EMORY UNIVERSITY

Homework 3

Student name: *Lance Ding*

Course: *CS470* – Professor: *Dr. Kai Shu*
Due date: *3/2025*

Collaboration Statement

The following resources were utilized in the completion of this project:

Dr. Shengpu Tang's CS334 notes from when I took the class
- General understanding refresher.

Dr. Kai Shu's CS470 slides.
- General understanding refresher.

https://datascience.stackexchange.com/questions/24339/how-is-a-splitting-point-chosen-for-continuous-variables-in-decision-trees
- Revealing how split points are calculated for continuous attributes

ChatGPT
- Bouncing ideas. Also writing testing code for me to benchmark against sklearn's decision tree classifier

https://medium.com/@ompramod9921/decision-trees-13a12ea04524
- Medium article that helped me refresh my understanding of gain ratio.

https://blog.stackademic.com/exploring-numpy-features-performance-vs-lists-7f0b43d2af5f
- Source documenting the difference in speed between python and numpy

https://en.wikipedia.org/wiki/K-medoids
- Source documenting history and common approaches to K-medoids

https://www.geeksforgeeks.org/ml-k-means-algorithm/
- Souce giving actual code and idea for k-means++

Experiment Design

Evaluation Metric. Before diving into the implementation of the decision tree, we can first explore the dataset. We see that this dataset has 303 entries and 14 columns, with a mix of categorical and continuous data types. With the exception of the metadata and target column, the rest of the columns contain biomarkers. The task is one of binary classification - use the available features and predict whether or not a patient has heart disease.

Now that we understand the task, we define our approach. Since we are in a setting where we are trying to predict if a patient has a heart disease, which can enable us to give life-saving treatment, we believe it is of utmost importance that we identify as many patients who actually have the heart disease correctly as possible. In the language of metrics, with the "Yes" label as positive, this is equivalent to maximizing:

$$\frac{TP}{TP + FN}$$

which is recall. However, we must also consider the ethical side-effects of maximizing recall, which may manifest in the increase in the number of false positives. This may induce stress in patients who are in actuality healthy, and may trigger the usage of additional tests to weed out false positives, causing unnecessary financial expenditure. Accounting for this shortcoming, we can instead utilize F-measures.

F-measures, or F-scores, can be computed as such:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \times \text{Recall}}{(\beta^2 \times \text{Precision}) + \text{Recall}}$$

Such a measure essentially gives us a "knob" that controls the balance of reducing false negatives and reducing false positives, much similar to the way regularization works in machine learning. The formula reveals that a lower β, i.e. $\beta < 1$, means precision is more emphasized, whereas a higher β, i.e. $\beta > 1$, means recall is more important. Since we still keep the primary objective of identifying patients who actually have heart disease, we choose to use the F_2 score, which places more twice as much emphasis on recall than precision.

Impurity Metric. For the actual decision tree, we implemented options for Entropy, Gini Index, and Gain Ratio as metrics for determining node impurity. Since there is a case to be made for each measure, we thought it would make sense to implement all three. However, we believe that Gain Ratio is the most suitable impurity measure. This is in part because of the nature of our dataset - a relatively small dataset with a mix of categorical and continuous variables. The discrete variables have more potential split points than the categorical ones, and this creates more possibilities for our model to overfit to noise rather than the underlying pattern since there may be splits where entropy is decreased but the model is not actually doing better. Gain ratio normalizes the effect of this by considering the inherent information in the node, thus reducing overfitting. Additionally, this allows us to have a more balanced idea of what features are actually important since the model will not be as biased towards features with more split points.

Since the above discussion is mostly theoretical, we conduct empirical testing. We utilize cross validation to mitigate random variation, and assess the best impurity measure for our dataset:

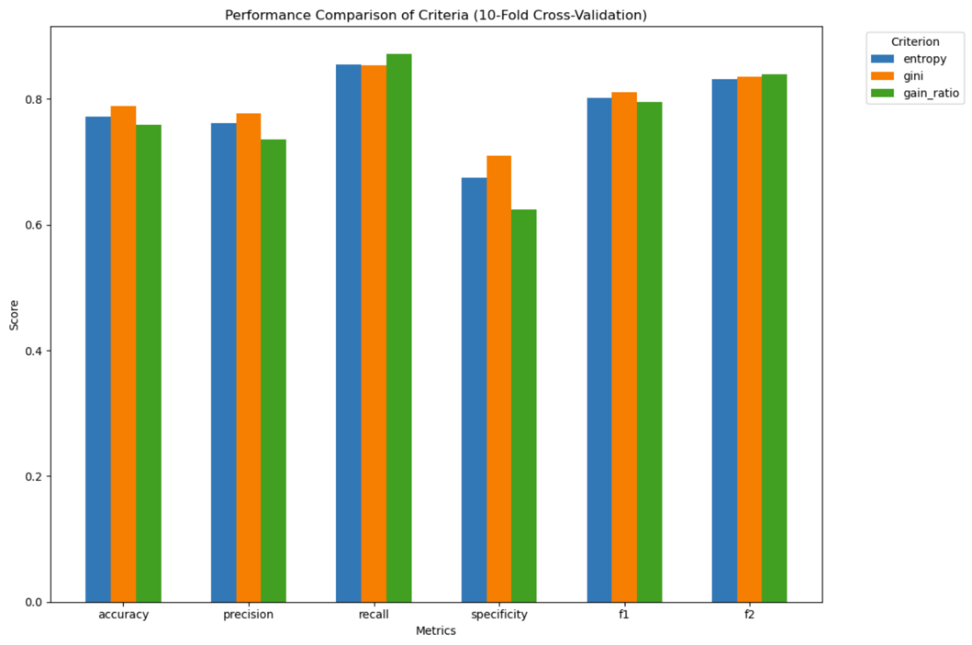


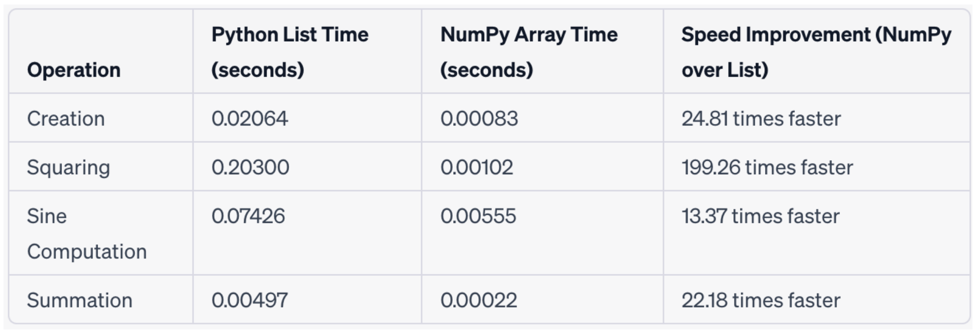
Figure 1: Performance of different impurity metrics

We see that for recall and F_2 score, which incidentally are the measures that we really care about, we get the best performance using the gain ratio as the impurity metric. As such, we choose to use the gain ratio as the default impurity metric.

Data Structures and Engineering. In terms of engineering, attempting to write clean code led to the usage of classes, where we have a Node class and a DecisionTree class. The use of classes grants much more ease in modular adjustments, and in general allows us to write more maintainable code. Since we wanted to create a classifier that made sense to operate and was easy to understand, we implemented the logic as an actual tree, with linked-list style connections between the nodes forming a directed graph from root to leaves. While this approach may be suboptimal in terms of computational overhead, we believe it is the approach that gives the greatest interpretability with the same amount of effort. Additionally, it made code very easy to write - simply perform a pre-order traversal from the root, recursing with the same logic until splits cannot be found, where decisions are made based on majority class.

With a bit of manipulation, we can transform all attributes into numeric attributes via label encoding and one-hot encoding, where all categorical variables get turned into binary variables. This enables us to use an approach that the internet suggested - take every possible split by computing the midpoints between consecutive values.

Optimizations. Regarding optimizations, we have included several in our implementaiton of the Decision Tree Classifier. Throughout all of the code, we utilize numpy's vectorized calculations where possible, which are much faster than native python computations. One source documents a comparison between native python and numpy for a series of operations on a list/array, and the results indicate that the usage of numpy provides for a significant speedup:



Operation	Python List Time (seconds)	NumPy Array Time (seconds)	Speed Improvement (NumPy over List)
Creation	0.02064	0.00083	24.81 times faster
Squaring	0.20300	0.00102	199.26 times faster
Sine Computation	0.07426	0.00555	13.37 times faster
Summation	0.00497	0.00022	22.18 times faster

Figure 2: Comparison of native python and numpy operation speed

We utilize unique values in the generation of splitpoints to eliminate redundant computations. To that end, we also give hyperparameter options for maximum tree depth and minimum samples for splitting to allow for early stopping, which can improve training speed and reduce overfitting. Additionally, during data processing, we transform all features into numeric types, saving memory usage compared to using strings. Finally, we never create or compute more copies of anything more than once, instead relying on slicing and indexing to facilitate a more efficient usage of memory.

Evaluation Methodology. Given that our dataset is relatively small, with just over 300 entries, we feel that cross validation is the best approach. A train-test-split (holdout) with such a small dataset would limit the amount of data available for training and may lead to high variance and overfitting. Bootstrap sampling is another possible approach, but again with such a small amount of data, the risk of overstating the model's performance due to repeated examples that are correctly classified is high. Therefore by process of elimination, some sort of cross-validation would be best. In order to keep the class balance, we utilize the stratified k-fold cross validation from the sklearn library.

In terms of evaluation metric, we previously discussed our choice of metric, and explained why the F_2 score is most suitable. However, we will still go through the interpretation of all of the listed metrics:

- Accuracy - Proportion of correctly classified patients both with and without heart disease
- Precision - Proportion of patients predicted to have heart disease who actually have it
- Recall (Sensitivity) - Proportion of patients with heart disease who are correctly identified by the model
- F-measure - Weighted harmonic mean of precision and recall. Measures both how well the model is able to correctly identify patients with heart disease, as well as how well the model doesn't overstep and incorrectly classify patients without heart disease.
- Specificity - Proportion of patients without heart disease who are correctly identified as negative

Results and Reflection. Now that we have fully discussed the implementation of the Decision Tree Classifier, we can discuss some results. In order to determine the best model, we ran a sweep of the *max_depth* hyperparameter from 1 to 10 inclusive.

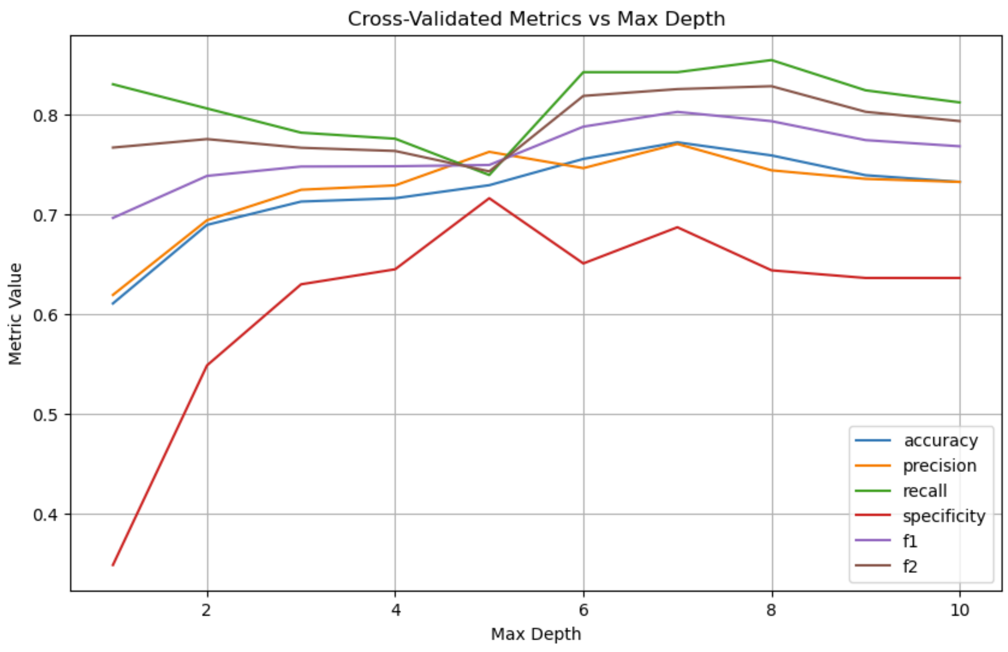
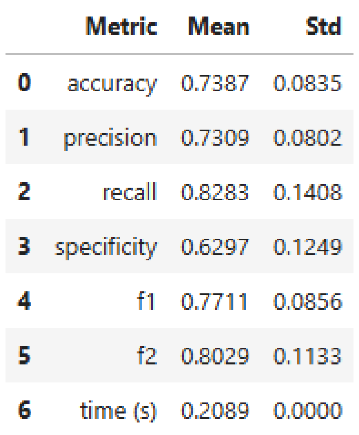


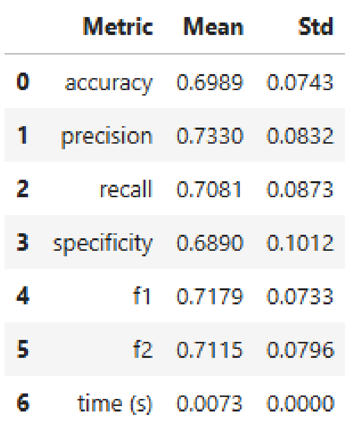
Figure 3: Hyperparameter Sweep Results

Interestingly, we see that at a maximum depth of just 1 (decision stump), we already observe performance for some metrics that is comparable to the highest performing hyperparameters. We can also see that the model achieves said highest performance between a max depth of 6 and 8. We did not sweep the *min_samples_split* hyperparamters, leaving it at 2. Using a max depth of 8 and Gain Ratio as the impurity metric, we observe the following performance from a 10-fold stratified cross-validation evaluation on the dataset:



	Metric	Mean	Std
0	accuracy	0.7387	0.0835
1	precision	0.7309	0.0802
2	recall	0.8283	0.1408
3	specificity	0.6297	0.1249
4	f1	0.7711	0.0856
5	f2	0.8029	0.1133
6	time (s)	0.2089	0.0000

(a) DecisionTree



	Metric	Mean	Std
0	accuracy	0.6989	0.0743
1	precision	0.7330	0.0832
2	recall	0.7081	0.0873
3	specificity	0.6890	0.1012
4	f1	0.7179	0.0733
5	f2	0.7115	0.0796
6	time (s)	0.0073	0.0000

(b) sklearn.tree.DecisionTreeClassifier

Figure 4: Implementation comparisons (average). *sklearn's* classifier is not tuned. Runtime has 0 standard deviation because we timed the overall execution then divided by number of folds

As we can see, we achieve some passable results. We attain a high level of F_2 score and recall as expected, since F_2 score favors recall and we chose the *max_depth* that gave us the best F_2 score. Notably, we are significantly higher than the default, untuned sklearn Decision Tree Classifier in the relevant (recall, F_2) metrics. However, our implementation runs orders of magnitudes slower.

We also conduct some interpretability testing via SHAP. According to the SHAP python library documentation, "SHAP (SHapley Additive exPlanations) is a game theoretic approach to explain the output of any machine learning model. It connects optimal credit allocation with local explanations using the classic Shapley values from game theory and their related extensions". Here we utilize the SHAP python library's KernelExplainer to conduct analysis.

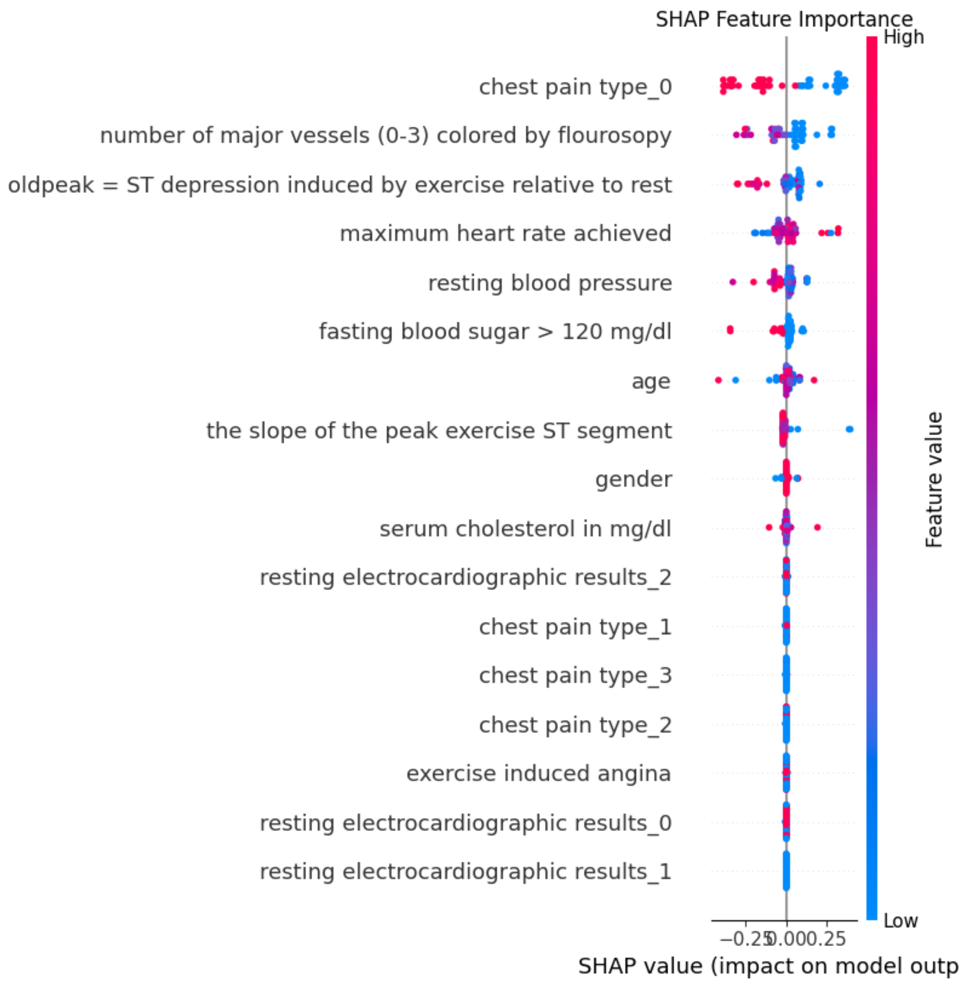


Figure 5: SHAP Summary Plot

We can see that there are several variables that are very important according to the SHAP analysis, which may explain our earlier discovery regarding the decision trees with a single layer were performing well. Other than that, the analysis reveals that heart disease covaries negatively with chest pain, negatively with the number of vessels colored by flourosopy, and negatively with ST depression. More domain knowledge is required to interpret those results and if they make sense.

Most of the development work was done in the two jupyer notebooks (*EDA_Dev.ipynb* and *Testing.ipynb*). All graphs that are in this report are available at

Testing.ipynb. For usage, the actual script that fits the description of the assignment is in *decision_tree.py*, and it takes the parameters that the assignment specifies. For usage instructions, refer to the *README.txt* as well as the *-h* page of the script. The default configuration of the script takes in a *data.csv*, *para2_file.txt* and *para3_file.txt* from the root directory of the script, and outputs a *para_4.txt* in the same directory by using a decision tree with *max_depth* 8 and Gain Ratio as the impurity measure. Note that this classifier only works for binary classification since that is the scope of the assignment. For testing purposes, we include a script *make_paras.py* that generates a train-test-split into the files to be used by the main script.

Overall, implementing the decision tree has taught me several things. The most prominent thing is the difficulty of optimization. We see sklearn's implementation being on the order of 10-100x faster than ours, and that kind of optimization isn't something that we can quickly attain. Additionally, this part of the assignment took me far longer than I expected, and showed me that indeed, as Dr. Kai Shu often says in class, you only truly understand by doing.

Clustering Algorithms

In this section, we discuss k-means++ and k-medoids, which are clustering algorithms that fulfill the same purpose as the k-means algorithm discussed in class - to minimize a "cost" often defined as the sum of the distances between cluster points and their centers. We provide motivations for the development of each of the two new algorithms based on the weaknesses of the k-means algorithm as well as the pseudocode of the new algorithms and their improvements.

The k-means algorithm is a classic clustering algorithm that groups points together by their distance to a set of centroids that act as "centers" for the clusters. It is an iterative, unsupervised learning method that partitions a dataset into k clusters by minimizing the variance (between points and centroids) within each cluster. It begins by randomly selecting k initial centroids and then assigns each data point to the nearest centroid based on a distance metric, typically Euclidean distance. After assigning all points, the centroids are recalculated as the mean of the points in each cluster. This process repeats until a stopping condition is met: 1) change in centroids is less than ϵ after an iteration or 2) a maximum number of iterations is reached.

k-means++. While powerful, k-means has several shortcomings. The first one is that it is sensitive to initial conditions and may converge to local minima that are far from globally optimal. Since k-means (uniformly) assigns initial centroids randomly, there is some optimization to be done, specifically following the idea that a better initial spread of centroids may lead to better results.

k-means++ attempts to address this issue. This variant of k-means assigns a random first centroid, then initializes centroids with probability proportional to the squared Euclidean distance of the candidate point to their nearest centroid, favoring points that are further away from already initialized centroids. The rest of the algorithm proceeds exactly the same as the base version of the k-means algorithm. As such, it is probably more sensible to refer to k-means++ as an initialization scheme rather than a full algorithm.

Algorithm 1 k-means++

```
1: Input: Dataset $X = \{x_1, x_2, \ldots, x_n\}$, Number of clusters $k$
2: Output: Cluster centroids $C = \{c_1, c_2, \ldots, c_k\}$
3: // k-means++ initialization
4: Choose the first centroid $c_1$ randomly from $X$
5: for $i = 2$ to $k$ do
6:     for each data point $x_j \in X$ do
7:         Let $o_j$ be the centroid closest to $x_j$
8:         Compute $D(x_j) = |x_j - o_j|^2$ as the squared Euclidean distance between $x_j$ and $o_j$
9:     end for
10:    Choose $c_i$ from $X$, where each $x_j$ has probability $\frac{D(x_j)}{\sum D(x)}$ of being selected
11: end for
12: // k-means clustering
13: repeat
14:     Assign each $x_j$ to the nearest centroid by Euclidean (L2) distance
15:     Update each centroid $c_i$ as the mean of assigned points
16: until centroids do not change
```

k-medoids. Another shortcoming of the k-means clustering algorithm - one that k-means++ doesn't address - is the sensitivity of means to outliers and noise. Much in the same way that outliers in datasets skew the mean, centroids also drift with extreme values. As such, using outlier-resistant measures to act as centers for clusters would theoretically form more outlier-resistant clusters and yield better results.

k-medoids addresses this issue by using actual data points as cluster "centers". This is different from k-means, which uses the average of the points in a cluster as the "center", meaning that the k-means centroids do not have to be actual data points. Essentially, this is the median to k-means' mean, and it is well established from statistics that medians are more resistant to skew than means.

However, k-medoids also has its own drawbacks. The most significant one is its computational cost. The k-medoids problem, where the task is to find the configuration of k medoids such that the total cost (sum of distances from points to their closest medoid) is globally minimized, is actually NP-hard. This means that unless $P = NP$, exact solutions to this problem require exponential time, which is considered intractable for large problem sizes. To mitigate the issue, we often take heuristics-based or greedy approaches that in practice often require only polynomial time to converge to local minima. One of the most popular approaches is the PAM (Partitioning Around Medoids) algorithm, which involves selecting medoids then iteratively swapping medoids with non-medoids greedily to minimize total cost.

Algorithm 2 k-medoids (PAM)

```
1: Input: Dataset $X = \{x_1, x_2, \ldots, x_n\}$, Number of clusters $k$
2: Output: Cluster medoids $M = \{m_1, m_2, \ldots, m_k\}$
3: // Initialization
4: Choose $k$ random points from $X$ as initial medoids $M$
5: // Clustering
6: repeat
7:     Assign each data point $x_j$ to the nearest medoid by Euclidean (L2) Distance
8:     Let bestDelta record the best cost change, bestSwap record the swap with the
   best cost change
9:     bestSwap = (null, null)
10:    bestDelta = 0
11:    for each medoid $m_i \in M$ do
12:        for each non-medoid point $x_j \in X$ do
13:            Calculate total cost if $m_i$ is swapped with $x_j$
14:            Let $\Delta$ be the cost difference after the swap
15:            if $\Delta <$ bestDelta then
16:                bestDelta = $\Delta$
17:                bestSwap = $(m_i, x_j)$
18:            end if
19:        end for
20:    end for
21:    if bestDelta $< 0$ then
22:        Swap medoid status between the two points in bestSwap
23:    end if
24: until bestDelta $\geq 0$ or max iterations reached
```

This algorithm once again displays the issues of sensitivity to initial conditions. As such, the k-means++ initialization scheme could be applied to this algorithm to potentially increase its performance.