Please refer to homework 1 and the class mechanics web page for homework policies and formatting/submission instructions. The submission checklist for this assignment is as follows:

`hw4.pdf`, `bags.m`, `trainForest.m`, `forestValue.m`, `oobError.m`

plus any other functions you wrote to run your experiments, as explained in problem 5. Keep in mind that your code must also show up in `hw4.pdf` .

Classification and Regression Forests

The MATLAB files provided with this assignment follow the class notes to implement a recursive version of classification trees. To streamline this assignment, the following main changes are made with respect to the notes:

- In classification problems, all labels are integers starting at 1, for easier compatibility with MATLAB indexing.
- Training and test data are stored into arrays with one data point per *row* rather than per column. For instance, a labeled classification set T with N samples $(\mathbf{x}, y)$ where features $\mathbf{x}$ are d-dimensional is represented by a data structure `T` with an $N \times d$ field `T.X` and an $N \times 1$ field `T.y`. We will also look at regression problems, in which the values are real numbers.[1] Those are stored by rows in `T.y` as well, so `T.y` is a column vector.
- Since we are not interested in classification or regression confidence values in this assignment, the leaves of a classification or regression tree contain the values (discrete labels for classification, real values for regression), rather than their distribution.
- Functions that compute impurity values for training take the set `T.y` of values as input, rather than all of `T` (the features are irrelevant for impurity).
- The top-level function `treeClassify` is renamed `treeValue`, so the same function can be used for both classification and regression.
- All the parameters and hyper-parameters of a tree are encapsulated into a single data structure that is returned by the provided function `treeClassifierDefaults`, which in turn calls `treeCommonDefaults`. This structure is passed to the tree for both training and classification. The call

  ```
  parms = treeClassifierDefaults(L);
  ```

  where `L` is a positive integer (the number of possible label values, which are then $\{1, \ldots, L\}$) returns a structure `parms` with the following fields:

  – `parms.maxDepth` is the maximum allowed tree depth and is set to `Inf` by default.
  – `parms.minSetSize` is the minimum allowed set size in any tree node and is set to 1 by default.
  – `parms.doRandomSplit` is a Boolean flag that specifies whether the split dimension at each node is to be chosen at random (`true`) or optimally. This flag is set to `true` by default.
  – `parms.classifier` is a Boolean flag that is `true` for classifiers and `false` for regressors.
  – `parms.nLabels` is set to `L`.
  – `parms.value` is a function that takes a vector of labels and returns a plurality label in the vector (that is, a label that has the most representatives in the vector).
  – `parms.impurity` is a function that takes a vector of labels and returns the empirical classification error of `parms.value` on that vector.
  – `parms.error` is a function that takes an array (not necessarily a vector) `fx` of computed labels and an equally-sized array `y` of true labels and returns an equally-sized array of error indicators (a `double` number that is 1 where `fx` and `y` differ and 0 where they coincide).
  – `parms.aggregateError` is a function that takes an array (not necessarily a vector) of error indicators as computed by `parms.error` and returns the mean error (the fraction of array values that are equal to 1).

  The first three fields above are set in `treeCommonDefaults` and are used for both classification and regression. Leave these fields unchanged. You may want to study these functions to understand more concretely what they do.

[1] They could also be vectors of real numbers, but we keep things simple here.

Coding Tips

The code provided with this assignment is recursive, and is both simple and slow. Because of this, first test and debug your code on small inputs, and write a single MATLAB script that computes all you need to solve the problems below. Go full size only once everything works.

If you need to run the same code multiple times with different inputs or parameters, it would be very error-prone to replicate code by cutting and pasting. Instead, write a small number of functions that take the necessary inputs and parameters and call them multiple times. Keep in mind that wrong conclusions are not given credit, even if they are reached through correct reasoning from wrong data. So it's in your interest to program in an orderly and modular way.

When you test a classifier on, say, all the pixels in an image, the provided recursive functions can only process one data point at a time, so it's OK to loop over data (including images) when using `treeValue` or `forestValue`. However, it is still bad form (and a waste of your time, and grounds for credit deductions) to use explicit `for` loops on images when more efficient MATLAB constructs are available.

1. As a reference for later problems, you will first use provided code to compute the gradient of two images by convolution with appropriate derivatives of a Gaussian, as we learned in class. The two images are produced by the functions `rings` and `waves`. Specifically, the call

```
[I, gI, qI] = rings(L, n);
```

where both `L` and `n` are positive integers makes an `n`×`n` image `I` that shows some rings. The function used to produce the image is known analytically, so it is possible to compute its gradient by exact calculus. The result is in `gI`, with `gI(:, :, 1)` containing the x derivative and `gI(:, :, 2)` containing the y derivative. Here and elsewhere in this assignment, we revert to the more natural convention of an x axis pointing to the right and a y axis pointing up, so that it is easier to visualize what is going on.

The output image `qI` is a version of `gI` quantized to `L` levels, and you will use this image in later problems. If you are interested in seeing how quantization is done, look at the provided function `quantize`.

You can make a numerical approximation `ngI` of `gI` by using the provided function `gradient`, whose output is in the same format as `gI`.

Run `rings(6, 64)` and make a single, full-page figure with a 4 by 2 array of images (four rows and two columns) in your PDF file. The rows show respectively the two components of `gI`,`ngI`, `abs(ngI - gI)`, and `qI`. Use the MATLAB `colorbar` command to put a color bar next to each of these four images, to make it possible to interpret the values of the gray levels. Make the color bar values legible by setting their font size to 36 points as follows:

```
cb = colorbar; set(cb, 'FontSize', 36);
```

Do *not* show `I` and do *not* show your code. Also, do *not* show the result of running `waves`. We will use that later. Use `imagesc` to display the images and put meaningful captions under each.

2. Write a MATLAB function with header

```
function b = bags(n, k)
```

that takes positive integers `n` and `k` and returns a $n \times k$ matrix `b`. Each column of `b` is a *random bag* out of $A = \{1, \ldots, n\}$, that is, a set of n values drawn out of A with replacement.

Hand in your code (both in the PDF file and as a separate MATLAB file), as well as the array resulting from the call

```
bags(5, 3)
```

Do not use any explicit loops.

3. The function `trainForest` provided with this assignment takes a training set `T`, a desired number `M` of trees, and a parameter structure `parms` as described above. This function currently ignores `M` and just calls `trainTree` to make a forest with a single tree. The resulting forest `phi` has field `phi.tau`, which contains the single tree, and a field `phi.used` that is set to a Boolean column vector with `N` true values, where `N` is the number of samples in `T`. This vector is currently set to all `true` values to signify that all samples in `T` were used to train the tree. This will change in your version.

Rewrite `trainForest` so that it calls `trainTree` repeatedly and returns a forest `phi` with `M` trees in `phi.tau`. Column `j` of the Boolean array `phi.used` should flag the samples in `T` that were used to train `phi.tau(j)`. Hand in your code (both in the PDF file and as a separate MATLAB file). No output yet.

[Hints: The call `unique(b)` where `b` is a column vector returns a unique, sorted list of the items in `b`. Do not delete the statement that prints out a progress message. This will be useful later, as `trainForest` is slow.]

4. The function `forestValue` provided with this assignment just calls `treeValue` once and returns its result. So calling either function returns the same result.

Rewrite `forestValue` so that it returns the value computed by the whole forest `phi`. Hand in your code (both in the PDF file and as a separate MATLAB file). No outputs yet.

5. Let us pretend that we do not know how to compute the gradient of an image. Since we just studied classification trees, it occurs to us that we could at least estimate the values of the gradient quantized to $L = 6$ values. Specifically, we can train two classification forests (one for I_x and one for I_y) with M trees on the image `I` and quantized gradients `qI` you computed earlier, and then use the resulting forest on new images for which we want to compute the gradient.

For training features, we simply take the 5×5 window around each pixel in the image `I` produced by `rings` (with the same arguments used earlier), reshape the 25 values into a 25-dimensional row vector, and use the provided functions `PCA` and `compress` to shorten these features to, say, 8 components.

For training labels, we use the exact labels in `qI`.

Of course, we can only compute features and labels for the central 60×60 part of the 64×64 image, for a total of 3600 training samples.

The provided function

```
function T = features(data, nFeatures, winSize)
```

computes these samples. The first argument is a data structure with fields `data.X` and `data.y`. In the example, these are 64 by 64 images, and `data.y` is either the x or the y component of the true quantized gradient. The value of `nFeatures` is `Inf` if all the pixels in the image are to be used for training. Otherwise, `nFeatures` pixels chosen at random are used.[2] The value of `winSize` is the side length of the square feature window (5 in the example). The function `features` does *not* do PCA. It returns a training set in the format described earlier.

Train two forests (one for I_x and one for I_y) with $M = 10$ trees using all the 3600 valid pixels in images `I` and `qI` from problem 1, and compute six images. The first two images show the true labels. The second two show the labels resulting from running your classifiers on the 64×64 training image produced by `rings`. The third two images show the binary training classification error on each component of the gradient of `I`. In the error images, a pixel is black if there is no error and white if there is.

Hand in your code (both in the PDF file and as a separate MATLAB file) and show the six resulting images in a single figure with two images per row and with appropriate captions. When you list your code in the PDF, make sure it fits on the page, possibly by breaking longer functions over multiple pages.

[Hints: Organize your code so that it is easy to run additional experiments with different training or test data. I chose to write the following functions to keep things clear, even at the cost of repeating some computation:

- `assemble` packages all the data (training and testing), plus fields that help me keep track of what experiment I am running into a single data structure `data`.
- `experiment` takes the structure computed by `assemble` plus the other parameters it needs and runs an entire experiment. It outputs a single structure `result` with all that the function computes. This function also writes images to files with appropriate names, so it is easy to include these into the solution PDF file. Designing appropriate file names take time and care and makes the function somewhat lengthy. However, this is effort well spent, to prevent mistakes when you show your results.

It is likely that you will add to these functions as you solve additional problems below. You may choose to do things differently, but explain clearly what you do as you show your code.]

6. Run the same tests as for the previous problem, but use the files output by `waves` for testing. Training is still done on the images from `rings`. No code, just show the usual six-image figure.

[Hint: You should expect significantly worse results for testing than those you obtained for training.]

7. Quantizing gradient values to six levels is a very drastic measure. A much better solution is to modify random forests to they can do regression in addition to classification. Interestingly, all we need to do is to write a new function `treeRegressorDefaults` to call instead of `treeClassifierDefaults` as discussed next.

Instead of minimizing the misclassification rate, a regressor minimizes the empirical Root Mean Squared Error (RMSE)

$$\overline{err}(f, T) = \sqrt{\frac{1}{N} \sum_{n=1}^{N} \|f(\mathbf{x}_n) - y_n\|^2}$$

where N is the number of training samples in T and the discrete label is now replaced by a real *value* y_n.

Instead of a plurality label, the regressor computes the empirical mean of the training values at a leaf of the tree:

$$m(Y) = \frac{1}{|Y|} \sum_{n=1}^{|Y|} y_n$$

[2] This argument is provided for generality. We will not use it in this assignment, so you can set it to `Inf` all the time.

where Y is the set of values at a leaf. Instead of the misclassification rate, impurity is the deviation from this mean, as measured by the empirical variance

$$\sigma(Y) = \frac{1}{|Y|} \sum_{n=1}^{|Y|} (y_n - m(Y))^2 \; .$$

These changes are implemented for you in the provided function `treeRegressorDefaults`. Note that the function takes no arguments.

If needed, modify any code you wrote for previous problems to also accommodate a regressor. No code in any of the tree or forest computations needs to change, if you wrote things properly (that is, if you used the fields of `parms` whenever possible and appropriate).

Then, train two random forests with $M = 10$ trees as regressors for the two components of the gradient. Use the same features as above, but now use as values the two components of `gI` rather than the labels `qI`.

There is no code to submit for this problem. Instead, compute six images. As before, the first two show the true values for each component of the gradient of `I`. The second two show the regression values resulting from running your regressors on the training image itself for each component of the gradient of `I`. The third two images show the regression error on each component of the gradient of `I`. Show the six resulting images in a single figure with appropriate captions. Each image should come with a color bar with font size 36, as explained earlier.

8. Repeat the previous problem using the images from `rings` for training and those from `waves` for testing. Only hand in the six-image figure. Each image should come with a color bar with font size 36, as explained earlier.

[Hint: Again, test results are worse than results on the training image.]

9. Write a MATLAB function with header

```
function e = oobError(phi, T, parms)
```

that computes the out-of-bag error estimate for a random forest `phi` trained on set `T` and with tree parameters `parms`. Your function should use the appropriate fields of the `parms` data structure returned by either `treeClassifierDefaults` or `treeRegressorDefaults` to compute error and aggregate error.

Hand in your code (both in the PDF file and as a separate MATLAB file). Also show a table with four rows and two columns (plus column and row headers) that lists the errors in your regression experiments (the one without label quantization). Specifically, for the x and y components of the gradient, the four rows should show (i) the aggregate numerical RMSE error deriving from computing the gradient with the provided `gradient` function; (ii) the aggregate empirical RMSE training error; (iii) the aggregate out-of-bag RMSE error estimate; (iv) the aggregate empirical test error. All errors are computed against the true gradient values produced by `waves` (for the testing errors) or `rings` (for all other errors). Make sure that the functions `parms.error` and `parms.aggregateError` are invoked for these calculations. The format of the table is as follows:

	$\partial I/\partial x$	$\partial I/\partial y$
Numerical		
Training		
Out-of-Bag		
Testing		

10. How do the values in the Training and Out-of-Bag rows in the previous table tell you whether your regressor generalizes well or overfits? Briefly explain the conclusion you draw from these values.

11. Compare the Out-of-Bag and Testing rows in the table above. How can you explain the results of this comparison?

12. Explain how you can use the field `phi.used` of a random forest to verify that each tree is trained on about 63 percent of all training samples. Show the result of doing so on any one of your forests (it does not matter which one you use).