2D1431 Machine Learning
Lab 3: Instance Based Learning & Neural Networks

Frank Hoffmann
e-mail: hoffmann@nada.kth.se

November 27, 2002

1 Introduction

In this lab you will learn about instance based learning algorithm (locally weighted regression) and artificial neural networks and apply both techniques to function approximation. You will also learn how to use cross-validation for parameter and feature selection. You will have to implement the code for locally weighted regression and cross-validation. You will use some existing code for the back-propagation learning algorithm. During the examination with the lab assistant, you will present the results, answers to the questions and the code that you wrote.
It is assumed that you are familiar with the basic concepts of instance based learning, artificial neural networks and cross-validation and that you have read chapters 4 and 8 in the course book *Machine Learning* [4]. For further reading on nearest neighbor algorithms for function approximation I recommend the article by ATKESON et al on locally weighted learning [1]. For an introduction on cross-validation and bootstrapping for accuracy estimation read the article by KOHAVI [3]. The paper by JOHN et al discusses the feature subset selection problem [2].
In the first part of the exercise we will use a synthetic function, the so-called Mexican hat function

$$f(x_1, x_2) = \frac{sin(x_1)sin(x_2)}{x_1 x_2} \tag{1}$$

A Matlab file `mexhat.m` to compute the Mexican hat function is provided. You can visualize the function with the following Matlab commands.

```
>> x1=-5.01:0.25:5.01;
>> x2=-5.01:0.25:5.01;
>> [x, y]=meshgrid(x1,x2);
>> f=mexhat(x,y);
>> mesh(x1,x2,f);
```

The training data and test data sets with 100 instances each are stored in the files `f_train.dat` and `f_test.dat`, with one instance $\{x_1, x_2, f(x_1, x_2)\}$ per line. The first two labs dealt with pattern recognition or classification problems, in which the objective is to assign instances to a discrete number of classes. In function approximation, the goal is to approximate an unknown continuous function such that the mapping realized by the learning algorithm is close enough to the true function. In the context of function approximation, the goal is to minimize the quadratic error between between the desired $f(x)$ and the actual output $\hat{f}(x)$ of the learner over the distribution D of training instances.

$$E = \sum_{x \in D} (f(x) - \hat{f}(x))^2 \tag{2}$$

2 Instance Based Learning

In contrast to learning algorithms that explicitly generate a description of the target function, instance based learners simply store the training examples. The output of a new instance is computed based on the relationship between the new instance and the stored examples. In general, the more similar the new instance is to a stored example, the more the target value of that example contributes to the predicted output of the unseen instance. In this exercise you will implement a distance-weighted nearest neighbor algorithm for function approximation. The contribution of the i-th example $\vec{x}^i, f(x^i)$ in the training set, depends on its distance $d(\vec{x}^i, \vec{x}^q)$ to the query point $\vec{x}^q$. A typical distance function is the Euclidean distance in which x_k is the k-th component of the vector $\vec{x}$.

$$d(\vec{x}^i, \vec{x}^q) = \sqrt{\sum_{k=1}^{n} (x_k^i - x_k^q)^2} \tag{3}$$

The kernel function $K(d(\vec{x}^i, \vec{x}^q))$ is a function of the distance that is used to determine the weight w^i of each training example.

$$w^i = K(d(\vec{x}^i, \vec{x}^q)) \tag{4}$$

There are a large number of possible kernel functions, at first we consider a Gaussian kernel

$$K(d) = e^{-\frac{d^2}{\sigma^2}} \tag{5}$$

where the parameter σ determines the width of the Gaussian.
The simplest form of locally weighted regression is locally weighted averaging in which the local model is simply a constant, namely the output $f(\vec{x}^i)$ associated to the training example $\vec{x}^i$. Weighting the data can be viewed as replicating the output of relevant (similar) instances and discarding irrelevant (dissimilar) instances. The weighted average estimate $\hat{f}(\vec{x}^q)$ of a query point $\vec{x}^q$ is computed as:

$$\hat{f}(\vec{x}^q) = \frac{\sum_{i=1}^{N} w^i f(\vec{x}^i)}{\sum_{i=1}^{N} w^i} = \frac{\sum_{i=1}^{N} K(d(\vec{x}^i, \vec{x}^q)) f(\vec{x}^i)}{\sum_{i=1}^{N} K(d(\vec{x}^i, \vec{x}^q))} \tag{6}$$

Notice, that the estimate $\hat{f}(\vec{x}^q)$ depends on the query point $\vec{x}^q$.
We are trying to find the best estimate for the output $\hat{f}(\vec{x}^q)$, using a local model that is a constant. Distance weighting the error criterion for the query point $\vec{x}^q$ corresponds to requiring the local model to fit nearby points well, with less concern for distant points.

$$E(\hat{f}(\vec{x}^q)) = \sum_{i=1}^{N} (\hat{f}(\vec{x}^q) - f(\vec{x}^i))^2 K(d(\vec{x}^i, \vec{x}^q)) \tag{7}$$

Our best estimate for $\hat{f}(\vec{x}^q)$ will minimize the error $E(\hat{f}(\vec{x}^q))$. Equation 7 becomes minimal for

$$\frac{\partial E(\vec{x}^q)}{\partial \hat{f}(\vec{x}^q)} = 0 \tag{8}$$

which is achieved by $\hat{f}(\vec{x}^q)$ in equation 6, in other words in this case weighting the error criterion and weighting the data is equivalent.

Assignment 1:

Write a Matlab function `nn(data,xq,sigma)` that computes the locally weighted average estimate $\hat{f}(\vec{x}^q)$ for a query point $\vec{x}^q$). Assume the usual data format for the parameter `data`, namely that the first $M-1$ columns contain the features $x_1, ...x_{M-1}$, and the last column contains the target value $f(\vec{x})$. Each row in `data` contains one example input-target pair $\vec{x}^i, f(\vec{x^i})$. The parameter `xq` is vector of length $M-1$ containing the query point $\vec{x}^q$).

The parameter `sigma` is a scalar that determines the width of the Gaussian kernel.
Load the training and test datasets into Matlab, and compute the root mean square error of the locally weighted averaging estimator over the test data for a parameter value of $\sigma = 1.0$.

```
>> traindata=load('f_train.dat');
>> testdata=load('f_test.dat');
>> [n,m]=size(traindata);
>> [k,l]=size(testdata);
>> ftest=nn(traindata,testdata(:,1:l-1),1.0);
>> error=(sum((ftest-testdata(:,l)).^2)/k)^0.5;
```

$E = \sqrt{\sum_{x \in X} (f(x) - \hat{f}(x))^2 / |X|} = \ldots$

A learning curve depicts the error as a function of the number of available training instances. Compute and display the root mean square error of locally weighted averaging for training set sizes of $10, 20, \ldots, 100$.

```
>> sz=10:10:n;
>> error=zeros(1,length(sz));
>> for i=1:length(sz)
>>  ftest=nn(traindata(1:sz(i),:),testdata(:,1:l-1),1.0);
>>  error(i)=(sum((ftest-testdata(:,l)).^2)/k)^0.5;
>> end
>> plot(sz,error,'r-');
```

Parameter selection aims to identify those parameters of a learning algorithm that result in optimal generalization. Cross-validation is used to estimate the error over future unseen instances. Typical parameters that can be optimized by means of cross-validation are the number of hidden neurons in an artificial neural network, the number of training steps or in our case the width σ of the kernel function. In K-fold cross validation the data set D is partitioned into K disjunctive subsets D_k of equal size. One of the subsets is used for validation and the remaining $K-1$ sets $D \backslash D_v$ are used for training. This process is repeated K times, such that each subset is used for validation exactly once. The overall estimate of the error is computed as

$$E(D) = \sum_{k=1}^{K} E(D_k | D \backslash D_k) = \sum_{k=1}^{K} \sum_{x^i \in D_k} (f(x^i) - \hat{f}(x^i))^2 \qquad (9)$$

where $E(D_k|D\backslash D_k)$ denotes the sum of squared errors over instances $x^i \in D_k$ when the learning algorithm is trained on the remaining instances $x^i \in D\backslash D_k$. In the extreme case for $K = |D|$ one obtains leave-one-out crossvalidation, in which as single instance x_k is used as the validation set, and the algorithm is trained on all other instances $D\backslash x_k$. Usually, leave-one-out crossvalidation is expensive, but in the case of instance based learning algorithm it will turn out to be computationally simple.

Assignment 2:

Use leave one-out cross-validation to determine the optimal value of the kernel width parameter σ for your weighted average algorithm, with respect to the Mexican hat function. Notice, that you are only supposed to use the training data `traindata` for parameter selection, not the test data `testdata`.

Leave one-out cross-validation for instance based learning is fairly simple. When computing the distance weighted average $\hat{f}_{cv}(\vec{x}^k)$ according to 6, the instance x_k itself is ignored while the average is computed over $\{D\backslash x_k\}$.

$$\hat{f}_{cv}(\vec{x}^k) = \frac{\sum_{i\neq k} w^i f(\vec{x}^i)}{\sum_{i\neq k} w^i} = \frac{\sum_{i\neq k} K(d(\vec{x}^i,\vec{x}^k))f(\vec{x}^i)}{\sum_{i\neq k} K(d(\vec{x}^i,\vec{x}^k))} \tag{10}$$

One simple way to compute $\hat{f}_{cv}(\vec{x}^k)$ in Matlab using the existing code for `nn` is to temporarily set $K(d(\vec{x}^k,\vec{x}^k)) = 0$. The total cross-validation error is computed as

$$E_{cv} = \sqrt{\sum_{x_i\in D} (f(x_i) - \hat{f}_{cv}(x_i))^2/|D|} \tag{11}$$

Write Matlab code that computes and plots the error estimated by means of leave-one-out cross-validation for parameter values $\sigma_i \in \{0.1, 0.2, \ldots, 1.0\}$. For which value of σ does the error become minimal?

$\sigma_{opt} = argmin_{\sigma_i}\, E_{cv} = \ldots$

Next compute the root mean square error on the test data, for weighted averaging with the optimal kernel width σ_{opt}.

$E_{\sigma_{opt}} = \sqrt{\sum_{x\in D}(f(x) - \hat{f}(x))^2/|D|} = \ldots$

Compare the error with the earlier result obtained by the algorithm with a default value of $\sigma = 1.0$ How large is the reduction in error achieved by using parameter selection?

You can visualize the approximation of the nearest neighbor and the error between the true function and the approximation with the following Matlab code where `sigma_opt` is the optimal value of σ calculated by means of cross-validation.

```
>> input=zeros(41^2,2);
>> input(:,1)=reshape(x,41^2,1);
>> input(:,2)=reshape(y,41^2,1);
>> fnn=nn(traindata,input,sigma_opt);
>> fnn=reshape(fnn,41,41);
>> figure(1);
>> mesh(x1,x2,fnn);
>> figure(2)
>> mesh(x1,x2,f-fnn);
```

3 Neural Networks

This part of the lab takes advantage of the freely available Netlab Neural Network software that accompanies the book "NETLAB - Algorithms for Pattern Recognition" [5]. Include the NETLAB toolbox in your Matlab search path and run the demo `demmlp1`.

```
>> addpath('/info/mi02/labs/netlab');
>> demmlp1;
```

If you are curious what pattern recognition methods the Netlab toolbox has to offer try the general demo `demnlab`. In the following you train a multi-layer perceptron on the training data generated from the Mexican hat function. The network is constructed with the function
`mlp(ninputs,nhidden,noutputs,func)`
where the first three parameters specify the number of neurons in the input, hidden and output layer, and `func` determines the type of activation function for the output neurons ('linear','logistic','softmax'). The Mexican hat function is a mapping from $\Re^2 \rightarrow \Re$, therefore, the proper network must have two input and one output neurons. For the time being consider a network with five hidden neurons and a linear activation function.

```
>> net = mlp(2,5,1,'linear');
```

Netlab uses an option vector to control parameters of the network training, such as the learning rate η, the momentum term α and the number of training cycles. By setting `options(1)=1` the learning algorithm outputs the sum of squared errors over all training instances at each cycle. For more details on the option vector type `help graddesc`.

```
>> eta=0.01;
>> alpha=0.5;
>> ncycles=100;
>> options=zeros(1,18);
>> options(1)=1;
>> options(14)=ncycles;
>> options(17)=alpha;
>> options(18)=eta;
```

The Netlab toolbox provides a number of different optimization algorithms for training the network, such as normal gradient descent, quasi Newton methods, conjugate gradients and scaled conjugate gradients. The function `netopt(net, options, x, t, alg)` is used for training the network on the input data `x` and the target values `t`. Notice that the dimensions of `x` and `t` have to match the number of input respectively output neurons in `net`. The parameter `alg` determines which optimization technique is used for training the network, gradient descent ('graddesc'), quasi Newton method ('quasinew'), conjugate gradients ('conjgrad'), scaled conjugate gradients ('scg') or Monte Carlo method ('hmc'). In case the gradient descent or any other optimization scheme diverges during training (the squared error increases substantially with each iteration) reduce the learning rate `eta` and restart with the training.
At first we train the network using the standard back-propagation algorithm based on gradient descent.

```
>> [net, options, sqerror] = netopt(net, options, ...
             traindata(:,1:l-1), traindata(:,l), 'graddesc');
>> plot(1:length(sqerror),log(sqerror),'r-');
```

The function `netopt` returns the network after training `net`, the options vector and the vector of squared errors after each iteration, of which we plot the logarithm. By means of the function `mlpfwd(net,x)` we can use the trained network `net` to predict the output value of new instances `x`, for example to compute the test set error.

```
>> y_test=mlpfwd(net, testdata(:,1:l-1));
>> error_test=sum((y_test-testdata(:,l)).^2)/k
```

where `error_test` is the mean squared error of an instance. If we compare the test error with the final training error, we might expect that we already tend to over-fit the network to the data.

```
>> y_train=mlpfwd(net, traindata(:,1:l-1));
>> error_train=sum((y_train-traindata(:,l)).^2)/n
```

Assignment 3:
To verify if the networks really over-fits the data we observe the test set and training error during training. For that purpose we compute the test and training set error every tenth iteration. We first reinitialize the weights of the network and then train it for 20 consecutive invokations of 10 cycles each, this time using the computationally more expensive but more efficient quasi Newton optimization method

```
>> net = mlp(2,5,1,'linear');
>> ncycles=10;
>> options(14)=ncycles;
>> eta=0.02;
>> options(18)=eta;
>> for i=1:20;
>>  [net, options, sqerror] = netopt(net, options, ...
               traindata(:,1:m-1), traindata(:,m), 'quasinew');
>>  y=mlpfwd(net, testdata(:,1:l-1));
>>  error_test(i)=sum((y-testdata(:,l)).^2)/k;
>>  y=mlpfwd(net, traindata(:,1:m-1));
>>  error_train(i)=sum((y-traindata(:,m)).^2)/n;
>> end
>> plot(1:20, log(error_train), 'r-', 1:20, log(error_test), 'b-');
```

Does the network over-fit the training data, and if so after how many training cycles should we stop training to avoid over-fitting? Notice, that the result might depend on the seed of the random number generator of Matlab, with different seeds the results might look very different. If you are in doubt, run the code several times with different initial seeds. The Matlab function `rand('state',n)` allows to set the initial seed.
In the following we use the scaled conjugate gradient optimization method ('scg') and train the network for 25 cycles. With the scaled conjugate gradient method it should be safe to increase the learning rate to $\eta = 0.05$, if you notice that the error diverges, set it back to a lower value.

```
>> eta=0.05;
>> ncycles=25;
>> options(14)=ncycles;
>> options(18)=eta;
```

Assignment 4:
Determine the optimal number of hidden neurons for approximating the Mexican hat function using K-fold cross-validation. Notice, that you are only supposed to use the training data `traindata` for parameter selection, not the test data `testdata`. The training data is split into K equally sized partitions. The learning algorithm is then run K times, each time using K-1 partitions as the training set and the other partition as the test set. The accuracy results from each of the K runs are then averaged to produce the estimated accuracy.
The Matlab helper function `[train,val]=fold(traindata,nfolds,n)` partitions the original data `traindata` into training (`train`) and validation (`val`) set. The parameter `nfolds` determines the number of folds, the parameter `n` which fold is used for validation. For example `fold(data,4,2)` assigns the second of four folds to the validation set, and folds $1, 3, 4$ to the training set. Train the network with h hidden units on the training set `train` and afterwards evaluate its approximation error on the validation set `val`. Repeat this procedure for all K folds, and average the errors on the K different validation sets.
What is the optimal number of hidden neurons ({2,4,6,. . . ,20}) that achieves the lowest cross-validation error using 4-fold cross-validation? If in doubt repeat your experiment, using a different seed and compute the average errors across different runs.
With only two input variables over-fitting the data is not very likely to happen. The problem of over-fitting becomes more apparent for larger dimensional inputs, in particular if some of the inputs are not even correlated with the output.
To investigate the effect of additional *useless* attributes, the original datasets are augmented by three additional features x_3, x_4, x_5. Each instance now contains five features, of which only the first two have an impact on the output, whereas the other three features can be considered as noisy background.

```
>> traindata=load('f_train_noise.dat');
>> testdata=load('f_test_noise.dat');
>> [n,m]=size(traindata);
>> [k,l]=size(testdata);
```

Assignment 5:
Repeat the experiments in assignments 3 and 4 with the noisy dataset. Make sure to set the number of inputs neurons to five, when constructing the network with `mlp`. When should one stop training to avoid over-fitting? What is the optimal number of hidden neurons?

4 Feature Subset Selection

Feature subset selection is concerned of finding a subset of features that allows a supervised learning algorithm to induce small, highly-accurate hypotheses [2]. Given a subset of features, the accuracy of the induced hypothesis is estimated by means of cross-validation.
A simple greedy algorithm, called backward-elimination, starts with the full set of features and removes the feature that most improves performance, or degrades performance slightly. A similar algorithm called forward-elimination, starts with the empty set of features and greedily adds features. Both algorithms can be blended into forward-backward elimination, by both considering adding or removing a feature at each step if it improves the performance.
Generating training sets that contain only a subset of features is straightforward in Matlab. Assume, that for the noisy dataset `traindata` with $\{x_1, x_2, x_3, x_4, x_5, f(x)\}$ we are interested in the subset $\{x_2, x_4, x_5, f(x)\}$.

```
>> traindata=load('f_train_noise.dat');
>> subset=[2 4 5 6];
>> traindatasub=traindata(:,subset);
```

Notice, that the order of features does not matter to the distance weighted averaging algorithm, as long as the target feature comes last. The Matlab function `setdiff` that computes set differences can be used to remove a feature from the set of current features. For example, to remove the third feature 4 from the current set of features $\{1, 3, 4, 5\}$ to obtain the set $\{1, 3, 5\}$ use the following Matlab code.

```
>> features = [1 3 4 5];
>> features=diffset(features,features(3));
```

Assignment 6:
Apply the instance based learning algorithm **`nn`** from the first assignment to the noisy dataset with three additional irrelevant features. Compute its error on the noisy testdata set.

error on testdata with complete set of features =
Use forward and backward elimination to identify the optimal subset of features. As in the second assignment, estimate the accuracy by means of leave-one-out crossvalidation. What are the optimal subsets of features obtained by forward- and backward-elimination and what errors do they demonstrate on the test set.
optimal subset of features with forward elimination =
optimal subset of features with backward elimination =
error on testdata with features from forward elimination =
error on testdata with features from backward elimination =

References

[1] C. Atkeson, A. Moore, and S. Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11:11–73, 1997.

[2] George H. John, Ron Kohavi, and Karl Pfleger. Irrelevant features and the subset selection problem. In *Proceedings of the 11th International Joint Conferences on Machine Learning*, pages 121–129. Morgan Kaufmann, 1994.

[3] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the International Joint Conferences on Artificial Intelligence IJCAI-95*. 1995.

[4] Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[5] Ian T. Nabney. *NETLAB - Algorithms for Pattern Recognition*. Springer, 2001.