Threshold clustering for massive data

by

Jianmei Luo

M.A., University of Missouri, 2014

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Statistics
College of Arts and Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2019

Abstract

Statistical clustering is the process of partitioning objects into clusters so that units within the same cluster have similar characteristics. Threshold clustering (TC) is a recently-developed method designed so that, given a pre-specified threshold t^*, each cluster contains at least t^* units and the maximum within-cluster dissimilarity (MWCD) is small. In fact, among all clusterings which contain at least t^* units within each cluster, the MWCD obtained through TC is, at most, 4 times as large as the smallest possible MWCD; finding the clustering with the smallest MWCD possible is known as the bottleneck threshold partitioning problem (BTPP). TC requires only $O(t^*n)$ time and space outside of the construction of a $(t^* - 1)$–nearest-neighbors graph.

This dissertation centers on extending the TC methodology in several different ways. First, we extend TC to instance selection problems on massive data through a procedure called iterated threshold instance selection (ITIS). It works as follows. First, TC is performed on the n units to form n^* clusters, each containing t^* or more units. Then, prototypes are formed by computing a center point for each of the n^* groups (e.g. a centroid or medoid). Finally, if the data is not yet sufficiently reduced, TC is applied again to the n^* prototypes. These steps are repeated until the data has been sufficiently reduced. Besides scaling down the data size, we use ITIS as a pre-processing step to allow for the use of more sophisticated clustering methods. A statistical clustering of all n units can be obtained by an approach called Iterative Hybridized Threshold Clustering (IHTC). In addition to improving efficiency, IHTC also prevents singular data points from being overfit by desired clustering methods; specifically, for m iterations of ITIS at size threshold t^*, IHTC ensures that clusters contain at least $(t^*)^m$ units. We give results from simulations and applications to real datasets showing that our methods reduce the run time and memory usage of the original clustering algorithms while still preserving their performance.

Second, we extend TC to hierarchical clustering in two ways. The first method, called hierarchical agglomerative threshold clustering (HATC), follows the hierarchical agglomerative clustering (HAC) paradigm closely. First, TC is performed on the n units to form several clusters, each containing t^* or more units. Then, the cluster with the smallest within-cluster dissimilarity is condensed into a single point, and TC is again performed on the reduced data. These steps are repeated until there are fewer than t^* data points. The second method repeatedly performs ITIS to form a cluster hierarchy. Initially, TC is performed on the n units. Then, a prototype is formed for each cluster, and TC is performed again on these prototypes. As before, these steps are repeated until there are fewer than t^* points remaining. Results from simulations show that these clustering methods perform comparably to, if not better than, HAC.

Finally, we fill in some gaps in understanding on how well an algorithm can approximate BTPP in polynomial time. Previous work has shown the existence of a 4-approximation polynomial-time algorithm to BTPP for arbitrary t^* and has shown that, when $t^* > 2$, no $(2 - \epsilon)$–approximation algorithm can be performed in polynomial time unless P = NP. We show that a polynomial-time 2–approximation algorithm exists when $t^* = 2$ and a polynomial-time 3–approximation algorithm exists when $t^* = 3$.

Threshold clustering for massive data

by

Jianmei Luo

M.A., University of Missouri, 2014

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Statistics
College of Arts and Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2019

Approved by:

Major Professor
Michael Higgins

Copyright

Abstract

Statistical clustering is the process of partitioning objects into clusters so that units within the same cluster have similar characteristics. Threshold clustering (TC) is a recently-developed method designed so that, given a pre-specified threshold t^*, each cluster contains at least t^* units and the maximum within-cluster dissimilarity (MWCD) is small. In fact, among all clusterings which contain at least t^* units within each cluster, the MWCD obtained through TC is, at most, 4 times as large as the smallest possible MWCD; finding the clustering with the smallest MWCD possible is known as the bottleneck threshold partitioning problem (BTPP). TC requires only $O(t^*n)$ time and space outside of the construction of a $(t^* - 1)$–nearest-neighbors graph.

This dissertation centers on extending the TC methodology in several different ways. First, we extend TC to instance selection problems on massive data through a procedure called iterated threshold instance selection (ITIS). It works as follows. First, TC is performed on the n units to form n^* clusters, each containing t^* or more units. Then, prototypes are formed by computing a center point for each of the n^* groups (e.g. a centroid or medoid). Finally, if the data is not yet sufficiently reduced, TC is applied again to the n^* prototypes. These steps are repeated until the data has been sufficiently reduced. Besides scaling down the data size, we use ITIS as a pre-processing step to allow for the use of more sophisticated clustering methods. A statistical clustering of all n units can be obtained by an approach called Iterative Hybridized Threshold Clustering (IHTC). In addition to improving efficiency, IHTC also prevents singular data points from being overfit by desired clustering methods; specifically, for m iterations of ITIS at size threshold t^*, IHTC ensures that clusters contain at least $(t^*)^m$ units. We give results from simulations and applications to real datasets showing that our methods reduce the run time and memory usage of the original clustering algorithms while still preserving their performance.

Second, we extend TC to hierarchical clustering in two ways. The first method, called hierarchical agglomerative threshold clustering (HATC), follows the hierarchical agglomerative clustering (HAC) paradigm closely. First, TC is performed on the n units to form several clusters, each containing t^* or more units. Then, the cluster with the smallest within-cluster dissimilarity is condensed into a single point, and TC is again performed on the reduced data. These steps are repeated until there are fewer than t^* data points. The second method repeatedly performs ITIS to form a cluster hierarchy. Initially, TC is performed on the n units. Then, a prototype is formed for each cluster, and TC is performed again on these prototypes. As before, these steps are repeated until there are fewer than t^* points remaining. Results from simulations show that these clustering methods perform comparably to, if not better than, HAC.

Finally, we fill in some gaps in understanding on how well an algorithm can approximate BTPP in polynomial time. Previous work has shown the existence of a 4-approximation polynomial-time algorithm to BTPP for arbitrary t^* and has shown that, when $t^* > 2$, no $(2 - \epsilon)$–approximation algorithm can be performed in polynomial time unless P = NP. We show that a polynomial-time 2–approximation algorithm exists when $t^* = 2$ and a polynomial-time 3–approximation algorithm exists when $t^* = 3$.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Foremost, I would like to express my sincere gratitude and deepest appreciation to my major professor: Dr. Michael Higgins. His motivation, enthusiasm, patience and valuable suggestions regarding my research, review of my writing and constant inspiration greatly helped to advance my research training and to complete this dissertation. I could not have imagined having a better advisor and mentor for my Ph.D study.

I would also like to extend my gratitude to Dr. Gary Gadbury, Dr. Christopher Vahl, Dr. Cen Wu, Dr. William Hsu and Dr. Marianne Korten for their precious time to serve as my committee members, and their helpful comments and insightful suggestions on improving the quality of my work. Moreover, I am thankful to those who have played a role in polishing my research writing skills. Thanks should also go to the faculty and staff of the department of statistics who made it a warm and enjoyable experience at K-State.

Lastly, I would like to thank my family for all their love and encouragement.

Chapter 1

Introduction

1.1 Introduction

Clustering, also known as unsupervised learning, is a well-studied problem in machine learning. It aims to group units with similar features together and separate units with dissimilar features (Friedman et al., 2001). Cluster analysis has been used in many fields like biology, management, pattern recognition, etc. Customer segmentation divides a customer base into groups of individuals that are similar so that companies can target groups more effectively and allocate marketing resource to gain best profit (Madhavaram and Hunt, 2008). Additionally, many methods (e.g. k-means clustering, hierarchical agglomerative clustering, etc.) have been developed that successfully tackle the clustering problem.

However, enormous amounts of data are collected every day. For example, Walmart performs more than 1 million customer transactions per hour (Cukier, 2010), and Google performs more than 3 billion searches per day (Sullivan, 2015). This becomes a massive accumulation of data. When working with data of such a large *size*, many of the state-of-the-art clustering methods become intractable. That is, massive data requires novel statistical methods to process this data; research on scaling up existing statistical algorithms and scaling down the size of data without loss of information is of critical importance (Jordan and Mitchell, 2015).

Instance selection is a commonly-used pre-possessing method for scaling down the size of data (Blum and Langley, 1997; Liu and Motoda, 1998, 2002). The goal of instance selection is to shorten the execution time of data analysis by reducing data size n while maintaining the integrity of data (Olvera-López et al., 2010). Instance selection methods rely on sampling, classification algorithms, or clustering algorithms. Commonly used sampling methods like simple random sampling and adaptive sampling require to determine the sample size beforehand. Based on a training set in which all the labels are known, we can use classification algorithms to identify which label the new instance belongs to. Methods rely on classification algorithms can identify which group a new instance belongs based on a training set of units whose label is known. When the labels are unknown, instance selection methods related to clustering algorithms should be considered. Previous work has shown methods reliant on clustering have better performance (accuracy) than some methods that rely on classification (Olvera-López et al., 2007, 2008, 2010; Raicharoen and Lursinsap, 2005; Riquelme et al., 2003). However, current instance selection methods that rely on classification often have faster runtimes.

On the other hand, threshold clustering (TC) is a recently developed method for clustering that is extremely efficient. TC is a method of clustering units so that each cluster contains at least a pre-specified number of units t^* while ensuring that the within-cluster dissimilarities are small. Previous work has shown that, when the objective is to minimize the maximum within-cluster dissimilarities, a solution within a factor of four of optimal can often be obtained in $O(t^*n)$ time and space when the $(t^* - 1)$-nearest-neighbors graph is given (Higgins et al., 2016). The runtime and memory usage required for TC is smaller compared to other clustering methods, for example, k-means and hierarchical agglomerative clustering (HAC).

1.2 Overview of Statistical Literature on Instance Selection

The nearest neighbor decision rule (NN) is an instance selection method related to classification developed by Cover and Hart (1967) which require large storage space if data size is massive. Also, NN is sensitive to noise. The condensed nearest neighbor rule (CNN) was proposed to overcome NN's drawback but it may contain noisy instances (Olvera-López et al., 2010). The edited nearest neighbor rule (ENN) can handle the noisy instance problem, and experiments done by Tomek (1976); Wilson (1972) shows the performance of ENN is better than that of NN rule. Selective nearest neighbor (SNN) aims to find a subset of the dataset that can not only make all decision boundaries unchanged, but also reduce computation requirement (Ritter et al., 1975). Ritter et al. (1975) reproduced the data used in P.Hart's paper (Hart, 1968) and figured out that SNN has more precise decision boundaries compared with using CNN. Aha et al. (1991) developed several instance-based methods (IB, IB2, IB3) that can reduce storage requirement while distinguishing noisy instance. However, IB3 is sensitive to dimension of instance. Wilson and Martinez (2000) proposed several additional reduction algorithms (DROP1-DROP5) for instance selection that have higher average generalization accuracy compared with the other ten existing reduction algorithms (eg: kNN, CNN, SNN, IB2, IB3 etc.). Generalized condensed nearest neighbor (GCNN) rule was proposed to extract prototype quickly and have accuracy close to those for the full dataset (Chou et al., 2006). Cano et al. (2003) found that performing genetic algorithms (GAs) as an instance selection method will have higher classification accuracy.

The k-means clustering algorithm (MacQueen et al., 1967) can be used in instance selection. This algorithm partitions the data into k groups, the mean of each group can be treated as prototype. However, k-means will converge to a locale optimal and sensitive to the initialization. Also, it tends to overfit the isolated units. Squashing data is another type of selection instance in clustering. It reduce the size of rows for the dataset while maintain the attributes as the original one. DuMouchel et al. (1999) defined a model free data squashing

method that constructs a smaller dataset while contains almost the same information as the original one. Owen (2003) developed a model dependent data squashing method based on empirical likelihood. The likelihood-based data squashing (LDS) method partitions data using a statistical model to squash the data (Madigan et al., 2002). Previous work showed DuMouchel (2002)'s method has higher computational complexity than LDS method, Owen's method has lowest computaional complexity among these three. Generalized-modified Chang algorithm (GMCA) is an CBL method that merges the two same-class nearest clusters into a new cluster and chooses centers from the new merged clusters as prototypes (Mollineda et al., 2002). Later, Lumini and Nanni (2006) proposed an CBL method that use centers for each cluster as prototype.

1.3 Dissimilarity Measurements

Consider a dataset with n units, numbered 1 through n. Each unit i has a response vector $\mathbf{y}_i$ and a d-dimensional covariate vector $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{id})$. For each pair of units i, j, the *dissimilarity* between i and j, denoted d_{ij}, can be computed. Often, the dissimilarity is chosen so that, if units i and j have similar values of covariates $\mathbf{x}$, then d_{ij} is small, and clustering methods often try to group units with small dissimilarity together. We assume that $d_{ij} \geq 0$, and that dissimilarities satisfy the triangle inequality; for any units i, j, k,

$$d_{ij} + d_{jk} \geq d_{ik} \tag{1.1}$$

Common choices of dissimilarities include Euclidean distance, Manhattan distance and Mahalanobis Distance distance.

- Euclidean distance

$$d_{ij} = \sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + ... + (x_{id} - x_{jd})^2} = ||\mathbf{x}_i - \mathbf{x}_j||_2$$

- Manhattan distance

$$d_{ij} = |x_{i1} - x_{j1}| + |x_{i1} - x_{j1}| + ... + |x_{i1} - x_{j1}| = ||\mathbf{x}_i - \mathbf{x}_j||_1$$

- Mahalanobis distance

$$d_{ij} = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^T S^{-1} (\mathbf{x}_i - \mathbf{x}_j)}$$

  where S is the $d \times d$-dimensional variance and covariance matrix of $\mathbf{X}$.

Dissimilarity matrix $\mathbf{D}$ is an n by n symmetric matrix containing all the pairwise dissimilarity between units in the dataset.

1.4 Overview of Statistical Literature on Clustering

Nowadays, hundreds of clustering algorithms have been proposed. In this study, we focus on two types of clustering method: partitional and hierarchical. K-means clustering is a partitional clustering method which finds all the clusters simultaneously. It begins with k initial units, iteratively reallocating units among cluster groups until they converge. Cluster membership is determined by computing the center of each cluster and assigning each unit to the cluster group with the closest center. This algorithm is based on minimizing within-cluster sum of square, which makes the clustering problem into an optimization problem. In theory, if we enumerate all possible clustering ways (k^n), the problem can be solved. This problem can not be solved in polynomial-time. Researchers proved that finding an exact solution to the k-means problem is NP-hard even when $k = 2$ (Drineas et al., 2004). Hierarchical clustering combines or divides existing groups to create a hierarchical structure that reflects the order of which groups are merged or divided.

1.4.1 K-means Clustering

The k-means clustering algorithm (Lloyd, 1982) is one of the most widely used and effective methods that attempts to partition units into exactly k-clusters.

Lloyd (1982) developed a local search procedure which is still used today for k-means clustering algorithm. At its core, k-means aims to form these clusters so that the sum of squares of Euclidean distances between the cluster and the cluster center is small. Specifically, k-means aims to solve the following optimization problem:

$$\sum_{c=1}^{k} \sum_{\mathbf{x} \in S_c} ||\mathbf{x} - \mu_{\mathbf{c}}||_2^2 \tag{1.2}$$

where $\mu_{\mathbf{c}}$ is the center of units in group S_c and $||\mathbf{x} - \mu_{\mathbf{c}}||_2$ is the Euclidean distance between unit $\mathbf{x}$ and $\mu_{\mathbf{c}}$. While this problem is NP-hard (Drineas et al., 2004), the k-means algorithm provides a fast, heuristic solution for this objective.

The k-means clustering algorithm proceeds as follows:

1. **(Initialization)** Randomly select a set of k units (referred as *centers*) from the dataset. K denote the number of clusters and it should be pre-specified.
2. **(Assignment)** Assign all the units to the nearest center, based on squared Euclidean distance, to form k temporary clusters.
3. **(Updating)** Recompute the mean of each cluster. Replace the *centers* with the new k cluster means.
4. **(Terminate)** Repeat step 2 and 3 until there is no further change for the *centers*.

The time complexity for the k-means clustering algorithm is $O(nkLd)$ and the space complexity is $O((k+n)d)$ (Firdaus and Uddin, 2015; Hartigan and Wong, 1979) where d is the number of attributes for each unit, L is the number of iterations taken by the algorithm to converge.

The k-means algorithm suffers from a number of drawbacks (Hastie et al., 2009).

High sensitivity to initial units being chosen is a main shortcoming of k-means clustering. This algorithm will converge to a local optimal, it can not guarantee to have global optimal. It is possible that different initial units yield to different clustering results. Since the number of clusters k is pre-specified, if the choice of k is not proper, it may cause poor results. Even though this procedure will guarantee to have k clusters, it trends to overfit the isolated units

which may lead to some clusters containing only a few units. Also, k-means clustering is not applicable for non-globular data.

Many methods have been developed to improve the quality of clustering by weakening the effect of initialization. Ball and Hall (1967) proposed a method using distance threshold T to guarantee the centers are separated. Finding a T that works properly for the data is a shortcoming. Mao and Jain (1996) proposed k-means using Mahalanobis distance that can detect hyperellipsoidal cluster. The cost is higher computation complexity. Arthur and Vassilvitskii (2007) developed a procedure (k-means++) to initialize the cluster centers before proceeding with the k-means clustering algorithm, experiments show his method run faster than k-means and has better accuracy.

Fränti et al. (1997, 1998) proposed genetic algorithms (GA) and tabu search (TS) that consider several possible initialization which depend less on the initialization The disadvantage is that these approaches are time consuming. Arthur and Vassilvitskii (2007) augmented k-means with a simple, randomized seeding technique that runs faster than k-means and has better performance. Fränti and Kivijärvi (2000) developed a new algorithm named randomized local search (RLS) that has similar result as GA and TA but runs much faster.

1.4.2 Hierarchical Agglomerative Clustering

Hierarchical agglomerative clustering (Ward Jr, 1963), which is also called HAC, is a "bottom up" approach that aims to build a hierarchy clusters. It initially treats each unit as a cluster and then continues to merge two clusters together until only one cluster remains.

For the HAC, we do not need to pre-specify the number of clusters. Desired number of clusters can be obtained by using a *dendogram*—a reverse tree that shows how the units are merged. The HAC proceeds as follows:

1. **(Initial Clusters)** Start with n clusters, each cluster only contains one unit.
2. **(Merge)** Merge the closest (most similar) pair of clusters into a single cluster.
3. **(Updating)** Recompute the distance between the new cluster and the original clusters.

4. **(Terminate)** Repeat step 2 and 3 until one cluster remains, the cluster contains n units.

In order to figure out which clusters should be merged, measurements about inter-cluster distance should be defined, these type of measurements are referred as *linkage criteria*. For clusters A and B, there are several commonly used linkage criteria:

- **Single linkage**. The distance between cluster A and B is defined to be the distance between the closest pair of units belonging to different clusters.

$$d_{single}(A, B) = \min_{\mathbf{x_i} \in A, \mathbf{x_i} \in B} d_{ij} \tag{1.3}$$

- **Complete linkage**.The distance between cluster A and B is represented by the maximum distance between any two units belonging to different clusters.

$$d_{complete}(A, B) = \max_{\mathbf{x_i} \in A, \mathbf{x_i} \in B} d_{ij} \tag{1.4}$$

- **Average linkage**. Also referred to as the Unweighted Pair Group Method with Arithmetic Mean (UPGMA) (Sokal, 1958). The distance between two clusters A and B is defined as the average distance of all pairs of units belonging to different clusters.

$$d_{average}(A, B) = \frac{1}{|A||B|} \sum_{\mathbf{x_i} \in A} \sum_{\mathbf{x_j} \in B} d_{ij} \tag{1.5}$$

  where $|A|, |B|$ represent the size of cluster A and B, respectively.

- **Centroid linkage**. The distance between two clusters A and B is distance between the centers of the clusters.

Single linkage works well for non-elliptical shaped data. However, it is sensitive to outliers and noise. Complete linkage is more robust to outliers and noise but does not works well for convex shaped dataset (Steinbach et al., 2004). Average linkage is an intermediate approach

between single linkage and complete linkage. Centroid linkage is conceptually simpler than the average of all pairwise distances in UPGMA but it may produce dendograms with inversions, which make visualization messy. "Cutting" the dendogram of average linkage and centroid linkage provides no interpretation. However, "cutting" the dendogram for single linkage and complete linkage can be interpreted. For average linkage and centroid linkage, cluster results may change if monotone transformation was applied on the dissimilarity measure. For example, change from Euclidean distance to Squared Euclidean distance will return different clustering (Friedman et al., 2001; James et al., 2013).

The linkage criteria should be pre-determined by the researcher. The choice of linkage will affect the clustering procedure and impact the cluster result. Almeida proposed a self-consistent outlier reduction approach to overcome outlier sensitive issues for single linkage clustering (Almeida et al., 2007). Ward's method does not directly define a measure of distance between two units or clusters, instead, two clusters with the smallest increase in the error sum of squares will merge (Ward Jr, 1963). This method is less susceptible to noise and outliers but does not work well for globular clusters. HAC requires linkage criteria to measure inter-cluster distance but initialization and the choice of k is no longer a problem to worry about. However, the time complexity of HAC is $O(n^2 \log(n))$ (Kurita, 1991) and space complexity is $O(n^2)$ (Jain et al., 1999). This complexity limits its application to massive data. Another hindrance of HAC is that every merging decision is final. Once two clusters are merged into a new cluster, there is no way to partition the new cluster in later steps.

1.5 Organization of the Dissertation

So far we have portrayed commonly used dissimilarity measurements, k-means clustering and hierarchical agglomerative clustering algorithms in a general way. Chapter 2 discussed in detail the threshold clustering (TC). We extend the threshold clcustering methodology to clustering problems under massive data setting through viewing threshold clustering as an instance selection procedure. We also extend TC to hierarchical agglomerative threshold clustering (HATC) which introduces in Chapter 4. In Chapter 3, a polynomial-time 3-

approximation algorithm when $t^* = 3$ and a polynomial-time 2-approximation algorithm when $t^* = 2$ are proposed. Finally, we conclude in Chapter 5.

Chapter 2

Iterated Hybridized Threshold Clustering for Massive Data

2.1 Introduction

Clustering, also known as unsupervised learning, is a well-studied problem in machine learning. It aims to group units with similar features together and separate units with dissimilar features (Friedman et al., 2001). Cluster analysis has been used in many fields like biology, management, pattern recognition, etc. Additionally, many methods (e.g. k-means clustering, hierarchical agglomerative clustering, etc.) have been developed that successfully tackle the clustering problem.

However, enormous amounts of data are collected every day. For example, Walmart performs more than 1 million customer transactions per hour (Cukier, 2010), and Google performs more than 3 billion searches per day (Sullivan, 2015). This becomes a massive accumulation of data. When working with data of such a large *size*, many of the state-of-the-art clustering methods become intractable.

That is, massive data requires novel statistical methods to process this data; research on scaling up existing statistical algorithms and scaling down the size of data without loss of information is of critical importance (Jordan and Mitchell, 2015).

Instance selection is a commonly-used pre-possessing method for scaling down the size of data (Blum and Langley, 1997; Liu and Motoda, 1998, 2002). The goal of instance selection is to shorten the execution time of data analysis by reducing data size n while maintaining the integrity of data (Olvera-López et al., 2010). Instance selection methods rely on sampling, classification algorithms, or clustering algorithms. Previous work has shown methods reliant on clustering have better performance (accuracy) than some methods that rely on classification (Olvera-López et al., 2007, 2008, 2010; Raicharoen and Lursinsap, 2005; Riquelme et al., 2003). However, current instance selection methods that rely on classification often have faster runtimes.

On the other hand, threshold clustering (TC) is a recently developed method for clustering that is extremely efficient. TC is a method of clustering units so that each cluster contains at least a pre-specified number of units t^* while ensuring that the within-cluster dissimilarities are small. Previous work has shown that, when the objective is to minimize the maximum within-cluster dissimilarities, a solution within a factor of four of optimal can often be obtained in $O(t^*n)$ time and space when the $(t^* - 1)$-nearest-neighbors graph is given (Higgins et al., 2016). The runtime and memory usage required for TC is smaller compared to other clustering methods, for example, k-means and hierarchical agglomerative clustering (HAC).

In this chapter, we propose the use of TC for instance selection. The proposed method, which is called iterated threshold instance selection (ITIS), works as follows. For a given t^*, TC is applied on n units to form n^* clusters; each cluster will contain at least t^* units. Then prototypes are formed by finding the center of each cluster. TC is applied again to the n^* prototypes if the data is not sufficiently reduced. Otherwise, the procedure is stopped.

We also propose using ITIS as a pre-processing step on large data to allow for the use of more sophisticated clustering methods. First, ITIS is applied to form a sufficiently small set of prototypes. Then, a more sophisticated clustering algorithm (e.g. k-means, HAC) is applied on this set of prototypes. Finally, a clustering on all n units is obtained by "backing out" the cluster assignments for the prototypes—for each prototype, the units used to form the prototype are determined; these units are assigned to the same cluster assigned to the

prototype. This clustering process on all n units is called Iterative Hybridized Threshold Clustering (IHTC).

We show, using simulations and applications of our algorithm to six large datasets, that IHTC combined with other clustering algorithms reduces the run time and memory usage of the original clustering algorithms while still preserving their performance. Additionally, we show IHTC also prevents singular data points from being overfit by desired clustering methods. Specifically, for m iterations of ITIS at size threshold t^*, IHTC ensures that each cluster contains at least $(t^*)^m$ units.

The rest of this chapter is organized as follows. A brief summary about clustering algorithms (k-means, HAC, TC) is given in section 2.2. Section 2.3 shows how to extend threshold clustering as an instance selection method and combine the iterated threshold instance selection method with other clustering methods. A simulation study is presented in section 2.4 and application of our methods on real datasets are presented in section 2.5. The last section discuss about our method.

2.2 Notation and Preliminaries

Consider a dataset with n units, numbered 1 through n. Each unit i has a response vector $\mathbf{y}_i$ and a d-dimensional covariate vector $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{id})$. For each pair of units i, j, the *dissimilarity* between i and j, denoted d_{ij}, can be computed. Often, the dissimilarity is chosen so that, if units i and j have similar values of covariates $\mathbf{x}$, then d_{ij} is small. We assume that $d_{ij} \geq 0$, and that dissimilarities satisfy the triangle inequality 1.1. Common choices of dissimilarities include Euclidean distance, Manhattan distance and average distance.

We define a *clustering* of a set of units as a partitioning of units such that units within each cluster of a partition have "small dissimilarity" and units belonging to two different clusters have "large dissimilarity." That is, at minimum, a clustering $\mathbf{v} = \{V_1, V_2, \ldots, V_m\}$ will satisfy the following properties:

1. **(Non-empty)**: $V_\ell \neq \emptyset$ for all $V_\ell \in \mathbf{v}$.

2. **(Spanning)**: For all units i, there exists a cluster $V_\ell \in \mathbf{v}$ such that $i \in V_\ell$.
3. **(Disjoint)**: For any two clusters $V_\ell, V_{\ell'} \in \mathbf{v}$, $V_\ell \cap V_{\ell'} = \emptyset$

The way of measuring "large" and "small" cluster dissimilarity will vary across clustering algorithms.

There are currently hundreds of available methods for clustering units. Moreover, some of these methods may be combined to construct additional *hybridized* clustering methods—our procedure for hybridizing is the major contribution of this chapter. For brevity, we apply our hybridizing procedure to two clustering methods—k-means and hierarchical clustering—with a note that this procedure may be applied to many other types of clustering. We now give a brief summary of these clustering methods.

2.2.1 K-means Clustering

The k-means clustering algorithm (Lloyd, 1982) is one of the most widely used and effective methods that attempts to partition units into exactly k-clusters.

The k-means clustering algorithm proceeds as follows:

1. **(Initialization)** Randomly select a set of k units (referred as *centers*) from the dataset. K denote the number of clusters and it should be pre-specified.
2. **(Assignment)** Assign all the units to the nearest center, based on squared Euclidean distance, to form k temporary clusters.
3. **(Updating)** Recompute the mean of each cluster. Replace the *centers* with the new k cluster means.
4. **(Terminate)** Repeat step 2 and 3 until there is no further change for the *centers*.

The time complexity for the k-means clustering algorithm is $O(nkLd)$ and the space complexity is $O((k+n)d)$ (Firdaus and Uddin, 2015; Hartigan and Wong, 1979) where d is the number of attributes for each unit, L is the number of iterations taken by the algorithm to converge.

The k-means algorithm suffers from a number of drawbacks (Hastie et al., 2009). First, there tends to be high sensitivity to the selection of initial units in Step 1. Additionally, it

tends to overfit isolated units leading to some clusters containing only a few units. Finally, the number of clusters k is fixed; if k is misspecified, k-means may perform poorly. In particular, many methods have been developed to mitigate problems due to initialization (Arthur and Vassilvitskii, 2007; Fränti and Kivijärvi, 2000; Fränti et al., 1997, 1998).

2.2.2 Hierarchical Agglomerative Clustering

Hierarchical agglomerative clustering (HAC) (Ward Jr, 1963) is a "bottom up" approach that aims to build a hierarchy clusters. It initially treats each unit as a cluster and then continues to merge two clusters together until only one cluster remains. HAC does not require a pre-specified number of clusters; Desired number of clusters can be obtained by using a *dendogram*—a tree that shows how the units are merged.

The HAC proceeds as follows:

1. **(Initial Clusters)** Start with n clusters, each cluster only contains one unit.
2. **(Merge)** Merge the closest (most similar) pair of clusters into a single cluster.
3. **(Updating)** Recompute the distance between the new cluster and the original clusters.
4. **(Terminate)** Repeat step 2 and 3 until one cluster remains, the cluster contains n units.

HAC requires linkage criteria to measure inter-cluster distance, but initialization and the choice of k is no longer a problem. However, the time complexity of HAC is $O(n^2 \log(n))$ (Kurita, 1991) and space complexity is $O(n^2)$ (Jain et al., 1999). This complexity limits its application to massive data. Another hindrance of HAC is that every merging decision is final. Once two clusters are merged into a new cluster, there is no way to partition the new cluster in later steps.

2.2.3 Threshold Clustering (TC)

Our hybridization method makes use of a recently developed clustering method called *threshold clustering* (TC). Initially this method was developed for performing statistical blocking of

massive experiments (Higgins et al., 2016). TC differs in two significant ways from traditional clustering approaches. First, TC does not fix the number of clusters formed, but instead, it ensures that each cluster contains a pre-specified number of units. Thus, TC is an effective way of obtaining many clusters, with each containing only a few units. Second, TC ensures the formation of a clustering with a small maximum within-group dissimilarity—more precisely, TC finds approximately optimal clustering with respect to a *bottleneck* objective—as opposed to an average or median within-group dissimilarity. The bottleneck objective is chosen not only to prevent largely dissimilar units from being grouped together, but also because these types of optimization problems often have approximate solutions that can be found efficiently (Hochbaum and Shmoys, 1986).

More precisely, let $\mathbf{B}(t^*)$ denote the set of all *threshold clusterings*—those clusterings $\mathbf{v}$ such that $|V_\ell| \geq t^*$ for each cluster $V_\ell \in \mathbf{v}$. The *bottleneck threshold partitioning problem* (BTPP) is to find the threshold clustering that minimizes the maximum within-cluster dissimilarity. That is, BTPP aims to find $\mathbf{v}^\dagger \in \mathbf{B}(t^*)$ satisfying:

$$\max_{V_\ell \in \mathbf{v}^\dagger} \max_{ij \in V_\ell} d_{ij} = \min_{\mathbf{v} \in \mathbf{B}(t^*)} \max_{V_\ell \in \mathbf{v}} \max_{ij \in V_\ell} d_{ij} \equiv \lambda; \tag{2.1}$$

here, λ is the optimal value of the maximum within-cluster dissimilarity.

It can be shown that BTPP is NP-hard, and in fact, no $(2-\epsilon)$–approximation algorithm for BTPP exists unless P = NP. However, Higgins et al. (2016) develops a threshold clustering algorithm (for clarity, the abbreviation TC refers specifically to this algorithm) to find a threshold clustering with maximum within-cluster dissimilarity at most 4λ. That is, TC is a 4–approximation algorithm for BTPP. The time and space requirement for TC are $O(t^*n)$ outside of the construction of a nearest neighbors graph (Higgins et al., 2016). Hence, TC may be used for large datasets, especially when the threshold t^* and the dimensionality of the covariate space are small.

TC uses graph theoretic ideas in its implementation. See Appendix C for graph theory definitions. TC with respect to a pre-specified minimum cluster size threshold t^* is performed as follows:

1. **(Construct nearest-neighbor subgraph)** Construct a $(t^* - 1)$-nearest-neighbors subgraph NG_{t^*-1} with respect to the dissimilarity measure d_{ij}.

2. (**Choose set of seeds**) Choose a set of units $\mathbf{S}$ such that

   (a) For any two distinct units $i, j \in \mathbf{S}$, there is no walk of length one or two in NG_{t^*-1} from i to j.

   (b) For any unit $i \notin \mathbf{S}$, there is a unit $j \in \mathbf{S}$ such that there exists a walk from i to j of length at most two in NG_{t^*-1}.

   Units in S are known as *seeds*.

3. (**Grow from seeds**) For each $\ell \in \mathbf{S}$, form a cluster of units V_ℓ^* comprised of unit ℓ and all units adjacent to ℓ in NG_{t^*-1}.

4. (**Assign remaining vertices**) Some units j may not be assigned to a cluster yet. These units are a walk of length two from at least one seed $\ell \in \mathbf{S}$ in NG_{t^*-1}. Assign the unassigned units to the cluster associated with seed ℓ. If there are several choices of seeds, choose the one that with the smallest dissimilarity $d_{\ell j}$.

The set of clusters $\mathbf{v} = \{V_\ell^*\}_{\ell \in \mathbf{S}}$ form a threshold clustering. Additionally, polynomial-time improvements to this algorithm—for example, in selecting cluster seeds or splitting large clusters—may improve the performance of TC without substantially increasing its runtime. An implementation of TC can be found in the R package `scclust` (Savje et al., 2017).

We construct a sample data to describe the four steps of threshold clustering algorithm which shown in Figure 2.1. In this problem we generate 20 units and with two covariates $(n = 20, d = 2)$, to make the algorithm visible and plot in $x - axis$ and $y - axis$. First of all, we treated each unit as vertex and use Euclidean distance to measure the dissimilarity between each pair of units. All the vertices are shown in (1.a). Then we defined threshold $t^* = 2$ and a $(t^* - 1)$-nearest-neighbors subgraph is obtained in (1.b). In (1.c), we found a set of seed $\mathbf{M}$ which represented as blue triangle, edges between the seeds and vertices adjacent to the seeds are highlighted. We formed clusters comprised by the seed and its

adjacent vertices, winch circled by red dash-line in (1.d). Remaining vertices are assigned to clusters which contain their closest seed and the final clusters can be seen in (1.e). Finally, the threshold clustering is comprised of 7 clusters and each cluster contains at least 2 units.

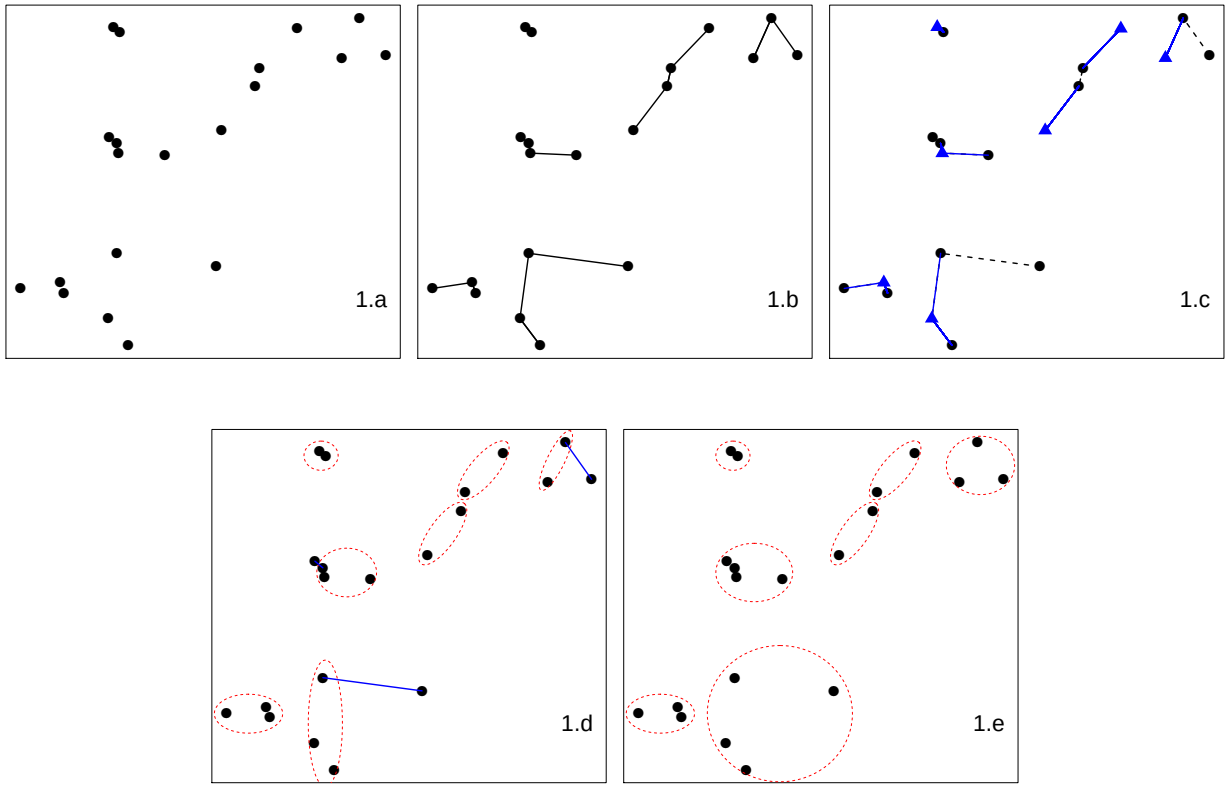


Figure 2.1: *An illustration of the threshold clustering algorithm described in Section 2.2.3 with* $n = 20$ *and* $t^* = 2$. *In (1.a), each unit is represented by a vertex. In (1.b), a* $(t^* - 1)$*-nearest-neighbors subgraph is obtained. Seeds are found in (1.c), represented by blue triangles. In (1.d) clusters are formed by grouping the cluster seed with its adjacent vertices. In (1.e), remaining vertices are assigned to clusters which contain their closest seed. The threshold clustering is comprised of 7 clusters, each with at least 2 units.*

2.3 Extension of Threshold Clustering

2.3.1 Threshold clustering as instance selection

Instance selection methods are used in massive data settings to efficiently scale down the size of the data (Leyva et al., 2015).

Common techniques for instance selection include subsampling (Pooja, 2013) and constructing *prototypes* (Plasencia-Calaña et al., 2014)—pseudo data points where each prototype represents a group of similar individual units. These methods tend to work quite well for massive data applications. Suppose the researcher desires to reduce the size of the data by a factor of α. We propose the following method for instance selection, which we call iterated threshold instance selection (ITIS):

1. **(Threshold clustering)** Perform threshold clustering with respect to a small size threshold t^* (e.g. $t^* = 2$) on the n data points to form n^* clusters, each containing t^* or more points.

2. **(Create prototypes)** Compute a center point for each of the n^* groups (e.g. centroid or medoid).

3. **(Terminate or continue)** If the data is reduced by a factor of α, stop. Otherwise, replace the n data points with the n^* centers and go back to Step 1.

An illustration of the ITIS procedure is given in Figure 2.2.

Instance selection may also be performed by choosing $t^* = \alpha$ and running one iteration of ITIS. However, the one iteration version does not seem to be as promising under massive data settings since the runtime of threshold clustering increases as t^* increases. See Appendix A for details.

Since the size of the data is reduced by a factor of t^* with each iteration, it follows that the computation for each subsequent iteration of ITIS decreases exponentially. Thus, the computational bottleneck of ITIS becomes the construction of a t^*–nearest-neighbors graph on all n units. This also suggests that the computation required of ITIS may be drastically

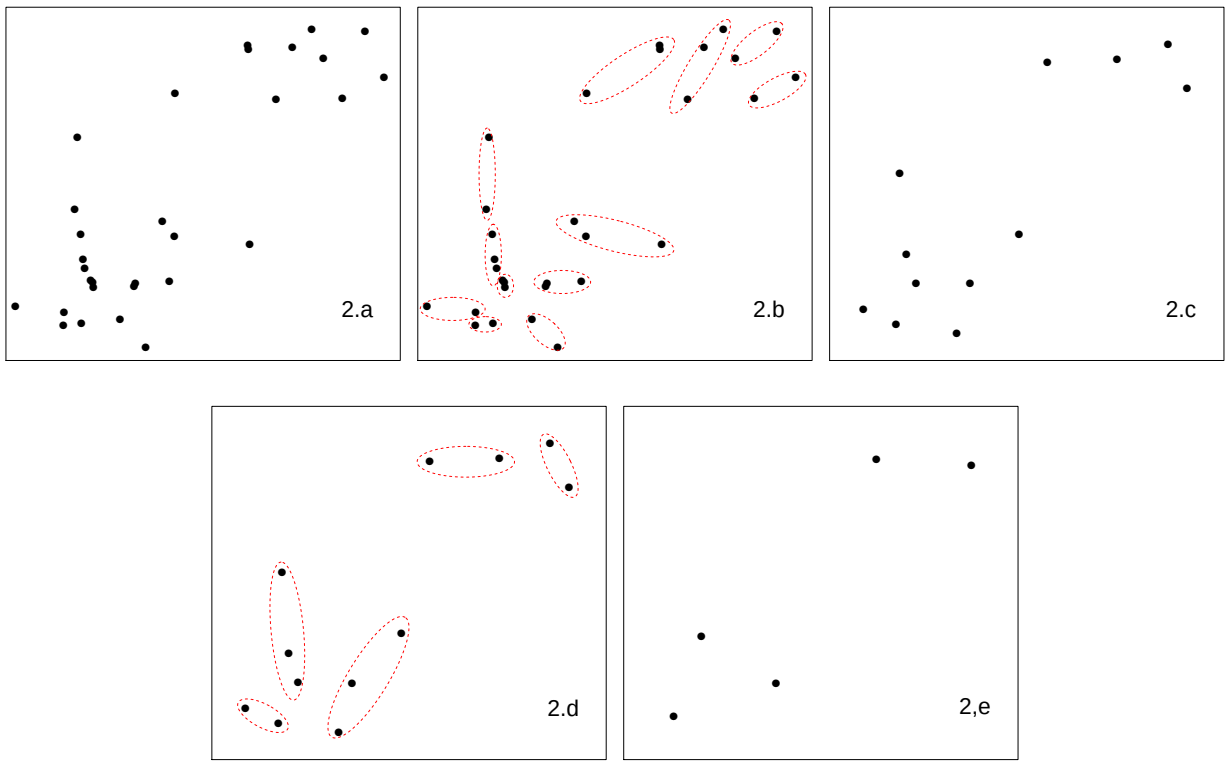


Figure 2.2: *An illustration of iterated threshold instance selection (ITIS) with threshod* $t^* = 2$ *on a sample of* $n = 30$ *data points drawn from a bivariate Gaussian mixture model. In (2.a), each unit is represented by a point. Applying threshold clustering on the data, 12 clusters are formed in (2.b). A prototype is formed for each of the 12 groups in (2.c). In (2.d), apply threshold clustering on the prototype, 5 clusters are formed. 5 prototypes are formed for the 5 clusters, the results are shown in (2.e). For this illustration, We perform two iterations of ITIS.*

improved through the discovery of methods for parallelization of threshold clustering. The iterative nature of ITIS does have a drawback; with each iteration, the prototype units become less similar to the units comprising the prototype. However, simulations suggest that this issue is not severe in massive data settings.

2.3.2 Iterative Hybridized Threshold Clustering

Often, researchers would like to use certain clustering methods (e.g. k-means, HAC, etc.) because of favorable or familiar properties of these clustering methods. However, under massive data settings, using such clustering techniques may not be feasible because of prohibitive

computational costs. We propose the following method for using ITIS as a pre-processing step on all n units to allow for the use of more sophisticated clustering methods. We call this procedure Iterative Hybridized Threshold Clustering (IHTC). It works as follows:

1. **(Create prototypes)** Perform iterative threshold instance selection with respect to a size threshold t^* on the n data points m times to form prototypes.

2. **(Cluster prototypes)** Cluster the prototypes (e.g. using k-means) obtained by 1.

3. **("Back out" assignment)** For each prototype, determine which of the original n units contributed to the formation of that prototype and assign these units to the cluster belonging to the prototype.

Figure 2.3 gives an illustration of IHTC with k-means.

IHTC reduces the size of the data by a factor of $(t^*)^m$, which can improve the efficiency of the original clustering algorithm. Additionally, IHTC also prevents overfitting of individual points, which may lead to a more effective clustering regardless with just a couple iterations of IHTC. Specifically, for m iterations of ITIS at size threshold t^*, IHTC ensures that each cluster contains at least $(t^*)^m$ units. Finally, we note that IHTC can be applied to most other clustering algorithms—not just k-means or HAC.

2.4 Simulation

We first demonstrate properties of IHTC using simulated data. We apply IHTC to samples of varying sizes where each data point is sampled from a mixture of three bivariate Gaussian distributions. Specifically, samples are drawn independently from a distribution with pdf:

$$f(\mathbf{x}) = 0.5p(\mathbf{x}|\mu_\mathbf{1}, \mathbf{\Sigma_1}) + 0.3p(\mathbf{x}|\mu_\mathbf{2}, \mathbf{\Sigma_2}) + 0.2p(\mathbf{x}|\mu_\mathbf{3}, \mathbf{\Sigma_3})$$

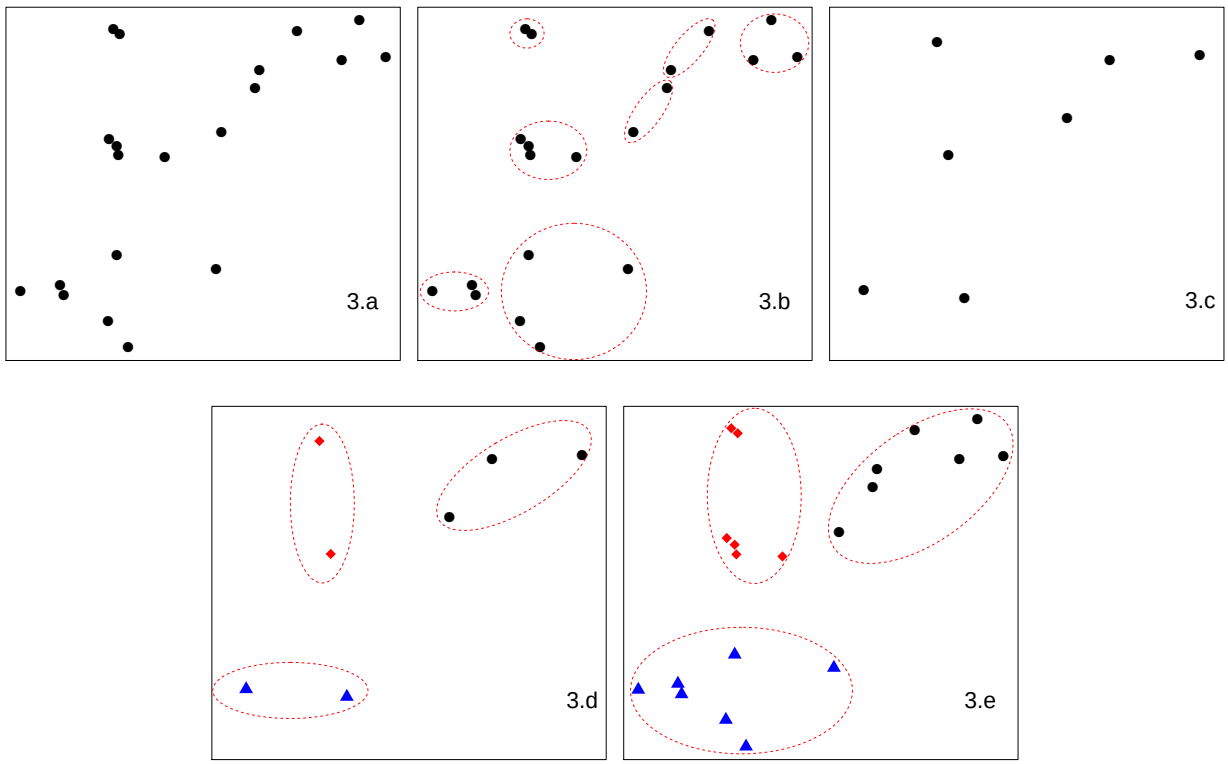


Figure 2.3: *An illustration of hybridized threshold clustering with k-means on bivariate data:* $n = 20, k = 3, t^* = 2$. *In (3.a), each unit is considered as a vertex and the dissimilarity between every pair units is computed. Applied threshold clustering with* $t^* = 2$ *on the 30 units, 7 groups are formed in (3.b). For each of the group, generate one prototype, 7 prototypes obtained in (3.c). Then apply k-means clustering on these 7 prototypes, 3 clusters obtained in (3.d). Units that contributed to form a prototype should be assign to the cluster which the prototype belongs to. The final clustering results are in (3.e).*

where $p(\mathbf{x}|\mu_\mathbf{j}, \mathbf{\Sigma_j})$ is the pdf of a Gaussian with parameters $\mu_\mathbf{j}$ and $\mathbf{\Sigma_j}$, $j = 1, 2, 3$,

$$\mu_\mathbf{1} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mu_\mathbf{2} = \begin{bmatrix} 7 \\ 8 \end{bmatrix}, \mu_\mathbf{3} = \begin{bmatrix} 3 \\ 5 \end{bmatrix},$$

$$\mathbf{\Sigma_1} = \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix}, \mathbf{\Sigma_2} = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{\Sigma_3} = \begin{bmatrix} 3 & 0 \\ 0 & 4 \end{bmatrix}.$$

We use Euclidean distance as our dissimilarity measure. The data size n varies between 10^4 and 10^8 observations and each setting is replicated 1000 times. For each simulation, we

record the run time in seconds and memory usage in megabytes for the whole procedure.

Algorithms are implemented in the R programming language. We use the `scclust` package to perform threshold clustering (Savje et al., 2017). The default `kmeans` and `hclust` functions in R are used for k-means and hierarchical agglomerative clustering respectively. This simulation was performed on the Beocat Research Cluster at Kansas State University, which is funded in part by NSF grants CNS-1006860, EPS-1006860, and EPS-0919443 (University, 2018). We perform simulations on a single core Intel Xeon E5-2680 v3 with 2.5 GHz processor with 30 GM of RAM at maximum.

2.4.1 IHTC with K-means Clustering

We perform IHTC with k-means with threshold $t^* = 2$ and number of clusters $k = 3$. The run time and memory usage are in Figure 2.4 and Table 2.1. Because we simulated data from a Gaussian mixture model, each cluster roughly should correspond to a different Gaussian distribution. Hence, we can use prediction accuracy to measure the performance of our methods. The prediction accuracy is the number of units correct clustered divided by data size n. The prediction accuracy for IHTC with k-means are in Figure 2.5 and Table 2.1. The first point of each curve (iteration $m = 0$) indicates the performance without pre-processing of the data; that is $m = 0$ indicates where only the original clustering algorithm is applied to the data.

From Figures 2.4 and 2.5, and from Table 2.1, we find that using IHTC with k-means decreases the runtime and memory required compared to without IHTC. After one iteration, the runtime and memory usage decreases by about half while the prediction accuracy remains the same. As the number of iterations increases, the additional improvements in runtime and memory usage decrease. After several iterations, runtime and memory usage tend to level off and prediction accuracy slowly decreases.

Iteration	Run Time (second)					Memory (Mb)					Accuracy				
(m)	10^4	10^5	10^6	10^7	10^8	10^4	10^5	10^6	10^7	10^8	10^4	10^5	10^6	10^7	10^8
0	0.143	1.613	18.773	218.43	2815	19.39	241.64	2556	27540	279346	0.9236	0.9239	0.9239	0.9239	0.9239
1	0.084	0.909	10.337	119.86	1767	8.70	99.14	1019	11097	110467	0.9236	0.9239	0.9239	0.9239	0.9239
2	0.072	0.647	7.886	97.07	1572	3.99	44.00	488	5462	54773	0.9232	0.9238	0.9239	0.9239	0.9239
3	0.058	0.550	6.975	88.52	1522	1.76	23.55	253	3104	31764	0.9225	0.9237	0.9239	0.9239	0.9239
4	0.053	0.502	6.534	83.86	1487	0.97	14.19	166	2150	21962	0.9214	0.9234	0.9238	0.9239	0.9239
5	0.053	0.497	6.332	81.46	1378	0.81	8.62	130	1761	17757	0.9187	0.9229	0.9238	0.9239	0.9239
6	0.051	0.487	6.272	80.90	1350	0.81	7.94	112	1614	16478	0.9128	0.9216	0.9235	0.9239	0.9239
7	-	0.487	6.263	80.62	1336	-	7.69	106	1560	15813	-	0.9196	0.9231	0.9238	0.9239
8	-	0.490	6.254	80.28	1305	-	7.56	105	1561	16005	-	0.9163	0.9224	0.9236	0.9239
9	-	0.490	6.243	80	1288	-	7.91	105	1574	16099	-	0.9085	0.9210	0.9234	0.9239
10	-	-	6.245	79.95	1252	-	-	106	1596	16512	-	-	0.9184	0.9227	0.9237
11	-	-	6.246	79.75	1268	-	-	109	1630	16587	-	-	0.9140	0.9218	0.9235
12	-	-	-	79.72	1247	-	-	-	1662	17036	-	-	-	0.9201	0.9233

Table 2.1: *Cluster performance of IHTC with K-means ($k = 3, t^* = 2$).*

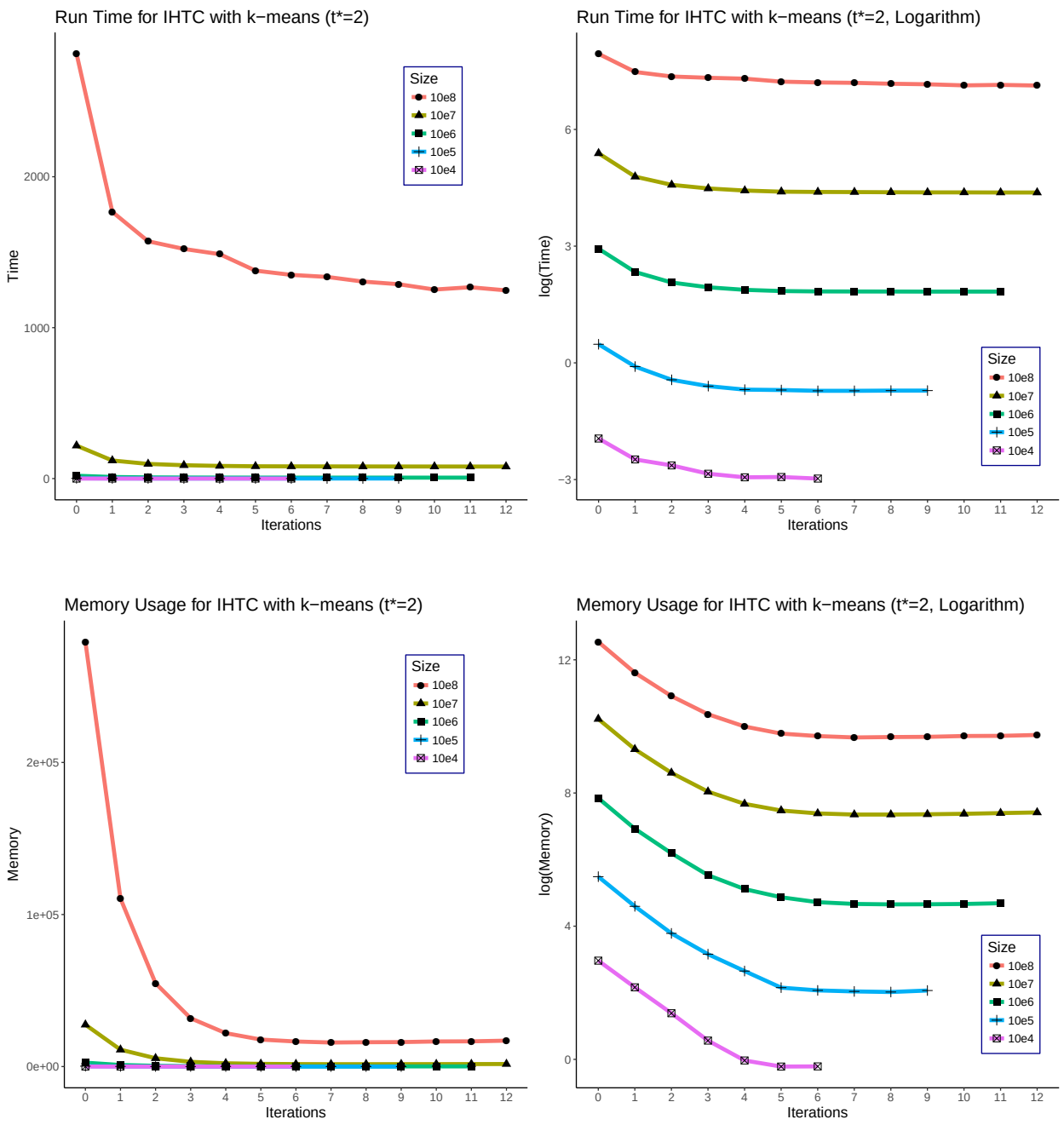


Figure 2.4: *Run time and memory usage of IHTC with k-means clustering* ($k = 3, t^* = 2$).

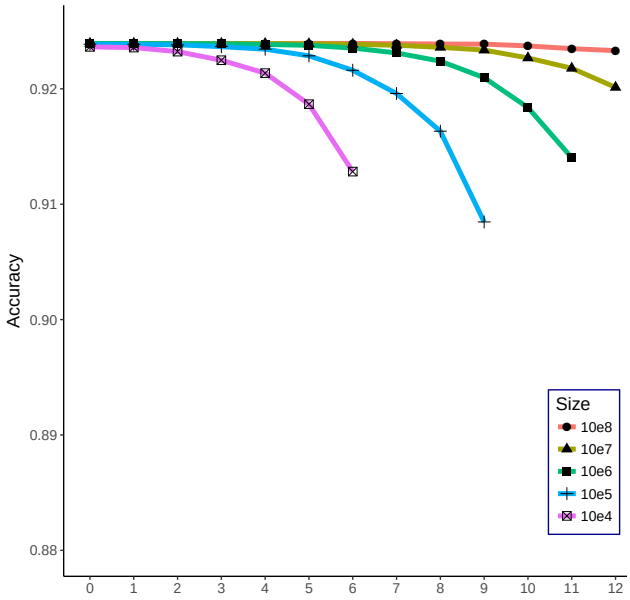


Figure 2.5: *Prediction accuracy of IHTC with k-means clustering ($k = 3, t^* = 2$).*

2.4.2 IHTC with HAC

HAC is an computationally expensive algorithm. For example, if data size exceed 65,536 datapoints, the `hclust` function in R will throw an error. We applied IHTC with HAC and consider with respect to threshold $t^* = 2$. Comparing the performance with pre-processing and without pre-processing, we found the reduction for runtime and memory requirement is dramatic when we apply IHTC with HAC, while prediction accuracy falls slightly. As the number iterations increase, the improvements in runtime and memory usage decrease, and after several iterations, runtime and memory tend to a certain stable value and prediction accuracy decreases. Thus, the number of iterations to perform is an unsolved problem. Figures 2.6 and 2.7, and Table 2.2 give the results of IHTC with HAC.

Iterations	Run Time (second)					Memory (Mb)					Accuracy				
m	10^4	10^5	10^6	10^7	10^8	10^4	10^5	10^6	10^7	10^8	10^4	10^5	10^6	10^7	10^8
Null	5.425	-	-	-	-	796.46	-	-	-	-	0.9122	-	-	-	-
1	0.940	88.619	-	-	-	177.27	20193.0	-	-	-	0.9142	0.9143	-	-	-
2	0.220	15.755	-	-	-	20.49	3537.4	-	-	-	0.9121	0.9129	-	-	-
3	0.073	2.992	-	-	-	7.01	459.9	-	-	-	0.9079	0.9132	-	-	-
4	0.048	0.752	62.28	-	-	1.11	117.9	10891	-	-	0.9091	0.9123	0.9126	-	-
5	0.044	0.393	15.41	-	-	0.73	28.7	1940	-	-	0.9012	0.9106	0.9124	-	-
6	0.045	0.350	7.44	-	-	0.74	11.2	435	-	-	0.8938	0.9075	0.9134	-	-
7	0.044	0.343	6.21	-	-	0.74	9.3	165	-	-	0.8864	0.9029	0.9121	-	-
8	-	0.342	5.99	93.8	-	-	9.3	120	2405	-	-	0.8984	0.9086	0.9137	-
9	-	0.342	5.98	90.4	-	-	9.4	115	1648	-	-	0.8896	0.9067	0.9123	-
10	-	-	5.99	89.2	1299	-	-	118	1564	19113	-	-	0.9012	0.9101	0.9159
11	-	-	6.04	89.7	1267	-	-	123	1523	16932	-	-	0.8949	0.9089	0.9139
12	-	-	-	90.1	1288	-	-	-	1579	16760	-	-	-	0.9066	0.9131
13	-	-	-	89.9	1253	-	-	-	1604	17041	-	-	-	0.8991	0.9109
14	-	-	-	-	1272	-	-	-	-	17505	-	-	-	-	0.9068
15	-	-	-	-	1283	-	-	-	-	17784	-	-	-	-	0.8987
16	-	-	-	-	1288	-	-	-	-	18220	-	-	-	-	0.8950

Table 2.2: *Cluster performance for IHTC with HAC ($t^* = 2$)*

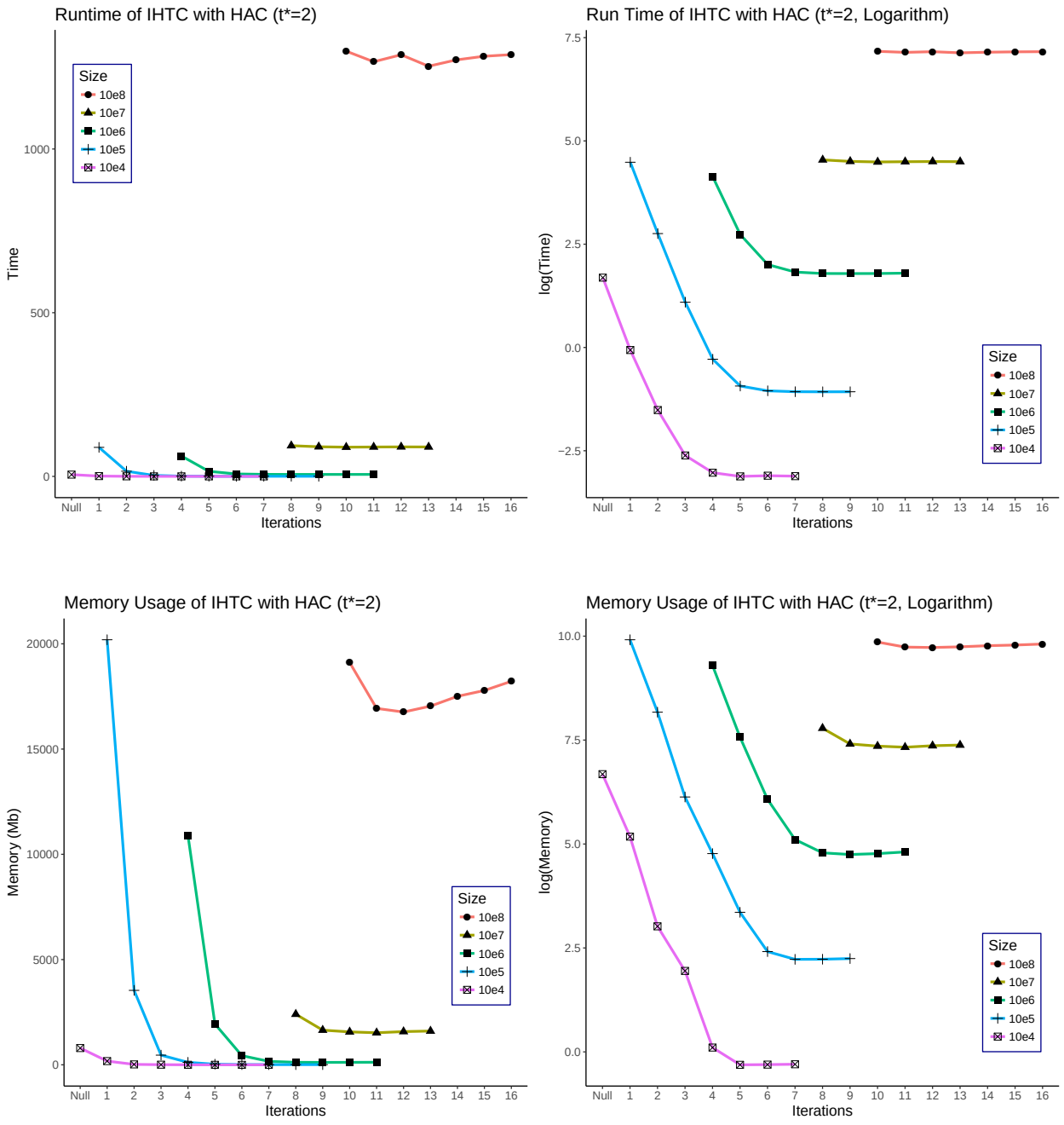


Figure 2.6: *Run time and memory usage of IHTC with HAC ($t^* = 2$).*

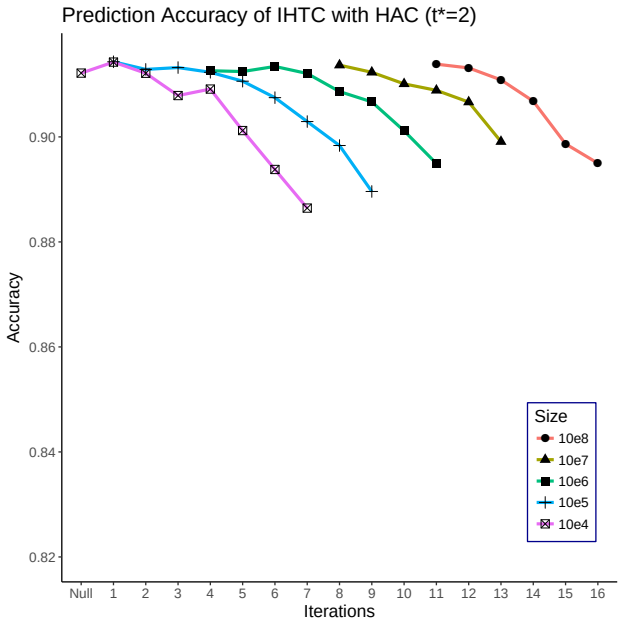


Figure 2.7: *Prediction accuracy of IHTC with HAC ($k = 3, t^* = 2$).*

2.5 Experiments

In this section, we will use our threshold clustering algorithm performance on several publicly available datasets. A brief description of each dataset can be found in Table 2.3. The Euclidean distance is used to measure the dissimilarity and principal component analysis (Friedman et al., 2001; Hotelling, 1933) is used for each dataset to perform feature selection. The number of classes (k) is chosen by the elbow of plot of within-cluster sum of squared distances for different k. All experiments run on a single CPU core laptop.

Name	Instances	Attributes	Classes
PM 2.5 (Havera, 2017)	41757	5	4
Credit Score (Kaggle, 2011)	120269	6	5
Black Friday (Dagdoug, 2018)	166986	7	4
Covertype (Blackard and Dean, 1999)	581012	6	7
House Price (Zillow, 2017)	2885485	5	5
Stock (Brogaard et al., 2014; Carrion, 2013)	7026593	5	7

Table 2.3: *Data description.*

Tables 2.4, 2.5, and 2.6, and Figures 2.8 and 2.9 demonstrate results of IHTC with k-

means and HAC for these datasets. Iteration $m = 0$ indicates the performance when no pre-processing was conducted; only k-means clustering was applied to the data. BSS/TSS is the ratio of between cluster sum of squares and total sum of squares. Larger ratio value indicates better cluster performance. The number of prototypes is the size of the reduced data after m iterations. We found that using IHTC with k-means or HAC decreases the runtime and memory required compared to without IHTC. After one iteration, the runtime and memory usage decreases by about half while preserving the value of the BSS/TSS ratio. As the number of iterations becomes large, the improvements in runtime and memory usage decrease and the BSS/TSS ratio decreases slowly.

To demonstrate the versitility of IHTC, we also consider its performance with the clustering method DBSCAN (Ester et al., 1996). The results of DBSCAN are contained in Appendix B.

Name	Run Time (second)				Memory Usage (Mb)				BSS/TSS				Number of Prototypes			
	$m=0$	$m=1$	$m=2$	$m=3$	$m=0$	$m=1$	$m=2$	$m=3$	$m=0$	$m=1$	$m=2$	$m=3$	$m=0$	$m=1$	$m=2$	$m=3$
PM 2.5	0.636	0.282	0.232	0.268	71.28	25.69	3.7	5.91	0.5347	0.5346	0.5345	0.5344	41757	17281	7166	2984
Credit Score	2.902	2.046	2.696	2.904	224.55	23.9	13.95	17.65	0.5187	0.5184	0.5178	0.5169	120269	49669	20471	8456
Black Fridy	2.802	0.468	0.522	0.554	317.05	32.95	24.5	28.91	0.3493	0.3456	0.3402	0.3226	166986	11868	4914	2017
Covertype	22.244	10.184	11.968	13.562	1073.9	387.66	150.46	172.78	0.4791	0.4806	0.4741	0.4787	581012	241072	99509	41102
House Price	110.08	40.24	58.02	65.05	5178.7	881.2	726.4	859.4	0.5589	0.5589	0.5589	0.5587	2885485	1196674	496442	206332
Stock	262	121.82	105.98	127.62	12528.9	4545.9	1943.8	2169.9	0.5829	0.5828	0.5825	0.5820	7026593	2952376	1226666	508366

Table 2.4: *Cluster performance for IHTC with k-means ($t^* = 2$).*

Performance	PM 2.5			Credit Score			Black Friday		
	$m=1$	$m=2$	$m=3$	$m=2$	$m=3$	$m=4$	$m=1$	$m=2$	$m=3$
Run Time (second)	11.7	1.9	0.48	18.87	3.79	3.07	5.66	1.01	0.60
Memory Usage (Mb)	3420.6	588.9	79.7	4799.3	730.98	21.32	1618.94	277.87	38.44
BSS/TSS	0.4964	0.4964	0.4964	0.4746	0.4612	0.4613	0.3176	0.3024	0.3142
Number of Prototypes	17281	7166	2984	20471	8456	3508	11868	4914	2017

Table 2.5: *Cluster performance for IHTC with HAC ($t^* = 2$).*

Performance	Covertype			House Price			Stock		
	$m = 4$	$m = 5$	$m = 6$	$m = 6$	$m = 7$	$m = 8$	$m = 7$	$m = 8$	$m = 9$
Run Time (second)	16.34	15.96	15.72	63.3	65.1	64.4	144.24	144.84	145.12
Memory Usage (Mb)	206.53	211.25	210.1	940.74	934.97	933.79	2415.6	2443.9	2471.9
BSS/TSS	0.4124	0.3982	0.4144	0.5213	0.5017	0.5017	0.4986	0.4945	0.4993
Number of Prototypes	17015	7015	2911	15014	6268	2598	15085	6267	2603

Table 2.6: *Cluster performance for IHTC with HAC ($t^* = 2$).*

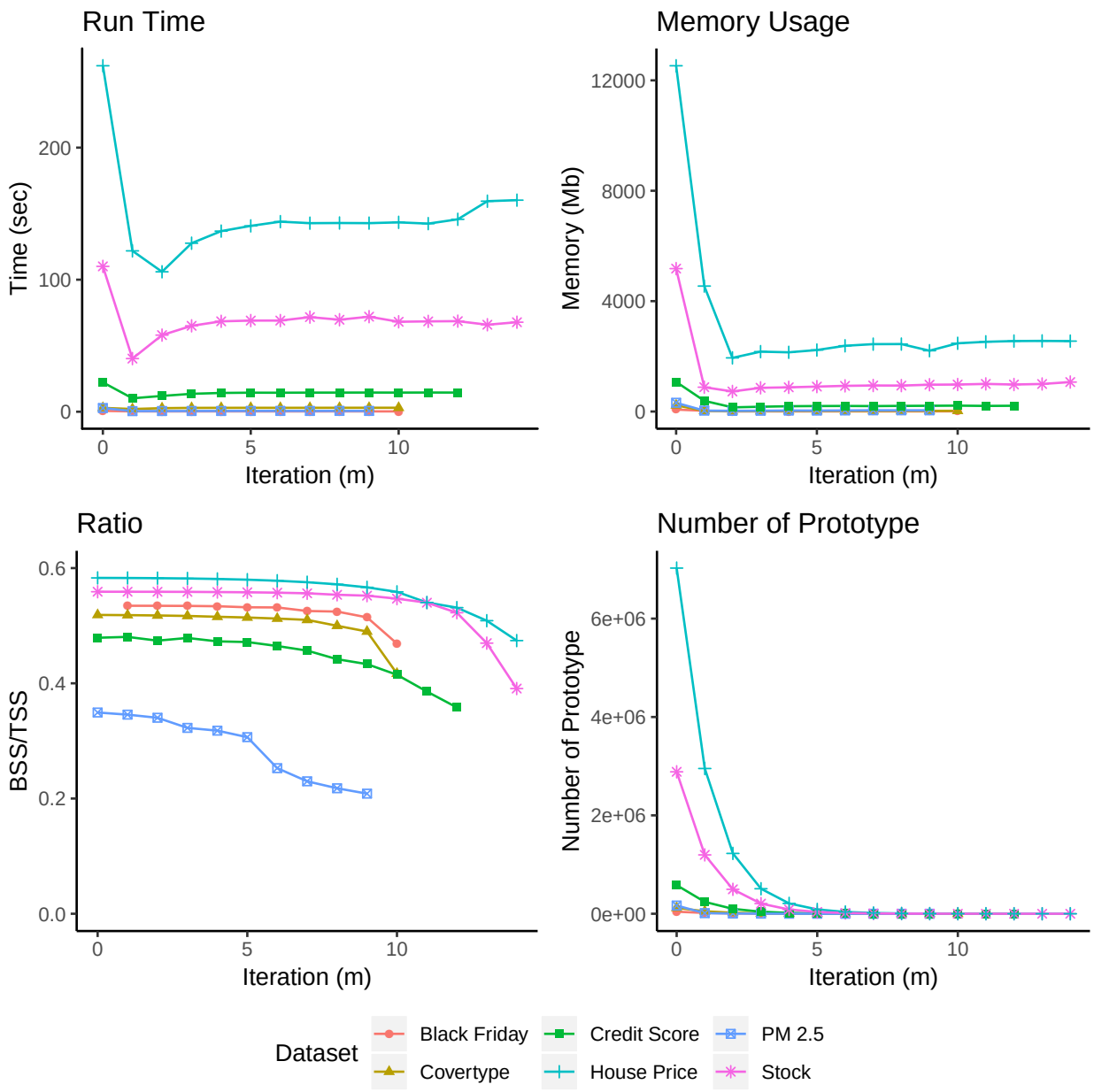


Figure 2.8: *Cluster performance for IHTC with K-means.*

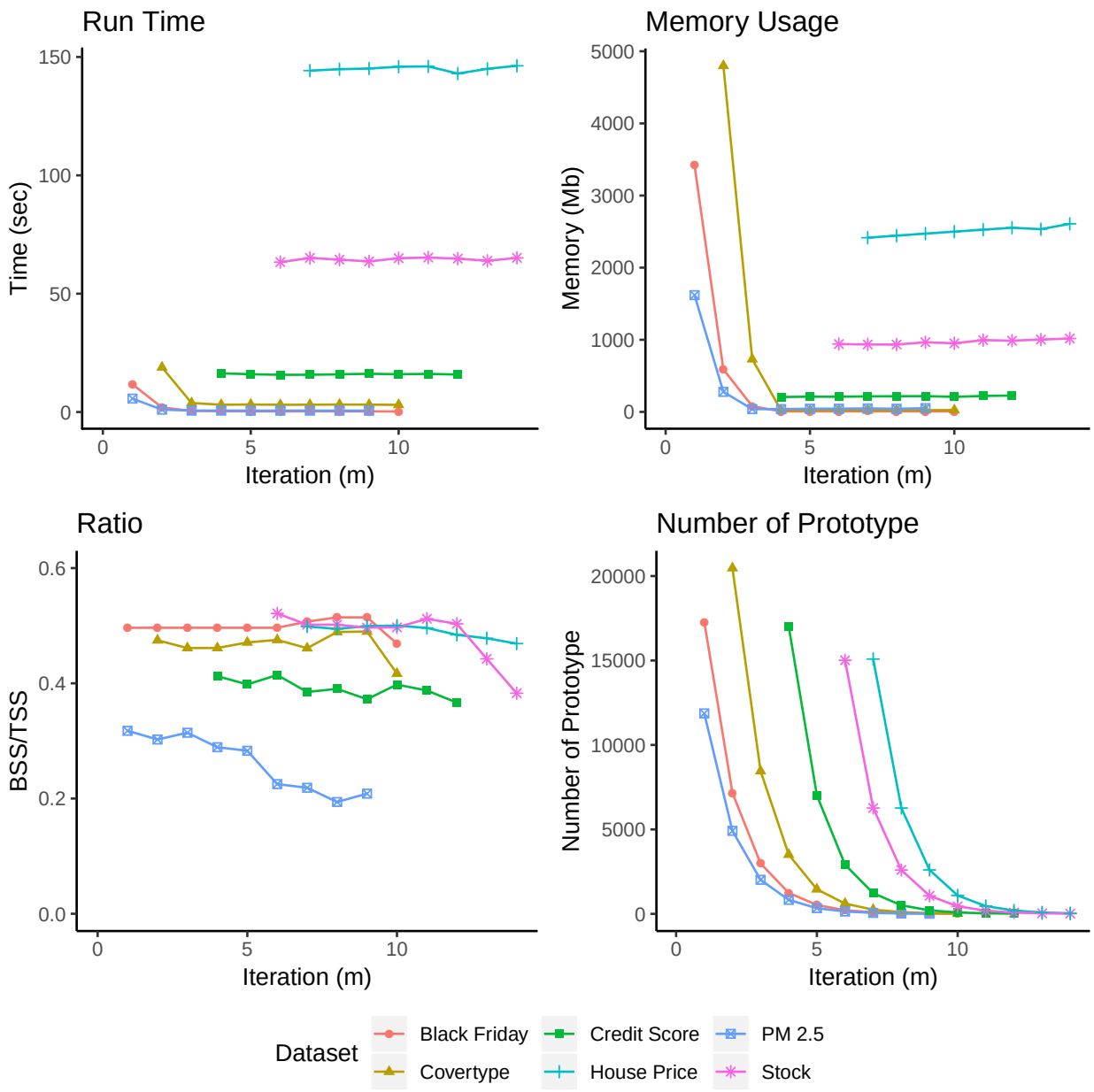


Figure 2.9: *Cluster performance for IHTC with HAC.*

2.6 Discussion

Many clustering methods share one ubiquitous drawback: as the size of the data becomes massive—that is, as the number of units n for the data becomes large—these methods become intractable. Even efficient implementations of these algorithms require prohibitive

computational resources under massive n settings. Recently, threshold clustering (TC) has been developed as a way of clustering units so that each cluster contains a pre-specified number of units and so that the maximum within-cluster dissimilarity is small. Unlike many clustering methods, TC is designed to form clustering comprised of many groups of only a few units. Moreover, the runtime and memory requirement for TC is smaller compared to other clustering methods.

In this chapter, we propose using TC as an instance selection method called iterative threshold instance selection (ITIS) to efficiently and effectively reduce the size of data. Additionally, we propose coupling ITIS with other, more sophisticated clustering methods to obtain method to sufficiently scale down the size of data. and introduced IHTC that can efficiently and effectively scale down the size of data so that more sophisticated clustering methods can be applied on the reduced data.

Simulation results and application on real datasets show that implementing clustering methods with IHTC may decrease their runtime and memory usage without sacrificing the their performance. For more sophisticated clustering methods, this reduction in runtime and memory usage may be dramatic. Even for the standard implementation of k-means in R, one iteration of IHTC decreases the runtime and memory usage by more than half while maintaining clustering performance.

Chapter 3

Hierarchical Agglomerative Threshold Clustering for Massive Data

3.1 Introduction

Hierarchical agglomerative clustering (HAC) is a "bottom up" approach that initially views each data point as a cluster and then continues to merge two clusters together until there is one cluster left. The merges of HAC are determined in a greedy way (Horowitz and Sahni, 1979). The time and space complexity limits its application to massive data.

In Chapter 2 we propose TC for instance selection that can efficiently and effectively scale down the size of data so that more sophisticated clustering methods can be applied on the reduced data. This chapter we would extend TC to hierarchical clustering. We consider two methods which works as follows. The first method, called hierarchical agglomerative threshold clustering (HATC), follows the hierarchical agglomerative clustering (HAC) paradigm closely. First, TC is performed on the n units to form several clusters, each containing t^* or more units. Then, the cluster with the smallest within-cluster dissimilarity is condensed into a single point, and TC is again performed on the reduced data. These steps are repeated until there are fewer than t^* data points. The second method repeatedly performs ITIS to form a cluster hierarchy. Initially, TC is performed on the n units. Then, a prototype is

formed for each cluster, and TC is performed again on these prototypes. As before, these steps are repeated until there are fewer than t^* points remaining.

Using simulation, we show that our methods reduces the run time and memory usage of the other clustering algorithms while still preserving their performance. Additionally, we show HATC prevents singular data points from being overfit by desired clustering methods.

The rest of this chapter is organized as follows. Section 3.2 shows how to extend threshold clustering to hierarchical agglomerative threshold clustering. Repeated performs ITIS to form a cluster hierarchy is given in section 3.3 A simulation study is presented in section 3.4. The last section discuss about our method.

3.2 Hierarchical Agglomerative Threshold Clustering (HATC)

The HATC is a bottom-up clustering method where each cluster has sub-clusters. It start with every unit in a cluster, then merges some clusters by satisfying some criteria in each successive iteration, it will terminate when all of the units are in one cluster.

The HATC works as follows. First, TC is performed on the n units to form several clusters, each containing t^* or more units. Then we compute the within-cluster distances. Suppose the cluster which has smallest within-cluster distance contains n^* units, we form a prototype for this cluster, discard these n^* units. The reduced dataset now contains $n-n^*+1$ units. Finally, if the data size smaller or equal to t^*, TC is applied again to the $n - n^* + 1$ units. Otherwise, stop.

1. **(Threshold Clustering)** Perform threshold clustering with respect to a size threshold t^* on the n data points to form several clusters.

2. **(Compute Within-cluster Distance)** Compute within-cluster distance for each cluster.

3. **(Condense Cluster)** Suppose the cluster which has smallest within-cluster distance

contains n^* units, compute a center point for these n^* units (e.g. centroid or medoid), replace the n data points with the $n - n^* + 1$ units.

4. **(Terminate or continue)** Repeat step 1 to step 3 until data size no greater than t^*.

Figure 3.1 gives an illustration of HATC.

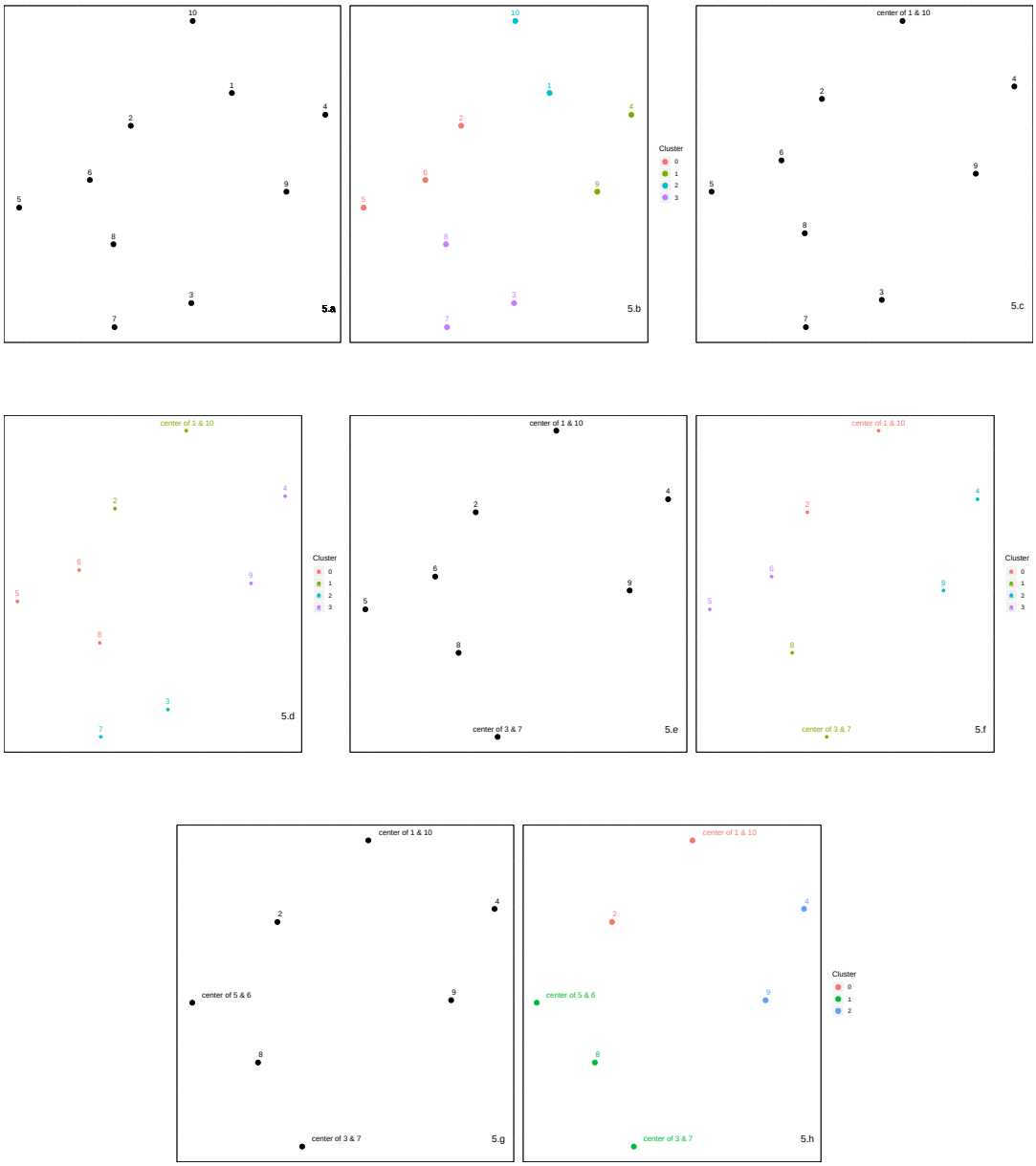


Figure 3.1: *An illustration of HATC on bivariate data:* $n = 10, t^* = 2$. *In graph (5.a), each unit is represented by a point. Applying threshold clustering on the data, 4 clusters are formed in (5.b). A prototype is formed for the cluster which has smallest within-cluster distance in (5.c). In (5.d), threshold clustering is performed again on the rest of points and 4 clusters are formed. The second prototype is formed for the cluster which has smallest within-cluster distance in (5.e). Apply TC on these 8 units to form 4 clusters in (5.f). Condense unit 5 and 6 as a new prototype in (5.g). The clustering result after 3 iterations of HATC are given in (5.h). These procedure will stop until only 3 units left.*

3.3 Cluster Hierarchy

We use TC as a clustering technique on massive datasets. In this case, for each iteration, the size of data will reduce by a factor of t^* and no other clustering algorithms are required. We propose the following method:

1. **(Threshold clustering)** Perform threshold clustering with respect to a size threshold t^* on n units, n^* clusters are formed.

2. **(Create prototypes)** Compute a center point for each of the n^* groups (e.g. centroid or medoid).

3. **(Terminate or continue)** If number of prototypes is no greater than t^*, go to Step 4. Otherwise, replace the n data points with the n^* centers and go back to Step 1.

4. **("Back out" assignment)** For each prototype, determine which of the original n units contributed to the formation of that prototype and assign these units to the cluster belonging to the prototype.

Figure 3.2 gives an illustration of this method.

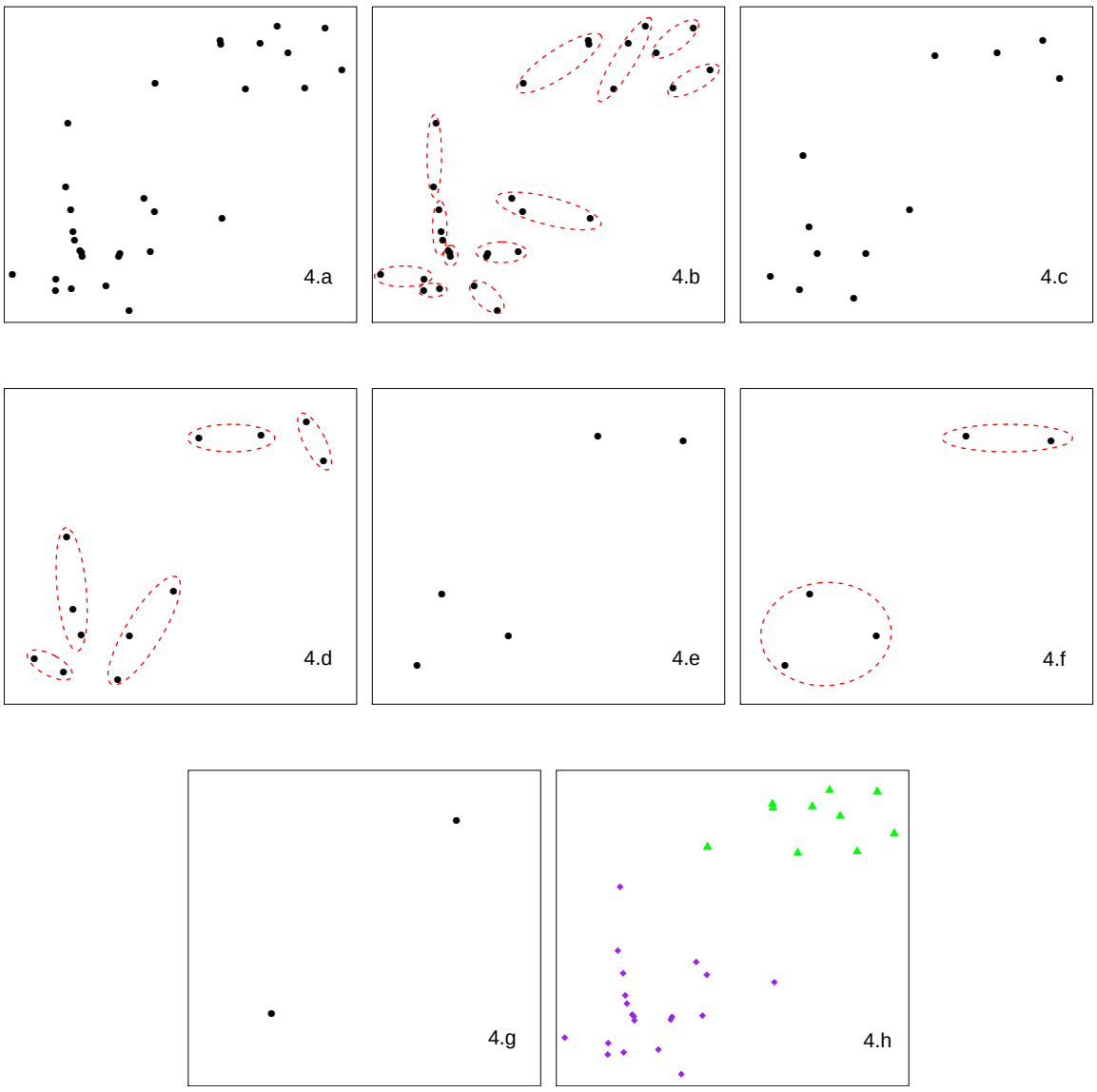


Figure 3.2: *An illustration of forming cluster hierarchy with ITIS) on bivariate data:* $n = 30, t^* = 2$. *In graph (4.a), each unit is represented by a point. Applying threshold clustering on the data, 12 clusters are formed in (4.b). A prototype is formed for each of the 12 groups in (4.c). In (4.d), threshold clustering is performed again on the prototypes and 5 clusters are formed. 5 prototype is formed in (4.e). Apply TC on these 5 prototypes to form 2 clusters in (4.f). Condense 2 clusters into 2 prototypes in (4.g). If desired number of cluster is 2, the "backed out" final clustering results are given in (4.h).*

Specifically, for m iterations of TC at size threshold t^*, this method ensures that each cluster contains at least $(t^*)^m$ units. This method prevents overfitting of individual points, which may lead to a more effective clustering regardless with just a couple iterations of TC. Besides this, this is an efficient clustering algorithm that can work with massive data on a laptop.

3.4 Simulation Study

We demonstrate properties of both methods using simulated data. The data is simulated with the same distribution as Section 2.4 introduced and we use Euclidean distance as our dissimilarity measure. Data size n varies between 10^3 and 10^5 observations and each setting is replicated 100 times. For each simulation, we record the run time in seconds and memory usage in megabytes for the whole procedure.

Algorithms are implemented in the R programming language: We use the SCCLUST (Savje et al., 2017) package to perform threshold clustering. The simulation was performed on a laptop with 2.6 Ghz processor and 16 GB of RAM.

The cluster performance in Table 3.1.

Method	Run Time (second)			Memory (Mb)			BSS/TSS			Accuracy		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
HAC	0.04	3.12	-	7.7	764	-	0.8322	0.8386	-	0.914	0.9281	-
Cluster Hierarchy	0.02	0.048	0.504	1.8	5.62	12.12	0.9037	0.8592	0.85	0.891	0.8901	0.8736
HATC	18	116		196	1058		0.987	0.988		0.9105	0.9012	

Table 3.1: *Cluster performance of our method and HAC ($t^* = 2$).*

Because we simulated data from a Gaussian mixture model, each cluster roughly should correspond to a different Gaussian distribution. Hence, we can use prediction accuracy to measure the performance of our methods. The prediction accuracy is the number of units correct clustered divided by data size n.

From Table 3.1, we find that performs ITIS to form a cluster hierarchy decrease the run-time and memory required compared to HAC. The ratio of between-cluster sum of square and total sum of square (BSS/TSS) for HATC is higher than the ratio for HAC. The pre-

diction accuracy for HATC is a little lower than that for HAC but is acceptable. HATC has longer runtime and higher accuracy than performs ITIS to form a cluster hierarchy.

3.5 Discussion

As an important tool of data clustering, HAC provides a structure that is informative and easy to decide the number of clusters with dendrogram. However, the computation complexity limits its application on massive dataset.

This chapter introduced two hierarchical clustering methods that reduce runtime and memory requirement while maintain cluster performance. HATC has better cluster performance than performs ITIS to form a cluster hierarchy but it has longer runtime. We may improve the implementation of HATC in the future to make them work better.

Chapter 4

Tighter Bounds on Approximation

4.1 Introduction

The bottleneck objective framework is often amenable to making claims about the availability of polynomial-time approximation algorithm (Hochbaum and Shmoys, 1986). In the case of threshold blocking, it can be shown, when $t \geq 3$, that no $(2-\epsilon)$-approximation algorithm for bottleneck threshold partitioning problem (BTPP) can terminate in polynomial time unless $P = NP$ (Higgins et al., 2016). However, threshold clustering which introduced in Higgins et al. (2016)'s paper only gives a 4-approximation algorithm. Thus, there is a gap in understanding on how well an algorithm can approximate BTPP in polynomial time. In some situations, it may be possible that to obtain an improved polynomial-time algorithm for BTPP with an approximation factor less than 4.

In this paper, we fill in some gaps for the relationship between α-approximation and t. We show that when $t = 2$, there is a polynomial-time $2-$approximation algorithm exists and when $t = 3$, a polynomial-time $3-$approximation algorithm exists. Table 4.1 provides information about what we found in this chapter and which part needs to be solved in the future.

The rest of this chapter is organized as follows. A brief set-up about graph theoretic framework is given in section 4.2. A polynomial-time $2-$approximation algorithm when

$t = 2$ is presented in section 4.3. Section 4.4 shows how to construct a polynomial-time 3−approximation algorithm. The last section discuss about future work.

<table>
<tr><td></td><td>$t = 2$</td><td>$t = 3$</td><td>$t = 4$</td><td>$t > 4$</td></tr>
<tr><td>$(2 - \epsilon)$-approximation</td><td>?</td><td colspan="3">NP-hard</td></tr>
<tr><td>2-approximation</td><td rowspan="2">Polynomial time</td><td>?</td><td rowspan="2" colspan="2">?</td></tr>
<tr><td>3-approximation</td><td>Polynomial time</td></tr>
<tr><td>4-approximation</td><td colspan="4">Polynomial time</td></tr>
</table>

Table 4.1: *Relationship between $\alpha-$approximation and t. Bold entries indicate results proven in this paper.*

4.2 Graph Theoretic Framework

Let $G = (V, E)$ denote a graph, where V is a set of n vertices and E is a set of edges. We suppose for now that G is *undirected* and *complete*—between any two distinct vertices $i, j \in V$, there is exactly one undirected edge $ij \in E$ joining these two vertices. Hence, E contains $n(n-1)/2$ edges. Suppose each edge ij has a weight w_{ij}, and suppose these weights satisfy the triangle inequality:

$$\forall\, ij, j\ell, i\ell \in E,\ w_{ij} + w_{j\ell} \geq w_{i\ell}. \tag{4.1}$$

Note that all distance metrics satisfy the triangle inequality by definition.

Definition 4.2.1. *A partition of V is a separation of V into non-empty blocks of vertices such that each vertex $i \in V$ belongs to exactly one block. Formally, a partition is a collection of sets of vertices $\mathbf{p} = \{V_1, V_2, \ldots, V_m\}$ satisfying:*

1. $\emptyset \neq V_\ell \subseteq V$ *for all* $V_\ell \in \mathbf{p}$.
2. $V_\ell \cap V_{\ell'} = \emptyset$ *for all* $V_\ell, V_{\ell'} \in \mathbf{p}, V_\ell \neq V_{\ell'}$.
3. $\bigcup_{\ell=1}^{m} V_\ell = V$.

Let $\mathbf{P}$ *denote the set of all partitions.*

Definition 4.2.2. *The size of a block $V_\ell \in \mathbf{p}$, denoted $|V_\ell|$, is the number of vertices contained in that block.*

Definition 4.2.3. *For a size threshold t, a partition $\mathbf{p} \in \mathbf{P}$ is a threshold partition with respect to t if each block $V_\ell \in \mathbf{p}$ satisfies $|V_\ell| \geq t$. Let*

$$\mathbf{P}_t \equiv \{\mathbf{p} \in \mathbf{P} : \forall\, V_\ell \in \mathbf{p},\ |V_\ell| \geq t\}, \tag{4.2}$$

denotes the set of all threshold partitions with respect to t.

Definition 4.2.4. *The bottleneck threshold partitioning problem (BTPP) with respect to t is to find a partition $\mathbf{p} \in \mathbf{P}_t$ that minimizes the objective:*

$$\max_{ij \in E(\mathbf{p})} w_{ij} = \max_{V_\ell \in \mathbf{p}} \max_{i,j \in V_\ell} w_{ij} \tag{4.3}$$

The minimum value of this objective is denoted by λ. An optimal partition is any partition $\mathbf{p}^\dagger$ satisfying

$$\max_{ij \in E(\mathbf{p}^\dagger)} w_{ij} = \min_{\mathbf{p} \in \mathbf{P}_t} \left(\max_{ij \in E(\mathbf{p}^\dagger)} w_{ij} \right) \equiv \lambda. \tag{4.4}$$

The original problem of finding a threshold clustering with respect to t such that minimizes the maximum within-cluster dissimilarity is equivalent to solving BTPP.

Definition 4.2.5. *The degree in G of a vertex $i \in V$ is its number of adjacent vertices in G:*

$$\deg(i) = \deg(i; G) \equiv |\{j \in V : ij \in E\}|. \tag{4.5}$$

Of particular note, if $\mathbf{p}^\dagger$ is an optimal partition for BTPP, then $\deg(i; G(\mathbf{p}^\dagger)) \geq t - 1$ for all $i \in V$.

Definition 4.2.6. *The bottleneck subgraph of G with bottleneck threshold ω, denoted $BG_\omega = (V, BE_\omega)$, has an edge $ij \in BE_\omega \subset E$ if and only if the weight of that edge $w_{ij} \leq \omega$:*

$$BE_\omega \equiv \{ij \in E : w_{ij} \leq w\}. \tag{4.6}$$

Definition 4.2.7. *The k-nearest-neighbors subgraph of G is a subgraph $NG_k = (V, NE_k)$ where an edge $ij \in NE_k$ if and only if j is one of the k closest vertices to i or i is one of the k closest vertices to j. Rigorously, for a vertex i, let $i_{(\ell)}$ denote the vertex that corresponds to the ℓ^{th} smallest value of $w_{ij}, ij \in E$, where ties are broken arbitrarily: $w_{ii_{(1)}} \leq w_{ii_{(2)}} \leq \cdots \leq w_{ii_{(m)}}$. Then*

$$NE_k \equiv \left\{ij \in E : j \in (i_{(\ell)})_{\ell=1}^{k} \text{ or } i \in (j_{(\ell)})_{\ell=1}^{k}\right\}. \tag{4.7}$$

Definition 4.2.8. *A graph $G' = (V, E')$ is a spanning subgraph of G if both graphs have the same vertex set V and $E' \subseteq E$. Note that subgraphs generated by partitions, bottleneck subgraphs and k-nearest-neighbors subgraphs are spanning subgraphs.*

Lemma 4.2.9. *If $\omega \geq \lambda$, where λ is the maximum within-block edge weight of an optimal partition $\mathbf{p}^\dagger$, then each vertex i in the bottleneck subgraph BG_w has $\deg(i; BG_\omega) \geq t - 1$.*

Proof. By Definition 4.2.4 of optimal partition, we know that

$$\forall\ ij \in E(\mathbf{p}^\dagger),\ \text{weight } w_{ij} \leq \lambda.$$

By Definition 4.2.6, for a bottleneck subgraph $BG_\omega = (V, BE_\omega)$, we have

$$\forall\ ij \in BE_\omega,\ \text{weight } w_{ij} \leq \omega.$$

Since $\omega \geq \lambda$, BG_ω contains all edges in $E(\mathbf{p}^\dagger)$. Base on Definition 4.2.8, we conclude that $G(\mathbf{p}^\dagger)$ is a spanning subgraph of BG_λ, thus

$$E(\mathbf{p}^\dagger) \subseteq BE_\lambda$$

Since each block of $\mathbf{p}^\dagger$ has at least t vertices,

$$\deg(i; G(\mathbf{p}^\dagger)) \geq t - 1$$

As the number of edges in the graph increase, the degree of a vertex must increase, thus

$$\deg(i; BE_\omega) \geq t - 1$$

□

We show the $(t-1)$-nearest-neighbors subgraph is a spanning subgraph of bottleneck subgraph BG_λ so that the largest edge weight can be bounded (Higgins et al., 2016).

Lemma 4.2.10. *For all* $ij \in NE_{t-1},\ w_{ij} \leq \lambda$.

Proof. Observe that each vertex i has $\deg(i; NG_{t-1}) \geq t-1$. Moreover, removing any edge from NG_{t-1} will necessarily force a vertex i to have $\deg(i) < t-1$.

By Lemma 4.2.9, each vertex i has $\deg(i; BG_\lambda) \geq t-1$. It can then be shown that $NE_{t-1} \subset BE_\lambda$. That is, NG_{t-1} is a spanning subgraph of BG_λ. Thus $NE_{t-1} \subset BE_\lambda$. Since

$$\forall\ ij \in BE_\lambda,\ w_{ij} \leq \lambda,$$

thus $\forall\ ij \in NE_{t-1},\ w_{ij} \leq \lambda$. □

Lemma 4.2.11. *If we can form groups that are connected in* NG_{t-1} *such that there is a path of* α *connecting any two units, then this would form a clustering satisfying* $\alpha-$*approximation.*

Proof. By Lemma 4.2.10, $\forall\ ij \in NE_{t-1},\ w_{ij} \leq \lambda$.
Suppose i and j has a path of α connecting them, by the triangle inequality,

$$w_{ij} \leq \alpha\lambda$$

□

4.3 A polynomial-time 2-approximation algorithm

4.3.1 Notation

Definition 4.3.1. *A spanning tree is a subgraph of G that has all the vertices covered with minimum possible number of edges. Also, a spanning tree does not have cycles.*

Definition 4.3.2. *A minimum spanning tree (MST) is a subgraph of G that connects all the vertices with smallest total cost, that is, sum for all weights of the edges in spanning tree is minimized and each vertex can be reached from others by following the edges.*

We can use depth first search (DFS) or breath first search (BFS) to construct a spanning tree in linear time. Kruskal's algorithm (Kruskal, 1956) runs in $O(|E|log|V|)$ time (Cormen et al., 2009) and it can be used to find minimum spanning tree. $|E|$ is number of edges for the graph and $|V|$ is number of units for the graph.

Pick a vertex (denoted by i) from a spanning tree arbitrarily, then we define some sets which shown below:

Let set S_1 include vertex i.

Let set S_2 include vertices that has a walk of length one from i. These vertices denote as $j_1, j_2, ..., j_p$ where p is size of set S_2, thus $S_2 = \{j_1, j_2, ..., j_p\}$.

Let set S_3 include vertices that has a walk of length two from i. These vertices denote as $k_1, k_2, ..., k_q$ where q is number of vertices belong to S_3, thus $S_3 = \{k_1, k_2, ..., k_q\}$.

Similar, S_4 include vertices that has a walk of length three from i.

4.3.2 Sketch of 2-approximation Algorithm

To show approximate optimality we must show that there exist a partition that is a valid blocking and that the subgraph generated by the partition contain no edge with a weight greater than 2λ.

The 2-approximation algorithm works as follows:

1. Form a $(2-1)$ nearest-neighbor graph NG_1.

2. Construct spanning tree through each connected subgraph.

3. Select one vertex i arbitrarily from the spanning tree to be the root.

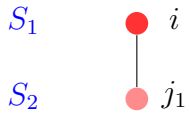
4. Construct groups as follows:

Note that if i, j are endpoints of an edge, w_{ij} denotes the dissimilarity between these vertices, by Definition of bottleneck threshold partitioning problem , the maximum dissimilarity between any two vertices in the same block is λ, $w_{ij} \leq \lambda$. If $(i = i_0, i_1, ..., i_m = j)$ is a walk of length m from i to j and the edge weights satisfy the triangle inequality , then the weight $w_{ij} \leq m\lambda$.

Case I: S_2 only include one vertex ($p = 1$)

(A) **If S_3 is an empty set**

We form a group comprised of vertices i and j_1.



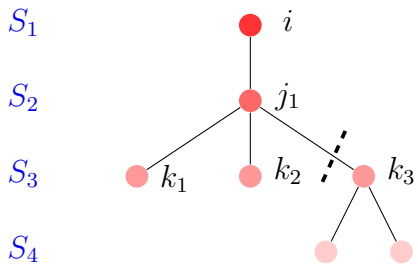
In this case, we has a group of size two and j_1 has a path of length one connected with i, then

$$w_{ij_1} \leq \lambda$$

.

(B) **If S_3 contains q vertices, $q \geq 1$, r vertices in S_3 have walks of length one from themselves to vertices in S_4, $q - r$ vertices don't**

Cut the edges between j_1 and these r vertices belong to S_3. Removing r edge from a tree breaks it into $(r + 1)$ separate subtrees. Thus, we can form a group comprised of vertices i, j_1 and the rest $(q - r)$ vertices in S_3. For the remaining r subtrees, since those r vertices in S_3 no longer connected with j_1, they can be viewed as new i for each subtree and consider all the cases which shown above.



In this case, we has a group of size $(q-r+2)$. Randomly pick two vertices within the group, there is a path of two or less that connecting them.

$$w_{ij_1} \leq \lambda$$

$$w_{j_1k_\ell} \leq \lambda$$

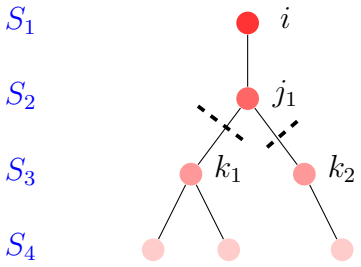
$$w_{ik_\ell} \leq w_{ij_1} + w_{j_1k_\ell} \leq 2\lambda$$

$$w_{k_\ell k_{\ell'}} \leq w_{j_1k_\ell} + w_{j_1k_{\ell'}} \leq 2\lambda$$

where $\ell = 1, ..., q, \ell' = 1, ..., q, \ell \neq \ell'$ and $k_\ell, k_{\ell'}$ belong to the same group.

(C) **If S_3 contains q vertices, $q \geq 1$ and every k_ℓ $(\ell = 1, ..., q)$ has a walk of length one from itself to vertices from S_4**

Cut all the edges between j_1 and k_ℓ. Removing q edge from a tree breaks it into $q+1$ separate subtrees. Thus, we can form a group comprised of vertices i and j_1. For the other q subtrees, vertices in S_3 that no longer connected with j_1 can be viewed as new i and consider all the cases which shown above.



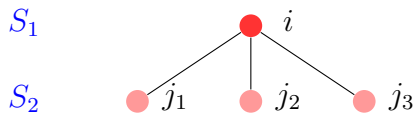
In this case, we has a group of size two and j_1 has a path of length one from j_1 to i.

$$w_{ij_1} \leq \lambda$$

Case II: S_2 include at least two vertices ($p \geq 2$)

(A) **If S_3 is an empty set**

Form a group comprised of node i and all the j_ℓ ($\ell = 1, ..., p$).



In this case, we has a group of size $(p+1)$ and there is a path of length two or less that between any two vertices within the group.

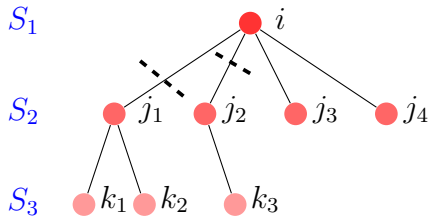
$$w_{ij_\ell} \leq \lambda$$

$$w_{j_\ell j_{\ell'}} \leq w_{ij_\ell} + w_{ij_{\ell'}} \leq 2\lambda$$

where $\ell = 1, 2, ..., p$, $\ell' = 1, 2, ..., p$ and $\ell \neq \ell'$.

(B) **If S_3 is not empty, r vertices in S_2 adjacent to vertices in S_3 and $(p-r)$ vertices don't, $p \geq r \geq 1$**

Cut all edges between i and those in S_2 which adjacent to vertices in S_3. Removing r edges from a tree breaks it into $(r+1)$ separate subtrees. Thus, we can form a group comprised of i and vertices belong to S_2 which do not adjacent to vertices from S_3. For the other r subtrees, those vertices from S_3 that no longer connected with vertices in S_2 can be viewed as new i for each subtree and consider all the cases that shown above.



In this case, we has a group of size $(r+1)$. Randomly pick two nodes within the group, there is a path of length two or less between them.

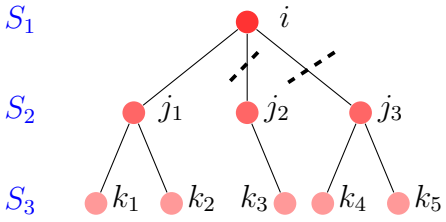
$$w_{ij_\ell} \leq \lambda$$

$$w_{j_\ell j_{\ell'}} \leq w_{ij_\ell} + w_{ij_{\ell'}} \leq 2\lambda$$

where $\ell = 1, ..., p, \ell' = 1, ..., p, \ell \neq \ell'$ and $j_\ell, j_{\ell'}$ belong to the same group.

(C) **If S_3 is not empty, every vertex in S_2 has a walk of length one from itself to vertices in S_3, S_4 is an empty set**

Cut $p-1$ edges between i and j_ℓ except for edge between i and j_1. Removing $(p-1)$ edges from a tree breaks it into p separate subtrees. For the subtree that contains i, j_1 and vertices connected with j_1, we should consider all the situations in Case I that have shown above. For the other $(p-1)$ subtrees, view nodes in S_2 that no longer connect with i as new root node i and consider all the cases that shown above.



The cutting procedure terminates when all the vertices have been considered. After all the cutting procedure, we separate the spanning tree into several groups, each group has at least 2 vertices and within each group, there are walks of length two or less between them.

4.4 A polynomial-time 3-approximation algorithm

Solving BTPP exactly is computationally expensive, and in fact, no $(2-\epsilon)$-approximation algorithm terminates in polynomial time unless $P = NP$ (Higgins et al., 2016). However, Higgins et al. (2016) derive a 4-approximation algorithm for BTPP that terminates in linear time and space outside of the construction of a nearest neighbor graph. In this section, we will show how to construct a polynomial-time 3-approximation algorithm to solve BTPP.

4.4.1 Notation

Definition 4.4.1. *For $i, j \in V$, a path from i to j of length m in G is a sequence of $m+1$ vertices $(i = i_0, i_1, \ldots, i_m = j)$ such that the edge $i_{\ell-1}i_\ell \in E$, $i_{\ell-1} \neq i_\ell$.*

Of particular note, i and j may be the same vertex for a path. A path is a walk in which all vertices are distinct (except possibly the first and last vertex).

Definition 4.4.2. *A cycle $C_m = (i_1i_2 - i_2i_3 - \ldots - i_mi_1)$ of length m $(m \geq 3)$ is a path that begins and ends on the same vertex.*

Of particular note, the number of vertices in C_m equals to the number of edges, and every vertex has degree 2.

Definition 4.4.3. *A junction i is the vertex that connects two or more cycles.*

Let $t = 3$, choose a vertex i in $(t-1)$-nearest-neighbor subgraph NG_2 arbitrarily, define the two closest neighbors of i as j and k. Edge $ij, ik \in NE_2$.

4.4.2 Sketch of 3-approximation Algorithm

To show approximate optimality we must show that there exist a partition that is a valid blocking and that the subgraph generated by the partition contains no edge with a weight greater than 3λ.

Lemma 4.4.4. *For any whole number $m \geq 3$, m mod $3p + 4q$ is either 0 or 1, where p and q are both whole numbers, $p + q > 0$.*

Proof. The reason is straight forward.

If m is divisible by 3, then we have $m = 3p$. Let $q = 0$, we can write $m = 3p + 4q$, thus m mod $3p + 4q$ is 0.

If m is not divisible by 3, then $m = 3p + 1 = 3(p - 1) + 4 = 3p' + 4q$. m mod $3p + 4q$ is 0. Or $m = 3p + 2 = 3(p - 1) + 5 = 3(p - 1) + 4 + 1 = 3p' + 4q + 1$, in this situation, m mod $3p + 4q$ is 1. □

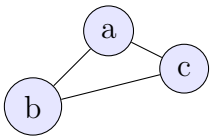
Lemma 4.4.5. *For any whole number $m > 5$, m is divisible by $3p + 4q$, where p and q are both whole numbers, $p + q > 0$.*

Proof. If m is divisible by 3, $m = 3p$. Let $q = 0$, we can write $m = 3p + 4q$, thus m is divisible by $3p + 4q$.

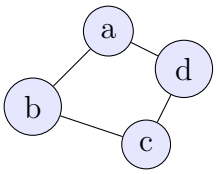
If m is not divisible by 3, then $m = 3p + 1 = 3(p - 1) + 4 = 3p' + 4q$, m is divisible by $3p + 4q$. Or $m = 3p + 2 = 3(p - 2) + 8 = 3(p - 2) + 4(2) = 3p' + 4q$, thus m is divisible by $3p + 4q$. □

Lemma 4.4.6. *For graph cycle of length $m, m \leq 5$, any vertex within the cycle C_m has a walk of length no more than three from itself to other vertices in C_m.*

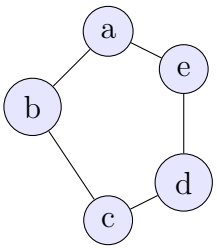
Proof. Since $3 \leq m \leq 5$, we can proof this Lemma by exhaustion. Let $C_3 = (ab - bc - ca)$, $C_4 = (ab - bc - cd - da)$, $C_5 = (ab - bc - cd - de - ea)$, the corresponding graphs are shown below.



C_3



C_4



C_5

Table 4.4.2 below contains all possible walks and their corresponding length for different cycles.

Cycle	Walk Length	Walk
C_3	1	$(a,b),(a,c),(b,c)$
C_4	1	$(a,b),(b,c),(c,d),(d,a)$
	2	$(a,b,c),(b,c,d),(c,d,a),(d,a,b)$
C_5	1	$(a,b),(b,c),(c,d),(d,e),(e,a)$
	2	$(a,b,c),(b,c,d),(c,d,e),(d,e,a),(e,a,b)$
	3	$(a,b,c,d),(b,c,d,e),(c,d,e,a),(d,e,a,b),(e,a,b,c)$

From Table 4.4.2, we found that from every vertex to other vertices within the cycle, there will be a walk of length three or less. □

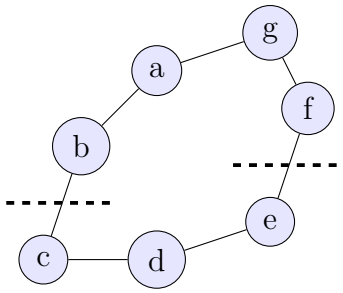
Lemma 4.4.7. *For any graph cycle C_m of length $m, m > 5$, we can partition the cycle into several subgraphs, each vertex in the subgraph has a walk of length three or less from itself to other vertices within the subgraph.*

Proof. By Lemma 4.4.5, m is divisible by $3a + 4b$, indicates we can cut several edges of the cycle so that each subgraph contains a sequence of 3 or 4 vertices.

If the subgraph contains a sequence of 3 vertices $(i = i_0, i_1, i_2, i_3 = j)$ and edge i_0i_1, i_1i_2, i_2i_3 belongs to the subgraph, then each vertex within the group has a walk of length two or less from itself to other vertices within the group.

If the subgraph contains a sequence of 4 vertices $(i = i_0, i_1, i_2, i_3, i_4 = j)$ and edge $i_0i_1, i_1i_2, i_2i_3, i_3i_4$ belongs to the subgraph, then each vertex within the group has a walk of length three or less from itself to other vertices within the group.

For example, suppose a cycle $C_7 = (ab - bc - cd - de - ef - fg - ga)$ has length= 7, we can cut edge bc and ef to form two groups. Each group contains at least 3 vertices. For the group that contains vertices a, b, g and f, a walk from b to f has length 3, walks from b to g, a to f have length 2, walks from b to a, a to g, g to f have length 1. For the group that contains vertices c, d and e, a walk from c to e has length 2, walks from c to d and d to e have length 1.



□

We can derive a threshold blocking ($t = 3$) that every vertex in a block has a walk of length three or less from itself to other vertices within the block.

The 3−approximation algorithm works as follows:

1. Form a $(3-1)$ nearest-neighbor graph NG_2.
2. Construct groups.
   (a) Start with an arbitrary vertex i and detect cycle using depth first search (DFS). The idea for DFS works as follows: for every visited vertex v, when we have found any adjacent vertex u, such that u is already visited, and u is not the parent of vertex v. Then one cycle is detected. If this cycle contains i and its 2 nearest-neighbors, mark vertices belonging to this cycle as visited, start with a new vertex.
   (b) If such a cycle do not exist, consider other cases which shown below.
   (c) If all the vertices are marked as visited, go to next step.
3. Partition groups.

For this procedure, once an edge ij is "cut", $ij \notin NE_{t-1}$. We will discuss grouping part first.

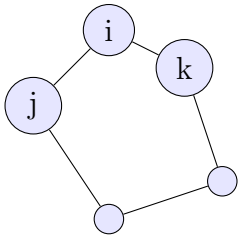
Part 1: Group forming

Let $t = 3$, for a given $(3-1)$-nearest neighbor subgraph NG_2 and an arbitrary vertex i from NG_2, suppose the two nearest neighbor vertices of i are j and k, we should consider the

following cases to divide all the vertiecs into groups. After all the n vertices are assigned to groups, we will go to the next part: partition. For the figures which contained in the proof, we use multiple abbreviations. "Ecycle" is short for existing cycle, "J" to abbreviate junction, "EJ" is short for existing junction.

Case I: An unique cycle For $i \in V$, use greedy search from both direction to find if there exist a cycle $C = \{ij - ... - ki\}$. We can use depth first search (DFS) to detect cycle in an undirected graph in $O(V + E)$ (Knuth, 1997) time where V is the number of nodes in the graph and E is the number of edges in the graph. The time complexity is linear in the size of the graph.

(A) **If such a cycle exist**, mark C as existing cycle. Vertices included in the cycle all mark as read and they form a group. Start with an unread vertex in V to detect new cycles.

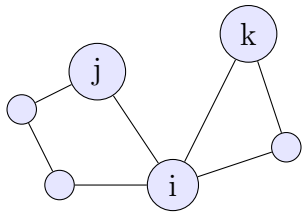


(B) **If such a cycle do not exist**, consider other cases which listed below.

Case II: Unique cycle to unique cycle Start from vertex $i \in V$, use greedy search to find if the following situations satisfied. If none of them happens, consider other cases which listed below.

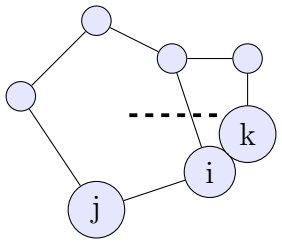
(A) **Two cycles share one vertex**

If there exist a cycle $C_{m_1} = (ij - ji_2 - ... - i_{m_1}i)$ with length m_1 and another cycle $C_{m_2} = (ik - ki_2...i_{m_2}i)$ with length m_2, $m_1 \geq m_2$, both cycles only share vertex i, no edges is shared by C_{m_1} and C_{m_2}, then a group comprised of vertices which included in cycle C_{m_1} can be formed. Mark those vertices as read and mark C_{m_1} as existing cycle. Start with a new unread vertex in V and find cycle with other unread vertices.



(B) **Two cycles share multiple vertices**

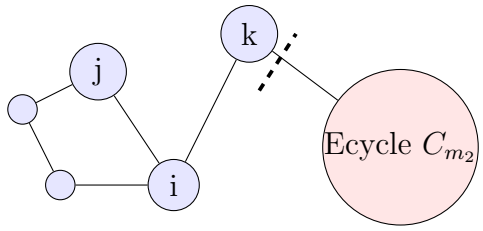
If there exist a cycle $C_{m_1} = (ij - ji_2 - ... - i_{m_1}i)$ with length m_1 and another cycle $C_{m_2} = (ik - ki_2...i_{m_2}i)$ with length m_2, these two cycles share p edges ($p \geq 1$) and more than one vertices, then merge C_{m_1} and C_{m_2} into a larger cycle $C_{m_1+m_2-2p} = (ij - ... - ki)$ by cutting all the sharing edges. New cycle has length $m_1 + m_2 - 2p$. Form a group comprised of vertices belongs to the new cycle $C_{m_1+m_2-2p}$, mark these vertices as read and mark $C_{m_1+m_2-2p}$ as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.



Case III: Unique cycle to existing cycle Start from vertex $i \in V$, use greedy search to find if the following situations happen. If there exist a cycle $C_{m_1} = (ij - ji_2 - ... - i_{m_1}i)$ and vertex k connects to an existing cycle C_{m_2}, then we can partition these vertices into groups following rules which shown below. If it doesn't happen, consider other cases which listed below.

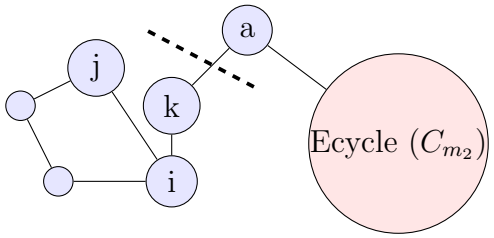
(A) **A walk of length 2 between two distinct cycles**

If there is a junction vertex k between two cycles, cut the edge between k and existing cycle C_{m_2}. Form a group comprised of vertices belong to C_{m_1} and k, mark them as read, mark C_{m_1} as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.



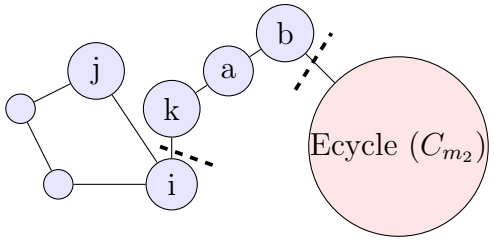
(B) **A walk of length 3 between two distinct cycles**

If there is a walk of length 3 from new cycle C_{m_1} to existing cycle C_{m_2}, vertices connect two cycles are k and a. Cut edge ak, add vertex a to the group that contains existing cycle C_{m_2} and mark a as read. Form a new group comprised of k and vertices within C_{m_1}, mark all the vertices in this group as read. Start with a new unread vertex in V and find cycles with other unread vertices.



(C) **A walk of length 4 between two distinct cycles**

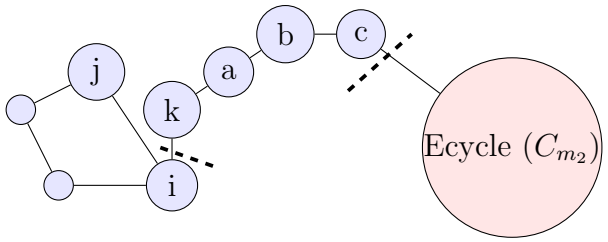
If there is a walk of length 4 from new cycle C_{m_1} to existing cycle C_{m_2}, vertices connect two cycles are a, b and k. Cut edge ik and edge between b and existing cycle C_{m_2}, form a new group comprised of vertices a, b and k, mark them as read. Mark vertices belongs to cycle C_{m_1} as read and form a new group. Mark C_{m_1} as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.



(D) **A walk of length 5 between two distinct cycles**

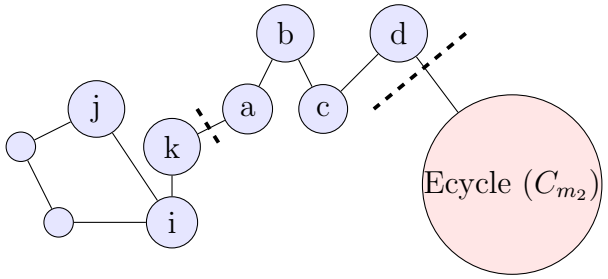
If there is a walk of length 5 from new cycle C_{m_1} to existing cycle C_{m_2}, vertices connect

two cycles are a, b, c and k. Cut edge ik and edge between c and existing cycle C_{m_2}, form a new group comprised of vertices a, b, c and k, mark them as read. Mark vertices belongs to cycle C_{m_1} as read and form a new group. Mark C_{m_1} as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.



(E) **A walk of length 6 between two distinct cycles**

If there is a walk of length 6 from new cycle C_{m_1} to existing cycle C_{m_2}, vertices connect two cycles are a, b, c, d and k. Cut edge ak and edge between d and existing cycle C_{m_2}, form a new group comprised of vertices a, b, c and d, mark them as read. Mark vertices belongs to C_{m_1} and k as read, form a new group. Mark C_{m_1} as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.



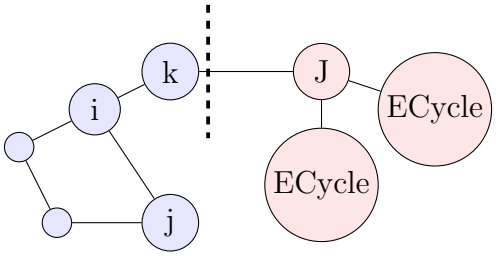
(F) **A walk of length greater than 6 between two distinct cycles**

If there is a path of length m' ($m' > 6$) from new cycle C_{m_1} to existing cycle C_{m_2}, there are $m' - 1$ vertices connect two cycles. By Lemma 4.4.5, those vertices can be partition as groups of size 3 or 4. Mark vertices connected two cycles as read. Form a new group comprised of vertices belong to cycle C_{m_1} and mark them as read. Mark C_{m_1} as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.

Case IV: Unique cycle to existing junction Start from vertex i, use greedy search to find if the following situations happen. If there exist a cycle $C_m = (ij - ji_2 - ... - i_m i)$ and k connects to an existing junction J, then we can partition these vertices into groups following rules which shown below. If it doesn't happen, consider other cases which listed below.

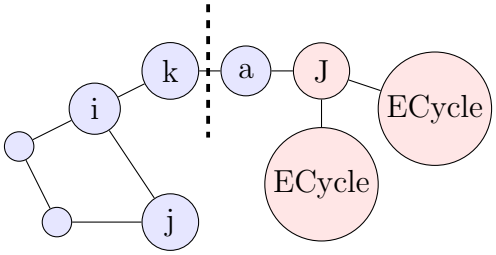
(A) **A walk of length 2 between cycle and J**

If there exist a vertex k connects cycle C_m and existing junction J, cut the edge between k and J. Mark vertices belongs to cycle C_m and k as read and form a new group. Mark C_m as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.



(B) **A walk of length 3 between cycle and J**

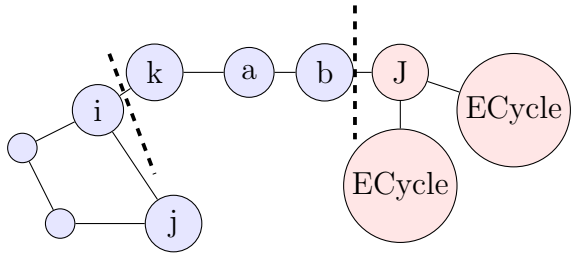
If there is a path of length 3 from C_m to J, vertices connect cycle and junction are k and a, cut edge ka, add vertex a to the group which contains J and mark a as read. Form a new group comprised of vertices belong to cycle C_m and k, then mark them as read. Mark C_m as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.



(C) **A walk of length 4 between cycle and J**

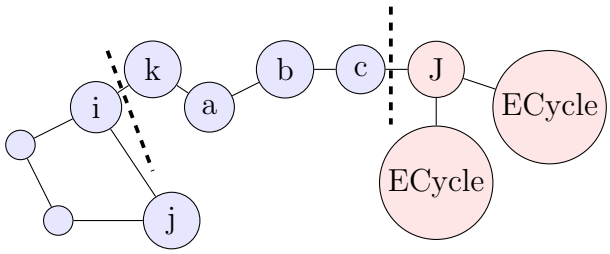
If there is a path of length 4 from C_m to J, vertices connect cycle and junction are a, b and k, cut edge ik and edge between b and existing junction J. Form a group

comprised of a, b and k, mark them as read. Mark vertices belongs to cycle C_m as read and form a new group. Mark C_m as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.



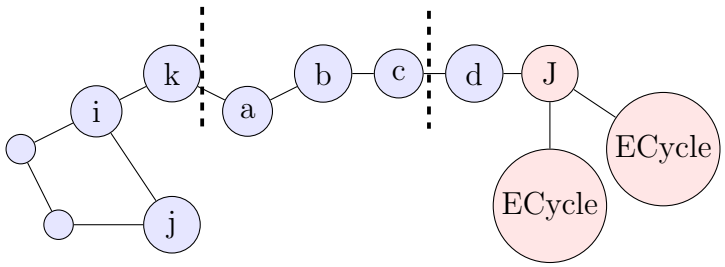
(D) **A walk of length 5 between cycle and J**

If there is a path of length 5 from C_m to J, vertices connect cycle C_m and junction are a, b, c and k, cut edge ik and edge between c and existing junction J. Mark vertices belongs to cycle C_m as read and form a new group. Vertices k, a, b, c form a new group and mark as read. C_m is marked as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.



(E) **A walk of length 6 between cycle and J**

If there is a path of length 6 from C_m to J, vertices connect cycle C_m and junction are a, b, c, d and k, then cut edge ka and cd. Mark vertices belongs to cycle C_m and k as read and form a new group. Vertices a, b, c form a new group and mark as read. Add vertex d to the group that contains J and mark d as read. C_m is marked as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.

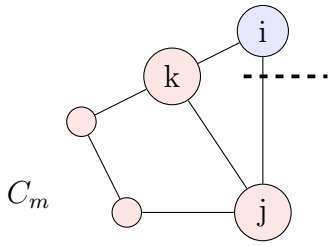


(F) **A walk of length greater than 6 between cycle and J**

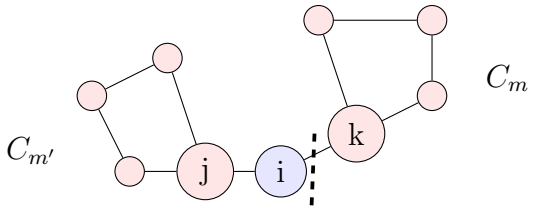
If there is a path of length m' ($m' > 6$) from new cycle C_m to existing junction J, there are $m' - 1$ vertices connect C_m and J. By Lemma 4.4.5, those vertices can be partition as groups of size 3 or 4. Mark vertices connected C_m and J as read. Form a new group comprised of vertices belong to cycle C_m and mark them as read. Mark C_m as existing cycle. Start with a new unread vertex in V and find cycles with other unread vertices.

Case V: Existing cycle to existing cycle Start from vertex i, use greedy search to find if the following situations happen. If none of them happened, consider other cases which listed below.

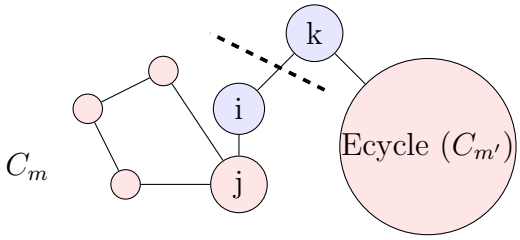
(A) **If k and j belong to the same existing cycle** $C_m = (jk - kk_2 - ... - k_{m-1}j)$, cut edge ij, add vertex i to the group that contains k and mark i as read. Start with a new unread vertex in V and find cycles with other unread vertices.



(B) **If k and j belong to two different existing cycles** $C_m = (kk_1 - ... - k_{m-1}k)$ **and** $C'_m = (jj_1 - ... - j_{m'-1}j)$, then cut edge ik, add vertex i to the group that vertex j belongs to and mark i as read. Start with a new unread vertex in V and find cycles with other unread vertices.

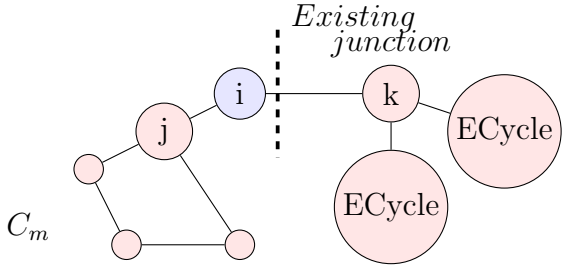


(C) **If j belongs to an existing cycle C_m and k connects to an existing cycle $C_{m'}$**, cut edge ik, add vertex i to the group that vertex j belongs and add k to the group that contains $C_{m'}$. Mark i and k as read. Start with a new unread vertex in V and find cycles with other unread vertices.



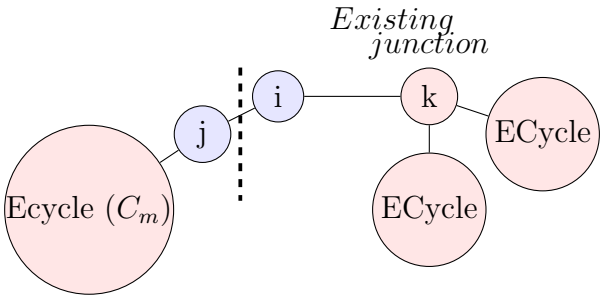
Case VI: Existing cycle to existing junction Start from vertex i, use greedy search to find if the following situations happen. If none of them happened, consider other cases which listed below.

(A) **If j belongs to existing cycle C_m and k is an existing junction**, cut edge ik, add vertex i to the group which contains j and mark i as read. Start with a new unread vertex in V and find cycles with other unread vertices.

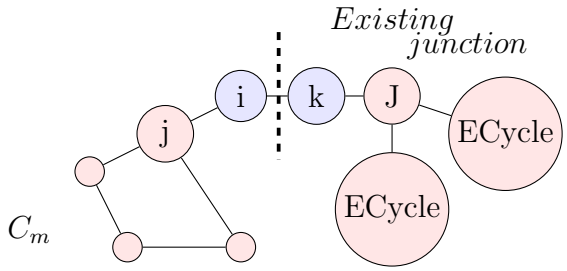


(B) **If j connects to existing cycle C_m and k is an existing junction**, cut edge ij, add vertex i to the group which contains k. Add vertex j to the group that contains

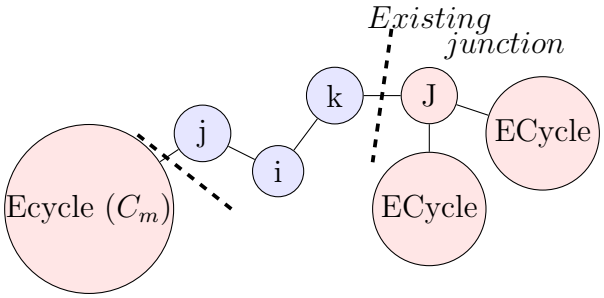
c. Mark i, j as read. Start with a new unread vertex in V and find cycles with other unread vertices.



(C) **If j belongs to existing cycle C_m and k connects to existing junction J**, cut edge ik and add vertex i to the group which contains j. Add k to the group that contains J. Mark i, k as read. Start with a new unread vertex in V and find cycles with other unread vertices.

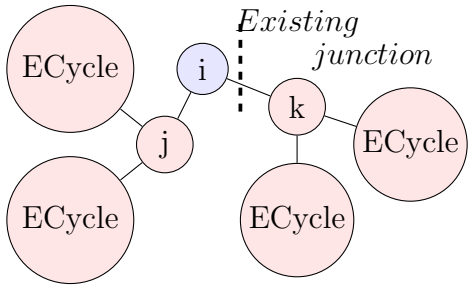


(D) **If j connects to existing cycle C_m and k connects to an existing junction J**, cut the edge between j and existing cycle C_m. Also cut edge between k and J. Form a group comprised of i, j, k and mark these vertices as read. Start with a new unread vertex in V and find cycles with other unread vertices.

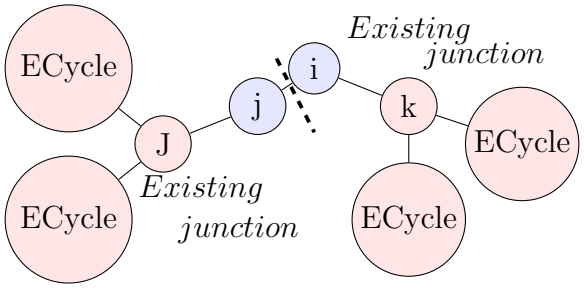


Case VII: Existing junction to existing junction Start from vertex i, use greedy search to find if the following situations happen. If none of them happened, consider other cases which listed below.

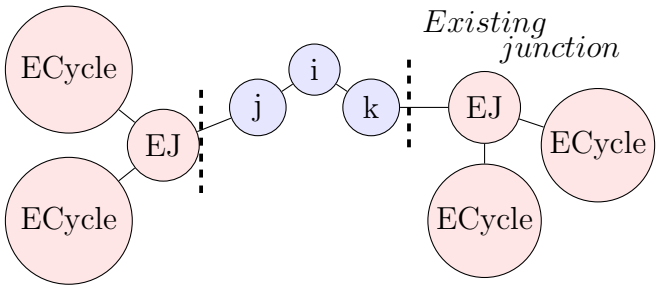
(A) **If j, k are two identical existing junctions**, cut edge ik and add i to the group that contains j, mark i as read. Start with a new unread vertex in V and find cycles with other unread vertices.



(B) **If j connects to an existing junction J and k is another existing junction**, cut edge ij and add j to the group which contains J, add i to the group that contains k. Mark both i, j as read. Start with a new unread vertex in V and find cycles with other unread vertices.



(C) **If j, k connect to two different existing junctions**, then cut edge between j and existing junction, cut edge between k and existing junction. Form a group comprised of i, j and k, mark i, j, k as read. Start with a new unread vertex in V and find cycles with other unread vertices.

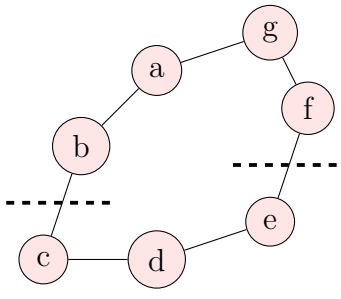


When all the vertex in V are marked as read, we go to the partition part. For each group, we will consider the following situations and perform the partition so that each block will have at least 3 vertices and each vertex within the same block will have a walk of length three or less from itself to other vertices within the block.

Part 2: Partition each group into blocks

For all i, j belongs to the same group, if i, j are endpoints of an edge, w_{ij} denotes the dissimilarity between these vertices, by Definition of bottleneck threshold partitioning problem , the maximum dissimilarity between any two vertices in the same block is λ, $w_{ij} \leq \lambda$. If $(i = i_0, i_1, ..., i_m = j)$ is a walk of length m from i to j and the edge weights of NG_2 satisfy the triangle inequality , then the weight $w_{ij} \leq m\lambda$.

Case I: An unique cycle When the group contains an unique cycle of length m, by Lemma 4.4.6 and Lemma 4.4.7, we can partition the cycle into several blocks so that each block contains a sequence of 3 or 4 vertices.



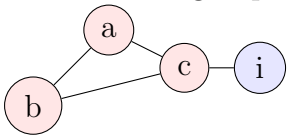
For example, we can partition a cycle $C_7 = ab - bc - ... - ga$ into two blocks by cutting edges bc and ef. For the block that contains a sequence of 4 vertices (b, a, g, f), $w_{bf} \leq w_{ba} + w_{ag} +$

$w_{gf} \leq 3\lambda$. For the block that contains a sequence of 3 vertices (c, d, e), $w_{ce} \leq w_{cd} + w_{de} \leq 2\lambda$.

Case II: Unit i connect with a cycle When the group contains a cycle C_m of length m and one vertex i connects to the cycle, we should consider the following situations.

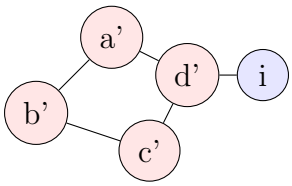
(A) $\mathbf{m \leq 4}$

Maintain the group as a block. Let $C_3 = (ab - bc - ca), C_4 = (a'b' - b'c' - c'd' - d'a')$.



$$w_{ia} \leq w_{ic} + w_{ca} \leq 2\lambda$$

$$w_{ib} \leq w_{ic} + w_{cb} \leq 2\lambda$$

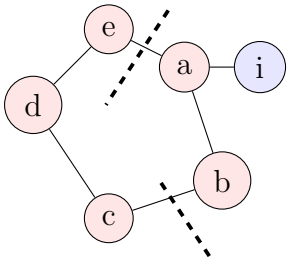


$$w_{ia'} \leq w_{id'} + w_{d'a'} \leq 2\lambda$$

$$w_{ib'} \leq w_{id'} + w_{d'c'} + w_{c'b'} \leq 3\lambda$$

(B) $\mathbf{m > 4}$ **and** $\mathbf{m \neq 7}$

Cut edge ae and bc so that form a block comprised of i, a and b. For the rest of the vertices, we can partition them into several blocks so that each block contains a sequence of 3 or 4 vertices.

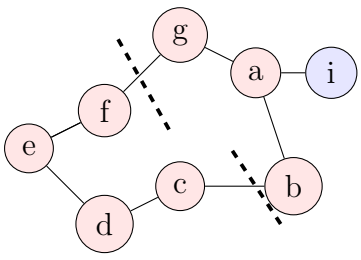


$$w_{ib} \leq w_{ia} + w_{ab} \leq 2\lambda$$

$$w_{ce} \leq w_{cd} + w_{de} \leq 2\lambda$$

(C) $\mathbf{m = 7}$

Suppose $C_7 = (ab - bc - ... - ga)$, i connects to a. Cut edge bc and gf, form a block comprised of i, a, b and g. For the rest of the vertices, they can form another block.



$$w_{ig} \leq w_{ia} + w_{ag} \leq 2\lambda$$

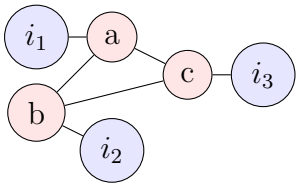
$$w_{ib} \leq w_{ia} + w_{ab} \leq 2\lambda$$

$$w_{cf} \leq w_{cd} + w_{de} + w_{ef} \leq 3\lambda$$

Case III When the group contains a cycle C_m of length m and there are p $(1 < p \leq n)$ vertices $i_1, i_2, ..., i_p$ connect to the cycle C_m, we should consider the following situations to perform partition.

(A) $\mathbf{m = 3}$

Suppose $C_3 = (ab - bc - ca)$, a block comprised of the cycle C_3 and all the connected vertices can be formed. The block size would be $m + p$.



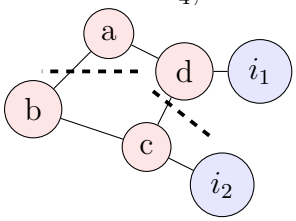
$$w_{i_1c} \leq w_{i_1a} + w_{ac} \leq 2\lambda$$

$$w_{i_1i_2} \leq w_{i_1a} + w_{ab} + w_{bi_c} \leq 3\lambda$$

$$w_{i_2i_3} \leq w_{i_2b} + w_{bc} + w_{ci_3} \leq 3\lambda$$

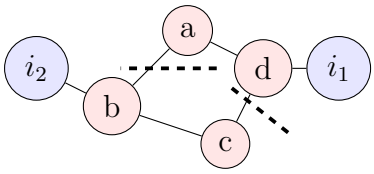
(B) $\mathbf{m = 4}$

Suppose $C_4 = (ab - bc - cd - da)$, we can partition each group into two blocks by cutting two edges of cycle C_4 such that each block contains at least one vertex i_ℓ that connect to C_4, the block size should be 3 or 4.



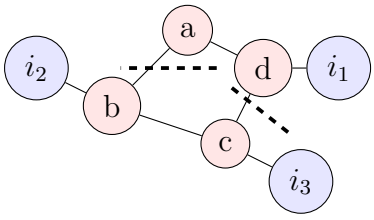
$$w_{i_1a} \leq w_{i_1d} + w_{da} \leq 2\lambda$$

$$w_{i_2b} \leq w_{i_2c} + w_{cb} \leq 2\lambda$$



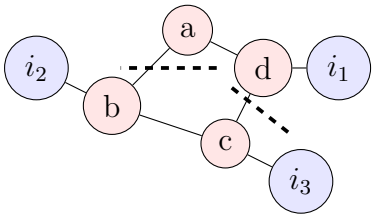
$$w_{i_1a} \leq w_{i_1d} + w_{da} \leq 2\lambda$$

$$w_{i_2c} \leq w_{i_2b} + w_{bc} \leq 2\lambda$$



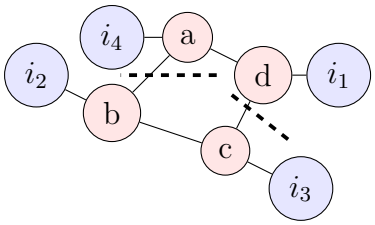
$$w_{i_1a} \leq w_{i_1d} + w_{da} \leq 2\lambda$$

$$w_{i_2i_3} \leq w_{i_2b} + w_{bc} + w_{ci_3} \leq 3\lambda$$



$$w_{i_1a} \leq w_{i_1d} + w_{da} \leq 2\lambda$$

$$w_{i_2i_3} \leq w_{i_2b} + w_{bc} + w_{ci_3} \leq 3\lambda$$

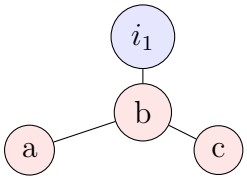


$$w_{i_1 i_4} \leq w_{i_1 d} + w_{da} + w_{di_4} \leq 3\lambda$$

$$w_{i_2 i_3} \leq w_{i_2 b} + w_{bc} + w_{ci_3} \leq 3\lambda$$

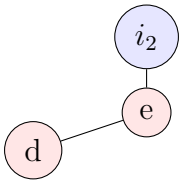
(C) **m > 4**

Partition cycle C_m and all connected vertices into blocks that satisfy one of the following patterns so that every vertex within the same block will have a walk of three or less from itself to other vertices within the block.

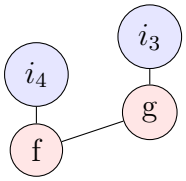


$$w_{i_1 a} \leq w_{i_1 b} + w_{ba} \leq 2\lambda$$

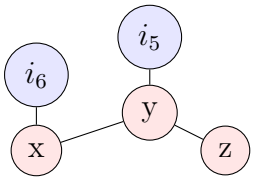
$$w_{ac} \leq w_{ab} + w_{bc} \leq 2\lambda$$



$$w_{i_2 d} \leq w_{i_2 e} + w_{ed} \leq 2\lambda$$



$$w_{i_3 i_4} \leq w_{i_3 g} + w_{gf} + w_{fi_4} \leq 3\lambda$$

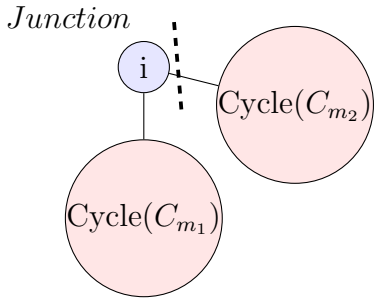


$$w_{i_3 i_4} \leq w_{i_3 x} + w_{xy} + w_{yi_4} \leq 3\lambda$$

$$w_{i_4 z} \leq w_{i_4 x} + w_{xy} + w_{yz} \leq 3\lambda$$

Note: Red nodes (eg: $a, b, c, d, e, f, g, x, y, z$) denote vertices belong to the cycle C_m, blue nodes (eg: $i_1, i_2, i_3, i_4, i_5, i_6$) denote vertices that connected to the cycle C_m.

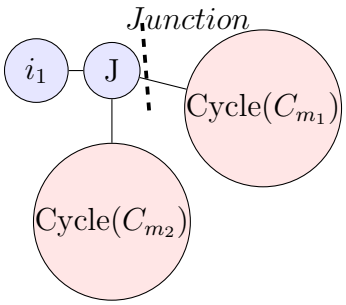
Case IV When the group contains a junction i that connects with two cycles C_{m_1}, C_{m_2}, we should cut edge between i and C_{m_2}, partition C_{m_2} following Case I's rule on page 68 and partition the rest of vertices (i and C_{m_1}) following Case II's rule on page 69.



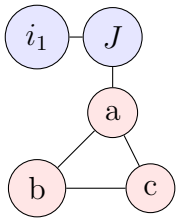
Case V When the group contains a junction J, two cycles C_{m_1}, C_{m_2} connected with J, $m_1 \leq m_2$, p $(p \geq 1)$ vertices $i_1, ..., i_p$ connect to junction J.

(A) $\mathbf{p = 1}$

Use i_1 to denote the vertex that connects to junction J, cut edge between junction J and cycle C_{m_1}, then partition C_{m_1} following Case I's rule on page 68. For cycle C_{m_2} and i_1, J, we should consider m_2 and partition them based on the following sceneries.



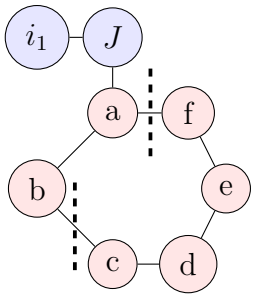
(a) If $m_2 = 3$, let $C_{m_2} = (ab - bc - ca)$, we can form a block comprised of C_{m_2} and i_1, J.



$$w_{i_1 b} \leq w_{i_1 J} + w_{Ja} + w_{ab} \leq 3\lambda$$

$$w_{i_1 c} \leq w_{i_1 J} + w_{Ja} + w_{ac} \leq 3\lambda$$

(b) If $m_2 = 6$, let $C_{m_2} = (ab - bc - ... - fa)$, we can cut edge bc and af so that two blocks can be formed, each block contains a sequence of 4 vertices.



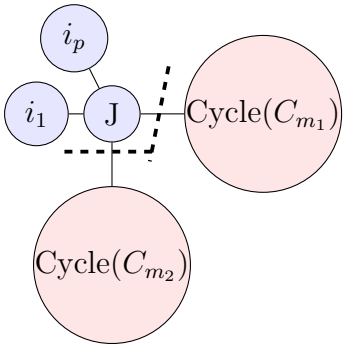
$$w_{i_1 b} \leq w_{i_1 J} + w_{Ja} + w_{ab} \leq 3\lambda$$

$$w_{fc} \leq w_{fe} + w_{ed} + w_{dc} \leq 3\lambda$$

(c) If $m_2 > 3$ and $m_2 \neq 6$, let $C_{m_2} = (aa_1 - a_1a_2 - ... - a_{m_2-1}a)$, cut edge aa_1 and $a_{m_2-1}a$ so that a block comprised of i, J and a is formed. For the rest of the vertices inside C_{m_2}, there is a walk of length $m_2 - 1$ from a_1 to a_{m_2-2}. by Lemma 4.4.4 and 4.4.5, we can partition the rest of the $m_2 - 2$ vertices into as sequence of 3 or 4 vertices so that every vertex in the block has a walk of length no more than three to other vertices within the block.

(B) $\mathbf{p > 1}$

Use $i_1, ..., i_p$ to denote vertices that connect to junction J, we cut edge between junction J and C_{m_1}, also, cut edge between J and C_{m_2}. Partition $\mathbf{c_1}$ and $\mathbf{c_2}$ based on Case I's rule on page 68. Form a block comprised of $i_1, ..., i_p$ and J.



$$C_{i_1, i_p} \leq C_{i_1 J} + C_{J i_p} \leq 2\lambda$$

After partitioning every group into blocks, we separate the all the n vertices into several blocks, each block has at least 3 vertices and within each block, a vertex has a walk of length three or less from itself to other vertices.

4.5 Discussion

In Section 4.3 and 4.4, it seems promising in deriving a polynomial-time 3-approximation algorithm when $t = 3$ and a polynomial-time 2-approximation algorithm when $t = 2$. However,

it is unclear how these algorithms perform under massive data settings.

Chapter 5

Conclusion

5.1 Introduction

As the size n of datasets become massive, many commonly-used clustering algorithms (e.g. k-means or hierarchical agglomerative clustering (HAC) require prohibitive computational cost and memory. In this dissertation, we propose a solution to these clustering problems by extending threshold clustering (TC) to problems of instance selection. TC is a recently developed clustering algorithm designed to partition data into many small clusters in linearithmic time (on average). This entire procedure for clustering is called *iterative hybridized threshold clustering* (IHTC). Through simulation results and by applying our methodology on several real datasets, we show that IHTC combined with k-means or HAC substantially reduces the run time and memory usage of the original clustering algorithms while still preserving their performance. Additionally, IHTC helps prevent singular data points from being overfit by clustering algorithms.

Given a set of n units, a threshold t, and a dissimilarity measure satisfying the triangle inequality that can be computed between every pair of units, the bottleneck threshold partitioning problem (BTPP) is to partition the n units so that each block of the partition contains at least t units and so that the the maximum dissimilarity between two units within the same block of the partition is minimized. Recent work has yielded a polynomial-time

4–approximation algorithm for BTPP.

Another extension of TC is to use TC as a clustering technique on massive dataset, we call it hierarchical agglomerative threshold clustering (HATC). In this case, no other clustering algorithms is required. Through simulation results and application on several real datasets, we show that HATC has better performance than HAC not only in runtime and memory requirement but also in ratio of BSS/TSS.

Additionally, it has been shown that, when $t > 2$, no polynomial-time $(2 - \epsilon)$–approx--imation algorithm for BTPP exists unless P = NP. In this dissertation, we also derive a polynomial-time 2–approximation algorithm when $t = 2$ and a polynomial-time 3–approximation algorithm when $t = 3$.

5.2 Future Work

Simulation results and application on real datasets show that implementing clustering methods with IHTC may decrease their runtime and memory usage without sacrificing the their performance. For more sophisticated clustering methods, this reduction in runtime and memory usage may be dramatic. Even for the standard implementation of k-means in R, one iteration of IHTC decreases the runtime and memory usage by more than half while maintaining clustering performance.

However, the threshold clustering algorithm we proposed in this dissertation is a 4-approximation algorithm for bottleneck threshold partitioning problem. In the future, we will explore the possibility to obtain an improved polynomial-time algorithm for BTPP with an approximation factor less than 4.

For the simulation of HATC, we let $t^* = 2$ and compare the performance of our methods with HAC. Later, we may compare the cluster performance for HATC with varying threshold t^* across different datasize. Also, we may include other indexes of cluster performance (silhoutte, etc) when we do the comparison.

Recently we build an R package for ITIS and IHTC, later we may improve the implementation of HATC and add them to this package.

Bibliography

David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.

JAS Almeida, LMS Barbosa, AACC Pais, and SJ Formosinho. Improving hierarchical cluster analysis: A new method with outlier detection and automatic clustering. *Chemometrics and Intelligent Laboratory Systems*, 87(2):208–217, 2007.

David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

Geoffrey H Ball and David J Hall. A clustering technique for summarizing multivariate data. *Systems Research and Behavioral Science*, 12(2):153–155, 1967.

Jock A. Blackard and Denis J. Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables, 1999.

Avrim L Blum and Pat Langley. Selection of relevant features and examples in machine learning. *Artificial intelligence*, 97(1-2):245–271, 1997.

Jonathan Brogaard, Terrence Hendershott, and Ryan Riordan. High-frequency trading and price discovery. *The Review of Financial Studies*, 27(8):2267–2306, 2014.

José Ramón Cano, Francisco Herrera, and Manuel Lozano. Using evolutionary algorithms as instance selection for data reduction in kdd: an experimental study. *IEEE transactions on evolutionary computation*, 7(6):561–575, 2003.

Allen Carrion. Very fast money: High-frequency trading on the nasdaq. *Journal of Financial Markets*, 16(4):680–711, 2013.

Chien-Hsing Chou, Bo-Han Kuo, and Fu Chang. The generalized condensed nearest neighbor rule as a data reduction method. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 2, pages 556–559. IEEE, 2006.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. The algorithms of kruskal and prim. *Introduction to algorithms*, 3:631–638, 2009.

Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

Kenneth Cukier. *Data, data everywhere: A special report on managing information*. Economist Newspaper, 2010.

Mehdi Dagdoug. Black friday: A study of sales trough consumer behaviours, 2018. URL https://www.kaggle.com/mehdidag/black-friday.

Petros Drineas, Alan Frieze, Ravi Kannan, Santosh Vempala, and V Vinay. Clustering large graphs via the singular value decomposition. *Machine learning*, 56(1-3):9–33, 2004.

William DuMouchel. Data squashing: constructing summary data sets. In *Handbook of Massive Data Sets*, pages 579–591. Springer, 2002.

William DuMouchel, Chris Volinsky, Theodore Johnson, Corinna Cortes, and Daryl Pregibon. Squashing flat files flatter. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 6–15. ACM, 1999.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

Sabhia Firdaus and Md Ashraf Uddin. A survey on clustering algorithms and complexity analysis. *International Journal of Computer Science Issues (IJCSI)*, 12(2):62, 2015.

Pasi Fränti and Juha Kivijärvi. Randomised local search algorithm for the clustering problem. *Pattern Analysis & Applications*, 3(4):358–369, 2000.

Pasi Fränti, Juha Kivijärvi, Timo Kaukoranta, and Olli Nevalainen. Genetic algorithms for large-scale clustering problems. *The Computer Journal*, 40(9):547–554, 1997.

Pasi Fränti, Juha Kivijärvi, and Olli Nevalainen. Tabu search algorithm for codebook generation in vector quantization. *Pattern Recognition*, 31(8):1139–1148, 1998.

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

Peter Hart. The condensed nearest neighbor rule (corresp.). *IEEE transactions on information theory*, 14(3):515–516, 1968.

John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Unsupervised learning. In *The elements of statistical learning*, pages 485–585. Springer, 2009.

David Havera. Beijing pm2.5 data data set, 2017. URL https://www.kaggle.com/djhavera/beijing-pm25-data-data-set/activity.

Michael J Higgins, Fredrik Sävje, and Jasjeet S Sekhon. Improving massive experiments with threshold blocking. *Proceedings of the National Academy of Sciences*, 113(27):7369–7376, 2016.

Dorit S Hochbaum and David B Shmoys. A unified approach to approximation algorithms for bottleneck problems. *Journal of the ACM (JACM)*, 33(3):533–550, 1986.

E Horowitz and S Sahni. Fundamentals of complete algorithms. *Pitman, London*, 1979.

Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

Kaggle. Give me some credit. Website, 2011. URL https://www.kaggle.com/c/GiveMeSomeCredit/data. last checked: 12.08.2018.

Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 1. Pearson Education, 1997.

Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

Takio Kurita. An efficient agglomerative clustering algorithm using a heap. *Pattern Recognition*, 24(3):205–209, 1991.

Enrique Leyva, Antonio González, and Raúl Pérez. Three new instance selection methods based on local sets: A comparative study with several approaches from a bi-objective perspective. *Pattern Recognition*, 48(4):1523–1537, 2015.

Huan Liu and Hiroshi Motoda. *Feature extraction, construction and selection: A data mining perspective*, volume 453. Springer Science & Business Media, 1998.

Huan Liu and Hiroshi Motoda. On issues of instance selection. *Data Mining and Knowledge Discovery*, 6(2):115–130, 2002.

Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

Alessandra Lumini and Loris Nanni. A clustering method for automatic biometric template selection. *Pattern Recognition*, 39(3):495–497, 2006.

James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

Sreedhar Madhavaram and Shelby D. Hunt. The service-dominant logic and a hierarchy of operant resources: developing masterful operant resources and implications for marketing strategy. *Journal of the Academy of Marketing Science*, 36(1):67–82, Mar 2008. ISSN 1552-7824. doi: 10.1007/s11747-007-0063-z. URL https://doi.org/10.1007/s11747-007-0063-z.

David Madigan, Nandini Raghavan, William Dumouchel, Martha Nason, Christian Posse, and Greg Ridgeway. Likelihood-based data squashing: A modeling approach to instance construction. *Data Mining and Knowledge Discovery*, 6(2):173–190, 2002.

Jianchang Mao and Anil K Jain. A self-organizing network for hyperellipsoidal clustering (hec). *Ieee transactions on neural networks*, 7(1):16–29, 1996.

Ramón Alberto Mollineda, Francesc J Ferri, and Enrique Vidal. An efficient prototype merging strategy for the condensed 1-nn rule through class-conditional hierarchical clustering. *Pattern Recognition*, 35(12):2771–2782, 2002.

J Arturo Olvera-López, J Ariel Carrasco-Ochoa, and J Francisco Martínez-Trinidad. Object selection based on clustering and border objects. In *Computer Recognition Systems 2*, pages 27–34. Springer, 2007.

J Arturo Olvera-López, J Ariel Carrasco-Ochoa, and J Fco Martínez-Trinidad. Prototype selection via prototype relevance. In *Iberoamerican Congress on Pattern Recognition*, pages 153–160. Springer, 2008.

J Arturo Olvera-López, J Ariel Carrasco-Ochoa, J Francisco Martínez-Trinidad, and Josef

Kittler. A review of instance selection methods. *Artificial Intelligence Review*, 34(2): 133–143, 2010.

Art Owen. Data squashing by empirical likelihood. *Data Mining and Knowledge Discovery*, 7(1):101–113, 2003.

Yenisel Plasencia-Calaña, Mauricio Orozco-Alzate, Heydi Méndez-Vázquez, Edel García-Reyes, and Robert PW Duin. Towards scalable prototype selection by genetic algorithms with fast criteria. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 343–352. Springer, 2014.

Saroj Ratnoo Pooja. A comparative study of instance reduction techniques. In *Proceedings of 2nd International Conference on Emerging Trends in Engineering and Management, ICETEM*. Citeseer, 2013.

Thanapant Raicharoen and Chidchanok Lursinsap. A divide-and-conquer approach to the pairwise opposite class-nearest neighbor (poc-nn) algorithm. *Pattern recognition letters*, 26(10):1554–1567, 2005.

José C Riquelme, Jesús S Aguilar-Ruiz, and Miguel Toro. Finding representative patterns with ordered projections. *Pattern Recognition*, 36(4):1009–1018, 2003.

G Ritter, H Woodruff, S Lowry, and T Isenhour. An algorithm for a selective nearest neighbor decision rule (corresp.). *IEEE Transactions on Information Theory*, 21(6):665–669, 1975.

Fredrik Savje, Michael Higgins, and Jasjeet Sekhon. *scclust: Size-Constrained Clustering*, 2017. URL https://CRAN.R-project.org/package=scclust. R package version 0.1.1.

Robert R Sokal. A statistical method for evaluating systematic relationship. *University of Kansas science bulletin*, 28:1409–1438, 1958.

Michael Steinbach, Levent Ertöz, and Vipin Kumar. The challenges of clustering high dimensional data. In *New Directions in Statistical Physics*, pages 273–309. Springer, 2004.

Danny Sullivan. Google still doing at least 1 trillion searches per year. *Retrieved September*, 29:2015, 2015.

Ivan Tomek. An experiment with the edited nearest-neighbor rule. *IEEE Transactions on systems, Man, and Cybernetics*, (6):448–452, 1976.

Kansas State University. Acknowledging use of beocat resources and/or personnel in publications, 2018. URL https://support.beocat.ksu.edu/BeocatDocs/index.php/Policy.

Joe H Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

D Randall Wilson and Tony R Martinez. Reduction techniques for instance-based learning algorithms. *Machine learning*, 38(3):257–286, 2000.

Dennis L Wilson. Asymptotic properties of nearest neighbor rules using edited data. *IEEE Transactions on Systems, Man, and Cybernetics*, (3):408–421, 1972.

Zillow. Zillow prize: Zillows home value prediction (zestimate), 2017. URL https://www.kaggle.com/c/zillow-prize-1.

Appendix A

Cluster Performance for IHTC with Varying Threshold Size

We compare the cluster performance for IHTC with varying threshold t^* across different data size. For this example, we generate data using the multivariate Gaussian model in Section 1.4. We set $k = 3$, n is between 10^4 and 10^8, and perform one iteration of IHTC ($m = 1$). We analyze performance across different thresholds: $t^* = 2, 4, 8, 16, 32, 64, 128, 256, 512$ and 1024. The runtime, memory usage and prediction accuracy for IHTC with k-means can be found in Figures A.1 and A.3, and Table A.1. Figures A.2 and A.3, and Table A.2 present the cluster performance for IHTC with HAC.

We find, that when the threshold t^* is small, pre-processing the data with IHTC decreases runtime and memory usage compared to without pre-processing, and prediction accuracy fluctuates within a narrow range. When t^* is large, the runtime for our method takes longer than the runtime without pre-processing. In general, across all data sizes, the runtime initially decreases before steadily increasing as t^* increases. On the other hand, despite increasing the threshold t^*, the memory usage for IHTC with k-means or HAC is continually reducing. Additionally, the prediction accuracy decreases slightly with larger values of t^*.

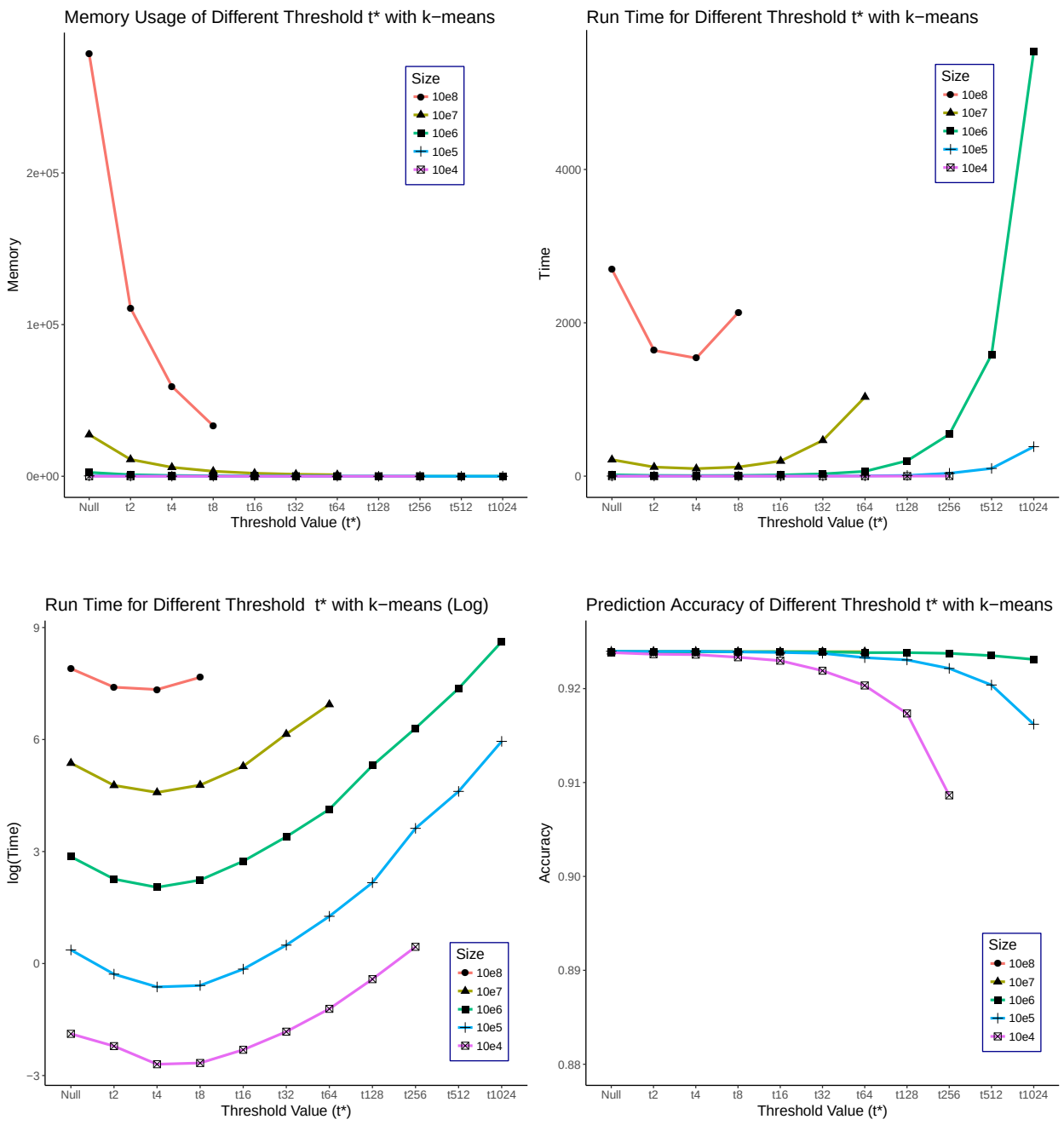


Figure A.1: *Run time and memory usage of different t^* with k-means ($k = 3, m = 1$).*

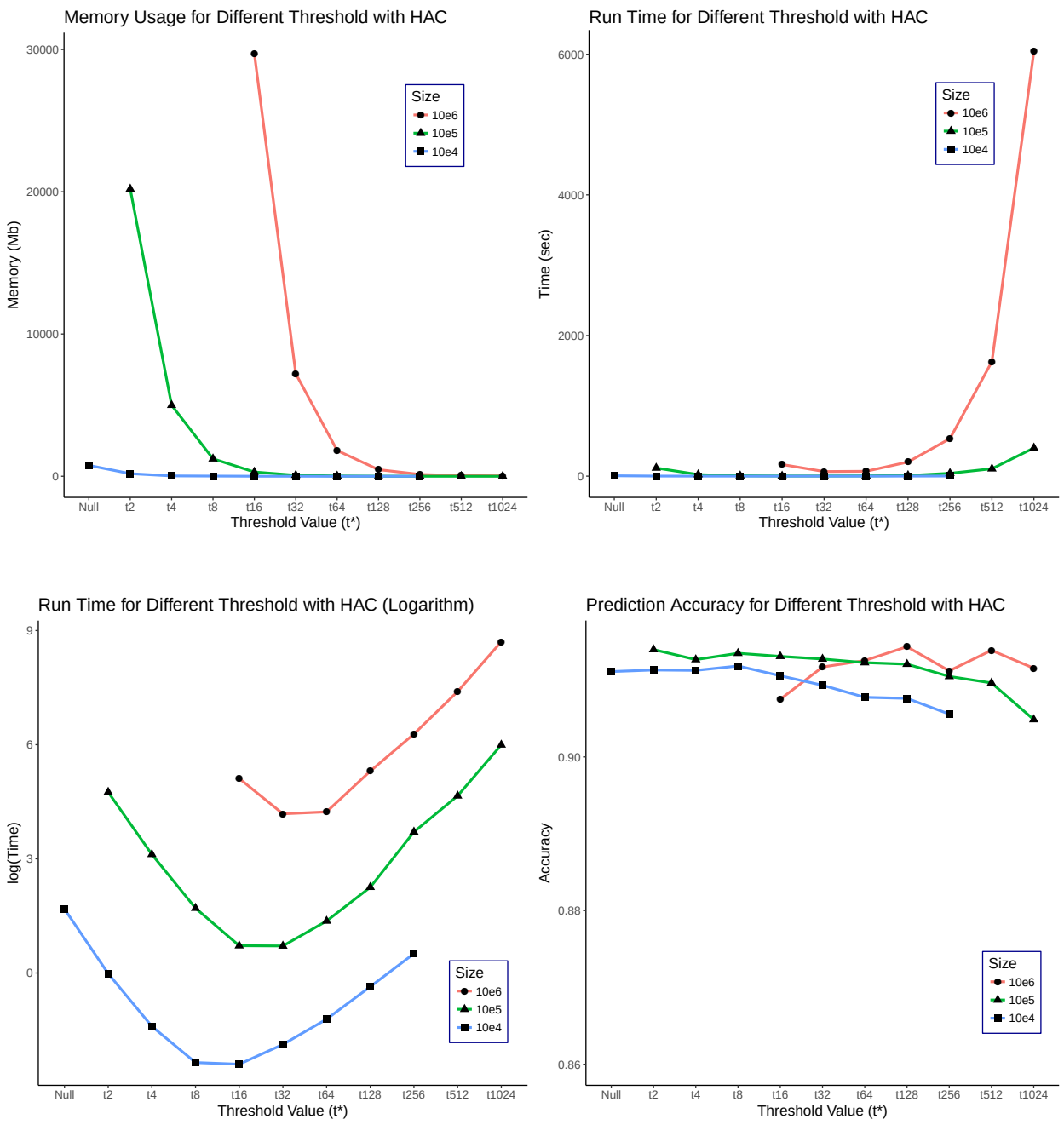


Figure A.2: *Runtime and memory usage of for different threshold t^* with HAC ($m = 1$).*

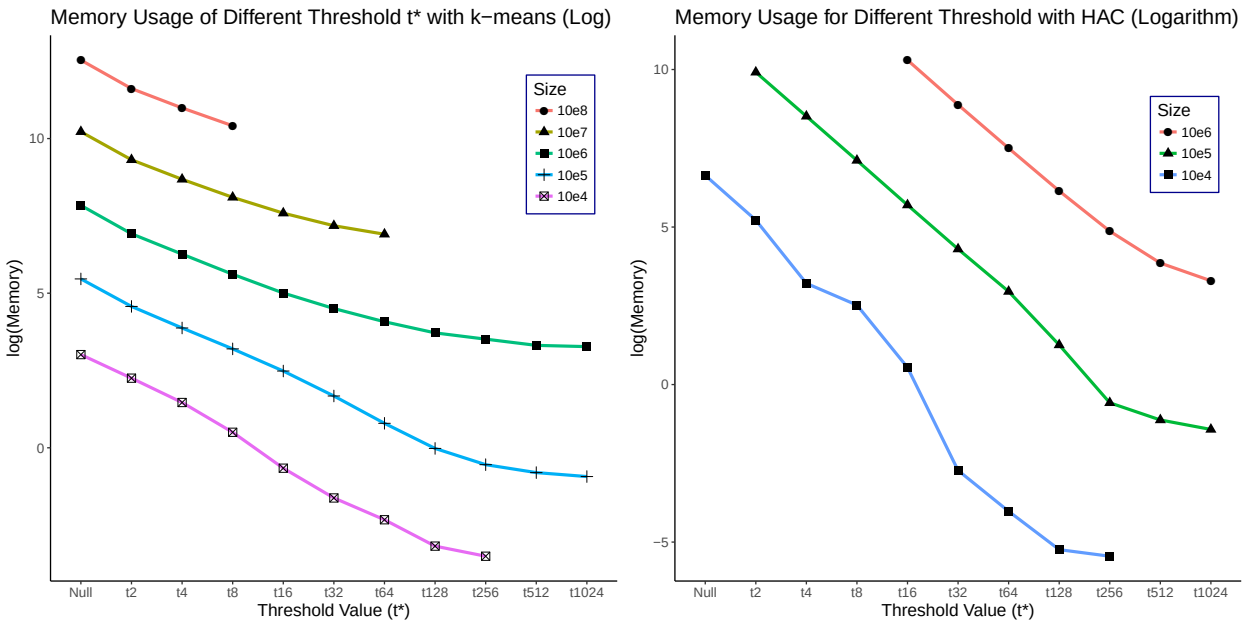


Figure A.3: *Prediction accuracy of for different threshold* t^* $(m = 1)$.

	Run Time (second)					Memory Usage (Mb)					Accuracy				
t^*	10^4	10^5	10^6	10^7	10^8	10^4	10^5	10^6	10^7	10^8	10^4	10^5	10^6	10^7	10^8
None	0.152	1.44	17.5	214	2697	20.37	235.36	2538	27468	278617	0.9238	0.9240	0.9239	0.9239	0.9239
2	0.109	0.75	9.6	118	1640	9.50	96.73	1014	11085	110919	0.9237	0.9240	0.9239	0.9239	0.9239
4	0.067	0.53	7.7	98	1540	4.33	48.17	525	5898	59288	0.9236	0.9240	0.9239	0.9239	0.9239
8	0.070	0.56	9.4	119	2137	1.65	24.50	274	3293	33316	0.9233	0.9239	0.9239	0.9239	0.9239
16	0.099	0.86	15.5	197	-	0.52	11.97	149	1975	-	0.9230	0.9238	0.9239	0.9239	-
32	0.161	1.64	29.8	467	-	0.20	5.34	90	1314	-	0.9219	0.9238	0.9239	0.9239	-
64	0.297	3.54	62.3	1032	-	0.10	2.21	59	999	-	0.9203	0.9233	0.9238	0.9239	-
128	0.658	8.72	200.8	-	-	0.04	0.98	41	-	-	0.9174	0.9231	0.9238	-	-
256	1.565	37.38	546.4	-	-	0.03	0.58	34	-	-	0.9086	0.9221	0.9238	-	-
512	-	100.56	1585	-	-	-	0.45	27	-	-	-	0.9204	0.9235	-	-
1024	-	384.17	5541	-	-	-	0.39	26	-	-	-	0.9162	0.9231	-	-

Table A.1: *Cluster performance for iterate once with k-means ($k = 3, m = 1$).*

	Run Time (s)			Memory Usage (Mb)			Accuracy		
t^*	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
Null	5.366	-	-	764.5	-	-	0.9111	-	-
2	0.984	115.9	-	184.8	20196	-	0.9113	0.9140	-
4	0.246	22.59	-	24.84	4992	-	0.9113	0.9127	-
8	0.095	5.491	-	12.46	1230	-	0.9118	0.9135	-
16	0.091	2.047	165.8	1.720	298.4	29701	0.9106	0.9131	0.9075
32	0.153	2.035	65.59	0.066	73.78	7207	0.9093	0.9127	0.9117
64	0.297	3.919	69.05	0.018	19.17	1813	0.9078	0.9123	0.9125
128	0.697	9.522	201.3	0.005	3.517	466.2	0.9076	0.9121	0.9143
256	1.662	40.73	534.3	0.004	0.561	130.2	0.9056	0.9105	0.9112
512	-	105.2	1630	-	0.325	47.29	-	0.9096	0.9139
1024	-	401.4	6042	-	0.241	26.97	-	0.9049	0.9115

Table A.2: *Simulation result for different threshold t^* with HAC ($m = 1$).*

Appendix B

Experiment for IHTC with DBSCAN

Table B.1 shows the result on the four datasets with the fewest instances. The parameters ϵ and $Minpts$ are decided by subsample of size 1000 of each dataset with a 10-fold cross-validation method. Comparing the performance for DBSCAN with and without IHTC, we find that IHTC with DBSCAN has shorter runtime but higher memory usage than DBSCAN itself. Total sum of squares (TSS) is equal to between-cluster sum of squares (BSS) plus within-cluster sum of squares. Higher ratio of BSS and TSS indicates the clusters are more compact. We found that the ratio of BSS and TSS is higher when applying IHTC, which shows our method has comparable clustering performance.

Name	Run Time (second)			Memory Usage (Mb)			BSS/TSS		
	$m=0$	$m=1$	$m=2$	$m=0$	$m=1$	$m=2$	$m=0$	$m=1$	$m=2$
PM 2.5	3.96	0.9	0.26	0.8	0.4	0.2	0.5036	0.5627	0.5336
Credit Score	25.78	4.16	2.66	3.3	1.2	38	0.4731	0.6015	0.6441
Black Fridy	9.26	0.64	0.56	13.5	62.1	66.2	0.3103	0.9657	0.9985
Covertype	233.9	223.8	231.4	13.3	15.6	15.6	0.1785	0.1785	0.1785

Table B.1: *Cluster performance for IHTC with DBSCAN ($t^* = 2$).*

Appendix C

Graph theory definitions

Let $G = (V, E)$ denote an undirected graph.

Definition C.0.1. *Vertices i and j are adjacent in G if the edge $ij \in E$.*

Definition C.0.2. *A set of vertices $I \subseteq V$ is independent in G if no vertices in the set are adjacent to each other. That is, for all $i, j \in I$, $ij \notin E$.*

Definition C.0.3. *An independent set of vertices I in G is maximal if, for any additional vertex $i \in V$, the set $i \cup I$ is not independent. That is, for all $i \in V \setminus I$, there exists $j \in I$ such that $ij \in E$.*

Definition C.0.4. *For $i, j \in V$, a walk from i to j of length m in G is a sequence of $m+1$ vertices $(i = i_0, i_1, \ldots, i_m = j)$ such that the edge $i_{\ell-1}i_\ell \in E$.*

Note that, if $(i = i_0, i_1, i_2, \cdots, i_m = j)$ is a walk of length m from i to j and the edge weights of G satisfy the triangle inequality (1.1), then the weight w_{ij} satisfies the inequality:

$$w_{ij} \leq m \max_{\ell}(w_{i_{\ell-1}i_\ell}). \tag{C.1}$$

Definition C.0.5. *The d^{th} power of G, denoted $G^d = (V, E^d)$, is a graph such that an edge $ij \in E^d$ if and only if there is a walk from i to j of length at most d in G.*