K-Nearest Neighbors

How KNN works in Classification and Regression problems?

Data science and Machine learning Online Course

INTRODUCTION:

The nearest neighbour method (k-Nearest Neighbors, or k-NN) is another very popular classification method that is also sometimes used in regression problems. This, like decision trees, is one of the most comprehensible approaches to classification. The underlying intuition is that you look like your neighbors. More formally, the method follows the compactness hypothesis: if the distance between the examples is measured well enough, then similar examples are much more likely to belong to the same class.

HYPOTHESIS:

To classify each sample from the test set, one needs to perform the following operations in order:

1. Calculate the distance to each of the samples in the training set.
2. Select *k* samples from the training set with the minimal distance to them.
3. The class of the test sample will be the most frequent class among those *k* nearest neighbors.

The method adapts quite easily for the regression problem: on step 3, it returns not the class, but the number — a mean (or median) of the target variable among neighbors.

There exist many important theorems claiming that, on "endless" datasets, it is the optimal method of classification. We consider k-NN to be a theoretically ideal algorithm which usage is only limited by computation power and the curse of dimensionality.

NEAREST NEIGHBORS METHOD IN REAL APPLICATIONS:

- k-NN can serve as a good starting point (baseline) in some cases;
- In Kaggle competitions, k-NN is often used for the construction of meta-features (i.e. k-NN predictions as input to other models) or for stacking/blending;
- The nearest neighbors method extends to other tasks like recommendation systems. The initial decision could be a recommendation of a product (or service) that is popular among the *closest neighbors* of the person for whom we want to make a recommendation;
- In practice, on large datasets, approximate methods of search are often used for nearest neighbors.

QUALITY OF CLASSIFICATION/ REGRESSION WITH KNN:

The quality of classification/regression with k-NN depends on several parameters:

- The number of neighbors *k*.
- The distance measure between samples (common ones include Hamming, Euclidean, cosine, and Minkowski distances). Note that most of these metrics require data to be scaled. Simply speaking, we do not want the “salary” feature, which is on the order of thousands, to affect

the distance more than “age”, which is generally less than 100.
- Weights of neighbors (each neighbor may contribute different weights; for example, the further the sample, the lower the weight).

CLASS KNeighborsClassifier IN SCIKIT-LEARN:

sklearn.neighbors.KNeighborsClassifier parameters:

- weights: uniform (all weights are equal), distance (the weight is inversely proportional to the distance from the test sample), or any other user-defined function;
- algorithm (optional): brute, ball_tree, KD_tree, or auto. In the first case, the nearest neighbors for each test case are computed by a grid search over the training set. In the second and third cases, the distances between the examples are stored in a tree to accelerate finding nearest neighbors. If you set this parameter to auto, the right way to find the neighbors will be automatically chosen based on the training set.
- leaf_size (optional): threshold for switching to grid search if the algorithm for finding neighbors is BallTree or KDTree;
- metric: minkowski, manhattan, euclidean, chebyshev, or other.

NEAREST NEIGHBORS ALGORITHMS:

Brute force:

Algorithm:

The most naive neighbor search implementation involves the brute-force computation of distances between all pairs of points in the dataset: for N samples in D dimensions, this approach scales as O[DN2].

- Efficient brute-force neighbors searches can be very competitive for small data samples.
- As the number of samples N grows, the brute-force approach quickly becomes infeasible.

K-D Tree:

- To address the computational inefficiencies of the brute-force approach, a variety of tree-based data structures have been invented
- The basic idea is that if point A is very distant from point B, and point B is very close to point C, then we know that points A and C are very distant, *without having to explicitly calculate their distance*.
- The computational cost of a nearest neighbors search can be reduced to O[DNlog(N)] or better.
- Though the KD tree approach is very fast for low-dimensional (D<20) neighbors searches, it becomes inefficient as D grows very large: this is one manifestation of the so-called “curse of dimensionality”.

Ball Tree:

- To address the inefficiencies of KD Trees in higher dimensions, the *ball tree* data structure was developed.
- Where KD trees partition data along Cartesian axes, ball trees partition data in a series of nesting hyper-spheres.
- This makes tree construction more costly than that of the KD tree, but results in a data structure which can be very efficient on highly structured data, even in very high dimensions.

Algorithm:

A ball tree recursively divides the data into nodes defined by a centroid C and radius r, such that each point in the node lies within the hyper-sphere defined by r and C. The number of candidate points for a neighbor search is reduced through use of the *triangle inequality*:

$$|x + y| < |x| + |y|$$

With this setup, a single distance calculation between a test point and the centroid is sufficient to determine a lower and upper bound on the distance to all points within the node.

CHOICE OF NEAREST NEIGHBORS ALGORITHM:

- *Brute force* query time grows as O[DN]
- *Ball tree* query time grows as approximately O[Dlog(N)]
- *KD tree* query time changes with D in a way that is difficult to precisely characterise. For small D (less than 20 or so) the cost is approximately O[Dlog(N)], and the KD tree query can be very efficient. For larger D, the cost increases to nearly O[DN], and the overhead due to the tree structure can lead to queries which are slower than brute force.

1.