CSC730: Report for Assignment 3

South Dakota School of Mines and Technology

Carson Price, Kris Jensen

February 7, 2024

1 Introduction

Our task with assignment three is to describe the OptiGrid algorithm, analyze the Optigrid code available on Github [1], in great detail. After analyzing the algorithm and the code, we will modify the demo example to accept a 2D dataset of 30000 points and create visualizations of the data points and cutting planes. If time permits, we can earn extra credit by extending the modifications and visualization to a 3D dataset of 30000 points.

2 Description of OptiGrid

The OptiGrid algorithms works on a dataset by first determining if the dataset can be contracted, dimensionally reduced, by projecting the data onto a d-1 space. This means that if the dataset can be divided into two seperate spaces by a plane, then the dataset can be seperated into at least two classes. Simply drawing a plane through the data is insufficient to determine the best plane to divide the data.

The algorithm will determine if a plane exists to divide the data by calculating the score of the plane. The score is determined by applying a kernel density estimation function to the data in the current dataset. Depending on the kernel density esimation function and bandwidth, the score will contains some number of peaks. If there are zero or one peaks, then the splitting for this dataset is complete. If the score is above a certain threshold, then the plane is considered a good plane to divide the data. The data is then labeled as being left or right, or above or below the plane.

We will refer to left or top as A and right or bottom as B. The total dataset, D, is the union of A and B. Upon succesfull splitting of A and B, then the algorithm will recursively apply the same process to A and B. The algorithm will continue to split the dataset until the dataset is no longer able to be split.

3 Analysis of OptiGrid Code

Let's begin by setting the stage for the OptiGrid code, by analyzing the pseudocode set out by Hinneburg and Keim [2]. Then the structure of the code will be outlined, followed by a detailed analysis of each function and its purpose.

OptiGrid(dataset D, q, min_cut_score)

1. Determine a set of contracting projections P = {P_0, . . ., P_k}
2. Calculate all projections of the dataset $D \rightarrow P_0(D)$, . . ., $P_k(D)$
3. Initialize a list of cutting planes $BEST\_CUTS \leftarrow \emptyset, CUT \leftarrow \emptyset$
4. FOR i=0 TO k Do
   a. CUT ← Determine best_local_cuts($P_i(D)$)
   b. CUT_SCORE ← score_best_local_cuts($P_i(D)$)
   c. Insert all cutting planes with a score ≥ min_cut_score into $BEST\_CUTS$

   END FOR
5. IF $BEST\_CUT = \emptyset$ THEN RETURN D as a cluster
6. Determine the q cutting planes with highest score from $BEST\_CUTS$ and delete the rest
7. Construct a Multidimensional Grid G defined by the cutting planes in $BEST\_CUTS$ and insert all data points $x \in D$ into G
8. Determine clusters, i.e. determine the highly populated grid cells in G and add them to the set of cluster C
9. REFINE(C)
10. FOREACH Cluster $C_i \in C$ DO OptiGrid(C_i, q, min_cut_score)

3.1 Functions of class OptiGrid

1. **__init__(d, q, max_cut_score, ...)**
   The *__init__* functions acts as the class constructor and is called automatically upon instantiation. It initializes the class variables and sets the default values for the parameters. The parameters include dataset dimension (**d**), number of cuts per iteration (**q**), the max cut score density of a plane (**max_cut_score**), noise level for dataset(**noise_level**), several parameters related to the kernel density estimation including bandwidth (**kde_bandwidth**),

grid ticks (**kde_grid_ticks**), sample size (**kde_num_samples**), tolerance (**kde_atol**) and (**kde_rtol**), and finally an argument for turning on or off output (**verbose**) . This function sets the initial conditions of the OptiGrid algorithm.

2. **fit**(**data**, **weights**)
   The fit function is the function that is called to start the OptiGrid algorithm. It is the main function that calls all the other functions in the class. The fit function takes in the dataset and the initial weights as a parameter.

   The fit function first records the data length and initializes the list of clusters. Following this setup, the *_iteration* method is called which begins the OptiGrid algorithm.

3. **_iteration**(**data**, **weights**, **cluster_indices**, ...)
   The first step in the *_iteration* function, a pseudo-private method, is to create an empty list of cuts. The list of cuts is generated by looping through all dimensions of the dataset and calling the $\_find\_best\_cuts$ function. This loop will generate all cutting planes from d=1 to d=len(self.d). The *current_dimension* parameter sent to the $\_find\_best\_cuts$ function is incremented by 1 each iteration.

   If the list of cuts is empty, then the function returns the dataset as a cluster and indicate there are no further cutting planes available for this dataset. If the list of cuts is not empty, the list of cutting planes is sorted by score. The cutting planes discovered in the previous step are then passed to GridLevel to construct a multi-dimensional grid. The first call to GridLevel is made with the cutting planes list to construct the grid for this iteration in the recursive call stack. Then a grid of cutting planes is created from the data and clusters. This grid data contains the information if the data is left or right of the cutting plane and encoded as either 0 or 2^i

   At this point, the algorithm has created a grid and the data is labeled as being left or right, or above or below the plane. This data needs to be iterated through to recursively apply the same process to the left and right datasets if the size of cluster exceeds 0. When the first call to $self.\_iteration$ is made, the algorithm will continue to split the dataset until the dataset is no longer able to be split. Finally, when this first call returns, the algorithm will have found all the clusters in the dataset.

   This function is the top level code that implements the pseudocode for the OptiGrid algorithm. Lines 66 through 68 accomplish pseudocode lines 1 through 6. Line 81 accomplishes psuedocode line 7. Line 83 fulfills pseudocode line 8. Lines 85 through 96 accomplish psuedocode step 9 and 10.

4. **_fill_grid**(**data**, **cluster_indices**, **cuts**)
   The semi-private method $\_fill\_grid$ is called to fill the grid with the data and cluster indices. Reviewing the pseudocode, this function accomplishes step 8. A labelling scheme described in definition 5 and definition 6 of [2] is used to label the data bifurcated by the cutting planes. The method loops through all the cuts and sets the *grid_index* location to either 0 or 2^i depending on the conditional broadcasting that determines on which side of the plane the datapoints lie.

5. **_create_cuts_kde**(**data**, **cluster_indices**, **cuts**, ...)
   The semi-private method $\_create\_cuts\_kde$ is called to create the cuts along each dimension of the data. The data is first estimated by the $\_estimate\_distribution$ function along the current dimension using a kernel-density estimation. The peaks of the kernel-density estimation are found using the $\_find\_peaks\_distribution$ function. If there are no peaks in the distribution, then no further cuts are to be made. If there are peaks in the distribution, they are sorted so that only q of the peaks are used to make the cuts. The remaining peaks are then used by the $\_find\_best\_cuts$ function for the current dimension. Based on the pseudocode, this function accomplishes steps 4a through 4c.

6. **_find_best_cuts**(**grid**, **kde**, **peaks**, **current_dimension**)
   The semi-private method $\_find\_best\_cuts$ is called to find the best cuts for a dimension based on the density estimation previously computed. To find the best peaks, each set of pairs of peaks is searched between. The best cut for each pair is recorded in a *best_cuts* as the cut location, the dimension the cut was in, and the density at the point of cut. This results in a list of cuts at each of the peak minimums.

7. **_find_peaks_distribution**(**kde**)
   The semi-private method $\_find\_peaks\_distribution$ is called to locate the indices of peaks for the kde along a given dimension. A list of indices

is assembled by iterating through the iteration while tracking the previous, current, and next value along the distribution. If the previous and next values are both less than that of the current value, then the current index is added to the list of peaks.

8. **_estimate_distribution**(**data**, **cluster_indices**, **current_dimension**, **percentage_of_values**, **weights**)
The semi-private mathod *_estimate_distribution* is called to perform a kernel-based estimation of the data along a certain dimension. The data is passed through numpy's implementation of gaussian kde using a random sample of the points in the current cluster and a bandwidth scaled by the standard deviation of the random sample. The range of the data is arranged into a linspace and the density along that range is evaluated according to the kde calculated. This results in a range of data and the density of that range scaled by the total amount of data used.

9. **score_samples**
The method *_score_samples* is called to predict the cluster for a set of data samples. For each sample in the list of samples, the function creates a list of each score as calculated by the *_score_sample* function.

10. **_score_sample**
The semi-private method *_score_sample* is called to predict the cluster index that a sample belongs to. Recursing through the tree-like structure of grid level, the sublevel of the sample is found until it results in the lowest sublevel of which the sample is assigned the cluster index.

3.2 Functions of class GridLevel

1. **__init__**
The initializer function for the *GridLevel* class sets up the lists used within other functions of *GridLevel*. This set up includes initializing the cutting planes that subdivide the cluster and the cluster index. If this level of the grid is not subdivided enough, the *cluster_index* is set to None.

2. **add_subgrid**
The method *add_subgrid* is called to add another layer to the tree grid structure. Lists for the subgrids within this *GridLevel* and the indices of the subgrid are added to lists to track them within the tree structure.

3. **get_sublevel**
The method *get_sublevel* is called to recursively search for the subgrid that a given datapoint lies in. The method searches to find if the datapoint is on the left or right side of each cutting plane while encoding the grid index according to the binary system discussed in definition 6 of [2]. The index calculated is then used to search through each of the subgrids of the current level which continues until the point's grid section is found.

4 Results

4.1 2-d results

Optigrid settings: 4 Clusters, Cluster size = [10000, 20000, 20000, 20000], Noise samples of 1000, *kde_bandwidth* of 0.2, *noise_level* of 0.1, and max cut score of 1

```
Cluster 0: Mean=[-5.31 -4.93], Std=[2.19 2.39]
Cluster 1: Mean=[-1.42e-03 4.87e+00], Std=[1.00
1.57]
Cluster 2: Mean=[ 5.17 -0.09], Std=[1.76 1.93]
Cluster 3: Mean=[-6.90 19.97], Std=[2.06 1.08 ]
```

The OptiGrid algorithm did a reasonable job of clustering the data. The clusters are well separated and the noise is minimal. The clusters are not perfectly circular, but the algorithm was able to capture the general shape of the clusters.

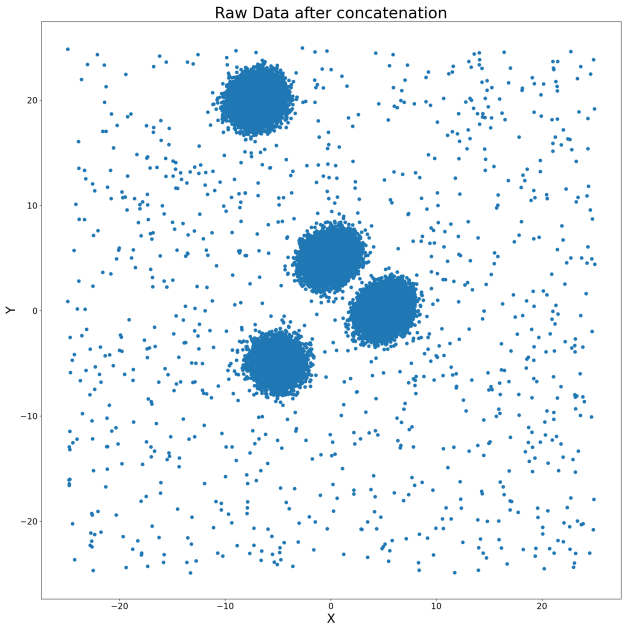

Figure 1: Raw 2-d data

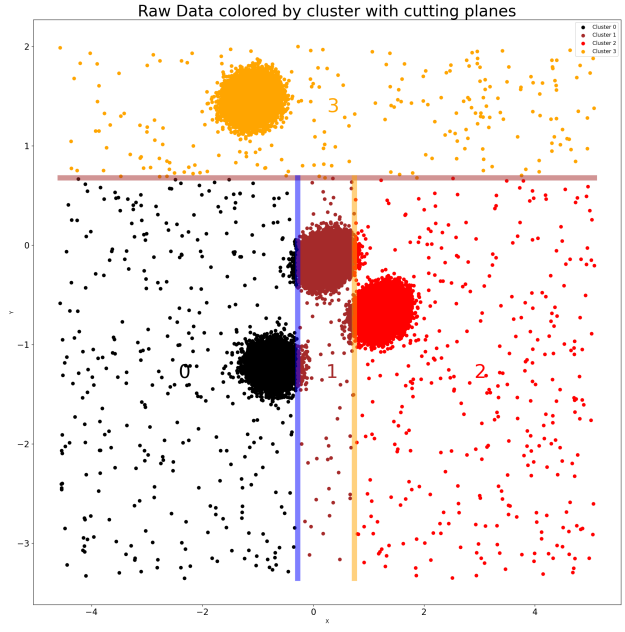

Figure 2: Clustered 2-d data

The raw data is shown in Figure 1 and the clustered data is shown in Figure 2. In the labelled figure, you can see the clusters are labelled by grid location and colored using the resistor color code. The noise is shown in the figure to show how it interacts with the grid. Adding the noise was critical to debug the grid line locations.

The algorithm chose two cutting planes in the bottom half that did not do a very good job of splitting the clusters. Also, the grid nature of the algorithm is on display with the interaction of cluster 1 with cluster 0 and cluster 2.

4.2 3-d results

In this section, we will present the results of the 3-d data. The following graphs will display the labelled cluster data and various views of the 3-d data. The clustered data is shown in Figure 3. The clusters were generated with a couple of functions that first generate a grouping of cluster centers then populates the cluster centers based on a number of parameters.

Optigrid settings: 7 Clusters, up to 1000 points per cluster, up to sigma of 2.0, *kde_bandwidth* of 0.1, *noise_level* of 0.1, max cut score of 0.3

Cluster 0: Mean=[0.35 -1.49 -1.08], Std=[0.17 0.17 0.35]
Cluster 1: Mean=[-0.53 -0.31 -0.77], Std=[0.20 0.26 0.33]
Cluster 2: Mean=[-1.36 0.83 -0.72], Std=[0.05 0.05 0.06]
Cluster 3: Mean=[-1.97 -1.98 1.14], Std=[0.06 0.09 0.11]
Cluster 4: Mean=[0.41 0.56 1.13], Std=[0.14 0.18 0.23]
Cluster 5: Mean=[1.22 0.68 0.98], Std=[0.04 0.06 0.07]

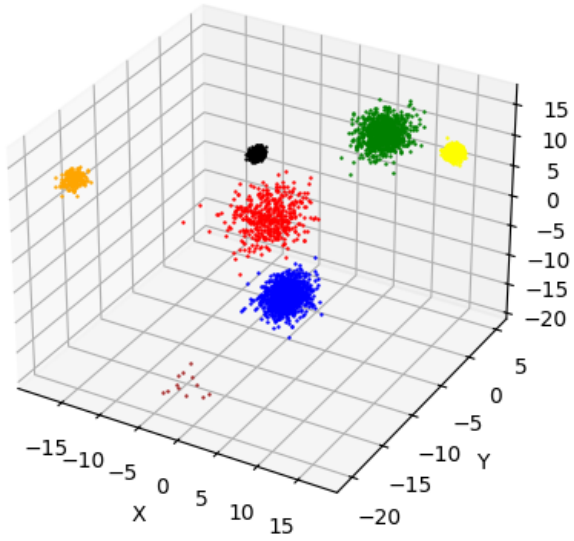

Figure 3: Clustered 3-d data

The upcoming images will show the results of

OptiGrid clustering. The first three images in the series will provide planar perspectives, while the next images will show other various perspectives.

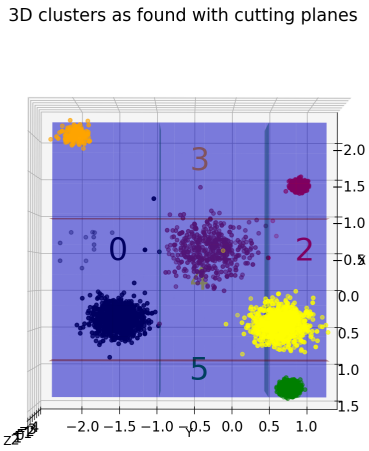

Figure 4: XY plane perspective of 3-d data

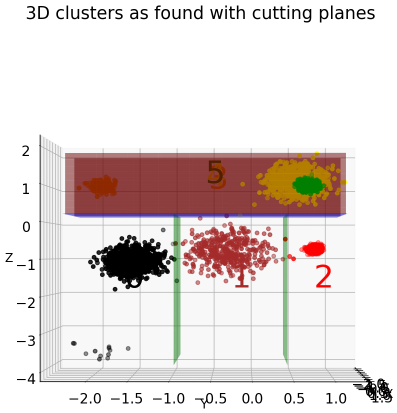

Figure 6: YZ plane perspective of 3-d data

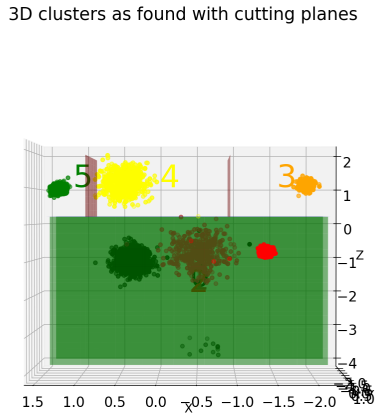

Figure 5: XZ plane perspective of 3-d data

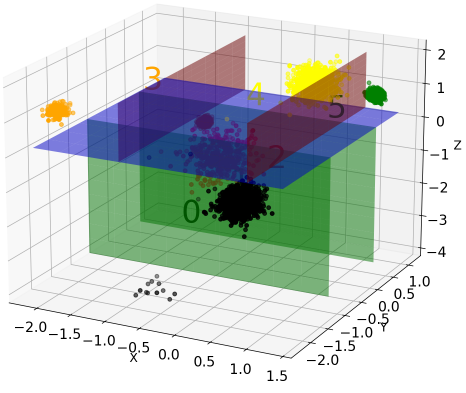

Figure 7: Perspective 1 of 3-d data

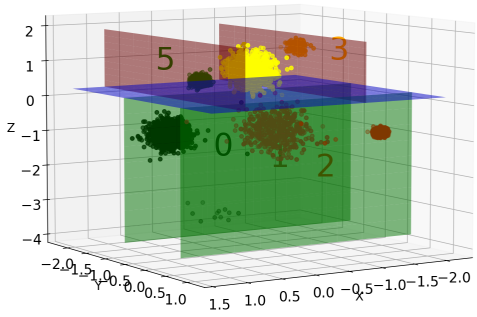

Figure 8: Perspective 2 of 3-d data

5 Conclusion

The algorithm achieved mixed results. The clusters used for this example were spread out to ensure the workings of the algorithm were correct, but in a real world application, this would not always be the case. The demo code used for the Optigrid implementation could be further improved if it allowed for cutting planes that are not all orthonormal to one another. In this way, more amorphous clusters could be split up across complex dimensions.

This project improved our knowledge of the Optigrid algorithm and of each of the algorithm's parts. Through documenting the example code given via deep description, we extended our understanding of the algorithm past that of just writing code about it. By experimenting with the demo code, we also expanded our use case of external repositories and libraries in order to produce a new result. Overall, this project helped us learn more about the Optigrid algorithm and how to use it for unsupervised learning.

6 References

[1] mihailescumihai/optigrid. (2019). GitHub. https://github.com/mihailescum/Optigrid

[2] Hinneburg, A., & Keim, D. A. (1999). Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering. In Proceedings of the 25th International Conference on Very Large Data Bases (pp. 506-517). Morgan Kaufmann Publishers Inc.