Comparison of Classifiers

We've looked at a number of ways of building classifiers including naïve Bayes, kNN, rules and neural networks. We also considered the proper way to measure their accuracy. In this lecture, we do two things: first we make some more remarks about neural networks; second we take a more rounded look at how to compare different classifiers.

1 Neural Networks

1.1 Speed of learning

The time taken by the learning step for a neural network can be very variable in practice and, in the worst case, is not good at all. If there are n_e examples and n_w weights, each epoch takes $O(n_e n_w)$ time, and, in the worst case, the number of epochs can be exponential in n, the number of inputs.

For example, we saw in the previous lecture that a large number of epochs were needed to learn exclusive-or. This was a small network, and we were presenting it with all possible examples!

There are various techniques to try to speed up learning in practice. One is to use a learning rate, α, that is closer to 1 (but see below). Another technique is to introduce another term into the weight update rules called a *momentum factor*. We won't go into details, but in the early epochs, this factor is close to zero, and then, in the remaining epochs, it is set to, e.g., 0.9. This gives the algorithm time to find a good general direction (making small changes) in the early epochs, and then increases the learning speed once that direction has been found.

Of course, another idea is to abandon sequential software simulations and use parallel hardware.

1.2 Convergence

Will the network learn the function that we are training it to learn? Quite apart from how long the process will take, there is the question of whether we will ever reach a situation of zero error (or near zero error).

As previously discussed for TLU learning, if the value chosen for the learning rate is too large, there is a danger of 'overshooting', and even oscillating around, the optimal weights. But there is another danger, which applies to networks but not to single TLUs. The error surface may have *local minima*. These are places where no small change in the weights makes the error any smaller, but the network has not yet reached the global minimum error.

Opinion differs on the significance of this problem; people disagree about whether it is common enough in real domains to be a worry. In domains where you do think that it is a problem, there are techniques for overcoming it. One approach is to allow the learning algorithm to make some weight changes that actually increase the error (see *simulated annealing*).

1.3 The role of the network architecture

As noted earlier, we want generalisation: accurate classifications on unseen data. Will we get it?

The network architecture or topology plays a major role here. How many hidden layers should there be? And in each of those hidden layers, how many hidden units should there be? The wrong choices can lead to poor generalisation.

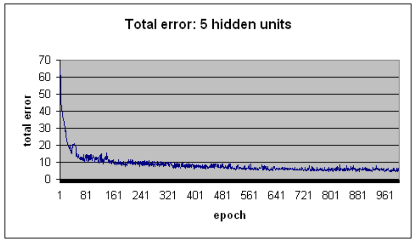
If the network is too small, it will not be able to represent the function that it needs to represent: it won't be able to draw the distinctions it needs for making good predictions even on the seen data.

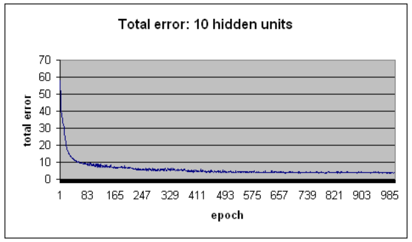
But, if the network is too big, it will have room to simply memorise the examples and act like a large look-up table.

To illustrate this a little, I trained a neural network on a dataset of 150 plants from the UCI Machine Learning Repository (whose URL is on the module web site). All of them are irises of some kind. There are four attributes (all numeric-valued): sepal length in cm; sepal width in cm; petal length in cm; and petal width in cm. There are no missing values. There are three classes: a plant can be of class Iris Setosa (class label $\langle 1, 0, 0 \rangle$), Iris Versicolour (class label $\langle 0, 1, 0 \rangle$) or Iris Virginica (class label $\langle 0, 0, 1 \rangle$). In the data, there are 50 examples of each of the three classes. The data is not noisy.

I had to decide on a network topology. Obviously, I need 4 input units (excluding the 'extra' threshold unit) and three output units. My Java implementation of back-prop only allows one hidden layer, so, with my software I've no real choice about how many hidden layers to use! But I still need to decide how many hidden units there will be in that layer. I decided to try out a number of different cases: I tried out 5 and 10 (excluding the 'extra' one).

Here are graphs of average error (using 5 iterations of repeated holdout) for different numbers of epochs and different numbers of hidden units:





How might we come up with a good network architecture? Some people claim to be good at deciding architectures on the basis of the input and output spaces. But even then, they follow their 'informed' guesswork by a period of experimentation with different architectures. The alternative is to seek to do this automatically. One approach is random generation of architectures — but there are too many possibilities to make this a good approach. Another approach is to try to evolve the architecture using a genetic algorithm (see future lecture). This sounds promising but it is so computationally expensive that it is rarely feasible. More common is to use a *hill-climbing* approach. Again we won't cover the details but basically you start with a single random architecture and make it successively smaller (or bigger) using information-theoretic measures to decide which weights (and hence hidden units) to retain.

Now we'll turn to the broader issue of how to compare classifiers. The remaining sections of these notes compare classifiers on a variety of different criteria.

2 Accuracy

Accuracy is the obvious way to compare classifiers. We have previously discussed how to measure it. Given a dataset of already-classified instances, you would use a suitable method (holdout, repeated holdout, k-fold cross-validation, repeated k-fold cross-validation or LOOCV) to measure the accuracy of each classifier. It is important that you use the same training and tests for each classifier. For example, if you are using the holdout method and you have partitioned the dataset into 66% training instances and 34% test instances, then each classifier that you are comparing should be trained on the same training set and tested on the same test set. It is also important that you don't cheat when setting parameters. This has to be done using a validation set — see the earlier lecture.

When comparing different classifiers, it can be useful to include a simple-minded classifier, to act as a benchmark. One simple benchmark is the classifier that always returns the majority class in its training set. So, for example, if the training set contains 75% spam and 25% ham, this classifier will always return *class* $=$ *spam*. It will be correct approximately 75% of the time! So if your more sophisticated classifiers (naïve Bayes or kNN or some new algorithm you've invented) can't do better than 75% accuracy then, for all their sophistication, they aren't learning predictive relationships from the data.

Once you have obtained accuracy figures for each classifier, you'll be able to see whether they outperform whichever classifiers you included as benchmarks, and you'll be able to see which of the classifiers is best (highest accuracy). At this point, you consult a statistics book or a statistician because you need to do a *significance test*. This will tell you how confident you can be that there really is a difference.

3 Task Characteristics

Different classifiers may be better suited to different tasks. For example, some might be better able to handle numeric attributes than others; some might be best-suited to tasks where there are only two classes; etc.

3.1 Input data types

Are some of the classifiers better-suited to numeric inputs? Boolean inputs? Symbolic inputs? Mixtures? I leave you to think about this for yourselves.

3.2 Number of classes

Are some of the classifiers better-suited to problems in which there are only two classes? How well do they handle more than two classes? I leave you to think about this for yourselves.

3.3 Missing values

It is not uncommon that instances in either the training set or the test set have missing values for certain of the attributes. For example, a bank might describe people who apply for loans using just two attributes: salary and age. A training instance might be missing the age, for example: $\{salary = 20000, class = noLoan\}$. A test instance might be missing the salary, for example: $\{age = 25\}$.

Making this more difficult is that missing values can have different interpretations:

Value unknown: we know the person has a salary but we don't know what it is;

Value inapplicable: we know the person does *not* have a salary; or

Status unknown: we do not know whether the person has a salary or not.

People building classifiers have not been very good at making these distinctions!

Can a classifier's learning step cope with missing values? Can a classifier's classification step cope with missing values? Suppose instances have three attributes, A, B and C, but some are missing a value for A.

Naïve Bayes: This copes well with missing values. There are several approaches, some better-motivated than others. We mention just a couple.

Learning: When computing $P(A = a|class = c)$, you simply ignore instances in the training set which lack a value for attribute A.

Classifying: As you know, for an instance with no missing values, we would want to compute $P(class = cl|A = a, B = b, C = c)$ for each class $cl \in L$. We would uses Bayes' rule and we would assume conditional independence to rewrite this as $P(A = a|class = cl) \times P(B = b|class = cl) \times P(C = c|class = cl) \times P(Cclass = cl)$ (ignoring the divisor as usual). If the query is missing a value for attributes A we can either

- simply ignore this attribute, i.e. compute $P(B = b|class = cl) \times P(C = c|class = cl) \times P(Cclass = cl)$; or
- average the probabilities for each of the possible values for this attribute. (Again there are more advanced variants in the case of continuous-valued attributes which have not been discretised.)

The second option is particularly well-suited to missing values that mean 'value inapplicable'.

kNN:

Learning: Nothing special needs to be done: just store the dataset.

Classifying: The local distance functions need to be redefined so that they can cope with missing values. A common approach is simply for the local distance function dist_A to return the maximum distance (typically 1) if either x or q is missing its value for attribute A. This is somewhat crude, and may not reflect the different kinds of missing values described above.

Rules:

Learning: Since we did not study any rule learning algorithms in this module, we can't really consider how well they cope with missing values.

Classifying: The forwards- and backwards-chaining algorithms do not need any modification. However, if a rule mentions attribute A and the instance to be classified has no value for A, then that rule will never succeed. This might result in the instance being misclassified or failing to be classified at all.

Neural networks: Neural networks cannot handle instances with missing values: some value must be placed onto each input unit. There are three possibilities — see the next paragraph.

There are, of course, three possibilities that work with all the classifiers: (a) remove from your dataset any instances that have missing values; (b) replace each missing values with a dummy value (if one is available) with the risk that this will make learning/classifying harder (e.g. it may turn a linearly separable function into one that is not linearly separable); and (c) replace missing values with 'made up' values, e.g. in the case of a numeric attribute, the mean or mode could be used.

3.4 Noise

We'll use the word 'noise' in a relatively narrow sense. A noisy training set is one in which instances may have incorrect values for the attributes or class. For example, the drinking dataset is very noisy: people misreported their weight, the duration of their drinking, and the number of units of alcohol consumed. If the breathalyser equipment was at all faulty or used incorrectly, then their class (under or over) might be incorrect too. Obviously, we cannot hope to learn a perfect classifier from a noisy training set. The question is: how sensitive are different classifiers to noise?

This question is best answered by running experiments and measuring accuracy. (We could even introduce random errors into the training set and see how quickly accuracy falls off.)

In general, rule-based systems (at least, of the kind we've looked at so far) tend to be very brittle in the face of noise, whereas naïve Bayes, kNN and neural networks tend to be more robust. The accuracy of naïve Bayes is unlikely to be badly damaged by small inaccuracies in the probabilities (particularly since we're only interested in seeing which is the largest); kNN can generally be made more robust by increasing k and, sometimes, by reducing the weight of local distances computed on unreliable attributes; the accuracy of a neural network is unlikely to be badly damaged by small inaccuracies in the weights.

4 Efficiency

When comparing different classifiers for efficiency, we need to consider four factors: their time- and space-efficiency for the learning step, and their time- and space-efficiency for the classifying step.

I leave you to think about this for yourselves.

5 Transparency

I am using the word 'transparency' to bundle together a number of issues concerning the degree to which human users will find different classifiers to be intelligible. There are two points where we might wish for 'transparency'.

Learning step. In the case of eager learners, after the learning step we might hope that the system can present an intelligible version of what it has learned.

Naïve Bayes classifiers can display their probabilities, and there are some nice graphical ways of presenting these to allow easy visualisation. For kNN, people have devised graphical ways of visualising the contents of the system. Rule-based systems can display their rules, and these may be intelligible to human experts. There is not much that a neural network can do: displaying their weights will help no one. There has been research into trying to extract rules from networks, so that these can be displayed. But, in general, we have to conclude that neural networks are a 'black-box'.

Classification step. A classifier should be able to *explain* why an instance has been classified the way that it has, and it should be able to report is *confidence* in its classification. Let's look at these in turn.

Explanations are particularly important in safety-critical domains (especially where life is endangered) or in domains where litigation risks and costs are high. A naïve Bayes classifier can present rudimentary explanations by presenting the probabilities that it computed. A rule-based system can display a trace of the rules that fired. This might be adequate if the rules are reasonably intelligible to begin with. Neural networks can offer nothing.

kNN's explanations have been found to be especially good: the classifications are based directly on already-classified instances (the neighbours) and these instances can be displayed to provide compelling explanations for the classification. Recent research (some of it done here in my UCC research group) is further improving kNN's explanations.

Users may also want to know how confident a classifier is that it is making the right classification. A naïve Bayes classifier can compare the probabilities it computes for each class: the more different the probabilities, the more confident it can be. A rule-based classifier has no way of quantifying its confidence, unless the rules themselves have probabilities associated with them. kNN classifiers can quantify their confidence based on: how similar the neighbours are to q and the degree to which the neighbours agree or disagree. Neural networks can do little to quantify their confidence.

6 Incorporation of prior knowledge

Classifiers differ in the ease with which they can incorporate prior knowledge. By prior knowledge, we simply mean knowledge that human experts already possess about the task. We don't want such knowledge to go to waste. Can it be used to initialise the system, either just prior to the learning step or even in place of the learning step?

In the case of a naïve Bayes classifier, for example, human experts might already have some sense of the probabilities needed. If so, the classifier could be built using these human-supplied probabilities or we could work out some way of combining the human-supplied probabilities with ones computed from a dataset.

Experts might be able to supply some or all of the rules needed by a rule-based classifier. The learning step, if takes place at all, would only need to learn additional rules to cover instances not already covered by the human-supplied rule.

When building a kNN classifier, human experts can use their knowledge to design knowledge-intensive local distance functions and also, if the global distance is to be a weighted sum or weighted average of the local distances, experts might be able to choose the weights (attributes importances). Human experts might also exercise judgement over which already-classified instances they would like to include in the kNN classifier's dataset.

There is little that human experts can do to encode their prior knowledge into neural networks. One line of research did take human-supplied rules and use these to influence the network architecture/topology and the initial weights (see the exercise at the end of these notes).

7 Incremental learning

Up to now, we have separated our presentation of classifiers into a learning step and a classification step. But suppose new already-classified examples become available after the system has finished its learning step and is now being used for classification. Indeed, often these new examples will arise because the user of the classifier will confirm or disconfirm whether the system got the classification right. For example, in a spam filter the user might be willing to give feedback in the hope of improved classification accuracy.

The ability to continue to learn is important: it can make a system adapt to changing circumstances. Consider spam filtering for example: $P(class = spam)$ is very different today from what it was a few years ago. So the question is: can we build versions of these classifiers that take on new training examples incrementally, or do they only operate in 'batch' mode?

Naïve Bayes classifiers can update probabilities in an incremental fashion. kNN classifiers can store new examples in their dataset as they arise. Of course, there is a risk that the dataset grows too large, causing a degradation in classification efficiency. However, there is research into ways of editing the dataset to reduce its size, while preserving its classification accuracy.

Rule-based systems and neural networks cannot, in general, take on new examples incrementally. The best we can do in general is add the new examples to the original training set and retrain the system from scratch on the enlarged set of instances.

Exercise (Part of a past exam question)

This question is about TLUs, neural networks and rule-based classifiers. Throughout, assume the following (which differ from what we used in lectures):

- the *activation function* of the TLUs, g, is defined as follows:

$$g(x) =_{\text{def}} \begin{cases} 1 & \text{if } x \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

  where θ is the threshold of the TLU; and
- -1 will denote false and 1 will denote true.

1. Design a TLU which has two inputs s_1 and s_2. The inputs can take values of -1 or 1. The TLU should compute the conjunction of s_1 and s_2, $s_1 \wedge s_2$. Give the two weights and the threshold.

2. Suppose you wanted your TLU to compute the conjunction of an arbitrary number n of inputs, $n \geq 1$. What weights and threshold would you use?

3. Similarly, give the weights and threshold for a TLU that computes the disjunction of two inputs, $s_1 \vee s_2$.

4. Similarly, give the weights and threshold for a TLU that computes the disjunction of n inputs, $n \geq 1$.

5. *Hence*, continuing to assume that all inputs s_i take on values of only -1 or 1, design a fully connected, layered, feedforward neural network that outputs a 1 in exactly the same circumstances that the following rules would conclude that p is true:

$$\boxed{\begin{array}{l}(s_1 \wedge s_3) \Rightarrow p \\ (s_1 \wedge s_2 \wedge s_4) \Rightarrow p\end{array}}$$

6. A knowledge engineer elicits a set of rules from a medic who specialises in diseases of the male sarcophagus. These disfiguring diseases, *tumulus*, *ossuary* and *cromlech*, are treated using either the drug *Placebin* or the drug *Incubio*:

$$\boxed{\begin{array}{l}(\textit{badBreath} \wedge \textit{dizziness}) \Rightarrow \textit{tumulus} \\ (\textit{badBreath} \wedge \textit{hairLoss} \wedge \textit{toothache}) \Rightarrow \textit{tumulus} \\ (\textit{dizziness} \wedge \textit{runnyNose}) \Rightarrow \textit{ossuary} \\ (\textit{hairLoss} \wedge \textit{runnyNose}) \Rightarrow \textit{ossuary} \\ \textit{redEars} \Rightarrow \textit{cromlech} \\ \textit{tumulus} \Rightarrow \textit{Placebin} \\ \textit{ossuary} \Rightarrow \textit{Incubio} \\ \textit{cromlech} \Rightarrow \textit{Placebin}\end{array}}$$

The knowledge engineer encodes the rules as a neural network using the ideas embodied in your answer to the previous question.

She then obtains some historical data, showing, for each patient, his symptoms and which of the two drugs was found to be effective. Using the back-propagation algorithm, she trains the neural network on the historical data.

Discuss the advantages that the knowledge engineer believes will ensue from this combined rule-and-network approach.