A Randomized Approximate Nearest Neighbors Algorithm

V. Rokhlin

Given a collection of n points $x_1, x_2, \ldots, x_n$ in $\mathbb{R}^d$ and an integer $k << n$, the task of finding the k nearest neighbors for each x_i is known as the "Nearest Neighbors Problem"; it is ubiquitous in a number of areas of Computer Science: Machine Learning, Data Mining, Artificial Intelligence, etc. The obvious algorithm costs order $d\,n^2\,log(k)$ operations, which tends to be prohibitively expensive in most non-trivial environments. There exist "fast" schemes, based on various "tree" structures. In very low dimensions, such methods are quite satisfactory; as the dimensionality increases, the algorithms become slow, and are replaced with approximate ones (i.e., instead of nearest neighbors, they find neighbors that are "somewhat close"). At some point, existing tree-based techniques become ineffective due to the notorious "curse of dimensionality"; many Machine Learning techniques can be viewed simply as attempts to avoid situations where the Nearest Neighbors Problem *has* to be solved.

I will discuss a randomized algorithm for the approximate nearest neighbor problem that is effective for fairly large values of d. The algorithm is iterative, and its CPU time requirements are of the order

$$T \cdot N \cdot (d \cdot (\log d) + k \cdot (d + \log k) \cdot (\log N)) + N \cdot k^2 \cdot (d + \log k),$$

with T the number of iterations performed; the probability of errors decreases exponentially with T. The memory requirements of the procedure are of the order $N \cdot (d + k)$.

A byproduct of the scheme is a data structure permitting a rapid search for the k nearest neighbors among $\{\boldsymbol{x}_j\}$ for an arbitrary point $\boldsymbol{x} \in \mathbb{R}^d$. The cost of each such query is proportional to

$$T \cdot (d \cdot (\log d) + \log(N/k) \cdot k \cdot (d + \log k)),$$

and the memory requirements for the requisite data structure are of the order

$$N \cdot (d + k) + T \cdot (d + N).$$

The algorithm utilizes random rotations and a basic divide-and-conquer scheme, followed by a local graph search. We analyze the scheme's behavior for certain types of distributions of $\{\boldsymbol{x}_j\}$, and illustrate its performance via several numerical examples.