Nearest Neighbor Algorithms for Text Classification

Steven Xu (350256)

1 Introduction

The k-nearest-neighbor algorithm is a method for classification, where a test instance is classified by using the majority vote of the k closest training examples in the feature space.

k-nearest-neighbors can in particular be used for text classification, where it can handle multiple classes directly, and achieves good effectiveness comparable but less than that of support vector machines.

A linear search has high computational time and memory requirements when there are a large number of documents to search through. We give a review of some more efficient algorithms, and compare them for their effectiveness on text classification.

2 Algorithms

A common approach to search for exact nearest neighbors is by organizing points in the database into a space partitioning tree, then using branch and bound to prune portions of the tree which are known to be non-optimal. Algorithms include the k-dimensional (kd) tree [2], the principal axis (pa) tree [1], and the vantage point (vp) tree [4].

However algorithms depend on a low intrinsic dimension of the dataset, as for sufficiently high dimensions none are better than linear search [3].

We describe these algorithms for finding the single nearest point. It is easy to find multiple nearest points by using a max-heap to store the best distances, then using the maximum distance in the heap for comparison during search.

2.1 KD Tree

A kd tree is a space partitioning binary tree, with an axis selected at each node to partition it's set of points. There are many ways to choose the axis, the simplest way is to cycle it amongst the dimensions of the dataset, described below.

2.1.1 Tree Construction

We start from the root node of the tree with the entire dataset of points, and select the first dimension of the dataset as the axis to partition the tree. The pivot is selected as the median of all points in the dataset, projected on the axis. The space is then partitioned in two, the points to the left and right of the pivot, assigned to the left and right child nodes. For each child i, we also store the minimum distance a projected point has to the pivot, lb_i, to aid in branch-and-bound search.

The process is then repeated for each child, with the axis selected as the one after the parent's axis. We continue until no points remain.

Each node stores the axis used, the pivot point, and the minimum distances to the pivot of the left and right child.

2.1.2 Tree Search

Let q be the query point. The distance between q and the root node's pivot is computed, and update this as the current best distance d.

For each child node i, we check whether the points in it can be closer to q than d, by intersecting a hypersphere $S(q, d)$ with the region's separating hyperplane. The distance between q and the hyperplane can be calculated by projecting the query point on the axis, computing it's distance to the pivot point, then adding lb_i. We have no intersections if this is greater than d, therefore the node can be pruned.

This is repeated until we have searched or eliminated all nodes.

2.2 Principal Axis Tree

A pa tree is similar to that of a kd tree, where the axis are selected to be the principal components of the dataset. This splits the data amongst the axis of most variance into a predefined n_c number of regions, to try to adapt and cluster the points in the dataset.

2.2.1 Tree Construction

PCA is first performed on the dataset to compute the principal axis.

Each point in the dataset is then projected onto the first principal axis, and split evenly into n_c regions, with the left and right separating hyperplanes for each region stored as projected points. This process is continued recursively until we have less than n_c points in a node, where it then becomes a terminal node.

Each node stores the principal axis, and the left and right separating hyperplanes for each region.

2.2.2 Tree Search

The depth-first search begins by projecting the query point onto the first principal axis, with a binary search to determine which region it is in. This process is repeated for child region until we reach a terminal node, where we compute the distance and store it as the current best. The algorithm then moves onto the parent node, where the elimination criterion is applied to the siblings of the terminal node.

This continues until we have searched or eliminated all nodes.

Regions closer to the one the query point is in are therefore visited first, to arrive at a smaller distance sooner.

The elimination criterion for the root regions compares it's closest separating hyperplane's distance to the query point, and the current best distance, same as a kd tree. For a child region r_n nested in parent regions $r_{n-1}, r_{n-2}, \ldots, r_1$, it can be proved, using the law of cosines [1], that for any point $x \in r_n$:

$$d(q,x)^2 \geq d(q,r_1)^2 + d(b_2,r_2)^2 + \cdots + d(b_n,r_n)^2$$

where $d(q,r_i)^2$ is the distance between q and r_i's closest separating hyperplane, and b_2 is q projected onto r_1's closest separating hyerplane, b_3 is b_2 projected onto r_2's separating hyperplane, and so forth.

If this inequality is not satisfied, the region can be pruned. For efficiency, we only need to compute the squared distance to every region.

2.3 Vantage Point Tree

Both the kd tree and the pa tree require Euclidean distances. The vantage point tree is a binary tree which can work with any metric. Cosine distance in particular can be turned into a metric.

Instead of using hyerplanes to partition the data, vantage point trees use spheres around a vantage point to partition data.

It has the theoretical property that a data distribution satisfying the Zero Point Spheres (ZPS) property (the probability of points being on a sphere is zero), and with a small enough distance bound, the nearest neighbor can be expected to be computed in $O(log(n))$ time with a database of size n [4].

2.3.1 Tree Construction

Starting from the root node, a vantage point p is chosen amongst the points in the database S. This can be random for speed, but there can be other criterions to ensure that the tree is more balanced [4].

We then define a sphere with radius $\mu = Median_{s \in S} d(p,s)$. The points in the node are partitioned in two, with the left child being inside and right child outside or on the sphere. This can be done by computing the distances between each point and p and μ. For each partition, upper and lower distance bounds to p are computed. This is performed recursively for each child node.

Each node stores p, μ, and the bounds.

2.3.2 Tree Search

The distance between the root node's pivot and the query is stored as the current best. The node visiting criterion then prefers to visit the left node if this is closer to the left child, or right node otherwise. A node is visited if the distance is within it's bounds.

This is repeated until we have searched or eliminated all nodes.

3 Experimental Setup

30,000 documents were used from the RCV1v2 document collection. To pre-process each document, the bag of words model was used with unit-length normalized tf-idf weights using the l_2 metric, with the following formulas:

$$tf_{d,t} = log(1 + f_{d,t})$$
$$idf_t = log(\frac{N}{f_t})$$

where $f_{d,t}$ is the number of times term t appears in the document d; $f_t$is the number of documents term t appears in throughout the collection; and N is the number of documents in the collection.

Euclidean distance was used as the distance function, which is rank-equivalent to cosine similarity due to unit-length normalization.

To compute the principal axes, power iteration was used with a maximum of 1,000 iterations.

We implemented all of the above algorithms in python. Due to python's performance limits compared to the linear search's C implementation, we evaluated query performance based on the number of distances computed instead of the total time spent.

Partial distance search were not used due to it's negligble impact on performance.

4 Experimental Results

4.1 Query Performance

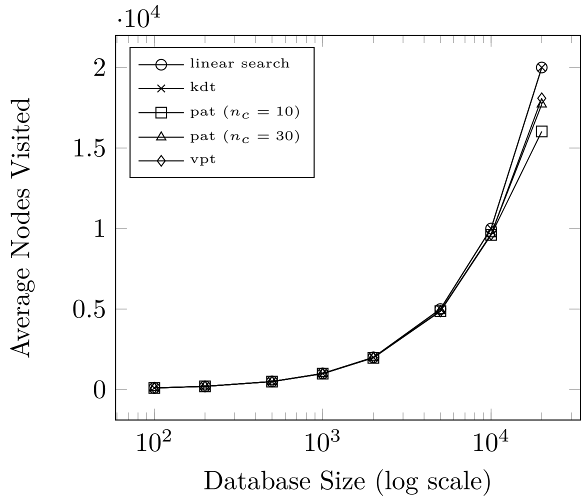

Figure 1: Database size vs. average nodes visited for 50 random test documents. kd trees were no better than a linear search.

None of the algorithms appeared to exhibit logarithmic query time, but both vp trees and pa trees performed better than linear search on average.

The performance benefit decreased when the minimum distance increased.

4.1.1 KD Tree

The kd tree was expected to perform poorly due to the high dimensionality of the data. This is since it reduces the effectiveness of the bound lb_i, which only involves a single dimension.

But it did not perform better at all compared to linear search. This can possibly be explained by the large number of dimensions involving rare words, which leads to the two partitions being highly unbalanced, therefore a highly unbalanced search tree. Since rare words also have low weights, this decreases lb_i further.

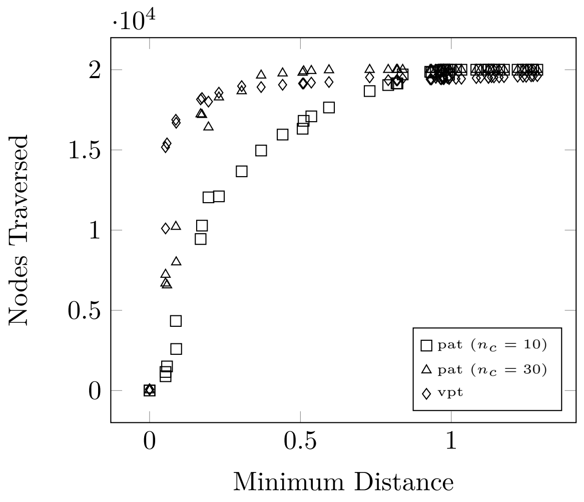

Figure 2: Nodes traversed vs minimum distance for all test points, for a database of size 20000.

4.1.2 Principal Axis Tree

Better choice of axes compared to the kd-tree leads to better efficiency. A smaller n_c also appears to be better.

Despite the lowest average query times, pa trees had worse performance when the minimum distance was large enough. In particular, there were 42% of test documents where all leaf nodes were visited for $n_c = 10$, and 64% for $n_c = 30$. This requires further investigation.

4.1.3 Vantage Point Tree

The ZPS property is not completely satisfied, as unit length normalization implies that all database points lie on a sphere. However the search tree still gave reasonable results.

4.2 Tree Construction

The pa tree had sudden jumps in time and memory. They could be explained by the increase in the depth of the search tree, leading to more principal component computations. More documents are required to see the $O(n \cdot log(n))$ behaviour.

The main storage costs of the pa tree come from the non-sparse principal axis. This could be decreased by using sparse PCA instead, so that there is linear instead of quadratic growth. The current implementation

The vp tree shows approximately $O(n \cdot log(n))$ time and memory use.

Overall, in our implementation, no search algorithm achieved significantly better query perfor-

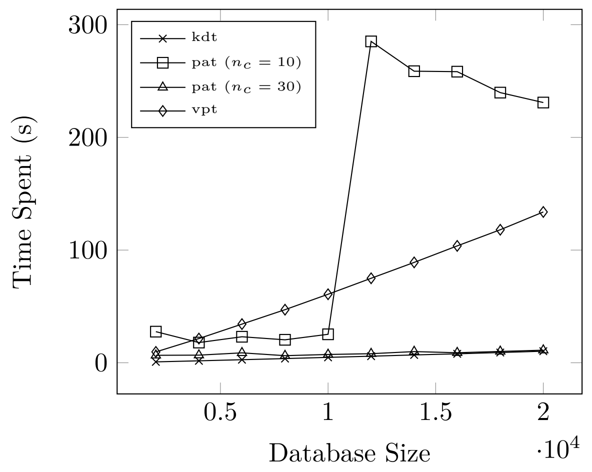

Figure 3: Tree construction CPU time.

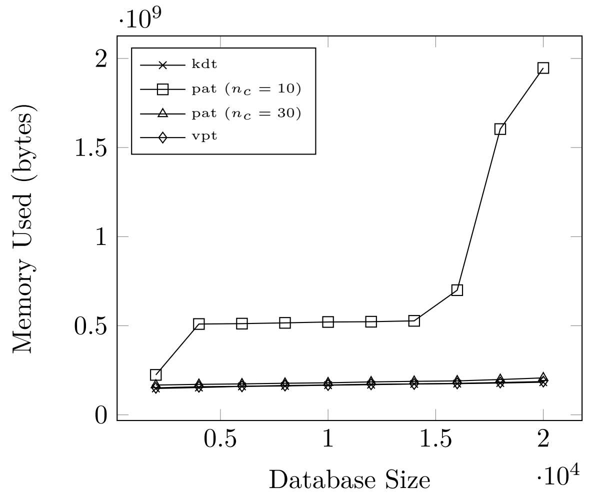

Figure 4: Tree process memory usage.

mance compared to linear search, but incurred much higher time and memory costs during tree construction. Therefore, unless it is known in advance that the query point is very close to some existing point, linear search is the most practical solution.

5 Concluding Remarks

We implemented and investigated three exact nearest neighbor algorithms based on space partitioning trees. None appeared to be particularly suitable to the problem of text classification.

There were however some positive results when the query point was close to an existing point in the database. This could be a point of further investigation to improve these algorithms.

Other exact nearest neighbor and approximate nearest neighbor techniques could be used as comparison in the future.

6 Bibliography

References

[1] James McNames. A fast nearest-neighbor algorithm based on a principal axis search tree. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(9):964–976, 2001.

[2] Robert F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(1-6):579–589, 1991.

[3] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, page 194–205, 1998.

[4] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, page 311–321. Society for Industrial and Applied Mathematics, 1993.