Randomized Decision Forests for Classification

Angelos Filos
Imperial College London
angelos.filos14@imperial.ac.uk

Youssef Rizk
Imperial College London
youssef.rizk14@imperial.ac.uk

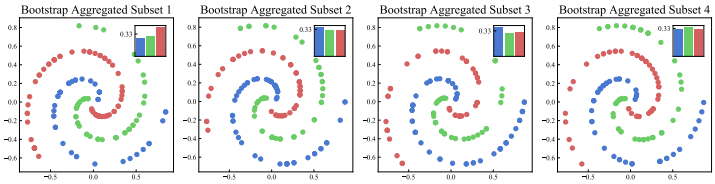


Figure 1. Subsets generated by random sampling of the full training set. Label class distributions provided at sub-axes.

1. Training Decision Forest

The Randomized Decision Forest (a.k.a Random Forest) (**RF**) classifier is an ensemble of simple sub-estimators, Decision Trees (**DT**), which heavily relies on the uncorrelation between the estimators comprising it. A fully deterministic training process would lead to identical sub-estimators and therefore a null ensemble forest. Thus randomness is introduced during training to "break correlation".

1.1. Bootstrap Aggregating (Bagging)

As a first step, each tree is trained on a slightly different training set, constructed after randomly sampling with (**Bagging**) or without replacement the complete training set. Let $\mathcal{S}_0$ denote the original training set and $\mathcal{S}_0^t$ the t-th subset generated from $\mathcal{S}_0$, where $t = 1, ..., T$ and T is the number of sub-estimator trees in the RF. We denote with n the size of $\mathcal{S}_0$ and n_t the size of $\mathcal{S}_t$. The size of the subsets n_t and the sampling process (with or without replacement) are hyperparameters that can be optimized for each application. For our purposes, however, we set $\boldsymbol{n_t = n}$ and **uniformly sample with replacement** from $\mathcal{S}_0$, obtaining four subsets, illustrated in Figure 1.

Because of the sampling with replacement, the generated subsets contain repeated data points, and almost 63.2% of them are unique in the case of $n_t = n$. Figure 17 shows uniqueness for different ratios $\frac{n_t}{n}$. Moreover, as shown in the sub-axes of Figure 1, the classes are almost equally represented (each label $\sim$ 33% density), which can be explained by the uniform sampling from an inherently uniformly represented data set (50 samples per class).

1.2. Split Functions

Each of the $\mathcal{S}_t$ subsets is provided as input to one of the constituent trees in the RF, and a tree is grown. We now consider a single tree. At each node in the tree, the incoming data is split into two disjoint data sets that go to the left and right child nodes, using a split function (a.k.a weak learner). The root note is split using different weak learners, namely axis-aligned, linear, quadratic, and cubic, and the resulting decision boundaries along with the class histograms of the root and child nodes are provided in Figure 2. The intrinsic flexibility of the higher-order polynomial decision functions (quadratic & cubic) is obvious, since they can learn non-linear decision boundaries, explained by their higher degrees of freedom.[1] Nonetheless, their excess discriminative ability can lead to model over-parameterization and thus overfit the training data.

Weak Learners Information Gain & `num_split`				
`num_split`	axis-aligned	linear	quadratic	cubic
1	0.0333	0.0131	0.0056	0.0053
5	0.1129	0.0825	0.1084	0.0895
10	0.1235	0.1529	0.1375	0.1088
50	0.1497	0.1529	0.1600	0.1508
1000	0.1697	0.1600	0.1919	0.1986

Table 1. Information Gain variation with number of splits

The parameters of the split function (threshold, coefficients/dimension) are randomly selected `num_split` times and the configuration which maximizes a criterion, in this case Information Gain (**IG**), is chosen. Interestingly, as summarized in Table 1, simpler weak learners (axis-aligned and linear) excel for small values of `num_split`, while the more sophisticated learners do better for larger numbers of random splits. This is a consequence of the aforementioned flexibility of sophisticated learners. Therefore, they are effective for large `num_split`, however, they add a computational burden on the model, as shown in Figure 18, and as a result for the remainder of this section, we use **axis-aligned weak learners**.

1.3. Growing the Tree

Using the axis-aligned weak learner and for fixed `num_split`=5, the data is split recursively between the left and right child nodes until the complete decision tree is grown, whose complete graph is provided in Figure 19 (including the weak learner parametrization at each node). Of particular interest are the leaf nodes, the distributions of which are visualized in Figure 3. We observe that for the majority, the leaf node contains a single class. In some cases, however, such as in leaves 4 & 5, the leaf distributions contain more than one class. This interesting result can be attributed to the difficulty in perfectly separating the different data classes, especially around the center of the spiral where the data is tightly clustered.

[1] The VC dimension of a n^{th} order polynomial kernel is $n + 1$, where $n_{axis-aligned} = 0$ and $n_{cubic} = 3$.

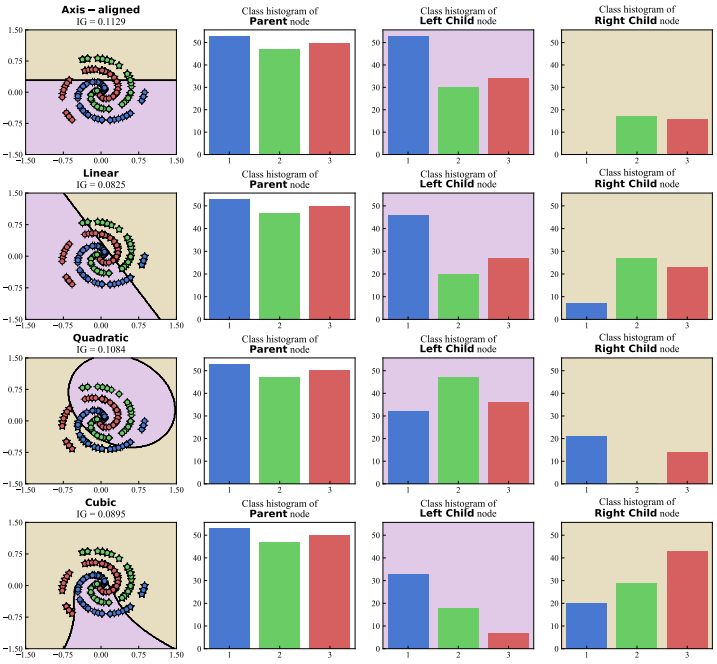


Figure 2. Root node best split for `num_split`=5, label class distributions of parent, left and right child respectively.

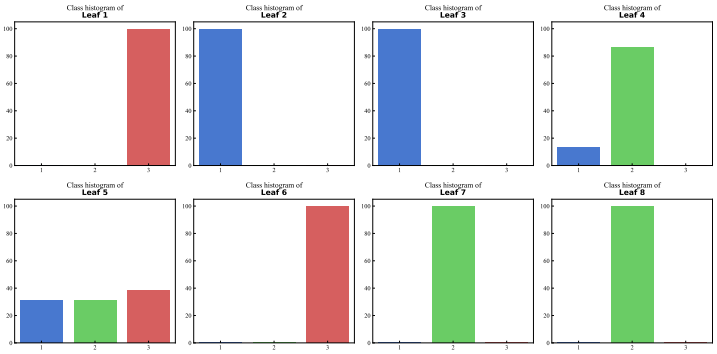


Figure 3. Class distributions of leaf nodes

1.3.1 Stopping Criteria

DT classifiers have the capacity to store every training sample in a single node, which would be catastrophic for generalization because of overfitting. Consequently, early stopping of the training/splitting process is vital when growing a tree. We considered maximum tree depth, d_{max}, which prevents the DT to grow indefinitely, allowing at most $2^{d_{max}} - 1$ nodes. Additionally, the minimum number of samples per node was thresholded, preventing nodes having too few samples. Finally, a minimum value of information gain was set, below which further splitting is not allowed. Importantly, we do not assign values to these stopping criteria here, but rather we cross-validate them for the particular application we are dealing with.

2. Evaluating The Decision Forest

In this section, the performance of the RF classifier is evaluated on a few sample test points as well as on a dense 2D grid. For illustration purposes, the trained RF consists of 3 decision trees, however, in the following subsections, the impact of this hyperparameter on the performance is validated.

Firstly, the four test points with coordinates $(x, y) = \{(-0.5, -0.7), (0.4, 0.3), (-0.7, 0.4), (0.5, -0.5)\}$ are classified.

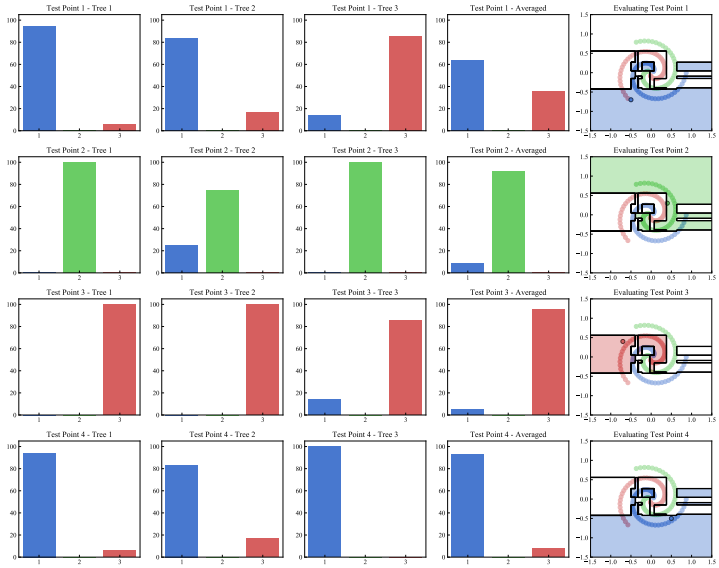


Figure 4. Classification of test points (one per row), class distributions of leaf nodes (columns 1-3), averaged class distribution (column 4) and decision boundaries of selected class (column 5).

The class distributions of the leaf nodes at which the test points arrive and the respective averaged class distributions are visualized in Figure 4. In particular, we note that for most of the points, the RF classifies them as one may intuitively have done. This is supported by the fact that the dominant class in the leaf nodes is consistent across the three trees. In one case, however, for the point $(x, y) = (-.5, -.7)$, there is an apparent misclassification as the point should have been intuitively classified as the red class. This inaccuracy may be attributable to the fact that this RF contains only 3 decision trees, which tend to overfit the training data, since the averaging effect of the committee machine is not fully exploited.

In the following subsections we explore the effect of various hyperparameters on the performance of the RF classifier, using **Leave-One-Out Grid Search Cross-Validation**. The exact algorithm is provided in Algorithm 1. We report the cross-validation error and use the configuration of parameters that minimizes it. We also show the decision contours for the 2D dense grid $(x, y) \in [-1.5, 1.5]^2$.

2.1. Maximum Tree Depth

As detailed in section 1.3, one of the stopping criteria we employ is the maximum depth of the tree. We vary the maximum tree depth of the decision forest, while the rest of the hyperparameters are kept fixed, and illustrate the decision boundaries in Figure 5a. We note that for shallower trees, (i.e `max_depth` = 2), the classification is inaccurate. We postulate that this is because the RF **underfits** the data, since with such a shallow depth, it is not able to split the data effectively. On the other hand, for deeper trees (i.e `max_depth` = 7, 11) the training dataset is **overfitted**, as seen by the numerous noisy spikes around the decision boundaries. Additionally, deeper trees are computationally more expensive to train. Consequently, we find that moderate tree depths (i.e `max_depth` = 5) are preferred, as they provide better generalization results are computationally more efficient.

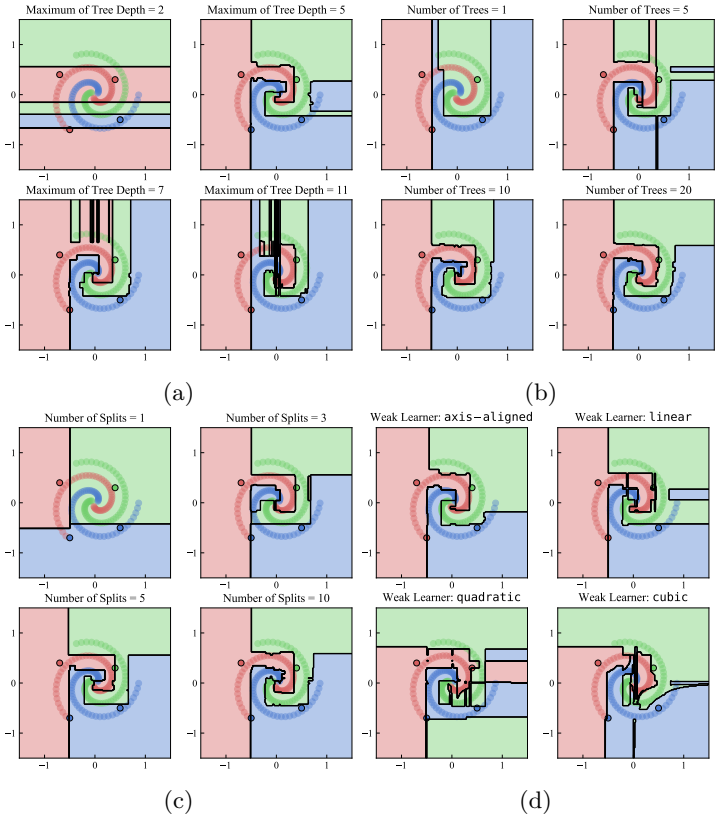


Figure 5. Decision Boundaries for different (a) Maximum Tree Depths, underfitting for shallow trees (i.e `max_depth`=2) and overfitting for very deep trees (i.e `max_depth`=7, 11) (b) Number of Trees, a moderate value (i.e `num_trees`=10) exploits the uncorrelation of the constituent DTs with reasonable time complexity (c) Number of Splits, controlling the degree of randomness, for (i.e `num_splits`$\geq$5) performance plateaus while training time complexity increases linearly (d) Weak Learners, when keeping the number of splits small (`num_splits` = 5), the `axis-aligned` learner outperforms the more sophisticated ones.

2.2. Number of Trees

When evaluating the test points earlier, we used an RF with 3 trees. In practice, this hyperparameter is also cross-validated. Again, we train and test the RF while varying the number of trees to examine the effect on the classification, the results of which are shown in Figure 5b. In theory, the advantage of a decision forest over a decision tree is that it is an ensemble of decorrelated trees, which may themselves overfit the data. Given enough constituent trees, however, individual tree inaccuracies are averaged out to improve generalization and robustness. This can be corroborated by Figure 5b. We can see that for an RF consisting of a single tree, the model tends to overfit the data, as seen by the gap in the blue class. As we increase the number of trees, we notice that the generalization results are improving. However, as with the maximum tree depth, having a higher number of trees entails more intensive computations. Thus, due to this trade-off, we select a moderate number of trees, namely 10, as this provides good generalization and affordable computation.

2.3. Degree of Randomness

Correlation between DTs of the RF is also broken by random splits, as introduced in subsection 1.2. In this subsection, we quantify the impact of this hyperparameter, number of splits[2], which controls the degree of randomness. According to Figure 5c, unsurprisingly, a small number of splits (i.e `num_splits` = 1, 3) leads to **underfitting** while larger values (i.e `num_splits` = 10) add complexity without an obvious performance boost. As a result, `num_splits` = 5 is chosen, maximizing accuracy with a reasonable time complexity.

Toy Spiral Dataset: Random Forest Model		
Classifier	Weak Learner	axis-aligned
	Maximum Tree Depth	5
	Number of Trees	10
	Minimum Samples at Node	5
	Number of Splits	5
	Information Gain Threshold	0.01
Score	Training Accuracy	100 %
	Testing Accuracy	99.3 %

Table 2. Random Forest model hyperparameters and performance summary on Toy Spiral dataset.

2.4. Other Hyperparameters

Additionally, the two remaining stopping criteria are cross-validated; the minimum number of samples in a leaf node and the minimum value of information gain. Finally, the weak-learner split function is validated. The qualitative results are provided in Figures 20a, 20b and 5d, respectively.

The final configuration of hyperparameters and the performance summary are provided in Table 2. We note that on average 1 out of 150 points is mis-classified, usually one close to the origin, where the three classes are almost overlapping.

3. Image Categorization of `Caltech 101`

Now the performance of the RF classifier is evaluated on a real dataset, the `Caltech 101` dataset. It consists of 10 classes, where we randomly select 15 images per class for training and testing, respectively. Instead of raw pixels, SIFT [2] affine transformations are applied and the extracted descriptors are used. Then a codebook is constructed via either k-Means clustering or Random Forest embeddings. The resulting histogram representations, Bag-of-(Visual)-Words (**BoW**), are subsequently used as features for the classification.

3.1. k-Means Codebook

The codebook construction algorithm using k-Means clustering is provided in Algorithm 2 and the `Python` code snippet in Listing 1.

3.1.1 Vector Quantisation Process

In Figure 6, we demonstrate training and testing image examples in original, SIFT and BoW representations. Surprisingly, the original images can have varying sizes, different widths and heights, while their BoW representation is consistent across all images, allowing us to use a single classifier for all images. The exact dimensionality of the BoW representation is controlled by the number of centroids (**vocabulary size**) used for k-means clustering, covered in subsection 3.1.2.

[2]All experiments are carried out using an **axis-aligned** split function.

SIFT is used for descriptor extraction[3], implemented using the `OpenCV` toolbox. As illustrated in Figure 6b, SIFT is applied to gray-scale images, and returns 128-dimensional descriptors. Using the training set images, we collect all the descriptors and uniformly sample $100K$ of them[4], which we use to train a k-means clustering model.

In terms of the k-means implementation, we use Mini-Batch k-means clustering [3] to improve the computational performance. This variant uses mini-batches (i.e. randomly drawn subsets) and assigns each element to its nearest centroids. The cluster centroids are subsequently updated taking into account the average of all points assigned to that centroid. The algorithm iterates between these steps until convergence or a certain iteration threshold is reached. The algorithm also uses k-means++ [1] for cluster centroid initialization (time complexity $\mathcal{O}(logk)$), further improving computational load. In Figure 21, we demonstrate the computational performance of the standard k-means over the mini-batch variant to show that the latter is indeed superior in that respect.

The BoW model is inspired by Natural Language Processing applications, where the codewords (in our case represented by the centroids) are given by the vocabulary, eliminating the information loss introduced by clustering in the visual codebook construction. Therefore, having calculated the cluster centers, **vector quantization** can be done to retrieve the BoW representation. We assign to every descriptor in an image the nearest centroid obtained from the k-means clustering. We then count the occurrences of each centroid, which ultimately produces the BoW histogram representation. The vector quantization is controlled by the number of clusters, and therefore centroids, used.

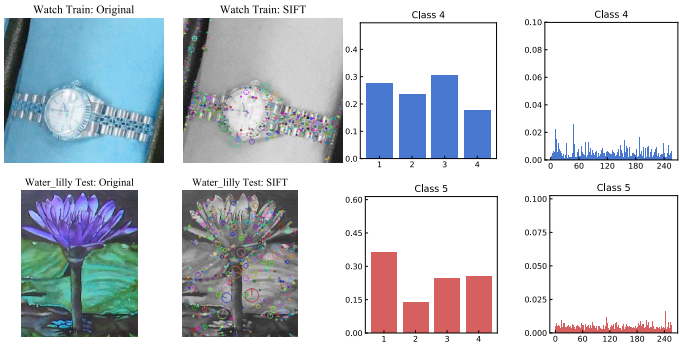
3.1.2 Vocabulary Size

As a consequence, the BoW representation is highly dependent on the **vocabulary size**, k, (i.e. the number of clusters). If k is chosen too small, generalization will be poor as there will not be enough discrimination between the various classes (underfitting). Conversely, if k is too large, generalization may improve at the cost of computational performance. Thus, the choice of k is a hyperparameter that should be validated. In Figure 6c, 6d the BoW representation of a train and test example is provided for $k = 4$ and $k = 256$, demonstrating the difference between the two feature vectors for different vocabulary sizes.

3.2. Random Forest Classifier

Establishing the transformation from pixels to BoW enables us to actually train the RF and test its performance. Given the 10 classes, we train the RF classifier using the BoW-represented training images. Certainly, as seen in Section 2, there are a variety of hyperparameters that strongly influence the classification accuracy of the model. In what follows, we discuss the effects of these hyperparameters on the model and their optimal values. Importantly, we conduct Leave-One-Out cross-validation, according to Algorithm 1. The selection of the hyperparameters is made based on the cross-validation error, though the training and testing errors are also provided.

[3]invariant to affine transformations.
[4]without replacement



(a) Original (b) SIFT (c) $K = 4$ (d) $K = 256$

Figure 6. Training (top) and testing (bottom) images. (a) Original image and (b) SIFT descriptors (c) BoW using $K = 4$ (d) BoW using $K = 256$.

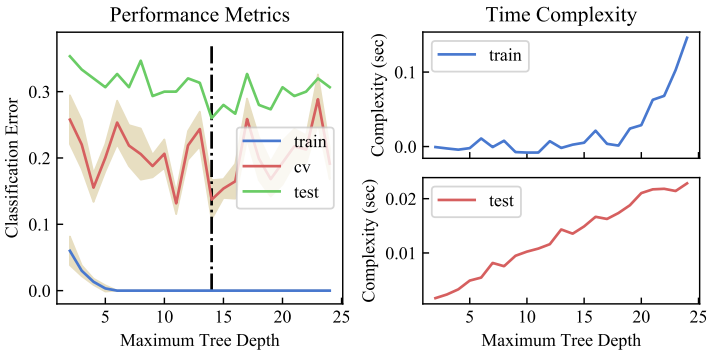


Figure 7. Classification accuracy (left) and time complexity (right) for various tree depths.

3.2.1 Maximum Tree Depth

As in Subsection 2.1, we consider the effects of maximum tree depth on classification. Figure 7 highlights the accuracy and timing performance of the model for various values of `max_depth`. In terms of accuracy, classification is compromised at both extremes; for smaller values, the RF underfits the data due to an inability to split the data effectively and for larger values, the RF begins to overfit as the leaf nodes contain fewer points. Concerning the timing efficiency, unsurprisingly timing increases for higher values of `max_depth`, both for training (exponentially) and testing (linearly) steps. Thus, as shown in Figure 7, we choose `max_depth` = 14 to achieve good generalization and timing performance.

3.2.2 Number of Trees

In Figure 8 the impact of number of trees is considered. As expected the classification accuracy increases with an increasing number of trees, since individual tendencies of trees to overfit are averaged out by the RF. This overfitting tendency is particularly dominant for lower numbers of trees as no averaging effect is observed, explaining the initial steepness of the curve and the following plateau. However, time complexity increases linearly with the number of trees, but both training and testing can be parallelized. Overall, `num_trees` = 900 is selected, achieving decent timing performance ($\approx$ 2s, $\approx$ 0.1s for training and testing, respectively), with a marginal decrease in accuracy.

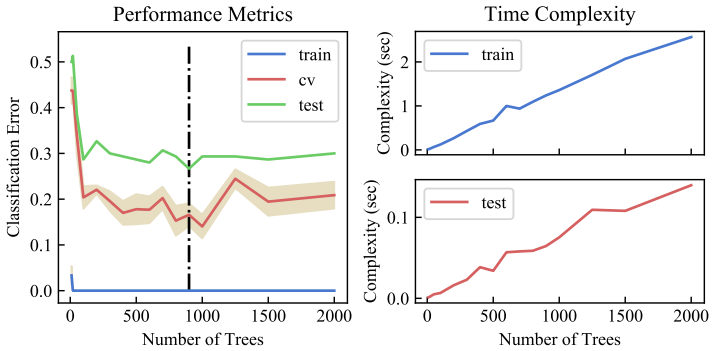


Figure 8. Classification accuracy (left) and time complexity (right) for various number of tree.

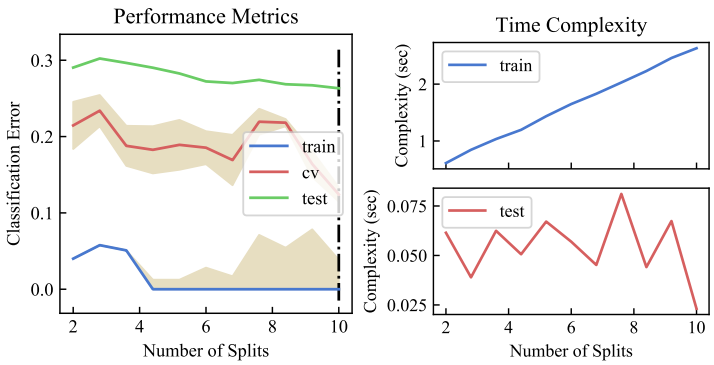


Figure 9. Classification accuracy (left) and time complexity (right) for various number of splits (degree of randomness).

3.2.3 Degree of Randomness

The degree of randomness is controlled by the number of random splits. In Figure 9 we observe that the accuracy improves as the number of splits increases, plateauing at `num_splits` = 7, while training complexity increases linearly, as expected.

3.2.4 Weak Learners

Axis-aligned, two-pixel test, linear, quadratic and cubic kernels are considered for node splitting. Figure 10 demonstrates the effect of the weak learner on accuracy and timing. Noticeably from the figure, we observe that the best cross-validation accuracy is achieved for the two-pixel learner, for which training and testing accuracy remain relatively stable. More interestingly, however, the computational timing performance of the model deteriorates with more sophisticated models, as expected since such models are intrinsically more computationally intensive than their simpler counterparts. Thus, we choose two-pixel weak learners for the model.

3.2.5 Vocabulary Size

Of particular relevance in the image classification problem is the vocabulary size of the BoW used. Figure 11 highlights the accuracy and timing properties of the vocabulary size. We notice that there is a steep initial decrease in the various error metrics for increasing vocabulary size small than approximately 50. This is supported by the discussion in subsection 3.1, as for small vocabulary sizes, the model underfits the data due to lack of discrimination between the various classes. We notice that after the minimum observed at a size of 50, there

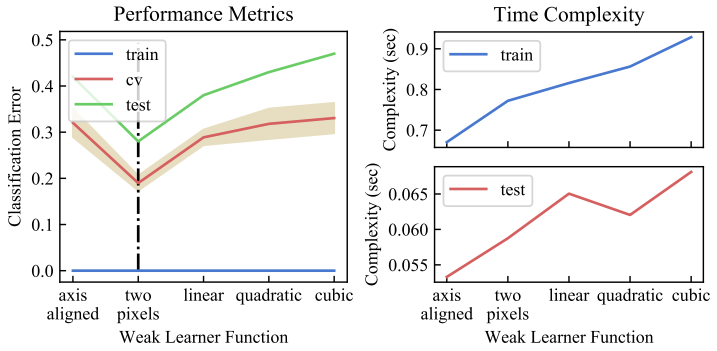


Figure 10. Accuracy and timing for various weak learners

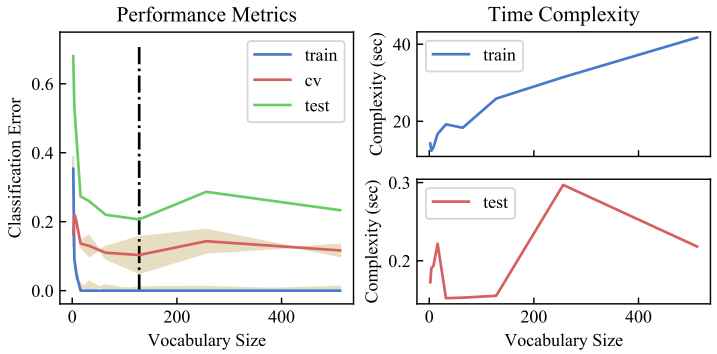


Figure 11. Classification accuracy (left) and time complexity (right) for various vocabulary sizes.

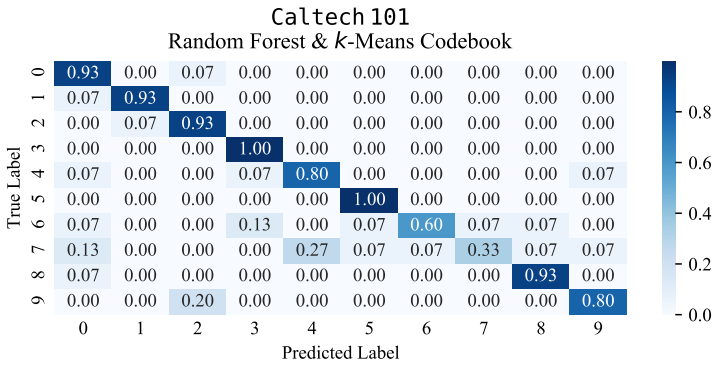


Figure 12. Confusion Matrix: Random Forest Classifier on k-Means codebook, very low performance (33%) on "watch" class 7.

is a gradual increase in the cross-validation and test errors for higher sizes. We postulate that this stems from the RF's overfitting due to the larger dimensionality of the data. Concerning the timing performance, we notice that execution time for both training and testing predictably increases with larger vocabulary sizes.

3.2.6 Optimal Configuration

The selection of the optimal hyperparameters is summarized in Table 3. The selected model achieves 83.4% accuracy on the test set, and its confusion matrix and success/failure examples of classification are provided in Figures 12 and 13, respectively.

According to the confusion matrix, the classifier poorly recognizes watches (class 7), treating them as wheelchairs (class 4) or ticks (class 1), intuitively because of their round shape (watch face, wheel and tick tail).

Figure 13. Random Forest Classifier on k-Means codebook: (top) examples of worst accuracy on "watch" class, (bottom) examples of correct classifications of respective classes.

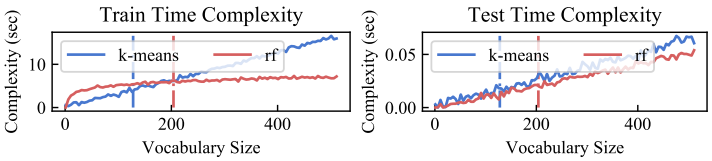


Figure 14. Vector Quantisation Process Complexity comparison for k-Means and RF Codebook.

3.3. Random Forest Codebook

3.3.1 Vector Quantisation Process

In this section we construct a Random Forest Codebook from the SIFT descriptors of the images, see Listing 2 for the `Python` code snippet. Unlike k-means, the RF codebook exploits the valuable discriminative information in the descriptor space, constructed in a supervised manner.

During training, labels are taken into account for the RF construction and the descriptors are assigned to"cluster", determined by the leaves they end up in. Each descriptor reaches T leaves for an RF with T DTs. This is an encoding process, where the descriptor space is mapped to leaf embeddings. The BoW representation is then obtained by collecting all leaf embeddings of an image, after transforming all of its SIFT descriptors. The exact number of leaves, or **vocabulary size** K, of the RF codebook is indirectly controlled and only an upper bound can be determined, equal to $K \leq \texttt{num\_trees} \times 2^{\texttt{max\_depth}-1}$.

3.3.2 Optimal Configuration

There are now two Random Forest models; one feature transformer and one classifier, both of which have hyperparameters to tune, similar to sections 2, 3.2. Leave-One-Out Grid Search Cross-Validation is used again for optimizing the hyperparameters. End-to-end validation is performed, testing all possible combinations of the hyperparameters for both the classifier and the feature transformer (codebook) at the same time. The optimal model parameters are given in Table 3.

The model achieves 77.3% classification accuracy on the test set and poorly classifies umbrella images, class 6 as seen in Figures 15 and 16. Additionally, we observe a prediction bias towards class 5, yin-yang images, since most mis-classified images are labeled as such by the model.

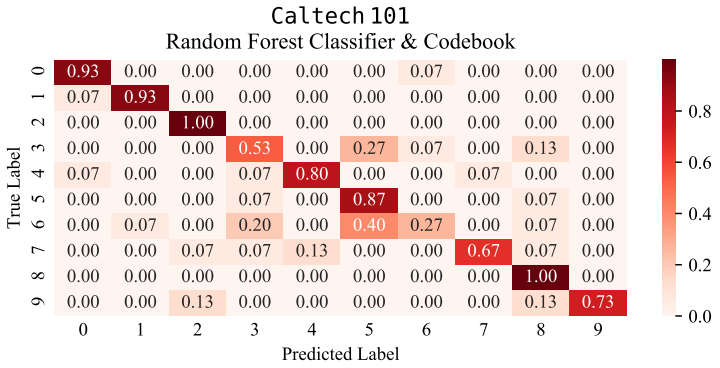


Figure 15. Confusion Matrix: Random Forest Classifier on Random Forest codebook, low performance (27%) on "umbrella" class 6.

`Caltech 101` Dataset: Random Forest Models			
Codebook	Algorithm	k-**Means**	**RF**
	Number of Centroids (k)	128	-
	Number of Trees	-	10
	Weak Learner	-	two-pixel
	Maximum Tree Depth	-	5
	Number of Splits	-	5
	Vocabulary Size	128	≈204
Classifier	Weak Learner	two-pixel	two-pixel
	Maximum Tree Depth	14	7
	Number of Trees	900	200
	Minimum Samples at Node	5	5
	Number of Splits	7	7
Score	Training Accuracy	100 %	100 %
	Validation Accuracy	86 %	80.6 %
	Testing Accuracy	83.4 %	77.3 %
Time	Vector Quantisation	23s	12.5s
	Training	1.7s	0.8s
	Testing	0.07s	0.06s

Table 3. Random Forest model hyperparameters and performance summary on `Caltech 101` dataset.

3.3.3 Comparison

Finally, we compare the two BoW implementations. The complete comparison matrix is summarized in Table 3. As far as accuracy is concerned, we note that the k-Means codebook results in a better recognition accuracy, 83.4 % over 77.3 % for RF. Nonetheless, the rich and compressed discriminative information that the RF codebook encodes results in a significantly simpler and therefore faster classifier, almost 5 times smaller than the one developed with the k-Means codebook. More precisely, the RF codebook uses an optimal 200 DTs and maximum depth 7 RF, trained in 12.5 seconds while the k-means uses a 900 DTs and maximum depth 14 RF, trained in 23 seconds. Moreover, we observe that the RF codebook scales much better than the k-Means variant, as suggested by Figure 14, where we empirically verify the logarithmic training complexity of RF, $\mathcal{O}(logK)$), and the linear complexity of k-Means, $\mathcal{O}(K)$, where K is the vocabulary size.[5]

[5]The number and dimensionality of the descriptors are the same for both methods.

References

[1] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[2] D. G. Lowe. Object recognition from local scale-invariant features, 1999. [Online]. Available: http://www.cs.ubc.ca/~lowe/papers/iccv99.pdf. [Accessed: 17- Feb- 2018].

[3] D. Sculley. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pages 1177–1178. ACM, 2010.

Appendix

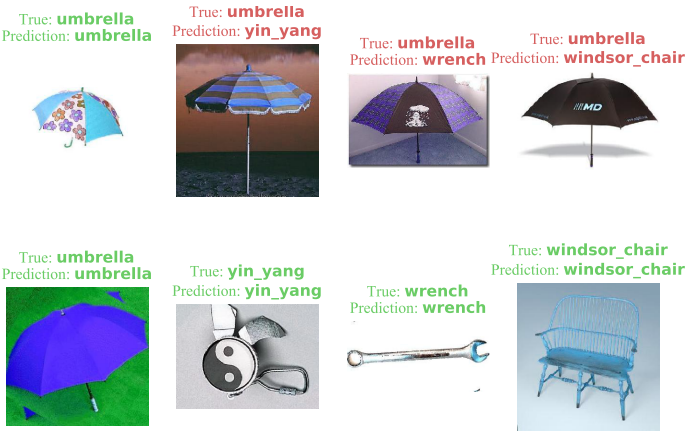


Figure 16. Random Forest Classifier on Random Forest codebook: (top) examples of worst accuracy on "umbrella" class, (bottom) examples of correct classifications of respective classes.

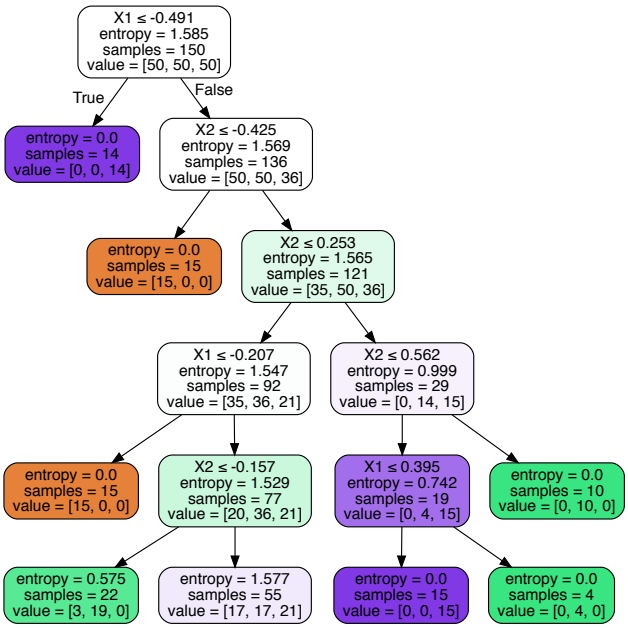


Figure 19. Decision Tree graph for leaf nodes in figure 3.

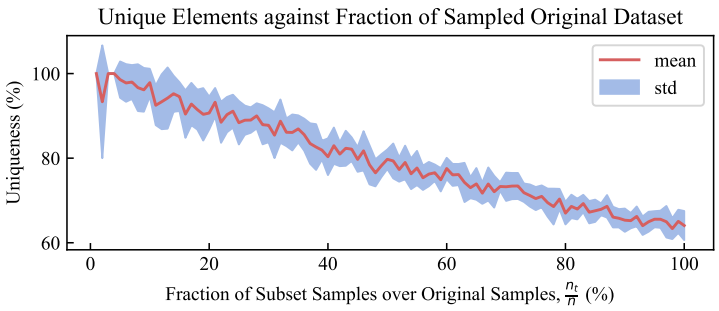


Figure 17. Bagging: Percentage **uniqueness** of points using bootstrap sampling with replacement, variation with **fraction** of the total set of samples.

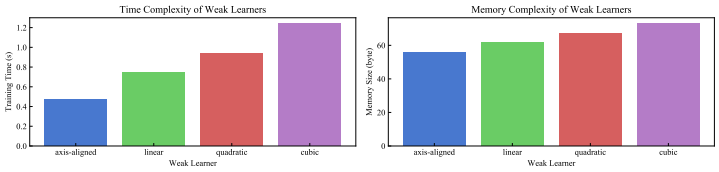
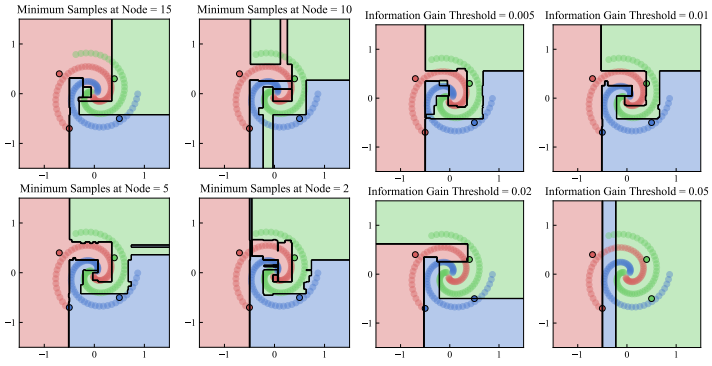


Figure 18. Weak learners mean time & memory complexity.

Algorithm 1: Leave-One-Out Cross-Validation

input : dataset $\mathcal{S}_0$
set of fixed hyperparameters $\boldsymbol{\theta_{fixed}}$
list of candidate hyperparameters $\boldsymbol{\Phi}$
output: optimal hyperparameter $\boldsymbol{\phi_*} \in \boldsymbol{\Phi}$

shuffle $\mathcal{S}_0$ in-place
initialize accuracy array, $\texttt{accuracy}(\boldsymbol{\phi}_i) \leftarrow \mathbf{0}$
foreach $\boldsymbol{\phi}_i \in \boldsymbol{\Phi}$ **do**
  accuracy$(\boldsymbol{\phi}_i) \leftarrow 0$
  foreach datapoint $s_i \in \mathcal{S}_0$ **do**
    fit classifier $\texttt{rf}_{\boldsymbol{\phi}_i}$ on $\{\mathcal{S}_0 - s_i\}$ using $[\boldsymbol{\theta_{fixed}};\ \boldsymbol{\phi}_i]$
    get test accuracy $\texttt{acc}_{s_i}$ on s_i
    $\texttt{accuracy}(\boldsymbol{\phi}_i) \leftarrow \texttt{accuracy}(\boldsymbol{\phi}_i) + \texttt{acc}_{s_i}$
  end
  normalize accuracy, $\texttt{accuracy}(\boldsymbol{\phi}_i) \leftarrow \frac{1}{|\mathcal{S}_0|}$
end
select optimal parameter $\boldsymbol{\phi_*} = \underset{\boldsymbol{\Phi}}{\text{argmax}}\ \texttt{accuracy}(\boldsymbol{\Phi})$



(a) (b)

Figure 20. Decision Boundaries for different (a) Minimum Number of Samples at Node, a moderate value (i.e `min_samples_split`=10) exploits the uncorrelation of the constituent DTs with reasonable time complexity (b) Information Gain Thresholds, a moderate value (i.e `ig_threshold`=0.001)avoids underfitting and overfitting, present to high and low values, respectively.

Algorithm 2: k-Means codebook generation

input : vocab size K
number of descriptors N
training & testing images, S_{train} & S_{test}
output: BoW representation of S_{train} & S_{test}

initialize $\mathcal{D}$ as empty set;
foreach $\mathcal{I}_i \in S_{\text{train}}$ **do**
  extract SIFT descriptors for $\mathcal{I}_i$ and add to $\mathcal{D}$
end
pick N randomly selected descriptors from $\mathcal{D}$;
train k-means on chosen descriptors, with $k = K$;
foreach $\mathcal{I}_i \in S_{\text{train}} \cup S_{\text{test}}$ **do**
  transform $\mathcal{I}_i$ from pixels to BoW representation using trained k-means
end

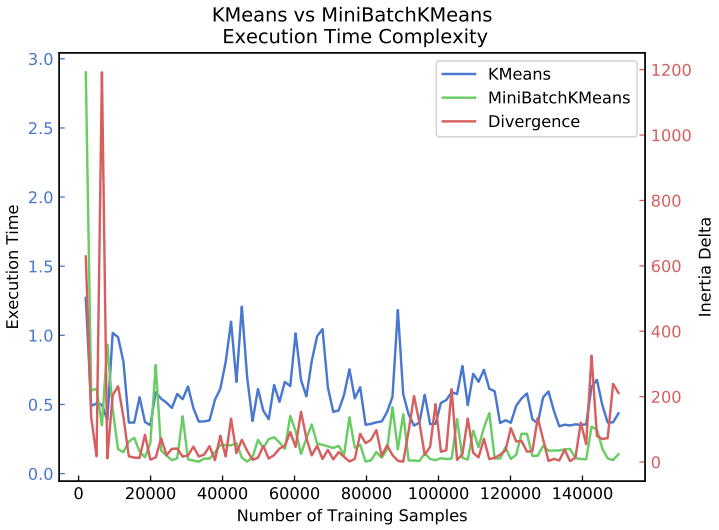


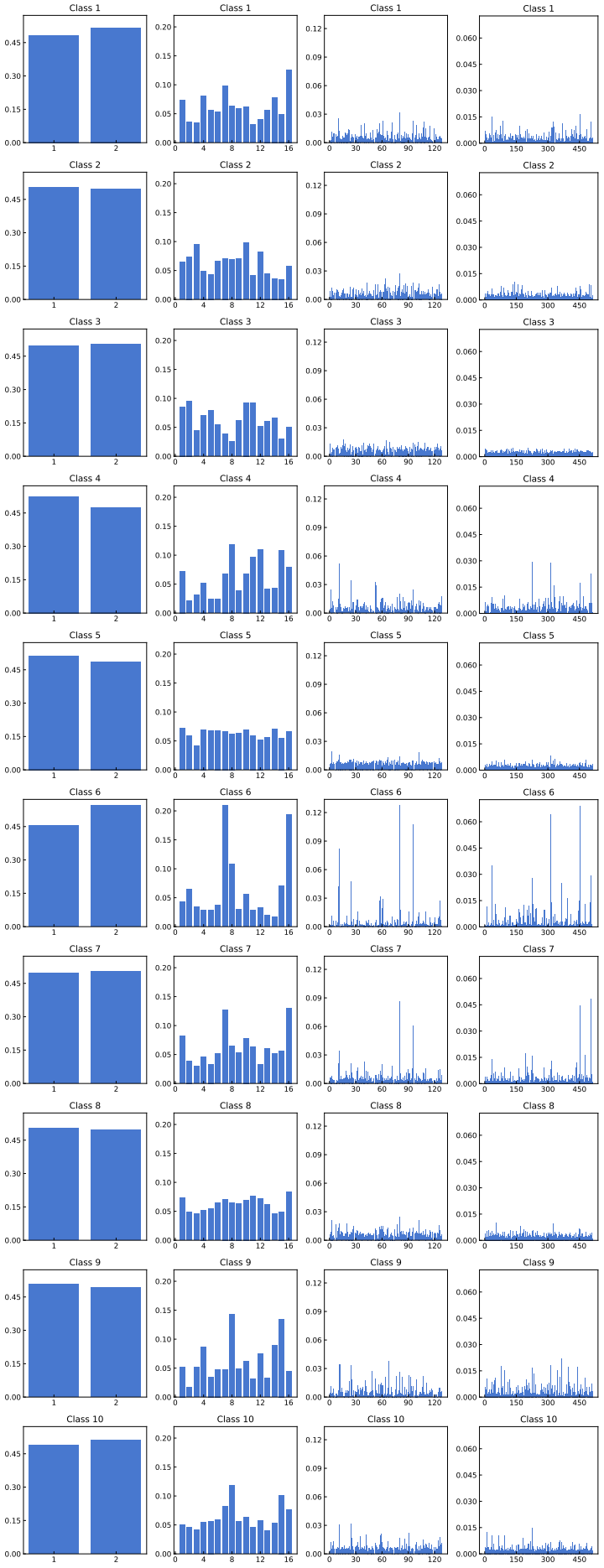
Figure 21. Computational performance of standard k-means over mini-batch variant.

```python
import os
import glob

from collections import defaultdict

import numpy as np
import cv2

from sklearn.cluster import MiniBatchKMeans

def KMeans_Codebook(num_features, num_descriptors):
    # root folder with images
    folder_name = 'data/Caltech_101/101_ObjectCategories'
    # list of folders of images classes
    class_list = os.listdir(folder_name)
    # macOS: discart '.DS_Store' file
    if '.DS_Store' in class_list:
        class_list.remove('.DS_Store')

    # SIFT feature extractor
    sift = cv2.xfeatures2d.SIFT_create()

    # TRAINING
    # list of descriptors
    descriptors_train = []
    raw_train = defaultdict(dict)
    # iterate over image classes
    for c in range(len(class_list)):
        # subfolder pointer
        sub_folder_name = os.path.join(folder_name, class_list[c])
        # filter non-images files out
        img_list = glob.glob(os.path.join(sub_folder_name, '*.jpg'))
        # shuffle images to break correlation
        np.random.shuffle(img_list)
        # training examples
        img_train = img_list[:15]
        # iterate over image samples of a class
        for i in range(len(img_train)):
            # fetch image sample
            raw_img = cv2.imread(img_train[i])
            img = raw_img.copy()
            # convert to gray scale for SIFT compatibility
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            # apply SIFT algorithm
            kp, des = sift.detectAndCompute(gray, None)
            # store descriptors
            raw_train[c][i] = des
            for d in des:
                descriptors_train.append(d)
    # NumPy-friendly array of descriptors
    descriptors_train = np.asarray(descriptors_train)
    # random selection of descriptors WITHOUT REPLACEMENT
    descriptors_random = descriptors_train[np.random.choice(
        len(descriptors_train), min(len(descriptors_train),
                                    num_descriptors),
        replace=False)]

    # TESTING
    raw_test = defaultdict(dict)
    # iterate over image classes
    for c in range(len(class_list)):
        # subfolder pointer
        sub_folder_name = os.path.join(folder_name, class_list[c])
        # filter non-images files out
        img_list = glob.glob(os.path.join(sub_folder_name, '*.jpg'))
        # testing examples
        img_test = img_list[15:30]
        # iterate over image samples of a class
        for i in range(len(img_test)):
            # fetch image sample
            raw_img = cv2.imread(img_test[i])
            img = raw_img.copy()
            # convert to gray scale for SIFT compatibility
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            # apply SIFT algorithm
            kp, des = sift.detectAndCompute(gray, None)
            # store descriptors
            raw_test[c][i] = des

    codebook_algorithm = MiniBatchKMeans(n_clusters=num_features,
                                         init='k-means++',
                                         batch_size=num_descriptors//100
                                         ).fit(descriptors_random)

    # vector quantisation
    data_train = np.zeros(
        (len(class_list)*15, num_features+1))

    for i in range(len(class_list)):
        for j in range(15):
            # determine centers distribution
            idx = codebook_algorithm.predict(raw_train[i][j])
            # set features
            data_train[15 *
                       (i)+j, :-1] = histc(idx,
                                           range(num_features)) / len(idx)
            # set label
            data_train[15*(i)+j, -1] = i

    # vector quantisation
    data_query = np.zeros(
        (len(class_list)*15, num_features+1))

    for i in range(len(class_list)):
        for j in range(15):
            # determine centers distribution
            idx = codebook_algorithm.predict(raw_test[i][j])
            # set features
            data_query[15 *
                       (i)+j, :-1] = histc(idx,
                                           range(num_features)) / len(idx)
            # set label
            data_query[15*(i)+j, -1] = i

    return data_train, data_query
```

Listing 1. k-Means Codebook Construction.

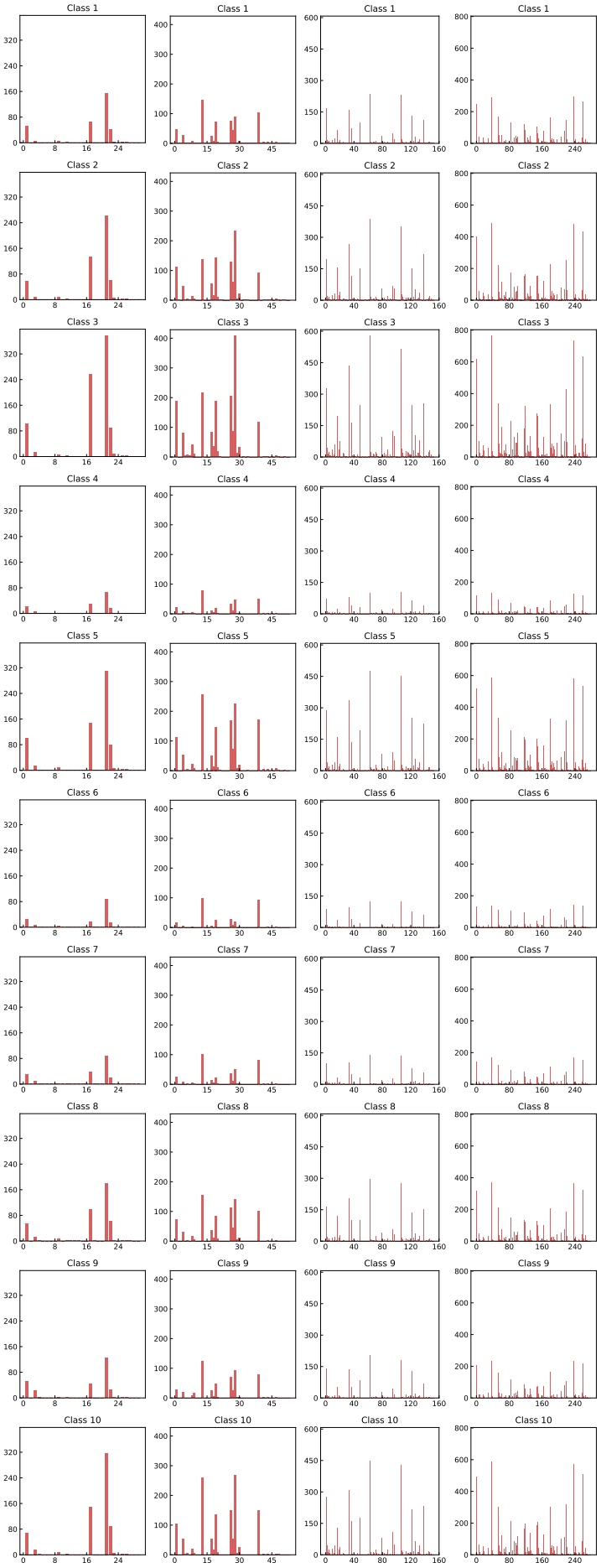
```python
import os
import glob

from collections import defaultdict

import numpy as np
import cv2

from sklearn.ensemble import RandomTreesEmbedding

def RandomForest_Codebook(num_features, num_descriptors):
    # root folder with images
    folder_name = 'data/Caltech_101/101_ObjectCategories'
    # list of folders of images classes
    class_list = os.listdir(folder_name)
    # macOS: discart '.DS_Store' file
    if '.DS_Store' in class_list:
        class_list.remove('.DS_Store')

    # SIFT feature extractor
    sift = cv2.xfeatures2d.SIFT_create()

    # TRAINING
    # list of descriptors
    descriptors_train = []
    raw_train = defaultdict(dict)
    # iterate over image classes
    for c in range(len(class_list)):
        # subfolder pointer
        sub_folder_name = os.path.join(folder_name, class_list[c])
        # filter non-images files out
        img_list = glob.glob(os.path.join(sub_folder_name, '*.jpg'))
        # shuffle images to break correlation
        np.random.shuffle(img_list)
        # training examples
        img_train = img_list[:15]
        # iterate over image samples of a class
        for i in range(len(img_train)):
            # fetch image sample
            raw_img = cv2.imread(img_train[i])
            img = raw_img.copy()
            # convert to gray scale for SIFT compatibility
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            # apply SIFT algorithm
            kp, des = sift.detectAndCompute(gray, None)
            # store descriptors
            raw_train[c][i] = des
            for d in des:
                descriptors_train.append(d)
    # NumPy-friendly array of descriptors
    descriptors_train = np.asarray(descriptors_train)
    # random selection of descriptors WITHOUT REPLACEMENT
    descriptors_random = descriptors_train[np.random.choice(
        len(descriptors_train), min(len(descriptors_train),
                                    num_descriptors),
        replace=False)]

    # TESTING
    raw_test = defaultdict(dict)
    # iterate over image classes
    for c in range(len(class_list)):
        # subfolder pointer
        sub_folder_name = os.path.join(folder_name, class_list[c])
        # filter non-images files out
        img_list = glob.glob(os.path.join(sub_folder_name, '*.jpg'))
        # testing examples
        img_test = img_list[15:30]
        # iterate over image samples of a class
        for i in range(len(img_test)):
            # fetch image sample
            raw_img = cv2.imread(img_test[i])
            img = raw_img.copy()
            # convert to gray scale for SIFT compatibility
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            # apply SIFT algorithm
            kp, des = sift.detectAndCompute(gray, None)
            # store descriptors
            raw_test[c][i] = des

    # K-Means clustering algorithm
    codebook_algorithm = RandomTreesEmbedding(
        n_estimators=num_features).fit(descriptors_random)

    n_out = codebook_algorithm.transform(raw_train[0][0]).sum(axis=0).shape[1]

    # vector quantisation
    data_train = np.zeros(
        (len(class_list)*15, n_out+1))

    for i in range(len(class_list)):
        for j in range(15):
            # set features
            data_train[15 * (i)+j, :-1] = codebook_algorithm.transform(
                raw_train[i][j]).sum(axis=0).ravel()
            # set label
            data_train[15*(i)+j, -1] = i

    # vector quantisation
    data_query = np.zeros(
        (len(class_list)*15, n_out+1))

    for i in range(len(class_list)):
        for j in range(15):
            # set features
            data_query[15 *
                       (i)+j, :-1] = codebook_algorithm.transform(
                raw_test[i][j]).sum(axis=0).ravel()
            # set label
            data_query[15*(i)+j, -1] = i

    return data_train, data_query
```

Listing 2. Random Forest Embeddings Codebook Construction.



(a) (b) (c) (d)

Figure 22. k-Means codebook centroids BoW representation: (a) $K = 2$ (b) $K = 16$ (c) $K = 128$ (d) $K = 512$.



(a) (b) (c) (d)

Figure 23. RF codebook BoW representation:
(a) `num_trees` = 1 (b) `num_trees` = 2 (c) `num_trees` = 5 (d) `num_trees` = 10.