Advanced Data Mining Project

Team Members:

Adiesha Liyanage, Kaveen Liyanage, Siddat Nesar

Introduction and Background

The purpose of our project is to implement the state of the art DBSCAN++ algorithm on python. The algorithm was introduced in **DBSCAN++: Towards fast and scalable density clustering** [1] by Jennifer Jang and Heinrich Jiang. The motivation of this algorithm is to provide a faster and scalable algorithm than the already existing DBSCAN clustering algorithm. DBSCAN seems to perform poorly when the input data set has a large number of points, mostly because it takes so much memory to build the neighborhood tree in the algorithm. This affects a lot when we have to experiment with the clustering with different epsilon values for our data set. When the epsilon value gets bigger, the DBSCAN algorithm tends to work slowly and consumes a higher chunk of memory.

In the proposed DBSCAN++ algorithm, it only requires to compute densities of a subset of chosen points. The trick is to find the best subset of points that would give a better clustering result. In the paper, they have shown that DBSCAN++ not only performs fast but has added robustness in the bandwidth hyperparameter, which helps the user to tweak the result of clustering. Authors have also shown statistical consistency guarantees showing the trade-off between computational cost and estimation rates.

We are going to implement this algorithm in python. In the paper, they have mentioned that in their calculations, they used KDTree to query the epsilon radii ball in the algorithm. So we will use KDTree library implementation in python to implement this. And they have provided two initialization concepts called uniform initialization and Greedy K-center initialization methods. We will implement both of these initialization methods in our implementation. We recreated the experiments they mentioned in the paper

After implementing the DBSCAN++ algorithm we have implemented the DBSCAN and DBSCAN++ algorithms (both) with an approximate nearest neighbour querying. We used some of the data sets mentioned in the paper and some others to experiment with our implementations to evaluate runtime and performance of the mentioned algorithms. We have used the library called Annoy [3] which is a python wrapper around C++ implementation of an approximate nearest neighbour library. This is extremely fast when compared with a library like KDTree. (A Fast Library for Approximate Nearest Neighbors). When we tested this on a data set with 43500 points with 9 attributes it was able to create the approximate nearest neighbour tree

under 15 seconds, unlike KDTree where we ran it for more than 3 hours and it did not finish. We will explain why we included the ANN on DBSCAN++ as well. This is a change to what mentioned in the paper [1]. They only implemented this in DBSCAN and they tested this against DBSCAN++ with KDTree implementation.

In our project there are three types of the experiment. We compared running times of DBSCAN results against DBSCAN++ results with both uniform and k-center initialization of data sets mentioned in the paper. Both algorithms used KDTree to query the epsilon-radius ball. The second experiment was to compare the performance of the DBSCAN against DBSCAN++ on wide range of hyperparameters. Then the third experiment was to compare the results of DBSCAN with ANN with DBSCAN++ with KDTree queries. This did not result in a good comparison because python KDTree took ages to run on a large data set. Thus we implemented ANN queries on DBSCAN++ as well.

This paper did not mention which language they used to build their implementations. However, we used the python to build these algorithms and compare whether we can recreate their results. We came across a huge problem, in terms of KDTree implementation in python. KDTree is very computationally intensive in terms of creating the KDTree. When we input a large data set it takes ages to build the tree, this is because after 20 levels of the tree KDTree goes into brute force method to build the tree. (This was mentioned in one of the presentations in the class - SciKit Learn DBSCAN Optimization by Michael and Ian). This posed us a big challenge on how to compare the DBSCAN with ANN against DBSCAN++ with KDTree implementation. We even tried increasing this level parameter and increasing the max recursion level parameter in the system, but still it was not allowing us to create the KDTree in a feasible time. This made us wonder how they compared the DBSCAN++ results with DBSCAN with ANN. Time for KDTree build up should be included in the algorithm running time so we couldn't input the built KDTree as an input to the DBSCAN++ as well because it would defeat the purpose. Our understanding was they were probably using a KDtree implementation of another language like C++ which it runs fast. Once KDTree was built the query time was very fast but time gained from running less number of queries could not compensate the time taken to build the KDTree in the case of comparing the DBSAN on ANN with DBSCAN++. Thus, what we did was we tweaked the DBSCAN++ to have ANN queries as well, and we tested this with DBSCAN on ANN results.

As discussed in the proposal discussion we decided to do this project in three steps.

1. Step 1 (Basic). Implementing the DBSCAN++ algorithm and experiment it on high dimensional data sets.
2. Step 2 (intermediate). Try to recreate the results in the paper and see whether this algorithm actually provides the benefit.
3. Step 3 (Advanced). Use fast implementation of the nearest neighbor algorithm in DBSCAN to compare against the DBSCAN++ algorithm.

We did all the three steps and we also tested DBSCAN++ on ANN as well.

Implementation details

DBSCAN++

We used KDTree implementation scipy to query the nearest neighbours. There two main initialization methods proposed in the paper. Uniform and K-Center initialization methods. For the uniform initialization method, we randomly sampled m number of points (m is a user input) and queried the nearest neighbours and added edges to the nearest neighbours in the neighbourhood tree.

Algorithm 2 DBSCAN++

Inputs: X, m, ε, minPts
$S \leftarrow$ sample m points from X.
$C \leftarrow$ all core-points in S w.r.t X, ε, and minPts
$G \leftarrow$ empty graph.
for $c \in C$ **do**
  Add an edge (and possibly a vertex or vertices) in G from c to all points in $X \cap B(c, \varepsilon)$
end for
return connected components of G.

For the K-Center initialization method we used a slightly better algorithm mentioned in the book Geometric approximation algorithms [3]. One of the interesting things about this initialization is that this is another type of clustering algorithm called The Greedy Clustering algorithm, rather than clustering the points to the clusters we are only picking the mid points of these clusters for the initializations. Initially when we implemented this k-center initialization took a considerable amount of time because of O(mn) time complexity. However, we vectorized the calculation to achieve better results. Even with this vectorized implementation of K-center initialization we saw some interesting results in regards to the running time against DBSCAN which contradicted with the paper. We will discuss this in the discussion of the results section.

Algorithm 3 Greedy K-center Initialization

Input: X, m.
$S \leftarrow \{x_1\}$.
for i from 1 to $m-1$ **do**
  Add $\text{argmax}_{x \in X} \min_{s \in S} |x - s|$ to S.
end for
return S.

```python
def kgreedyinitialization(data, k, m, norm=None):
    n = data.shape[0]
    distance = np.full(shape=(n), fill_value=np.inf, dtype=float)
    S = set()
    slicedData = data.iloc[:, 0:k]
    data_numpy_a = slicedData.to_numpy()
    for p in range(0, m):
        index_max = np.argmax(distance)
        S.add(index_max)
        baseTuple = np.array(slicedData.iloc[index_max, 0:k])
        logging.info("p: %d", p)
        distance = vec_returnMin(data_numpy_a, baseTuple, distance, k)
    return S

```

For the approximate nearest neighbour implementation what we did was to use a library called Annoy [4] which wraps a C++ implementation of an approximate nearest neighbour tree. This runs extremely fast for large data sets. Plugging this to DBSCAN was not that difficult. All we had to do was to replace the KDTree querying by this, but there was a problem in regards to epsilon-readius querying. This library did not allow epsilon-radius queries. So what we did was we queried 2*minpts the number of points for each query and we checked, what are the points in the epsilon-radius. Then we checked whether there are more than minpts in the radius to mark a point as a core point. We only added edges to the neighbouring points that are inside the epsilon-readius ball. By querying 2*minpts we were able to minimize the uncertainty of not picking all the points in the epsilon-radius ball. However this added extra O(minpts) time to check the min points, which is very small compared to O(mn) also we vectorized the calculation so that it will run even faster.

```python
# neighbourhood = neighbourhoodtree.get_nns_by_item(i, int(minpts * 1.5), include_distances=True)
# query 1.5 times the points the min points in case we miss some points
neighbourhoodindices, neighbourhooddistnaces = neighbourhoodtree.get_nns_by_item(i, int(minpts * 2.0),
                                                                                 include_distances=True)
querycount += 1

# remove the i object and its distance from the queried result
# neighbourhooddistnaces.remove(neighbourhoodindices.index(i))
del neighbourhooddistnaces[neighbourhoodindices.index(i)]
neighbourhoodindices.remove(i)

# run a lambda to get the boolean array that satisfies the radius condition: find the points that are in the
# given radius
tempmapelementinradius = list(map(lambda x: x <= eps, neighbourhooddistnaces))
elementindicesinradius = np.array(neighbourhoodindices)[tempmapelementinradius]
```

Method signatures:
DBSCAN - DBSCAN(data, k, eps, minpts)
DBSCAN++ - DBSCANPP(data, k, eps, minpts, factor, initialization)
DBSCAN on ANN - DBSCANANN(data, k , eps, minpts)
DBSCAN++ on ANN - DBSCANPPANN(data, k , eps, minpts, factor, initialization)

The parameter “k” is the number of attribute columns in the data set. A beautiful observation about this DBSCAN++ is that when we pass the factor = 1 it should act as DBSCAN algorithm, therefore, we only needed to create one algorithm for both DBSCAN and DBSCAN++ by carefully modifying. We even added some other functionality to plot the neighbourhood graph for our analysis. (It is useful when we need to tweak the parameters)

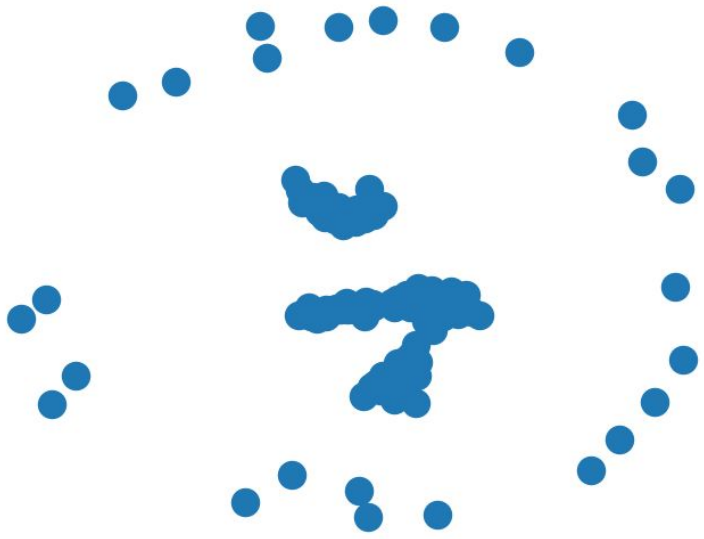
```
def dbscanp(data, k, eps, minpts, factor, initialization=None, plot=False, plotPath='data/result.png', norm=None,
            leafsize=10):
```

DBSCAN and DBSCAN++ method signature

```
def dbscanann(data, k, eps, minpts, factor=1.0, initialization=None, plot=False, plotPath='data/result.png',
              norm='euclidean', ntrees=10, n_jobs=-1):
```

DBSCAN and DBSCAN++ on ANN method signature

Example for plots generated for neighbourhood graphs of .



Data sets used

The data sets mentioned in Table 1 are also used by the authors of the reference paper. These are publicly available data sets. One of the data sets used in this work was not the same as the reference paper. The paper mentions the **spam** data set contains 1401 samples, but the data used for this work contains 4601 samples. All the other properties were kept the same as the reference paper. In table 1, n refers to number of samples, D refers to the number of features, c is the number of clusters, and m is the sample points used for DBSCAN++.

Data set	n	D	c	m
iris	**150**	**4**	**3**	**3**
wine	**178**	**13**	**3**	**5**
spam	**4601**	**57**	**2**	**793**
libras	**360**	**90**	**15**	**84**
mobile	**2000**	**20**	**4**	**112**
zoo	**101**	**16**	**7**	**8**
seeds	**210**	**19**	**7**	**6**
letters	**20000**	**16**	**26**	**551**
shuttle	**43500**	**9**	**7**	**4350**

Table 1

Data set preprocessing

For the experiments we have only used the datasets given in the table1. Since, the original paper has not provided the exact sources for the datasets we have used the only some of the datasets that are publicly available. Also the “spam” dataset we have found has a different number of training points than the one discussed in the original paper. Since we are experimenting with a sufficient number of datasets with various numbers of *n* and *d* values.

Assessment metrics used are the following

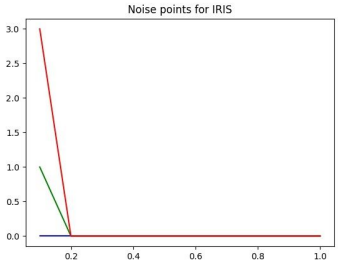
Adjusted Rand index : Rand index[5] is a similarity measurement between the clusters. Adjusted Rand index (ARAND)[6] is the RAND index adjusted for the chance. We have used the sklearn implementation of the ARAND for the experiments[7].

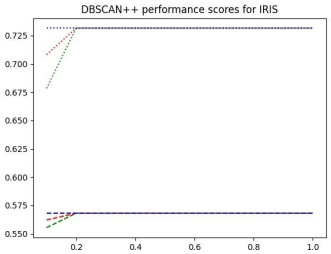
Adjusted Mutual Information Score : Mutual information is the statistical measure between the randomness of two variables sampled simultaneously. To account for the chance, the adjustment of the mutual information is the adjusted mutual information score. It has been implemented from a sklearn module[8].

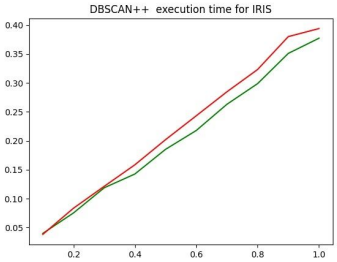
Experiment Results

DBSCAN and DBSCAN++ results for the effect of the factor for the number of outliers are conducted with each of the following epsilon values. Since, the original paper has not specified the exact values of the epsilon used with the experiments we will be choosing the epsilon value which gives fair performances on every method.

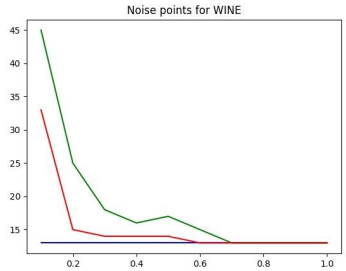
Data set	epsilon
iris	1.25
wine	50
spam	20
libras	1.1
mobile	325
zoo	8
seeds	1

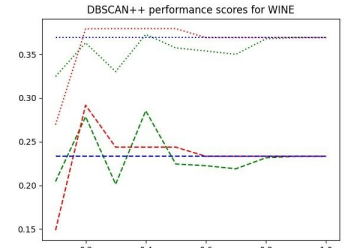


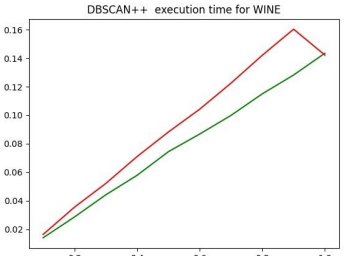




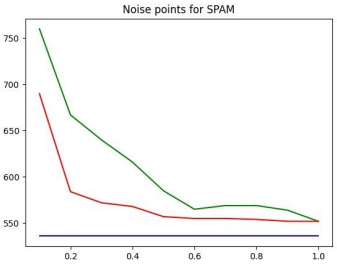
Iris Dataset

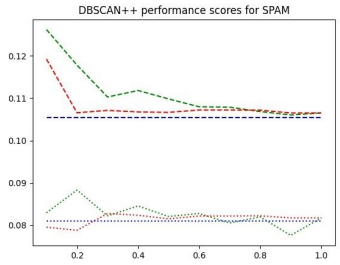


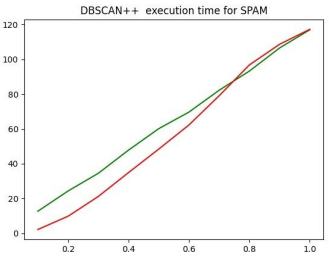




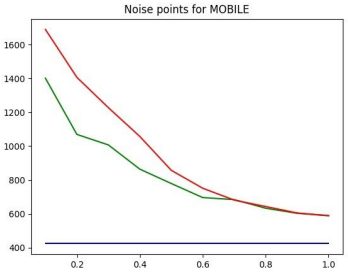
Wine Dataset

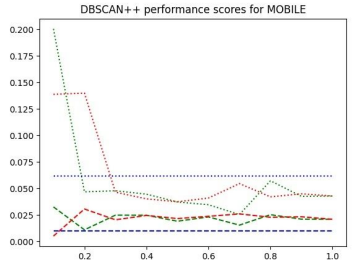


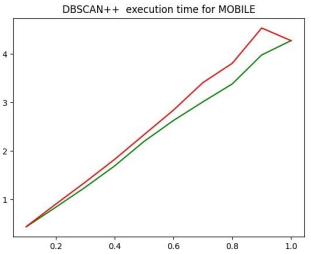




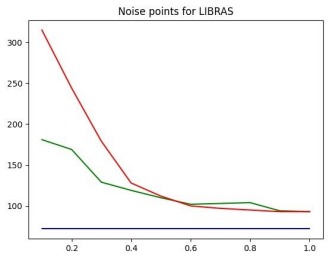
Spam dataset

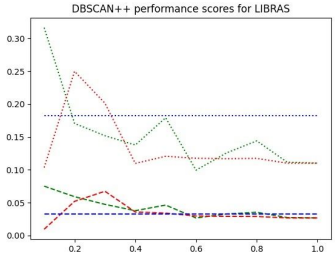


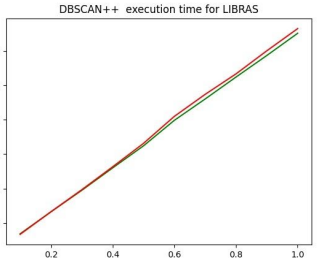




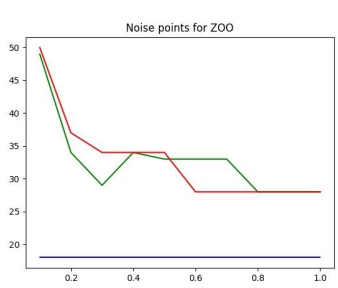
Mobile Dataset

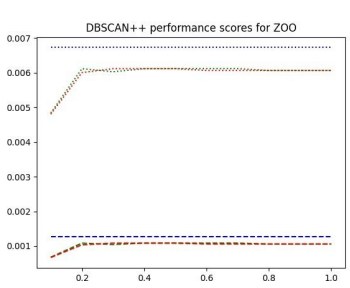


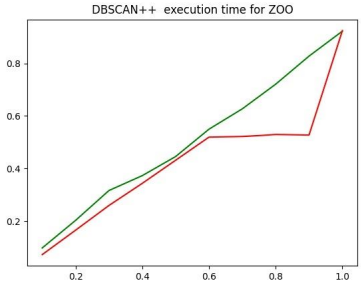




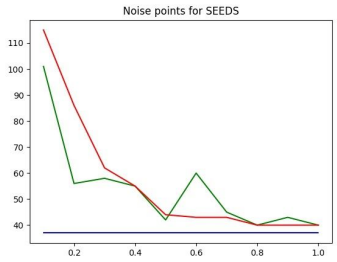
Libras Dataset

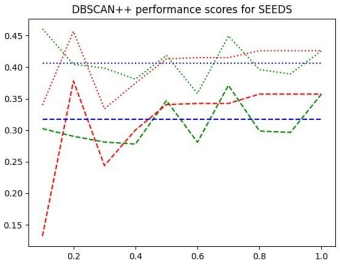


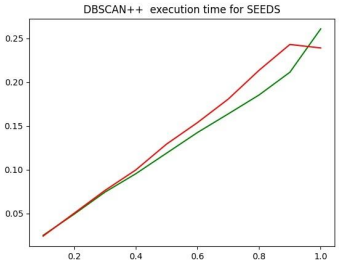




Zoo dataset



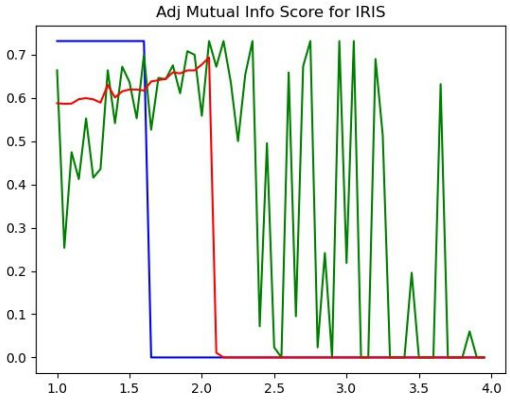


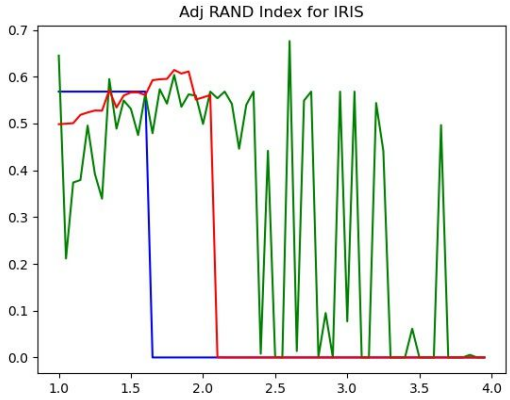


Seeds dataset

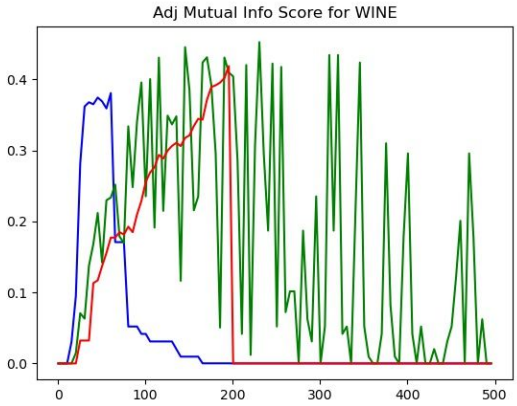
Above shown is the running times of DBSCAN++ algorithms and the outliers recognized. As you can see when the factor (m/n) increases the running time increases. When m/n reaches to 1 DBSCAN++ becomes DBSCAN algorithm. After passing some m/n factor, the number of outliers becomes a constant. This means DBSCAN++ will no longer recognize anymore outliers. There is a reason for this behaviour. We will explain this in the discussion section. When you look at the running time when the m/n factor increases running time increases gradually. For obvious reasons uniform initialization runs faster for most factors.

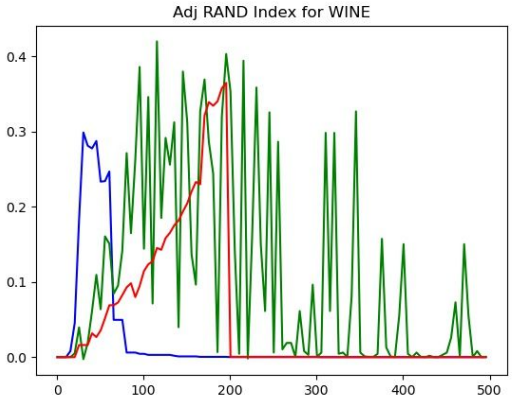
DBSCAN and DBSCAN++ robustness results. Green and Red lines shows the DBSCAN++ Uniform and K-Center results while Blue line shows the results of DBSCAN experiments.



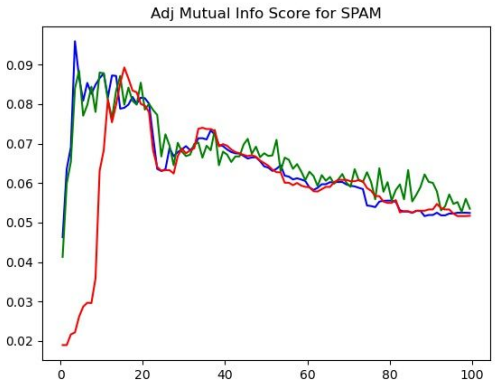


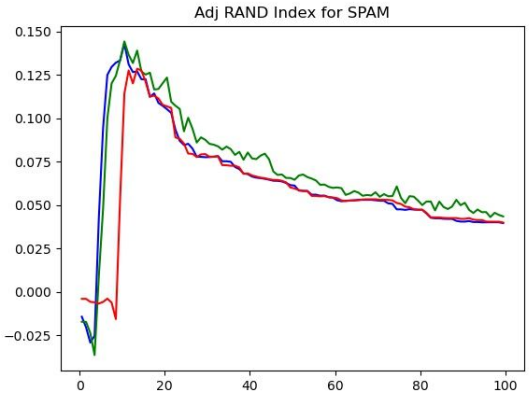
Iris data set



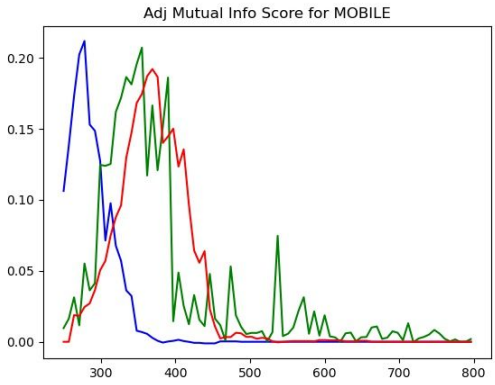


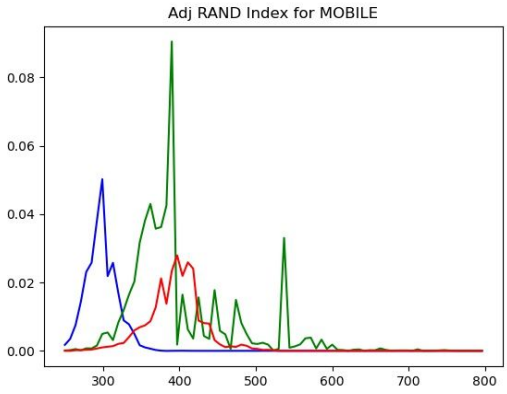
Wine data set



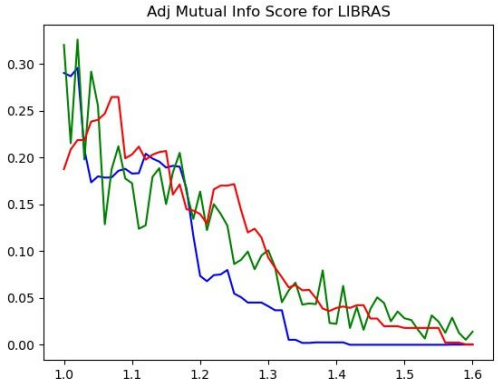


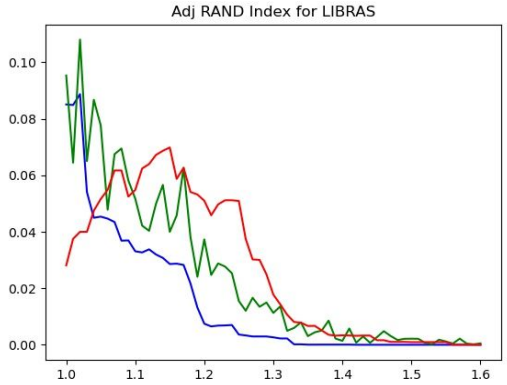
Spam data set



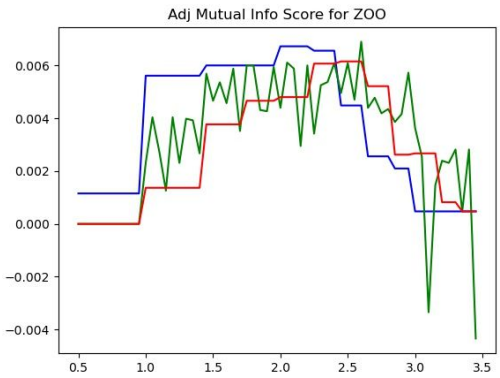


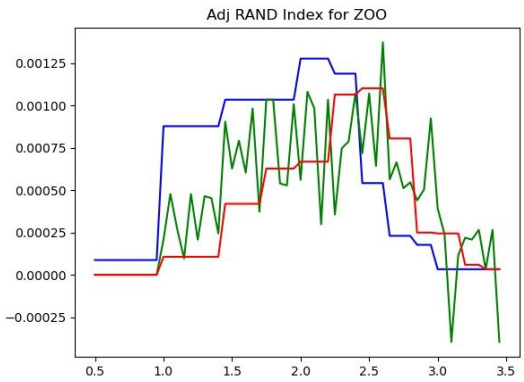
Mobile data sets



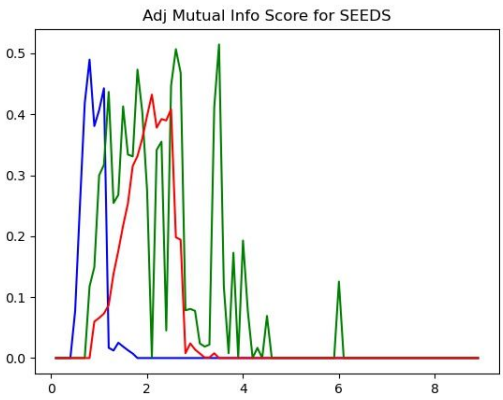


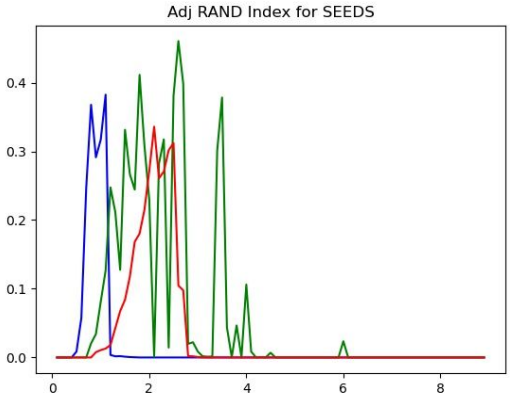
Libras data set



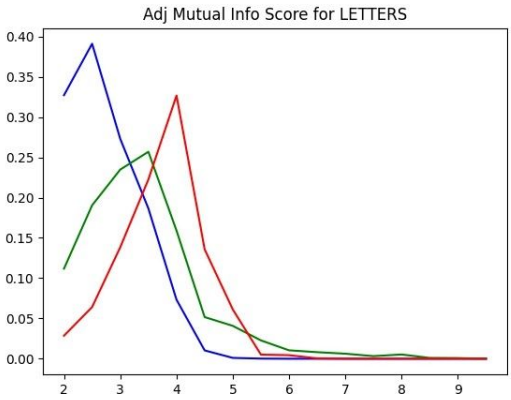


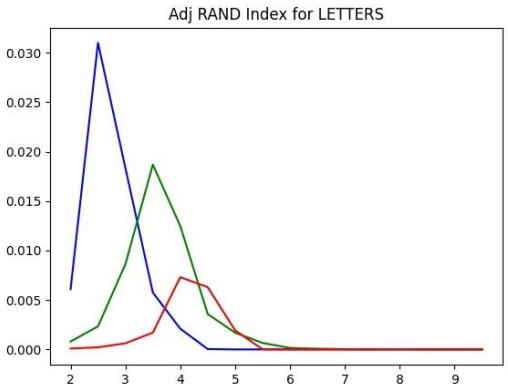
Zoo data set





Seeds data set

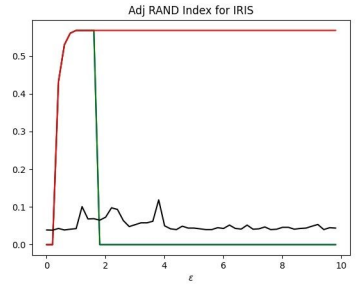


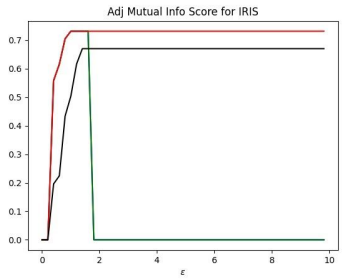


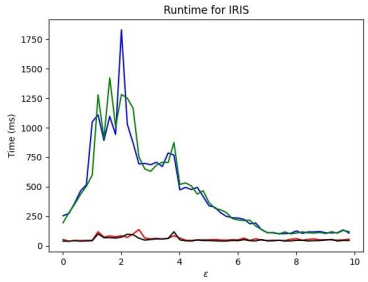
Letters dataset

We got almost similar plots for robustness experiments as in the paper [1]. They used the factor values mentioned in Table 1 to carry out these experiments. As you can see in most of the plots DBSCAN results (Blue) are not showing good performance in a short range epsilon change. For example in iris data set mutual and adjusted rand index scores drops after passing 1.0-1.6 range while K-Center (Red) provides good performance in the range 1.0 - 2.1 eps values and Uniform initialization (Green) results fluctuates the performance values but still allows better result in a wide range of epsilon parameter. There are even some cases where DBSCAN++ gives better results than the DBSCAN algorithm. For example Libras and wine data set give better performance results for some epsilon values than the best performance value for DBSCAN algorithm.

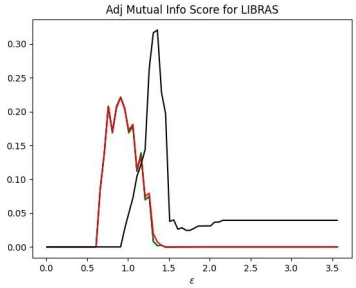
Experiments with approximate nearest neighbour

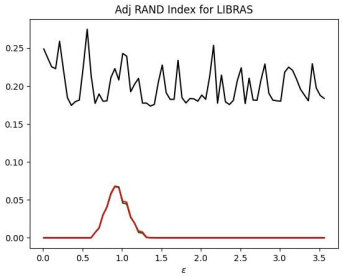


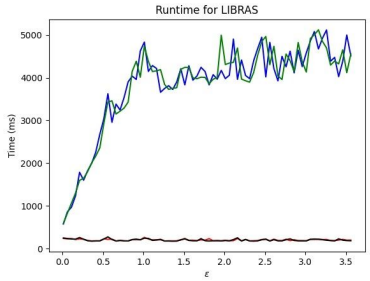




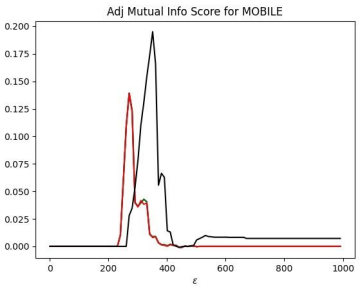
Iris data set (factor = 0.1)

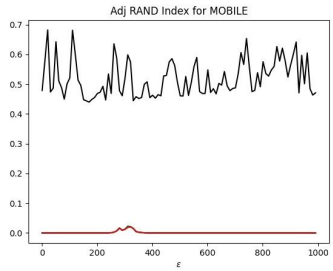


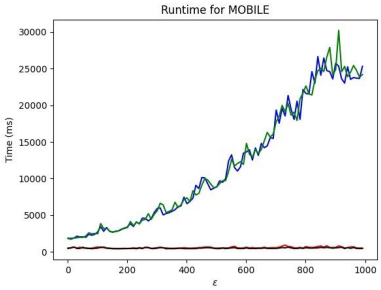




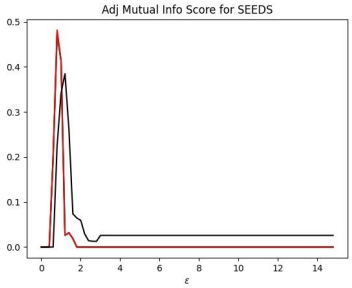
Libras data set (factor = 0.1)

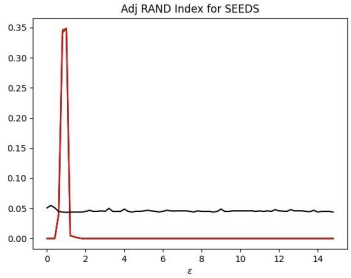


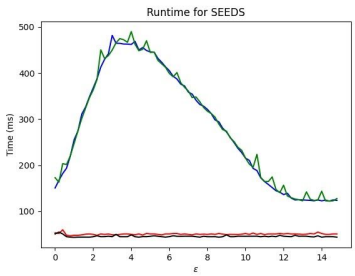




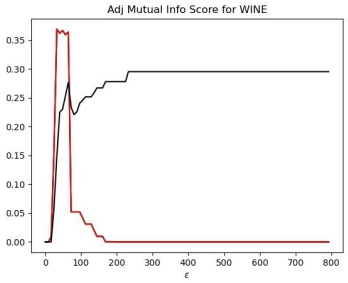
Mobile data set (factor = 0.1)

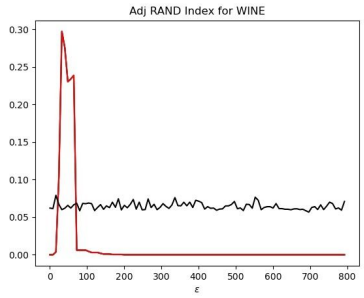


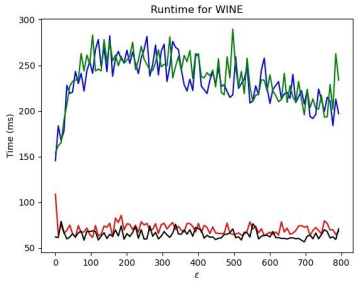




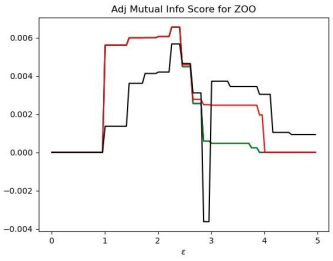
Seeds data set (factor = 0.1)

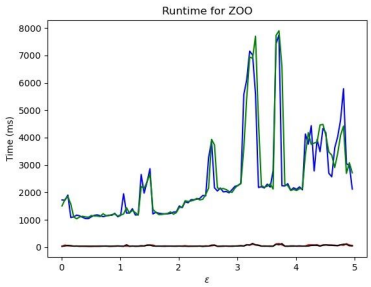


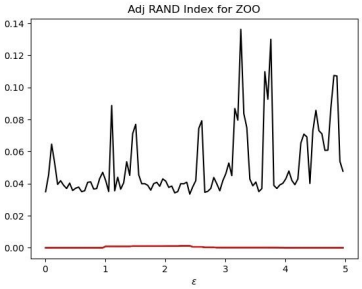




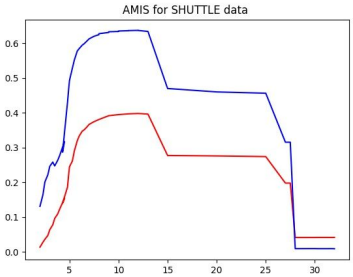
Wine data set (factor = 0.1)

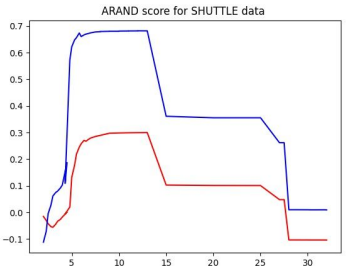






Zoo data set (factor = 0.1)





Shuttle data set

Red - DBSCAN on ANN **Green - DBSCAN++ on K-center** Blue- DBSCAN
Black - DBSCAN++ on ANN with k-center

We also ran DBSCAN on ANN and DBSCAN++ on ANN with k-center on a large data set shuttle. however , we cannot run DBSCAN or DBSCAN++ because of the KDtree limitations.

For the DBSCAN on ANN experiments we, added a new criteria as well we tried DBSCAN++ on k-center for the experiments as well. This is because in our experiments DBSCAN++ on k-center took similar running time as DBSCAN. For the iris data set, we can clearly see a improvement in running time for both DBSCAN on ANN and DBSCAN++ on k-center. Infact, running time wise we can see a huge improvement when using ANN on DBSCAN and DBSCAN++. However, we cannot conclude anything about the performance of DBSCAN

against DBSCAN++ with k-center with these values. For large data set (shuttle) it shows higher robustness for both DBSCAN and DBSCAN++ on ANN. but this is not the case for all the data.

Discussion

We can see in the robustness experiments, when the sub sampling is too low we cannot see good results, but when we increase the sampling level we gradually see an increase in the accuracy, and past some point in sub sampling level accuracy remains high, then we see a trade off in time. So if we can figure out that optimal value that accuracy remains high and time running time is low that's the best trade off we can find.

When we look at the outlier graph we can see when the m/n factor is small the the number of outliers recognized by the DBSCAN++ is high and when the factor value increases and close in to 1 outliers recognized comes to a constant. There is a reason for this behaviour. DBSCAN++ will always recognized a fraction of core-points, that is recognized by DBSCAN on the same hyperparameter values. This is very intuitive and all the other points will be recognized as outliers by the DBSCAN++. When we gradually increase the factor number of core-points recognized by the DBSCAN++ will be increased as well. But after some point the DBSCAN++ will reach the saturation level of core-points. Note that DBSCAN++ will never identify a point as a core-point unless it is a core-point in DBSCAN as well. Thus, when DBSCAN++ reaches this factor where it recognize all the core-points it will not recognize any more outliers. This is why after some factor, the number of outliers becomes a constant.

We can explain the robustness in the following way, When we tune DBSCAN with an epsilon parameter, epsilon contributes more on both designation of points as core points and their inter connection with neighbouring points. Thus, small changes in epsilon value result in significant change in the clustering result. On the other hand, DBSCAN++ suffers less from this epsilon value until it becomes very large. This means it reduces tenuous connections between connected components that are actually far away. This is the reason why DBSCAN++ is less sensitive to hyper parameters. Practically this benefits the user when tweaking the hyperparameters to get better results. This shows that robustness experimental results mentioned in paper matches our results as well.

One of the interesting things that we saw in the running time of the DBSCAN++ is that, for the K-center initialization it does not give a speed up as the original paper mentions. When compared DBSCAN time with DBSCAN++ on K-center it gives either similar running time or DBSCAN++ time is higher than DBSCAN running time. However, DBSCAN++ on uniform initialization gives a better speed up. What we understood was that In both DBSCAN and DBSCAN++ we have to spend the same time to build the KDTree. But DBSCAN++ should always get a less number of queries to do, than the DBSCAN. so their should be a expected speedup from running the less number of queries. However, time taken to run KDTree queries

are really extremely low, thus, we do not save a lot of time for smaller data sets. On top of that K-Center initialization has to run O(mn) time code to initialize the points. This process on the other hand takes some time, in most cases initialization time is greater than the time saved from running less queries. This is why we get this interesting result. On the other hand uniform initialization takes far less time than DBSCAN, which is well suited if we wanted to tweak the parameters.

From the approximate nearest neighbour implementations we cannot conclude anything about the DBSCAN on ANN or DBSCAN++ on ANN. For some data sets, DBSCAN gives better performance while for some DBSCAN++ performs better. However, we saw a considerable increase in the robustness to hyperparameters in DBSCAN on ANN. We can explain this in the same way we explained DBSCAN++. Since we are running approximate nearest queries we might not get all the neighbours from the query result. Thus we might ignore some of the core-points that DBSCAN would recognize. Therefore, we see a considerable robustness to change in hyperparameters on DBSCAN on ANN. We think DBSCAN on ANN is more useful than DBSCAN++ because it increases the robustness to hyperparameters while enabling large data sets to be clustered.

References:

[1] Jang, J. and Jiang, H., 2019, May. DBSCAN++: Towards fast and scalable density clustering. In *International Conference on Machine Learning* (pp. 3019-3029). PMLR.

[2] Muja, Marius, and David Lowe. "Flann-fast library for approximate nearest neighbors user manual." *Computer Science Department, University of British Columbia, Vancouver, BC, Canada* (2009).

[3] Har-Peled, S. (2011). 3. In *Geometric approximation algorithms* (pp. 32-33). Providence, RI: American Mathematical Society.

[4] https://github.com/spotify/annoy

[5] Rand, William M. "Objective criteria for the evaluation of clustering methods." *Journal of the American Statistical association* 66.336 (1971): 846-850.

[6] Hubert, Lawrence, and Phipps Arabie. "Comparing partitions." *Journal of classification* 2.1 (1985): 193-218.

[7] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html, Accessed: 22nd November, 2020

[8 https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_mutual_info_score.html Accessed: 22nd November, 2020