COMPSCI 527 Homework 3

Cody Lieu, Yixin Lin

October 8, 2015

Problem 1(a)

```
function Y = compress(X, V, mu, k)

if isempty(k) || k > size(V, 2)
    k = size(V, 2);
end

Ac = X - ones(size(X, 1), 1)*mu;

V = V(:, 1:k);
Y = Ac*V;

end
```

We made a figure, but we plot over it repeatedly in the following questions. If you want to see what it looks like, look at those.

Problem 1(b)

$$\sigma^2 = [4.22820.24270.07820.0238]^T$$

Since we project down to the first two dimensions, the fraction of the variance is

$$\frac{4.2282 + 0.2427}{4.2282 + 0.2427 + 0.0782 + 0.0238} = 0.97769468$$

. So the percentage of variance we capture with our PCA to 2 dimensions is 97.7769%.

Problem 2(a)

```
function tau = trainTree(S, depth, random, dMax, sMin)

if depth == 0
    if nargin < 3 || isempty(random)
        random = false;
    end

    if nargin < 4 || isempty(dMax)
        dMax = Inf;
    end
```

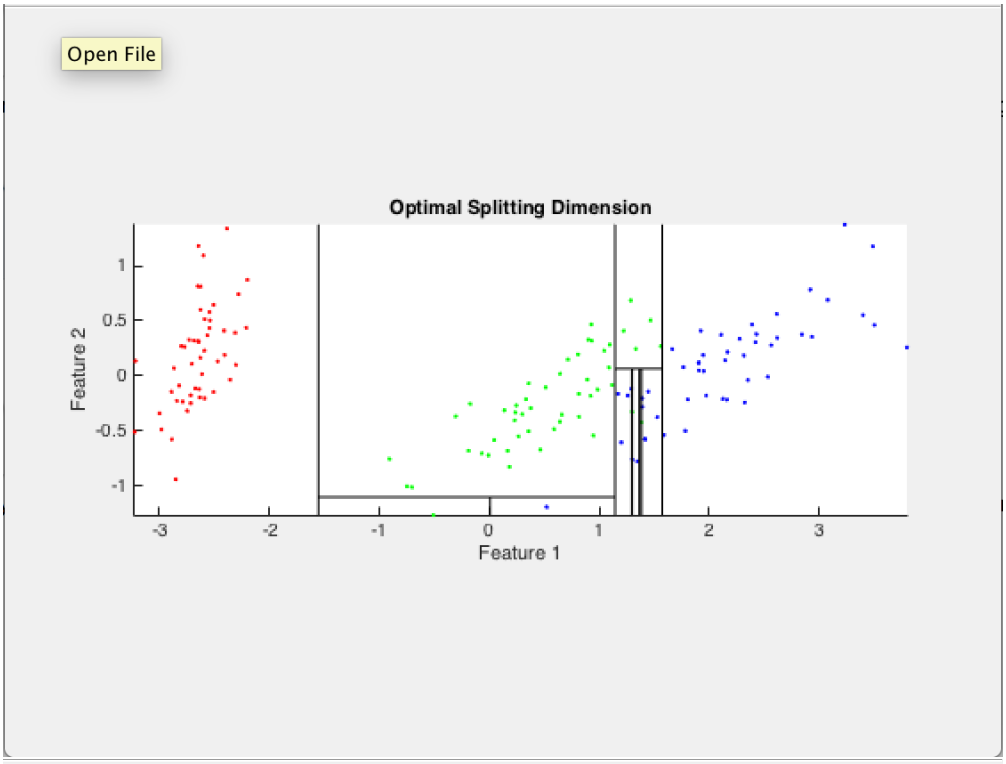
```
    if nargin < 5 || isempty(sMin)
        sMin = 1;
    end
end

tau.d = [];
tau.t = [];
tau.L = [];
tau.R = [];
tau.p = [];

% Your code here
if OkToSplit(S, depth)
    [L, R, dOpt, tau.t] = findSplit(S);

    % Still needs the case where x has two y's
    if (dOpt == -1)
        tau.p = distribution(S);
    else
        tau.d = dOpt;
        tau.L = trainTree(L, depth+1, random, dMax, sMin);
        tau.R = trainTree(R, depth+1, random, dMax, sMin);
    end
else
    tau.p = distribution(S);
end

    function answer = OkToSplit(S, depth)
        % Your code here
        answer = (impurity(S) > 0) && (size(S.X, 1) > sMin) && (depth < dMax);
    end

    function [LOpt, ROpt, dOpt, tOpt] = findSplit(S)
        % Your code here
        iS = impurity(S);
        deltaOpt = -1;

        LOpt = [];
        ROpt = [];
        dOpt = -1;
        tOpt = -1;

        for d = 1:size(S.X, 2)
            dimensionToSplit = d;
            if random
                dimensionToSplit = randi([1 size(S.X, 2)]);
            end
            x = S.X(:, dimensionToSplit);
            list = thresholds(x);
            for l = 1:size(list, 2)
                curCol = S.X(:, dimensionToSplit);

                curL.X = S.X(curCol <= list(l), :);
                curL.y = S.y(curCol <= list(l), :);
                curL.labelMap = S.labelMap;
```

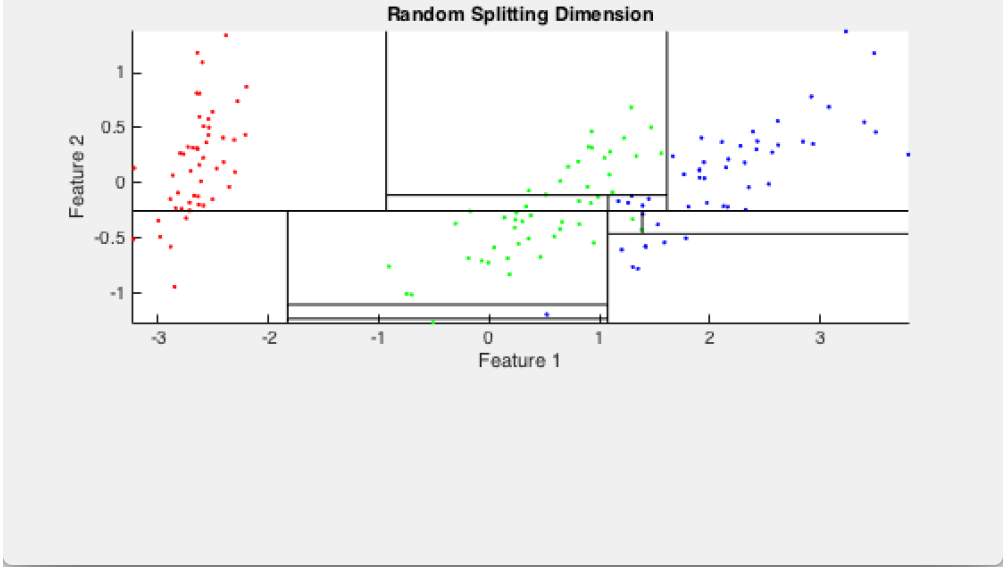
```
                curR.X = S.X(curCol > list(l), :);
                curR.y = S.y(curCol > list(l), :);
                curR.labelMap = S.labelMap;

                delta = iS - (size(curL.X, 1)/size(S.X, 1))*impurity(curL) - (size(curR.X, 1)/size(S.X, 1)
                if delta > deltaOpt
                    deltaOpt = delta;
                    LOpt = curL;
                    ROpt = curR;
                    dOpt = dimensionToSplit;
                    tOpt = list(l);
                end
            end
        end
    end

    function p = distribution(S)
        % Your code here
        p = zeros(size(S.labelMap));
        n = size(S.y, 1);
        for i = 1:n
            p(S.y(i)) = p(S.y(i)) + 1;
        end
        p = p / n;
    end

    function i = impurity(S)
        % Your code here
        % Using the Gini index as opposed to err(S)
        i = 1;
        n = size(S.y, 1);
        for j = 1:size(S.labelMap)
            p = (size(find(S.y == j), 1) / n) ^ 2;
            i = i - p;
        end
    end

    function list = thresholds(x)
        % Your code here
        % This function produces a sorted list of split thresholds to try
        % given a vector x of the values in a single feature dimension
        x = sort(unique(x));
        ud = size(x) - 1;
        list = [];
        for i = 1:ud
            list(i) = (x(i) + x(i + 1)) / 2;
        end
    end
end
```

Problem 2(b)

```
function p = treePartition(tau, box)

xMin = box(1, 1);
```

```
xMax = box(2, 1);
yMin = box(1, 2);
yMax = box(2, 2);

if tau.d == 1
    p = [tau.t, yMin; tau.t, yMax];

    if isempty(tau.L.p)
        leftBox = [xMin, yMin; tau.t, yMax];
        p = cat(3, p, treePartition(tau.L, leftBox));
    end

    if isempty(tau.R.p)
        rightBox = [tau.t, yMin; xMax, yMax];
        p = cat(3, p, treePartition(tau.R, rightBox));
    end
else
    p = [xMin, tau.t; xMax, tau.t];

    if isempty(tau.L.p)
        leftBox = [xMin, yMin; xMax, tau.t];
        p = cat(3, p, treePartition(tau.L, leftBox));
    end

    if isempty(tau.R.p)
        rightBox = [xMin, tau.t; xMax, yMax];
        p = cat(3, p, treePartition(tau.R, rightBox));
    end
end

end
```


Open File
Optimal Splitting Dimension
Feature 2
Feature 1
1
0.5
0
-0.5
-1
-3
-2
-1
0
1
2
3


Random Splitting Dimension
Feature 2
Feature 1
1
0.5
0
-0.5
-1
-3
-2
-1
0
1
2
3

Problem 3(a)

```
function y = treeClassify(x, tau)

if isempty(tau.d)
    indicesOfMax = find(tau.p == max(tau.p));
    y = indicesOfMax(1);
else
    y = treeClassify(x, split(x, tau));
end

    function child = split(x, tau)
        if x(tau.d) <= tau.t
            child = tau.L;
        else
            child = tau.R;
        end
    end
end
```

Problem 3(b)

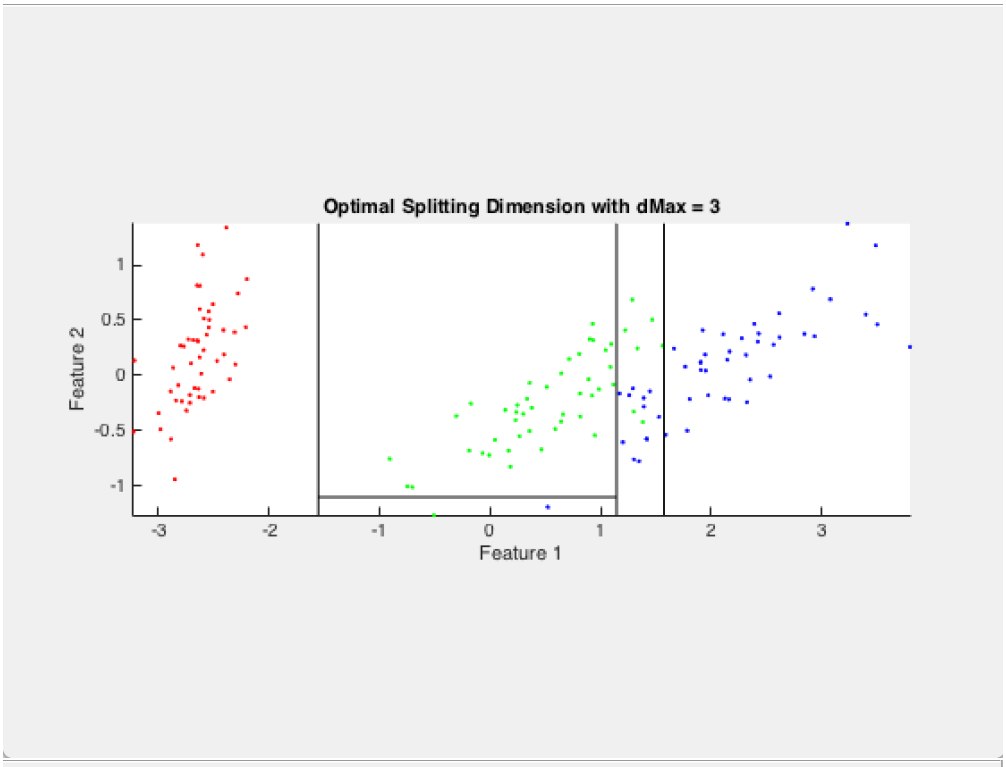
```
function e = err(tau, S)

numWrong = 0;
n = size(S.X, 1);
for i = 1:n
    y = treeClassify(S.X(i, :), tau);
    if y ~= S.y(i)
        numWrong = numWrong + 1;
    end
end

e = numWrong / n;

end
```

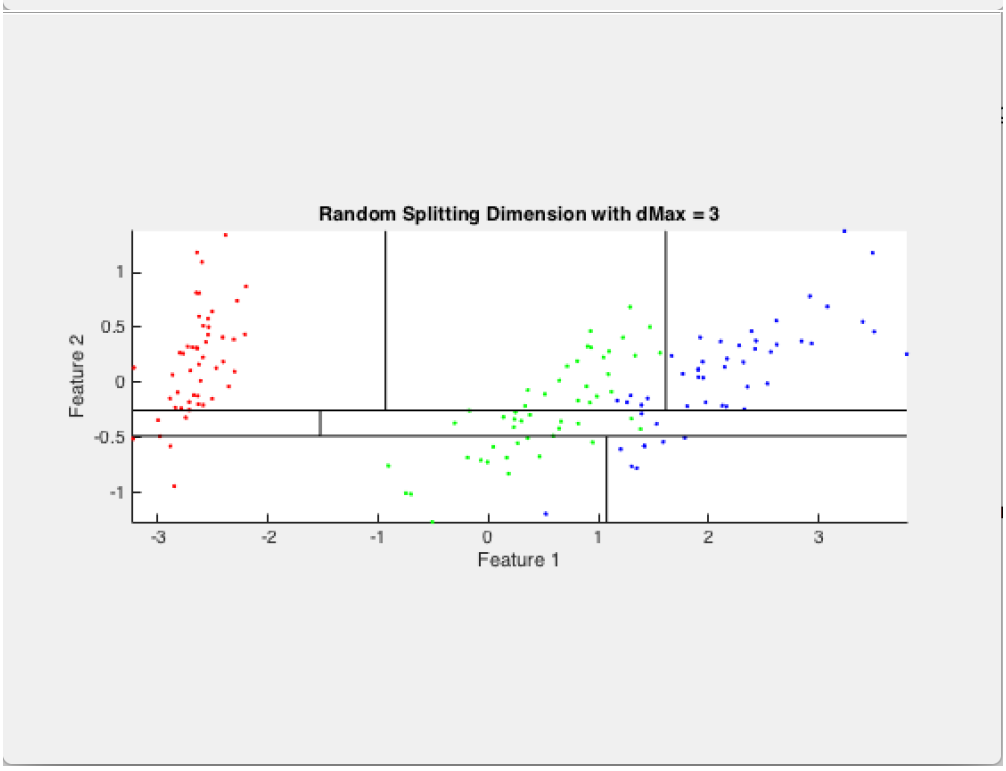
All training errors are 0.

Problem 3(c)

The training errors are 0, which makes sense since we didn't limit the depth of the tree or the minimum number of training samples in a leaf, which means we will keep adding nodes until we have 0 impurity (aka we classify all data in training set completely correctly).

Problem 3(d)


Optimal Splitting Dimension with dMax = 3
Feature 2
Feature 1
1
0.5
0
-0.5
-1
-3
-2
-1
0
1
2
3


Random Splitting Dimension with dMax = 3
Feature 2
Feature 1
1
0.5
0
-0.5
-1
-3
-2
-1
0
1
2
3

The error rates are positive, which makes sense since we are limiting the depth of the tree. This means that the training of our model stops before we perfectly classify all the data in the training set.

Error rates with d-max = 3:

With optimal splitting dimension: 0.0533

With random splitting dimension: 0.0800

The first error rate is lower than the second, since we're picking the optimal splitting dimension each time instead of picking it at random.

Problem 3(e)

```
function e = cverr(S, K)

dMax = 3;
sMin = 1;

N = size(S.X, 1);

% Compute size of each bucket and the range of each bucket to index into
% the random permutation of integers from 1 to N
% this seems convoluted and there's probably a better way to do it
bucketSizes = zeros(1, K) + floor(N/K);
for i = 1:mod(N, K)
    bucketSizes(i) = bucketSizes(i) + 1;
end
separatingIndices = zeros(1, K);
separatingIndices(1) = bucketSizes(1);
for i = 2:K
    separatingIndices(i) = separatingIndices(i - 1) + bucketSizes(i);
end

[~, index] = sort(rand(N, 1));

% build trainingSet from K - 1 partitions then test on the last set
numWrong = 0;
trainingSet.labelMap = S.labelMap;
for k = 1:K
    trainingSet.X = [];
    trainingSet.y = [];
    for j = 1:K
        if k ~= j
            startIndex = separatingIndices(j) - bucketSizes(j) + 1;
            endIndex = separatingIndices(j);
            set = index(startIndex:endIndex);
            trainingSet.X = [trainingSet.X; S.X(set, :)];
            trainingSet.y = [trainingSet.y; S.y(set, :)];
        end
    end
    tree = trainTree(trainingSet, 0, false, dMax, sMin);
    % This code can be eliminated by using err.m and multiplying by size,
    % but not sure if it's worth it
    for x = separatingIndices(k) - bucketSizes(k) + 1:separatingIndices(k)
        y = treeClassify(S.X(index(x), :), tree);
        if y ~= S.y(index(x))
            numWrong = numWrong + 1;
        end
```

```
    end
end

e = numWrong / N;

end
```

Problem 3(f)

[0.0733 0.0800 0.1000 0.1000 0.0800]

The values aren't the same because k-fold cross validation uses k random samples of the data as buckets. This means that the resulting errors will be slightly different simply due to the random sampling process.

Problem 3(g)

[0.0600 0.0600 0.0600 0.0600 0.0600]

The values are the same, which makes sense since leave-one-out cross validation always uses each datum as sample data once (i.e. each random partition is equivalent to another because each bucket only contains one datum).

Problem 3(h)

The cross-validation method is called "leave-one-out cross-validation". This is the extreme version of k-fold classification, where the model is trained on each data point except N times. This gives us a lower variance on the estimated error. However, this is seldom used simply because we repeat the model-training N times, which is computationally very expensive.

Problem 3(i)

```
% Picking which dimensions to use

clearvars;
clc;

load fisheriris
T.X = meas;
[T.y, T.labelMap] = numberize(species);

clc;

cErrors = [];

for DOut = 1:4
        S.X = meas(:, DOut);
        S.y = T.y;
        S.labelMap = T.labelMap;

        cErrors(DOut) = cverr(S,  length(S.y));
end

clc;
```

```
disp(cErrors)
```

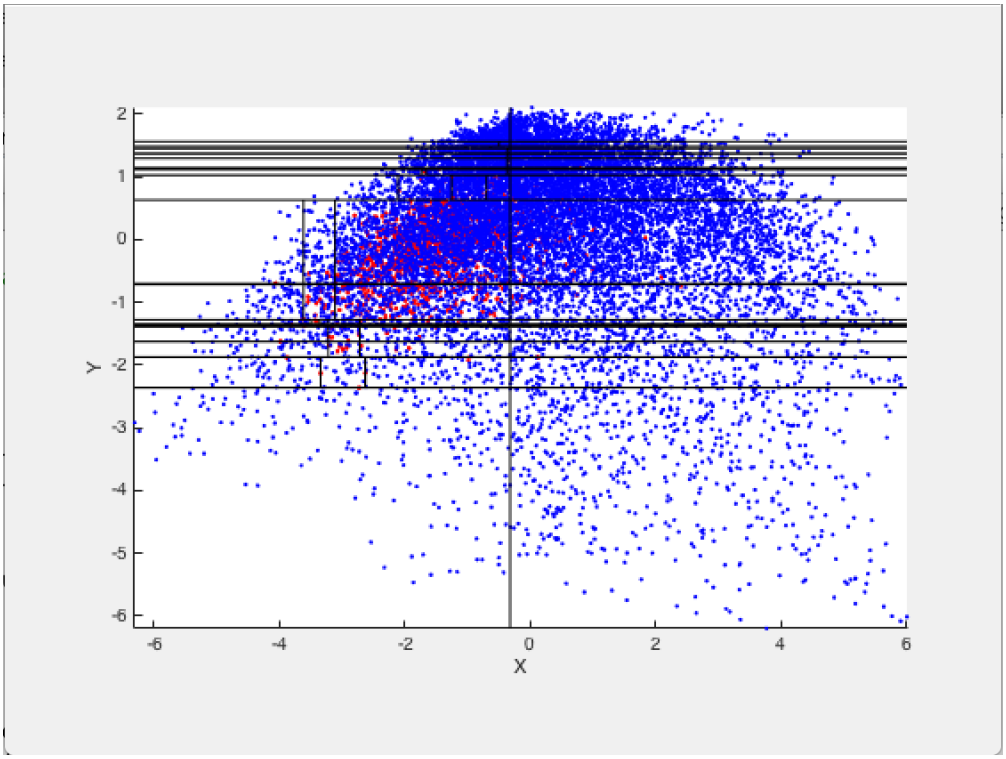
Leave-one-out cross-validation errors: [0.2667 0.4867 0.0600 0.0467]
We would choose the first dimension, since the first dimension lets us get the smallest error rate.

Problem 3(j)

The error rate for the previous classifier is worse than that of using the highest-variance feature from PCA, since using PCA would choose an equal or better axis that explains as much or more of the variance.

Problem 4(a)

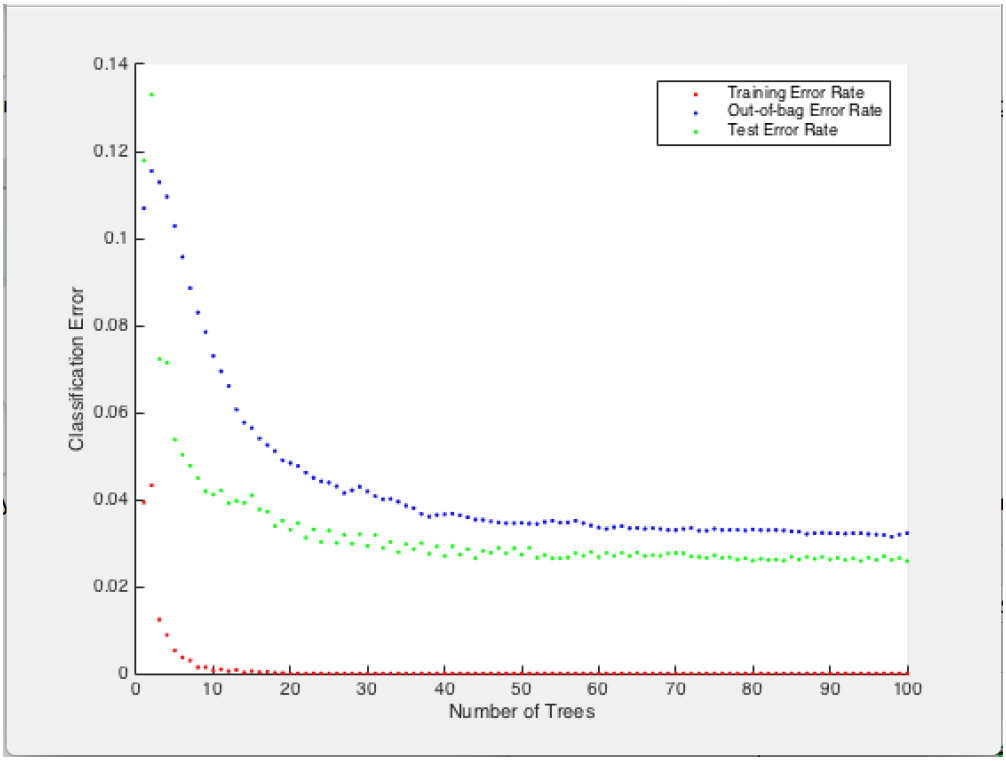
```
function p = ctreePartition(tau, box, index)
% kick off this function with ctreePartition(tau, box, 1)
xMin = box(1, 1);
xMax = box(2, 1);
yMin = box(1, 2);
yMax = box(2, 2);

if isequal(tau.CutPredictor(index), tau.PredictorNames(1))
    p = [tau.CutPoint(index), yMin; tau.CutPoint(index), yMax];

    if tau.Children(index, 1) ~= 0
        leftBox = [xMin, yMin; tau.CutPoint(index), yMax];
        p = cat(3, p, ctreePartition(tau, leftBox, tau.Children(index, 1)));
    end

    if tau.Children(index, 2) ~= 0
        rightBox = [tau.CutPoint(index), yMin; xMax, yMax];
        p = cat(3, p, ctreePartition(tau, rightBox, tau.Children(index, 1)));
    end
else
    p = [xMin, tau.CutPoint(index); xMax, tau.CutPoint(index)];

    if tau.Children(index, 1) ~= 0
        leftBox = [xMin, yMin; xMax, tau.CutPoint(index)];
        p = cat(3, p, ctreePartition(tau, leftBox, tau.Children(index, 1)));
    end

    if tau.Children(index, 2) ~= 0
        rightBox = [xMin, tau.CutPoint(index); xMax, yMax];
        p = cat(3, p, ctreePartition(tau, rightBox, tau.Children(index, 2)));
    end
end

end
```



Problem 4(b)

We don't expect low generalization error rate, since a single tree is a weak classifier. This is because it's only trained on a random subset of the data. This is also empirically supported by our plot, which shows that the single forest misclassifies a large number of the total dataset.

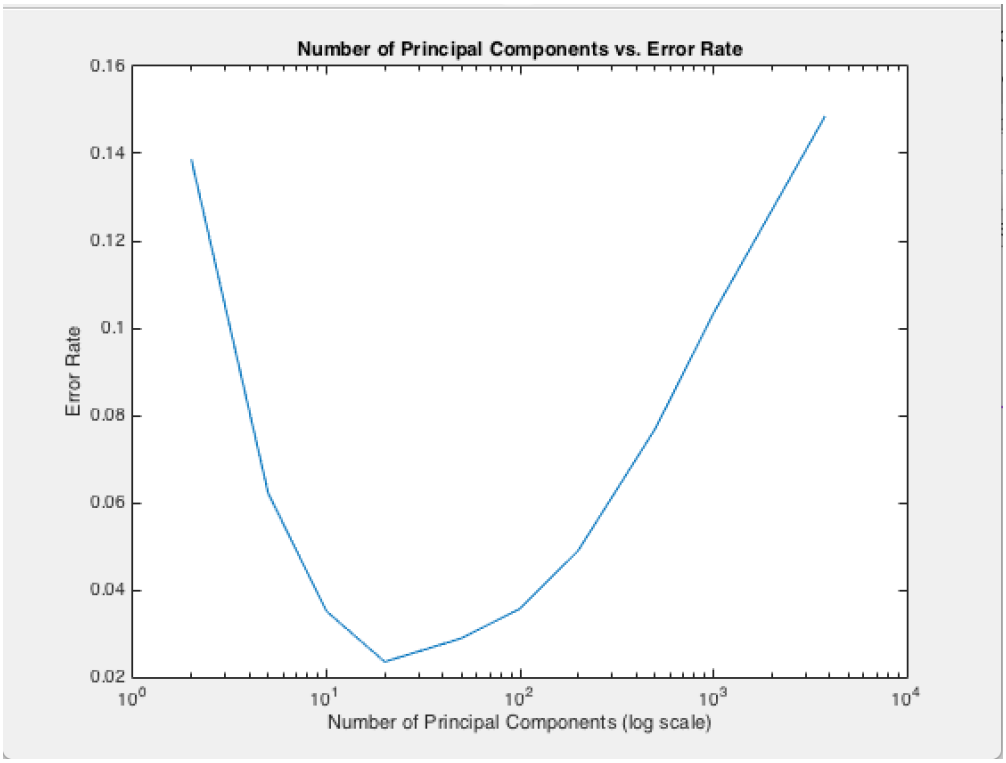
Problem 4(c)



Problem 4(d)

Out-of-bag error rate and test error rate are good estimates of the generalization error, since they measure the error rate of the model on data it hasn't trained on. (Training error drops to zero quite fast, which isn't connected to generalization rate since training error depends on how well the model does on data it's trained on.) The shape of these plots tell us that random forests tend not to overfit (i.e. they generalize quite well), since the generalization error rate doesn't increase as the model complexity increases.

Problem 4(e)



Problem 4(f)

We would use 20 components, since that's the number that gives us the least out-of-bag error rate. PCA generally helps, since we throw away the dimensions that explain the least amount of variance.

However, PCA may also hurt by throwing away variance that helps the model (e.g. by losing spatial information in a certain dimension), for example when the last few dimensions (ranked by variance) are important for classification.

Problem 4(g)

```
[HOGTrain, HOGTest] = readData('INRIA_Person_Database');

[VTrain, muTrain, sigma2Train] = PCA(HOGTrain.X);
forestOptimalComponents = fitensemble(compress(HOGTrain.X, VTrain, muTrain, 20), HOGTrain.y, 'Bag', 100, '
[VTest, muTest, sigma2Test] = PCA(HOGTest.X);
labels = predict(forestOptimalComponents, compress(HOGTest.X, VTest, muTest, 20));

errorRates = zeros(2);
N = size(labels, 1);
for i = 1:N
    if labels(i) == 1 && HOGTest.y(i) == 1
        % True Positive
        errorRates(1, 1) = errorRates(1, 1) + 1;
    elseif labels(i) == 2 && HOGTest.y(i) == 2
        % True Negative
```

```
            errorRates(2, 2) = errorRates(2, 2) + 1;
        elseif labels(i) == 1 && HOGTest.y(i) == 2
            % False Positive
            errorRates(1, 2) = errorRates(1, 2) + 1;
        elseif labels(i) == 2 && HOGTest.y(i) == 1
            % False Negative
            errorRates(2, 1) = errorRates(2, 1) + 1;
        end
    end

    % CORRECTED ERROR RATES FORMULA
    sums(1, 1) = sum(errorRates(:, 1));
    sums(2, 1) = sum(errorRates(:, 2));
    errorRatesPercentage(:, 1) = errorRates(:, 1) ./ sums(1, 1) .* 100;
    errorRatesPercentage(:, 2) = errorRates(:, 2) ./ sums(2, 1) .* 100;
    errorRatesPercentage = round(errorRatesPercentage, 1);
```

Our **counts** are as follows:

$$\begin{array}{ll} TP = 480 & FP = 91 \\ FN = 652 & TN = 4439 \end{array}$$

Thus, our **straight percentages** of TP, FP, FN, and TN are as follows:

$$\begin{array}{ll} TP = 8.5\% & FP = 1.6\% \\ FN = 11.5\% & TN = 78.4\% \end{array}$$

However, the following describes our **rates per definitions of True/False Positive/Negative Rates**.

To compute these values, we referenced this Wikipedia page on Sensitivity and specificity: `https://en.wikipedia.org/wiki/Sensitivity_and_specificity`. The equations are repeated below for convenience:

$$\text{True Positive Rate} = \frac{\sum TruePositive}{\sum ConditionPositive}$$
$$\text{False Negative Rate} = \frac{\sum FalseNegative}{\sum ConditionPositive}$$
$$\text{False Positive Rate} = \frac{\sum FalsePositive}{\sum ConditionNegative}$$
$$\text{True Negative Rate} = \frac{\sum TrueNegative}{\sum ConditionNegative}$$

Where the sum of Condition Positive is equal to the sum of the first column and the sum of Condition Negative is equal to the sum of the second column.

$$\begin{array}{ll} TP = 42.4\% & FP = 2.0\% \\ FN = 57.6\% & TN = 98.0\% \end{array}$$