Nearest Neighbors Classifiers

Raúl Rojas
Freie Universität Berlin

July 2014

In pattern recognition we want to analyze data sets of many different types (pictures, vectors of health symptoms, audio streams, etc.) and classify each piece of data as belonging to one of several possible classes. In a picture, for example, we would like to find out if a face is present (the classes would be "face" and "no-face"). In a piece of audio data we could be looking for one of several possible phonemes. Typically, we do not deal with the raw data directly. We transform first the raw data into a vector of numerical "features". These encode what, for our purposes, is important about the data. When dealing with speech, we could, for example, compute the total energy present in 10 frequency bands. The ten numbers obtained would represent the "extracted features".

Pattern recognition typically works in a pipeline as shown in Fig. 1.

Figure 1: The pattern recognition pipeline

Feature extraction is tightly coupled to the kind of classifier we want to use. Not surprisingly, good features are those which produce better classifiers. If we think of the data in feature space we would like to have a good spatial separation, that is, given a data set containing examples from two classes, we would like the data points for each class to occupy easily distinguishable (or separable) regions of space.

There are two possible ways of describing the data cloud produced by a data set. In *parametric methods* we model each class using a specific probability distribution (such as a Gauß or Poisson distribution). In non-parametric methods we do not presuppose an analytic distribution and try to model the cloud of data using the

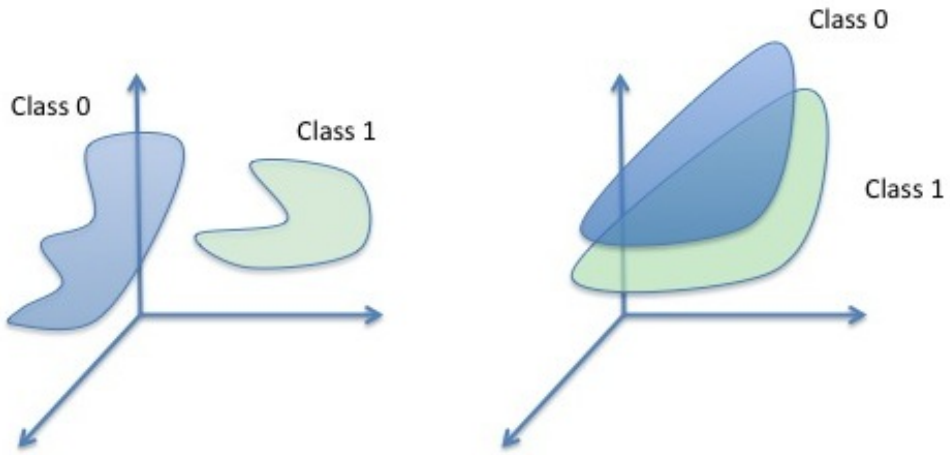

Figure 2: The data from two classes. In the feature space to the left, the classes are easily separable. In the feature space to the right, the classes overlap

data itself or "representatives" of the location of the data points. In parametric methods we have "parameters" such as the mean $\vec{\mu}$ of the distribution, and its covariance Σ. If we use a Gaussian multivariate distribution, we can displace the model by varying $\vec{\mu}$, we can increase or decrease the spread of the distribution by modifying Σ.

The simplest non-parametric classifier is the k-NN classifier (k nearest neighbors). If we have a data set containing two classes, we store the feature vectors $\vec{x}_i$ and their class y_i as pairs $(\vec{x}_i, y_i)$ When given a new data chunk $\vec{x}$ of unknown class, we just compute he distance of $\vec{x}$ to all N data points $\vec{x}_i$. If $\vec{x}_j$ is the nearest neighbor of $\vec{x}$ (that is, the point with the smallest distance to $\vec{x}$), we assign $\vec{x}$ the class y_j. This would be a 1-NN approach. If we look at the k nearest neighbors and take a majority vote, we have a k-NN classifier. It is that simple.

How good is a k-NN classifier?

Surprisingly, a 1-NN classifier is not that bad, when the number of data points is large, so that the probability density of the data set is well approximated.

If we know the distributions of the data for two classes, we could use the approach illustrated in Fig. 3.

Given the data distributions shown, we would always pick class A for data right of the decision boundary, and class B otherwise. There is always the possibility of error, for example for the point x, since the probability $p(A|x)$ and $p(B|x)$ are both non-zero. But since $p(A|x) > p(B|x)$, in the figure, we would always pick

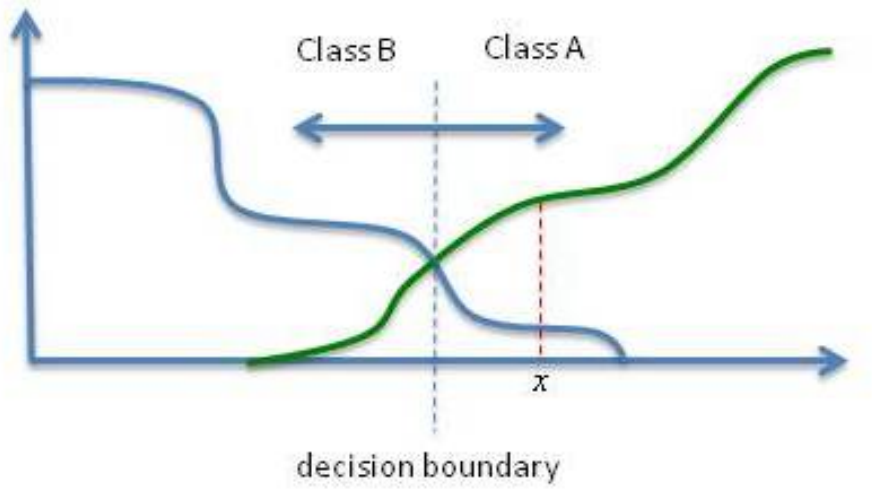

Figure 3: The probability distribution of two classes along a single numerical feature

class A over B, since that minimizes the error rate. The error rate is $p(B|x)$ when there are only two classes and $p(A|x) + p(B|x) = 1$.

Now consider what happens when we look at a small slice of the one-dimensional distribution, as shown in Fig. 4.

A few data points of the empirical distribution fall in the small segment (represented by black and red crosses for the two classes). A percentage P_A of the points is of class A. A percentage P_B is of class B. If that is all we know, P_A and P_B are the Bayes probabilities of both classes in the small slice which has been selected. If we do not have any information about the distribution of the data points in the slice, but we know P_A and P_B, and also $P_A > P_B$, we always assign class A for any new data point which happens to fall in this slice. This is what a Bayes classifier would do. The error rate for a classification based on just the knowledge of P_A and P_B is P_B (since a fraction P_B of the classifications will be wrong, when we always assign class A). However, the error rate for a 1-NN classifier (which sometimes assigns the class A and sometimes the class B) would be different.

There are two cases: the new point we are classifying is of class A. If we look for its nearest neighbor in the data, and since class A dominates P_A of the slice volume, with probability P_A it will be correctly classified. But the nearest neighbor could be a point of class B (since we now know how the training set data points are distributed in the slice). The point would be then misclassified with probability P_B

And conversely: if the point to be classified is of class B, with probability P_A it will be misclassified as of class A, while with probability P_B it will be correctly

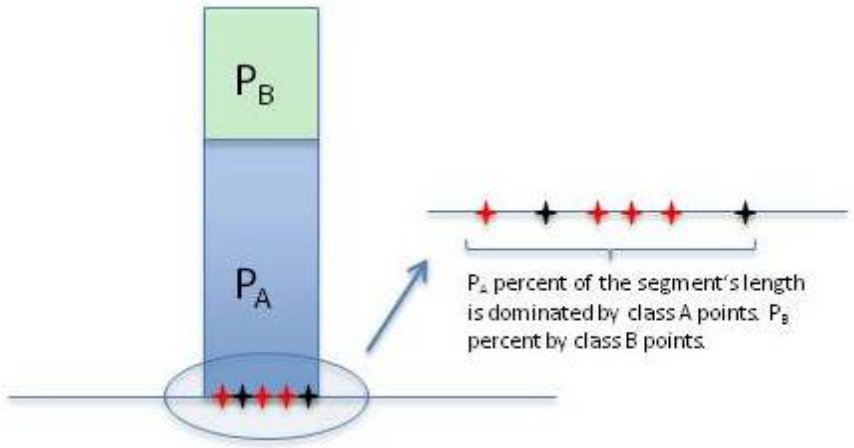

Figure 4: A slice of the probability distribution of two classes. The "maginification" shows that in this slice of data space points of class A dominate a fraction P_A of the slice, as nearest neighbors, while points of class B dominate a fraction P_B.

classified. That means: The expected error rate of the 1-NN classifier is then

$$r = P_A P_B + P_B P_A.$$

If $P_A + P_B = 1$ we have

$$r = 2P_B(1 - P_B) = 2P_B - 2P_B^2.$$

For example, if $P_B = 5\%$, the Bayes classifier with knowledge only of the P_A and P_B values will always select class A (since $P_A > P_B$) and the error rate will be $P_B = 5\%$.

The 1-NN classifier, with knowledge of the distribution of data points, will have an expected error rate $r = 0.1 - 2 \times 0.0025 \approx 10\%$. That is, a simple 1-NN classifier has an error rate which is *at most* twice as bad as the Bayes classifier which has knowledge of the real P_A and P_B. This result is valid for one slice of the data space and also for the whole data space, once we integrate the error rate across all slices of the distribution.

This simple result is rather striking. A k-NN classifier is almost trivial – but is not that bad after all. The difference between P_B and $2P_B - 2P_B^2$ has a maximum at $P_B = \frac{1}{4}$. When the optimal classifier has error $\frac{1}{4}$, the 1-NN classifier would have an expected error of $\frac{3}{8}$.

Organizing the data in trees

The disadvantage of k-NN classifiers is the Big Data problem. Keeping all N data points in storage and computing N distances for each k-NN classification is

wasteful of memory and processing time. Can we do better, for example organizing the data in binary trees for a faster search?

The answer is a qualified *yes*. In some cases data strucures such as quad-trees or *kd*-trees can be used to organize the data search. Unfortunately, in the worst-case even such structures fail and we then have to examine the complete data before being able to find the nearest neighbor for a data point.

A *kd*-tree is built by successively partitioning the data set (of dimension d) across the different dimensions in sequence. Each partition is done by projecting the data set onto the chosen dimension (x or y in Fig. 5), picking the median of the projections, and using its value to define a dichotomy of the data space at a node of a binary tree. The node has two children (one for each half-space defined by the cut) and the partitioning continues with each partitioned data set in a recursive manner, until all points have been assigned to a parent node or to a leaf.

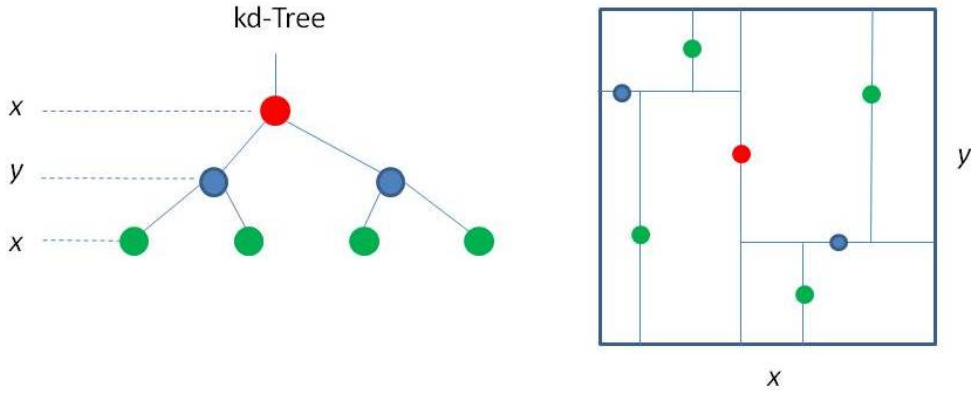

Figure 5: Building a kd-tree. Each node in the tree represents a point in the data set. A point divides space along the dimension x or y, alternated.

The partitioning in each branch is stopped when the data set for a new region is empty. Once we have the data organized in a binary tree, we can use depth-first search for inspecting the complete tree, looking for the nearest neighbor. Remember, a data point is sitting at each node in the binary tree. Fig. 6 shows (in red) a depth-first traversal of the binary tree of nodes. The convention has been used in the Figure of making the left branch of each node cover the half-space where the new point is located, whereas the right branch covers the opposite half-space. A depth-first traverasl going through the complete data set will find the nearest neighbor to a point P, for sure.

However, we can do better, by keeping a record of the current "nearest point" found and its distance to he new data point $\vec{x}$ being classified. When doing the depth-first search, we can *prune* a right branch of the tree whenever the sphere with center at the new data point $\vec{x}$ and radius r does not intersect the half-space

represented by the branch being pruned. There is no need to inspect the region across the cut, since no neighbor at a distance smaller than r can be found there.

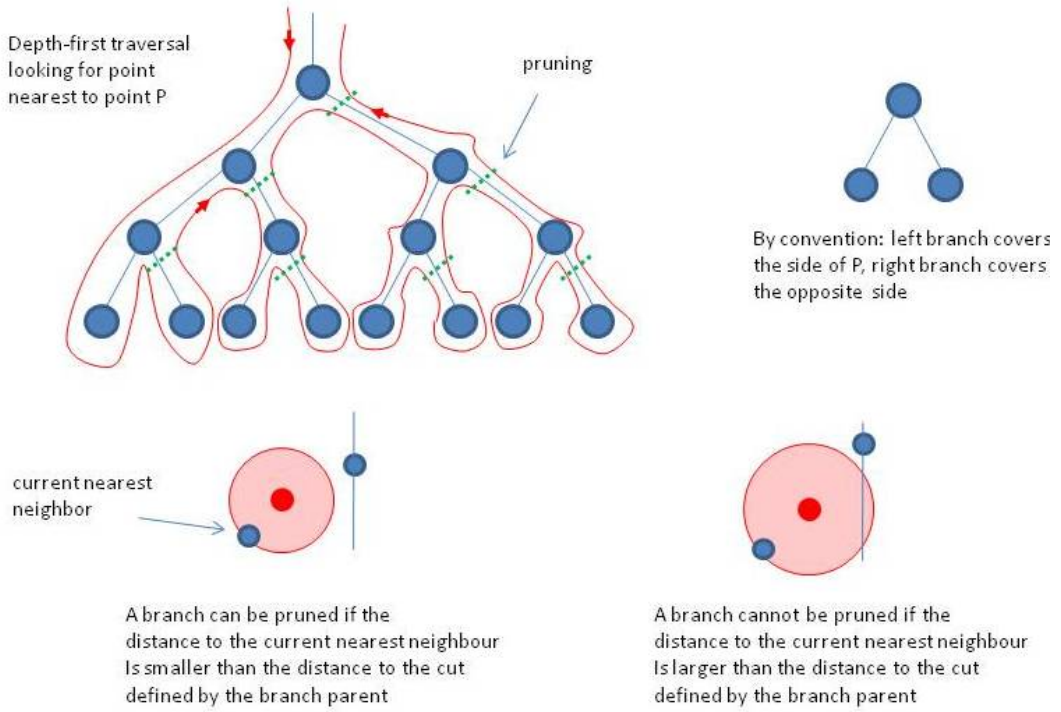

Figure 6: The depth-first traversal can be optimized by pruning branches defining regions too far away from the point whose nearest neighbor is being searched.

What we should expect, therefore, is that pruning can reduce significantly the number of nodes visited. In the best case we obtain the nearest neighbor in $\log_2(N)$ steps. In the worst case, we cruise through the complete tree without pruning any branches.

Hashing

Another alternative for a k-NN classifier is o use hash functions to preorder the data points in "buckets" that can be inspected saving effort at classification time.

Assume, for example, that we have a classification problem in three dimensions. We can project the data onto ten buckets for each dimensions.

We have, after projecting across three dimensions, 10^3 buckets, (that is, subdivisions of the data set). When a new point arrives for classification, we compute on which bucket it will project, and compare them only with the points in this bucket. This would save a factor 10^3 in computations.

Hashing is useful whenever we can find a set of "good" hashing functions, and when we can reduce the dimensionality of the data. If we could map ten-dimensional data points straight into n buckets, we would save a good deal of work.

The hashing-method is, however, not exact. Only probabilistic bounds can be assigned to the result (since the new point in its bucket can be dangerously close to the boundary of the bucket) but has proven to be useful in many real-world applications.