Learning from Data
Nearest Neighbour Classification

Course lecturer: Amos Storkey
a.storkey@ed.ac.uk
Course page : http://www.anc.ed.ac.uk/∼amos/lfd/

In classification, we have a training set of data which contains both attributes $\mathbf{x}$ and a class label c. For example, the vector $\mathbf{x}$ might represent an image of a digit, and c labels which digit it is, $c \in \{0, 1, \ldots, 9\}$. A dataset D of P training datapoints is given by $D = \{\mathbf{x}^\mu, c^\mu\}$, $\mu = 1, \ldots, P$. The aim is, given a novel $\mathbf{x}$, to return the "correct" class $c(\mathbf{x})$. A simple strategy we adopt in this chapter can be very loosely stated as:

In other words, 'just say whatever your neighbour says!'

Things $\mathbf{x}$ *which are similar (in* $\mathbf{x}$*-space) should have the same class label*

(This is a kind of smoothness assmuption. Note that in this chapter, we won't explicitly construct a 'model' of the data in the sense that we could generate fake representative data with the model. It is possible, however, to come up with a model based on the above neighbourhood type idea which does just this. We will see how to do this when we learn about density estimation in a later chapter.)

What does 'similar' mean?

The key word in the above strategy is 'similar'. Given two vectors $\mathbf{x}$ and $\mathbf{y}$ representing two different datapoints, how can we measure similarity? Clearly, this would seem to be rather subjective – two datapoints that one person thinks are 'similar' may be to someone else dissimilar.

The dissimilarity function $d(\mathbf{x}, \mathbf{y})$

Usually we define a function $d(\mathbf{x}, \mathbf{y})$, symmetric in its arguments ($d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$) that measures the dissimilarity between the datapoints $\mathbf{x}$ and $\mathbf{y}$.

It is common practice to adopt a simple measure of dissimilarity based on the *squared euclidean distance* $d(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y})^T (\mathbf{x} - \mathbf{y})$ (often more conveniently written $(\mathbf{x} - \mathbf{y})^2$) between the vector representations of the datapoints. There can be problems with this but, in general, it's not an unreasonable assumption. However, one should bear in mind that more general dissimilarity measures can, and often are used in practice.

1 Nearest Neighbour

To classify a new vector $\mathbf{x}$, given a set of training data $(\mathbf{x}^\mu, c^\mu), \mu = 1, \ldots, P$:

1. Calculate the dissimilarity of the test point $\mathbf{x}$ to each of the stored points, $d^\mu = d(\mathbf{x}, \mathbf{x}^\mu)$.
2. Find the training point $\mathbf{x}^{\mu^*}$ which is 'closest' to $\mathbf{x}$ by finding that μ^* such that $d^{\mu^*} < d^\mu$ for all $\mu = 1, \ldots, P$.
3. Assign the class label $c(\mathbf{x}) = c^{\mu^*}$.

In the case that that there are two or more 'equidistant' (or equi-dissimilar) points with different class labels, the most numerous class is chosen. If there is no one single most numerous class, we can use the K-nearest-neighbours case described in the next section.

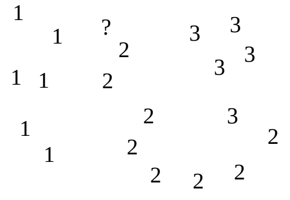

Figure 1: In nearest neighbour classification, a new vector with an unknown label, ?, is assigned the label of the vector in the training set which is nearest. In this case, the vector will be classified as a 2.

```
function y = nearest_neighbour(xtrain, xtest, t)
% calculate the nearest (single) neighbour classification
% (uses the squared distance to measure dissimilarity)

ntrain = size(xtrain,2); % number of training points
ntest = size(xtest,2); % number of test points

% Compute squared distances between vectors from the training and test sets

% This is the obvious (but very slow way) to calculate distances :
for i = 1:ntrain
  for j = 1:ntest
    sqdist(i,j) = sum((xtrain(:,i)-xtest(:,j)).^2);
  end
end

% This is the super fast way (in MATLAB) to do this :
% sqdist = repmat(sum(xtrain'.^2,2),1,ntest)+ ...
%     repmat(sum(xtest'.^2,2)',ntrain,1)-2*xtrain'*xtest;


[vals, kindex] = min(sqdist); y = t(kindex);
```

The following is a small demo that uses the above nearest-neighbour function.

```
% Nearest Neighbour demo : 3 classes

xtrain(:,1:10) = randn(2,10); % 10 two-dimesional training points
label(1,1:10)=1; % class one
xtrain(:,11:20) = randn(2,10)+repmat([2 2.5]',1,10); % 10 two-dimesional training points
label(1,11:20)=2; % class two
xtrain(:,21:30) = randn(2,10)-repmat([2 2.5]',1,10); % 10 two-dimesional training points
label(1,21:30)=3; % class three

% plot the training data :
plot(xtrain(1,1:10),xtrain(2,1:10),'r.');text(xtrain(1,1:10),xtrain(2,1:10),'1');
hold on
plot(xtrain(1,11:20),xtrain(2,11:20),'b.');text(xtrain(1,11:20),xtrain(2,11:20),'2')
plot(xtrain(1,21:30),xtrain(2,21:30),'g.');text(xtrain(1,21:30),xtrain(2,21:30),'3')

% now find the class for a bunch of novel points xquery
xquery = 2.5*randn(2,15) + repmat([1.5 0.5]',1,15);

nn_label = nearest_neighbour(xtrain, xquery, label)
plot(xquery(1,:),xquery(2,:),'k.')

nn_label1=find(nn_label==1);nn_label2=find(nn_label==2);nn_label3=find(nn_label==3);
text(xquery(1,nn_label1),xquery(2,nn_label1),'1');
text(xquery(1,nn_label2),xquery(2,nn_label2),'2');
text(xquery(1,nn_label3),xquery(2,nn_label3),'3'); hold off
```

The decision boundary

Voronoi Tessellation

In general, the decision boundary is the boundary in input space such that our decision as to the class of the input changes as we cross this boundary. In the nearest neighbour algorithm above based on the squared euclidean distance, the decision boundary is determined by the lines which are the perpendicular bisectors of the closet training points with different training labels, see fig(2). This is called a Voronoi tessellation. The decision boundary for the data from fig(1) is shown in fig(3).

1.1 Problems with Nearest Neighbours

The nearest neighbours algorithm is extremely simple yet rather powerful, and used in many applications. There are, however, some potential drawbacks:

How should we measure the distance between points? Typically one uses the

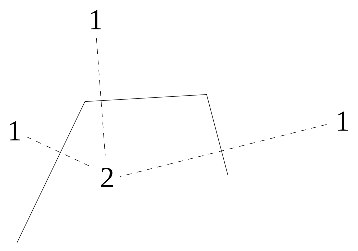

Figure 2: The decision boundary for the nearest neighbour classification rule is piecewise linear with each segment corresponding to the perpendicular bisector between two datapoints belonging to different classes.

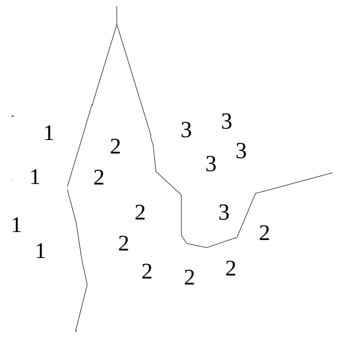

Figure 3: Decision boundary for the nearest neighbour classification rule.

euclidean square distance, as given in the algorithm above. This may not always be appropriate. Consider a situation such as in fig(4), in which the euclidean distance leads to an undesirable result. If we use the Eucliean distance, $(\mathbf{x} - \mathbf{y})^T(\mathbf{x} - \mathbf{y})$ then the distance between the orthogonally transformed vectors $M\mathbf{x}$ and $M\mathbf{y}$ (where M^TM is the identity matrix) remains the same. (This is not true for the Mahalanobis distance). Since classification will be invariant to such transformations, this shows that we do not make a sensible model of how the data is generated – this is solved by density estimation methods – see later chapter.

Invariance to linear transformation

Mahalanobis Distance

The Mahalanobis distance $(\mathbf{x} - \mathbf{y})^T A^i(\mathbf{x} - \mathbf{y})$ where usually A^i is the inverse covariance matrix of the data from class i can overcome some of these problems. I think it's better to use density estimation methods.

Data Editing

In the simple version of the algorithm as explained above, we need to store the whole dataset in order to make a classification. However, it is clear that, in general, only a subset of the training data will actually determine the decision boundary. This can be addressed by a method called data editing in which datapoints which do not affect (or only very slightly) the decision boundary are removed from the training dataset.

Dimension Reduction

Each distance calculation could be quite expensive if the datapoints are high dimensional. Principal Components Analysis (see chapter on linear dimension reduction) is one way to address this, by first replacing each high dimensional datapoing $\mathbf{x}^\mu$ with it's low dimensional PCA components vector $\mathbf{p}^\mu$. The euclidean distance of the of two datapoints $\left(\mathbf{x}^a - \mathbf{x}^b\right)^2$ is then approximately given by $\left(\mathbf{p}^a - \mathbf{p}^b\right)^2$ – thus we need only to calculate distance among the PCA representations of data. This can often also improve the classification accuracy.

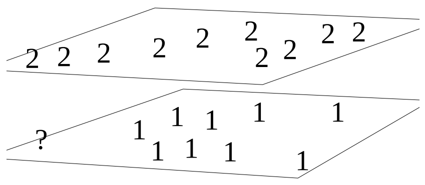

Figure 4: Consider data which lie close to (hyper)planes. The euclidean distance would classify ? as belonging to class 2 – an undesirable effect.

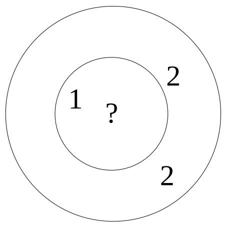

Figure 5: In K-nearest neighbours, we centre a hypersphere around the point we wish to classify. The first circle corresponds to the nearest neighbour method, and we would therefore class ? as class 1. However, using the 3 nearest neighbours, we find that there are two 2's and one 1 – and we would therefore class ? as a 2.

Sensitivity to outliers

An outlier is a 'rogue' datapoint which has a strange label – this maybe the result of errors in the database. If every other point that is close to this rogue point has a consistently different label, we wouldn't want a new test point to take the label of the rogue point. K nearest neighbours is a way to more robustly classify datapoints by looking at more than just the nearest neighbour.

2 K Nearest Neighbours

As the name suggests, the idea here is to include more than one neighbour in the decision about the class of a novel point $\mathbf{x}$. I will here assume that we are using the Euclidean distance as the simmilarity measure – the generalisation to other dissimilarity measures is obvious. This is achieved by considering a hypersphere centred on the point $\mathbf{x}$ with radius r. We increase the radius r until the hypersphere contains exactly K points. The class label $c(\mathbf{x})$ is then given by the most numerous class within the hypersphere. This method is useful since classifications will be robust against "outliers" – datapoints which are somewhat anomalous compared with other datapoints from the same class. The influence of such outliers would be outvoted.

How do we choose K?

Clearly if K becomes very large, then the classifications will become all the same – simply classify each $\mathbf{x}$ as the most numerous class. We can argue therefore that there is some sense in making $K > 1$, but certainly little sense in making $K = P$ (P is the number of training points). This suggests that there is some "optimal" intermediate setting of K. By optimal we mean that setting of K which gives the best generalisation performance. One way to do this is to leave aside some data that can be used to test the performance of a setting of K, such that the predicted class labels and the correct class labels can be compared. How we define this is the topic of a later chapter.

Generalisation

3 Handwritten digit Example

We will apply the nearest neighbour technique to classify handwritten digits. In our first experiement, we will first look at a scenario in which there are only two digit types, zeros, and ones. There are 300 training examples of zeros, and 300 training examples of ones, fig(6). We will then use the nearest neighbour method to predict the label of 600 test digits, where the 600 test digits are distinct from the training data and contain 300 zeros and 300 ones (although, of course, the test label is unknown until we assess the performance of our predictions). The nearest neighbour method, applied to this data, predicts correctly the class label of all 600 test points. The reason for the high success rate is that examples of zeros and ones are sufficiently different that they can be easily distinguished using such a simple distance measure.

In a second experiment, a more difficult task is to distinguish between ones and sevens. We repeated the above experiment, now using 300 training examples of ones, and 300 training examples of sevens, fig(7). Again, 600 new test examples (containing 300 ones and 300 sevens) were used to assess the performance. This time, 18 errors are found using nearest neighbour classification – a 3% error rate for this two class problem. The 18 test points on which the nearest neighbour method makes errors are plotted in fig(8). Certainly this is a more difficult task than distinguishing between zeros and ones. If we use $K = 3$ nearest neighbours, the classification error reduces to 14 – a slight improvement. Real world handwritten digit classification is big business. The best methods classify real world digits (over all 10 classes) to an error of less than 1% – better than human performance.

State of the art

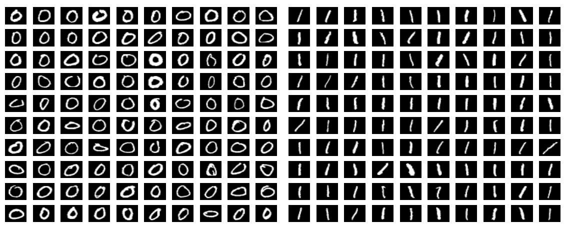

Figure 6: (left) Some of the 300 training examples of the digit zero and (right) some of the 300 training examples of the digit one.

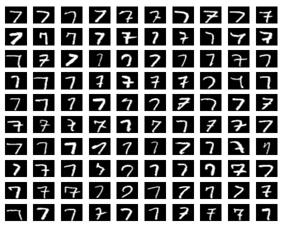

Figure 7: Some of the 300 training examples of the digit seven.

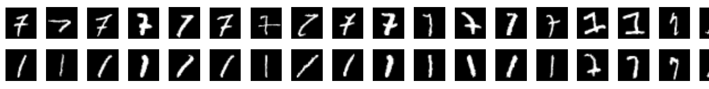

Figure 8: The Nearest Neighbour method makes 18 errors out of the 600 test examples. The 18 test examples that are incorrectly classified are plotted (above), along with their nearest neightbour in the training set (below).