PA 3

Project Report
SS 2024

Group 4

Friedrich Alexander (11729489)
Wolf Christoph (11716756)

Contents

1 Implementations

1.1 Lloyd's Algorithm

The implementation of Lloyd's algorithm found in the **LloydsAlgorithm.py** file, follows the standard algorithm for K-Means.
Upon initialization of the algorithm, the initial centroids are either chosen to be the first k points in the data, or randomly chosen from the data points, depending on the value of **random_init** within the **main.py** file. Everything else happens in the **fit** method, which iterates at most **max_iter** times, or until the centroids do not change anymore.

Each iteration starts by assigning every data point to a cluster and check for convergence. If the algorithm has not converged, the centroids are then updated, handled by the **_update_centroids** method. Additionally current loss is computed and additional metrics stored. The loss is defined as the average squared intra cluster distance to the assigned centroid. After convergence, or after the maximum number of iterations is reached, NMI and the time spend training are computed and stored.

1.2 Locality Sensitive Hashing

The LloydsAlgorithmLSH class is a version of the normal k-means clustering algorithm that uses Locality-Sensitive Hashing (LSH). This class extends our normal Lloyd's algorithm by integrating LSH to reduce the number of distance calculations between points and centroids.

The implementation of LSH for Kmeans, found in the **LSHLloyd.py** file, is based on the description provided in the pdf on Moodle:

Upon initialization, the class generates the required hash functions and hash tables, and hashes the data points into these tables, grouping similar points together in hash buckets.
The **fit** method in our class iteratively updates the centroids and reassigns points to clusters until convergence or the maximum number of iterations is reached. This process includes hashing the centroids, assigning points to clusters based on hash buckets, handling unassigned points by direct distance calculation, and updating centroids based on the new assignments.

We created and organized hash functions as follows:

- A vector **a** generated from a normal distribution (np.random.normal), which matches the dimensionality of the data.

- A scalar **b** drawn from a uniform distribution (np.random.uniform), within the range of the bucket size.

Each hash function uses the formula

$$\textbf{hash\_value} = \lfloor \frac{(\text{point} \cdot a + b)}{\text{bucket\_size}} \rfloor$$

, where the dot product between the data point and the vector a is computed, shifted by b, and divided by the bucket size before applying the floor function.

Multiple hash functions are grouped into hash tables. The number of hash tables (num_hash_tables) and the number of hash functions per table (num_hashes_per_table) are configurable parameters.

When a data point is hashed, it is processed by each hash function in a table, resulting in a tuple of hash values (one per hashing table). This tuple serves as the key for the hash bucket in a table. If a data point hashes to the same bucket as one of the centroids in at least one hashing table, this is considered a match and the data point is assigned to that centroid. In the case that it should match with multiple centroids, only the first match is considered.

1.3 Coresets

The implementation of coresets for Kmeans, found in the **coresets.py** file, is based on the paper provided in the assignment.

Upon initialization the coreset and the corresponding weights are computed. This is handled within the **_build_coreset** method that samples points from the data set, based on algorithm 1 from the paper. Everything else is then handled by the sklearn implementation of K-Means, that fits a clustering model to the coreset, which is then used to predict all points in the data.

1.4 Convergence

The convergence in this implementation of Lloyd's Algorithm for k-means clustering is determined by two main factors:

- the cluster assignments do not change between iterations,
- or the centroids' relative change falls below a specified tolerance (tol).

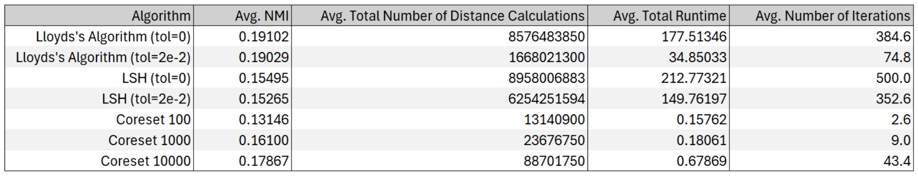
Cluster Assignment: In each iteration, data points are assigned to the nearest centroids. If the cluster assignments remain the same as in the previous iteration, the algorithm considers itself converged.

Centroid Update: The convergence is also checked by the relative change in centroids' positions, which, if below the tolerance, indicates convergence.

The algorithm stops iterating once either of these convergence criteria is met or the maximum number of iterations is reached. The algorithms are run twice, once with tolerance 0, i.e. only strict convergence, and once with tolerance 2e-2 within the **main.py** file. We decided to include this secondary convergence criterium, as we noticed very little change in loss and assignments over later iterations.

2 Results

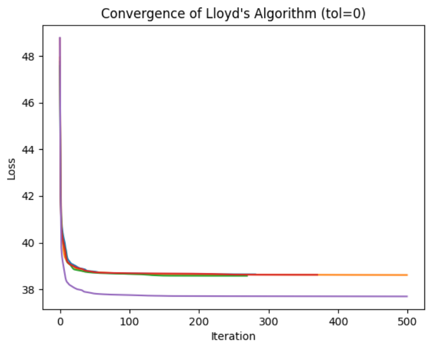
2.1 Comparison of Algorithms



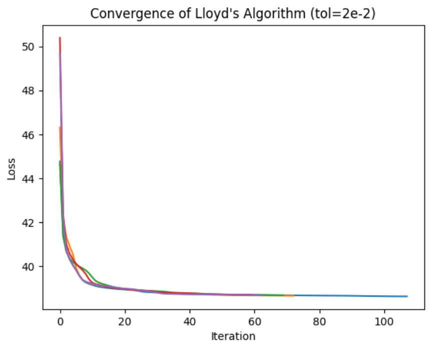
Algorithm	Avg. NMI	Avg. Total Number of Distance Calculations	Avg. Total Runtime	Avg. Number of Iterations
Lloyds's Algorithm (tol=0)	0.19102	8576483850	177.51346	384.6
Lloyds's Algorithm (tol=2e-2)	0.19029	1668021300	34.85033	74.8
LSH (tol=0)	0.15495	8958006883	212.77321	500.0
LSH (tol=2e-2)	0.15265	6254251594	149.76197	352.6
Coreset 100	0.13146	13140900	0.15762	2.6
Coreset 1000	0.16100	23676750	0.18061	9.0
Coreset 10000	0.17867	88701750	0.67869	43.4

Figure 1: Results over all algorithms

Highest accuracy, with regards to NMI, was achieved by the base algorithm, with an average NMI of about 0.191. However, core sets outperform the other implementations in the other three metrics with comparable accuracy for the largest core set size. For locality sensitive hashing the average Normalized Mutual Information (NMI) achieved for the selected hyperparameters (num_hash_tables=5, num_hashes_per_table=5, bucket_size=4.0) is a bit lower at about 0.15. When selecting the hyperparameters, we encountered a trade-off between runtime and accuracy. The more points we could assign during the hashing part, the fewer distance calculations were necessary, which reduced the runtime. However, this often came at the cost of lower clustering accuracy. During our grid search, we aimed to find a good balance between these competing factors, settling on hyperparameters that provided a reasonable compromise.



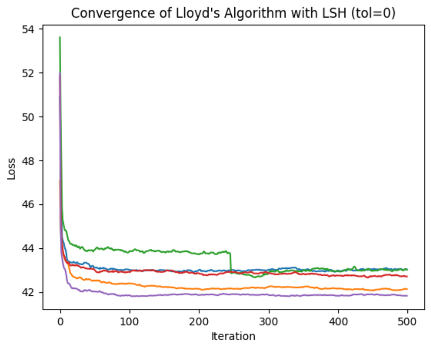
(a) tol=0



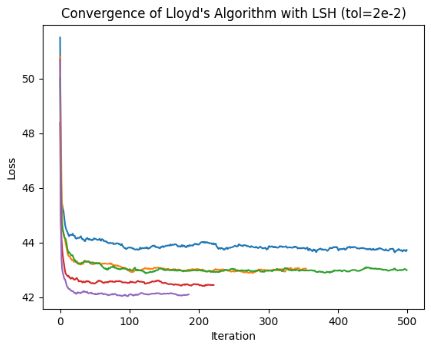
(b) tol=2e-2

Figure 2: Convergence plots for Lloyd's Algorithm

In Figure 2, we see the five respective runs used for deriving the average values for the baseline Lloyds Algorithm in table one. The initial k centroids were selected at random. As we can see, the algorithm converged consistently to a similar loss level, and applying a tolerance for the secondary convergence criterium greatly reduces the number of iterations necessary for convergence. After about 50 iterations, it reaches a point where only very few points switch clusters, but it still takes many more iterations until there are no more cluster reassignments under the initial convergence criterium.



(a) tol=0



(b) tol=2e-2

Figure 3: Convergence plots for Lloyd's Algorithm with LSH

In figure 3 we see the five respective runs for our LSH implementation, again with random initial k centroids. Our algorithm for LSH did not converge before reaching the maximum number of iterations (500) when tol was set to 0. This was because in each iteration, some points changed their buckets during the hashing assignment, preventing the clusters from stabilizing completely. With our selected hyperparameters (num_hash_tables=5, num_hashes_per_table=5, bucket_size=4.0), on average between 20 and 30 thousand datapoints per iteration were assigned to a cluster via hashing. This reduced the needed distance calculations per iteration by about 4.5 million, and the time per iteration from about 0.46 to 0.42 seconds. The reason why the total number of distance calculations and total runtime for LSH is still higher than for the baseline algorithm is, as hinted above, the missing (or slower) convergence of the LSH algorithm. That situation might change when working with larger datasets, as the relative advantage of reducing the required distance calculations through hashing could become more pronounced, potentially leading to a significant speed-up, provided the convergence issues are addressed.

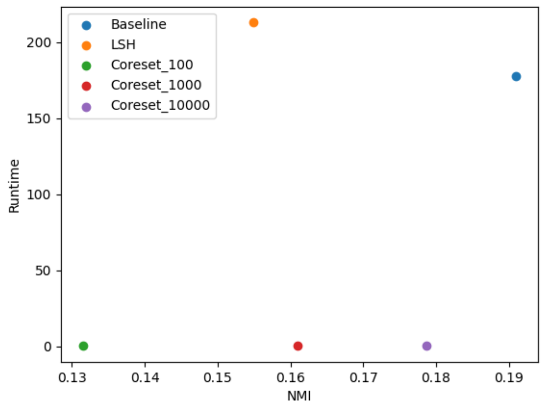


Figure 4: Normalized Mutual Information Score vs. run time

Figure 4 shows the comparison of NMI vs. Runtime for the different algorithms and implementations, considering only the initial convergence criterium, i.e. tolerance is set to 0. Here it is easy to see, that while Baseline algorithm achieves the best results, the core set approach is magnitudes faster, while achieving similar NMI score for large enough core set. LSH could not compete with the other implementations in this scenario.

2.2 Variance of NMI for different core set sizes

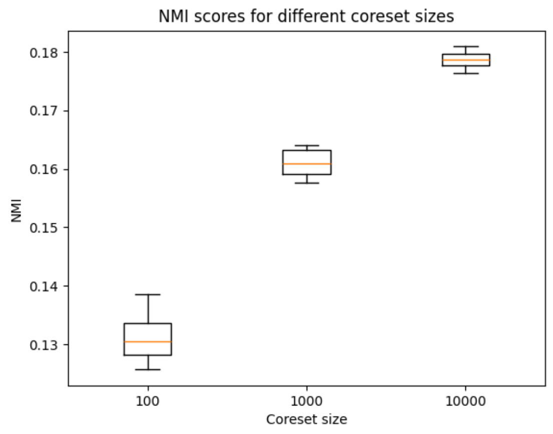


Figure 5: Variance of NMI score over different core set sizes

The variance of the accuracy, displayed in figure 5, is computed by running the algorithm multiple times and computing the variance of the recorded NMI scores over all runs. This is then represented using boxplots. In these we can see that the mean of the NMI scores increases with the size of the coreset for our choice of sizes, while the variance decreases. This is to be expected, as a larger coreset should be a better representation of the underlying distribution within the data, and thus yield smaller variance through sampling.