CS786P—Topics in Artificial Intelligence
Computational Learning

Assignment 1

Fall 2001
Department of Computer Science
University of Waterloo

Due: in class, *Monday, October 29*
Worth: 25% of final grade (5 questions of equal weight, so each question is worth 5%)
Instructor: Dale Schuurmans, DC1310, x3005, dale@cs.uwaterloo.ca

---

Note 3 of the 5 questions require you to write small Matlab programs which are to be submitted by email. When finished, please send a *single* tar file containing all of your .m files to dale@cs.uwaterloo.ca with a subject heading "A1 solutions".

Question 1 (Learning real predictors—linear hypotheses)

In this exercise you will write simple MATLAB functions to learn minimum error linear functions and test them on simulated data. You will need to be familiar with `ops`, `slash`, `rand`, `randn`, `function`, `lp`, `plot`, and `print`. (Type `help` ⟨topic⟩ in MATLAB if you need to learn about any of these.)

We will consider three ways of generating data `X`, `y`. All will involve the same target function defined by the weight vector `u`$= [01...1]'$. The training data will have the form

$$\mathtt{X} = \begin{bmatrix} 1 & x_{1,1} & \cdots & x_{1,n-1} \\ \vdots & & & \vdots \\ 1 & x_{t,1} & \cdots & x_{t,n-1} \end{bmatrix} \quad \mathtt{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_t \end{bmatrix}$$

Data generation:

```
n = 2                                % dimension
t = 10                               % training size
u = [0; ones(n-1,1)]                 % target weights
sigma = 0.1                          % noise level
X = [ones(t,1) rand(t, n-1)]         % training patterns
```

Generative model 1: `y = X*u + randn(t,1)*sigma`
Generative model 2: `y = X*u + randn(t,1)./randn(t,1)*sigma`
Generative model 3: `y = X*u + randn(t,1).*randn(t,1)*sigma`

(a) (1%) Write a MATLAB function `minL2(X,y)` which takes a $t \times n$ matrix `X` and $t \times 1$ vector of target values `y` and returns an $n \times 1$ vector of weights `w2` corresponding to the minimum *sum of squared errors* linear function.

Note: You must implement the algorithm as shown in class. This involves solving a system of linear equations. You can do this using MATLAB's `slash` operator, but you *cannot* use the built in `regress` function.

Your function must be able to handle arbitrary n and t.

(b) (1%) Write a MATLAB function `minL1(X,y)` which returns $n \times 1$ vector of weights `w1` corresponding to the minimum *sum of absolute errors* linear function. (**Note**: You can use MATLAB's built in `lp` function to solve linear programs.)

(c) (1%) Write a MATLAB function `minLoo(X,y)` which returns $n \times 1$ vector of weights `woo` corresponding to the minimum *maximum absolute error* linear function. (**Note**: You can use MATLAB's built in `lp` function to solve linear programs.)

(d) (1%) For each of the generative models 1, 2 and 3:

A: Generate a random training set `X`, `y` using the model, and solve for each kind of linear function: `w2 = minL2(X,y)`, `w1 = minL1(X,y)`, and `woo = minLoo(X,y)`.

B: Produce a 2D plot of the training data and the three hypotheses corresponding to `w2`, `w1`, and `woo`.

```
clf
plot(X(:,2)',y','w*')
hold
plot([0 1],[w2(1) sum(w2)],'w-')
plot([0 1],[w1(1) sum(w1)],'w-.')
plot([0 1],[woo(1) sum(woo)],'w:')
print -deps experiment.1.3.<k>.ps
```

C: Report the three different kinds of error each of the three hypotheses obtained on the training data:

		type of error		
		L_1 error	L_2 error	L_∞ error
type	w1			
of	w2			
function	woo			

D: Generate `te = 1000` test examples from the same generative model and report the same matrix of results for the test data.

Hand in a plot and two tables for each generative model.

(e) (1%) For each generative model: Repeat (d) parts A, C and D 100 times and accumulate the sum of each kind of error for each kind of function in two matrices: one for the training errors and one for the testing errors. Report the *averages* for each kind of error and each kind of function in two tables (one training error and the other testing error).

Question 2 (Learning real predictors—polynomial bases)

In this exercise you will write a MATLAB function to compute minimum error polynomials and test it on simulated data. This will demonstrate overfitting dramatically! For this exercise it will be useful to know about MATLAB's `polyval` and `axis` functions.

We will consider two ways of generating the data, but will only consider one dimensional input so the data will (initially) have the form

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_t \end{bmatrix} \quad \mathtt{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_t \end{bmatrix}$$

Data generation:

```
t = 10            % training size
sigma = 0.1       % noise level
x = rand(t, 1)    % training patterns
```

Generative model 1: `y = x > 0.5` (step function)

Generative model 2: `y = 0.5 - 10.4*x.*(x-0.5).*(x-1) + sigma*randn(t,1)` (noisy cubic)

(a) (2%) Write a MATLAB function `minL2poly(x,y,d)` which takes a $t \times 1$ vector of training inputs **x** and a $t \times 1$ vector of target values `y` and returns a $(d+1) \times 1$ vector of weights `c` corresponding to the minimum *sum of squared errors* polynomial of degree `d`. That is, `c` should be a vector of weights `c` $= [c_1, ..., c_{d+1}]'$, which will be interpreted as specifying a polynomial $p(x) = c_1 x^d + c_2 x^{d-1} + \cdots + c_d x + c_{d+1}$.

Note: You must implement the algorithm as shown in class. This involves expanding each training input x_i into a vector $(x_i^d, x_i^{d-1}, ..., x_i, 1)$ to obtain

$$\mathtt{X} = \begin{bmatrix} x_1^d & x_1^{d-1} & \cdots & x_1 & 1 \\ \vdots & & & & \vdots \\ x_t^d & x_t^{d-1} & \cdots & x_t & 1 \end{bmatrix} \quad \mathtt{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_t \end{bmatrix}$$

and then solving a system of linear equations to obtain `c`. You can do this using MATLAB's `slash` operator, but you *cannot* use the built in `polyfit` function.

Your function must be able to handle arbitrary t and d.

(b) (2%) For each of the generative models 1 and 2:

A: Generate a random training set **x**, y using the model, and solve for the best fit polynomials of degree 1, 3, 5, and 9: `c1 = minL2poly(x,y,1)`, `c3 = minL2poly(x,y,3)`, `c5 = minL2poly(x,y,5)`, `c9 = minL2poly(x,y,9)`.

B: Produce a 2D plot of the training data and the four hypotheses corresponding to `c1`, `c3`, `c5`, and `c9`.

```
clf
axis([0 1 -0.5 1.5]);
hold
plot(X(:,d)',y','w*')
xx = (0:1000)/1000;
yy = <mean value of generative model at xx>
plot(xx,yy,'w:')

plot(xx,polyval(c1,xx),'w-')
plot(xx,polyval(c3,xx),'w-.')
plot(xx,polyval(c5,xx),'w-')
plot(xx,polyval(c9,xx),'w-.')
print -deps experiment.1.4.<k>.ps
```

C: Report the *mean* sum of squares error (*i.e.*, the sum of squares error divided by t) that each of the four hypotheses obtained on the training data:

		L_2 error
	c1	
	c3	
polynomial	c5	
	c9	

D: Generate `te = 1000` test examples from the same generative model and report the mean sum of squares error on the *test* data (*i.e.*, the sum of squares error divided by `te`) in the same form as above.

Hand in a plot and two tables for each generative model.

(c) (1%) For each generative model: Repeat (b) parts A, C and D 100 times and accumulate the sum of mean squared errors for each degree in two matrices: one for the training errors and one for the testing errors. Report the *averages* for each kind of mean error at each degree in two tables (one training error and the other testing error).

Question 3 (Learning real classifiers—support vector machines)

In this exercise you will write simple MATLAB functions to learn maximum margin linear discriminants and test them on simulated data. You will need to be familiar with `qp`.

We will consider three ways of generating data `X`, `y`. The training data will have the form

$$\mathtt{X} = \begin{bmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & & \vdots \\ x_{t,1} & \cdots & x_{t,n} \end{bmatrix} \quad \mathtt{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_t \end{bmatrix} \quad \text{for } x_{i,j} \in I\!R \text{ and } y_i \in \{-1, 1\}.$$

Data generation:

```
n = 2                % dimension
t = 10               % training size
u = ones(n,1)        % target weights   (models 1 & 2)
v = -0.5*n           % target offset    (models 1 & 2)
p_pos = 0.5          % prob of positive example
mu_pos = ones(n,1)   % mean loc for pos (model 3)
mu_neg = zeros(n,1)  % mean loc for neg (model 3)
```

Generative model 1: target linear discriminant

```
X = rand(t,n)
y = sign( X * u + v )
```

Generative model 2: target quadratic discriminant

```
X = rand(t,n)
y = sign( X.^2 * u + v )
```

Generative model 3: noisy linear discriminant (Naive Bayes—Gaussian)

```
X = randn(t,n)
y = 2 * (rand(t,1) < p_pos) - 1
pos = find(y > 0)
neg = find(y < 0)
X(pos,1) = X(pos,1) + mu_pos(1); X(pos,2) = X(pos,2) + mu_pos(2)
X(neg,1) = X(neg,1) + mu_neg(1); X(neg,2) = X(neg,2) + mu_neg(2)
```

(a) (1%) Write a MATLAB function `[lambda,v] = maxL2marg(X,y)` which takes a $t \times n$ matrix `X` and $t \times 1$ vector of target labels `y` and returns: an $1 \times t$ vector of Lagrange multipliers `lambda` and a scalar offset `v`, corresponding to the maximum L_2 margin linear discriminant classifier $\hat{y} = \text{sign}(\mathtt{w} \cdot \mathbf{x} + \mathtt{v})$ such that $\mathtt{w} = \sum_{i=1}^{t} \lambda_i y_i \mathbf{x}_i$.

Also write a MATLAB function `classify(Xtest,X,y,lambda,v)` which takes a $te \times n$ matrix `Xtest`, a $t \times n$ matrix `X`, a $1 \times t$ matrix `lambda` and a scalar `v`, and returns a $te \times 1$ vector of classifications `ytest` on the test patterns.

Note: For `maxL2marg` all you need to do is compute the vector of Lagrange multipliers $\boldsymbol{\lambda}$ that *maximizes* the objective

$$L(\boldsymbol{\lambda}) \quad = \quad \sum_{i=1}^{t} \lambda_i - \frac{1}{2} \sum_{i=1}^{t} \sum_{k=1}^{t} \lambda_i \lambda_k y_i y_k \, \mathbf{x}_i \cdot \mathbf{x}_k \qquad (1)$$

subject to the constraints $\sum_{i=1}^{t} \lambda_i y_i = 0$ and $\lambda_i \geq 0$. You can do this using MATLAB's `qp` operator to recover the weight vector `w`. To recover the offset value `v`, just solve for `v` in the equation $\lambda_i(y_i(\mathtt{w} \cdot \mathbf{x}_i + \mathtt{v}) - 1) = 0$ corresponding to the largest Lagrange multiplier λ_i.

Your functions must be able to handle arbitrary n, t and te.

(b) (1%) Write a MATLAB function `[lambda,v] = softL2marg(X,y,c)` which takes an additional scalar argument `c` and returns `lambda` and `v` corresponding to the maximum "soft" margin linear discriminant classifier.

(Recall that the maximum soft margin discriminant is defined by a weight vector `w` and offset `v` that minimizes the objective

$$L(\mathtt{w}, \mathtt{v}) \quad = \quad \|\mathtt{w}\|^2 + \mathtt{c} \sum_{i=1}^{t} \xi_i$$

subject to the constraints $y_i(\mathtt{w} \cdot \mathbf{x}_i - \mathtt{v}) \geq 1 - \xi_i$, and $\xi_i \geq 0$ for $i = 1, ..., t$, where ξ_i are new "slack" variables that allow the minimum margin bound to be violated by an amount ξ_i on example i. These slacks allows the learning algorithm to "give up" on a few training points to acheive a better minimum margin on the remaining points.)

Note: For `softL2marg` all you have to do is compute the vector of Lagrange multipliers $\boldsymbol{\lambda}$ that maximizes the same objective as above (1) subject to the same set of constraints, except for the slight modification that $0 \leq \lambda_i \leq \mathtt{c}$. This recovers the weight vector `w`. (Note that the slack variables ξ_i actually disappear in the dual formulation.) To recover the offset value `v`, just use the same procedure as above (using any λ_i such that $0 < \lambda_i < \mathtt{c}$).

Your function must be able to handle arbitrary n and t.

(c) (1%) Write a MATLAB function `[lambda,v] = polyL2marg(X,y,d)` which takes an additional argument `d` and returns `lambda` and `v` corresponding to the maximum L2 margin *polynomial* discriminant classifier $\hat{y} = \text{sign}(\mathtt{w} \cdot \Phi^d(\mathbf{x}) + \mathtt{v})$, where $\mathtt{w} = \sum_{i=1}^{t} \lambda_i y_i \Phi^d(\mathbf{x}_i)$ and $\Phi^d : \mathbf{x} \mapsto (1,\ \mathbf{x},\ \mathbf{x}\mathbf{x}^\top,\ \ldots,$ "degree d terms of $x_j \in \mathbf{x}$"). That is, Φ^d expands each input pattern $\mathbf{x}$ into a vector that contains all terms of up to degree `d` comprised of the original elements $x_j \in \mathbf{x}$.

Also write a MATLAB function `classifypoly(Xtest,X,y,lambda,v,d)` which returns a $te \times 1$ vector of classifications `ytest` on the test patterns.

Note: Your solutions must never expand the input patterns into their full representation, nor should they explicitly compute the weight vector `w`. You can avoid this computation by using the kernel function $K^d(\mathbf{x}_i, \mathbf{x}_k) = (\mathbf{x}_i \cdot \mathbf{x}_k + 1)^d$ which has the very useful property that $(\mathbf{x}_i \cdot \mathbf{x}_k + 1)^d = \Phi^d(\mathbf{x}_i) \cdot \Phi^d(\mathbf{x}_k)$ for any $\mathbf{x}_i$, $\mathbf{x}_k$ and `d`.

Your functions must be able to handle arbitrary n, t, te and `d`.

(d) (1%) For each of the generative models 1, 2 and 3:

A: Generate a random training set `X`, `y` using the model, and solve for each kind of discriminant function:
```
[lm, vm]  =  maxL2marg(X,y),
[ls, vs]  =  softL2marg(X,y,1),
[l2, v2]  =  polyL2marg(X,y,2),
[l3, v3]  =  polyL2marg(X,y,3),
```

B: Produce a 2D plot of the training data and the first two hypotheses corresponding to `lm,vm` and `ls,vs`. (It is too hard to plot the nonlinear decision boundaries of the the other two hypotheses.)

```
clf; axis([0 1 0 1]); hold; axis('square');
pos = find(y > 0); plot(X(pos,1)',X(pos,2)','g+');  % pos examples
neg = find(y < 0); plot(X(neg,1)',X(neg,2)','rx');  % neg examples
plot([0 1],[-v -v-u(1)]/u(2),'y:');                 % target discr
Z=X; for j = 1:n, Z(:,j) = X(:,j) .* y; end
wm = lm * Z;                                         %  max margin:
plot([0 1],[vm vm-wm(1)]/wm(2),'b-');                %      discrim
spm = find(lm > 0); plot(X(spm,1)',X(spm,2)','co'); %     supports
ws = ls * Z;                                         % soft margin:
plot([0 1],[vs vs-ws(1)]/ws(2),'m-');                %      discrim
sps = find(ls > 0); plot(X(sps,1)',X(sps,2)','yo'); %     supports
print -deps experiment.2.3.<k>.ps
```

C: Report the *mean* misclassification error (*i.e.*, the sum of misclassification errors divided by t) that each of the four hypotheses obtained on the training data:

		misclassification error	
classifier	`lm,vm`		(use `classify`)
	`ls,vs`		(use `classify`)
	`l2,v2`		(use `classifypoly`)
	`l3,v3`		(use `classifypoly`)

D: Generate `te = 1000` test examples from the same generative model and report the mean misclassification error on the *test* data (*i.e.*, the sum of misclassification errors divided by `te`) in the same form as above.

Hand in a plot and two tables for each generative model.

(e) (1%) For each generative model: Repeat (d) parts A, C and D 100 times and accumulate the sum of mean misclassifcation errors for each classifier in two matrices: one for the training errors and one for the testing errors. Report the *averages* of each kind of mean error for each classifier in two tables (one training and the other testing error).

Question 4 (Learning propositional classifiers)

Consider the problem of learning a propositional classifier $h : \{0,1\}^n \to \{0,1\}$ from training examples $D = \langle\langle \mathbf{x}_1, y_1\rangle, ..., \langle \mathbf{x}_t, y_t\rangle\rangle$. Early machine learning research considered a "general to specific" ordering of hypotheses, where h_1 is "more general" than h_2 if it classifies strictly more objects as positive; *i.e.*, if $h_2^{(-1)}(1) \subset h_1^{(-1)}(1)$. For a given hypothesis space H and training set D the *version space* $VS(H, D)$ is defined to be the hypotheses in H that correctly classify every example in D; *i.e.*, $VS(H,D) = \{h \in H : h(\mathbf{x}_i) = y_i \text{ for all } \langle \mathbf{x}_i, y_i\rangle \in D\}$. The *candidate elimination* algorithm implicitly represents a version space by explicitly storing (and updating) the boundary of "most general" consistent classifiers $G \subset H$ and the boundary of "most specific" consistent classifiers $S \subset H$. This exercise shows why we didn't use candidate elimination as a basis for constructing efficient learning algorithms in class.

Consider the class *PCONJ* of boolean functions over $\{0,1\}^n$ defined by conjunctions of positive variables $h(\mathbf{x}) = x_{i_1} \wedge x_{i_2} \wedge ... \wedge x_{i_k}$. In class we saw that a simple algorithm was able to find a consistent *PCONJ* classifier for given data (if one exists) in $O(nt)$ time, where n is the number of attribute variables, and t is the number of training examples.

(a) (1%) For *PCONJ*, what are the maximum and minimum sizes that the S set could have? (Assume that a consistent *PCONJ* hypothesis exists for the training data, and express your answer as a function of n, the number of attribute variables.)

(b) (1%) How about for the G boundary set? Demonstrate a training set D that forces the G set to be large.

(c) (1%) Given the answers to (a) and (b), what is a lower bound on the worst case run time of the candidate elimination algorithm on *PCONJ*?

(d) (1%) Answer questions (a), (b) and (c) for the class *PDISJ*, which consists of *disjunctions* of positive variables $h(\mathbf{x}) = x_{i_1} \vee x_{i_2} \vee ... \vee x_{i_k}$.

(e) (1%) Why didn't we use candidate elimination for any of our algorithms?

Question 5 (Learning theory—nearest neighbor classifiers)

In this question you will perform a simple theoretical analysis of the "nearest neighbor" classification strategy. Here classification rules $h : [0,1]^n \to \{0,1\}$ are defined by a set of labelled prototype points $\langle \mathbf{c}_1, \ell_1 \rangle, ..., \langle \mathbf{c}_N, \ell_N \rangle$, $\mathbf{c}_k \in [0,1]^n$, $\ell_k \in \{0,1\}$, such that test examples are classified according to the nearest prototype in the list. That is, we classify $\mathbf{x} \in [0,1]^n$ according to $h(\mathbf{x}) = \ell_K$ where $K = \operatorname{argmin}_{k=1}^N \|\mathbf{c}_k - \mathbf{x}\|_\infty$.[1] Express your answers to the following as a function of n.

(a) (1%) For any point $\mathbf{c} \in [0,1]^n$ what is the probability that a random example $\mathbf{x}$ falls within ϵ of $\mathbf{c}$; *i.e.*, such that $\|\mathbf{c} - \mathbf{x}\|_\infty \le \epsilon$? (You can assume that ϵ is less than the smallest distance from $\mathbf{c}$ to any boundary of $[0,1]^n$.)

Assume we draw training points from a uniform distribution on $[0,1]^n$. What is a lower bound on the number of random training points needed to guarantee that some point falls within ϵ of $\mathbf{c}$ with probability at least $1 - \delta$?

(b) (1%) The simplest nearest neighbor learning algorithm just uses the training data $\langle \mathbf{x}_1, y_1 \rangle$, ..., $\langle \mathbf{x}_t, y_t \rangle$ as the prototype centers. Consider the class of hypotheses $H_{\text{nn}} = \{h : [0,1]^n \to \{0,1\}\}$ that this algorithm might potentially generate. What is the cardinality of H_{nn}? What is the VC dimension of H_{nn}?

(c) (1%) Consider the restricted class of hypotheses H_{corners} defined by nearest neighbor classifiers with prototypes at the *corners* of the unit cube $[0,1]^n$. How many corners are there on the unit cube $[0,1]^n$? What is the cardinality of H_{corners}? What is the VC dimension of H_{corners}?

(d) (1%) (For this part, use $\|\cdot\|_2$ instead of $\|\cdot\|_\infty$.) Consider the even further restricted class of hypotheses H_{2proto} defined by nearest neighbor classifiers that use only two prototypes, $\langle \mathbf{c}_1, \ell_1 \rangle$ and $\langle \mathbf{c}_2, \ell_2 \rangle$, $\mathbf{c}_i \in [0,1]^n$, $\ell_i \in \{0,1\}$. What is the cardinality of H_{2proto}? What is the VC dimension of H_{2proto}?

(Hint: for this question you can use the fact that the VC dimension of the class of linear discriminant classifiers on $\mathbb{R}^n$ is $n+1$.)

(e) (1%) Answer the following for each of $H = \{H_{\text{nn}}, H_{\text{corners}}, H_{\text{2proto}}\}$. Given an arbitrary target function $f \in H$ and an arbitrary distribution P on $[0,1]^n$, what is a sufficient training sample size to guarantee that f can be learned to within ϵ accuracy with probability at least $1 - \delta$? Express your answer as a function of n, ϵ, and δ.

Can you identify a target function f in H_{corners} that would cause the simple nearest neighbor learning algorithm to perform poorly? (Just give an intuitive justification. It is not necessary to provide a formal analysis.)

[1] Note that one could use an arbitrary norm $\|\cdot\|$ to measure the distances between patterns, but to simplify the computation in this question we will use the L_∞ norm $\|\mathbf{c}_k - \mathbf{x}\|_\infty = \max_{j=1}^N |c_{k,j} - x_j|$ rather than the conventional Euclidean norm. It is easy to see that $\|\cdot\|_\infty$ actually gives a lower bound on $\|\cdot\|_2$.