Open in app Get started

Shubhtripathi Follow
Jan 10, 2021 · 8 min read · Listen

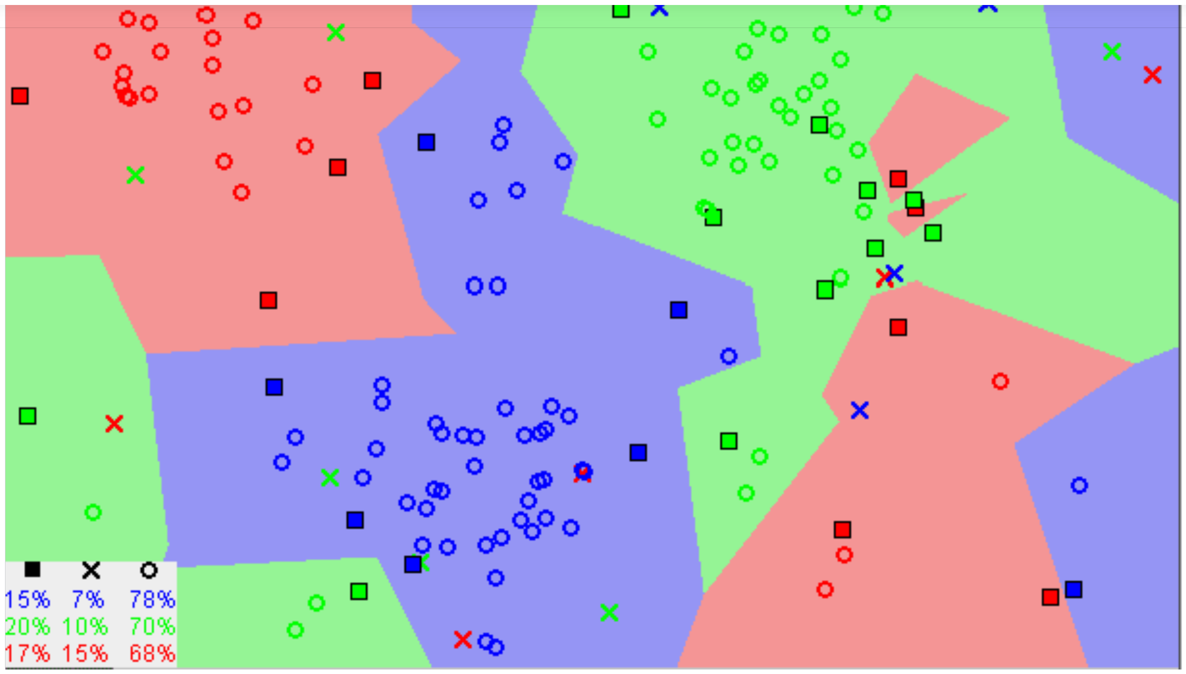
Save

KNN

In this blog we will cover KNN and some commonly used methods to implement it.

k-NN(k-Nearest Neighbors) is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until function evaluation. Since this algorithm relies on distance for classification, normalizing the training data can improve its accuracy dramatically. It assumes that similar things exist in close proximity. In other words, similar things are near to each other. More this assumption is true for the data, more accurate result it leads to.

It is a non-parametric algorithm, *i.e.* it doesn't make any assumption for distribution of data.



The output depends on whether *k*-NN is used for classification or regression:

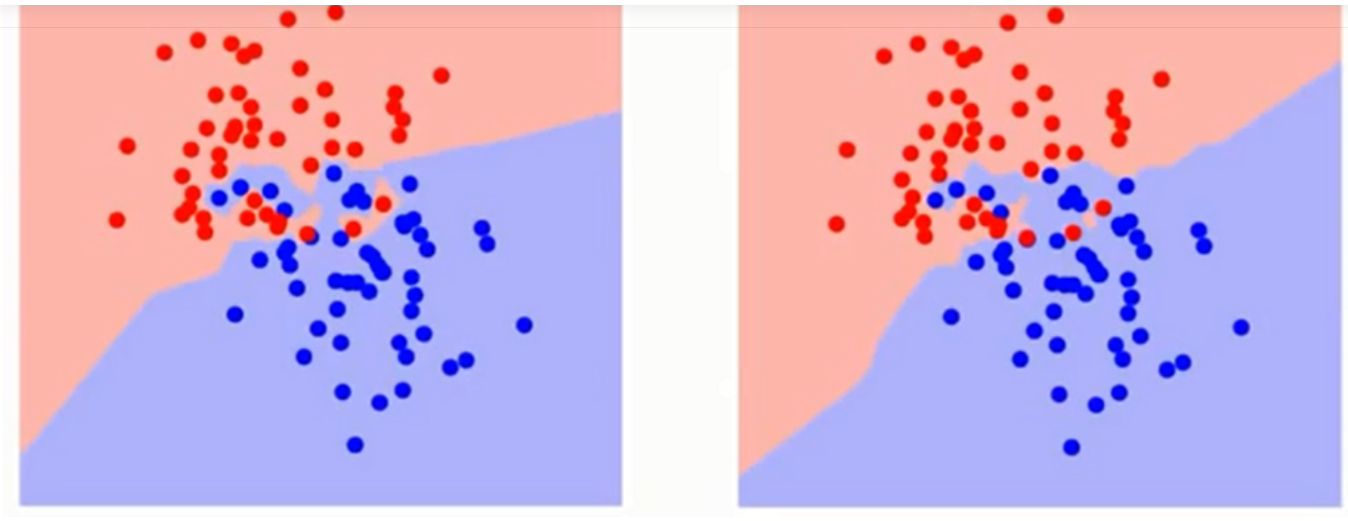
- In *k-NN classification*, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its *k* nearest neighbors (*k* is a positive integer, typically small). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.
- In *k-NN regression*, the output is the property value for the object. This value is the average of the values of *k* nearest neighbors.

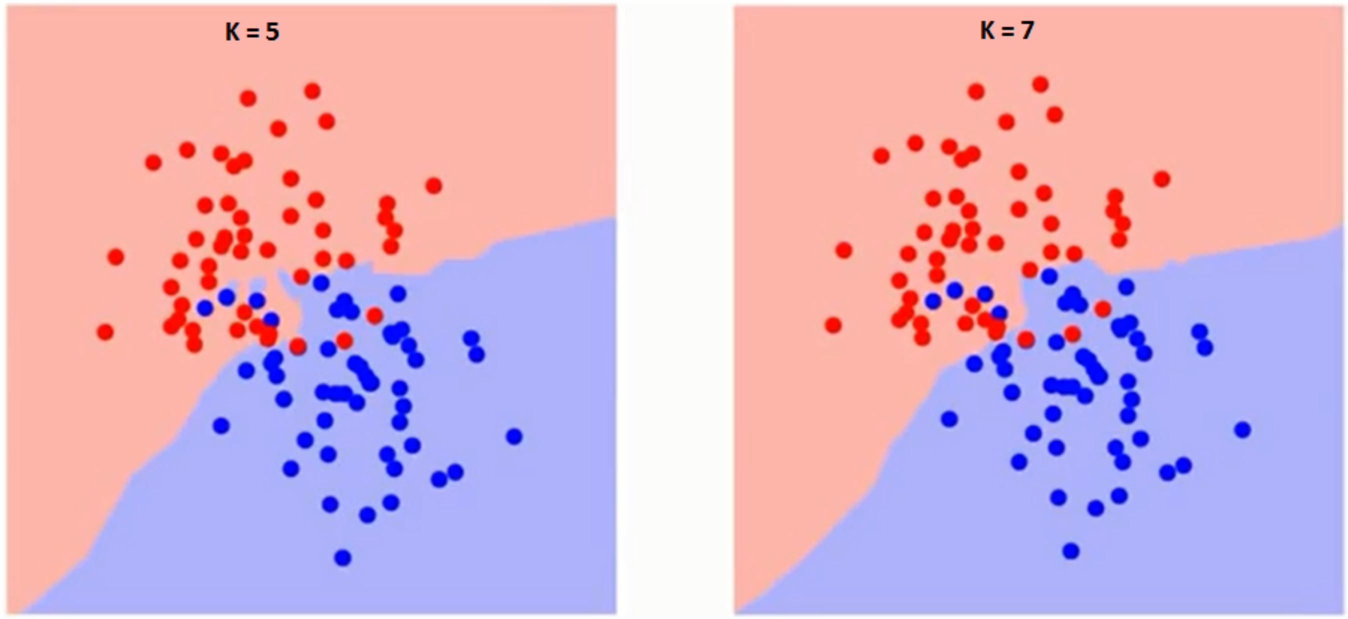
However, it is more widely used in classification problems in the industry.

A commonly used distance metric for continuous variables is Euclidean distance. For discrete variables, such as for text classification, another metric can be used, such as the **overlap metric** (or Hamming distance).

Choosing hyperparameter-

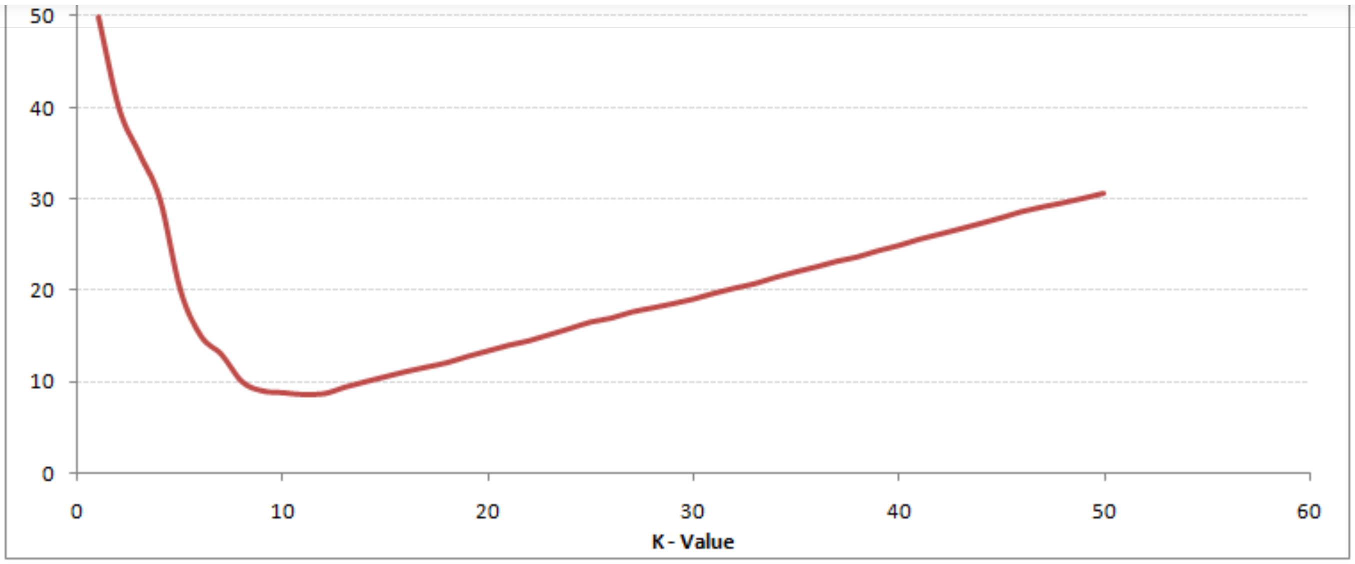
Open in app Get started





If you watch carefully, you can see that the boundary becomes smoother with increasing value of K. With K increasing to infinity it finally becomes all blue or all red depending on the total majority.

Very small value of *k leads to overfitting* as it considers only few points. Also,it can be easily influenced by outliers. Even very large value of k leads to high error as it starts to take more global structure and running out of it’s boundary. Hence, a good value of k is very necessary.



To select the K that's right for your data, we run the KNN algorithm several times with different values of K and choose the K that reduces the number of errors we encounter while maintaining the algorithm's ability to accurately make predictions when it's given data it hasn't seen before.

The best choice of k depends upon the data; generally, larger values of k reduces effect of the noise on the classification, but make boundaries between classes less distinct. The special case where the class is predicted to be the class of the closest training sample (i.e. when $k = 1$) is called the nearest neighbor algorithm. We take odd values of k to avoid ties.

Implementation-

We can implement a KNN model by following the below steps:

1. Load the data
2. Initialize K to your chosen number of neighbors

3. For each example in the data

3.1 Calculate the distance between the query example and the current example from the data.

5. Pick the first K entries from the sorted collection

6. Get the labels of the selected K entries

7. If regression, return the mean of the K labels

8. If classification, return the mode of the K labels

We can implement it in sklearn as-

> *from sklearn.neighbors import KNeighborsClassifier*
>
> *from sklearn.neighbors import KNeighborsRegressor*

The basic nearest neighbors classification uses uniform weights: that is, the value assigned to a query point is computed from a simple majority vote of the nearest neighbors. Under some circumstances, it is better to weight the neighbors such that nearer neighbors contribute more to the fit. This can be accomplished through the `weights` keyword. The default value, `weights = 'uniform'`, assigns uniform weights to each neighbor. `weights = 'distance'` assigns weights proportional to the inverse of the distance from the query point.

Algorithms for implementing K-NN in sklearn-

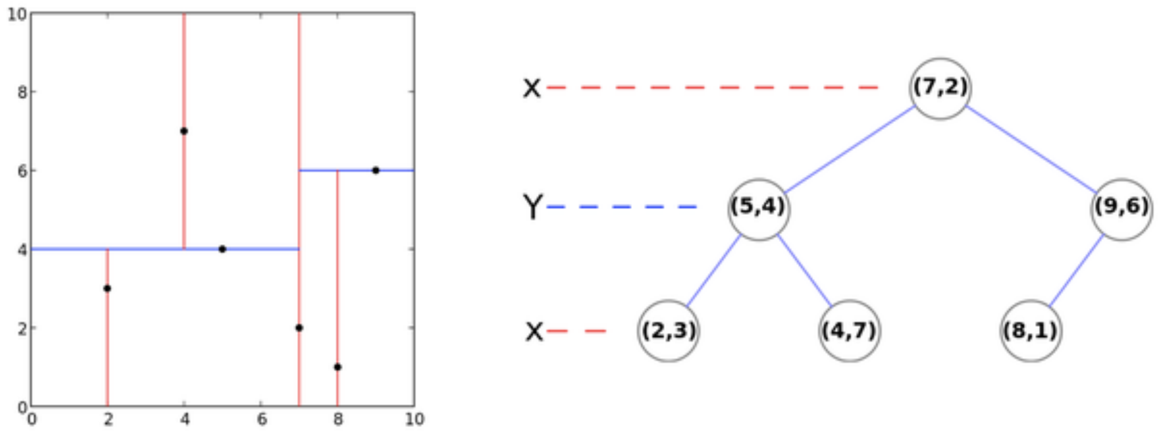
We have a 'algorithm' parameter in sklearn's KNN. We can set it to implement KNN by different algorithms. The choices are-

algorithm= 'brute'-

This uses brute approach which is discussed as above. It's complexity is O(D*N*N) which is very high. Hence we use K-D tree for reducing the comlexity. For small sample sizes a brute force search can be more efficient than a tree-based query.

algorithm= 'kd_tree'-

Basically, what it does is that it constructs a binary tree on basis of splitting across some cost increases to nearly O[DN], and the overhead due to the tree structure can lead to queries which are slower than brute force. For addressing this, we use ball tree. *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.*



Splitting looks like this when we split on every data point.

algorithm='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.

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*:

$|x+y| \leq |x|+|y|$

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. It's time complexity is O[D*log(N)] the number of query points is at least the same order as the number of training points, and that `leaf_size` is close to its default value of `30`.

> ***Note:-*** *Neither KD-tree, nor Ball Tree guarantees that the split will keep true nearest points close in a tree. In this sense, the use of a tree implies "information loss", and the larger the tree, the higher the chance to put the true neighbors into a wrong branch. In extreme case, there's simply no tree at all (called* brute force*), so all points are the candidates for the neighbours, as a result the algorithm is guaranteed to find the exact solution.*

leaf_size parameter in knn sklearn-

It is tuned when kd tree or ball tree is used. The leaf size controls the minimum number of points in a given node, and effectively adjusts the tradeoff between the cost of node traversal and the cost of a brute-force distance estimate.

Note that with larger leaf size, the build time decreases: this is because fewer nodes need to be built. For the query times, we see a distinct minimum. For very small leaf sizes, the query slows down because the algorithm must access many nodes to complete the query. For very large leaf sizes, the query slows down because there are too many pairwise distance computations. If we were to use a less efficient metric function, the balance between these would change and a larger leaf size would be warranted.

There's also a **RadiusNeighborsClassifier** and **RadiusNeighborsRegressor** in sklearn which vote among neighbors within a given radius. But it can be only used for low dimensional data. Also it is used rarely.

Advantages

1. The algorithm is simple and easy to implement.
2. There's no need to build a model, tune several parameters, or make additional assumptions.
3. The algorithm is versatile. It can be used for classification, regression, and search
4. Most of the classifier algorithms are easy to implement for binary problems and needs

Open in app Get started

Drawbacks-

1. The algorithm gets significantly slower as the number of examples and/or predictors/independent variables increase as it have to account for every training point during prediction.

2. **Does not work well with high dimensionality** as this will complicate the distance calculating process to calculate distance for each dimension. Because the distance will be larger and larger, it will become more and more “unlike”, and for KNN, a highly distance-dependent algorithm, this will also affect the accuracy.

3. A drawback of the basic “majority v cation occurs when the class distribution is skewed. That is, examples of a more frequent class tend to dominate the prediction of the new example, because they tend to be common among the *k* nearest neighbors due to their large number. One way to overcome this problem is to weight the classification, taking into account the distance from the test point to each of its *k* nearest neighbors. The class (or value, in regression problems) of each of the *k* nearest points is multiplied by a weight proportional to the inverse of the distance from that point to the test point. Another way to overcome skew is by abstraction in data representation.

About Help Terms Privacy

4. The accuracy of the *k*-NN algorithm can be severely degraded by the presence of noisy or irrelevant features, or if the feature scales are not consistent with their importance.

Get the Medium app

Download on the App Store GET IT ON Google Play

5. iti also has no capability to deal with missing values.

6. KNN doesn’t know which attributes are more important. Also it is a non-interpretable model.

As KNN’s main disadvantage is lot’s of time, it is not preferred for practical use. Also there are regression or classification algorithms that give better accuracy in less time. However, there are other applications of KNN like finding missing value in dataset, ordering data on basis of similarity, etc.