Randomised Decision Forest Laboratory

Tae-Kyun Kim and Mang Shao
Imperial College London
June 2014

In this lab session, we learn the machine learning algorithm called randomised decision forest (RF) by Matlab. Following the lecture on RF, we implement the algorithm, see through the core parts of the algorithm step-by-step, while training and testing it on the toy data sets and Caltech101 image categorisation data set. Try also different parameter values of RF and see their effects. Although the lab session is limited due to the given time and efficiency of the Matlab codes provided, for a real-scale problem, ponder about why RF is so important in computer vision and machine learning and in comparison to other alternative methods, in terms of scalability, efficiency, multi-classes, and generalisation. For its various up-to-date application work, please have a look at http://www.iis.ee.ic.ac.uk/icvl/publication.htm.

1. Getting Started

Start Matlab, go to the directory where the provided files are. Open the main_guideline.m then copy and paste lines of code to the command window of Matlab and press Enter to execute them, step by step, as guided below.

Do initialisation.
```
>> init;
```

Set the random forest parameters.
```
>> param.num = 10; % Number of trees
>> param.depth = 5; % trees depth
>> param.splitNum = 3; % Number of trials in split function
>> param.split = 'IG'; % Currently support 'information gain' only
```

2. Loading datasets

We provide three toy data sets, each of which is a set of 2D points with their class labels, and the Caltech101 image categorisation data set to experience a real vision problem. Start with the Toy Spiral data set.

Select a dataset.
```
>> [data_train, data_test] = getData('Toy_Spiral'); % {'Toy_Gaussian',
'Toy_Spiral', 'Toy_Circle', 'Caltech'}
```

Check the training and testing data loaded. The data formats are

```
data_train(:,1:2) : [num_data x dim] Training 2D vectors,
data_train(:,3) : [num_data x 1] Label of training data, {1,2,3}.
```

Plot the training data.
```
>> plot_toydata(data_train);
```

The test data formats are

```
data_test(:,1:2) : [num_data x dim] Testing 2D vectors, 2D points
in the uniform dense grid within the range of [-1.5, 1.5],
data_train(:,3) : N/A.
```

Plot the testing data.

```
>> scatter(data_test(:,1),data_test(:,2),'.b');
```

3. Training Random Decision Forest

We learn the random forest using the training data above. Given a single set of data points, we first generate multiple subsets by Bagging (Boostrap Aggregating). Each of the bagged data sets is formed by randomly subsampling the original data set. Then, we grow a tree per each subset, by recursively splitting the tree nodes using the information gain. The tree growth stops when it reaches a predefined depth or meets stopping criterions.

3.1. Generating data subsets randomly

We do bagging to create subsets of training data.

```
>> [N,D] = size(data_train);
>> frac = 1 - 1/exp(1); % Bootstrap sampling fraction: 1 - 1/e (63.2%)
>> [labels,~] = unique(data_train(:,end));
```

We generate e.g. 4 subsets and plot them here. Observe that they are as a subset of the original data set, and are somehow different from each other as they are randomly drawn.

```
>> for T = 1:4
>>    idx = randsample(N,ceil(N*frac),1); % A new training set for each trees
is generated by random sampling from dataset WITH replacement.
>>    prior = histc(data_train(idx,end),labels)/length(idx);
>>    subplot(2,2,T);
>>    plot_toydata(data_train(idx,:));
>> end
```

3.2. Splitting nodes

Now we take a data subset generated above, and grow a tree by splitting the data. Here we grow the first tree, out of (param.num) trees.

```
>> T=1; % the trees number 1
```

In order to split nodes, we measure the entropy of the class distributions using the function below.

```
function H = getE(X) % Entropy

   cdist= histc(X(:,1:end), unique(X(:,end))) + 1;
   cdist= cdist/sum(cdist);
   cdist= cdist .* log(cdist);
   H = -sum(cdist);

end
```

Split the first (root) node. We try a few of split functions and thresholds randomly chosen, decide the best ones in terms of information gain, and save them in the node.

```
>> ig_best = -inf;
>> for n = 1:3
>>     dim = randi(D-1); % Pick one random dimension as a split function
>>     d_min = single(min(data_train(idx,dim))); % Find the data range of this
dimension
>>     d_max = single(max(data_train(idx,dim)));
>>     t = d_min + rand*((d_max-d_min)); % Pick a random value within the
range as threshold
>>
>>      idx_ = data_train(idx,dim) < t; % Split data with this dimension and
threshold
>>
>>     % calculate information gain
>>
>>     L = data_train(idx_,dim);
>>     R = data_train(~idx_,dim);
>>     H = getE(data_train(:,dim)); % Calculate entropy
>>     HL = getE(L);
>>     HR = getE(R);
>>     ig = H - sum(idx_)/length(idx_)*HL - sum(~idx_)/length(idx_)*HR
>>
>>     if ig_best < ig
>>         ig_best = ig; % maximu information gain saved
>>         t_best = t; % the best threhold to save
>>         dim_best = dim; % the best split function (dimension) to save
>>         idx_best = idx_;
>>     end
>>
>>     % Visualise the split function and its information gain
>>     visualise_splitfunc(idx_,data_train(idx,:),dim,t,ig,0);
>>     drawnow;
>>     pause; % please hit any key to continue
>> end
```

We visualise the best split function saved.

```
>> visualise_splitfunc(idx_best,data_train(idx,:),dim_best,t_best,ig_best,0);
```

Initialise the base node.

```
>> trees(T).node(1) = struct('idx',idx,'t',nan,'dim',-1,'prob',[]);
```

We now split the nodes recursively in the same way above to further grow a tree. Split the nodes recursively.

```
>> for n = 1:2^(param.depth-1)-1
>>     [trees(T).node(n),trees(T).node(n*2),trees(T).node(n*2+1)] =
>>splitNode(data_train,trees(T).node(n),param);
>> end
```

The tree growth is done, we save the class distributions in the leaf nodes.

```
>> makeLeaf;
```

Visualise the class distributions of the first 9 leaf nodes for example. Observe that most leaf nodes have a dominating class, and most data points belonging to the class, thus exhibiting peaky class distributions. The

information gain, which we used to split nodes above, is maximised when we have peak distributions in both left and right child nodes. As we further split nodes, class distributions get more peaky, containing data of only one class. Yet some leaf nodes may show more uniform class distributions, where there are similar numbers of data points of more than two classes. They are uncertain leaf nodes.

```
>> visualise_leaf;
```

Similarly, we grow multiple trees, a tree per a subset of data.

```
>> trees = growTrees(data_train,param);
```

4. Evaluating Test Data by the Random Decision Forest

Using the random forest we trained above, we evaluate novel data points in the test dataset. Let us grab the few data points and evaluate them one by one by the leant RF. We pass down a test data point the trees, then average the class distributions of the leaf nodes which the data point arrives at. The class which has the maximum probability in the averaged class distribution is assigned to the data point.

```
>> test_point = [-.5 -.7; .4 .3; -.7 .4; .5 -.5];
>>
>> figure(1)
>> plot_toydata(data_train);
>> plot(test_point(:,1), test_point(:,2), 's', 'MarkerSize',20,
'MarkerFaceColor', [.9 .9 .9], 'MarkerEdgeColor','k');
>>
>> for n=1:4
>>    figure(2)
>>    subplot(1,2,1)
>>    plot_toydata(data_train);
>>    subplot(1,2,2)
>>    leaves = testTrees([test_point(n,:) 0],trees);
>>
>>    % average the class distributions of leaf nodes of all trees
>>    p_rf = trees(1).prob(leaves,:);
>>    p_rf_sum = sum(p_rf)/length(trees);
>>
>>    % visualise the class distributions of the leaf noes which the data
>>    % point arrives at
>>    for L = 1:size(leaves,2)
>>        subplot(3,5,L); bar(p_rf(L,:)); axis([0.5 3.5 0 1]);
>>    end
>>    subplot(3,5,L+3); bar(p_rf_sum); axis([0.5 3.5 0 1]);
>>
>>    figure(1);
>>    hold on;
>>    plot(test_point(n,1), test_point(n,2), 's', 'MarkerSize',20,
>>'MarkerFaceColor', p_rf_sum, 'MarkerEdgeColor','k');
>>    pause;
>>end
>>hold off;
>>close all;
```

We evaluate all the data points in the dense 2D grid, and visualise their classification results, color encoded. Compare the result of RF with that of Support Vector Machine (SVM), a standard nonlinear classification method.

```
>> leaves = testTrees_fast(data_test,trees);
>>
>> for T = 1:length(trees)
>>    p_rf_all(:,:,T) = trees(1).prob(leaves(:,T),:);
>> end
>>
>> p_rf_all = squeeze(sum(p_rf_all,3))/length(trees);
```

Do SVM for comparison.

```
>> disp('Training SVM...');
>> model_svm = svmtrain(data_train(:,end),data_train(:,1:end-1),'-t 2 -b 1 -c
>> 10');
>>
>> disp('Testing SVM...');
>> [p_svm, accuracy_svm, p_svm_prob] = svmpredict(data_test(:,end),
>>data_test(:,1:end-1), model_svm, '-b 1');
```

Visualise the classification results of both RF (left) and SVM (right).

```
>> visualise(data_train,p_rf_all,p_svm_prob);
```

5. Trying different parameter values of RF and see the effects

RF has a few of important parameters, which need to be set carefully to achieve the good results. Here we try changing the number of trees and the depth of trees. Try the different parameter values and observe the differences. Using more trees makes the decision boundary smooth, leading to better generalisation. However, it increases computation time and memory usage. The depth of trees controls data fitting. Too small trees lead to underfitting, too deep trees to overfitting, both of which need to be avoided.

Set the random forest parameters. We change the number of trees.

```
>> for N = [1,3,5,10,20]; % Number of trees, try {1,3,5,10, or 20}
>>    init;
>>    param.num = N;
>>    param.depth = 5; % trees depth
>>    param.splitNum = 3; % Number of trials in split function
>>    param.split = 'IG'; % Currently support 'information gain' only
>>
>>    % Select dataset
>>    [data_train, data_test] = getData('Toy_Spiral'); % {'Toy_Gaussian',
>>'Toy_Spiral', 'Toy_Circle', 'Caltech'}
>>
>>    % Train Random Forest
>>    trees = growTrees(data_train,param);
>>
>>    % Test Random Forest
>>    testTrees_script;
>>
>>    % Visualise
>>    visualise(data_train,p_rf,[],0);
>>
>>    pause;
```

```
>> end
```

Change the tree depth.

```
>> for N = [2,3,5,11]; % Tree depth, try {2,3,5, or 11}
>>    init;
>>    param.num = 20; % number of trees
>>    param.depth = N; % trees depth
>>    param.splitNum = 3; % Number of trials in split function
>>    param.split = 'IG'; % Currently support 'information gain' only
>>
>>    % Select dataset
>>    [data_train, data_test] = getData('Toy_Spiral'); % {'Toy_Gaussian',
>>'Toy_Spiral', 'Toy_Circle', 'Caltech'}
>>
>>    % Train Random Forest
>>    trees = growTrees(data_train,param);
>>
>>    % Test Random Forest
>>    testTrees_script;
>>
>>    % Visualise
>>    visualise(data_train,p_rf,[],0);
>>
>>    pause;
>> end
```

6. Experiment with Caltech101 dataset for image categorisation

We have seen the RF on the toy data above. In this part, we see how we apply RF to a real vision problem. We use a small subset of Caltech101 dataset for image categorisation. The dataset contains a number of different object categories (or classes), and a large number of images per category. In principle, we convert images to high dimensional vectors using the popular technique called bag-of-words, and then apply RF to the vectors in the same way above for the toy data experiments. Note that the toy data has 2 dimensional vectors, while the bag-of-words vectors of images usually have a higher-dimension.

Initialise and set the RF parameters.

```
>> init;
>> param.num = 20; % number of trees
>> param.depth = 10; % trees depth
>> param.splitNum = 3; % Number of trials in split function
>> param.split = 'IG'; % Currently support 'information gain' only
```

Select dataset. We do bag-of-words technique to convert images to vectors (histogram of codewords).
Set 'showImg = 0' in getData.m to stop displaying training and testing images and their feature vectors.

```
>> [data_train, data_test] = getData('Caltech'); % Select dataset
```

Train Random Forest.

```
>> trees = growTrees(data_train,param);
```

Test Random Forest.

```
>> testTrees_script;

Show recognition accuracy and confusion matrix.
>> confus_script;
```