COMPSCI 527 — Homework 3

Due on October 8, 2015

Work on this assignment either alone or in pairs. You may work with different partners on different assignments, but you can only have up to one partner for any one assignment. You may not talk about this assignment with others until all of you have handed in their work. See Mechanics→Homework on the class web page for details on the homework policy.

Hand in your work as explained in the instructions for homework 1 (of course, change `hw1` to `hw3`).

The first three problems ask for somewhat inefficient but simple code written **from scratch** (except for `pca`), both so you understand the details and because we want to explore different ways to randomize trees in a random forest, while the MATLAB implementation envisions bagging only. The last problem lets you use the MATLAB Statistics toolbox, which is implemented more efficiently and has several facilities we can use in our exploration.

1. If you type

```
load fisheriris
```

at the MATLAB prompt, the two variables `meas` and `species` are loaded into your workspace. The feature array `meas` is $N \times D = 150 \times 4$ and the label cell array `species` is $N \times 1 = 150 \times 1$. This is our training set for this and the subsequent two problem. Let us make a data structure with this set as follows:

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

where the function `numberize` is provided with this assignment. It takes a cell array of strings and converts it to integers according to the map `T.labelMap` it returns. The integers start at 1, and value ℓ in `T.y` corresponds to string `T.labelMap{`ℓ`}`. So the `labelMap` shows the meaning of the numerical labels.

Data arrays in the MATLAB Statistics toolbox store features as rows rather than columns. This may require some care when translating concepts from the class notes—where data are in columns—to MATLAB code. Even when we do not use the Statistics toolbox, we use its conventions in this assignment, to make it easier to write consistent code.

The features in the training set `T` are four-dimensional, $D = 4$. While this is a small dimensionality, we want to reduce it even further, to $D = 2$, so we can plot and draw things. We perform this reduction by Principal Component Analysis (PCA) using the function `pca` from the MATLAB Statistics toolbox. The somewhat awkward syntax of this function is encapsulated for you in the function `PCA` (uppercase) provided with this assignment. If you say

```
[V, mu, sigma2] = PCA(T.X, k)
```

with $k \leq D$, then the output transformation matrix V is $D \times k$, and corresponds to the transpose

$$V = U^T$$

of the matrix U described in the class notes on the PCA. This is because MATLAB features are rows rather than columns of `T.X`, so you can multiply data by V itself from the right rather than having to transpose U. If you want the full, $D \times D$ transformation matrix, you can use any of the following statements:

```
[V, mu, sigma2] = PCA(T.X)
[V, mu, sigma2] = PCA(T.X, Inf)
[V, mu, sigma2] = PCA(T.X, size(T.X, 2))
```

The $1 \times D$ row vector `mu` is the mean of the rows of `T.X`. The vector `sigma2` contains D entries. These entries are the variances (variances, not standard deviations!) of the diagonal empirical covariance matrix of the data centered with `mu` and transformed by `V`.

(a) Write a function with header

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

that takes an $N \times D$ feature matrix `X`, a PCA transformation matrix `V` and row vector `mu` as computed by `PCA`, and an optional integer scalar argument `k`. The function `compress` returns a $N \times D_{\text{out}}$ matrix `Y` with the principal components of the data in `X`. If `k` is unspecified or is at least equal to the number h of columns in `V`, then the dimensionality D_{out} of the output data is equal to h. Otherwise, D_{out} is equal to `k`.

Show your code for `compress` only. To include the code in your LaTeX file use the command

```
\VerbatimInput[fontsize=\small,fontshape=n,xleftmargin=15mm]{compress.m}
```

and make sure your lines of code are short enough to fit on the page. Use a similar command for other functions in this assignment.

Also **make a figure** that shows the data in the `fisheriris` training set `T` reduced in dimensionality as follows:

```
[V, mu, sigma2] = PCA(T.X);
T.X = compress(T.X, V, mu, 2);
```

To display the data, plot each point `T.X(n, :)` as a red, green, or blue dot depending on whether `T.y(n)` is 1, 2, or 3. It is more efficient to loop over the three label values rather than over the points (recall that `plot` can take vector inputs). Once the plot is complete, scale and tighten the axes properly as follows:

```
axis equal
axis tight
```

(b) What fraction of the variance in the input data does the PCA capture? Explain your reasoning and show your calculation.

2. In this exercise you will grow (train) a classification tree from scratch with code written in recursive form. While you should follow the class notes closely as far as code structure, some functions will have more arguments, in order to try out various options. Also, MATLAB does not allow question marks in function names, so your `split?` function will be called `OkToSplit` (admittedly not as nice). Also check that impurity is positive in `OkToSplit`. What the functions below are supposed to do is explained in the class notes.

(a) Write `trainTree` in recursive form following the notes and without using functions from the MATLAB Statistics toolbox. However, use header

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

The meaning of the arguments is as follows (the term "optional" below means that the caller need not specify the argument, but your function still needs to handle the corresponding parameter properly):

- `S` is a training set of the same format as the set `T` used in the previous exercise.
- `depth` is the depth of the node currently being built (zero for the root).
- `random` (optional) is `true` if the dimension on which to split at every node is to be chosen at random. Default is `false`.
- `dMax` (optional) is the maximum depth of the tree. Default is `Inf`.
- `sMin` (optional) is the minimum number of training samples in a leaf. Default is 1.

A skeleton for `trainTree` that also shows how to handle optional arguments is provided with this assignment. If the function calls itself internally, it needs to specify all arguments, so they are passed properly to other invocations. All helper functions should be included in the same file (see skeleton). It is OK to use `histcounts` if you like, but, if you do, be careful with its arguments. Keep it simple! In particular, the tree data structure should be as described in the notes. Just **show your code** for now.

PROGRAMMING NOTE. If the training set contains two samples $(\mathbf{x}_1, y_1)$ and $(\mathbf{x}_2, y_2)$ with $\mathbf{x}_1 == \mathbf{x}_2$ but $y_1 \neq y_2$, then it may be impossible to split the set any further. Logically, you would have to check for this occurrence in `OkToSplit`, but it is more efficient to keep this function simple. Instead, allow for the possibility that `findSplit` returns nothing, and stop splitting in that case.

(b) When the training features are two-dimensional, we can draw line segments on the feature plane that show how a particular classification tree partitions the plane. Write a function with header

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

that takes a tree grown by `trainTree` on a training set with two-dimensional features and a bounding box, and returns an array of line segments. Specifically, if you plot the features as in exercise 1(a), type

```
box = [get(gca, 'XLim'); get(gca, 'YLim')]';
```

(after the `axis tight` command) to create the initial bounding box. The variable `box` will be an array of the following form:

$$\left[\begin{array}{cc} x_{\min} & y_{\min} \\ x_{\max} & y_{\max} \end{array}\right]$$

that describes a rectangle that bounds all the features (the y symbol here denotes the second component of a 2D feature, not a label). If J segments are needed to describe the partition, then the output `p` is a $2 \times 2 \times J$ array, where `p(:, :, j)` has the form

$$\left[\begin{array}{cc} x_a & y_a \\ x_b & y_b \end{array}\right]$$

where the two points (x_a, y_a) and (x_b, y_b) are the endpoints of segment `j`. You can plot all segments in black with the single command

```
plot(squeeze(p(:, 1, :)), squeeze(p(:, 2, :)), 'k');
```

Show your code for `treePartition` and **two figures** with the resulting partitions, each superimposed on the plot you made in exercise 1(a), for two different trees trained with the following statements:

```
tau1 = trainTree(T, 0, false);
tau2 = trainTree(T, 0, true);
```

(so you use default values for `dMax` and `sMin`). Here, `T` is the two-dimensional `fisheriris` training set from before. Put intelligible titles or captions on your figures so it is clear which is which. Something like *Optimal Splitting Dimension*, not just `tau1`. [Hint: Write a recursive function.]

3. We now write a classifier that uses a tree grown with `trainTree`.

(a) Write a function with header

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

following the class notes, and **show your code**. The function returns the label `y` that classification tree `tau` assigns to feature `x`.

HINT: To check if the current node is a leaf you can simply say

```
if isempty(tau.d)
```

(b) Write a function with header

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

that takes a tree `tau` grown with `trainTree` and a training set `S` and computes the empirical error on `S` of the classification tree `tau`. It is OK to loop over training samples for this question. **Show your code** for `err` and **report the training errors** for the trees `tau1` and `tau2` you developed earlier.

(c) Explain briefly why the empirical errors you found in your previous answer make sense.

(d) **Show two figures** with the partitions (superimposed on the training data plot) for trees

```
tau3 = trainTree(T, 0, false, 3);
tau4 = trainTree(T, 0, true, 3);
```

and **report the corresponding training error rates**. **Explain briefly** why these two error rates are positive and why the relation between them makes sense.

(e) Write a function with header

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

that computes the K-fold cross-validation error rate on training set `S` without using functions from the MATLAB Statistics and Machine Learning toolbox. Your trees should use $d_{\text{max}} = 3$, $s_{\text{min}} = 1$, and optimal split dimensions. Make sure that the folds are as equal in size as possible, but the samples in each of them are chosen randomly out of `S`. **Show your code.**

HINT: If you need a random permutation of the integers from 1 to N you can say

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

(f) Run the following command 5 times on the `fisheriris` training set `T` compressed to two-dimensional features, and report the values you obtain:

```
cve = cverr(T, 4);
```

Why are they not the same?

(g) Do the same with

```
looe = cverr(T, length(T.y));
```

HINT: You may want to write a `for` loop, start execution, and go for coffee.

(h) What is the name of the cross-validation method you used in your answer to the previous question? In what way is it better than 4-fold cross-validation? Why do people not use the method in the previous question all the time?

(i) In the problems above, we reduced the dimensionality of the features in training set `T` by PCA. Another way to reduce the dimensionality from D to D_{out} is to pick D_{out} of the D features, that is, pick D_{out} of the D columns of `T.X`. The problem of which D_{out} columns to choose is called *feature selection*. Solve this problem by leave-one-out cross-validation for the full `fisheriris` training set with $D = 4$ and with $D_{\text{out}} = 1$. (So you are choosing the single most informative feature out of four). **Show your code** for doing this, **report all your cross-validation errors**, and **state what column you would choose.**

(j) How does the leave-one-out cross-validation error for the classifier based on the feature you chose in your previous answer compare with that for a classifier that uses the highest-variance feature computed by PCA? **Explain.** Do **not** show your code for this question, just give your answer and explanation.

4. We now explore random forests and HOG features with the tools available in the MATLAB Computer Vision System and Statistics and Machine Learning toolboxes. These toolboxes are alway somewhat in flux. I developed this assignment and my own sample solution with version R2015a of MATLAB. Older versions may miss some of the functionality or have slightly different syntax. So please use version R2015a or later, or modify your call syntax as needed.

Towards the bottom of the web page `http://pascal.inrialpes.fr/data/human/`, in the paragraph just above **Disclaimer**, you will find a link to the INRIA Person Database that provides separate training and test data for pedestrian detectors. Download that set and install it in some directory whose name (either the absolute name or the name relative to your MATLAB working directory for this assignment) you place into the MATLAB string variable `dataDir`. In your data directory you will then have a subdirectory called `INRIAPerson`, which is what the `.tar` file you downloaded expands to. Do not change that subdirectory name or anything inside it.

The function `readData` provided with this assignment is called as follows:

```
[HOGTrain, HOGTest] = readData(dataDir);
```

This function also has an optional argument to specify the size of the HOG window, but you will not need to set this argument for this assignment—just use the default value. Running this function takes a while because it computes HOG descriptors to build a training set and a test set from many images. For the training set, it reads 2416 small pedestrian images and 1218 larger images that do not contain pedestrians. It then extracts ten random windows from each of the larger images, to use as negative samples. Finally, it computes HOG descriptors for all of the 2,416 + 12,180 = 14,596 resulting images and places them in the `HOGTrain` structure with appropriate labels and label map `{'+', '-'}`. To compute HOG descriptors, `readData` calls `imgHOG`, also provided with this assignment, which in turn calls the `extractHOGFeatures` function from the Computer Vision System toolbox. This entire procedure is then repeated for the test data, which contains 5,656 samples.

You may want to look at some of the images to get a feel for what they look like. If you want to see the HOG descriptors for image `img`, you can call `extractHOGFeatures` with two output arguments. The first is the HOG descriptor, and the second is an image that visualizes the descriptor. You can view the second output with the MATLAB function `plot`.

To grow a random forest with `nTrees` trees on training set `Train`, you say

```
forest = fitensemble(Train.X, Train.y, 'Bag', nTrees, 'tree', 'Type', 'classification');
```

where `fitensemble` is part of the MATLAB Statistics and Machine Learning toolbox. This function does only bagging, and provides no way to also randomize the split dimension.

(a) Read the MATLAB help for the `ClassificationTree` class to figure out how MATLAB represents trees. Focus in particular on the following properties of the class:

- `Children`, which corresponds to $\tau.L$ and $\tau.R$ in the class notes;
- `CutPoint`, which corresponds to $\tau.t$;
- `CutPredictor`, which corresponds to $\tau.d$;
- `PredictorNames`, which corresponds to a training set's `labelMap`.

Once you understand the representation, copy your `treePartition` function from problem 2 to a file `ctreePartition.m`, and modify the latter to make it work on MATLAB classification trees. Test your code on the first tree of a 100-tree forest trained on a version of the `HOGTrain` set compressed by PCA (use your `compress` function) to 2-dimensional features. Once you have the forest in variable `forest`, you can retrieve the first tree with the call

```
t1 = forest.Trained{1};
```

Show both your code for the function `ctreePartition` only **and the figure** obtained by overlaying the partition on top of color-coded data points, as you did in problem 2. Draw positive points in blue and negative ones in red, and remember to issue the commands

```
axis equal
axis tight
```

before you compute the partition, so you use the correct bounding box as explained in problem 2. Use an entire page for the picture and rotate it 90 degrees as follows:

```
\begin{sidewaysfigure}
    \centerline{\includegraphics[width=\textwidth]{<your file name here>}}
    \caption{Partition induced by the first tree in the forest.}
    \label{fig:large_partition}
\end{sidewaysfigure}
```

The `sidewaysfigure` environment is defined in the LaTeX `rotating` package, which is loaded for you in the `.tex` template file.

(b) Do you expect a low generalization error rate from the classification tree `t1` you computed in part (a) of this problem? Explain briefly.

(c) The MATLAB Statistics and Machine Learning toolbox provides functions `loss` and `oobloss` that compute the error rate and out-of-bag error rate for a classifier. Read the help pages for these functions[1], then use these functions with the option `'mode'` set to `'cumulative'` to create a plot of the following three quantities as functions of the number of trees used in the forest trained on the full training set `HOGTrain` (without dimensionality reduction):

1. Training error rate
2. Out-of-bag error rate
3. Test error rate

[1]The function name `loss` is overloaded. You want the help page for the version for classification and regression trees.

You only need a single forest with `nTrees` equal to 100 to create these plots. Turn in a single diagram with the three plots in different colors. Label the abscissa "number of trees" and the ordinate "classification error" and add a legend (look up the help for the `legend` command) that shows which curve is which. **Just show your figure, not your code.**

(d) Which plot(s) in your previous figure are good estimates of the generalization error? Why? What does the shape of these plots tell us about the tendency of random forests to overfit?

(e) Let us now fix the number of trees in the forest to 100, and let us examine the effects of compressing the HOG features with PCA. Let

```
nPCA = [3780 1000 500 200 100 50 20 10 5 2];
```

be a vector of feature dimensions to consider. For each element in `nPCA`, compute the out-of-bag error rate with the function `oobLoss` and without specifying any option, so you get a single rate for the entire forest rather than a cumulative plot. Use the MATLAB function `semilogx` to plot the error rate versus the number of principal components with the `x` axis on a logarithmic scale. Your diagram should have appropriate labels on the axes. **Give only a figure, not your code.** [Hint: Compute a full PCA just once, and then use `compress` several times.]

(f) How many components would you use? Can you guess why PCA helps or hurts? Explain briefly.

(g) For the 100-tree random forest you just examined and an optimal number of principal components, make a 2×2 table that shows true positive, true negative, false positive, and false negative *testing* error rates. Express these values with percent figures, rounded to one digit after the decimal period. The MATLAB function `predict` can be used to classify a dataset with a given forest. The `.tex` file provided with this assignment has a template table for you. **Show your code and your table.**

HINT: To say that a test sample yields a "true positive" means "this classification result is *positive* in that the classifier said that it is a pedestrian, and it is *true* in that the annotation in the test set said so as well." Similarly for the other three error rates.