Coursework 1: Randomized Decision Forest

Laura Palacio Garcia (CID: 01322823)
Imperial College London
lp2816@ic.ac.uk

Jenna Luchak (CID: 01429938)
Imperial College London
jkl17@ic.ac.uk

1. Training Decision Forest

Randomized decision forests generalize well to previously unseen data [2] and improve the prediction [5] thanks to randomizing the training dataset (Bagging) and the features (randomized node optimization). We performed Bagging (Boostrap Aggregating) by randomly drawing samples of the same size from the training data [5], creating a new training subset for each tree. The samples were drawn with replacement so that the same data point could be chosen more than once. Figure 1 shows four different data subsets created, where some points might be the same within different subsets. Notice that the 3 different labels or classes correspond to the 3 main colors in the RGB model (red, green and blue).

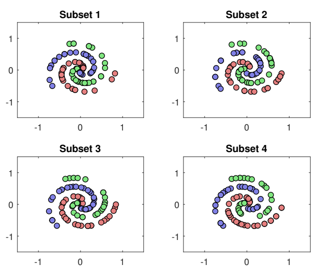
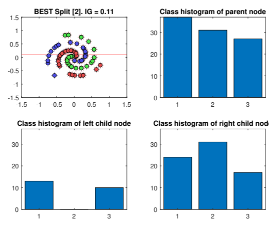


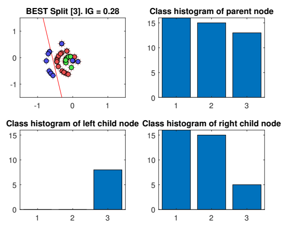
Figure 1. Four data subsets generated by Bagging.

The size of each subset should be large because, otherwise, we would obtain a weak combined prediction. Our subsets have 95 samples, which corresponds to 63.2% of the total number of training data points ($N = 150$ samples). Alternatively, we created subsets of 150 samples with replacement, but the information gain was smaller. Larger subset sizes create less diversity among trees since there is almost no randomization (causing the ensemble model to have no effect). In this case, for large N, we would expect each subset to have 63.2% unique data points.

When splitting a node, the goal is to find the random split function that produces the highest information gain and confidence in the prediction, since we want the histograms of the child nodes to be more pure (exhibit a value in only one class). In Figure 2 below, we tried two split functions: axis-aligned and two-pixel test (a type of linear function). There, we see that the distribution in the parent node is more or less uniform because we have almost the same number of points for each color (as in the right child node histograms). On the other hand, in Figure 2(b), the two-pixel test has a higher information gain since it has separated the data more successfully because the left child node only contains data points belonging to label 3 (color blue).



(a) Axis-aligned



(b) Two-pixel test

Figure 2. Two different split functions with the class histograms of the node and its two children nodes, and a measure of the information gains.

The information gain also depends on the degree of randomness ρ (*param.splitNum* in the code), which is the number of split functions we randomly evaluate at each node until choosing the best split function. Thus, if ρ is very small, every tree in the forest is very different between each other (high randomness). On the other hand, if ρ is large, then the algorithm finds the best split function after having tried a lot of random thresholds. Consequently, it is less random because all trees in the forest will be exactly the same (large tree correlation) [2].

After trying different split functions and degrees of randomness, we created Table 1 that shows the average information gain across all trees across 10 iterations. From Table 1, we see that the two-pixel test generates slightly larger information gain than axis-aligned and ρ does not seem to have a very significant effect (although it seems that $\rho = 5$ gives overall the best result). Given the circular distribution of the data (as seen in Figure 2), neither the axis-aligned nor the two-pixel test are able to separate the data very well. Maybe a nonlinear weak learner could have worked better although it is computationally expensive and the other two functions have been proved to work well.

Weak learner	ρ	Information gain
Two-pixel test	3	0.1948
	5	0.1958
	10	0.1835
	15	0.1788
Vertical axis-aligned	3	0.1176
	5	0.1236
	10	0.1241
	15	0.1142
Horizontal axis-aligned	3	0.1186
	5	0.1233
	10	0.1184
	15	0.1165

Table 1. Information gain with respect to split function and degree of randomness used in randomised decision forest.

Finally, the tree has fully grown and, during testing, the input data reaches a leaf node, where it is classified using the class distribution at the leaf. In Figure 3, we show the class distributions for 9 leaf nodes. We can observe that 6 of them have a distribution such that, if an input data point reaches that leaf node, it will always be associated to a specific color (since that label has a probability of almost 1). However, the class or color prediction is harder in leaf nodes 5, 7 and 8.

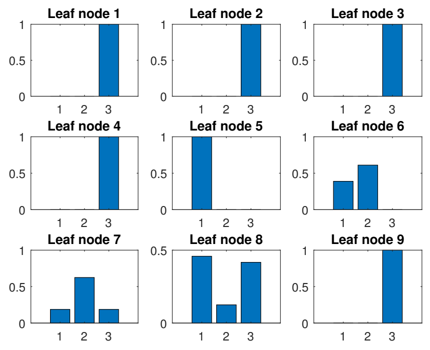


Figure 3. Class distributions of 9 leaf nodes.

The choice of stopping criteria influences the shape of the trees. Here, we stop the tree when a maximum number of depth has been reached because learning very deep trees increases the model complexity and can lead to overfitting. Also, we create a leaf node when the node contains less than 5 training points because it would not be useful to split the data more. Alternatively, we could have stopped when the information gain was smaller than a threshold.

2. Evaluating Decision Forest on the test data

In Figure 4, we show the class distributions of the leaf nodes (for each tree) reached by the third test point, as well as the averaged class distribution across all leaf nodes.

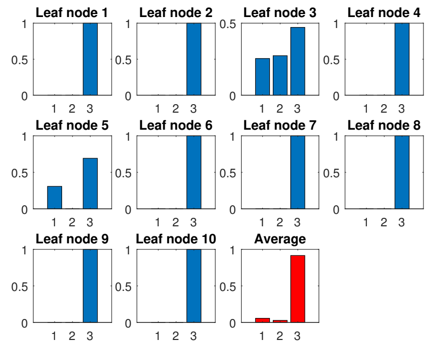
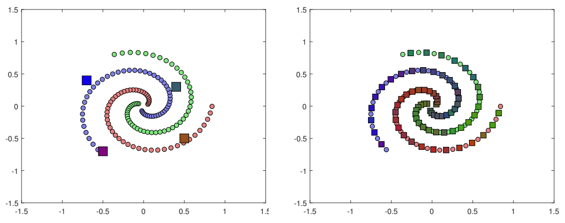


Figure 4. Class distributions of the leaf nodes the data point $[-0.7000, 0.4000]$ arrives to in every of the 10 trees and the averaged class distribution across all leaf nodes.

Then, we can show the predicted color for these four test points in Figure 5(a). The third test point is clearly blue since it had an average maximum probability of belonging to label 3 (as seen in Figure 4). As we observe in both subfigures in Figure 5, since the class distribution is a 3-dimensional vector of RGB values, we can plot the exact predicted color of each test point as a combination of the probabilities of the 3 main colors in RGB (instead of taking the label with the highest probability from the average class distribution).

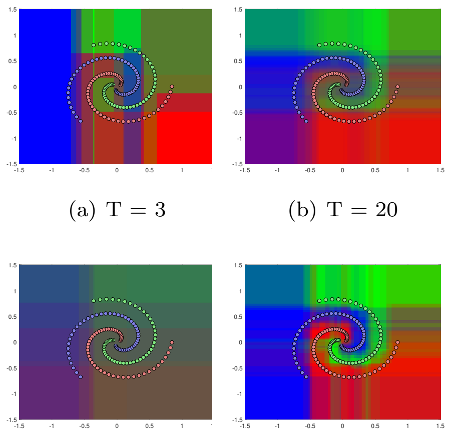


(a) Four test points (b) Half of training data

Figure 5. Color classification of different test points.

Our *data_test* from the Toy Spiral data set does not contain labels. Therefore, in order to test the accuracy of our model, we divided the training data again into two separate subsets: half of the data belongs to the training subset and the other half to the testing subset. This was done by saving alternate samples from each label to a new testing dataset and then erase those from the training dataset. Thus, this allows to have the same amount of data points for each color (label) in every subset. So, we will test our model with a subset of data that was not used in the training process and the predicted colors are shown in Figure 5(b).

Then, we evaluated all data points in a dense 2D grid and the classification results obtained for different parameters are shown in Figure 6. The default parameters were 10 trees, maximum depth: 5, and $\rho = 3$.



(a) T = 3 (b) T = 20

(c) D = 2 (d) D = 11

Figure 6. Visualization of the classification results for all data points in a dense 2D grid for different values of number of trees (T) and tree depth (D).

As we can already see from Figure 6, we obtain more accurate predictions for larger number of trees, number of split functions (degree of randomness) and tree depth. When we increase the tree depth, there are more vertical lines in the dense grid so the prediction is not uniform in an area that should be the same color (as opposed to what happens when we increase the number of trees). With larger degree of randomness, the dense grid classification results also becomes smoother.

With the aim of observing this effect better, we calculated the accuracy of our model by using half the training data as testing data (as we did in Figure 5(b)). We take the true color of each data point and compare it with the color with the highest probability in the average class distribution across all leaf nodes. With the aim of generalizing our model, we repeated this process 10 times and obtained a 85.0667% average accuracy (using the default model parameters with an axis-aligned split function), whereas the two-pixel test obtained a better accuracy (89.7333%).

Then, we tested the classification performance with different values of number of trees (1, 3, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100), depth of trees (2, 5, 7, 11) and degree of randomness (2, 5, 7, 11, 15). We concluded that the testing accuracy increases with the number of trees, tree depth and degree of randomness (as in Figure 8). For instance, the accuracy ranged from 60% to 90% when we increased the number of trees and tree depth. As for the values we tried for ρ, the accuracy increased from 82% to 88% with variations in between since it is not very stable.

Based on the figure we obtained, we decided that the optimal parameters for this classification problem are 40 trees with a maximum depth of 7, and a degree of randomness of 11. The main disadvantage of random forests is that it is hard to choose the value of these parameters. For instance, learning very deep trees increases the model complexity and can lead to overfitting, while the degree of randomness affects the generalization of the classification forest.

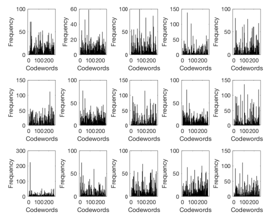
3. Experiment with Caltech101 dataset for image categorisation

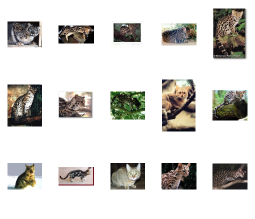
3.1. K-means codebook

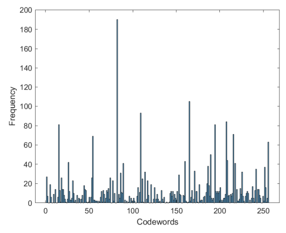
The vocabulary size of our K-means codebook is a matrix of size [150 x *numBins*]. The number of rows in the codebook represents the number of images tested and trained in this data set; a constant value. There are 10 different classes, each has 15 images, for a total of 150. The number of columns is dependent on the variable *numBins*, which represents the number of clusters (bins) used in the algorithm K-means. The visual codebook is a set of cluster centers created by k-means that returns the centroid location of all clusters. The bags of visual words was constructed for every image using the codebook, via a vector quantization process that maps each patch vector to a certain code word.

When the default parameters for Matlab's built in function of K-means was executed, the simulation failed to converge. This behavior was expected, as K-means is time demanding and converges only to local optima;it depends on initialization. To solve this convergence issue, we implemented K-means using new techniques to reduce complexity. We tried increasing the maximum iterations of the algorithm, changing the initial clustering method from K-means++ algorithm (Matlab's default K-means algorithm) to the original K-means algorithm, employing an "Empty Action" parameter (*drop*) to remove any empty clusters (code words of frequency zero), and changing the number of clusters used; none of these techniques aided convergence. However, through this process we did learn that K-means++ computes slightly faster than K-means (139.081 seconds compared to 140.331 seconds), and a positive linear relationship exists between computation time and bin size. These two results were expected. When you look at the computational complexity equation of K-means: $O(DN'K)$, where N is the number of patches, K is the number of clusters and D is the descriptor dimensionality; complexity is proportionally dependent on K. As number of clusters increases, computation time increases and complexity increases which causes K-means to perform more iterations every simulation. Maybe using Principal Component Analysis (which performs dimensionality reduction [3]) could have helped the algorithm converge.

The vector quantization process was an iterative process. Firstly, we initialize the bag-of-words matrix. Secondly, a three part *for* loop was initialized, to ensure that every patch from every image in every class of the data set was mapped to a code word. Thirdly, the distance between the *ith* patch vector and all of the code words was calculated, such that the minimum distance (using *vecnorm*) calculated was returned and the corresponding index of the code word was printed into a histogram vector. The bag-of-words histograms for the training and testing cat images are shown in Figure 7 below.


(a) Histogram of training images


(b) Training images



(c) Histogram of a testing image

(d) Testing image

Figure 7. Bag-of-words histograms of example training and testing images from the category of wildcats.

We plotted histograms for all 15 images from class 7 (wildcat) for training and showed only one for a testing image for clarity. From Figure 7(a) and Figure 7(c), it was found that for a specific class all of the histograms expressed similar distributions. Most of the code words have an even frequency of patch vectors mapped to them, and a few unique outliers spiking to larger frequencies. This result suggests that our K-means codebook is fairly consistent at mapping the patch vectors of images in the same class. Although the histograms across the same class exhibit the same behavior, each image mapped a unique codebook, as can be seen by the varying frequencies across similar code words in Figure 7(a). This is expected, as wildcats have similar discernible features, such as color of fur, spots, and facial features but some of the images have distinctly different features such as the background colors or the fact that the cat is in a different position, a different size and less noticeable.

3.2. RF classifier

We used the training and testing data set, in the form of bag-of-words, obtained in Question 3.1, to train and test a random forest classifier. In order to determine the optimal classifier parameters, we investigated the vocabulary size, type of weak learner, number of trees, depth of trees, and degree of randomness (also called number of split functions ρ).

The approach we used to do the parameter sweep is cross-validation, which prevents over fitting. First, we randomly separate the training data (150 images) into K folds. In our case we chose $K = 5$ so that each subset would contain exactly 30 images. Then, the first 4 subsets (80% of the data) are used for training and the remaining one (20%) will be used for testing.. Therefore, we chose the parameters that produced a high average classification accuracy and a low standard deviation across the 5 folds and 10 iterations.

First, we explored visual vocabulary; how the number of code words affects the classification accuracy of our random forest classifier. The maximum number of colors an image can possibly display at one time is $2^8 = 256$ since image information is stored in 8-bit color graphics, while the lowest number of colors is 2^1 or 2 [1]. We chose to investigate vocabulary size by mapping our training data to a codebook of sizes 2^i, where $i = 1$ to 10. Random forest classifier has the same parameters for every trial: 40 numbers of trees, maximum depth of 8 trees, 20 split functions and an axis-aligned weak-learner. The reasoning behind the choice of these optimal parameters will be discussed throughout the investigation of this report.

At the same time, we investigated the type of weak-learner (axis-aligned and two-pixel test) used in the *splitNode* function of the random forest classifier. Therefore, all 10 vocabulary sizes were repeated for both weak-learners. The classification accuracy and total computation time for each size of codebook and each weak-learner can be seen in the Table 2 below.

From Table 2, we see that, in the case of axis-aligned weak learners, classification accuracy fluctuates up and down as the number of clusters used in K-means increases. The peak accuracy (60.67%) occurred at 256 clusters for the axis-aligned split function, whereas the accuracy of the two-pixel test never improved past 36.53%. The accuracy of the K-means algorithm, when using a proper weak learner, has a

	Axis-aligned		Two-pixel test	
Number of clusters	Accuracy [%]	Time [seconds]	Accuracy [%]	Time [seconds]
2	33.53	8.234	29.87	5.180
4	34.60	8.385	28.53	9.958
8	52.80	14.941	25.87	12.773
16	55.20	24.008	36.53	21.208
32	58.73	32.653	28.80	37.863
64	53.20	57.025	23.07	63.019
128	57.93	90.770	25.07	101.264
256	60.67	132.344	29.87	176.945
512	59.93	217.640	25.07	218.371
1024	46.93	259.450.315	22.53	258.968

Table 2. Classification accuracy and computation time with respect to the number of clusters and type of weak learner.

threshold codebook size since the accuracy does not increase for larger codebook sizes (1024) but the computation time extremely increases. Also, although each weak learner takes similar computation time for different sizes of codebooks, axis-aligned is comparatively more accurate than the two-pixel test across all code book sizes. This result might be unexpected, as the two-pixel test is slightly more complex and supposed to perform better; although it always depends on the data available and what type of function is able to split it better, which in our case might be the axis-aligned.

The last parameters we investigated were number of trees, maximum depth of trees and ρ. We used axis-aligned and 256 clusters for this investigation, as per the conclusions from the previous parameter tests.

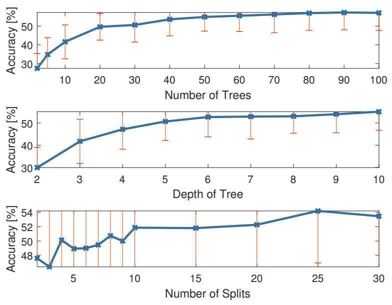


Figure 8. A random forest classifier when a specific param eter is manipulated and the average accuracy is calculatec over 10 trials across 5 folds. Error bars are representative of the standard deviation across all trials.

The methodology for this investigation was iterative and involved three separate tests. For each test, one o the three parameters was manipulated while the othe two parameters remained constant, this way we coulc observe the effects of each parameter individually. Ir the first iteration, constant parameters were used: 10 trees, tree depth: 5 and $\rho = 3$. However, for subsequent iterations, the constant parameters were changed to the optimal value obtained in the previous iteration.

The results from the first iteration are presented in Figure 8, where we see that, as number of trees increases from 0 to 70, the accuracy of the model increases and then plateaus. However, between 40 and 70 trees, accuracy does not improve by a significant margin (3%) and standard deviation is small. Since more trees increase complexity and computation time, 40 is optimal for this iteration. A similar relationship can be concluded for the manipulation of the maximum depth of trees and ρ; the peak accuracies occurred at 8 and 20, respectively. The constant and optimal parameters chosen for the second iteration were number of trees 40, maximum tree depth of 8 and $\rho = 20$. This process was repeated until the optimal parameters for the current trial were equivalent to the previous trial. The parameters behaved similarly across all iterations with the exception of ρ in the final iteration because, as ρ changed, accuracy varied by at most 2%. Therefore, it can be concluded that the accuracy of the random forest classifier is dependent on all parameters; however, when number of trees and maximum depth of trees are optimal, ρ has no effect on accuracy. Also, the sensitivity of accuracy with respect to ρ was the least stable (largest standard deviation).

Using the optimal parameters found, we ran our simulation for 10 trials in order to calculate our model's optimal behavior, as well as build a confusion matrix. The total computation time for our model was 54.6440 seconds, with training time being 2.5922 seconds and testing 0.4505 seconds, which corresponds to 4.7438% and 0.8244% of the total computation time. Thus, we conclude that computation time was dominated by the K-means algorithm and training takes approximately 10 times longer than testing. The average accuracy and standard deviation were 60.2000% and 1.2192%, respectively.

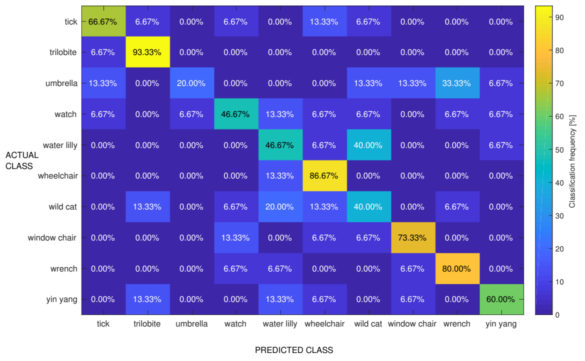
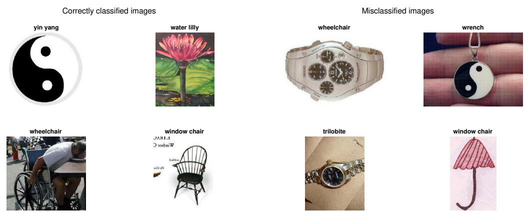


Figure 9. Confusion Matrix of a random forest classifier that uses a K-means codebook mapped from 256 clusters.

The confusion matrix (Figure 9) is a 10 by 10 matrix representing the correlation accuracies between the predicted and actual class labels given in the data set. The diagonal elements of the confusion matrix represent the percentage of instances that the predicted class was the actual class. We see in Figure 9 that the "trilobite", "wheelchair" and "wrench" classes were the most likely to be classified correctly (over 80%), while the class least likely to be classified correctly was the "umbrella" (20%).

Examples of some images that were correctly and incorrectly classified can be seen in Figure 10 below. A possible reason for misclassification may be because some images of different classes have similar features, such as the "water Lilly" and "umbrella" displaying similar colors and shapes, or the "yin and yang" symbol and "watch" both being round. Similar image features may lead to similarly mapped codebooks, thus misclassification.



(a) Correctly classified (b) Incorrectly classified

Figure 10. Example images randomly selected from the data set with the predicted labels.

3.3. RF codebook

Given the same data set, a new visual codebook was created using random forest. We grew trees with our descriptors matrix for all images (with the labels). Then, we tested the trees using the training matrix with all the descriptors and we obtained the leaves each specific descriptor for the image arrived at in each tree. Here, we used *testTrees_fast* instead of simply *testTrees* because it was significantly faster. Finally, the bag-of-words representations of the training image obtained by the RF codebook is obtained by adding the frequency of all the leaves for every image. We repeated the same process in order to create the testing images.

This training and testing data set, in the form of bag of words, was trained and tested using the same random forest classifier as detailed in Question 3.2. In Figure 11, we used again 5-fold cross-validation to compare the classification accuracy with respect to different random forest parameters (number of trees, maximum depth of trees, degree of randomness, weak learner) between the K-means codebook and the RF codebook.

If we compare the results from Figure 11 to those obtained in Figure 8, we can see that the results are more or less the same if we compare building a RF codebook and a K-means one. When we used a two-pixel test as split function, the accuracy dropped to 29.1333%. Finally, there are two RF codebook parameters that can be changed: *PHOW_Sizes* and *PHOW_Step*, which correspond to the scales at which the dense SIFT features are extracted and the pixel steps of the grid at which the dense SIFT features are extracted (the lower, the denser), respectively [4]. No significant accuracy changes occurred after modifying these parameters.

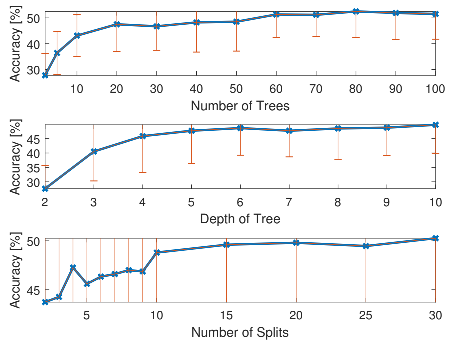


Figure 11. Same as in Figure 8 but using the RF codebook.

The computational complexity equation for K-means was outlined in Question 3.1, it is $O(DN'K)$, while the computational complexity equation for random forest is $O(\sqrt{D}N' \log K)$, where N is the number of patches, K is the number of clusters (or leaf nodes) and D is the descriptor dimensionality. Based on these formulas, it is clear to see that K and D have a larger affect on the complexity of K-means than in random forest. This statement is supported by the comparison in the computational time calculations (averaged across 10 trials) for both methods.

First, we found that building an RF codebook (28.7463 seconds) was much faster than a K-means codebook (54.6440 seconds). Then, we tested the optimal parameters found in Question 3.2 (40 trees, maximum depth: 8, degree of randomness: 20) and we obtained an average accuracy of 61.2667% with a standard deviation of only 1.7902%, while the average training time was 3.1123 seconds and the testing time was 0.5587 seconds. So, the accuracy is similar to when using the K-means codebook although the advantage of using an RF codebook is that it is much more computationally efficient since we have reduced the time to build a codebook to half. Since the data set in this project was large, the random forest was expected to perform more optimally and faster since labelling a descriptor with a balanced tree requires $O(\log K)$ operations whereas K-means costs $O(KD)$.

References

[1] BBC Bitesize. Encoding images: Colour depth. [Online]. Available: https://www.bbc.com/education/guides/zqyrq6f/revision/2, 2018. [Accessed March 11, 2018].

[2] Kari Pulli (Senior Director, NVIDIA Research). Machine Learning for Vision: Random Decision Forests and Deep Neural Networks. [Online]. Available: http://web.stanford.edu/class/cs231m/spring-2014/docs/machine-learning.pdf, 2014. [Accessed March 2, 2018].

[3] J. Shlens. A tutorial on principal component analysis. *Measurement*, 51, 2005.

[4] The VLFeat Authors. VLFeat - Documentation. [Online]. Available: http://www.vlfeat.org/matlab/vl_phow.html, 2007. [Accessed March 14, 2018].

[5] Trevor Hastie (Stanford University). Trees, Bagging, Random Forests and Boosting. [Online]. Available: http://jessica2.msri.org/attachments/10778/10778-boost.pdf, 2004. [Accessed March 6, 2018].

Appendix

Question 1: Training Decision Forest

In order to show four data subsets obtained from Bagging, we added this code in function *growtrees.m* (inside the *for* loop after generating indexes for the training dataset):

```
% Plot subset training data used for this tree
if visualize
  if T == 1
    figure
  end
  if T <= 4
    subplot(2,2,T)
    plot_toydata(data(idx,:));
    title(['Subset ', num2str(T)])
  end
end
```

The size of the subset is calculated in *grwoTrees.m*:

```
% Bootstraping aggregating
% A new training set for each tree is generated by random sampling from dataset WITH replacement.
%idx = randsample(N,N,1); % same size as training data
idx = randsample(N,ceil(N*frac),1); % size is 63.2% of total training data
prior = histc(data(idx,end),labels)/length(idx);
```

Then, inside *spliNode.m*, we added the possibility to try another split function such as two-pixel test (using a *switch* statement that considers the split function we want to use or *mode*):

```
switch mode
 case 'axis'
 % Axis-aligned
 m = 0; % input empty of visualise (used only for linear)
 b = 0; % input empty of visualise (used only for linear)
 dim = randi(D-1); % Pick one random dimension (1 is vertical and 2 is horizontal)
 d_min = single(min(data(:,dim)))+eps; % Find the data range of this dimension
 d_max = single(max(data(:,dim)))-eps;
 t = d_min + rand*((d_max-d_min)); % Pick a random value within the range as threshold
 idx_ = data(:,dim) < t;
 case 'two'
% Two-pixel test
 dim = 3; % used just to visualise (so
     it does not get confused with axis-
     aligned)
 d_xmin = single(min(data(:,1))) + eps;
 d_xmax = single(max(data(:,1))) - eps;
 d_ymin = single(min(data(:,2))) + eps;
 d_ymax = single(max(data(:,2))) - eps;
 x_t = d_xmin + rand([1,2])*((d_xmax-
     d_xmin));
 y_t = d_ymin + rand([1,2])*((d_ymax-
     d_ymin));
 m = (y_t(2)-y_t(1))/(x_t(2)-x_t(1)); %
     slope
 b = y_t(1) - m*x_t(1); % initial value
 t = m*data(:,1) + b; % threshold
 idx_ = data(:,2) < t;
end
```

To visualize the two-pixel test split function, we modified *visualise_splitfunc.m* so that it took it into account. So, we added the slope and the initial value of the function as inputs and we added another condition in the *if* statement:

```
function visualise_splitfunc(idx_best,
    data,dim,t,ig_best,iter,m,b) % Draw
    the split line
r = [-1.5 1.5]; % Data range

subplot(2,2,1);
if dim == 1 % vertical axis-aligned
    plot([t t],[r(1),r(2)],'r');
elseif dim == 2 % horizontal axis-
    aligned
    plot([r(1),r(2)],[t t],'r');
else % linear / two-pixel function
    plot([r(1) r(2)],[m*r(1)+b m*r(2)+b
        ],'r');
end
```

The function to update the information gain also changes and we included new inputs and outputs:

```
function [node, ig_best, idx_best] =
    updateIG(node,ig_best,ig,t,idx,dim,
    idx_best,m,b) % Update information
    gain
if ig > ig_best
  ig_best = ig;
  node.t = t;
  node.dim = dim;
  idx_best = idx;
  node.m = m; % update
  node.b = b;
else
  idx_best = idx_best;
end
end
```

In order to calculate the average best information gain for different split functions, we created a new output for *growTrees* called *global_ig*, which was a matrix containing the information gain for every tree for each time we split a node:

```
% Initialise base node
tree(T).node(1) = struct('idx',idx,'t',
    nan,'dim',-1,'prob',[],'m',[],'b'
    ,[]);

% Split Nodes
for n = 1:2^(param.depth-1)-1
  [tree(T).node(n),tree(T).node(n*2),
      tree(T).node(n*2+1),ig_best] =
      splitNode(data,tree(T).node(n),
      param,mode);
  global_ig(T,n) = ig_best;
end
```

Then, in *main_guideline*, we iterated *growTrees* 10 times and calculated the average information gain like this (*mode* is the type of split function we use):

```
% Grow all trees
for it = 1:10
  [trees,global_ig] = growTrees(
      data_train,param,mode);
  final_ig(it) = nanmean(nanmean(
      global_ig));
end
mean(final_ig)
```

Finally, instead of using function *visualise_leaf.m*, we preferred to create our own figure to visualize the first 9 leaf nodes and this is the code we added inside *main_guideline.m*:

```
figure
idx_tree = 1;
idx_node = 1;
for i = 1:9
  try
    prob_leaf = trees(idx_tree).leaf(
        idx_node).prob;
  catch
    idx_tree = idx_tree + 1;
    idx_node = 1;
    prob_leaf = trees(idx_tree).leaf(
        idx_node).prob;
  end
  subplot(3,3,i)
  bar(prob_leaf);
  title(['Leaf node ', num2str(i)])
  idx_node = idx_node + 1;
end
```

Question 2: Evaluating Decision Forest on the test data

In this section, we tested the performance of our RF. This is the code written to Visualise the class distributions of the leaf nodes which the data point arrives at and the averaged class distribution:

```
all_probs = [];
for n = 1:size(test_point,1)
  leaves = testTrees([test_point(n,:)
      0],trees,mode);
  % average the class distributions of
      leaf nodes of all trees
  p_rf = trees(1).prob(leaves,:);

  if length(trees) == 1
    p_rf_sum = p_rf;
  else
    p_rf_sum = sum(p_rf)/length(trees);
  end

  % Plot class distributions of the
      leaf nodes
  figure
  for idx_tree = 1:length(trees)
    subplot(3,4,idx_tree)
    bar(p_rf(idx_tree,:));
    title(['Leaf node ', num2str(
        idx_tree)])
  end
  subplot(3,4,11)
  bar(p_rf_sum, 'r')
  title('Average')

  all_probs(n,:) = p_rf_sum;
end
```

In order to evaluate all the data points in the dense 2D grid and visualise their classification results, we used this code:

```
%% Test on the dense 2D grid data, and
    visualise the results
if test_train == 0
  for n = 1:size(data_test,1)
    leaves = testTrees(data_test(n,:),
        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);

    all_probs_test(n,:) = p_rf_sum;
  end
  visualise(data_train,all_probs_test
      ,0,0)
end
```

The *testTrees* function had to be changed so that it took into account the two-pixel test. So, the input *mode* specifies the type of split function that has been used for growing the trees.

```
t = tree(T).node(idx).t;
split_b = tree(T).node(idx).b;
split_m = tree(T).node(idx).m;
dim = tree(T).node(idx).dim;
% Decision
switch mode
 case 'axis'
  decision = data(m,dim) < t; % Pass
      data to left node (axis-aligned)
 case 'two'
  decision = data(m,2) < split_m*data(m
      ,1) + split_b;
end

if decision % Pass data to left node (
    two-pixel test)
  idx = idx*2;
else
  idx = idx*2+1; % and to right
end
```

This is the code we created to separate the *data_train* equally into testing and training data:

```
% Separate training and testing
if test_train == 1
  data_test = [];

  % Label 1 (color red)
  idx = find(data_train(:,3) == 1);
  idx_test = idx(1:2:length(idx));
  data_test = data_train(idx_test,:);
  data_train(idx_test,:) = [];

  % Label 2 (color green)
  idx = find(data_train(:,3) == 2);
  idx_test = idx(1:2:length(idx));
  data_test = [data_test; data_train(idx_test,:)];
  data_train(idx_test,:) = [];

  % Label 3 (color blue)
  idx = find(data_train(:,3) == 3);
  idx_test = idx(1:2:length(idx));
  data_test = [data_test; data_train(idx_test,:)];
  data_train(idx_test,:) = [];
end
```

Then, we calculated the accuracy of our model like this:

```
%% Accuracy
if test_train == 1
  true_labels = [];
  predicted_labels = [];
  for n = 1:size(test_point,1)
    true_labels(n) = data_test(n,3);
    [~, predicted_labels(n)] = max(all_probs(n,:));
  end
end
accuracy = sum(true_labels == predicted_labels)/size(test_point,1)*100;
```

Finally, we wanted to test the effect of different RF parameters on the accuracy of our model and we created three *for* loops to evaluate this. We used 5-fold cross-validation and, in order to provide proof that the model is generalized, we repeated the same process in 10 iterations. This is the loop created to test the effect of changing the number of trees and plot the average accuracy (the other two loops will be almost the same):

```
%% Change the RF parameter values and evaluate

% Set the random forest parameters
num_trees = [1, 3, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]; % Number of trees
depth_trees = [2, 5, 7, 11]; % Maximum depth of trees
splitNum_trees = [2, 5, 7, 11, 15]; % Number of split functions to try (p)

original_data_train = data_train;

% Reorder data randomly
randidx = randperm(size(original_data_train,1));
original_data_train = original_data_train(randidx,:);

% Cross-validation: Create the 5 subsets
K = 5; % number of folds
subset_size = size(original_data_train,1)/K; % 150/5 = 30 images for each subset

% 5-fold cross-validation: K-1 subsets for training and 1 for validation
for iter = 1:10
    for fold = 1:K
        % Validation subset (only one subset)
        start_idx = subset_size*(fold-1) + 1;
        stop_idx = start_idx + subset_size - 1;
        data_test = original_data_train(start_idx:stop_idx,:); % train and test in same data

        % Training subsets (the rest of the K-1 subsets)
        data_train = original_data_train;
        data_train(start_idx:stop_idx,:) = []; % exclude the subset used for testing

            %% Testing number of trees: param_num
        for i = 1:length(param_num)
            param.num = param_num(i);
            param.depth = 8;
            param.splitNum = 20;

            % Train Random Forest
            tic; % Start timer
            tree = growTrees(data_train,param,mode);
            stop = toc; % Stop the timer
            fprintf('TIC TOC Train random forest: %g\n', stop);

            % Evaluate/Test Random Forest
            tic; % Start timer
            for n=1:size(data_test,1) %
```

```
                Iterate through all rows of test data
                leaves = testTrees([data_test(n,:) 0],tree,mode); % Call the testTrees function
                % average the class distributions of leaf nodes of all trees
                p_rf = tree(1).prob(leaves,:);
                p_rf_sum = sum(p_rf)/length(tree);
                [~,predicted_label(n)] = max(p_rf_sum);
            end
            stop = toc; % Stop the timer
            fprintf('TIC TOC Test random forest: %g\n',stop);

            % Calculate accuracy of classifier
            actual_label = data_test(:,end);
            in_tree_accuracy(fold,i) = sum(actual_label == predicted_label')/length(actual_label)*100;
        end

        end
    % Calculate average and std accuracy across folds
    trees_accuracy(iter,:) = mean(in_tree_accuracy);
    trees_standard_dev(iter,:) = std(in_tree_accuracy);
end

% Calculate average and std accuracy across iterations
global_trees_accuracy = mean(trees_accuracy);
global_trees_standard_dev = mean(trees_standard_dev);

figure,
subplot(3,1,1)
plot(param_num, global_trees_accuracy,'-x','linewidth',2);
hold on
errorbar(param_num,global_trees_accuracy,global_trees_standard_dev);
xlim([min(param_num) max(param_num)]);
ylim([min(global_trees_accuracy) max(global_trees_accuracy)]);
xlabel('Number of Trees');
ylabel('Accuracy [%]');
```

Question 3: Experiment with Caltech101 dataset for image categorisation

Question 3.1: K-means codebook

The K-means codebook was obtained by modifying the function (getData). First, We built the clusters for each patch vector in the entire data set (*desc_tr*) by calling the algorithm for K-means from Matlab's toolbox. For our investigation on convergence, multiple calls of the K-means function were tested and are commented below the default. The number of clusters created by K-means is represented by variable *"numBins"* in our code. For our investigation on codebook size, this variable was manipulated. We also calculated the computation time it took the K-means function to cluster the data and printed it out. The code that we inserted into (getData) is as follows.

```
%% K-means clustering for each patch in the entire data set
% Call the kmeans function
% Inputs:
%   desc_sel': The transpose of the entire data set.
%
%   numBins: Number of clusters.
%
% Outputs:
%   index: 100,000 x 1 - Returns the cluster index for all
%          100,000 patch vectors
%
%   center: numBins x 128 returns the centroid location of each
%            cluster in desc_sel.

tic; % Initiate the timer
numBins = 256; % Number of clusters default.

% Call k means algorithm. Uncomment the line for investigation on convergence
```

```
[~,center] = kmeans(desc_sel',numBins,'MaxIter',100); % default
%[~,center] = kmeans(desc_sel',numBins,'EmptyAction','drop');
%[~,center] = kmeans(desc_sel',numBins,'Start','sample');

stop_kMeans = toc; % Stop the timer
fprintf('TIC TOC K-means: %g\n', stop_kMeans); % Print the time it takes to run kmeans
```

Second, we built a vector quantization method for the training data, such that every patch vector in every image was mapped to it's corresponding closest cluster. The data we added for vector quantization is as follows.

```
%% Vector Quantization training data
disp('Encoding Images...')

tic; % Initiate timer
numb_images = 0; % Initiate number of images counter
data_train = zeros(numel(desc_tr), numBins+1); % Initiate data_train

% For every single image in the training data set,
% determine the closest cluster from the kmeans algorithm relative
% to each patch vector, and create a Bag Of Words (BOW).
for r = 1:1:size(desc_tr,1) % Iterate through all rows
    for c = 1:1:size(desc_tr,2) % Iterate through all columns
        image = desc_tr{r,c}; % Define the current loops image
        hist_vector = zeros(1,length(image)); % Reset

        for i = 1:1:length(image) % Iterate through all patch vectors
            db_I = double(image(:,i)); % Convert to double
            db_center = double(center'); % Convert to double and transpose

            % Calculate the distance between the current patch
            % vector and all of the clusters
            distances = vecnorm(db_I - db_center);

            % Determine the index of the cluster corresponding to
            % the minimum distance and print it into a vector
            [~,cluster_num] = min(distances);
            hist_vector(i) = cluster_num;
        end
        % Use the histogram function to create a bag of words for every image
        numb_images = numb_images + 1; % Update number of images counter
        [BOW,~] = hist(hist_vector, numBins); % Generate bag of words vector

        % Visual Vocabulary - Training Data
        data_train(numb_images,1:end-1) = BOW; % Store BOW words
        data_train(numb_images,end) = r; % Label

        if r == 7 %& (c==13 | c==14 |c==15)
            % Plot a histogram for training data bag of words
            if c == 1
                f1 = figure;
                f2 = figure;
            end
            figure(f1)
            subplot(3,5,c)
            histogram(hist_vector, numBins);
            xlabel('Codewords');
            ylabel('Frequency');

            % Plot the image
            subFolderName = fullfile(folderName,classList{r});
            imgList = dir(fullfile(subFolderName,'*.jpg'));
            I = imread(fullfile(
                    subFolderName,imgList(images_training(r,c)).name));
            figure(f2)
            subplot(3,5,c)
            imshow(I);
        end
    end
end
stop = toc; % Stop the timer
fprintf('TIC TOC bag of words training: %g\n', stop);

% Clear unused variables to save memory
clearvars desc_tr desc_sel
```

The vector quantization process was replicated for the testing data, as follows.

```
%% Vector Quantization testing data
disp('Encoding Images...')

tic; % Initiate timer
numb_images = 0; % Initiate number of images counter
data_query = zeros(numel(desc_te), numBins+1); % Initiate data_test

% For every single image in the testing data set,
% determine the closest cluster from the kmeans algorithm relative
% to each patch vector, and create a Bag Of Words (BOW).
for r = 1:1:size(desc_te,1) % Iterate through all rows
    for c = 1:1:size(desc_te,2) % Iterate through all columns
        image = desc_te{r,c}; % Define the current loops image
        hist_vector = zeros(1,length(image)); % Reset

        for i = 1:1:length(image) % Iterate through all patch vectors
            db_I = double(image(:,i)); % Convert to double
            db_center = double(center'); % Convert to double and transpose

            % Calculate the distance between the current patch
            % vector and all of the clusters
            distances = vecnorm(db_I - db_center);

            % Determine the index of the cluster corresponding to
            % the minimum distance and print it into a vector
            [~,cluster_num] = min(distances);
            hist_vector(i) = cluster_num;
        end
        % Use the histogram function to create a bag of words for every image
        numb_images = numb_images + 1; % Update number of images counter
        [BOW,~] = hist(hist_vector, numBins); % Generate bag of words vector

        % Visual Vocabulary - Testing Data
        data_query(numb_images,1:end-1) = BOW; % Store BOW words
        data_query(numb_images,end) = r; % Label

       if r == 7 %& (c==13 | c==14 |c==15)
           % Plot a histogram for training data bag of words
           if c == 1
               f3 = figure('Units','normalized','Position',[.5 .1 .4 .9]);
                  suptitle('Testing image representations: 256-D histograms');
                f4 = figure;
           end

           figure(f3)
           subplot(3,5,c)
           histogram(hist_vector, numBins);
           xlabel('Codewords');
```

```
            ylabel('Frequency');

            % Plot the image
            subFolderName = fullfile(folderName,classList{r});
            imgList = dir(fullfile(subFolderName,'*.jpg'));
            I = imread(fullfile(subFolderName,imgList(images_testing(r,c)).name));
            figure(f4)
            subplot(3,5,c)
            imshow(I);
        end
    end
end
stop = toc; % Stop the timer
fprintf('TIC TOC bag of words testing: %g\n', stop);

% Clear unused varibles to save memory
clearvars desc_te
```

Question 3.2: RF classifier

In order to build our random forest classifier, we first called the K-means codebook created in function *getData*. Please note that the entire RF classifier is inside of a *for* loop so that we were able to investigate the classifiers accuracy across different number of clusters. After obtaining the output variables from *getData* the RF classifier was built. For every cluster size tested, the classifier was run 10 times, and from those results, the average accuracy and standard deviation of those accuracies was returned. The code for these two steps are as follows.

```
init;
numBins = 256; % Optimal number of clusters
%numBins = [2 4 8 16 32 64 128 256 512 1024]; % Uncomment for testing how number of clusters affects k means
mode = 'axis';
for i=1:length(numBins)
    %% Collect data from K means Codebook
    [data_train, data_test, stop_kMeans(i), images_training, images_testing] = getData('Caltech',numBins(i));

    %% Build RF classifier

    for it = 1:10 % Iterate the random forest classifier over 10 trials
        % Set the random forest parameters
        param.num = 40; % The number of trees
        param.depth = 8; % Maximum depth of the trees
        param.splitNum = 20; % Number of set of split parameters theta i.e. p = 3
        param.split = 'IG'; % Objective function 'iformation gain' Degree of randomness parameter

        % Train Random Forest
        tic; % Start timer
        tree = growTrees(data_train, param,mode);
        stop_train(i) = toc; % Stop the timer
        fprintf('TIC TOC Train random forest: %g\n', stop_train(i));

        % Evaluate/Test Random Forest
        tic; % Start timer
        for n=1:size(data_test,1) % Iterate through all rows of test data
            leaves = testTrees([data_test(n,:)],tree,mode); % Call the testTrees function
            % average the class distributions of leaf nodes of all trees
            p_rf = tree(1).prob(leaves,:);
            p_rf_sum = sum(p_rf)/length(tree);
            [~,predicted_label(n)] = max(p_rf_sum);
        end
        stop_test(i) = toc; % Stop the timer
        fprintf('TIC TOC Test random forest: %g\n', stop_test(i));

        % Calculate accuracy of
            classifier
        actual_label = data_test(:,end)
            ;
        accuracy(it) = sum(actual_label
            == predicted_label')/length
            (actual_label)*100;
    end

    avg_accuracy(i) = mean(accuracy');
        % Calculate average accuracy
    standard_dev(i) = std(accuracy'); %
        Calcuulate stadard deviations
        across trials
end
```

Next, we created an if statement to follow the *for* loop that trained and tested the K-means code book. If there is more than 1 size of codebook being investigated, a graph can be plotted to visual the change in accuracy of the model. The code used for this is as follows.

```
% Plot a graph of the classification
    accuracy for each trial only if
    testing
% various numbers of clusters for k
    means
if length(numBins)>1
figure,
plot(numBins,avg_accuracy,'-ob');
set(gca,'fontsize',16);
xlabel('Number of Clusters');
ylabel('Classification Accuracy');
axis([min(numBins) max(numBins) 25 50])
    ;
end
```

Next, another *if* statement was generated for the RF classifier. If only one codebook size is being tested, a confusion matrix will be calculated and displayed. The code for this section is as follows.

```
%% Calculate the confusion matrix
num_classes = 10;
confusion_matrix = zeros(num_classes,
    num_classes); % initialize
for test_class = 1:num_classes
    idx_test = find(actual_label ==
        test_class);
    if isempty(idx_test) == 0 % there
        is at least one value of this
        class
        for pred_class = 1:num_classes
            idx_pred = find(
                predicted_label(idx_test
                ) == pred_class);
            if isempty(idx_pred) == 0 %
                there is at least one
                value of this class
                confusion_matrix(
                    test_class,
                    pred_class) = length
                    (idx_pred)/length(
                    predicted_label(
                    idx_test));
            end
        end
    end
end

% Plot confusion matrix with colormap
figure
ax1 = axes('Position',[0 0 1 1],'
    Visible','off');
ax2 = axes('Position',[.2 .2 .7 .7]);
imagesc(confusion_matrix*100)
c = colorbar;
c.Label.String = 'Classification
    frequency [%]';
%title('Confusion matrix for 4-way
    classification')

% Create strings of the percentage in
    confusion matrix
value_perc = num2str(confusion_matrix
    (:)*100,'%0.2f');
value_perc = [value_perc, repmat('%',
    numel(confusion_matrix),1)];
value_perc = strtrim(cellstr(value_perc
    )); % remove any space padding

% Find the value in the middle of
    square to place text with accuracy
[x,y] = meshgrid(1:size(
    confusion_matrix,1));
strings = text(x(:),y(:),value_perc(:),
    'HorizontalAlignment','center');
mid_pos = mean(get(gca,'CLim')); % find
    middle value of the color range

% Change text colors
text_colors = repmat(confusion_matrix
    (:)*100 < mid_pos,1,3);
set(strings,{'Color'},num2cell(
    text_colors,2));

```

```
% Define labels and replace spaces with "newline" for better visualisation
labels = {'tick','trilobite','umbrella','watch','water lilly','wheelchair',...
          'wild cat','window chair','wrench','yin yang'};

% Change axis
set(gca,'XTick',1:size(confusion_matrix,1),'XTickLabel',labels,...
    'YTick',1:size(confusion_matrix,1),'YTickLabel',labels);
%ytickangle(90); % rotate 90 degrees the Y axis labels

% Add the labels of actual and predicted class
axes(ax1) % sets ax1 to current axes
text(0.09,0.55,{'ACTUAL';'CLASS'})
text(0.45,0.12,'PREDICTED CLASS')
```

Lastly, two figures were plotted. One to visualize images that were correctly classified by our RF classifier, and the other to display images misclassified. The code for this section is as follows.

```
%% Plot example success/failures

folderName = 'Caltech_101/101_ObjectCategories';
classList = dir(folderName);
classList = {classList(3:end).name}; % 10 classes

numImages = 4; % to plot
success = predicted_label' == actual_label;

% Find index of images correctly classified and the ones missclassified
idx_success = find(success == 1);
idx_failure = find(success == 0);

% Randomly select some images to plot
rand_images_success = randsample(idx_success,numImages);
rand_images_failure = randsample(idx_failure,numImages);

% Create global list images used for testing
all_images_testing = [];
for row = 1:size(images_testing,1)
    all_images_testing = horzcat(all_images_testing,images_testing(row,:));
end

% Plot success and failures
fig_success = figure;
fig_failure = figure;
for m = 1:length(rand_images_success)
    % Success
    label = predicted_label(rand_images_success(m));
    num_image_folder = all_images_testing(rand_images_success(m));

    % Plot the correctly classified image
    subFolderName = fullfile(folderName,classList{label});
    imgList = dir(fullfile(subFolderName,'*.jpg'));
    I = imread(fullfile(subFolderName,imgList(num_image_folder).name));
    figure(fig_success)
    subplot(2,2,m)
    imshow(I);
    title(labels(label))

    % Failure
    pred_label = predicted_label(rand_images_failure(m));
    true_label = actual_label(rand_images_failure(m));
    num_image_folder = all_images_testing(rand_images_failure(m));

    % Plot the misclassified image
    subFolderName = fullfile(folderName,classList{true_label});
    imgList = dir(fullfile(subFolderName,'*.jpg'));
    I = imread(fullfile(subFolderName,imgList(num_image_folder).name));
    figure(fig_failure)
    subplot(2,2,m)
    imshow(I);
    title(labels(pred_label))
end
```

```
figure(fig_success)
suptitle('Correctly classified images')

figure(fig_failure)
suptitle('Misclassified images')
```

Question 3.3: RF codebook

We created a new function *getData_rf* to obtain the training and testing images. First, we obtain the matrix with the descriptors for the training images (*desc_tr*), by copying the same code as in *getData*. Then, we add the labels in the last column and create a visual vocabulary by applying RF and building a tree. The new code we added after getting *desc_tr* is:

```
% Add labels
for idx_row = 1:size(desc_tr,1)
  for idx_col = 1:size(desc_tr,2)
    desc_tr_labels{idx_row,idx_col} = [
        desc_tr{idx_row,idx_col},repmat(
        idx_row,128,1)];
  end
end

% Build visual vocabulary (codebook)
    for 'Bag-of-Words method'
desc_sel = single(vl_colsubset(cat(2,
    desc_tr_labels{:}), 10e4)); %
    Randomly select 100k SIFT
    descriptors for clustering

param.num = 10; % The number of trees
param.depth = 5; % Maximum depth of the
     trees
param.splitNum = 3; % Number of set of
    split parameters theta i.e. p = 3
param.split = 'IG'; % Objective
    function 'information gain' Degree
    of randomness parameter
tree = growTrees(desc_sel',param,'axis'
    );
```

Then, this tree is used to find the leaves where each descriptor of every image arrives at. The next step is to calculate the frequency of each leaf node in the image (count the number of times the descriptors arrive at that leaf). This is how we create the training images. The code is the following:

```
% Training images
total_num_leaves = tree(size(tree,2)).
    leaf(end).label;
count_image = 1;
for i = 1:size(desc_tr,1)
  for j = 1:size(desc_tr,2)
    leaves = testTrees_fast(desc_tr{i,j
        }',tree); % for each image

    % Frequency histogram
    % Count each leaf
    for num_leaf = 1:total_num_leaves
      freq_leaves(num_leaf) = length(
          find(leaves == num_leaf));
    end

    %         figure
    %         plot(freq_leaves)
    %         xlabel('Leaf node');
    %         ylabel('Frequency');

    % From image to frequency of leaves
        across all trees
    images_train(count_image,:) =
        freq_leaves; % Store BOW words
    images_train(count_image,end) = i;
        % Label
    count_image = count_image + 1;
  end
end
```

Then, we create the testing images in the same way:

```
%% TESTING
cnt = 1;
% Load Images -> Description (Dense
    SIFT)
for c = 1:length(classList)
  subFolderName = fullfile(folderName,
      classList{c});
  imgList = dir(fullfile(subFolderName,
      '*.jpg'));
  imgIdx_te = imgIdx{c}(imgSel(1)+1:sum
      (imgSel));

  for i = 1:length(imgIdx_te)
    I = imread(fullfile(subFolderName,
        imgList(imgIdx_te(i)).name));
    if size(I,3) == 3
      I = rgb2gray(I);
    end
    [~, desc_te{c,i}] = vl_phow(single(
        I),'Sizes',PHOW_Sizes,'Step',
        PHOW_Step);
  end
end

% Training images
```

```
total_num_leaves = tree(size(tree,2)).leaf(end).label;
count_image = 1;
for i = 1:size(desc_te,1)
  for j = 1:size(desc_te,2)
    leaves = testTrees_fast(desc_te{i,j}',tree); % for each image

    % Frequency histogram
    % Count each leaf
    for num_leaf = 1:total_num_leaves
      freq_leaves(num_leaf) = length(find(leaves == num_leaf));
    end

%
%          figure
%          plot(freq_leaves)
%          xlabel('Leaf node');
%          ylabel('Frequency');

    % From image to frequency of leaves across all trees
    images_test(count_image,:) = freq_leaves; % Store BOW words
    images_test(count_image,end) = i; % Label

    count_image = count_image + 1;
  end
end
```

Finally, in *main_guideline.m*, we train and test the classifier using the training and testing images obtained from *getData_rf*. We inspected the optimal parameters by applying 5-fold cross-validation again, as shown in Appendix 3.3. The code we show now is the one we used when we trained and tested the model using the optimal parameters across 10 iterations:

```
init

% Build codebook
[data_train, data_test] = getData_rf(); % RF codebook

% Set the random forest parameters
param.num = 40;
param.depth = 8;
param.splitNum = 20;
param.split = 'IG'; % Objective function 'iformation gain' Degree of randomness parameter
mode = 'axis';
for iter = 1:10
    % Train Random Forest
    tic; % Start timer
    tree = growTrees(data_train,param,mode);
    stop_train(iter) = toc; % Stop the timer

    % Evaluate/Test Random Forest
    tic; % Start timer
    for n=1:size(data_test,1) % Iterate through all rows of test data
        leaves = testTrees([data_test(n,:) 0],tree,mode); % Call the testTrees function
        % average the class distributions of leaf nodes of all trees
        p_rf = tree(1).prob(leaves,:);
        p_rf_sum = sum(p_rf)/length(tree);
        [~,predicted_label(n)] = max(p_rf_sum);
    end
    stop_test(iter) = toc; % Stop the timer

    % Calculate accuracy of classifier
    actual_label = data_test(:,end);
    accuracy(iter) = sum(actual_label == predicted_label')/length(actual_label)*100;
end

% Calculate average and standard deviation accuracy and computation time
std_accuracy = std(accuracy')
avg_accuracy = mean(accuracy)
avg_stop_train = mean(stop_train)
avg_stop_test = mean(stop_test)
avg_time = avg_stop_train + avg_stop_test
```