Fast Discriminative Visual Codebooks using Randomized Clustering Forests

1. Introduction

In (1), an image classification method is described. More specifically, it addresses the so-called codebook construction step. Most image classification methods, when applied to an input image to classify, follow three steps:

1. Features are extracted from the input image.
2. The set of features is coded into a smaller vector using the visual codebook.
3. The code is then classified using a standard vector classifier.

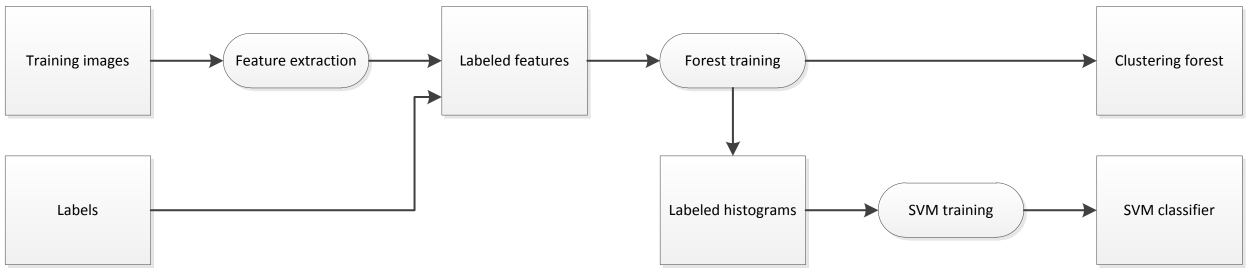
The article does not address the first and last steps but suggests the methods that could be used. Its main point is to describe the construction and usage of the visual codebook, based on extremely randomized clustering forests.

2. Algorithm

a. Overview

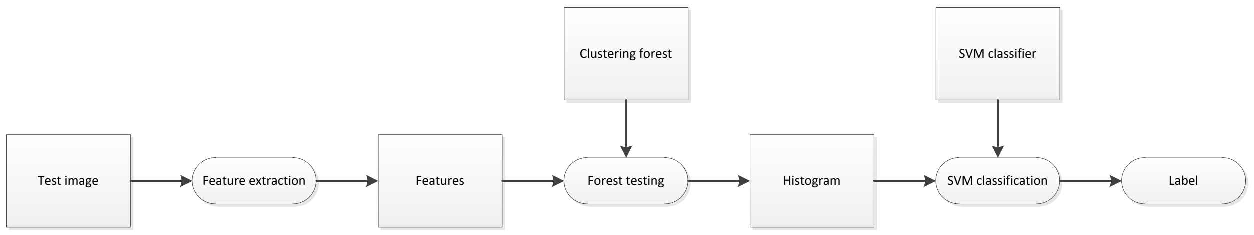
As in most classification algorithms, the proposed algorithm consists of two steps: training from a base of images which classification labels are known and testing, that is, using it to classify images which labels are unknown.

The training process is as follows:



After extracting features from the training images, a forest is trained with these features and the associated image labels; the same training features are coded by the forest to give labeled histograms that are then used to train an SVM classifier.

The testing process is as follows:



After extracting features from the test image, the features are submitted to the clustering forest. The resulting histogram is then fed to the SVM classifier from which we get the final label.

b. Feature extraction

The features are vectors extracted from the images. The paper suggests three types of features:

1. Fixed size square patches extracted at random scales and random positions, in the HSL color space, are reshaped as vectors.
2. Fixed size square patches extracted at random scales and random positions, in the HSL color space, are applied a 2D Haar wavelet transform and then reshaped as vectors.
3. SIFT key points are extracted and the associated descriptors are kept.

The article suggests using, for the square patches a width of 16 pixels, thus resulting in 768 dimension features for the HSL based features. It also suggests using 67 features per train image and 8000 features per test image.

When a mask exists that defines the zone of interest in the image, the features are extracted around points within this mask. For the patch based features, this is done explicitly. For the SIFT features, the image is convoluted with the mask, imposing 0 value outside of it. Due to the fact SIFT key points are extrema in a difference of Gaussians pyramid, the points outside of the interest area are naturally excluded.

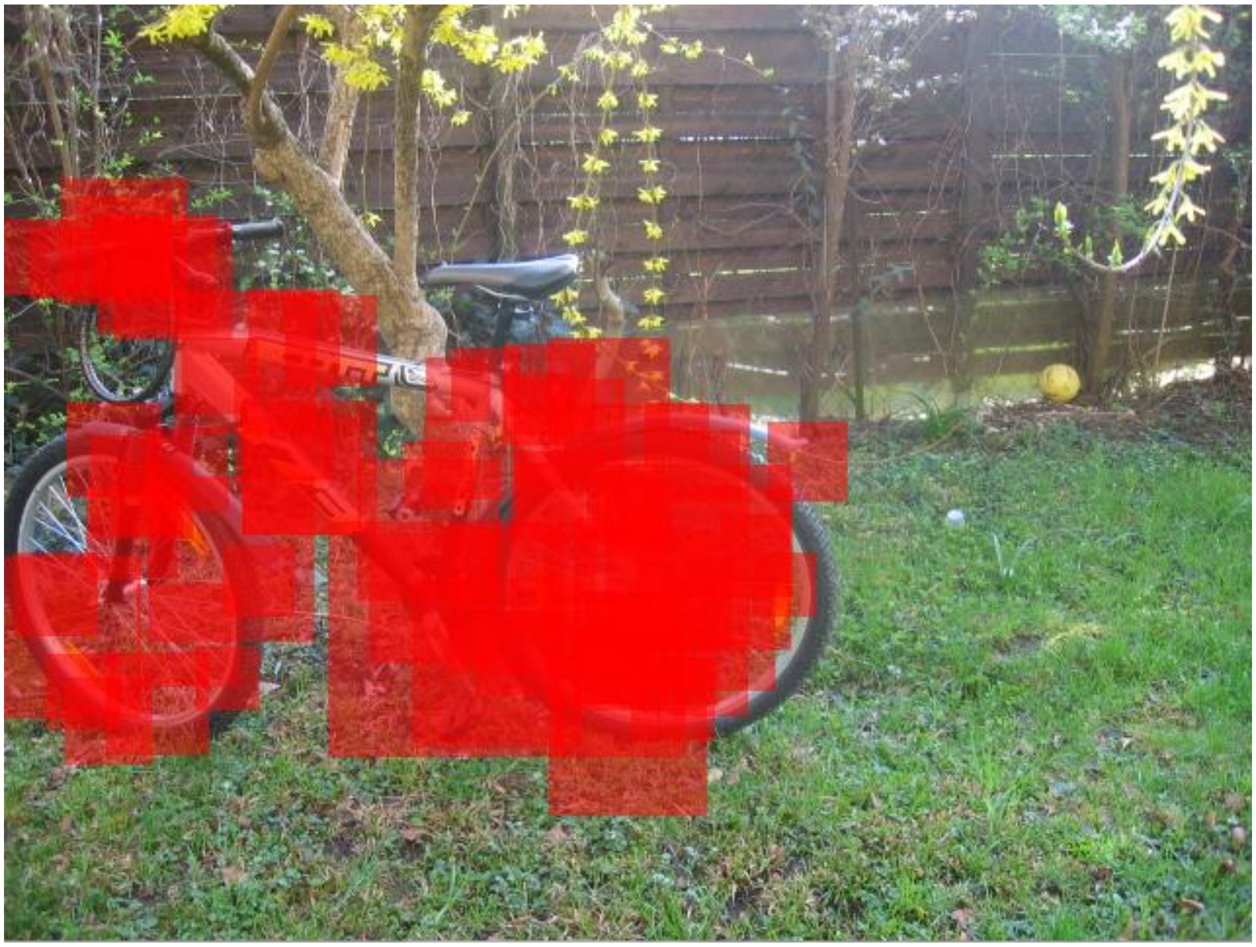


Figure 1 - Example of random patches extracted from an interest area and displayed using a plotting class built in the implementation.

c. Clustering forests

i. Using a clustering forest

A clustering forest is a set of clustering trees. These trees are binary trees. Each node that is not a leaf contains a binary test. When a feature is passed to the tree, the test checks whether one particular scalar component of the feature is bigger than a particular threshold. The result of the test determines which of its two child-trees the feature will be tested through next. Whenever a leaf is reached, the index of the leaf in the entire tree is kept.

When a set of features is tested through a tree, the result is a histogram, each bin corresponding to a leaf index and the content of each bin corresponding to the number of features that were associated with the corresponding leaf

When a forest is used, each tree results in a histogram; all the histograms are concatenated to result in a single one.

A leaf can also be associated to a label (cf. 2.c.ii, an unmixed leaf); another way to use the clustering forest, is to check whether a test feature reaches in each tree an unmixed leaf with a specific label.

ii. Training a randomized clustering forest

In order to build an efficient clustering tree from a set of labeled training features, we use a randomized method. The tree is built from the root to the leaves. At each node, a scalar feature component index and a threshold are chosen randomly following this iterative process:

1. A feature component index is chosen uniformly randomly. The maximum and minimum for this scalar feature component among the current set are measured.
2. A threshold is chosen uniformly randomly between the minimum and maximum.
3. The score of the partition implied by this binary test is evaluated as follows:
$$S = \frac{2I(L,P)}{H(L)+H(P)} = 2\left(1 - \frac{H(L,P)}{H(L)+H(P)}\right)$$
Where L is the label distribution among the current set, P is the distribution of the partition implied by the current binary test, I is the mutual information and H the entropy.
4. If the score is higher than a fixed value or if the maximum number of iterations has been reached, we stop. Otherwise, we start again at 1.

Whenever a set contains features with the same label (i. e. an unmixed set) or with the same feature values, it is considered indivisible and the node is a leaf (with no binary test). If the set is unmixed, its label is kept with the leaf.

The article does not precisely suggest any generic choice for the “high enough” score value and the maximum number of iterations. It however states that choosing 0.5 for the former and the feature dimensionality for the later should be a strong reinforcement of the discriminative power of the trees.

The trees are grown independently. In order to control the number of their leaves, they are pruned after being grown. While they have more leaves than desired, the final node (i. e. a node which child-trees are both leaves) with the lowest score is removed.

The algorithm suggests, for 150 input images and the previously mentioned parameters, pruning the trees so they have each 1000 leaves.

d. Final classifier

The article does not address the details of the last step of the classification process but only suggests using a common classification method such as a linear SVM taking as input the binarized histograms.

However, one could question the relevance of a linear classifier for such data. Obviously, the linear SVM should incorporate a bias. Another perhaps more relevant choice would be to use a linear SVM with a homogenous kernel map approximating a more appropriate similarity measure for histograms, such as the Chi2 measure.

We did not use this last option because of performance issues, the use of linear kernel maps approximating nonlinear kernels increasing significantly the dimensionality.

We chose to train one linear biased SVM per label from the training features. For each label, we process as follow. Each training image is tested through the forest and its binarized histogram is associated to 1 if the image label is the current label and -1 otherwise. This data is used to train the linear biased SVM.

To use the classifier on an input image histogram, we check the classifier value for each label and keep the classifier which value is positive and the highest to determine which class label is the best. Alternatively, we can keep all the labels for which the classifier is positive and high enough. This can make sense as an image can have multiple classes.

3. Implementation

a. Code

The algorithm is implemented in C++. It uses the CImg library (2) as an internal image and matrix/vector format. The main classes of the program are:

- *TrainingSet*
  This class serves as a container for training features, and their labels. It can compute some statistics on these features. It can split these features according to a binary test, creating tow new *TrainigSet* objects without duplicating the data in memory.
- *FeatureExtractor*
  This class can process images and extract features. The HSL transform as well as the Haar transform is performed using the CImg library. The SIFT transform is performed using the VLFeat library (3).
- *ErcTree*
  This class represents a randomized clustering tree. It can build such a tree from a *TrainingSet* object. It can prune an *ErcTree* object. It can also return a leaf pointer from a test feature. It can also code an *ErcTree* object into an XML string. It can build an *ErcTree* object from an XML document represented using the TinyXML library (4) from an XML string it generated.
- *ErcForest*
  This class stores a set of *ErcTree* objects. It can build histograms from test feature vectors. It can also check if a test feature is attributed an unmixed leaf in one of the trees. It can open and save an *ErcForest* object to an XML file, using the TinyXML library.
- *Classifier*
  This class is used to perform the classification tasks. It is associated with an *ErcForest* object. It can train the SVM classifier from a *TrainingSet* object unsing the VLFeat library. It can open and save to a file an SVM classifier. It can also find and display, from a test image and its features, the points corresponding to features associated with unmixed leaves in the associated *ErcForest* object.

Other secondary classes are defined to generate uniform random numbers, to manage the input files images, to plot key points on images as well as generic data structures such as a binary-tree based sorted associative list, a "nullable" vector container and a timer.

b. Usage

The program is used in the following way.

To train a model and save the forest to a file named “forest.xml” and the SVM classifier to a file named “classifier.bin”, the following command line is used: “ERCF.exe paths.txt” Where “paths.txt” is a text file within which each line is a search path (i. e. a path possibly containing wildcards) targeting the input data. One line of two is for the input images, and the other for the masks. If no masks should be used, the corresponding lines should be left blank.

To test an image “image.jpg” with an existing model for which the forest is stored in the file “forest.xml” and the SVM classifier in the file “classifier.bin”, the following command line should be used: “ERCF.exe forest.xml classifier.bin image.jpg”.

The necessary parameters are then asked to the user through the command prompt.

In the test case, for each class label, the classifier function value is displayed as well as the points corresponding to unmixed leaves.

4. Conclusion

Despite following the most precisely possible the indications provided in the article and optimizing the implementation, no good results could be obtained from the GRAZ-02 set (5) even though it is supposed to provide very good results according to the article. The poor classification results might be explained by an incorrect implementation of the SVM in the implementation, as suspected by the previously mentioned concerns about using a linear SVM as a final classifier. However, the inaccuracy of the unmixed leaves test is completely independent from that issue.

5. Bibliography

1. *Fast Discriminative Visual Codebooks using Randomized Clustering Forests.* **Frank Moosmann, Bill Triggs, Frederic Jurie.**

2. *The CImg Library - C++ Template Image Processing Toolkit.* **Tschumperlé, David.**

3. *VLFeat: An Open and Portable Library of Computer Vision Algorithms.* **A. Vedaldi, B. Fulkerson.** 2008.

4. *TinyXML.* **Lee Thomason, Yves Berquin, Andrew Ellerton.**

5. *GRAZ-02 database.* **Andreas Opelt, Axel Pinz.**