SciKit DBSCAN Implementation and Optimization Lecture Summary

Our lecture dove into the source code behind SciKit Learn to understand how the DBSCAN clustering algorithm was implemented in a more efficient manner than the theoretical pseudocode we covered in class. We began with a quick review of the DBSCAN algorithm; reviewing some terminology such as the epsilon parameter that represents the size of a point's neighborhood, and the minpts parameter that represents the number of neighbors in a point's neighborhood to consider said point as a core point. The DBSCAN algorithm basically does a depth-first search of all core points, and finds points that are density connected to them in order to form clusters.

We then took a look at what SciKit does differently. To start, the Python module allows you to input a few different parameters, such as: 'algorithm', 'leaf_size', and 'n_jobs'. These values let you specify a nearest neighbors algorithm for efficiently calculating point distances, the number of points in a region to stop further recursively dividing at, and the number of CPU cores for the calculations, respectively. You can also choose to give weights to the input dataset in order to automatically assign some points as core points. We mainly focused on the different types of nearest neighbors algorithms that could be used to avoid having to build a pairwise distance matrix for all 'n' points in the dataset. Before we got into that however, we briefly covered the analysis of these different algorithms compared to brute force distance calculations. In all cases the query time for the nearest neighbors algorithms used by SciKit were faster than a simple brute force calculation, except for when querying high amounts of neighbors. When using datasets with more and more samples the increase in efficiency actually grew when using the nearest neighbors algorithms, but with high dimensionality the efficiency difference decreases with these same algorithms.

Our look at the nearest neighbors algorithms used in SciKit Learn began with KD trees. KD trees are constructed by choosing a dimension of your dataset and sorting the points by the value of that dimension. The median point is then chosen as the split point and all points above it are put into one region and all points below are put into a separate region. The next dimension in the data is chosen and these regions are recursively divided into smaller and smaller regions, while cycling through the data's dimensions when splitting. This is done until the specified leaf size is reached, and then a tree is constructed with the split points acting as the nodes, similar to a decision tree classifier. The points in each region are then stored in the corresponding leaves of the tree. The time complexity to build a kd tree is $O(n \log n)$, much better than that of a pairwise distance matrix which is $O(n^2)$. When SciKit queries each core point to find how many neighbors it has, now the KD tree can simply be traversed to find those nearest neighbors without checking the distance to all other points in the dataset. When a KD tree is queried a queue of nearest points is held with its size being equal to the number of desired nearest neighbors. As the tree is traversed the value of the queried point is compared to each of the node's split point values and the proper subtree is recursively traversed after adding the split point to the queue. Once a leaf has been reached, brute force is used to determine the distance to all points in the region and they are added to the queue if the queue isn't full, or they replace values currently in the queue if they are less than those current values. There is an issue that arises, however, when a query point is right on the boundary of two regions because even though it belongs to one region its nearest neighbors could belong to the adjacent region. To address this issue a hypersphere is constructed around the query point and if the sphere's boundary crosses any split planes the tree is traversed back to that adjacent region and the distance to all points in that region are checked as well.

The second possible nearest neighbors algorithm that SciKit can use for efficiently calculating distances is the ball trees algorithm. Ball trees perform the same basic function as KD trees, but are constructed slightly differently. To create a ball tree the centroid of the dataset must first be calculated. The farthest point from the centroid is then determined (call it point A), and then the furthest point from A is found (call it point B). All points in the dataset, including the centroid, are then assigned to whichever point they are closer to, A or B. The two groups of points are referred to as balls or spheres. Then, the centroid of each of the balls is recalculated to factor in all the new points. This is recursively performed until the specified leaf size is reached. A tree is then constructed with the centroid of each ball acting as the nodes of the tree, and the smallest balls being the leaves of the tree in which those balls' points are stored. The time complexity of creating a ball tree is $O(n (\log n)^2)$ time which isn't as good as KD trees, but still better than a pairwise distance matrix. Similar to KD trees, a queue of nearest points is stored when querying the tree for nearest neighbors. The tree is searched in a very similar manner as well, except for the fact that the nodes of the tree aren't added to the queue while traversing the tree because they are usually just fabricated centroid values and not actual data points. The same issue arises with nearest points being in adjacent balls, so a hypersphere is drawn around the query point in the same fashion to determine if any adjacent ball points need to be checked.

Finally, we briefly touched on cover trees because they are a more widely used method than ball trees. A cover tree has several different layers of nodes, with each node being a parent to nodes in the layer below. Each layer of a cover tree has an index value i that gets smaller deeper into the tree. All nodes have a distance to their children that is less than 2^i where i is the index level of the parent. All nodes also have a distance to their siblings (other nodes in the same layer) that is greater than 2^i. Effectively this means that a points nearest neighbor is always it's parent. We didn't dive too much into the details of these trees other than they are very useful and still offer a large time efficiency advantage over brute force calculations. If you are interested in reading more about them the two articles linked below are very interesting.

Cover Trees:
https://www.cc.gatech.edu/~isbell/reading/papers/cover-tree-icml.pdf

Faster Cover Trees:
http://proceedings.mlr.press/v37/izbicki15.pdf