows and columns up to the m principal components).
Then, if we want to apply PCA, we need to make some kind of dimensionality reduction, which is not actually applied with this algorithm that only applies SVD and approximates the components of each point by ”weighting” them. Through the matrices obtained by SVD, the principal components can be found through the dot product of the matrices **U** and **smat** in the algorithm, after having set all the components ”beyond” the m-th one to zero by exploiting the diagonal matrix obtained through **s**. Reducing the dimensions of the points makes sense, especially if we are going to run this algorithm over a dataset with high dimensions.
Problem is, the results were not that great - at least, not as expected. Approaching higher and higher dimensions, the running time decreased a bit, which made sense, but the ground clustering was recovered just as much as the implementation with the only k-means did - meaning an average of 6% per cluster. This may be due to the fact that if we want to recover the ground clustering and we want to to this efficiently exploiting the PCA algorithm, we need to highlight the component that we know is the one which clusters the point, that is the one we set when building the dataset as *data(i)(j) = 1*. If we set this to one and then we normalize it - that is something we should do when applying PCA since all the other points are just noise from a Gaussian and are already normalized, and normalizing all the other zeroes in the row of the identity matrix would make no sense - by dividing it by the standard deviation of the Gaussian, we find out that PCA combined with k-means recovers the ground clusters in a very efficient way and with great results. This is due to the fact that, for higher values of the standard deviation and number of points, the random Gaussian value that may be generated for a component of a specific point could have a variance with respect to the mean of the Gaussian (that is always set to zero) that is so high that the PCA algorithm may think to select it as a principal component, and if the point we care about the most in our dataset (the one on the main diagonal of our identity matrix) is always set to one, it will be eventually discarded by the PCA because its variance with respect to the Gaussian is not great enough. So, doing this in the PCA algorithm is a way to ”highlight” the point and tell the algorithm that that component is indeed a principal component, and maybe the most important one.
One could also set directly, when building the dataset, *data(i)(j) = 1/ sigma*, and the k-means algorithm would probably benefit from this, but then again we would loose the definition of identity matrix when building the dataset, so I decided to normalize this point with respect to the Gaussian values only in the PCA algorithm, that is also something which highlights the performance of PCA itself with respect to the lonely implementation of k-means.

2.2 How the retrieved and ground clusters were compared

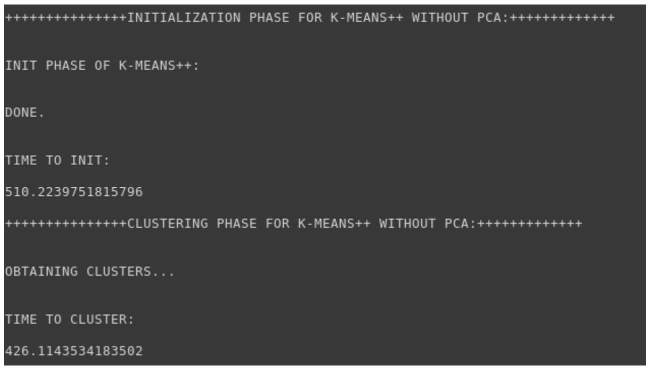
I don’t think the method that I exploited is 100% accurate, but it should give an idea of the amount of ground clusters that were retrieved when applying

k-means.What I did was retrieving the ground clusters by reversing the function exploited to create the dataset: for each point i, if the j-th cluster is such that i%k == j, put point **i** in cluster **j**. This had to be done twice, once for the dataset without PCA, once for the dataset with PCA, since obviously the points were projected in the second so I needed to have two ground clusters - even if, theoretically speaking, they represent the same ground clusters. Another simple and probably more efficient approach would be to retrieve the ground cluster and then make a copy of it applying on its points the PCA algorithm, but the computational effort was not so huge during this operation so I let it be.
The clusters retrieved with k-means and the ones retrieved with PCA and k-means implementations are then compared to the ground clusters aforementioned. This is done by mapping the clusters in the ground clustering to the ones with the highest **Jaccard** similarity among the retrieved clusters.
Notice that this approach could not be optimal: indeed, we do not know a priori whether one retrieved cluster $\mathbf{c_i}$ with the highest Jaccard similarity for one ground cluster $\mathbf{c_j}$, has the highest Jaccard similarity for c_j and c_j only. It could also be the best matching for another cluster in the ground clustering as far as we know, leaving out another cluster, so this mapping may not be univocal, there may be conflicts. Still, it's one of the fastest - and most greedy - ways to compare the clusters and in practice seemed to work fine.

2.3 Results

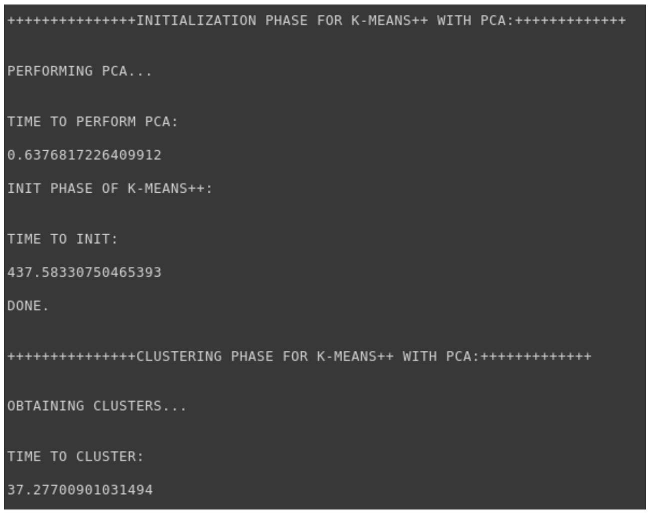
These results were obtained by running the code in a notebook with a GPU hardware in **colab**. For most of the inputs that were proposed for the variables, the running time for the k-means was so high that I thought it was pointless to try and let it finish, so I only tried some of those inputs - the ones with the lowest values - for k-means and then run some tests with PCA and k-means implementation.
So we could say that, by reducing the dimensionality of the points, the first implicit result that is something that was expected, is that the running time for the second configuration - PCA and k-means - is way lower than the running time of k-means alone: in terms of efficiency, it's the best option.
It is also in terms of ground clusters recovered - if the algorithm for the comparison does not trick us.
Pictured in **Figure 1** and **Figure 2** are the results for inputs **k = 50, n = 1000, d = k, stdev = 1/k**.

Some preliminary results can be obtained by looking at this comparison: if we apply k-means, in this case, we find out that the running time for the whole process is much less in the second configuration, the one in which PCA was actually exploited. This is due to the fact that with PCA we are only working on the **k** (we set m=k in the PCA algorithm) principal components of each point, that is to say on half of the total components of each point. This results in a much more efficient implementation in terms of running time and computational



```
++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

INIT PHASE OF K-MEANS++:

DONE.

TIME TO INIT:
510.2239751815796
++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
426.1143534183502
```

Figure 1: Running time for k-means only with k=50, n=1000, d=k, stdev=1/k



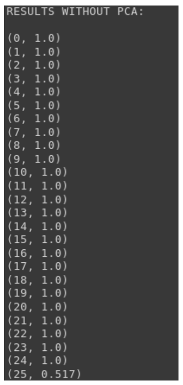
```
++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:
0.6376817226409912
INIT PHASE OF K-MEANS++:

TIME TO INIT:
437.58330750465393
DONE.

++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
37.27700901031494
```

Figure 2: Running time for PCA and then k-means with k=50, n=1000, d=k, stdev=1/k

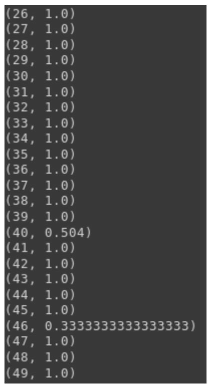
effort, since the PCA algorithm is pretty fast and in less than one second gives us the reduction. Are the results good enough?

In the case of k-means only (pictured in **Figure 3**, we can say that apart from a small numbers of ground clusters, all the other ground clusters (the majority) were retrieved correctly. But this is a behavior which is improved by the PCA algorithm.

As we can see from the results picutred in **Figure 4**, in this case when

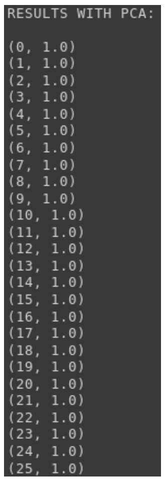


```
RESULTS WITHOUT PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)
(10, 1.0)
(11, 1.0)
(12, 1.0)
(13, 1.0)
(14, 1.0)
(15, 1.0)
(16, 1.0)
(17, 1.0)
(18, 1.0)
(19, 1.0)
(20, 1.0)
(21, 1.0)
(22, 1.0)
(23, 1.0)
(24, 1.0)
(25, 0.517)
```

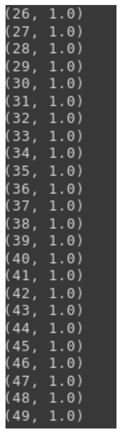


```
(26, 1.0)
(27, 1.0)
(28, 1.0)
(29, 1.0)
(30, 1.0)
(31, 1.0)
(32, 1.0)
(33, 1.0)
(34, 1.0)
(35, 1.0)
(36, 1.0)
(37, 1.0)
(38, 1.0)
(39, 1.0)
(40, 0.504)
(41, 1.0)
(42, 1.0)
(43, 1.0)
(44, 1.0)
(45, 1.0)
(46, 0.3333333333333333)
(47, 1.0)
(48, 1.0)
(49, 1.0)
```

Figure 3: Results for k-means only with k=50, n=1000, d=k, stdev=1/k



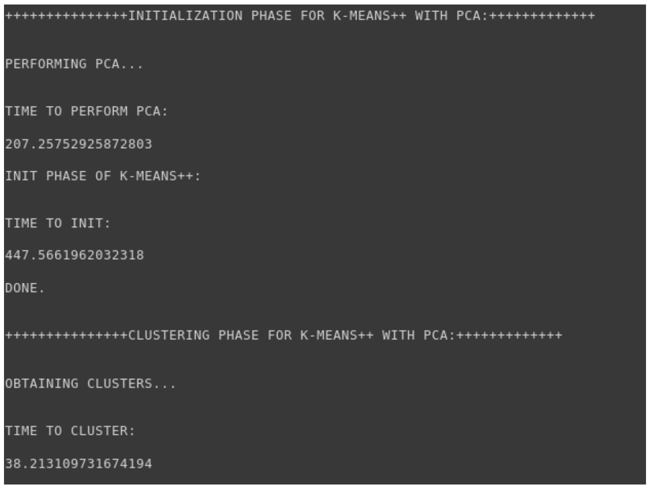
```
RESULTS WITH PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)
(10, 1.0)
(11, 1.0)
(12, 1.0)
(13, 1.0)
(14, 1.0)
(15, 1.0)
(16, 1.0)
(17, 1.0)
(18, 1.0)
(19, 1.0)
(20, 1.0)
(21, 1.0)
(22, 1.0)
(23, 1.0)
(24, 1.0)
(25, 1.0)
```



```
(26, 1.0)
(27, 1.0)
(28, 1.0)
(29, 1.0)
(30, 1.0)
(31, 1.0)
(32, 1.0)
(33, 1.0)
(34, 1.0)
(35, 1.0)
(36, 1.0)
(37, 1.0)
(38, 1.0)
(39, 1.0)
(40, 1.0)
(41, 1.0)
(42, 1.0)
(43, 1.0)
(44, 1.0)
(45, 1.0)
(46, 1.0)
(47, 1.0)
(48, 1.0)
(49, 1.0)
```

Figure 4: Results for PCA and k-means with k=50, n=1000, d=k, stdev=1/k

applying PCA and then k-means, **all** the ground clusters were retrieved correctly. This is due to the fact that PCA highlights only the principal components, that are the ones which determine the clustering given the cost function of the k-means algorithm, and among those principal components there is for sure the one that we have highlighted in the PCA algorithm by dividing it by the standard deviation of the noise, that is the value in the identity matrix which actually determines the clustering: for this value of standard deviation, we find out that the value in the identity matrix determining the clustering is such that it has a variance with respect to the noise that makes PCA select it as a principal component, so the k-means is carried out on this set of reduced points and there is a gain in the whole accuracy of the process.
We can confirm this by running the algorithms with the same input parameters, but changing the dimension of the points by setting **d = 100*k**. This will result in the points having much more noise, and should make it harder for k-means to recover the original ground clustering: indeed, even if we cannot make any assumption on the results of the k-means only implementation, I noticed the running time was so high that it was actually pointless to let it finish, so I only collected the results for the PCA and k-means implementation. The running time is pictured in **Figure 5**.



```
++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:

207.25752925872803

INIT PHASE OF K-MEANS++:

TIME TO INIT:

447.5661962032318

DONE.

++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:

38.213109731674194
```

Figure 5: Running time for PCA and then k-means with k=50, n=1000, d=100*k, stdev=1/k

As we can see from **Figure 5**, this running time is not much different from the one we obtained before with PCA and k-means with **d = k** - indeed, the running time for the initialization of the **k-means++** algorithm and the execution of **k-means** to cluster is practically identical to before, and this is due to the fact that, by carrying out a dimensionality reduction with PCA, we are basically still

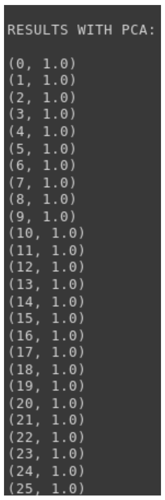
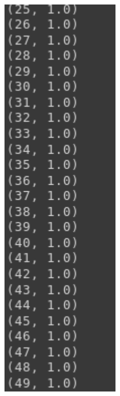
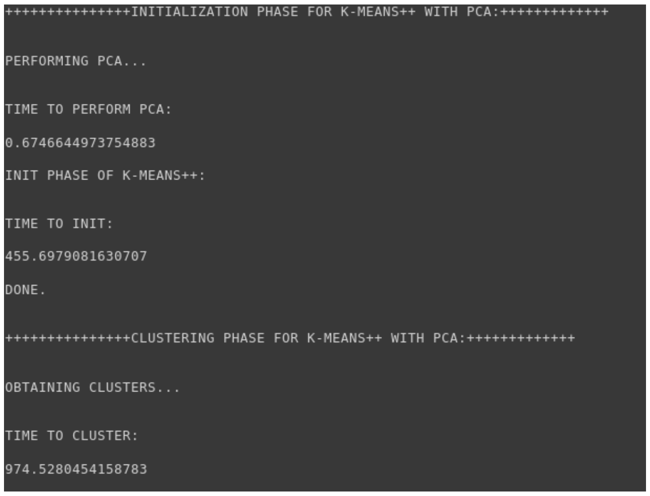

```
RESULTS WITH PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)
(10, 1.0)
(11, 1.0)
(12, 1.0)
(13, 1.0)
(14, 1.0)
(15, 1.0)
(16, 1.0)
(17, 1.0)
(18, 1.0)
(19, 1.0)
(20, 1.0)
(21, 1.0)
(22, 1.0)
(23, 1.0)
(24, 1.0)
(25, 1.0)
```

```
(25, 1.0)
(26, 1.0)
(27, 1.0)
(28, 1.0)
(29, 1.0)
(30, 1.0)
(31, 1.0)
(32, 1.0)
(33, 1.0)
(34, 1.0)
(35, 1.0)
(36, 1.0)
(37, 1.0)
(38, 1.0)
(39, 1.0)
(40, 1.0)
(41, 1.0)
(42, 1.0)
(43, 1.0)
(44, 1.0)
(45, 1.0)
(46, 1.0)
(47, 1.0)
(48, 1.0)
(49, 1.0)
```

Figure 6: Results for PCA and k-means with k=50, n=1000, d=100*k, stdev=1/k

working as before, where we had **d = k**. Obviously the **PCA** takes much more time to be performed since the points to be reduced have a higher dimensionality, but apart from this, the running time is still the same for the clustering and last but not least this implementation is much more efficient than k-means only, where the running time is much more higher.
Pictured in **Figure 6** are the results about the ground clustering.

As we can see, the ground clusters are once again completely retrieved. This confirms the hypothesis that by running PCA, not only we are carrying out a dimensionality reduction so that we gain in efficiency, but also that no matter how much noise we add, we are still able to retrieve the ground clustering because the principal components we care the most are highlighted - and, especially, the one determining the cluster in the identity matrix.
For **d=100*k**^{2} the analysis couldn't be carried out, due to the fact that the dimensionality is too high and the **colab** notebook, notwithstanding its **25GB** of RAM, crashed because it run out of RAM *when building the dataset.* This is a hardware limit that couldn't be overcome, but for some tests that I run on lower input values, we can see that by increasing the dimensionality this much, also PCA has quite some problems in recovering the ground clustering and in terms of running time (see the end of this paragraph).
We could see what happens when we apply the initial configuration, but we change the value of the variable of the standard deviation as suggested in the text. For those values, we are actually increasing the standard deviation, for example if we set $\mathbf{stdev} = \mathbf{1}/\sqrt{k}$, meaning we are actually increasing the variance of the Gaussian we are exploiting to retrieve the noise. This means that now the points of the noise that will be added as components to each of the point in the

dataset, will possibly have a much higher value than before: we are telling the random Gaussian function to select those components on a much wider range, so what we should expect is that PCA will find lots of principal components among the noise and will possibly ignore the point in the identity matrix which determines the cluster of each point in some cases, resulting in a much more difficult analysis and then clustering process. I did not run the code for the k-means only implementation because, once again, this implementation took forever to finish even on codelab, due to the fact that by setting the standard deviation to higher values the noise is somehow ”stronger”, and we result in a lack of accuracy. Pictured in **Figure 7** is the running time for PCA and k-means implementation with **k=50, n=1000, d=k, stdev=1/**$\sqrt{k}$.

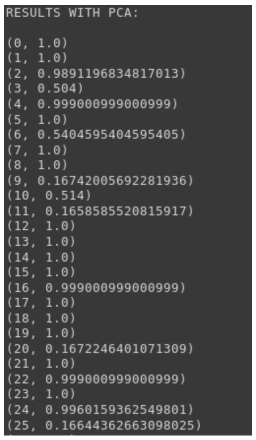


```
++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:
0.6746644973754883
INIT PHASE OF K-MEANS++:

TIME TO INIT:
455.6979081630707
DONE.

++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
974.5280454158783
```

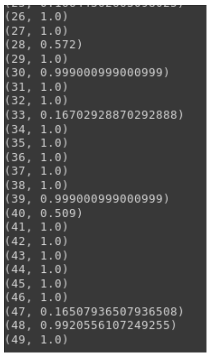
Figure 7: Running time for PCA and then k-means with k=50, n=1000, d=k, stdev=1/$\sqrt{k}$.

As we can see from the picture, the running time is very bad even with PCA, simply because the noise got stronger and now the points, when running the k-means algorithm, have a configuration that, notwithstanding the PCA that was carried out before, is still heavily biased byt the noise of the Gaussian itself. Pictured in **Figure 8** are the results obtained with the comparison with the ground clusters.

As we can see from the picture, the results are affected by the growing weight of the noise. If we wish to improve these results, we should either highlight the point which determines the cluster more - but this would bias the whole clustering process, I wouldn’t do this, and would feel like cheating - or we could, instead, set the **m** principal components that are retrieved by the PCA algorithm to a higher value, so that we could retrieve that point which determines the cluster



```
RESULTS WITH PCA:

(0, 1.0)
(1, 1.0)
(2, 0.9891196834817013)
(3, 0.504)
(4, 0.999000999000999)
(5, 1.0)
(6, 0.5404595404595405)
(7, 1.0)
(8, 1.0)
(9, 0.16742005692281936)
(10, 0.514)
(11, 0.1658585520815917)
(12, 1.0)
(13, 1.0)
(14, 1.0)
(15, 1.0)
(16, 0.999000999000999)
(17, 1.0)
(18, 1.0)
(19, 1.0)
(20, 0.1672246401071309)
(21, 1.0)
(22, 0.999000999000999)
(23, 1.0)
(24, 0.9960159362549801)
(25, 0.16644362663098025)
```



```
(26, 1.0)
(27, 1.0)
(28, 0.572)
(29, 1.0)
(30, 0.999000999000999)
(31, 1.0)
(32, 1.0)
(33, 0.16702928870292888)
(34, 1.0)
(35, 1.0)
(36, 1.0)
(37, 1.0)
(38, 1.0)
(39, 0.999000999000999)
(40, 0.509)
(41, 1.0)
(42, 1.0)
(43, 1.0)
(44, 1.0)
(45, 1.0)
(46, 1.0)
(47, 0.16507936507936508)
(48, 0.9920556107249255)
(49, 1.0)
```

Figure 8: Results for PCA and k-means with k=50, n=1000, d=k, stdev=$1/\sqrt{k}$.

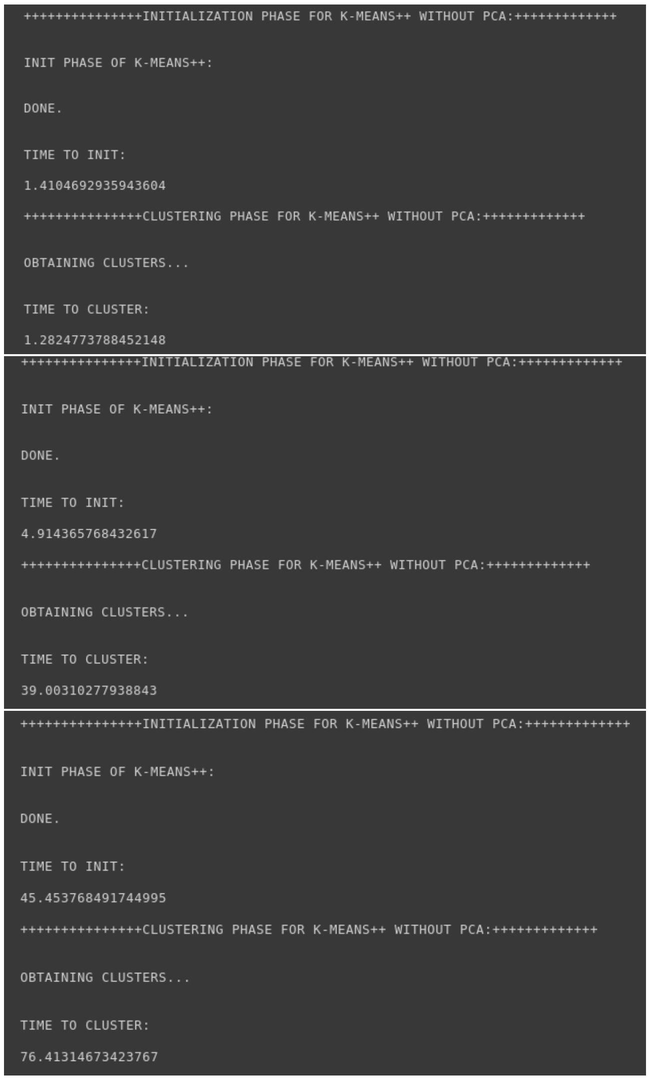
in the identity matrix that is currently ”overwhelmed” by the noise by setting m = $\frac{k}{\epsilon}$. This, unfortunately, would also mean to recover more components of noise for each point, and we would benefit less from the efficiency improvement given by the PCA algorithm because we would reduce less the dimensionality of the points, so here we should find an accurate trade-off between efficiency and results. One thing we could try is to set **k = 100** which would mean to give a higher weight to th identity matrix components in the dataset and also make PCA able to retrieve 100 principal components instead of only 50.
Obviously, by setting **k** to higher values, we'll have to exclude once again the k-means only implementation, because the running time of k-means without PCA applied would once again be too high and would make no sense to wait for it to reach the termination condition.
Something that I didn't change in the initial input parameters was **n**, the number of points, since it would result in a running time and computational memory effort that would be too high even for codelab to handle, but if what I understood from this experiment is actually correct, then I would assume that by setting **n** to higher values, the random function retrieving the Gaussian values would probably have a much higher probability of picking more and more ”noise” values that maximize the variance in a similar way of increasing **d**, leading to some bad results also in the PCA and k-means implementation in terms of running time and ground clusters recovered, because it would have more values to pick (which is a reasoning that maybe trusts too much the law of large numbers). Also, the most important point is probably that by increasing **n** what happens is that we have more points as input to k-means, and *the number of points is actually the only parameter, beside the number of clusters* ***k****, that cannot be reduced somehow by PCA*, so the running time in both cases will be much higher.
If we try to set higher values for both **d** and **standard deviation**, we will get very bad results in both cases of k-means only and PCA and then k-means (this

is shown in the results below for lower inputs). This may be due to the fact that we now have more dimensions on each point, so more components from the Gaussian, and each of these components taken in a Gaussian with a much wider variance, meaning we are basically summing the negative effects we discussed above and, for high values of **d** and **standard deviation**, not even PCA is able to help. Once again, I think some trade-off is needed in this case to retrieve the ground clusters.

I run some other tests with lower input values that I am attaching here at the end of this .pdf file and I won't comment, since they basically confirm what was written up to this point. They help visualizing what has been discussed so far. Number of clusters and number of points as input is always the same: **k = 10, n = 400**, then I varied **d**, **stdev** and then both.

2.4 Varying d

Even on a smaller input, it is still visible how by changing the dimensions of the points in the dataset, both the running time and the results will vary from the very beginning when applying k-means only, but we can also clearly see that when we apply PCA before applying k-means, as expected, running time and results are not as affected as in k-means only.

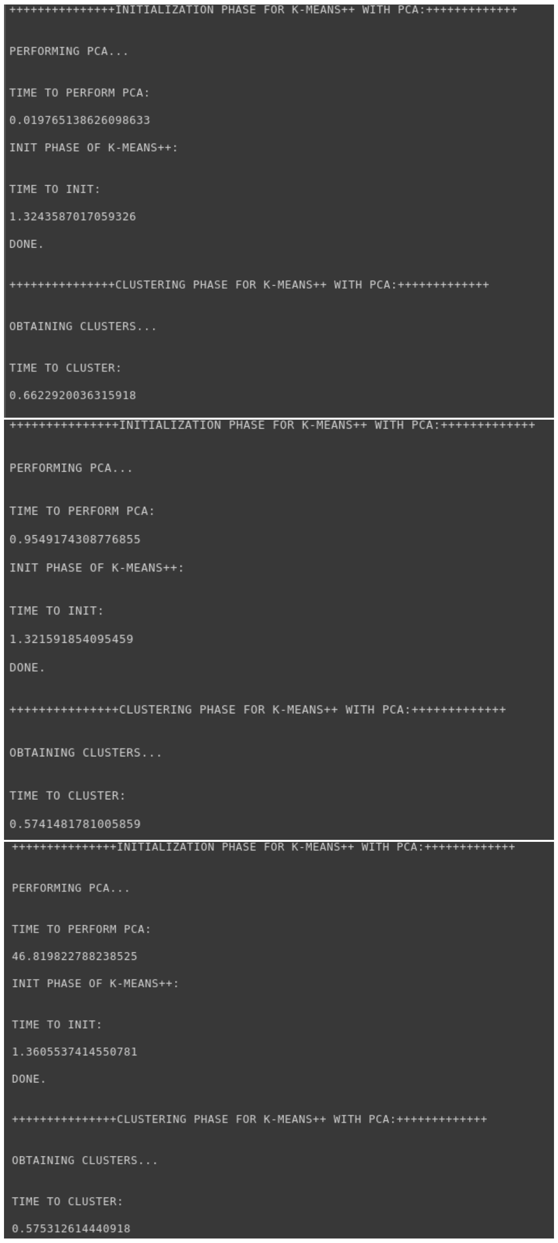


```
+++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

INIT PHASE OF K-MEANS++:

DONE.

TIME TO INIT:

1.4104692935943604

+++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:

1.2824773788452148
```

```
+++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

INIT PHASE OF K-MEANS++:

DONE.

TIME TO INIT:

4.914365768432617

+++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:

39.00310277938843
```

```
+++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

INIT PHASE OF K-MEANS++:

DONE.

TIME TO INIT:

45.453768491744995

+++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:

76.41314673423767
```

Figure 9: From up to down, running time for k-means only with d=k, d=100*k, d=100*k^2.

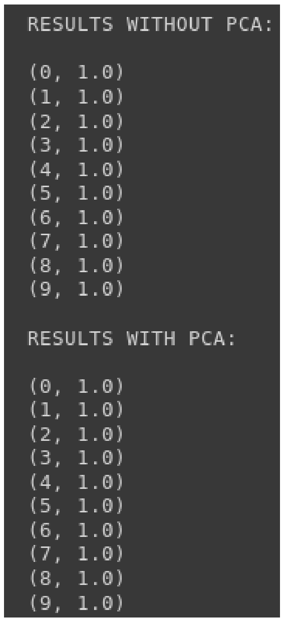


```
++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:
0.019765138626098633
INIT PHASE OF K-MEANS++:

TIME TO INIT:
1.3243587017059326
DONE.

++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
0.6622920036315918
```

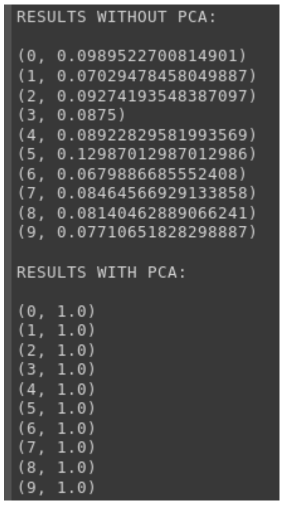
```
++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:
0.9549174308776855
INIT PHASE OF K-MEANS++:

TIME TO INIT:
1.321591854095459
DONE.

++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
0.5741481781005859
```

```
++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:
46.819822788238525
INIT PHASE OF K-MEANS++:

TIME TO INIT:
1.3605537414550781
DONE.

++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
0.575312614440918
```

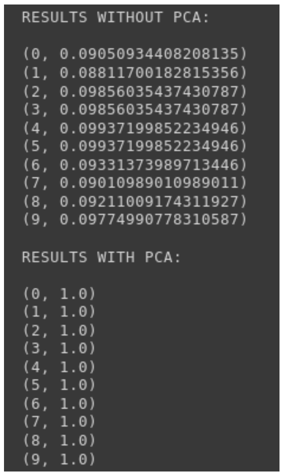
Figure 10: From up to down, running time for PCA and then kmeans with d=k, d=100*k, d=100*k^2. Notice how with PCA running time is resistant wrt to these variations.



```
RESULTS WITHOUT PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)

RESULTS WITH PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)
```



```
RESULTS WITHOUT PCA:

(0, 0.0989522700814901)
(1, 0.07029478458049887)
(2, 0.09274193548387097)
(3, 0.0875)
(4, 0.08922829581993569)
(5, 0.12987012987012986)
(6, 0.0679886685552408)
(7, 0.08464566929133858)
(8, 0.08140462889066241)
(9, 0.07710651828298887)

RESULTS WITH PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)
```

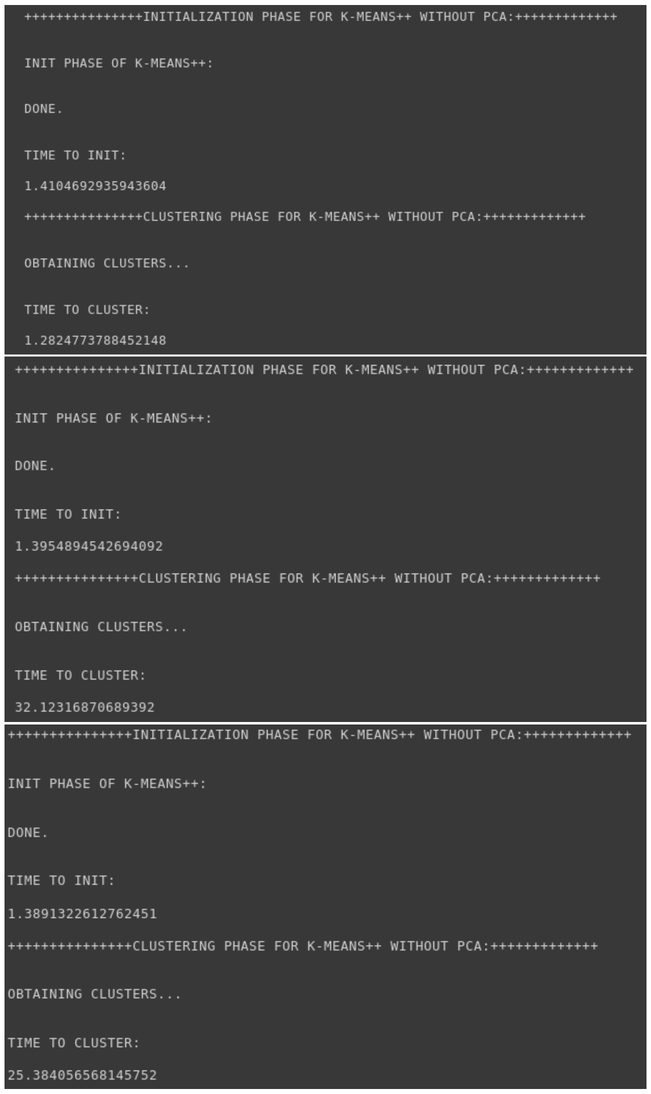


```
RESULTS WITHOUT PCA:

(0, 0.09050934408208135)
(1, 0.08811700182815356)
(2, 0.09856035437430787)
(3, 0.09856035437430787)
(4, 0.09937199852234946)
(5, 0.09937199852234946)
(6, 0.09331373989713446)
(7, 0.09010989010989011)
(8, 0.09211009174311927)
(9, 0.09774990778310587)

RESULTS WITH PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)
```

Figure 11: From left to right, results of retrieved ground clusters with d=k, d=100*k, d=100*k^2. Notice how with PCA, results seem to be resistant too, at least up to this input size.

2.5 Varying standard deviation:

When varying the standard deviation, we are varying the variance of the Gaussian from which we take the values for the noise. This means that in some cases, especially for a very big standard deviation, the components of the noise for every point could ”hide” to PCA the point which clusters the point, and so we could end up with the principal components that are chosen for each point that are just a bunch of ”noise”, resulting in a very bad clustering with respect to the ground one, and the efficiency will be affected too.
By applying k-means without PCA, obviously, we still have very bad results, because the noise is litterally ”too strong” and the components selected very far from the mean in the Gaussian will make the points change clusters.
Images are on the next page (I’m sorry for this very bad page make-up).

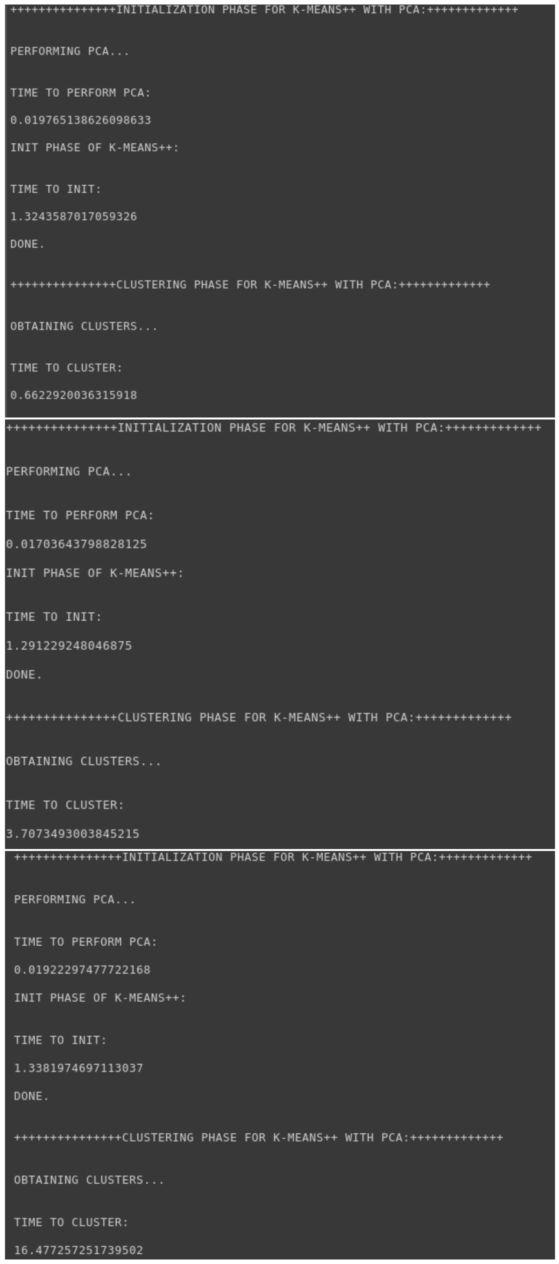


```
++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

INIT PHASE OF K-MEANS++:

DONE.

TIME TO INIT:

1.4104692935943604

++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:

1.2824773788452148
```

```
++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

INIT PHASE OF K-MEANS++:

DONE.

TIME TO INIT:

1.3954894542694092

++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:

32.12316870689392
```

```
++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

INIT PHASE OF K-MEANS++:

DONE.

TIME TO INIT:

1.3891322612762451

++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:

25.384056568145752
```

Figure 12: From left to right, running time for k-means only with stdev=1/k, stdev=$1/\sqrt{k}$, stdev=0.5.

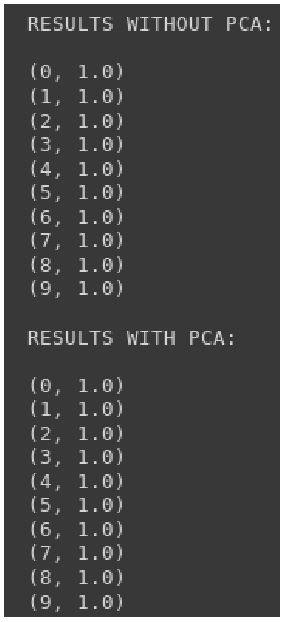


```
+++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:
0.019765138626098633
INIT PHASE OF K-MEANS++:

TIME TO INIT:
1.3243587017059326
DONE.

+++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
0.6622920036315918
```

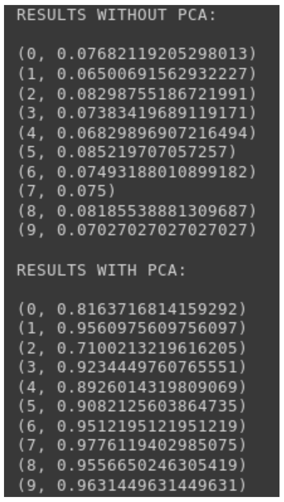
```
+++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:
0.01703643798828125
INIT PHASE OF K-MEANS++:

TIME TO INIT:
1.291229248046875
DONE.

+++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
3.7073493003845215
```

```
+++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:
0.01922297477722168
INIT PHASE OF K-MEANS++:

TIME TO INIT:
1.3381974697113037
DONE.

+++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
16.477257251739502
```

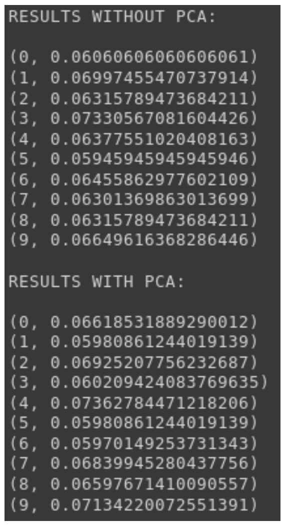
Figure 13: From left to right, running time for PCA and then k-means with stdev=1/k, stdev=$1/\sqrt{k}$, stdev=0.5.



```
RESULTS WITHOUT PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)

RESULTS WITH PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)
```



```
RESULTS WITHOUT PCA:

(0, 0.07682119205298013)
(1, 0.06500691562932227)
(2, 0.08298755186721991)
(3, 0.07383419689119171)
(4, 0.06829896907216494)
(5, 0.085219707057257)
(6, 0.07493188010899182)
(7, 0.075)
(8, 0.08185538881309687)
(9, 0.07027027027027027)

RESULTS WITH PCA:

(0, 0.8163716814159292)
(1, 0.9560975609756097)
(2, 0.7100213219616205)
(3, 0.9234449760765551)
(4, 0.892601431980906)
(5, 0.9082125603864735)
(6, 0.9512195121951219)
(7, 0.9776119402985075)
(8, 0.9556650246305419)
(9, 0.9631449631449631)
```

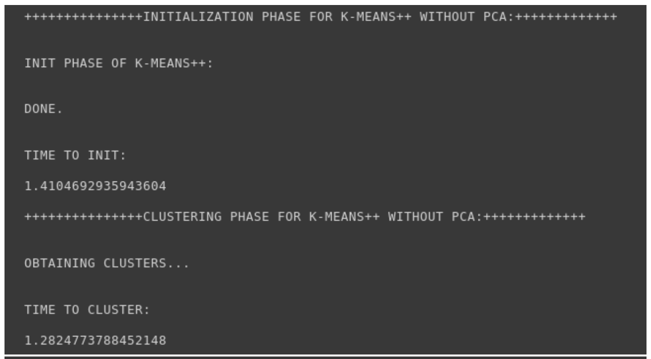


```
RESULTS WITHOUT PCA:

(0, 0.06060606060606061)
(1, 0.06997455470737914)
(2, 0.06315789473684211)
(3, 0.07330567081604426)
(4, 0.06377551020408163)
(5, 0.05945945945945946)
(6, 0.06455862977602109)
(7, 0.06301369863013699)
(8, 0.06315789473684211)
(9, 0.06649616368286446)

RESULTS WITH PCA:

(0, 0.06618531889290012)
(1, 0.05980861244019139)
(2, 0.06925207756232687)
(3, 0.060209424083769635)
(4, 0.07362784471218206)
(5, 0.05980861244019139)
(6, 0.05970149253731343)
(7, 0.06839945280437756)
(8, 0.06597671410090557)
(9, 0.07134220072551391)
```

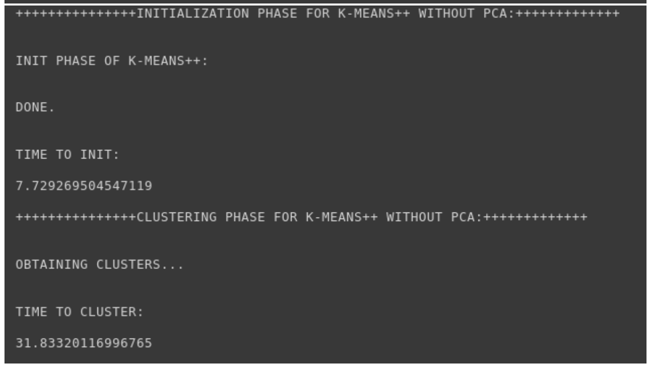
Figure 14: From left to right, results in ground clusters retrieved with stdev=1/k, stdev=$1/\sqrt{k}$, stdev=0.5.

2.6 Varying both d and stdev

This is an example of how the combined effect of a very high standard deviation and very high value of dimensions can give very bad results, even applying PCA before k-means.
In this way, the noise is just too strong, we cannot recover the right information from the clusters.

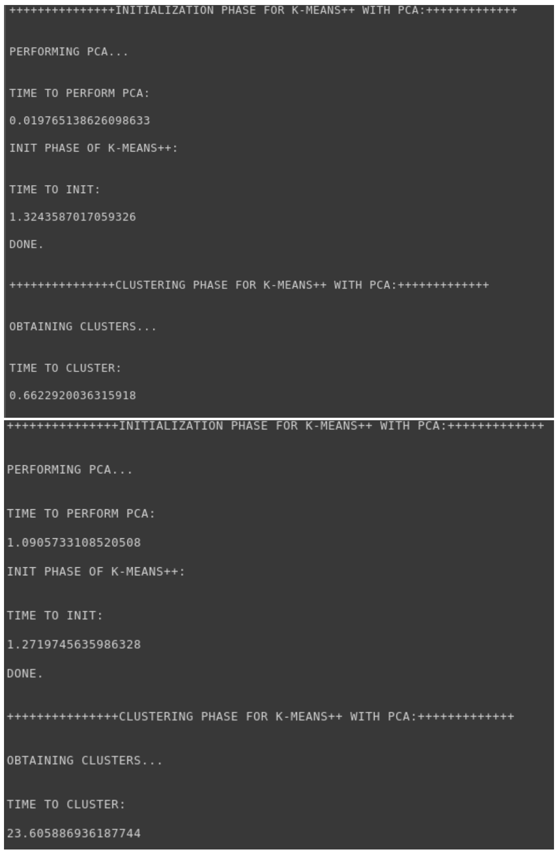


```
 +++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

 INIT PHASE OF K-MEANS++:

 DONE.

 TIME TO INIT:

 1.4104692935943604

 +++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

 OBTAINING CLUSTERS...

 TIME TO CLUSTER:

 1.2824773788452148
```



```
+++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

INIT PHASE OF K-MEANS++:

DONE.

TIME TO INIT:

7.729269504547119

+++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITHOUT PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:

31.83320116996765
```

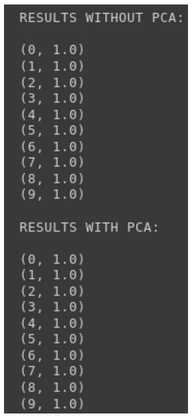
Figure 15: From up to down, running time for k-means only with d=k, stdev=1/k, then d= 100*k, stdev=$1/\sqrt{k}$.



```
+++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:
0.019765138626098633
INIT PHASE OF K-MEANS++:

TIME TO INIT:
1.3243587017059326
DONE.

+++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
0.6622920036315918
```

```
+++++++++++++++INITIALIZATION PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

PERFORMING PCA...

TIME TO PERFORM PCA:
1.0905733108520508
INIT PHASE OF K-MEANS++:

TIME TO INIT:
1.2719745635986328
DONE.

+++++++++++++++CLUSTERING PHASE FOR K-MEANS++ WITH PCA:+++++++++++++

OBTAINING CLUSTERS...

TIME TO CLUSTER:
23.605886936187744
```

Figure 16: From up to down, running time for PCA and the k-means with d=k, stdev=1/k, then d= 100*k, stdev=$1/\sqrt{k}$.


```
RESULTS WITHOUT PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)

RESULTS WITH PCA:

(0, 1.0)
(1, 1.0)
(2, 1.0)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)
```

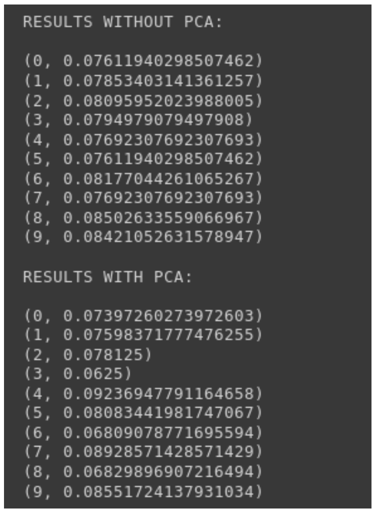

```
RESULTS WITHOUT PCA:

(0, 0.07611940298507462)
(1, 0.07853403141361257)
(2, 0.08095952023988005)
(3, 0.0794979079497908)
(4, 0.07692307692307693)
(5, 0.07611940298507462)
(6, 0.08177044261065267)
(7, 0.07692307692307693)
(8, 0.08502633559066967)
(9, 0.08421052631578947)

RESULTS WITH PCA:

(0, 0.07397260273972603)
(1, 0.07598371777476255)
(2, 0.078125)
(3, 0.0625)
(4, 0.09236947791164658)
(5, 0.08083441981747067)
(6, 0.06809078771695594)
(7, 0.08928571428571429)
(8, 0.06829896907216494)
(9, 0.08551724137931034)
```

Figure 17: From left to right, results in ground clusters retrieved with d=k, stdev=1/k, then d= 100*k, stdev=$1/\sqrt{k}$.

References

[Wikipedia - K-medians clustering] https://en.wikipedia.org/wiki/K-medians_clustering

[CSE 291: Unsupervised Learning - Lecture 3: The k-medoid clustering problem] proof inspired to this lesson: https://cseweb.ucsd.edu/∼dasgupta/291-unsup/lec3.pdf