Chapter 11
Re-sampling Methods

Abstract. The subject of this chapter are the re-sampling methods which are a special set of ensemble learning techniques. In the context of image fusion the re-sampling methods create an ensemble of input images $I_k, k \in \{1,2,\ldots,K\}$, from a single base image I^*. In this chapter we shall concentrate on two important re-sampling methods: bootstrapping and boosting.

11.1 Introduction

In the context of image fusion, we use the term ensemble learning to denote the fusion of K input images $I_k, k \in \{1,2,\ldots,K\}$, where the I_k are all derived from the same base image I^*. In the previous chapter we constructed the I_k by applying different signal processing algorithms to I^*. These algorithms include image transformations, normalizations, feature extraction, thresholding and segmentation algorithms. In this chapter we consider a different (re-sampling) approach in which we apply an ensemble of K (fixed) classifiers $C_k, k \in \{1,2,\ldots,K\}$ to I^*. In the re-sampling method the C_k are obtained by training the parameters of a parametric classifier S on re-sampled training data T^*. If D_k denotes the decision map obtained by applying C_k to I^*, then we fuse the $D_k, k \in \{1,2,\ldots,K\}$, together to obtain a fused decision map $\widetilde{D}$. In general, $\widetilde{D}$ is more accurate than any one of the D_k.

The strength of the re-sampling method is that it allows for a systematic generation of a virtually unlimited number of classifiers C_k and the corresponding decision maps D_k in a natural way.

We start with bootstrapping, which is perhaps the most widely used re-sampling method.

11.2 Bootstrapping

Bootstrapping is a basic re-sampling technique in which we generate which an ensemble of K fixed classifiers $C_k, k \in \{1,2,\ldots,K\}$, given a base training data base T^*.

Given T^* we generate K bootstrapped training sets $T_k, k \in \{1,2,\ldots,K\}$, by sampling T^* with replacement (see Ex. 3.10). Then, given a parametric classifier S we create an ensemble of fixed classifiers $C_k, k \in \{1,2,\ldots,K\}$, by separately training S on each training set T_k [1]. If we apply C_k to the test image I^* we obtain a decision map D_k. Finally, we obtain a fused decision map $\widetilde{D}$ by fusing the D_k together:

$$\widetilde{D} = f(D_1, D_2, \ldots, D_K) ,$$

where f is an appropriate fusion operator.

In many cases we use the majority vote rule to fuse the D_k. In this case the combination of bootstrapping and majority vote rule is known as "bagging". In general bagging is useful when the C_k are weak. By this we mean that the performance of the classifier is slightly better than random but is unstable: changes in the training data cause significant changes in the fixed classifier.

We illustrate the concept of bagging on two different applications. The first application illustrates the traditional use of bagging a supervised classifier. The second application illustrates a recent development in which we use bagging in an unsupervised classifier or clustering algorithm.

11.3 Face Recognition with Bagging

We consider face recognition in an unconstrained environment where the appearance of a given face may vary due to changes in lighting, pose and facial expression [5]. In many face recognition applications, only a small number of training samples for each subject are available. These samples are not able to capture all the facial appearance variations. By bootstrapping the training set T^* we generate several subsets $T_k, k \in \{1,2,\ldots,K\}$, of samples from the original training dataset. Each subset T_k is then used to train a classifier C_k.

Given a test face we classify it using the classifiers $C_k, k \in \{1,2,\ldots,K\}$. If D_k is the decision obtained with the classifier C_k, then we obtain a fused decision $\widetilde{D}$ by fusing the D_k together using the majority-vote rule. Blu and Jain [5] found the use of bootstrapping made a substantial improvement in the face recognition accuracy: increasing from 81% to 88.7%.

11.4 Bagged Nearest Neighbor Classifier

The nearest neighbor (NN) classifier (see Sect. 10.9) is simple but very effective classifier which is widely used in many real-world classification systems. It is not, however, a weak classifier, and consequently (conventional) bagging will not improve its performance. However, by creating bootstrapped training sets which are

[1] A parametric classifier is a classifier which has free parameters in it. Optimal values for these parameters are found by training the classifier on a training set. Once the parameter values are specified we have a fixed classifier.

smaller in size than the original training set, we "weaken" the NN classifier which can now be bagged.

Example 11.1. Bagged Nearest Neighbor Classifier [3]. Given a base training set T^* containing M images $y_m, m \in \{1,2,\ldots,M\}$, we create a set of K bootstrapped training sets $T_k, k \in \{1,2,\ldots,K\}$, where each training set T_k contains N images which are selected by randomly sampling T^* with replacement and $N \approx 0.7M$.

Given a test image I we classify it using the NN classifier on each bootstrapped training set $T_k, k \in \{1,2,\ldots,K\}$. Suppose the NN classification obtained with the kth training set is a decision label l:

$$D_k = l \ ,$$

then the bagged NN classification of I is

$$\widetilde{D} = l \qquad \text{if} \qquad \sum_{k=1}^{K} \delta(D_k, l) > \frac{K}{2} \ ,$$

where

$$\delta(a,b) = \begin{cases} 1 \text{ if } a = b \ , \\ 0 \text{ otherwise} \ . \end{cases}$$

11.5 Bagged K-means Clustering

Traditionally, re-sampling methods are used in supervised learning applications in order to improve classification accuracy. Recently re-sampling methods have been used to improve unsupervised clustering algorithms.

In this context we use bagging to generate and aggegrate multiple clusterings and to assess the confidence of cluster assignments for individual observations. The motivation is to reduce the variability in the partitioning results via averaging.

Given an input base image I^* we may segment it into a L label decision image D^* using a K-means clustering algorithm (see Ex. 6.8). Let $G_1, G_2, \ldots, G_K$ denote L cluster centers or cluster gray-levels. Then each pixel gray-level $g_m \equiv I^*(m)$ is assigned to a given cluster:

$$\delta_{ml} = \begin{cases} 1 \text{ if } g_m \text{ is assigned to } G_l \ , \\ 0 \text{ otherwise} \ . \end{cases}$$

We then use the assignment matrix δ_{ml} to create the decision image D^*:

$$D^*(m) = l \qquad \text{if} \qquad \delta_{ml} = 1 \ .$$

The K-means algorithm attempts to find the set of cluster centers $G_l, l \in \{1,2,\ldots,L\}$, such that the total error is a minimum:

$$(G_1, G_2, \ldots, G_L) = \arg\min_{G_k} \sum_{m=1}^{M} \sum_{k=1}^{L} \delta_{mk} |g_m - G_k| ,$$

using the following iterative procedure. In each iteration we calculate the assignment matrix δ_{ml} using the cluster centers G_l calculated in the previous iteration. The cluster centers are then re-calculated using the new assignment matrix. The entire process for T iterations is:

```
for $t = 1 : T$
    for $m = 1 : M$
        $\delta_{mk}^{(t)} = \begin{cases} 1 \texttt{ if } |g_m - G_l^{(t-1)}| = \min_h |g_m - G_h^{(t-1)}| \\ 0 \texttt{ otherwise} \end{cases}$
    end
    for $l = 1 : L$
        $G_l^{(t)} = \sum_{m=1}^{M} \delta_{ml}^{(t)} g_m / \sum_{m=1}^{M} \delta_{ml}^{(t)}$
    end
end
```

Dudoit and Fridlyand [1] show how we may improve the performance of the cluster algorithm by bagging.

Example 11.2. Bagged K-means Clustering Algorithm [1]. The steps in the bagged K-means clustering algorithm are:

1. Transform the base image I^* into a column vector $I^*(m), m \in \{1,2,\ldots,M\}$
2. Form K bootstrapped column vectors $I^{(k)}(m), k \in \{1,2,\ldots,K\}, m \in \{1,2,\ldots,M\}$ by sampling I^* with replacement.
3. Train the K-means cluster algorithm on each bootstrapped column vector $I^{(k)}$, i. e. learn L cluster centers $G_l^{(k)}, l \in \{1,2,\ldots,L\}$.
4. For each k, permute the labels l so that $G_1^{(k)} < G_2^{(k)} < \ldots < G_L^{(k)}$. This ensures the semantic equivalence of the labels l (see Ex. 5.5).
5. For each set of cluster centers $G_l^{(k)}, k \in \{1,2,\ldots,K\}$, classify the pixels in I^*:
$$D^{(k)}(m) = \arg\min_l (|I^*(m) - G_l^{(k)}|) .$$
6. For each pixel $m, m \in \{1,2,\ldots,M\}$, form a bagged decision $\widetilde{D}(m)$ using a majority-vote rule:

$$\widetilde{D}(m) = l \text{ if } \textstyle\sum_{k=1}^{K} \delta(D^{(k)}(m), l)) \geq \frac{K}{2} ,$$

where

$$\delta(a,b) = \begin{cases} 1 \text{ if } a = b , \\ 0 \text{ otherwise} . \end{cases}$$

11.6 Boosting

Boosting is closely related to bagging except the training sets $T_k, k \in \{1,2,\ldots,K\}$, are no longer independent, instead they are created sequentially. If C_k denotes the classifier which is trained on T_k, then T_{k+1} is created in accordance with the classification accuracy obtained with C_k.

Adaboosting is probably the most successful boosting algorithm. It creates an ensemble of fixed classifiers $C_k, k \in \{1,2,\ldots,K\}$, as follows. Let $T^*(i), i \in \{1,2,\ldots,N\}$, denote the individual samples in T^*. At the kth iteration, each sample $T^*(i)$ is assigned a weight $w_k(i)$. Together, the training samples $T^*(i)$ and the weights $w_k(i)$ constitute a *weighted* training set T_k. At the kth iteration we create a fixed classifier C_k by training a parametric classifier S on the weighted training set T_k. For the next iteration $(k+1)$ we update $w_k(i), i \in \{1,2,\ldots,N\}$, by increasing the weight of $w_k(i)$ if C_k incorrectly classifies $T^*(i)$ and decreasing the weight of $w_k(i)$ if C_k correctly classifies $T^*(i)$.

The following is the pseudo-code for the two-class adaboost algorithm:

Example 11.3. Adaboost Algorithm

```
Initialize $w_0(i) = 1/N, i \in \{1,2,\ldots,N\}$
For $k = 0 : K$
   Generate $C_k$ by training $S$ on a weighted
       training set $T_k$
   Find samples $T^*(i)$ which are misclassified by $C_k$:
       $\delta_k(i) = \begin{cases} 1 \text{ if } C_k \text{ misclassifies } T^*(i) \\ 0 \text{ otherwise.} \end{cases}$
   Calculate $E_K$ and $\beta_k$:
       $E_k = \sum_{i=1}^{N} \delta_k(i) w_k(i)$, $\beta_k = E_k/(1-E_k)$
   Update the weight vector:
      $w_{k+1}(i) = w_k(i)(\delta_k(i) + (1-\delta_k(i))\beta_k)$
   Normalize $w_{k+1}$: $w_{k+1}(i) = w_{k+1}(i)/\sum_{i=1}^{N} w_{k+1}(i)$
end
```

Given the classifiers $C_k, k \in \{1,2,\ldots,K\}$, we classify a test sample as follows: Let D_k be the classification of the test sample obtained with the classifier C_k. Then we fuse the D_k using a weighted majority-vote rule, where the weight given to each classifier C_k is proportional to its accuracy on the weighted training set used to train C_k. The final classification of the test sample is $\widetilde{D}$:

$$\widetilde{D} = l \qquad \text{if} \qquad \sum_{k=1}^{K} \log(1/\beta_k)\delta(D_k, l) \geq \frac{1}{2}\sum_{k=1}^{K} \log(1/\beta_k) \; .$$

where

$$\delta(a,b) = \begin{cases} 1 \text{ if } a = b \; , \\ 0 \text{ otherwise} \; . \end{cases}$$

Adaboost is widely used in classification applications. Since its introduction in 1997 [2] it has established itself as a high-performance general purpose classifier. However, in noisy situations, such as those with label noise, the adaboost algorithm may perform poorly. In this case we may use the ave2boost algorithm. This is a modified version of adaboost specifically designed for noisy situations. In ave2boost we regulate how the weight vector w_k is updated and we modify the weighted-vote rule. For the sake of completeness we give the pseudo-code for ave2boost:

Example 11.4. Ave2boost Algorithm [8].

```
Initialize $w_0(i) = 1/N, i \in \{1,2,\ldots,N\}$
For $k = 0 : K$
   Generate $C_k$ by training $S$ on weighted
       training set $T_k$
   Find samples $T^*(i)$ which are misclassified by $C_k$:
       $\delta_k(i) = \begin{cases} 1 \text{ if } C_k \text{ misclassifies } T^*(i) \\ 0 \text{ otherwise.} \end{cases}$
   Calculate $E_k, \beta_k$ and $\gamma_k$:
       $E_k = \sum_{i=1}^{N} \delta_k(i) w_k(i)$, $\beta_k = E_k/(1-E_k)$, $\gamma_k = \frac{2(1-E_k)k+1}{2E_k k+1}$
   Update the weight vector:
      $w_{k+1}(i) = w_k(i)(\delta_k(i) + (1-\delta_k(i))\beta_k)$
   Calculate regularization factors:
       $v_{k+1}(i) = w_{k+1}(i) / \sum_{i=1}^{N} w_{k+1}(i)$
   Normalize $w_{k+1}$: $w_{k+1}(i) = (k w_{k+1}(i) + v_{k+1}(i))/(k+1)$
end
Classify test sample with each $C_k$:
    $D_k = l$ if $C_k$ identifies test sample as belonging
          to the $l$th class
Final classification of test sample is:
    $\widetilde{D} = l$ if $\sum_{k=1}^{K} \log(1/(\beta_k\gamma_k))\delta(D_k,l) \geq \frac{1}{2}\sum_{k=1}^{K} \log(1/(\beta_k\gamma_k))$
    where $\delta(D_k,l) = 1$ if $D_k = l$, otherwise $\delta(D_k,l) = 0$.
```

11.7 Viola-Jones Algorithm

The conventional adaboost algorithm uses only one parametric classifier S. Viola and Jones (VJ) [10] remove this restriction and use instead M parametric classifiers $S_m, m \in \{1,2,\ldots,M\}$. In each iteration $k, k \in \{1,2,\ldots,K\}$, we select one fixed classifier C_k as follows:

1. Generate M fixed classifiers $c_{k,m}, m \in \{1,2,\ldots,M\}$, by training $S_m, m \in \{1,2,\ldots, M\}$, on the weighted training set T_k.
2. Select the fixed classifier with the smallest weighted error:

$$C_k = c_{k,m} \qquad \text{if} \qquad m = \arg\min_m \sum_{i=1}^{N} \delta_{k,m}(i) w_k(i) \ ,$$

where

$$\delta_{k,m}(i) = \begin{cases} 1 \text{ if } c_{k,m} \text{ misclassifies } T^*(i) \ , \\ 0 \text{ otherwise} \ . \end{cases}$$

11.8 Boosted Object Detection

The VJ algorithm was originally developed for real-time object detection. In this application the number of parametric classifiers $S_m, m \in \{1,2,\ldots,M\}$, is very large. For example, in typical application concerning face detection in an input image, $M \approx 180000$ and special attention must therefore be paid to generating and efficiently training the S_m.

The boosted object detection algorithm works as follows. We create N training samples $T^*(i), i \in \{1,2,\ldots,N\}$, by dividing several training images into overlapping blocks of size $L \times L$. Each block B_i constitutes a training sample $T^*(i)$, where

$$y(i) = \begin{cases} 1 \text{ if } B_i \text{ contains an object} \ , \\ 0 \text{ otherwise} \ . \end{cases}$$

In each iteration k of the algorithm, we probe the blocks $B_i, i \in \{1,2,\ldots,N\}$, with a $L \times L$ ternary mask ϕ_m (Fig. 11.1) and compare the result to a threshold θ:

$$d_{k,m}(i) = \begin{cases} 1 \ \text{ if } p \sum_{(x,y)} \phi_m(x,y) B(x,y) \geq p\theta \ , \\ 0 \text{ otherwise} \ . \end{cases}$$

where p is a polarity variable ($p = -1$ or $+1$) which determines if the $(\sum_{(x,y)} \phi_{k,m}(x,y) \times B(x,y))$ should be greater, or smaller, than θ. We fix the polarity variable p and the threshold θ by minimizing the weighted error:

$$e_{k,m} = \sum_{i=1}^{N} w_k(i) |d_{k,m}(i) - y(i)| \ .$$

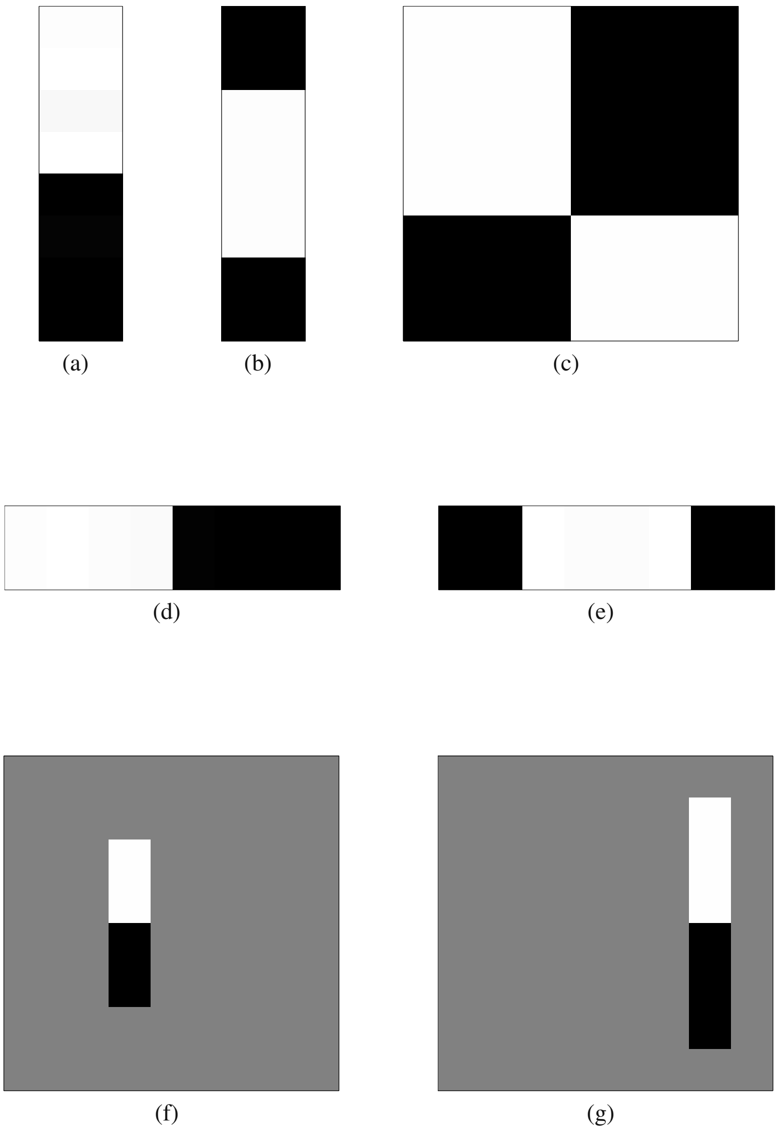

Fig. 11.1 **(a)**-**(e)** Show five basic masks which contain only -1 and $+1$ values. **(f)** Shows a ternary mask ϕ_m obtained by placing the first basic mask in a block of zeros. **(g)** Show a ternary mask ϕ_l obtained by scaling the first basic mask and placing it in a block of zeros. In the figure, white, black, and gray stand, respectively, for values of $+1$, -1 and 0.

The following example is the pseudo-code for the Viola-Jones algorithm.

Example 11.5. Viola-Jones Object Detection Algorithm [6, 10].

1. Extract the features $f_m(i) = \sum_{(x,y)} \phi_m(x,y)B_i(x,y), m \in \{1,2,\ldots,M\}$, for each training block $B(i), i \in \{1,2,\ldots,N\}$.
2. Initialize the weight $w_1(i)$ for each training block i:

   $$w_1(i) = \begin{cases} 1/(2N_{true}) & \text{if } B(i) \text{ contains an object }, \\ 1/(2N_{false}) & \text{if } B(i) \text{ does not contain an object }, \end{cases}$$

   where N_{true} is the number of training blocks which contain an object and N_{false} is the number of training blocks which do not contain an object.
3. For $k = 1 : K$ perform the following:
   a. Normalize the weights $w_k(i)$:

   $$w_{k+1}(i) = w_k(i) / \sum_{i=1}^{N} w_{k+1}(i) \ .$$

   b. For each features f_m create a fixed classifier $c_{k,m}$ by training S_m on the weighted error:

   $$e_{k,m} = \sum_{i=1}^{N} w_k(i) |d_{k,m}(i) - y(i)| \ ,$$

   c. Feature selection. Choose the fixed classifier C_k with the lowest weighted error E_k:

   $$\left.\begin{array}{l} C_k = c_{k,m} \\ E_k = e_{k,m} \end{array}\right\} \qquad \text{if} \qquad m = \arg\min_n (e_{k,n}) \ ,$$

   d. Update the weights $w_k(i)$:

   $$w_{k+1}(i) = w_k(i)\beta_k^{(1-\Delta_k(i))} \ .$$

   where $\beta_k = E_k/(1-E_k)$ and

   $$\Delta_k(i) = \begin{cases} 1 \text{ if } C_k \text{ incorrectly classifies } B(i) \ , \\ 0 \text{ otherwise }. \end{cases}$$

4. For a test block B we separately classify it using the fixed classifiers $C_k, k \in \{1,2,\ldots,K\}$. If $D_k, k \in \{1,2,\ldots,K\}$, are the corresponding decisions, then we combine the D_k to obtain a fused decision $\widetilde{D}$:

   $$\widetilde{D} = \begin{cases} 1 \text{ if } \sum_{k=1}^{K} \alpha_k D_k \geq \sum_{k=1}^{K} \alpha_k/2 \ , \\ 0 \text{ otherwise }, \end{cases}$$

   where $\alpha_k = \log(1/\beta_k)$.

For real-time processing, the parametric classifiers S_m are constructed by scaling a basic binary mask (containing only the values -1 and $+1$) and placing it anywhere in a $L \times L$ background of zeros. This method of construction facilitates real-time processing by using method of integral images which we now explain in the following example.

Example 11.6. Integral Image. Given an $L \times L$ image block B with pixel gray-levels $B(x,y), x,y \in \{1,2,\ldots,L\}$, we pre-compute the following sum:

$$I_\Sigma(x,y) = \sum_{u=1}^{x} \sum_{v=1}^{y} B(u,v) \; .$$

Then the feature value $f(B) = \sum_{(x,y)} B(x,y)\phi(x,y)$, corresponding to the mask ϕ shown in Fig. 11.2 may be efficiently calculated in four operations:

$$f(B) = I_\Sigma(x_2,y_2) - I_\Sigma(x_1,y_1) - (I_\Sigma(x_4,y_4) - I_\Sigma(x_3,y_3)) \; .$$

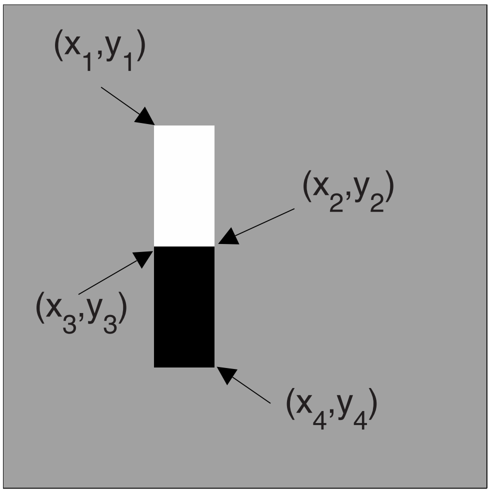

Fig. 11.2 Shows a ternary mask ϕ obtained by placing a binary mask in a block of zeros. In the figure white, black and gray stand, respectively, for values of $+1$, -1 and 0. The top left-hand corner and bottom right-hand corner of the $+1$ block are (x_1,y_1) and (x_2,y_2). Similarly, the top left-hand corner and bottom right-hand corner of the -1 block (x_3,y_3) and (x_4,y_4).

11.9 Software

STPRTOOL. A statistical pattern recognition toolbox. Authors: Vojtech Franc and Vaclav Hlovac.

11.10 Further Reading

A modern book devoted to ensemble methods is [7]. Ref. [4] is a detailed report on implementation of the Viola-Jones algorithm for real-time object detection. For a recent extension of the Viola-Jones algorithm for object detection see [9]

References

1. Dudoit, S., Fridlyand, J.: Bagging to improve the accuracy of a clustering procedure. Bioinformatics 19, 1090–1099 (2003)
2. Freund, Y., Schapire, R.E.: Decision-theoretic generalization of on-line learning and an application to boosting. J. Comp. System Sci. 55, 119–139 (1997)
3. Hall, P., Samworth, R.J.: Properties of bagged nearest neighbor classifiers. J. R. Statist. Soc. 67B, 363–379 (2005)
4. Jensen, O.H.: Implementing the Viola-Jones face detection algorithm. MSc thesis, Technical University of Denmark (2008)
5. Lu, X., Jain, A.K.: Resampling for Face Recognition. In: Kittler, J., Nixon, M.S. (eds.) AVBPA 2003. LNCS, vol. 2688. Springer, Heidelberg (2003)
6. Martins, R., Pina, P., Margques, J.S., Silveira, M.: Crater detection by a boosting approach. IEEE Geosci. Remote Sensing Lett. 6, 127–131 (2009)
7. Okum, O., Valentini, G.: Supervised and Unsupervised Ensemble Methods and Their Applications. Springer, Heidelberg (2008)
8. Oza, N.C.: Ave2boost: boosting for noisy data. In: Roli, F., Kittler, J., Windeatt, T. (eds.) MCS 2004. LNCS, vol. 3077, pp. 31–40. Springer, Heidelberg (2004)
9. Pavani, S.-K., Delgado, D., Frangi, A.F.: Haar-like features with optimally weighted rectangles for rapid object detection. Patt. Recogn. (in print, 2009)
10. Viola, P., Jones, M.: Robust real-time face detection. Int J. Comp. Vis. 57, 137–154 (2004)