6.867 Machine Learning

Problem Set 4

Due Friday 11/7

Please address all questions and comments about this problem set to `6.867-staff@ai.mit.edu`. You will need to use MATLAB for some of the problems but most of the code is provided. If you are not familiar with MATLAB, please consult

`http://www.ai.mit.edu/courses/6.867/matlab.html` and the links therein.

Problem 1: Regularized Least-Squares Feature Selection

In this problem we consider a regularized approach to feature selection in a simple regression context. Suppose we have training inputs $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n)'$ and corresponding outputs $\mathbf{y} = (y_1, \ldots, y_n)'$, where $y_i \in \mathcal{R}$. We train a linear predictor $\hat{y}(\mathbf{x}; \mathbf{w}) = \mathbf{w}'\phi(\mathbf{x})$ based on a collection of M features $\phi(\mathbf{x}) = (\phi_1(\mathbf{x}), \ldots, \phi_M(\mathbf{x}))'$. The estimation criterion is the following regularized least-squares objective:

$$J(\mathbf{w}; \lambda) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2}(y_i - \mathbf{w}'\phi(\mathbf{x}_i))^2 + \lambda \|\mathbf{w}\|_1 \tag{1}$$

where $\|\mathbf{w}\|_1$ is the l_1 norm

$$\|\mathbf{w}\|_1 = \sum_{k=1}^{M} |w_k| \tag{2}$$

We seek the optimum parameters $\hat{\mathbf{w}} = \hat{\mathbf{w}}(\lambda)$ (functions of λ) that minimize the regularized objective.

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w} \in \mathcal{R}^M} J(w; \lambda) \tag{3}$$

The use of the l_1 norm penalty leads to a sparse solutions such that, as λ becomes large, many of the parameters $\hat{\mathbf{w}}$ are forced to zero. The corresponding features are then ignored (irrelevant) in the prediction $\hat{y} = \hat{\mathbf{w}}'\phi(\mathbf{x})$.

To solve this l_1-regularized least-squares problem, we consider a simple *coordinate descent* approach to optimization. In this approach we adjust one parameter at a time so as to minimize the objective while keeping the remaining parameters fixed.

$$\hat{w}_k = \arg \min_{w_k} J(w; \lambda) \tag{4}$$

By letting k iterate over the set of all indices $\{1, \ldots, M\}$, successively refining single parameters, and repeating this iteration over parameters many times, the parameter vector asymptotically converges to the global minimum of the convex objective function.

In general, the coordinate descent method requires a subroutine to perform line minimization (minimization along the chosen coordinate). Here we can solve the line minimization in closed form.

The regularization penalty is not smooth, however, so the typical approach of setting derivatives to zero needs to be broken down into cases. While this is fairly simple in our regression context here, we follow a slightly more general approach, one that will be useful for solving other problems of this type.

How do we take derivatives of non-differentiable functions? The *subdifferential* of a convex function $f(w)$ is defined as

$$\partial f(w) = \{s | f(w + \Delta) \geq f(w) + s\Delta, \; \forall \, \Delta \in \mathcal{R}\} \tag{5}$$

In other words, a subdifferential is the range of slopes s such that the line through $(w, f(w))$ with slope s "supports" f, e.g. contains the graph of f in it's upper half-space. This is a set-valued generalization of the normal derivative and reduces to the normal derivative $\partial f(w) = \{\frac{\partial f(w)}{\partial w}\}$ whenever f is differentiable.

For example, the subdifferential of the absolute value function $f(w) = |w|$ (which is not differentiable at $w = 0$) is

$$\partial f(w) = \begin{cases} \{-1\}, & w < 0 \\ [-1, +1], & w = 0 \\ \{+1\}, & w > 0 \end{cases} \tag{6}$$

We will use the following result from non-smooth analysis:

Optimality Condition: $\hat{w}$ is a global minimizer of a convex function $f(w)$ if and only if $0 \in \partial f(\hat{w})$.

For example, the optimality condition $0 \in \partial f(w)$ for the absolute value function $f(w) = |w|$ holds if and only if $w = 0$. Hence, $w = 0$ is the (unique) global minimizer of $|w|$.

Below, we guide you through the analysis to solve the line minimization problem (4).

(1-1) (10pts) Show that the subdifferential of $J(w; \lambda)$ with respect to parameter w_k is

$$\begin{aligned} \partial_{w_k} J(\mathbf{w}; \lambda) &= (a_k w_k - c_k) + \lambda \partial_{w_k} |w_k| & (7) \\ &= \begin{cases} \{a_k w_k - (c_k + \lambda)\}, & w_k < 0 \\ [-c_k - \lambda, -c_k + \lambda], & w_k = 0 \\ \{a_k w_k - (c_k - \lambda)\}, & w_k > 0 \end{cases} & (8) \end{aligned}$$

with

$$a_k = \frac{1}{n} \sum_i \phi_k^2(\mathbf{x}_i) \tag{9}$$

$$c_k = \frac{1}{n}\sum_i \phi_k(\mathbf{x}_i)(y_i - \mathbf{w}'_{-k}\phi_{-k}(\mathbf{x}_i)) \quad (10)$$

where $\mathbf{w}_{-k}$ (respectively ϕ_{-k}) denote the vector of all parameters (features) except for parameter w_k (feature ϕ_k). (*Hint.* The subdifferential is given by $\frac{\partial J(\mathbf{w};0)}{\partial w_k} + \lambda\partial_{w_k}|w_k|$ since the mean squared term is differentiable.). Observe that the numbers a_k are non-negative (and typically positive) and constant w.r.t. the parameters. Each c_k depends on all parameters *except* for the parameter w_k that we wish to update. Can you interpret the coefficient c_k?

(1-2) (10pts) Solve the nonsmooth optimality condition for $\hat{w}_k$ s.t. $0 \in \partial_{w_k} J(\hat{w}_k)$. You may find it helpful to consider each of the following cases separately:

(a) $c_k < -\lambda$

(b) $c_k \in [-\lambda, +\lambda]$

(c) $c_k > +\lambda$

In each case, provide a hand-drawn plot $\partial_{w_k} J(w_k)$ versus w_k and label the zero-crossing $\hat{w}_k$. Express $\hat{w}_k$ as a function of a_k, c_k and λ. Also, provide a hand-drawn plot of $\hat{w}_k$ versus c_k. What role does the regularization parameter λ play in this context relative to the coefficient c_k?

We have provided you with training and test data in the file `reg_least_sq.mat`. Load this into MATLAB via `load reg_least_sq`. You should then find four structures `train_small`, `train_med`, `train_large` and `test` each containing an $n \times m$ matrix X and a $n \times 1$ vector `y`. Below, you will write code to implement and test the coordinate descent method for solving the l_1-regularized least squares problem (skeletons provided). For simplicity, we will only consider linear regression, e.g. set $\phi(\mathbf{x}) = \mathbf{x}$.

(1-3) (10pts) First, write a MATLAB subroutine based on the skeleton `reg_least_sq.m` to solve for $\hat{\mathbf{w}}$ given the training data (X, y), regularization parameter λ and an initial guess $\mathbf{w}_0$ to seed the coordinate descent algorithm. Your procedure should repeatedly iterate over $k \in \{1, \ldots, M\}$ and set $w_k = \hat{w}_k$ (use your earlier result). All but the evaluation of c_k and $\hat{w}_k$ is already provided in the skeleton `reg_least_sq.m`.

Test your procedure by evaluating $\hat{\mathbf{w}}(1)$ for any of the training sets. Initialize your procedure with $\mathbf{w}_0 = 0$.

In practice, it is often useful to solve for $\hat{w}(\lambda)$ for a range of λ values rather than for just one particular λ. For instance, we typically do not know what value of λ to use but might instead have some idea as to how large training error is still acceptable or how large $\|\mathbf{w}\|_1$ the penalty can be. This requires that we try many values of λ. One could also set λ based on cross-validation or other generalization error analysis.

(1-4) (10pts) Complete the skeleton `hw4_prob1.m` to run your coordinate descent algorithm for a sequence of λ values $[.01 : .01 : 2.0]$. At $\lambda = 0$, we set $\hat{\mathbf{w}}(0) = (X'X)^{-1}X'\mathbf{y}$, the usual least-squares parameter estimate.

Plot each of the following versus λ:

(a) the training error $J(\hat{\mathbf{w}}(\lambda); 0)$

(b) the regularization penalty $\|\hat{\mathbf{w}}(\lambda)\|_1$

(c) the minimized objective $J(\hat{\mathbf{w}}(\lambda); \lambda)$

(d) the number of non-zero parameters $\|\hat{\mathbf{w}}(\lambda)\|_0$ (the l_0 norm)

(e) the test error

Run this experiment for each of the three training data sets. Comment on the behaviour of each of these quantities as a function of λ. For each training set, what value of λ minimizes the test error? How does this vary with the size of the training set? How could we estimate (from the training data) the appropriate value of λ to use so as to approximately minimize the test error?

Other variations you may wish to consider include: select among polynomial features rather than components of the input vector directly, or find a solution by either (i) minimizing $\|\mathbf{w}\|_1$ subject to a constraint on the squared error (feature selection should not change the objective by more than η) or (ii) minimizing the least-squares error directly but subject to an l_1 constraint on how large the parameters can be, i.e. $\|\mathbf{w}\| < W$.

Problem 2: Boosting

Here we derive boosting algorithm from a slightly more general perspective that will be applicable for a class of loss functions including the exponential loss discussed in the lecture.

The goal is again to generate discriminant functions of the form

$$h_m(\mathbf{x}) = \alpha_1 h(\mathbf{x}; \theta_1) + \ldots + \alpha_m h(\mathbf{x}; \theta_m) \tag{11}$$

where you can assume that the weak learners $h(\mathbf{x}; \theta)$ are decision stumps whose predictions are ± 1; any other set of weak learners would be fine without modification. We successively add components to the overall discriminant function in a manner that will separate the estimation of the weak learners from the setting of the votes α to the extent possible. We will focus here on the algorithms and try to understand the complexity of these methods in problem 3.

Let's start by defining a set of useful loss functions. The only restriction we place on the loss is that it should be a monotonically decreasing and differentiable function of its argument. The argument in our context is $y_i h_m(\mathbf{x}_i)$ so that the more the discriminant function agrees

with the ± 1 label y_i, the smaller the loss. The simple exponential loss we have already considered, i.e.,

$$\text{Loss}(y_i h_m(\mathbf{x}_i)) = \exp(-y_i h_m(\mathbf{x}_i)) \tag{12}$$

certainly conforms to this notion. So does the logistic loss

$$\text{Loss}(y_i h_m(\mathbf{x}_i)) = \log\left(1 + \exp(-y_i h_m(\mathbf{x}_i))\right) \tag{13}$$

The logistic loss has a nice interpretation as a negative log-probability. Indeed, recall that for an additive logistic regression model

$$-\log P(y = 1|\mathbf{x}, \mathbf{w}) \quad = \quad -\log \frac{1}{1 + \exp(-z)} = \log(1 + \exp(-z)) \tag{14}$$

where $z = w_1\phi_1(\mathbf{x}) + \ldots + w_m\phi_m(\mathbf{x})$ and we omit the bias term for simplicity. By replacing the additive combination of basis functions with the combination of weak learners, or $h_m(\mathbf{x})$, we have an additive logistic regression model where the weak learners serve as the basis functions. The difference is that both the basis functions (weak learners) and the coefficients multiplying them will be estimated. In the logistic regression model we typically envision a fixed set of basis functions.

The estimation criterion for the combination is simply the empirical loss:

$$J(h_m) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}\Big(y_i h_m(\mathbf{x}_i)\Big) \tag{15}$$

where the summation is over the available training examples.

Let us now try to derive the boosting algorithm in a manner that can accomodate any loss function of the type discussed above. To this end, suppose we have already included $k-1$ component classifiers

$$h_{k-1}(\mathbf{x}) = \hat{\alpha}_1 h(\mathbf{x}; \hat{\theta}_1) + \ldots + \hat{\alpha}_{k-1} h(\mathbf{x}; \hat{\theta}_{k-1}), \tag{16}$$

and we wish to add another $h(\mathbf{x}; \theta)$. The estimation criterion for the overall discriminant function, including the new component with votes α, is given by

$$J(\alpha, \theta) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}\Big(y_i h_{k-1}(\mathbf{x}_i) + y_i \alpha h(\mathbf{x}_i; \theta)\Big) \tag{17}$$

Note that we explicate only how the objective depends on the choice of the last component and the corresponding votes since the parameters of the $k-1$ previous components along with their votes have already been set and won't be modified further.

We will first try to find the new component or parameters θ so as to maximize its potential in reducing the empirical loss, potential in the sense that we can subsequently adjust the

votes to actually reduce the empirical loss. More precisely, we set θ so as to *minimize* the derivative

$$\frac{d}{d\alpha}J(\alpha,\theta)_{|\alpha=0} = \frac{1}{n}\sum_{i=1}^{n}\frac{d}{d\alpha}\text{Loss}\Big(y_i h_{k-1}(\mathbf{x}_i) + y_i\alpha h(\mathbf{x}_i;\theta)\Big)_{|\alpha=0} \quad (18)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\text{dL}\Big(y_i h_{k-1}(\mathbf{x}_i)\Big)y_i h(\mathbf{x}_i;\theta) \quad (19)$$

where $\text{dL}(z) = d\text{Loss}(z)/dz$. Note this derivative $\frac{d}{d\alpha}J(\alpha,\theta)_{|\alpha=0}$ precisely captures the amount by which we would start to reduce the empirical loss if we gradually increased the votes for the new component with parameters θ. Minimizing this reduction seems like a sensible estimation criterion for the new component or θ. This plan permits us to first set θ and then subsequently optimize α to actually minimize the empirical loss.

Let's rewrite the algorithm slightly to make it look more like a boosting algorithm. First, let's define the following weights and normalized weights on the training examples:

$$W_i^{(k-1)} = -\text{dL}\Big(y_i h_{k-1}(\mathbf{x}_i)\Big) \quad (20)$$

$$\tilde{W}_i^{(k-1)} = \frac{W_i^{(k-1)}}{(\sum_j W_j^{(k-1)})} \quad (21)$$

These weights are guaranteed to be non-negative since the loss function is a decreasing function of its argument (its derivative has to be negative or zero). In this notation,

$$\frac{d}{d\alpha}J(\alpha,\theta)_{|\alpha=0} = -\frac{1}{n}\sum_{i=1}^{n}W_i^{(k-1)}y_i h(\mathbf{x}_i;\theta) \quad (22)$$

$$= -\frac{1}{n}(\sum_j W_j^{(k-1)})\cdot\sum_{i=1}^{n}\frac{W_i^{(k-1)}}{(\sum_j W_j^{(k-1)})}\,y_i h(\mathbf{x}_i;\theta) \quad (23)$$

$$= -\frac{1}{n}(\sum_j W_j^{(k-1)})\cdot\sum_{i=1}^{n}\tilde{W}_i^{(k-1)}\,y_i h(\mathbf{x}_i;\theta) \quad (24)$$

By ignoring the multiplicative constant (constant at iteration k) we will estimate θ by *minimizing*

$$-\sum_{i=1}^{n}\tilde{W}_i^{(k-1)}\,y_i h(\mathbf{x}_i;\theta) \quad (25)$$

where the normalized weights $\tilde{W}_i^{(k-1)}$ sum to one. This is the same as maximizing the weighted agreement with the labels.

We are now ready to cast the steps of the boosting algorithm in a form similar to the algorithm given in the lectures

Step 1 Find any classifier $h(\mathbf{x}; \hat{\theta}_k)$ that performs better than chance with respect to the weighted training error:

$$e_k = 0.5 - 0.5 \sum_{i=1}^{n} \tilde{W}_i^{(k-1)} \, y_i h(\mathbf{x}_i; \hat{\theta}_k) \tag{26}$$

Note that the weights are normalized.

Step 2 We set the votes α_k for the new component by minimizing the overall empirical loss

$$J(\alpha, \hat{\theta}_k) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}\Big(y_i h_{k-1}(\mathbf{x}_i) + y_i \alpha h(\mathbf{x}_i; \hat{\theta}_k) \Big) \tag{27}$$

Step 3 We recompute the normalized weights for the next iteration according to

$$\tilde{W}_i^{(k)} = -c \cdot \text{dL}\Big(\overbrace{y_i h_{k-1}(\mathbf{x}_i) + y_i \hat{\alpha} h(\mathbf{x}_i; \hat{\theta}_k)}^{y_i h_k(\mathbf{x}_i)} \Big) \tag{28}$$

where c is chosen so that $\sum_i \tilde{W}_i^{(k)} = 1$.

(2-1) Show that the three steps in the algorithm corresponds exactly to AdaBoost when the loss function is the exponential loss $\text{Loss}(z) = \exp(-z)$. More precisely, show that the setting of α_k based on the new weak learner and the weight update to get $\tilde{W}_i^{(k)}$ would be identical to AdaBoost in this case.

(2-2) What are the normalized weights if we use the logistic loss instead? Express the weights as a function of the agreements $y_i h_k(\mathbf{x}_i)$, where we have already included the k^{th} weak learner.

(2-3) Show that for any valid loss function of the type discussed above, the new component $h(\mathbf{x}; \hat{\theta}_k)$ just added at the k^{th} iteration would have weighted training error exactly 0.5 relative to the updated weights $\tilde{W}_i^{(k)}$. If you prefer, you can show this only in the case of the logistic loss.

(2-4) Now, we have provided you with most of the boosting algorithm with the logistic loss and decision stumps. The available components are `build_stump.m`, `eval_boost.m`, `eval_stump.m`, and the sceleton of `boost_logistic.m`. The sceleton includes a bi-section search of the optimizing α but is missing the piece of code that updates the weights. Please fill in the appropriate weight update.

`model = boost_logistic(X,y,10);` returns a cell array of `10` stumps. The routine `eval_boost(model,X)` evaluates the combined discriminant function corresponding to any such array.

(2-5) We have provided a dataset pertaining to cancer classification (see `cancer.txt` for details). You can get the data by `data = loaddata;` which gives you training examples `data.xtrain` and labels `data.ytrain`. The test examples are in `data.xtest` and `data.ytest`. Run the boosting algorithm with the logistic loss for 50 iterations and plot the training and test errors as a function of the number of iterations. Interpret the resulting plot.

Note that since the boosting algorithm returns a cell array of component stumps, stored for example in `model`, you can easily evaluate the predictions based on any smaller number of iterations by selecting a part of this array as in `model{1:10}`.

Problem 3: VC-dimension

In this problem, we will investigate the VC-dimension of various sets of classifiers. A *classifier* is a function from some input space to the binary class labels +1,−1. A classifier can also be described as a subset of the input space which gets the label +1. For example, a linear classifier in the plane $\mathcal{R}^2$, can be described by a half-plane. For this reason, we can discuss the family of linear classifiers as the set of all half-planes (and possibly also the plane itself and the empty set).

We say that a class (i.e. set) $\mathcal{H}$ of classifiers *shatters* a set of points $X = \{x_1, x_2, \ldots, x_n\}$ if we can classify the points in X in all possible ways. More precisely, *for all* 2^n possible labeling vectors $y_1, y_2, \ldots, y_n \in \{-1, 1\}^n$, there exists a classifier $h \in \mathcal{H}$ such that $h(x_i) = y_i$ for all i. For any possible labelings of the points, there has to be a classifier in our set that reproduces those labels. Using the set notation for classifiers, this means that for any subset of examples $X' \subseteq X$ (indicating the subset of points labeled +1), there exist a classifier $h \in \mathcal{H}$ such that $X \cap h = X'$ (the set of points for which h assigns label +1 includes X' but not the rest of X). It is important to understand that shattering is a property of a set of classifiers– not of a single classifier. A single classifier cannot shatter even a single point.

The *VC-dimension* of a set $\mathcal{H}$ of classifiers is the size of the largest set of points X that can be shattered by $\mathcal{H}$.

Part I: Linear Classifiers

Consider the class $\mathcal{H}_d$ of linear classifiers in $\Re^d$. Each classifier in this class is parameterized by a vector $\mathbf{w} \in R^d$ and has the form:

$$h_{\mathbf{w}}(\mathbf{x}) = \begin{array}{ll} 1 & \text{if } \mathbf{w}'\mathbf{x} > 0, \\ -1 & \text{otherwise} \end{array} .$$

Note that we do not allow a bias term w_0, and thus the separating hyperplanes must pass through the origin.

(3-1) Show that there exists a set of d points in $\Re^d$ that can be shattered by $\mathcal{H}_d$. What does this tell us about the VC-dimension of $\mathcal{H}_d$?

Next, we would like to prove that no set of $d+1$ points can be shattered. To do so, we will rely on the convexity of half-spaces. Note that both closed and open half-spaces, and so both the positive and negative regions, are convex. So, for example, any convex combination of positive points is also positive.

We will also use the following theorem:

Radon's Theorem: *Any set of d+2 points in $\Re^d$ can be partitioned to to sets S_1 and S_2 such that the convex hulls of S_1 and S_2 intersect.*

(3-2) Prove that no set of $d+1$ points can be shattered by $\mathcal{H}_d$. Conclude that the VC dimension of $\mathcal{H}_d$ is exactly d. (Hint: for sets of points that do not include the origin, use Radon's Theorem on the points plus the origin (note that the origin is always classified as negative). Use a separate, simpler, argument for why a set of points that includes the origin cannot be shattered).

(3-3) We would now like to use the above result in order to bound the VC dimension of linear classification in *feature space*. We say that a classifier class $\mathcal{H}$ is linear over d features if every classifier $h \in \mathcal{H}$ is of the form

$$h(x) = \begin{array}{ll} 1 & \alpha_1\phi_1(x) + \alpha_2\phi_2(x) + \cdots + \alpha_d\phi_d(x) > 0, \\ -1 & \text{otherwise} \end{array}$$

for *fixed* feature functions $\phi_1(x), \ldots, \phi_2(x)$, but where α varies between the classifiers in the class. Show that the VC dimension of such a class $\mathcal{H}$ is *at most* d.

Note that the VC dimension of the class might be *less then* d, for example if some of the features depend on each other, or if not all α vectors are allowed. This result can be used to bound the VC dimension of many sets of classifiers such as linear classifiers based on second order polynomial features.

Part II: Decision Stumps

We would now like to analyze the VC-dimension of the set of classifiers given by a linear combination of m decision stumps in $\Re^d$. First, let us consider the VC-dimension of the set of decision stumps themselves. Recall that a stump classifier $h_{i,a,b}$ in $\Re^d$ (in set notation) is specified by an axis-parallel half-space: $h_{i,a,b} = \{\mathbf{x} \in \Re^d | ax_i - b \geq 0\}$ (a can always be taken to be $+1$ or -1).

(3-4) For any set of n points in $\mathcal{R}^d$, show that the stumps can only classify the n points in at most $2dn$ different ways. That is, there are at most $2dn$ different labelings on the points which are attainable using stump classifiers. (Hint: count the number of possible classifications using stumps on each axis)

(3-5) Use the above bound to show that the VC-dimension of stumps is at most $2(\log_2 d+1)$. (Hint: use the fact that $\log_2 n < n/2$ and so $n - \log_2 n > n/2$)

(3-6) *(Optional)* Consider a $d = 2^\delta$ dimensional space. Suggest a set of δ point in $\mathcal{R}^d = \mathcal{R}^{(2^\delta)}$ that can be shattered by stumps, and show how it can be shattered. Conclude that the VC-dimension of stumps in R^d is at least $\lfloor \log_2 d \rfloor$ (i.e. $\log d$ rounded down to the nearest integer).

Combining the results, we see that the VC-dimension of stumps is roughly $\log d$.

Although the VC-dimension of stumps is fairly low, combining stumps, as we did in AdaBoost, is much more expressive. A linear combination of m stumps is a classifier given by $h(\mathbf{x}) = \text{sign}(\alpha_1 h(\mathbf{x};\theta_1) + \cdots + \alpha_m h(\mathbf{x};\theta_m))$, where $h(\mathbf{x};\theta_k)$ are decision stumps, and $\alpha_k \in \mathcal{R}$ are their corresponding coefficients (the coefficients would be positive in boosting but we will allow them to be negative here as well). Note that this is not a linear combination of fixed basis functions, since each stump in the combination can be adjusted.

Any fixed set of m stumps provide a feature representation of the input points $\mathbf{x}$, where the $m-$dimensional feature vector is given by $\phi(\mathbf{x}) = [h(\mathbf{x};\theta_1), \ldots, h(\mathbf{x};\theta_m)]'$. The coefficients $\alpha_1, \ldots, \alpha_m$ therefore define a linear classifier in this feature space. We have already shown that the VC-dimension of such a set of linear classifiers is at most m (the number of terms in the linear combination). We need to know how many labelings we can generate with this set of classifiers. This is given by the following result:

Lemma: *A set of classifiers with VC-dimension d can generate at most $(2n/d)^d$ labelings over $n \geq d$ points.*

The rationale behind this result goes as follows. In principle any subset of $r \leq d$ out of n points could be shattered by the set of classifiers. Thus, each such subset contributes 2^r possible labelings, and there are many ways of selecting subsets of size r out of n. The lemma gives a bound on the sum of all these up to $r = d$:

$$\sum_{r=0}^{d} 2^r \left(\begin{array}{c} n \\ r \end{array} \right) \leq (2n/d)^d \tag{29}$$

(3-7) Using the above result show that for any set of n points in $\Re^d$, the number of labelings of the n points using linear combinations of m decision stumps is at most $(2dn)^m (2n/m)^m$. (*Hint:* use the above lemma for any distinct set of n binary feature vectors $\phi(\mathbf{x}_1), \ldots, \phi(\mathbf{x}_n)$, where $\phi(\mathbf{x}_i) = [h(\mathbf{x}_i;\theta_1), \ldots, h(\mathbf{x}_i;\theta_m)]'$, that we can generate by adjusting the m stumps. The number of distinct binary feature vectors is related to the number of possible labelings that the stumps can generate.)