Problem Set 4

See website for due date

What to submit (staple in this order):

1. grade4.txt
2. Write up (put all code, comments, figures and results into a single document),
3. Written question answers

Grading. We will randomly choose a subset of submissions and verify that 1) the code produces the answers and 2) the points computation in "grade.txt" is correct. There are three written problems, one of them will be chosen at random and will be worth 30 points. Total: 130 points.

Note: The programming part of the assignment is different from the rest: it is less structured, with minimal provided starter code. It is therefore up to you to understand how each part works (which may require some research), implement it, and test it thoroughly before putting the parts together. Use the class discussion list as a resource if you get stuck, but do not reveal specific answers.

Programming Questions: Boosting

1. Boosting Overview

The goal of the general approach of boosting is to improve the accuracy of any given learning algorithm. In boosting we first create a classifier with accuracy on the training set greater than chance, and then add new component classifiers to form an ensemble whose joint decision rule has arbitrarily high accuracy. In such a case we say the classification performance has been "boosted." In overview, the technique trains successive component classifiers with a subset of the training data that is "most informative" given the current set of component classifiers. The subset is sampled from a distribution over the data points. Classification of a test point $\boldsymbol{x}$ is based on the combined outputs of the component classifiers, e.g., via voting, or (weighted) average.

2. AdaBoost

The following is an explanation and pseudocode from (Duda, Hart and Stork 2001). Please refer to the assigned readings (Shapire's tutorial, Bishop) for more information.

There are a number of variations on basic boosting. The most popular, AdaBoost — from "adaptive" boosting — allows the designer to continue adding weak learners until some desired low training error has been achieved. In AdaBoost, each training pattern receives a weight which determines its probability of being selected for a training set for an individual component classifier. If a pattern is accurately classified, then its chance of being used again in a subsequent component classifier is reduced; conversely, if a pattern is not accurately classified, then its chance of being used again is raised. In this way, AdaBoost "focuses in" on the informative or "difficult" patterns.

Let the training patterns and their labels in dataset **D** be denoted x_i and y_i, respectively and let $W_k(i)$ be the kth (discrete) distribution over all these training samples. We initialize these weights W_k to to be uniform in iteration k=1. On each subsequent iteration k, we first draw a training set at random according to these weights, and train a component classifier (weak learner) C_k on the patterns selected. Next, we increase weights of examples misclassified by C_k and decrease weights of examples correctly classified by C_k. Patterns chosen according to this new distribution are used to train the next classifier, C_{k+1}, and the process is iterated. The AdaBoost procedure is then:

1 **begin initialize** $\mathcal{D} = \{\mathbf{x}_1, y_1, \mathbf{x}_2, y_2, \ldots, \mathbf{x}_n, y_n\}, W_1(i) = 1/n, k_{max}$
2 $k \leftarrow 0$
3 **do** $k \leftarrow k + 1$
4 Train weak learner C_k using $\mathcal{D}$ sampled according to distribution $W_k(i)$
5 $E_k \leftarrow$ Training error of C_k measured on $\mathcal{D}$ using $W_k(i)$
6 $\alpha_k \leftarrow \frac{1}{2}\ln[(1 - E_k)/E_k]$
7 $W_{k+1}(i) \leftarrow \frac{W_k(i)}{Z_k} \times \begin{cases} e^{-\alpha_k} & \text{if } h_k(\mathbf{x}_i) = y_i \text{ (correctly classified)} \\ e^{\alpha_k} & \text{if } h_k(\mathbf{x}_i) \neq y_i \text{ (incorrectly classified)} \end{cases}$
8 **until** $k = k_{max}$
9 **return** C_k and α_k for $k = 1$ to k_{max} (ensemble of classifiers with weights)
10 **end**

Note especially in line 5 that the error for classifier C_k is determined with respect to the distribution $W_k(i)$ over D on which it was trained. In line 7, Z_k is simply a normalizing constant computed to insure that $W_k(i)$ represents a true distribution, and $h_k(x_i)$ is the category label (+1 or -1) given to pattern x_i by component classifier C_k. Naturally, the

loop termination criterion in line 8 can be replaced by the criterion of sufficiently low training error of the ensemble classifier.

The final classification decision of a test point x is based on a discriminant function that is merely the weighted sums of the outputs given by the component classifiers:

$$g(\mathbf{x}) = \left[\sum_{k=1}^{k_{max}} \alpha_k h_k(\mathbf{x}) \right].$$

The classification decision for this two-category case is then simply sign[g(x)].

So long as each component classifier is a weak learner, the total training error of the ensemble can be made arbitrarily low by setting the number of component classifiers, k_{max}, sufficiently high. To see this, notice that the training error for weak learner C_k can be written as $E_k = 1/2 - G_k$ for some positive value G_k. Thus the ensemble training error is:

$$\begin{aligned} E = \prod_{k=1}^{k_{max}} \left[2\sqrt{E_k(1-E_k)} \right] &= \prod_{k=1}^{k_{max}} \sqrt{1-4G_k^2} \\ &\leq \exp\left(-2 \sum_{k=1}^{k_{max}} G_k^2 \right) \end{aligned}$$

It is sometimes beneficial to increase k_{max} beyond the value needed for zero ensemble training error as this may improve generalization. While in theory a large k_{max} could lead to overfitting, simulation experiments have shown that overfitting rarely occurs, even when k_{max} is extremely large.

This outline and the framework code in ex4.m will guide you through the exercise.

Note: For most parts of this exercise, there is no reference value. For those parts, you should produce your own intermediate outputs that demonstrate they work as intended, along with an explanation in your write up, and include your well-commented source code.

2.0 Visualizing the data

The first part of ex4.m will load the training data and display it on a 2-dimensional plot by calling the function plotData; it should display a figure like Figure 1. Circles and crosses are used to represent the class labels for this two-class problem. Note that the classes are not linearly separable, so a linear classifier will not do well.

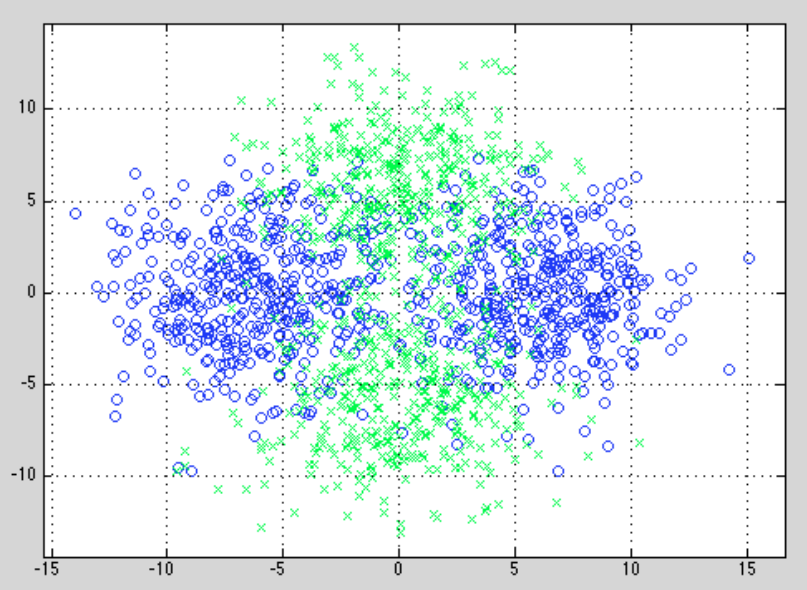

Figure 1a: Visualizing the training data

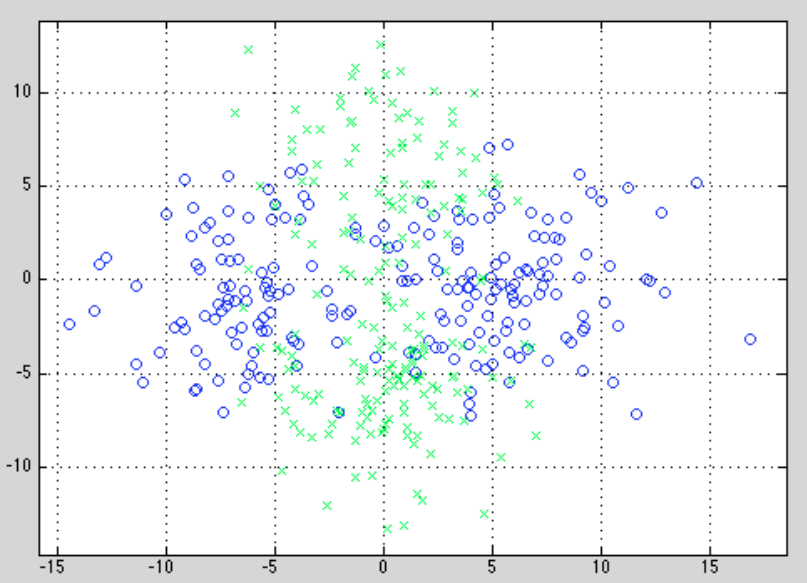

Figure 1b: Visualizing the testing data

2.1 Weaker Learner and Decision Boundary (weakLearnerC.m, plotDecisionBoundary.m)

These are the first two functions you need to implement. You may implement any classifier you want for this weak learner. The only constraints are that (1) it should outperform random guessing and (2) it should be very fast to train.

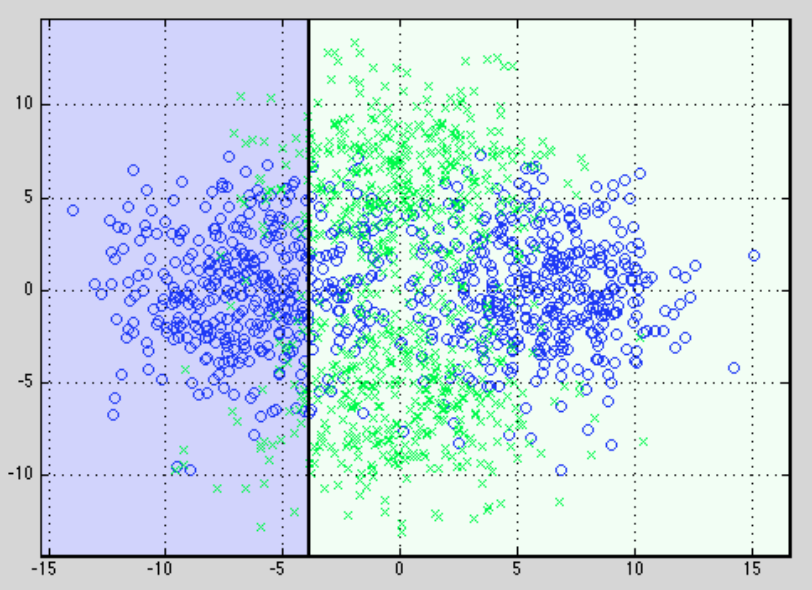

Figure 2: Decision boundary of one possible weak learner

Figure 2 shows one possible weak learner with its decision boundary. This is simply a function that checks if the value of one dimension of the input is greater than some threshold (its only parameter), and outputs +1 if it is, -1 otherwise. The parameter that minimizes training error can be found simply by doing an exhaustive search across the dimensions and the range of training input values. Note, you can use both greater-than and less-than checking. Other weak learner classifiers can be used.

For this part of the exercise, implement the weak learner using the following input and output format:

Input/output format:
function [test_targets,...] = weakLearnerC(train_patterns, train_targets, test_patterns, other_params)
test_targets: Predicted targets for test data
train_patterns: Train patterns
train_targets: Train targets
test_patterns: Test patterns

[15 points for this part, put your plots, source code and its explanation in the writeup].

Next, implement a modified version of the function that plots a decision boundary (from problem set 2), which takes an arbitrary hypothesis. Use it to visualize a sample weak learner as in Figure 2.

Input/output format:
function plotDecisionBoundary(X, y)
plots the data points with + for the positive examples and o for the negative examples, and plots the decision boundary as a black line. X is assumed to be a either
1) Mx3 matrix, where the first column is an all-ones column for the intercept.
2) MxN, N>3 matrix, where the first column is all-ones

[5 points for this part, put your plots, source code and its explanation in the writeup].

2.2 Weighted Training Error E (errorE.m)

Next, implement the function that computes the weighted training error in line 5 of the algorithm.

```
Train set errors: Class 1: 0.56. Class 2: 0.078. Total: 0.31
```

Figure 3. Weighted Training error E_k

Figure 3 shows a sample weighted training error Ek for some iteration k. Use the following format (this function should be very straight forward).

Input/output format:
function Ek = errorE(Hk, train_targets, Wk)
Ek: training error of Ck
Hk: weak classifier's predictions on the training data
train_targets: true labels of training data
Wk: distribution Wk

[20 points for this part, put your source code, its explanation, and the result (screen shots, sample numbers, etc.) in the writeup].

2.3 Distribution W (distributionW.m)

In this function, you need to generate a new distribution according to the pseudo code lines 6~7. For the 'other params' you may want to pass in the training labels and predictions of classifier Ck.

Input/output format:
function Wkplus1 = distributionW(Wk, Ek, other params)
Wkplus1: distribution Wkplus1

Wk: distribution Wk
Ek: error

[10 points for this part, put your source code, intermediate results and explanation in the writeup].

2.4 Sample from Distribution (distSample.m)

In the next part, you should implement a function that, given a discrete distribution over the space {1,...,m}, returns n samples from that distribution.

Recall that a discrete random variable X is a random variable that has a probability mass function p(x) = P(X = x) for any x in S, where S = {x_1, x_2, ..., x_k} denotes the sample space, and k is the (possibly infinite) number of possible outcomes for the discrete variable X, and suppose S is ordered from smaller to larger values. (In our case, S is {1,...,m}.) Then the CDF, F for X is

$$F(x_j) = \sum_{i \leq j} p(x_i)$$

Discrete random variables can be sampled by slicing up the interval (0, 1) into subintervals which define a partition of (0, 1):

$$(0, F(x_1)), (F(x_1), F(x_2)), (F(x_2), F(x_3)), ..., (F(x_{k-1}), 1),$$

generating U = Uniform(0, 1) random variables, and seeing which subinterval U falls into. Octave/Matlab, as many other programming languages, have a built in function for sampling from a uniform distribution. Let

$$I_j = \mathcal{I}\left(U \in (F(x_j), F(x_{j+1}))\right)$$

Then

$$P(I_j = 1) = P\left(F(x_{j-1}) \leq U \leq F(x_j)\right) = F(x_j) - F(x_{j-1}) = p(x_j)$$

where $F(x_0)$ is defined to be 0. So, the probability that $I_j = 1$ is same as the probability that $X = x_j$, and this can be used to generate from the distribution of X. As an example, suppose that X takes values in S = {1, 2, 3} with probability mass function defined by the following table:

p(x)	x
p_1	1

p_2	2
p_3	3

To generate from this distribution we partition (0, 1) into the three sub-intervals (0, p_1), (p_1, p_1+p_2), and (p_1+p_2, p_1+p_2+p_3), generate a Uniform(0, 1) variable, and check which interval the variable falls into.

Implement the above approach as the following function:

Input/output format:
function x = distSample(W, n)
Returns n values sampled from the discrete distribution specified by vector W (should result in error if W is not a valid discrete probability distribution.)

As a sanity check, sample ten values from the following probability vector:
`distSample([0 0.3 0.7 0 0],10)`
You should get something like:
`2 3 2 3 3 2 3 2 3 3`

[10 points for this part, put your source code, intermediate results and explanation in the writeup].

2.5 AdaBoost (AdaBoost.m)

Finally, put all the functions you implemented together in the final AdaBoost.m function, and run it on the provided training dataset using ex4.m. Plot the resulting boundary as in Figure 4.

Input/output format:
function [test_targets, E] = AdaBoost(train_patterns, train_targets, test_patterns, Kmax, weakLearnerC, other_params)
train_patterns: Train patterns
train_targets: Train targets
test_patterns: Test patterns
test_targets: Predicted targets
Kmax: maximum number of iterations
E: error on the training data

Experiment with setting k_{max} to 20, 50, 100 and plot the training and test error as a function of these. Also plot the decision boundary for each. What conclusion can you draw from these plots?

[20 points for this part, put your source code, its explanation and the result (plots, conclusion, etc.) in the writeup].

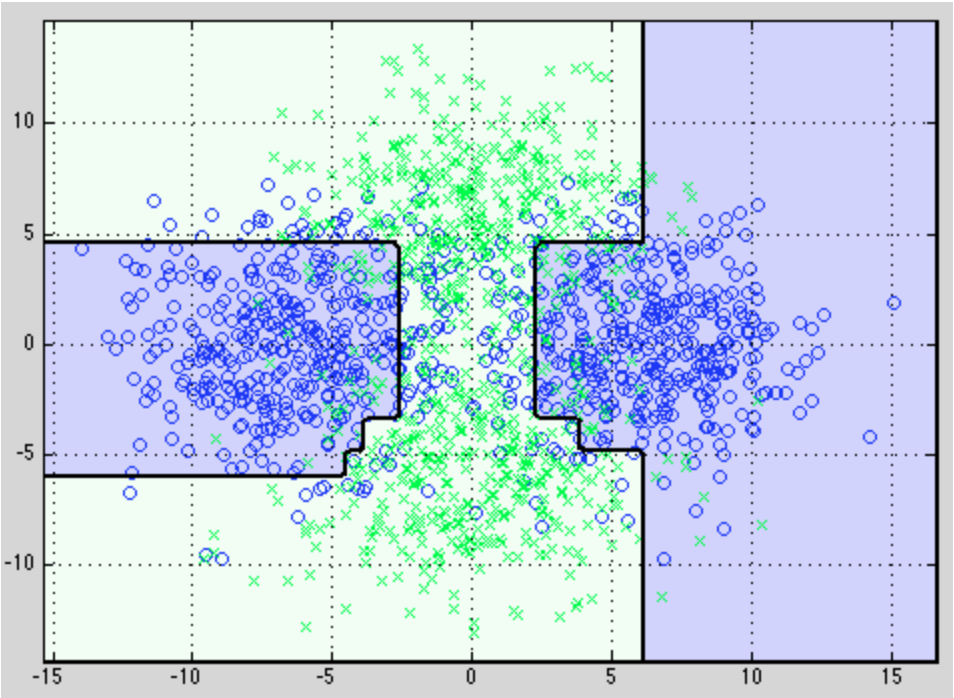

Figure 4. AdaBoost with kmax set to 20

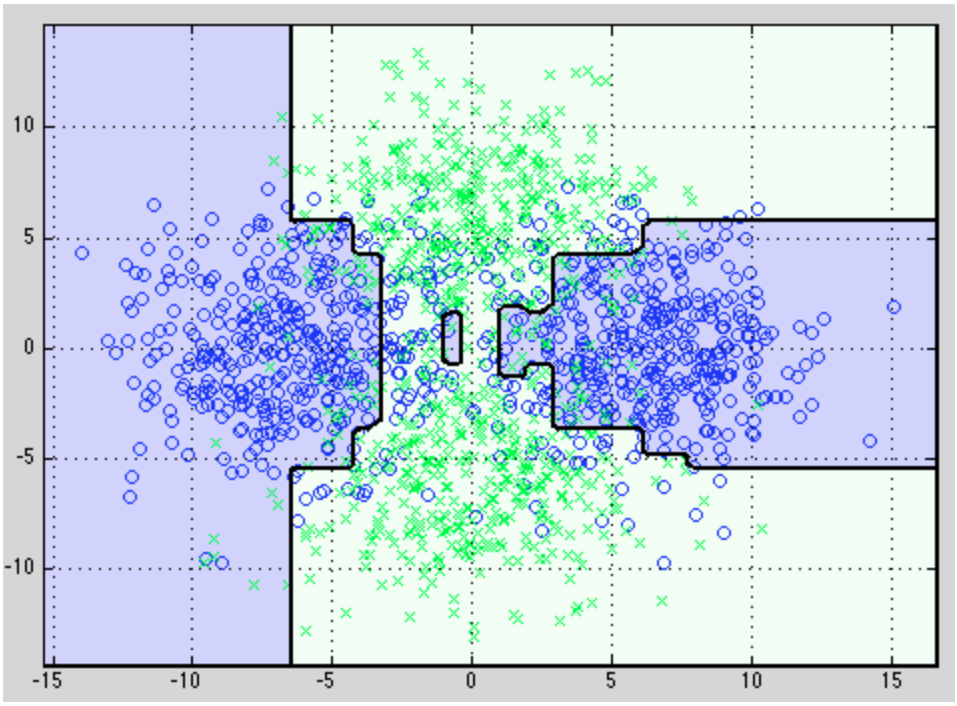

Figure 5. AdaBoost with kmax set to 50

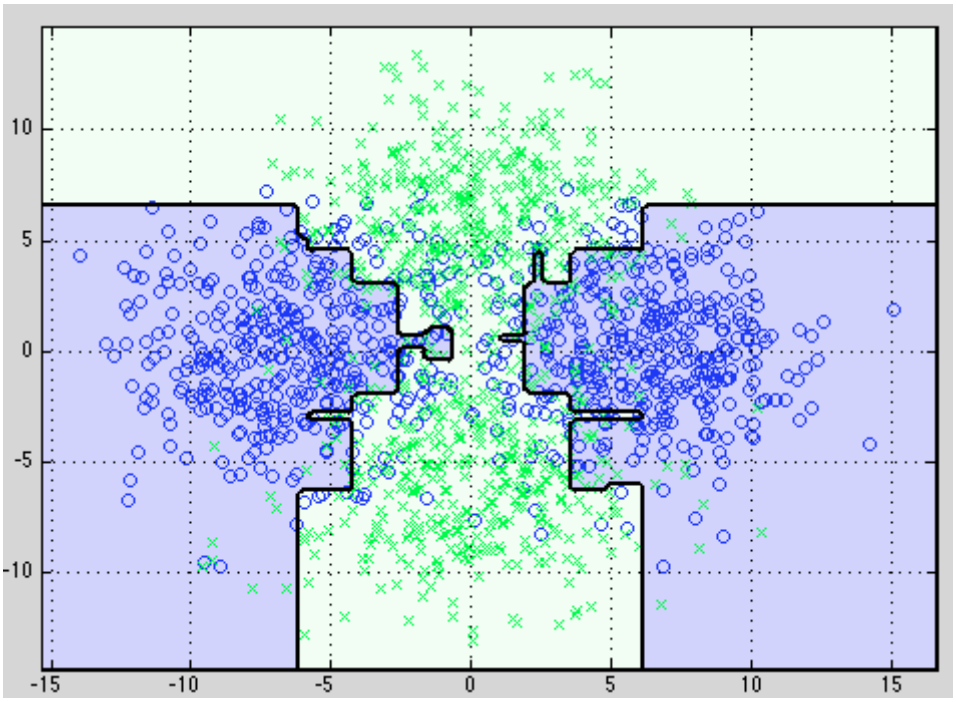

Figure 6. AdaBoost with kmaxset to 100

2.6 Smiley Face Dataset

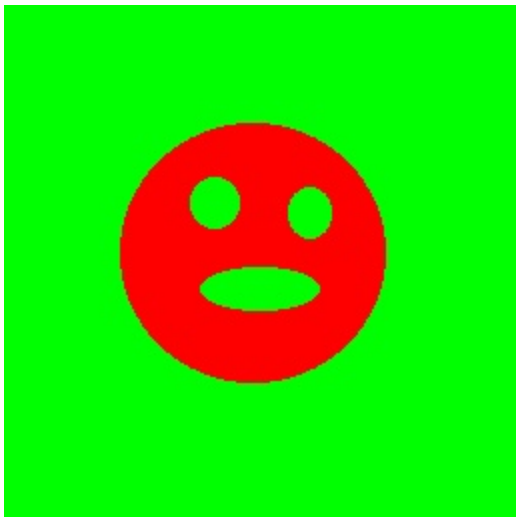

Figure 7: Smiley Face

In this part of the exercise, you will use your Adaboost code on an image dataset (smile.bmp/smile.mat), which is included in the starter code folder. First, we need to convert the image data locations into two classes: -1 for green points and 1 for red points. For example, the first data point will correspond to the upper leftmost (green) pixel and will be x=[1 1], y= -1. We have done this for you and included the resulting training data in smile_10_percent.mat. Note that we subsampled the data and kept only 10% to speed up training.

Split this data set randomly into training (80%) and testing (20%). Using ex4.m to call functions, train your Adaboost algorithm against this training dataset, and test on the test set. Plot the resulting classifier boundary on both datasets.

Did AdaBoost work on this dataset? What makes it a difficult dataset compared to the previous one?

[20 points for this part, put your source code, explanation, and the result (screen shots, numbers, etc.) in the writeup].

Written Questions

Problem 1: Boosting

This is an exercise from Bishop that asks you to derive part of the Adaboost algorithm from the point of view of sequential minimization of the exponential error function. First you should read Ch 14.3.1, which describes this alternative view of Adaboost.

14.6 ($\star$) **www** By differentiating the error function (14.23) with respect to α_m, show that the parameters α_m in the AdaBoost algorithm are updated using (14.17) in which ϵ_m is defined by (14.16).

Problem 2: Kernels

Read section 6.2 in Bishop and do the following exercise.

6.5 ($\star$) **www** Verify the results (6.13) and (6.14) for constructing valid kernels.

Problem 3: Maximum Margin Classifiers

Read section 7.1 and do the following exercise.

7.2 ($\star$) Show that, if the 1 on the right-hand side of the constraint (7.5) is replaced by some arbitrary constant $\gamma > 0$, the solution for the maximum margin hyperplane is unchanged.

Additional Practice Problems (Ungraded)

Read section 6.1 on Dual Representations in Bishop, and work through all of the steps of the derivations in equations 6.2-6.9. You should understand how the derivation works in detail.